Spiral.

What is a spiral? Regarding the math, it is first of all represented as a 2-dimensional object. In 2-dimensions, how does one represent a *point* z ?

One represents it as the product of 2 independent 1-dimensional points u, v, hence:

$$z := (u, v)$$

But the story doesn't end there. There are two main coordinate systems in representing 2D points. This is to say, there are two orientations as such: a local one, and a global one.

The better known global system is the *rectlinear coordinate system*: (x, y) where x tells you to move left or right, and y tells you to move up or down. There are many specialized graphs generated when one of these points y can be expressed in terms of the other x:

$$y = f(x) \quad \text{for some “rule”} \quad f$$

The other coordinate orientation is the *polar coordinate system*: (r, θ) where r is the radius away from the origin (thus, the idea of location is localized to the viewpoint of the origin), and θ is the angle from a ray that starts at the origin (usually this angle-origin or ray-origin is the x-axis). As with the rectlinear system, there are many specialized graphs generated when one of these points r is expressed in terms of the other θ:

$$r = f(\theta) \quad \text{for some “rule”} \quad f$$

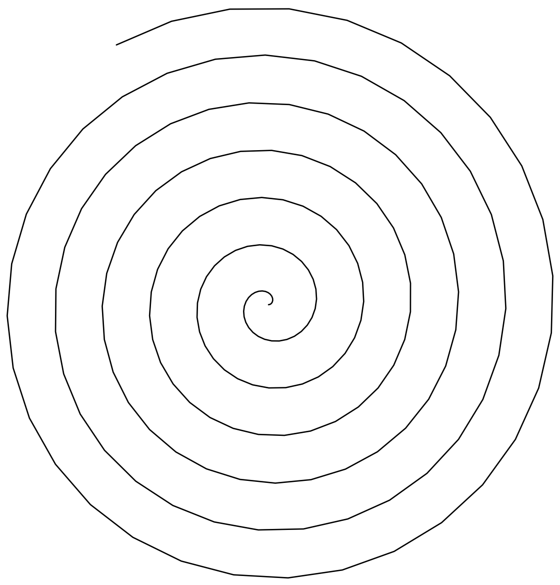
So what is a spiral? Think of it like this: If you start at the center, then as you move along the path, you are both continually rotating yourself around that center, but as well, you are moving outward. If you give it a little thought, you'll realize that if you were to represent this in one of these two potential 2D systems, it would be easier doing so in the polar system. As you increase the angle θ you also increase the radius r, this is to say: $r = f(\theta) = \theta$, or to short-form this: $r(\theta) = \theta$. This incidentally is the *identity function*.

Finally, we need to translate this into rectlinear coordinates (as many software programs that will draw your spiral for you don't recognize polar coordinates). For it, you need these rules of translation:

$$x = r\cos(\theta), \quad y = r\sin(\theta)$$

and since we know $r(\theta) = \theta$, we can remove the r within these rules to define (x, y) in the simpler terms of θ alone:

$$x = \theta\cos(\theta), \quad y = \theta\sin(\theta)$$



As you'll note though in the above graphic, this spiral is nice and smooth in the center, but becomes choppy at the edges. The reason for this is the following code:

```
import graph;
unitsize(1cm);

path spiral(int begin=0, int end, real scalar, real step(int), real dir=1)
{
        path spiral;
        for (int k=begin; k < end; ++k)
        {
                real theta=step(k);
                real s=scalar*theta;
                real t=dir*theta;
                spiral=spiral--s*(cos(t), sin(t));
        }

        return spiral;
}

path angle_spiral(int count, real scalar=0.1, real angle=0.2, real dir=1)
{
        real step_angle(int k) {return angle*k;}
        return spiral(end=count, scalar, step_angle, dir);
}
```

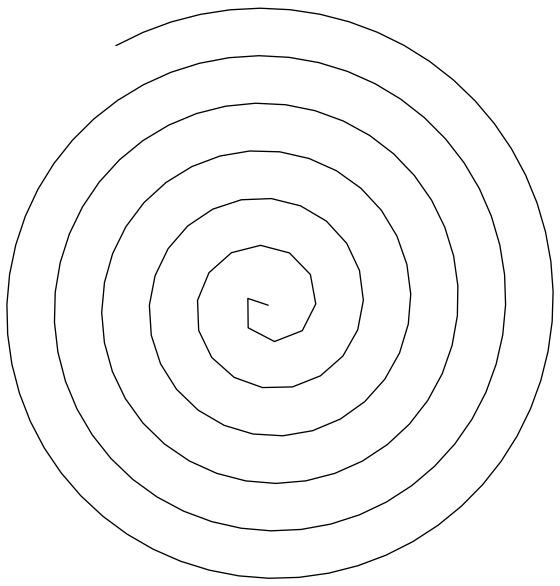
The code is written in *Asymptote*, the final call has $count = 200$. As you can see I am approximating the spiral by line segments, but as each new point in the segment is defined by a fixed size increment of the angle, the points become further and further apart as one goes outward.

As an aside: Yes! I know, this code isn't optimized for efficiency, but given it only needs to be rendered and re-rendered a handful of times to make this exact document, I'm being lazy. Thanks for noticing :)

Okay, so what can we do about the approximation problem? Instead of fixing the size of the angle increments, we can fix the size of the arc-length increments when generating the spiral. This is done with the following code (continuing from the previous code):

```
path arc_spiral(int count, real scalar=0.1, real length=0.398, real dir=1)
{
        real step_arc(int k) {return sqrt(2length/scalar)*sqrt(k);}
        return spiral(end=count, scalar, step_arc, dir);
}
```

One then ends up with this:



Notice though now, the choppiness is backward? The center is choppy and the edges are smooth. That doesn't solve our problem, it just creates a new one.

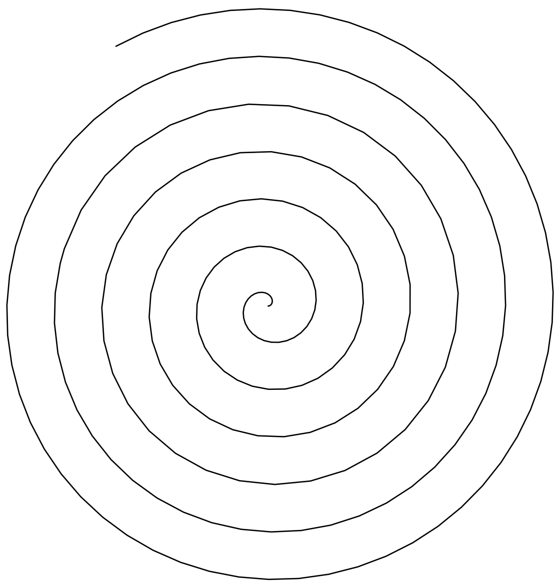
Another approach then is to fix the line segment lengths, but I've taken a look at the math, and although I can "brute-force" the computations needed to generate the points (path) that make up what is otherwise the spiral, I have not found a way to simplify the computational complexity. In simpler terms: The tradeoff of a slight perceptual improvement is not worth the increase in programming and computational complexity. This is especially true because one can simply take a hybrid method:

Since the first spiral approximation is smooth from the start of the spiral to somewhere in the middle, and the second spiral approximation is smooth from somewhere in the middle to the end, then why not just use the first half of the first, and the second half of the second?

Here's the code:

```
path hybrid_spiral(int count, real scalar=0.1, real angle=0.2, real dir=1, real div=0.5)
{
        real length=scalar*(count-1)*(angle*angle)/2;
        real step_angle(int k) {return angle*k;}
        real step_arc(int k) {return sqrt(2length/scalar)*sqrt(k);}
        return spiral(end=ceil((count-1)*sqrt(div)), scalar, step_angle, dir)--
                spiral(begin=floor((count-1)*div), end=count, scalar, step_arc, dir);
}
```

And here's the hybrid spiral:



Yes! you might say “Hey, you can still see the line segments!”, but notice how they are by human perception standards relatively even in length with respect to each other? At this point then, if you can still see the “straightness”, it’s just a matter of fine tuning the approximation. In this case, one would take a smaller angle and extend the count to match.

And that people, is how it’s done.

fin