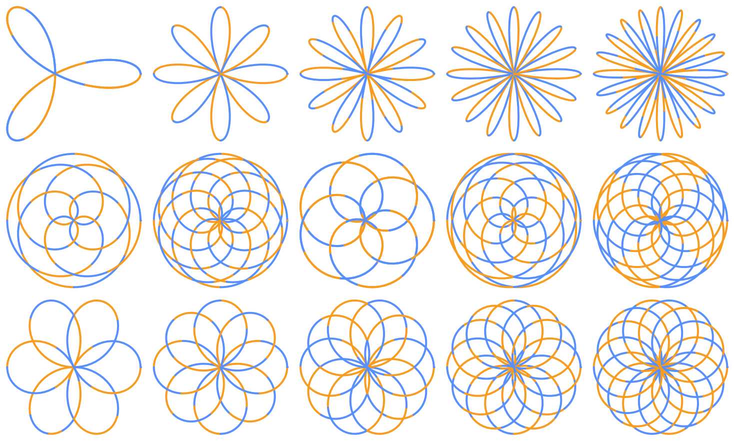


Roses (or rhodonea curves), plotted using the orthogonal distance fitting algorithm of this blog post. Each rose is represented as a path of connected Bezier curves. The alternating colors identify the individual segments of the paths. Figure by author.

How to Use Orthogonal Distance Fitting to Produce Ultra-compact Vector Graphic Plots

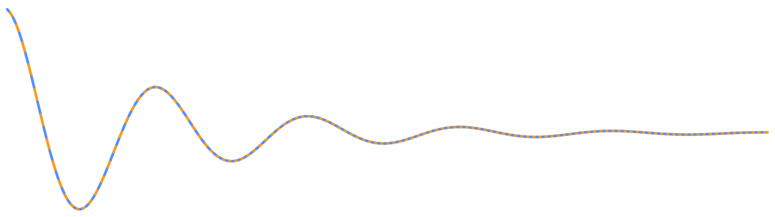
Ryan Burn

1. Introduction

Vector graphics are a natural medium for visualizing mathematical functions. With vector graphics, a function is approximated by segments of connected cubic Bezier curves that are rasterized (i.e. converted into pixels) when they are displayed [1]. Because rasterization is delayed, vector graphic images are naturally more portable than pixel-based images as the rendering can be tailored to its display environment. No matter how much you zoom in on a vector graphics plot, you will always see crisp line segments; whereas if you zoom in enough on a rasterized image, you will eventually see the grainy blocks that represent the pixels.

While vector graphics are an excellent format for plotting, producing good vector graphics can be challenging. Consider, for example, how we might adopt this example plot from matplotlib [2] for the function $f(t) = \exp(-t)\cos(-2\pi t)$, $0 \leq t \leq 5$ to produce an SVG image:

```
import matplotlib.pyplot as plt
import numpy as np
def f(t):
  return np.exp(-t) * \
     np.cos(2*np.pi*t)
t = np.arange(0.0, 5.0, 0.02)
plt.plot(t, f(t))
plt.savefig('fig.svg')
```



Matplotlib approximates the smooth plot with 250 piecewise connected line segments (shown with the alternating colors). While the plot looks good, it is much larger than it needs to be. If, instead, we were producing a rasterized image like a PNG, the details of how the curve is constructed wouldn't matter; but with a vector graphic the individual line segments are passed through and preserved in the outputted image. With some adjustments, though, we can improve the size substantially without sacrificing the quality of the image.

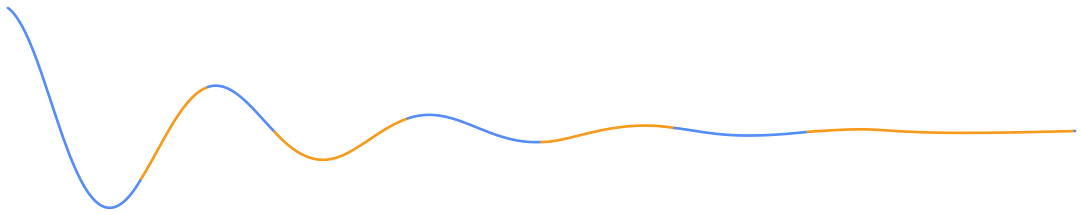
The basic primitive of vector graphics is the parametric cubic, represented as a cubic Bezier curve. With piecewise cubic Bezier curves, we have a lot more knobs we can adjust to approximate functions than if we restrict ourselves to piecewise linear or piecewise (nonparametric) cubic segments. Using the orthogonal distance fitting (ODF) algorithm from this paper, we can produce a plot of the function that requires only 8 Bezier segments and

[1] As far as I'm aware, all of the major vector graphics formats (e.g. SVG, postscript) use cubic Bezier curves as the core primitive. While some of the formats provide other basic graphics like circles, etc, those are all just wrappers on top of cubic Bezier curve approximations.

[2] From `https://matplotlib.org/stable/gallery/pyplots/pyplot_two_subplots.html`

is visually indistinguishable from the matplotlib graphic. Below I show the plot's representation in TikZ. Later I will show how we can easily turn the TikZ command into an SVG image for the web using MetaPost.

```
\draw  (0.00000, 2.50000) ..  controls (0.39088, 2.10882) and (0.61233, -1.05415) ..
       (1.18750, 0.36510) ..  controls (1.60393, 1.31193) and (1.71252, 2.10800) ..
       (2.37500, 0.95127) ..  controls (2.88922, 0.15365) and (3.15477, 0.95162) ..
       (3.56250, 1.11921) ..  controls (3.98299, 1.31658) and (4.26123, 0.78781) ..
       (4.75000, 0.82415) ..  controls (5.02221, 0.81871) and (5.38203, 1.12388) ..
       (5.93750, 0.99939) ..  controls (5.96426, 1.01981) and (6.36975, 0.82102) ..
       (7.12500, 0.95127) ..  controls (8.08129, 1.04760) and (7.44859, 0.87986) ..
       (9.50000, 0.96171);
```



The algorithm builds on Alvin Penner's work [1]. Given a parametric function f, it first fits a Chebyshev series to approximate f. For analytic functions, interpolation in Chebyshev nodes provides rapid geometric convergence [2]. This is far better than the typically fourth order convergence you will get with interpolation in cubic splines and much better than the quadratic convergence you will get with piecewise linear interpolation [3]. By interpolating in Chebyshev nodes, we can reduce the number of times we need to evaluate f. Using a trust-region optimizer, the algorithm then looks for a cubic Bezier curve that optimally approximates the target function. If the maximal orthogonal distance between the fitted Bezier curve and the Chebyshev series for f is less than a target threshold, then, great, we're done; otherwise, we split the domain in two and repeat the process. I detail the steps below.

Algorithm F (*fit Bezier path to a function*). Given a parametric function $f(t) = (f_x(t), f_y(t))$, $a \le t \le b$, fit a Bezier path, g, to f so that

$$\text{argmax}_t \{ \text{argmin}_s \|g(s) - f(t)\| \} < \text{target threshold}.$$

F1. Using the algorithm developed by Jared Aurentz and Lloyd Trefethen from [5], fit Chebyshev series $\tilde{f}_x$, $\tilde{f}_y$ to f_x, f_y. When f is analytic or differentiable to a high degree, the fit is usually close to machine precision so going forward we assume that we can use $\tilde{f}$ as a proxy for f and that any loss in accuracy will be negligible.

F2. Using a trust region optimizer [6] and Penner's algorithm [1], fit a Bezier curve g to minimize

$$\int_a^b \text{argmin}_s \left\| g(s) - \tilde{f}(t) \right\|^2 \sqrt{\tilde{f}_x'(t)^2 + \tilde{f}_y'(t)^2}\, dt.$$

More details for this step are provided in §3.

F3. Compute

$$M = \text{argmax}_t \{ \text{argmin}_s \left\| g(s) - \tilde{f}(t) \right\| \}.$$

If M is less than the target threshold, terminate; otherwise, set

$$f_l(t) = \tilde{f}(t) \quad \text{for} \quad a \le t \le \frac{a+b}{2}, \qquad f_r(t) = \tilde{f}(t) \quad \text{for} \quad \frac{a+b}{2} \le t \le b,$$

and repeat steps F1 through F3 for f_l and f_r until the threshold is reached. See §4 for more details.

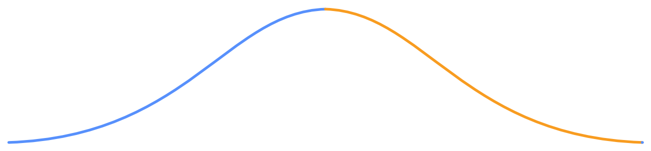
In the next section, I describe how to fit arbitrary functions with Algorithm F using the Python package `bbai` (`https://github.com/rnburn/bbai`).

2. Functionality Tour

The below code demonstrates the basic task of fitting a function to a specified window:

```
from bbai.graphics import \
    BezierPath
import numpy as np
def f(t):
  return np.exp(-t * t)
path = BezierPath(
    dst_xmin=0, dst_xmax=9.5,
    dst_ymin=0, dst_ymax=2)
path.fit(f, -2, 2)
print(path.tikz_)
```

```
\draw (0.000, 0.000)..controls (2.684, 0.092) and (3.273, 1.952)
..    (4.750, 2.000)..controls (6.229, 1.951) and (6.815, 0.092)
..    (9.500, 0.000);
```

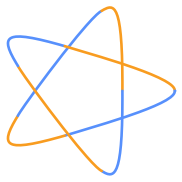


By default, the library will scale the plot so that the function just fits in the window defined by `dst_xmin`, `dst_xmax`, `dst_ymin`, and `dst_ymax`; but that can be changed by also specifying a source window with `src_xmin`, `src_xmax`, `src_ymin`, and `src_ymax`. The algorithm uses 1.0×10^{-2} as the default maximum orthogonal distance threshold which, for perspective, is one hundredth of TikZ's default line width.

We can also fit parametric functions by providing both an x and a y function.

```
from bbai.graphics import \
    BezierPath
import numpy as np
R, r, d = 5, 3, 5
def fx(t):
  return (R-r)*np.cos(t) \
    + d*np.cos((R-r)/r*t)
def fy(t):
  return (R-r)*np.sin(t) \
    - d*np.sin((R-r)/r*t)
path = BezierPath(
    dst_xmin=0, dst_xmax=2.5,
    dst_ymin=0, dst_ymax=2.5)
path.fit(fx, fy, 0, 2*np.pi*r*d/R)
print(path.tikz_)
```

```
\draw (2.500, 1.250)..controls (2.533, 1.060) and (1.400, 0.745)
..    (0.850, 0.578)..controls (-0.049, 0.321) and (-0.145, 0.403)
..    (0.148, 0.876)..controls (0.197, 0.986) and (1.203, 2.351)
..    (1.405, 2.450)..controls (1.594, 2.564) and (1.722, 2.598)
..    (1.716, 1.250)..controls (1.722, -0.033) and (1.609, -0.085)
..    (1.405, 0.049)..controls (1.203, 0.149) and (0.197, 1.514)
..    (0.149, 1.624)..controls (-0.137, 2.086) and (-0.067, 2.185)
..    (0.851, 1.921)..controls (1.203, 1.809) and (2.534, 1.455)
..    (2.5000, 1.2500);
```



3. The Fitting Algorithm

This section breaks down Step F2 in greater detail where we fit a Bezier curve, g, to approximate $\tilde{f}$. The algorithm is similar to Penner's algorithm from [1] but with a few modifications. A cubic Bezier curve is parameterized by four points. In the fitting algorithm, we match the endpoints of $\tilde{f}$ which gives us two points or four parameters that we can adjust[1]. Let θ denote the adjustable parameters; let B_θ denote the cubic Bezier curve with parameters θ that matches the endpoints $\tilde{f}(a)$ and $\tilde{f}(b)$; and define

$$s_t = \operatorname{argmin}_s \left\| B_\theta(s) - \tilde{f}(t) \right\|^2, \qquad 0 \le s \le 1.$$

Basic calculus tells us that s_t must either be an endpoint or it must satisfy the equation

$$\left\langle \tilde{f}(t) - B_\theta(s_t),\ B'_\theta(s_t) \right\rangle = 0.$$

As B_θ is a parametric cubic, the values of s that satisfy the equation are the roots of a quintic which can be easily solved for by finding the eigenvalues of its associated colleague matrix [2].

[1] Matching derivatives at the endpoints might also be a good approach. That would give us two parameters we could adjust instead of four.

We can use a Clenshaw Curtis quadrature to approximate the objective function

$$\begin{aligned} h(\theta) &= \int_a^b \left\| B_\theta(s_t) - \tilde{f}(t) \right\|^2 \sqrt{\tilde{f}'_x(t)^2 + \tilde{f}'_y(t)^2}\, dt \\ &\approx \sum_{i=1}^N \left\| B_\theta(s_{t_i}) - \tilde{f}(t_i) \right\|^2 \sqrt{\tilde{f}'_x(t_i)^2 + \tilde{f}'_y(t_i)^2}\, w_i. \end{aligned}$$

The gradient and Hessian of h can be computed using the implicit function theorem. See [1] for the equations. We can now put together the fitting algorithm.

Algorithm B (*fit a Bezier curve to a function*). Given a parametric function $\tilde{f}(t) = (\tilde{f}_x(t), \tilde{f}_y(t))$, $a \le t \le b$, find parameters θ to minimize $h(\theta)$.

B1. If possible, select θ_0 so that B_{θ_0} matches the curvature of $\tilde{f}$ at a and b; otherwise, select θ_0 so that B_{θ_0} is the line segment that passes through $\tilde{f}(a)$ and $\tilde{f}(b)$.

B2. Starting from θ_0 and using h, ∇h, and $\nabla^2 h$, step a trust-region optimizer [6] until either we've come suitably close to an optimum or we've exceeded a predetermined maximum number of steps.

The major advantage of using a trust-region optimizer for Step B2 is that it won't get stuck at saddle points. By using second order information combined with an adaptive "trust region", a trust-region optimizer can still make progress even when ∇h is near zero and $\nabla^2 h$ is indefinite.

4. The Max Distance Algorithm

This section breaks down Step F3 in greater detail where given a cubic Bezier curve, g, we compute the maximum orthogonal distance from g to $\tilde{f}$. With s_t defined as in §3, put

$$r(t) = \left\| g(s_t) - \tilde{f}(t) \right\|^2$$

and define the following algorithm:

Algorithm M (*find maximum orthogonal distance*). Given a parametric function $\tilde{f}(t) = (\tilde{f}_x(t), \tilde{f}_y(t))$, $a \le t \le b$ and a Bezier curve g, solve

$$\text{argmax}_t\ r(t).$$

M1. Set $D_{max} \leftarrow 0$.

M2. Using the bisection method, start from the interval $[a, b]$ and find a local maximum t_0 of r. Set $D_{max} \leftarrow max(D_{max}, r(t_0))$.

M3. Put $s_0 \leftarrow s_{t_0}$ and let s'_{l0}, s''_{l0}, s'_{r0}, and s''_{r0} denote the left and right derivatives for

$$\frac{d}{dt} s_t \Big|_{t=t_0} \quad \text{and} \quad \frac{d^2}{dt^2} s_t \Big|_{t=t_0}.$$

Note that the left-hand and right-hand values may differ. Define the functions

$$\begin{aligned} \tilde{r}_l(t) &= \left\| g\left(s_0 + s'_{l0}(t - t_0) + \frac{s''_{l0}}{2}(t - t_0)^2 \right) - \tilde{f}(t) \right\|^2 \quad \text{and} \\ \tilde{r}_r(t) &= \left\| g\left(s_0 + s'_{r0}(t - t_0) + \frac{s''_{r0}}{2}(t - t_0)^2 \right) - \tilde{f}(t) \right\|^2. \end{aligned}$$

M4. Find the smallest $h_l > 0$ and $h_r > 0$ such that

$$\tilde{r}_l(t_0 - h_l) = r(t_0) \quad \text{and} \quad \tilde{r}_r(t_0 + h_r) = r(t_0).$$

If suitable h_l and h_r exist, repeat Steps M2 through M4 for $[a,\, t_0 - h_l]$ and $[t_0 + h_r, b]$. Otherwise, return D_{max}.

Remember from Algorithm F that $\tilde{f}$ is represented as a Chebyshev series so it's an easy step to compute the Chebyshev series representations for $\tilde{r}_l$ and $\tilde{r}_r$. Thus, Step M4 can be accomplished by applying Boyd's root finding algorithm [4].

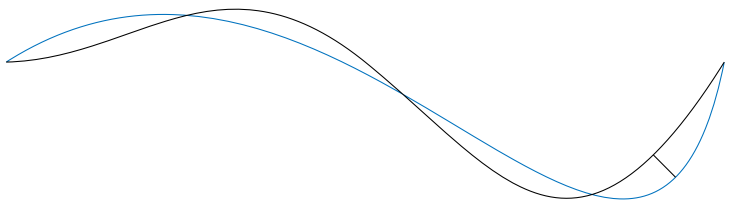
The figure below shows the result of the max distance algorithm when trying to fit the function

$$f(t) = (.1 + t)\sin t, \qquad 0 \le t \le 2\pi$$

with only a single Bezier curve segment,

```
\draw (0.000, 1.800)..controls (4.484, 4.660) and (8.448, -3.442)..(9.500, 1.800);
```

We can see that even though there are multiple local optima, Algorithm M correctly identifies the global optimum.



5. How to Produce SVG Images

In this section, I'll show how to produce an SVG image from the TikZ path that `bbai` generates. I'll also show how we can draw axes and annotate. One way to produce SVG images is embed the TikZ drawing commands into a latex document, run `lualatex` with the `--output-format=dvi` option, then use `dvisvgm` to convert the `dvi` file to an SVG image, as described in the TikZ manual (see §10.2.4 of [7]). However, if the goal is to produce an SVG graphic, I find it easier to to use the MetaPost tool which can output to SVG directly [8].

MetaPost provides a picture drawing language. Using its command line utility `mpost`, which should be included as part of a TeX Live installation, we can quickly produce an SVG plot. It accepts the same command for drawing paths as TikZ. Here, for instance, is how we would produce an SVG image for the plot from §1.

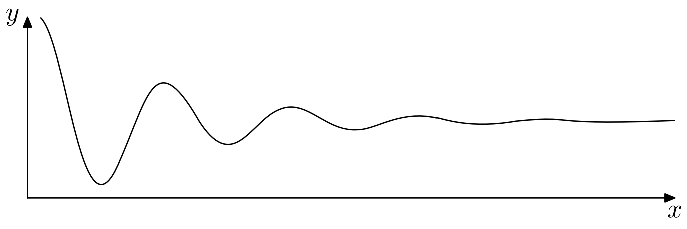
```
% plt.mp
outputformat := "svg";
outputtemplate := "%j-%c.svg";
prologues:=3;
beginfig(1);
  \draw (0.00000, 2.50000) ..  controls (0.39088, 2.10882) and (0.61233, -1.05415) ..
    (1.18750, 0.36510) ..  controls (1.60393, 1.31193) and (1.71252, 2.10800) ..
    (2.37500, 0.95127) ..  controls (2.88922, 0.15365) and (3.15477, 0.95162) ..
    (3.56250, 1.11921) ..  controls (3.98299, 1.31658) and (4.26123, 0.78781) ..
    (4.75000, 0.82415) ..  controls (5.02221, 0.81871) and (5.38203, 1.12388) ..
    (5.93750, 0.99939) ..  controls (5.96426, 1.01981) and (6.36975, 0.82102) ..
    (7.12500, 0.95127) ..  controls (8.08129, 1.04760) and (7.44859, 0.87986) ..
    (9.50000, 0.96171);
endfig;
end.
```

Running the command

```
    mpost plt.mp
```

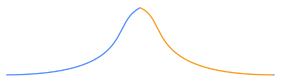
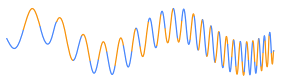
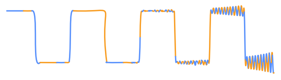
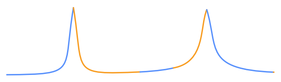
will produce an SVG for the path as `plt-1.svg`. We can use the MetaPost commands `drawarrow` and `label` to add some axes. Here is source code after rescaling and adding the axes:

```
% plt.mp
outputformat := "svg";
outputtemplate := "%j-%c.svg";
prologues:=3;
beginfig(1);
  path pth;
  pth := (0.00000, 2.50000) ..  controls (0.39088, 2.10882) and (0.61233, -1.05415) ..
    (1.18750, 0.36510) ..  controls (1.60393, 1.31193) and (1.71252, 2.10800) ..
    (2.37500, 0.95127) ..  controls (2.88922, 0.15365) and (3.15477, 0.95162) ..
    (3.56250, 1.11921) ..  controls (3.98299, 1.31658) and (4.26123, 0.78781) ..
    (4.75000, 0.82415) ..  controls (5.02221, 0.81871) and (5.38203, 1.12388) ..
    (5.93750, 0.99939) ..  controls (5.96426, 1.01981) and (6.36975, 0.82102) ..
    (7.12500, 0.95127) ..  controls (8.08129, 1.04760) and (7.44859, 0.87986) ..
    (9.50000, 0.96171);
  draw pth scaled 50;
  numeric xlim, ylim;
  xlim := xpart urcorner currentpicture;
  ylim := ypart urcorner currentpicture;
  drawarrow (-10, -10) -- (xlim, -10);
  drawarrow (-10,-10) -- (-10, ylim);
  label.bot(btex $x$ etex, (xlim, -10));
  label.lft(btex $y$ etex, (-10, ylim));
endfig;
end.
```



6. Benchmarks

In this sections I measure how long it takes me to compute Bezier paths for various functions. I didn't spend a lot of time optimizing the algorithm's implementation so I'm sure that these numbers could be improved substantially. The main takeaway should be that the algorithm is at least fast enough to be practical for many common cases. The examples are all taken from [2], and all the functions are fit over the range $-1 \le t \le 1$.

Function	Num Segments	Elapse (sec)	
$1/(1+25t^2)$	2	0.028	
$\sin(6t) + \sin(60 \exp t)$	48	0.559	
$\tanh(20 \sin 12t) + 0.02 \exp(3t) \sin(300t)$	87	1.572	
$1/(1+1000(t+.5)^2) + 1/\sqrt{1+1000(t-.5)^2}$	5	0.103	

6. Conclusions

For analytic or functions that are differentiable to a high degree, we've seen that Algorithm F provides an efficient and practical way to generate a minimal representation as a Bezier path, which can then be used to produce a vector graphic.

One area of future work might be to extend the algorithm to work better for functions with kinks (i.e. points where the function is not analytic).

References

[1] Alvin Penner. Fitting a cubic Bézier to a parametric function. *The College Mathematics Journal,* 50(3): 185–196, 2019.

[2] Lloyd N. Trefethen. Approximation theory and approximation practice. *SIAM*, 2020.

[3] C. A. Hall. On error bounds for spline interpolation. *Journal of Approximation Theory,* 1: 209–218, 1968.

[4] John Boyd. Computing zeros on a real interval through Chebyshev expansion and polynomial rootfinding. *SIAM Journal on Numerical Analysis,* 40(5): 1666–1682, 2003.

[5] Jared Aurentz, Lloyd N. Trefethen. Chopping a Chebyshev series. *ACM Transactions on Mathematical Software,* 43(4): 1–21, 2017.

[6] Jorge Nocedal, Stephen J. Wright. Numerical optimization, second edition. *Springer*, 2000.

[7] The TikZ and PGF Packages, 2026. `https://tikz.dev`.

[8] John D. Hobby, Metapost, 2024. `https://www.tug.org/docs/metapost/mpman.pdf`.