Mission 6: Gosperize

Start date: 05 September 2017
Due: 11 September 2017, 23:59

Readings:

- Textbook Sections 1.3.2

Background

This mission has **three** tasks.

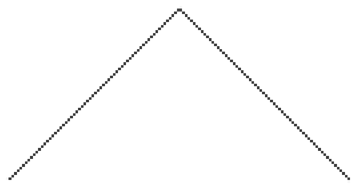
Task 1:

To show off the power of our drawing language, let's use it to explore fractal curves. A fractal curve is a curve which, if you expand any small piece of, you get something similar to the original. The Gosper curve, for example, is a fractal curve that is neither a true 1-dimensional curve, nor a 2-dimensional region of the plane, but something in between. Fractals have striking mathematical properties. Fractals have received a lot of attention over the past few years, partly because they tend to arise in the theory of non-linear differential equations, but also because they are pretty, and their finite approximations can be easily generated with recursive computer programs.

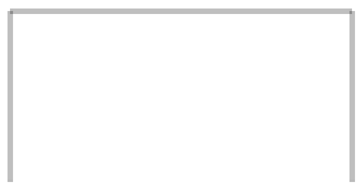
For example, Bill Gosper (see Annex A for more info on him!) discovered that the infinite repetition of a very simple process creates a rather beautiful image, now called the **Gosper Curve**. At each step of this process there is an approximation of the Gosper curve. The next approximation is obtained by adjoining two scaled copies of the current approximation, each rotated by 45 degrees.

The figure below 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.

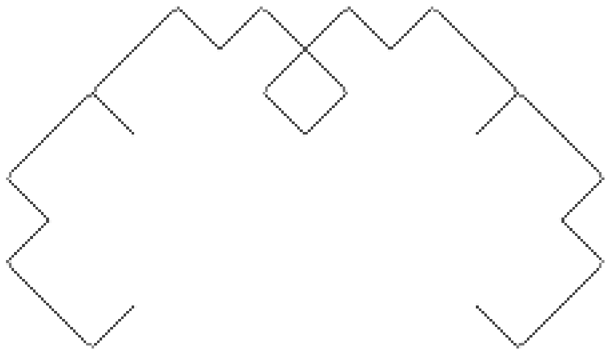
```
show_connected_gosper(0);
```



```
show_connected_gosper(1);
```



```
show_connected_gosper(2);
```



```
show_connected_gosper(5);
```

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 0.5 and up by 0.5, so its beginning coincides with the endpoint of the rotated, scaled first curve.

This leads to the Curve-Transform gosperize:

```
function gosperize(curve){
    var scaled_curve = (scale(math_sqrt(2) / 2))(curve);
    return connect_rigidly((rotate_around_origin(math_PI / 4))(scaled_curve),
                           (translate(0.5, 0.5))
                               ((rotate_around_origin(-math_PI / 4))(scaled_curve)));
}
```

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

```
function gosper_curve(level){
    return (repeated(gosperize, level))(unit_line);
}
```

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

```
function show_connected_gosper(level){
    return (draw_connected(200)) ((squeeze_rectangular_portion(-0.5, 1.5, -0.5, 1.5))
                                  (gosper_curve(level)));
}
```

Useful functions that you might need:

```
function compose (f, g) {
    return function (x) {
        return f(g(x));
    };
}

function identity(x) {
    return x;

}

function repeated(f, n) {
    if (n === 0) {
        return identity;
    } else {
        return compose(f, repeated(f, n - 1));
    }
}
```

Your Task:

Define a function show_points_gosper such that evaluation of

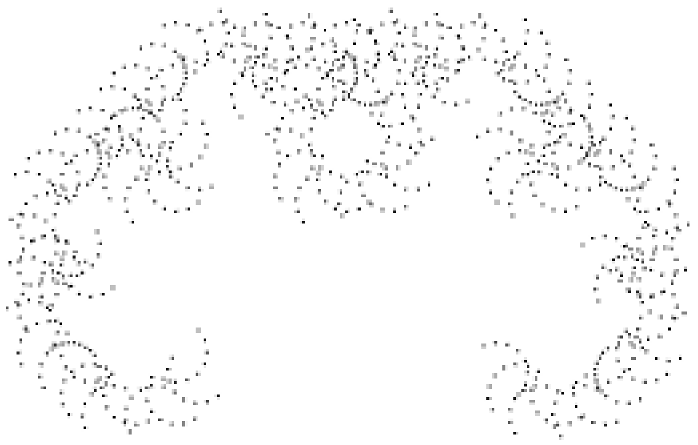
```
show_points_gosper(level, number_of_points, initial_curve);
```

will plot `number_of_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,

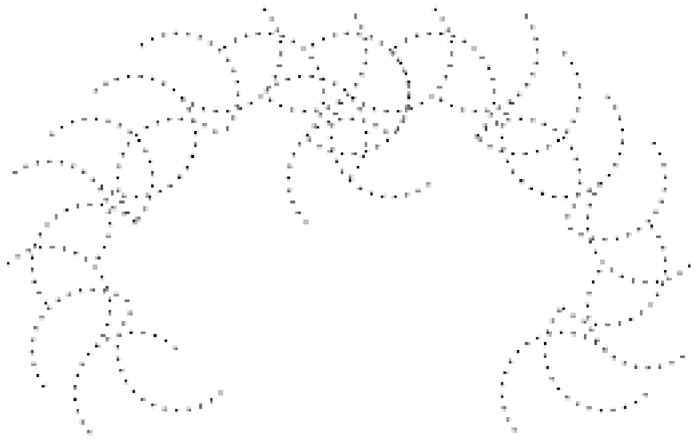
```
show_points_gosper(level, 200, unit_line);
```

should display the same points as `show_connected_gosper(level)`, but without connecting them. Your function should also work with arbitrary curves, and not just unit_line.

Test your program by gosperizing the arc of the unit circle running from 0 to π, which has been defined for you as `arc`. 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). 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. Some sample tests are given in the figures below.



```
show_points_gosper(7, 1000, arc);
```



```
show_points_gosper(5, 500, arc);
```

Task 2:

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 varying the angle with each step.

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

```
angle-at : Natural-Number -> Number.
```

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

```
function param_gosper(level, angle_at){
    if (level === 0){
        return unit_line;
    } else {
        return (param_gosperize(angle_at(level)))(param_gosper(level - 1, angle_at));
    }
}
```

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

```
function param_gosperize(theta){
    return function(curve){
               var scale_factor = 1 / math_cos(theta) / 2;
               var scaled_curve = (scale(scale_factor))(curve);
               return connect_rigidly((rotate_around_origin(theta))(scaled_curve),
                                      (translate(0.5, math_sin(theta) * scale_factor))
                                        ((rotate_around_origin(-theta))(scaled_curve)));
           };
}
```

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

`param_gosper(level, function(level){ return math_PI/4; })`

Designing `param_gosperize` requires 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_around_origin`. Basically,

`(rotate_around_origin(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 endpoints 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 endpoint on the `x` axis, then scaling it to put the endpoint at `(1,0)`:

```
function put_in_standard_position(curve){
    var start_point = curve(0);
    var curve_started_at_origin = (translate(-x_of(start_point),
                                             -y_of(start_point)))(curve);
    var new_end_point = curve_started_at_origin(1);
    var theta = math_atan2(y_of(new_end_point), x_of(new_end_point));
    var curve_ended_at_x_axis = (rotate_around_origin(-theta))
                                    (curve_started_at_origin);
    var end_point_on_x_axis = x_of(curve_ended_at_x_axis(1));
    return (scale(1 / end_point_on_x_axis))(curve_ended_at_x_axis);
}
```

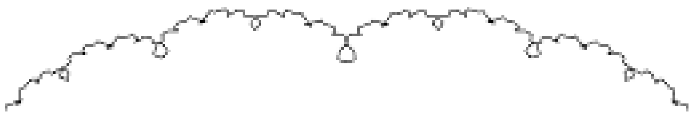
Your Task:

Show how to redefine `param_gosperize` using the functions `put_in_standard_position` and `connect_ends` to handle the trigonometry. `your_param_gosperize` must produce the same output as `param_gosperize` above. In order to avoid naming conflicts, you will be using the function name `your_param_gosperize` instead of the original `param_gosperize`.

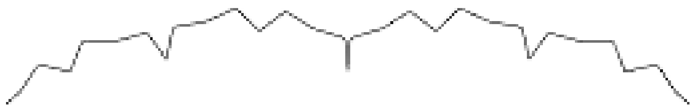
Your definition should be of the form

```
function your_param_gosperize(theta){
    return function(curve){
                return put_in_standard_position(connect_ends(...,
                                                             ...));
           };
}
```

To test your program, generate some parameterized Gosper curves where the angle changes with the level `n`. The function `your_param_gosper` has been defined for you and it uses `your_param_gosperize` for the purpose of testing. Sample answers are shown below.



```
(draw_connected(200))(your_param_gosper(10, function(n) {return math_PI / (n
+ 2); }));
```



```
(draw_connected(200))(your_param_gosper(5, function(n) { return math_PI / 4
/ math_pow(1.3, n); }));
```

Task 3:

We now have three functions to compute gosper curves:

1. `gosper_curve`
2. `param_gosper` with argument `function(level) { return math_PI / 4; }` using the "hand-crafted" definition of `param_gosperize` above
3. `your_param_gosper` in Task 2 that uses `your_param_gosperize` based on `put_in_standard_position`.

One of the functions provided in our library is `timed`. `timed` takes in a function `f`, and returns a function `g` that performs the same computation as `f`. When an argument is passed to `g`, it will perform the computation, returning whatever value `f` would have returned had the argument been passed to `f`, and display on the interpreter how long this computation took.

For example, evaluating

```
(timed((timed(gosper_curve))(4000)))(0.1);
```

will print out the time to generate the level 4000 gosper_curve, and then the time to compute the point at 0.1. Be warned that the resolution of `timed` is not very high. The accuracy of any elapsed time values below 20 milliseconds is suspect.

Your Task:

Compare the time measurements of these functions for computing selected points on the curve at a **significant level**. Significant level refers to one that does not take a ridiculous amount of time to perform the function, but enough to show the difference in time measurements between the different functions. Do this **at least 5 times** and take the average time measurements for each of the functions for more accurate results. Present your findings in a **neat and organized manner**. Point out a key difference between these functions that contributes to the difference in speeds.

Submission

To submit your work for this mission, copy the url on your browser and email it to your respective Avengers. Strictly follow to the deadlines set at the start of this file.

IMPORTANT: Make sure that everything for your programs to work is on the left hand side and **not** in the interpreter pane on the right! This is because only that program is preserved in the url you have emailed to us.

Annex A: Bill Gosper

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!