National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2024/2025

Mission 5
Curve Manipulation

Release date: 17 February 2025
Due: 06 March 2025, 23:59

Required Files

- mission05-template.py
- hi_graph.py

Background:

Grandmaster Ben is pleased that the disciples are putting in effort to conjure magic. He prepares them for their next lesson and cautions them that this is the only beginning. They have to be mentally prepared to deal with the increasing difficulty as the training of curves proceeds.

Grandmaster Ben begins the second lesson by presenting some recursive problems as warm-up exercises. He then demonstrates some difficult curve manipulation techniques. He instructs his disciplines to follow him and practice as he demonstrates.

After some practice, the disciples begin to get the hang of curve manipulation. Grandmaster Ben is pleased and encourages them further:

"Yes, now I can feel that you have some control over your curve drawing. Now try to imagine it floating around in that empty space of your mind, twisting and warping, forming a connected image..."

Information:

For your convenience, the template file **`mission05-template.py`** contains a line to load the Python source file **`hi_graph.py`**. Use the template file to answer the questions.

This mission has **three** tasks.

Task 1: (4 marks)

It is useful to have operations which combine curves into new ones. We let Binary-Transform be the type of binary operations on curves,

$$\text{Binary-Transform} : (\text{Curve}, \text{Curve}) \to \text{Curve}.$$

The function `connect_rigidly` is a simple Binary-Transform.

```
def connect_rigidly(curve1, curve2):
    def connected_curve(t):
        if (t < 0.5):
            return curve1(2*t)
        else:
            return curve2(2*t - 1)
    return connected_curve
```

Evaluation of `connect_rigidly(curve1, curve2)` returns a curve consisting of `curve1` followed by `curve2`; the starting point of the curve returned by `connect_rigidly(curve1, curve2)` is the same as that of `curve1` and the end point is the same as that of `curve2` (`curve1` and `curve2` can be disconnected).

There is another, possibly more natural, way of connecting curves. The curve returned by `connect_ends(curve1, curve2)` consists of a copy of `curve1` followed by a copy of `curve2` after it has been rigidly translated so its starting point coincides with the end point of `curve1`. The end product is a continuous curve.

It is important to note that, in order to make the starting point of `curve2` coincides with the end point of `curve1`, you can only shift or scale `curve2`, which means that you *cannot* rotate `curve2`.

Write a definition of the Binary-Transform `connect_ends`. It is **recommended** that you use `connect_rigidly` in your `connect_ends` function.

Hint

You may want to use the following functions provided in `hi_graph.py` in your solution:

- `translate` returns a Curve-Transform which rigidly moves a curve given distances along the x and y axes.
- `scale_xy` returns a Curve-Transform which stretches a curve along the x and y coordinates by given scale factors.

Note: You are **NOT** required to use *both* these methods. However, you may be penalised if you did not use these functions where obviously appropriate, i.e. if you manually scaled and translated in a similar fashion to `translate` and `scale_xy` when you could have easily used the functions. This applies for other functions like `x_of` as well.

Task 2: (4 marks)

To show off the power of our drawing language, let's use it to explore fractal curves. Fractals have striking mathematical properties[1]. Fractals have received a lot of attention over the past few years, partly because they tend to arise in the theory of nonlinear differential equations, but also because they are pretty, and their finite approximations can be easily generated with recursive computer programs.

For example, Bill Gosper[2] discovered that the infinite repetition of a very simple process creates a rather beautiful image, now called the *Gosper C Curve*. At each step of this process there is an approximation to the Gosper curve. The next approximation is obtained by adjoining two scaled copies of the current approximation, each rotated by 45 degrees.

Figure 1 shows the first few approximations to the Gosper curve, where we stop after a certain number of levels: a level-0 curve is simply a straight line; a level-1 curve consists of two level-0 curves; a level 2 curve consists of two level-1 curves, and so on. The figure also illustrates a recursive strategy for making the next level of approximation: a level-n curve is made from two level-$(n-1)$ curves, each scaled to be $\sqrt{2}/2$ times the length of the original curve. One of the component curves is rotated by $\pi/4$ (45 degrees) and the other is rotated by $-\pi/4$. After each piece is scaled and rotated, it must be translated so that the ending point of the first piece is continuous with the starting point of the second piece.

We assume that the approximation we are given to improve (named `curve` in the function) is in standard position. By doing some geometry, you can figure out that the second curve, after being scaled and rotated, must be translated right by .5 and up by .5, so its beginning coincides with the end point of the rotated, scaled first curve.

This leads to the Curve-Transform `gosperize`:

```
def gosperize(curve):
    scaled_curve = scale(sqrt(2)/2)(curve)
    left_curve = rotate(pi/4)(scaled_curve)
    right_curve = translate(0.5,0.5)(rotate(-pi/4)(scaled_curve))
    return connect_rigidly(left_curve, right_curve)
```

Now we can generate approximations at any level to the Gosper curve by repeatedly gosperizing the unit line,

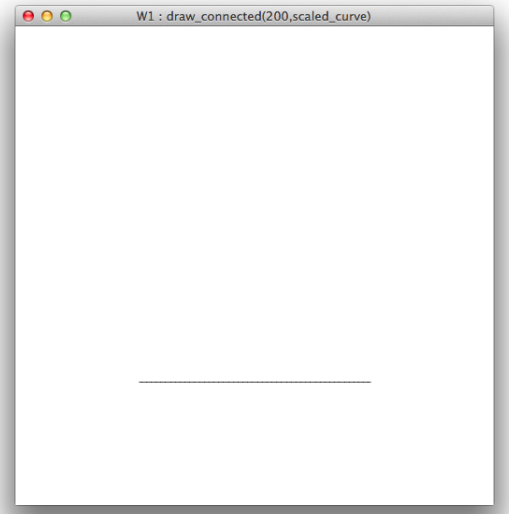
```
def gosper_curve(level):
    return repeated(gosperize, level)(unit_line)
```

[1] A fractal curve is a "curve" which, if you expand any small piece of it, you get something similar to the original. The Gosper curve, for example, is neither a true 1-dimensional curve, nor a 2-dimensional region of the plane, but rather something in between.

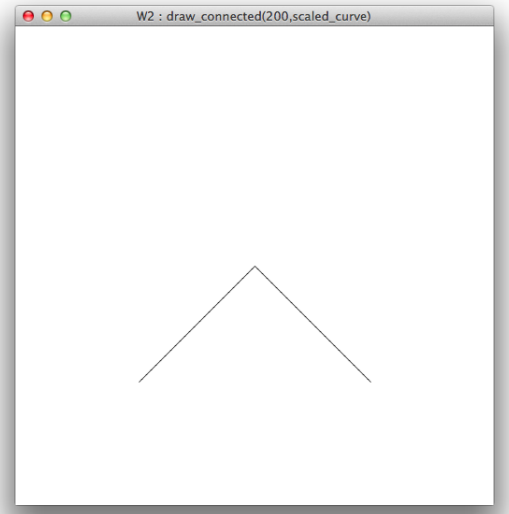
[2] Bill Gosper is a mathematician now living in California. He was one of the original hackers who worked for Marvin Minsky in the MIT Artificial Intelligence Laboratory during the '60s. He is perhaps best known for his work on the Conway Game of Life—a set of rules for evolving cellular automata. Gosper invented the "glider gun", resolving Conway's question as to whether it is possible to produce a finite pattern that evolves into an unlimited number of live cells. He used this result to prove that the Game of Life is Turing universal, in that it can be used to simulate any other computational process!

To look at the level `level` gosper curve, evaluate `show_connected_gosper(level)`:

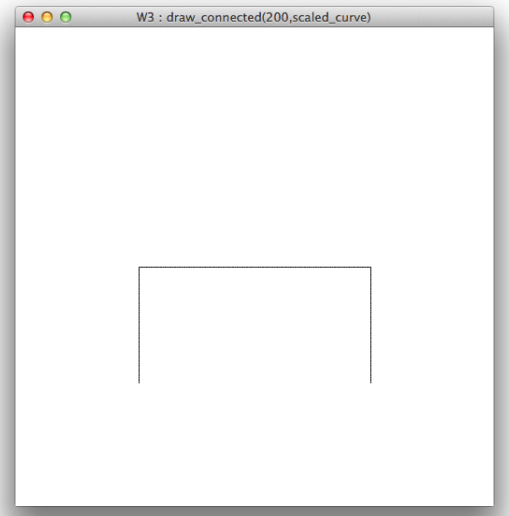
```
def show_connected_gosper(level):
    squeezed_curve = squeeze_curve_to_rect(-0.5, -0.5, 1.5, 1.5) \
                                           (gosper_curve(level))
    draw_connected(200, squeezed_curve)
```



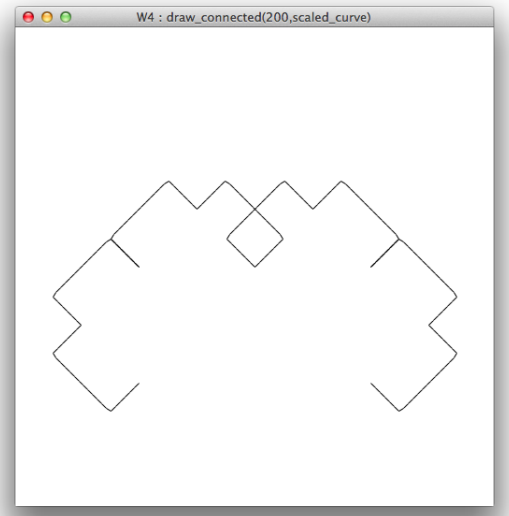
show_connected_gosper(0)



show_connected_gosper(1)



show_connected_gosper(2)



show_connected_gosper(5)

Figure 1: Samples for Gosper Curve.

Your Task:

Define a function `show_points_gosper` such that evaluation of

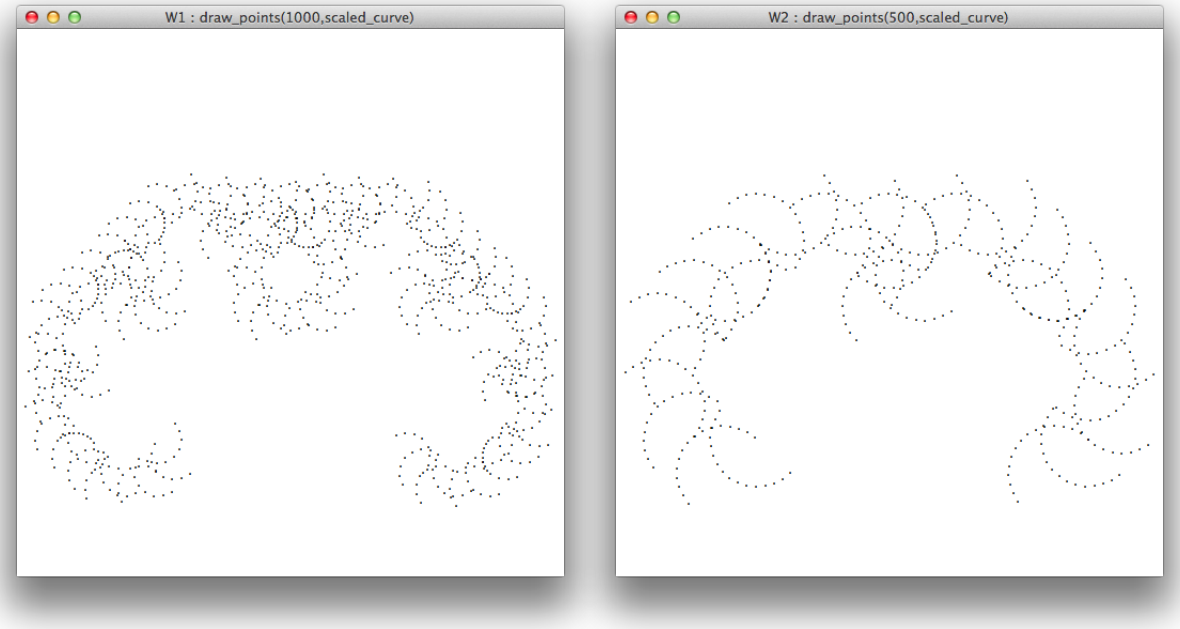
```
show_points_gosper(level, num_points, initial_curve)
```

will plot `num_points` unconnected points of the level `level` gosper curve, but starting the gosper-curve approximation with an arbitrary `initial_curve` rather than the unit line. For instance,

```
show_points_gosper(level, 200, unit_line)
```

should display the same points as `show_connected_gosper(level)`, but without connecting them. However, you should also be able to use your function with arbitrary curves. (You can find the description of function `squeeze_curve_to_rect` in the file `hi_graph.py`; you don't need to understand it in detail to do this task.)

Test your code by gosperizing the arc of the unit circle running from 0 to π. Find some examples that produce interesting designs. (You may also want to change the scale in the plotting window and the density of points plotted.[3]) Some sample tests are given in Figure 2. **Note that you will be penalised for the usage of additional unnecessary transformation functions.**



show_points_gosper(7, 1000, arc) show_points_gosper(5, 500, arc)

Figure 2: Sample Tests for Task 1.

[3]One of the things you should notice is that, for larger values of n, all of these curves look pretty much the same. As with many fractal curves, the shape of the Gosper curve is determined by the Gosper process itself, rather than the particular shape we use as a starting point. In a sense that can be made mathematically precise, the "infinite level" Gosper curve is a fixed point of the Gosper process, and repeated applications of the process will converge to this fixed point.

Task 3: (4 marks)

The Gosper fractals we have been playing with have had the angle of rotation fixed at 45 degrees. This angle need not be fixed. It need not even be the same for every step of the process. Many interesting shapes can be created by changing the angle from step to step.

We can define a function `gosper_curve_with_angle` that generates Gosper curves with changing angles. `gosper_curve_with_angle` takes a level number (the number of levels to repeat the process) and a second argument called `angle_at_level`. The function `angle_at_level` should take one argument, the level number, and return an angle (measured in radians) as its answer:

$$\texttt{angle-at} : (\text{Py-NonNeg-Int}) \rightarrow \text{Py-Num}.$$

The function `gosper_curve_with_angle` can use this to calculate the angle to be used at each step of the recursion:

```
def gosper_curve_with_angle(level, angle_at_level):
    if level == 0:
        return unit_line
    else:
        angle = angle_at_level(level)
        return gosperize_with_angle(angle) \
                            (gosper_curve_with_angle(level-1, \
                                                     angle_at_level))
```

The function `gosperize_with_angle` is almost like `gosperize`, except that it takes an another argument, the angle of rotation:

```
def gosperize_with_angle(theta):
  def inner_gosperize(curve):
      scale_factor = (1 / cos(theta)) / 2
      scaled_curve = scale(scale_factor)(curve)
      left_curve = rotate(theta)(scaled_curve)
      right_curve = translate(0.5,sin(theta)*scale_factor) \
                             (rotate(-theta)(scaled_curve))
      return connect_rigidly(left_curve, right_curve)
  return inner_gosperize
```

For example, the ordinary Gosper curve at level `level` is returned by

```
gosper_curve_with_angle(level, lambda lvl: pi/4)
```

Designing `gosperize_with_angle` required using some elementary trigonometry to figure out how to shift the pieces around so that they fit together after scaling and rotating. It's easier to program if we let the computer figure out how to do the shifting.

One convenient Curve-Transform is `rotate`. Basically,

```
rotate(theta)(arg_curve)
```

will return a curve obtained by rotating `arg_curve` around the origin for an angle of `theta`.

Another convenient Curve-Transform is `put_in_standard_position`. We'll say a curve is in *standard position* if its start and end points are the same as the unit_line, namely it starts at the origin $(0,0)$, and ends at the point $(1,0)$. We can put any curve whose start and end points are not the same into standard position by rigidly translating it so its starting point is at the origin, then rotating it about the origin to put its end point on the x axis, then scaling it to put the end point at $(1,0)$:

```
def put_in_standard_position(curve):
    start_point = curve(0)
    curve_at_origin = translate(-x_of(start_point), \
                                -y_of(start_point))(curve)

    new_end_point = curve_at_origin(1)
    theta = atan2(y_of(new_end_point), x_of(new_end_point))
    curve_ended_at_x = rotate(-theta)(curve_at_origin)
    end_point_on_x = x_of(curve_ended_at_x(1))

    return scale(1/end_point_on_x)(curve_ended_at_x)
```

Your Task:

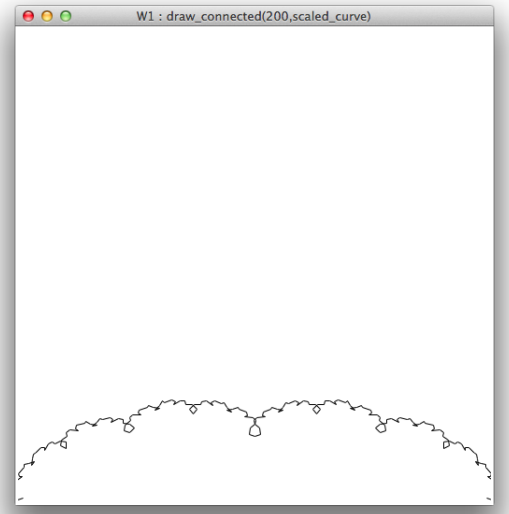
Show how to redefine `gosperize_with_angle` using the functions `connect_ends` and `put_in_standard_position` to handle the trigonometry[4]. In order to avoid naming conflicts, you will be using the function name `your_gosperize_with_angle` instead of the original `gosperize_with_angle`. **Note that you will be penalised for the usage of additional unnecessary transformation functions.**

Your definition should be of the form

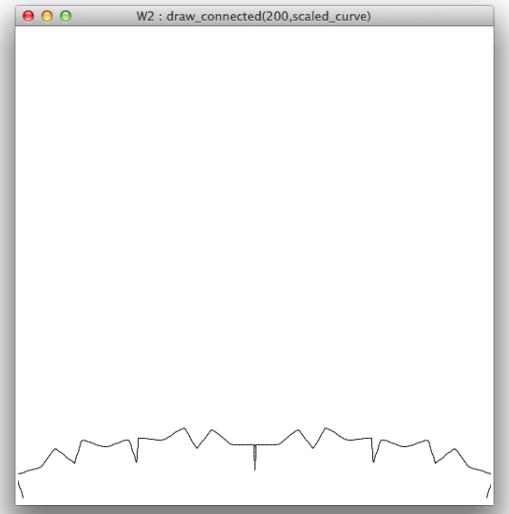
```
def your_gosperize_with_angle(theta):
    def inner_gosperize(curve_fn):
        return put_in_standard_position(connect_ends(..., ...))
    return inner_gosperize
```

To test your code, generate some parameterized Gosper curves where the angle changes with the level n. The function `your_gosper_curve_with_angle` has been defined for you and it uses `your_gosperize_with_angle` for the purpose of testing. Sample answers are shown in Figure 3.

[4]your_gosperize_with_angle must give the same output as the definition of gosperize_with_angle above.



draw_connected(200, your_gosper_curve_with_angle(10, lambda lvl: pi/(2+lvl)))



draw_connected(200, your_gosper_curve_with_angle(5, lambda lvl: (pi/(2+lvl))/(pow(1.3, lvl))))

Figure 3: Sample answers for Task 3.