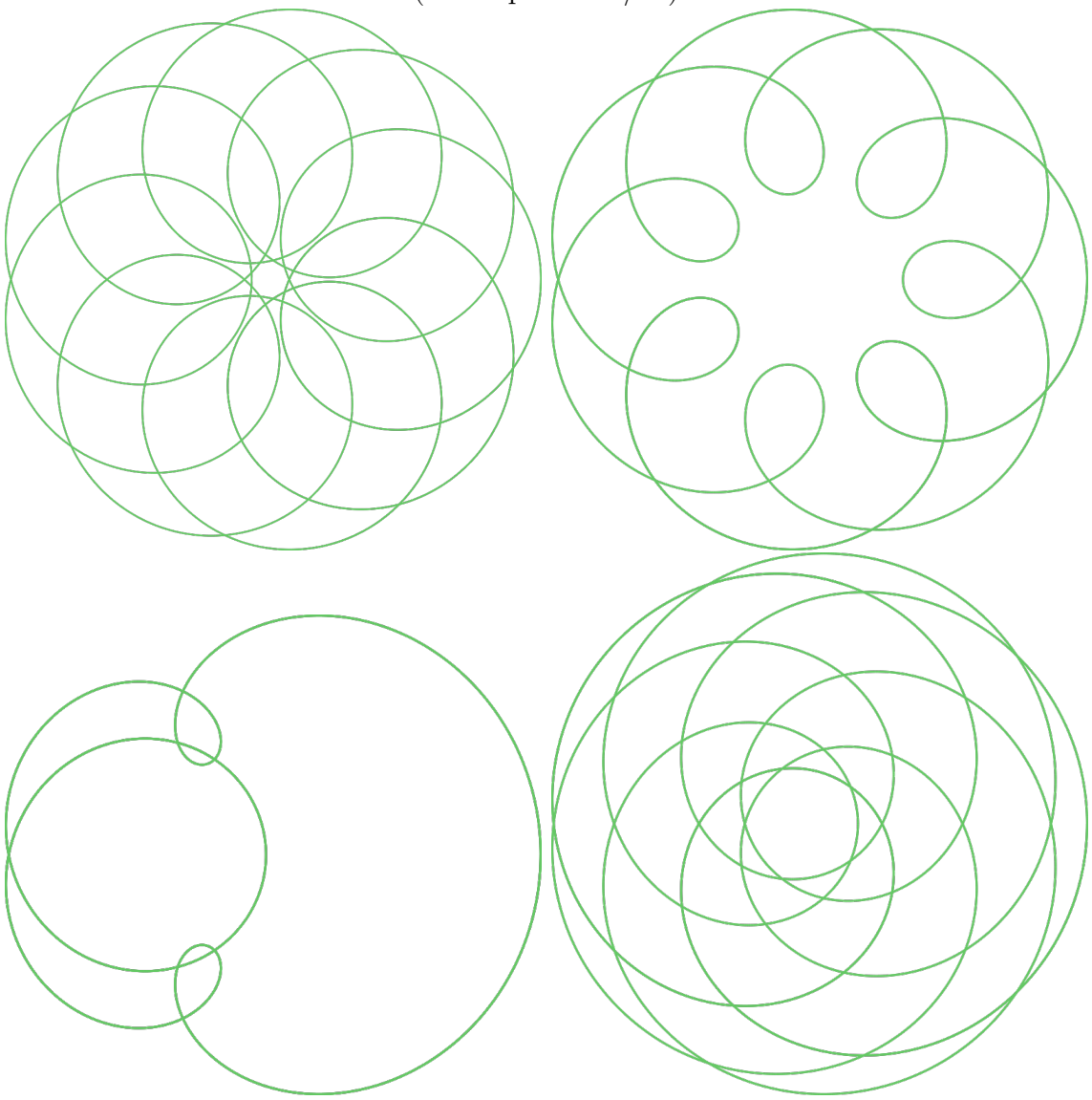
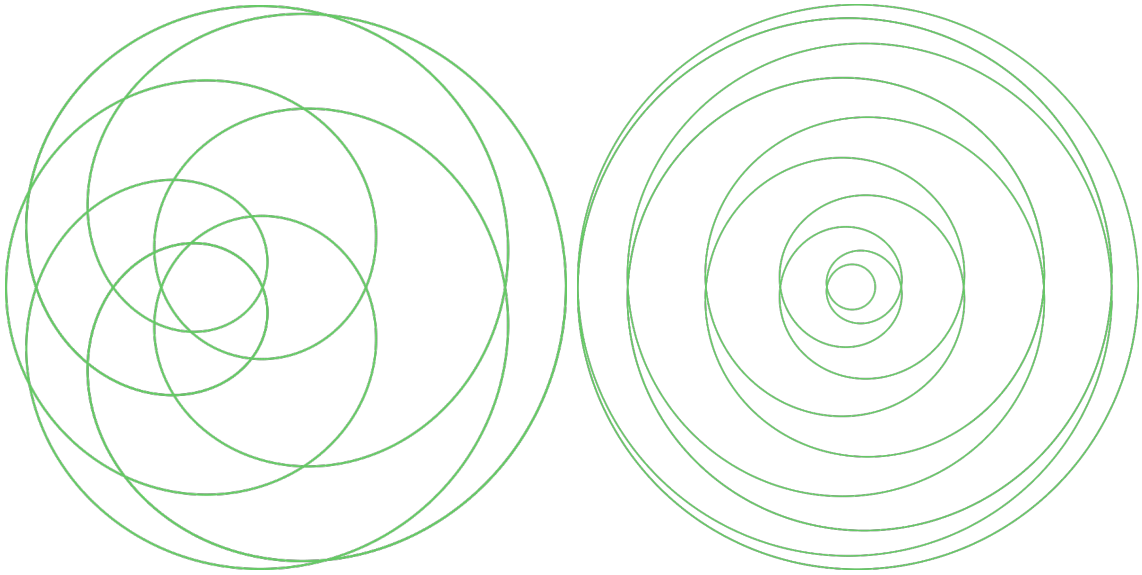
Project 1: Drawing with epicycles

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

(Last updated 2/20)





1 Introduction

Procedural art is the process of writing programs that generate artistic artifacts, such as drawings, music, or text. There's a huge range of options for procedural art, ranging from largely human-guided pieces, where the program acts like a paint-brush, to randomized art where the program is in charge of most of the artistic design.

In this project you will be making a simple procedural art generator that creates images of swirly drawings similar to what can be made by a Spirograph drawing system. By either hand-choosing input parameters to this program, or randomly choosing inputs from a fixed range, a variety of interesting (often quite circular) images can be drawn.

The educational goal of this project is to give you more in-depth and self-guided experience developing a larger program in Python. In this case we will be using a primarily object-oriented design paradigm. While we will be specifying class names, and a few essential method names, we will also be giving you a fair amount of control over how to implement these methods. As such, plan on spending some time working through your program design before you begin implementation. Likewise, there will be less testing support for this project than there is in the weekly labs. You should plan on spending some time deciding exactly how to test your program.

The approach we are going to use to draw the swirly drawings has two core parts: the Scalable Vector Graphics (SVG) image format and the idea of "Drawing with Epicycles".

2 SVG graphics

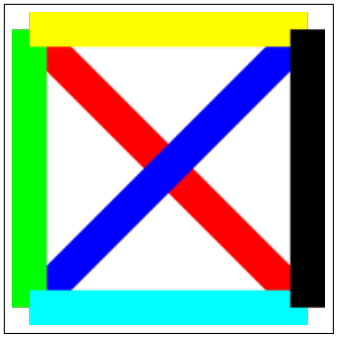
Scalable Vector Graphics (SVG) is an image format that describes an image as a series of drawn shapes, rather than specifying a color value for each pixel. By describing an image in

this way the image can be re-rendered for any display medium. For example, if a very high-resolution version of a SVG image is needed, the image can simply be rendered at a higher resolution. This property is why SVG images are considered "Scalable" unlike traditional rasterized images the image will look about the same on any display and can be freely scaled.

While many software exists that can render SVG software, the most common such software is web browsers. This may seem like a strange software to serve as the SVG viewer of choice, but the SVG format did grow out of various web standards. Likely if you double-click or otherwise try to open an SVG file on your computer your default web browser will open the file. I recommend testing this early on in this homework as being able to see your SVG file will be useful in testing your code.

One additional property that makes SVG files interesting for small-scale procedural graphical applications like this one, is that SVG is a text-based format (Specifically, SVG is a type of XML document, for those who have experience working with XML documents)

Below is an example SVG file, and a rendering of it's image (SVG cannot be embedded into a pdf, you can view the original SVG through canvas)



```
<svg xmlns='http://www.w3.org/2000/svg' viewBox='0 0 10 10'>
<!-- This is a comment. -->
<line x1='1' y1='1' x2='9' y2='9' style='stroke:rgb(255,0,0);' />
<line x1='1' y1='9' x2='9' y2='1' style='stroke:rgb(0,0,255);' />
<line x1='1' y1='1' x2='1' y2='9' style='stroke:rgb(0,255,0);' />
<line x1='1' y1='1' x2='9' y2='1' style='stroke:rgb(255,255,0);' />
<line x1='9' y1='9' x2='1' y2='9' style='stroke:rgb(0,255,255);' />
<line x1='9' y1='9' x2='9' y2='1' style='stroke:rgb(0,0,0);' />
</svg>
```

SVG is a full featured image specification language with a dizzying number of features. Information about the full specifications can be found online. The example above, however, shows off all the features that we will need.

The first line declares that this file is an svg file and establishes the "view box" - the dimensions that will be used in the image. In this case the view box is "0 0 10 10" indicating a minimum x value of 0, a minimum y value of 0, a width of 10 and a height of 10. The point (0,0) is the top-left of the image, with x increasing as you go right, and y increasing as you go down.

Below the first line are several comment lines. Comments are started with <!--, can

extend multiple lines, and end with −−>.

Then there is a range of line commands. Line commands are listed with a series of paramaters, in particular the beginning and ending positions of the line (x1, y1, x2, y2) and the style of the line. Many style options are available, but we will focus on the stroke color (the color of the line) represented as an RGB (red, green, blue) value (indicating how red, green, and blue the color is in the range 0 - 255) The Line commands start with a < and end with / >. Finally, the last line </svg> indicates the end of file and is required.

You will be required to build a basic SVG python class. An example program using the SVG class is provided below. This produces the example above. As this needs to write files using python's file IO (which we have not covered in class) a basic starter for this is provided. The starter will demonstrate how to open a file in python, and how to write to that file using a print statement.

```
svg = SVG("example.svg", 0, 0, 10, 10)

svg.comment("This is a comment.")
svg.setColor(255, 0, 0)
svg.drawLine(1,1,9,9)
svg.setColor(0, 0, 255)
svg.drawLine(1,9,9,1)
svg.setColor(0,255,0)
svg.drawLine(1,1,1,9)
svg.setColor(255,255,0)
svg.drawLine(1,1,9,1)
svg.setColor(0,255,255)
svg.drawLine(9,9,1,9)
svg.setColor(0,0,0)
svg.drawLine(9,9,9,1)
svg.closeFile()
```

3 Drawing with Epicycles

The word epicycle refers to a small circle, whose center moves around a larger circle. This idea was originally used when trying to describe the motion of planets across the night sky. It turns out that it is quite hard to model the other planets as orbiting around the earth (Seeing as how they don't) To explain the behavior of some planets, therefore, it was modeled that the planets orbit in a small cycle (the epicycle) whose center point orbits around the earth. While we no longer need this idea in the study of astronomy, it remains useful for some forms of generative art.

For our program we will take this idea to it's programmatic extreme. Assume we are given a series of k cycles, defined by a rotation speed and a radius. The first cycle describes

a circle moving around the origin. The point orbiting around the first cycle serves as the center for the second cycle. Likewise the point orbiting around the center of the second cycle (and indirectly around the origin) serves as the center of the third cycle, and so-forth. By choosing series of radius and rotation speeds, we can create many different drawings. In fact it can be shown that, with enough cycles, any image can be approximated. (We will largely concern ourselves with systems with only a few cycles, the more cycles you use the more likely the result is to look silly.)

Explained again, but more mathematically, we are given a series of radii $R_1, R_2, \ldots R_n$ and a series of speeds $w_1, w_2, \ldots w_n$, and a time t. We can define first a point x_1, y_1 as

$$x_1 = 0 + R_1 \cos(tw_1)$$

$$y_1 = 0 + R_1 \sin(tw_1)$$

We then define x_2, y_2 as:

$$x_2 = x_1 + R_2 \cos(tw_2)$$

$$y_2 = y_1 + R_2 \sin(tw_2)$$

and so-forth. Ultimately we are concerned with the location of point x_n, y_n, and wish to plot the curve it makes as we vary t.

You will create a class Epicycle which will take a constructor parameter describing the list of speeds and radiuses to simulate. By calling a method simulate (specifying the range of time values t to use, and what step size to take) we create the series of points for x_n, y_n computed at various times. The Epicycle object then has a method to get this list of points, as well as minimum and maximum values to help in drawing the shape.

Connecting these two parts will be one final piece of code that will need to use SVG and Epicycle methods, as well as converting from the list of points returned by Epicycle, to the series of lines to be drawn by SVG.

4 Implementation

You will be required to write three things: The SVG class, the Epicycle class, and a driver script.

4.1 SVG Class

The SVG class must have the following methods. As usual the methods must be named as listed, but the parameters (except for self) can be named whatever you want.

- __init__ (self , fileName, minX, minY, width, height) Constructor Takes a fileName (for the SVG file) the minimum X value and Y value that you want visible, and the width and height of the area you want visible Opens the file for output and outputs the required initial SVG file. This is partially provided for you on canvas, however the provided version only outputs some of the required information.

- closeFile (self) This method should write the final line of the svg and then close the file. Partially provided.
- setColor(self , r , g, b) This method should update the current draw color for the next lines. The default color should be black (0, 0, 0)
- comment(self, contents) This method should output a comment. This will be useful for debugging.
- drawLine(self, x1, y1, x2, y2) This method should output an SVG command for a line starting at point (x1, y1) and ending at point (x2, y2)

4.2 Epicycle Class

- __init__ (self , speed, radius) Constructor The speed parameter should be a python list containing numbers to specify the speed multiplier for the cycles. The radius parameter should be a python list containing numbers to specify the radii of each cycle. You can assume these are the same length.
- simulate(self , maxTime, dt) This method sohuld perform the actual simulation of the epicycle system computing a list of tuples (x,y) representing the point in (x,y) space. This method does not need to return this list of points, only compute it. The generated list should contain one point at time 0, the second at time dt, the next at $dt + 1$ and so-forth until $maxTime$ is reached. For example, simulate(10, 0.1) should produce 100 points, for $t = 0, 0.1, 0.2, \ldots, 10$. Note, Python's "range" function cannot be used with a non-integer step sizes.

  The dt parameter controls how "smooth" of a curve we get. If you make it too small the shape will visibly be made of short line segments, instead of appearing like a curve. If dt is too small the simulation will take a long time and the output SVG will be very big.

  The maxTime parameter controls how much of the shape to simulate. If too small of a value is chosen the shape will not be complete, if too large of a value is chosen the shape drawn curve will overlap itself (wasting time and file size)
- getPoints(self) return the most recently generated list of points. If simulate has not been called, this should return None
- getMinX(self) return the minimum x value in the most recently generated list of points. If simulate has not been called, this should return None.
- getMinY(self) return the minimum Y value in the most recently generated list of points. If simulate has not been called, this should return None.
- getMaxX(self) return the maximum x value in the most recently generated list of points. If simulate has not been called, this should return None.

- getMaxY(self) return the maximum y value in the most recently generated list of points. If simulate has not been called, this should return None.
- getWidth(self) return the width (maxX - minX) of the most recently generated list of points. If simulate has not been called, this should return None.
- getHeight(self) return the height (maxY - minY) of the most recently generated list of points. If simulate has not been called, this should return None.

4.3 Final parts

You should additionally write a python function that will take an instance of Epicycle and a filename, and create the SVG file (using SVG and Epicycle methods).

- render(epicycle , filename) Create an SVG and save the provided epicycle object's most simulation as an image. filename specifies the name of the svg file to save to.

You will likely also want to write a script of some sort to initialize the epicycle object so you can run it and get random art!

My script will be provided on canvas, but you are free to write your own.

5 Deliverables

For this project we will want a single python file. This should contain both classes (Epicycle, and SVG) the rendering method (render) and any further script you have written to generate art with. We ask that you name your file `<YourName>_Project1.py` before uploading it to Canvas. This will help us resolve any issues with identifying files during automated testing that might arise. Your work must be submitted on Canvas by Friday, March 8, 2019 at 6:00pm.