COSC 2P95 – Lab Exercise 4 – Section 01 – IO

Lab Exercises are uploaded via the Sakai page.
Since you need to submit both images and your source file, package it all up into a .zip to submit.

This week, we'll be extending on last week's discretization problem.
If you'll recall, one of the formulas we used was this one:
https://www.wolframalpha.com/input/?i=plot+%28%281%2F2*sin%28x%29%29%2B%281%2F2*cos%28y%29%29%29,+x%3D-4..6,+y%3D-12..5

We're going to modify it slightly by adding a new parameter, z:
https://www.wolframalpha.com/input/?i=plot+%28cos+z*%281%2F2*sin%28x%29%29%2B%281%2F2*cos%28y%29%29%29,+x%3D-4..6,+y%3D-12..5,+z%3D0

Thus far, since z is set to 0, and cos(0) is 1, there isn't yet any change to the actual produced results.
Rather than outputting to either a 'bitmap' or a dump of floating-point values, you're instead going to be generating a PGM file.

Portable Graymap File Format

The Portable Pixmap (PPM), Portable Graymap (PGM), and Portable Bitmap (PBM) file formats are all part of the same family of naive (PNM) image formats.
They all revolve around a minimal ASCII-based header, followed by direct pixel information.
We'll be using the PGM format, which means we can only produce greyscale images.

Note that, in addition to being very easily googled, this format is also explained in your `man` pages!
Just type: `man pgm`

There are two main versions of PGMs: raw and plain. We'll be using the plain version, which means we'll be generating images that rely solely on ASCII encoding.
The man page has everything you need (you can also access man pages from sandcastle, so you can easily do this from home). To be clear, we're using the P2 *magic number*, and the maximum value we'll allow is 255.

Single Frame

You'll be writing two programs (you can actually write them as a single program if you wish; it'll just be explained as two). Let's look at the simpler one first.
Based on the new version of the formula above (which multiplies the entire expression by cos(0), i.e. 1), create a .pgm file, called `file0001.pgm`.

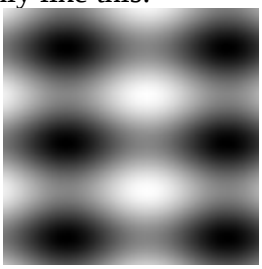
Tips:

- The calculated values from that formula are all in the range of -1..+1
- If you scale each value to being from 0..2, multiply by 127.5, and then truncate the decimal component, you'll get a legal value, appropriately scaled for the greyscale pixel
- The filename is just to make the next part easier
  - You can open a file by using a C-style string (i.e. a char array) as the filename
  - Since it's a char array, you can easily change individual characters
    - Don't forget that 0 and '0' aren't the same thing, but '0'+1 yields '1'

Still ask the user for the precision (the number of graduations). Now, that single number will also indicate the height and width of the generated image.

You may not be able to view the generated image on your own computer. The picture viewer in Linux shouldn't have a problem with it. But failing that:
`convert file0001.pgm file0001.png`
will convert it to a .png format that you can download from sandcastle and easily view at home, if desired.

The frame from this section should look roughly like this:



Animation

(Again, this may be part of the above program, but will be treated as a separate one for explanation's sake)
Suppose you were to generate another frame where z had a value of 0.1, and then another where it was 0.2, and then 0.3, etc.
Eventually, you'd have several images that were very similar, but slightly different from one from to the immediate successor. Hypothetically, if you had a batch of such images, you could combine them into an animation!

- Prompt the user for the number of graduations, as before
- Also ask for the number of frames to generate
- A single pass through that number of frames will take z from 0 to 6.28318530718
  - That means you'll need a loop of #frames iterations, and calculate the scaled values for z as before (it's just easier math this time)
- #frames images will be created. The filenames will be `animXXXX.pgm`, where XXXX is the frame #
  - Again, don't forget that you can just set “animXXXX.pgm” as the char array, and then manually change those four positions allocated for digits.

Once you have a folder full of `anim0000.pgm`, `anim0001.pgm`, etc. you can very easily convert it into an animated gif:

```
convert -delay 1 anim*.pgm animation.gif
```

Unfortunately, the .gif may not display properly in the image-viewer in Linux. If not, just drag&drop it onto a browser to view it. You should see an infinitely-repeating pattern. Assuming you chose enough frames (e.g. 60), it should be pretty smooth.

Requirements for Submission:

- All pixel data must be stored in a tabular format
  - That is, each column must have the same width. In this case, 4 characters, including the space
- Don't worry about adhering to PGM's 80-characters-per-line restriction. It isn't enforced
- Because this is the sort of visualization that can be very helpful for things like research papers, etc., you're encouraged to try visualizing the other functions as well (just not for submission)

To submit: you need to include the following:

- Your source file
- A single .pgm file of one frame of the function
- A single animated .gif

As stated at the beginning, pack everything into a .zip before submitting.
Note that you can just delete the .pgm files you used to construct the animation.