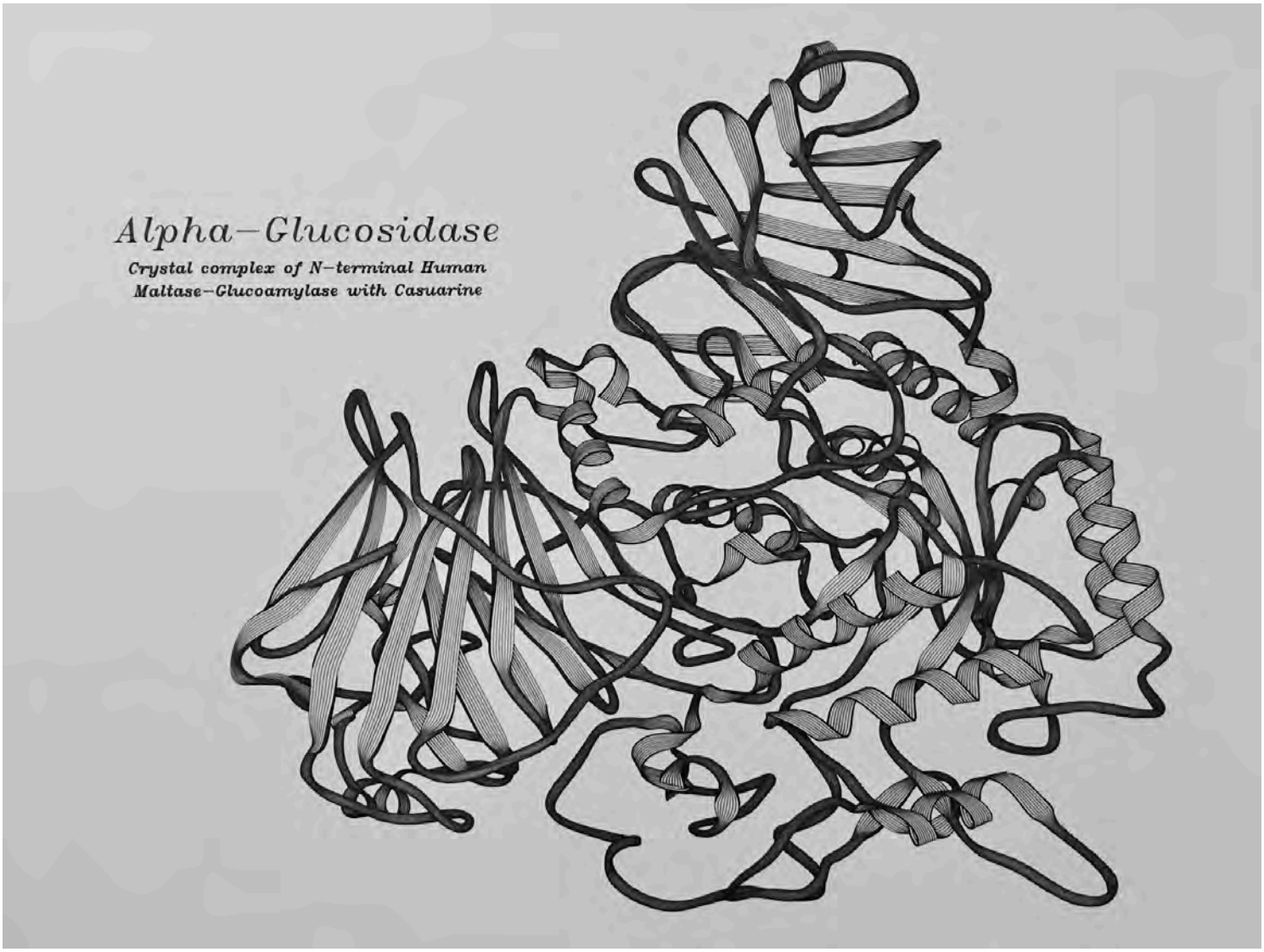
Pen Plotter Programming: The Basics

Michael Fogleman Follow 5 min read · Mar 28, 2017



Alpha-glucosidase ribbon diagram drawn with an AxiDraw v3 pen plotter.

Plotters were a thing before we had nice laser and inkjet printers. But now they're coming back like 80s music (lol), this time at the intersection of code

and art. Compared to printers, which are way faster, plotters are nice because they produce results that look hand-drawn. And it's fun to watch them draw! I've now purchased two modern hobbyist plotters, first the Makeblock XY Plotter and more recently the AxiDraw v3. I used each purchase as an opportunity to hack up some code for controlling the plotter and for generating interesting artwork.

In this article I'll share a few tidbits of information when it comes to programming plotters to do their plotting efficiently. Note that none of this is necessary if you're using the software included with the plotters, but it's useful if you're trying to programmatically control the plotters yourself.

These plotters basically consist of two stepper motors (one for the X-axis and one for the Y-axis) and a servo motor to move the pen up and down. You can put in just about any drawing implement (pen, marker, pencil, charcoal, etc.) and move it around on the paper to draw something.

We're dealing with vector coordinates here and ultimately it all boils down to a bunch of line segments. Curves? Series of segments. Fills? Gonna have to make multiple passes or do some cross-hatching. In terms of data structures, we can look at it this way. A **Point** is an (x, y) coordinate. A **Path** is an ordered sequence of Points. And a **Drawing** is a set of Paths that we wish to draw (typically drawing order doesn't matter).

The servo motor allows moving the pen to a specific vertical position, but we can generally think of this as a binary: up (not drawing) or down (drawing). You might imagine trying to vary the pen pressure, but this is very difficult without a perfectly flat and parallel surface, and there's very little motor resolution between not-touching and completely-touching, so we won't consider that further.

So, our language to the plotter consists of these actions: pen up, pen down, and move to position. We write some code to generate our Drawing, and now we want to send it to the plotter. Naively, we can just iterate through the paths and draw them:

```
for each path in drawing:
    move to first point in path
    lower pen
    for each point in path:
        move to point
    raise pen
```

This will work, but we'll quickly become impatient, scrutinizing the inefficient ways of the robot doing its work. The plotter will spend a lot of time traveling, with pen up, from one side of the page to another. It will raise

and lower the pen at essentially the same position for no obvious reason. Let's make it better.

Ordering Paths

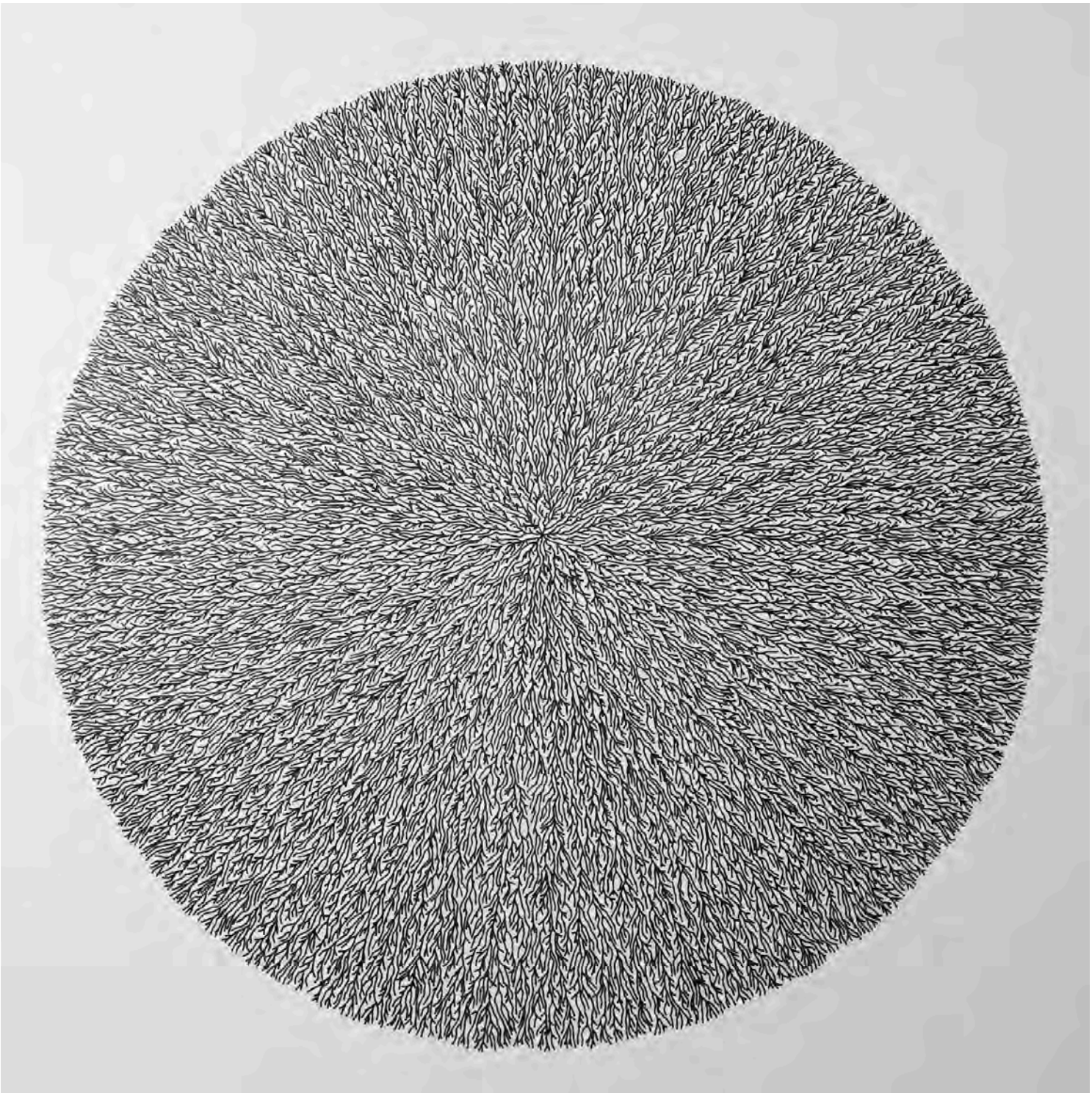
First, we want to re-order the paths such that non-drawing movement is minimized. This is basically the traveling salesman problem, which cannot be solved in the general case. But a greedy solution works well enough. Start with an arbitrary path and draw it. Next, find the nearest undrawn path and draw it. Repeat. Note that a path has two endpoints! So, if we'll allow it, the algorithm can draw a path in reverse if that endpoint is nearest. Imagine we want to draw a bunch of parallel lines starting at X = 0 and ending at X = 1, at various Y values. Most efficiently we would draw a line from 0 to 1 and draw the next from 1 to 0, alternating direction each time. To compute this ordering efficiently, I suggest a spatial hash index.

Joining Paths

Once we order the paths, we can join together consecutive paths whose endpoints are within some tolerance. This way if path A ends very close to where path B starts (within the diameter of the pen tip, basically), we don't need to bother lifting the pen. This can save us a lot of time too, depending on the nature of the drawing. This is trivial to implement.

Simplifying Paths

Once we've sorted and joined the paths, we should simplify them, again to some specified tolerance. The Ramer–Douglas–Peucker algorithm is an easy way to implement polyline simplification. This reduces the amount of data we need to send to the plotter. But more importantly it improves the results of the motion planning algorithm. (I implemented my own constant-acceleration motion planner, but that deserves an entire article of its own.)



Growth pattern based on Poisson-disc sampling. Drawn with a Makeblock XY Plotter.

Some other conveniences include scaling and/or rotating the drawing to fit the page, centering the drawing on the page, and rendering a PNG of the drawing so we can see how it will look. It's also nice to have a progress bar to see how much time is remaining (simple drawings can be done in under a minute, but complex drawings can easily take over an hour).

I also implemented a turtle graphics interface for generating drawings. That makes it really elegant to generate, for example, a Dragon curve.

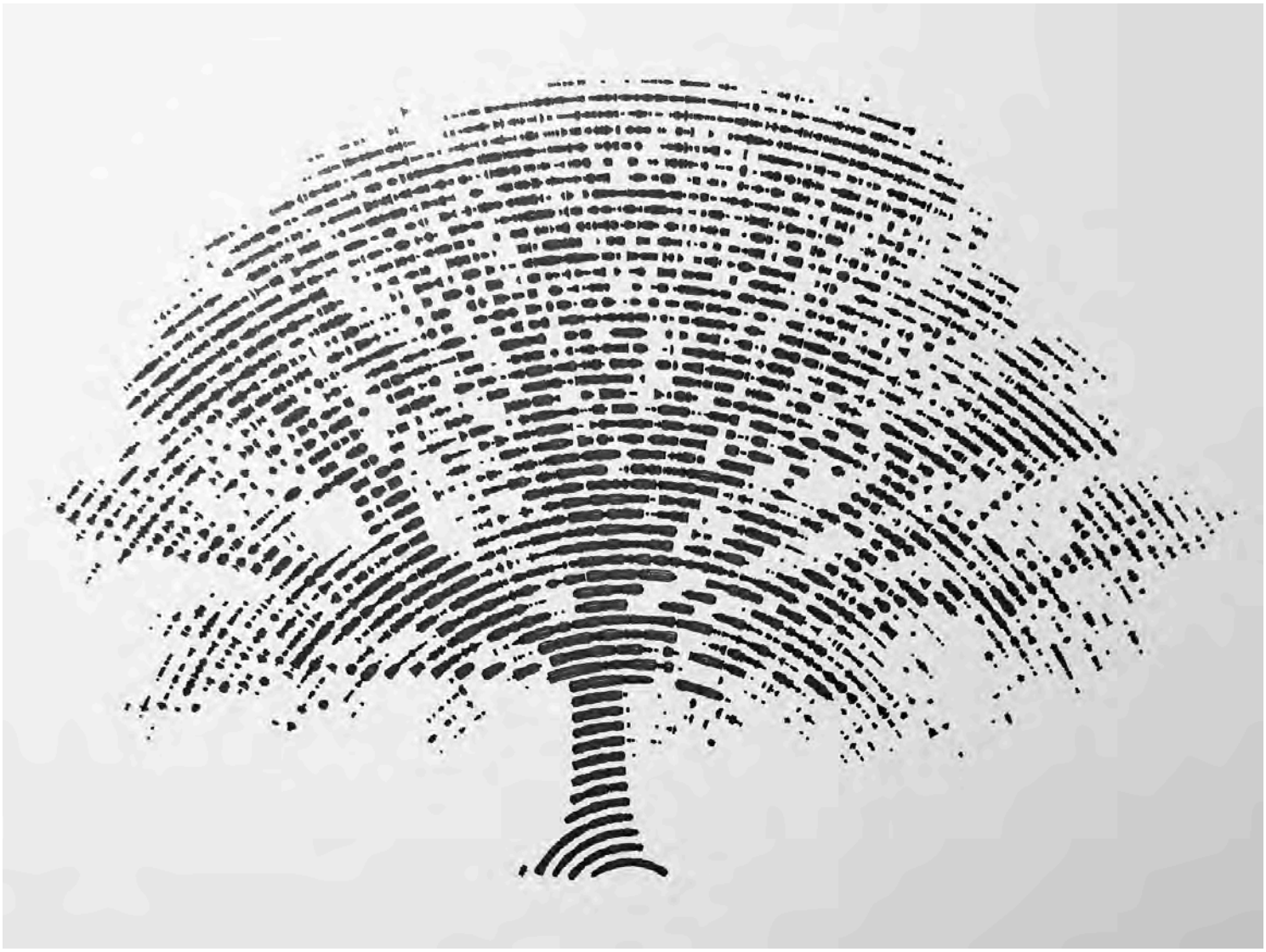
```
# generate a dragon curve using turtle graphics

import xy

def main(iterations):
    t = xy.Turtle()
    for i in range(1, 2 ** iterations):
        t.forward(1)
        if (((i & -i) << 1) & i) != 0:
            t.circle(-1, 90, 36)
        else:
            t.circle(1, 90, 36)
    xy.draw(t.drawing)

if __name__ == '__main__':
    main(13)
```

These features are all available in my xy (for Makeblock XY) and axi (for AxiDraw) Python libraries.



"Tree Rings" — drawn with a Makeblock XY Plotter.