MAT 325 Numerical Analysis

Spring 2022

Sampling and Parameterizing Implicit Curves

Nathaniel Jones

Part A

Introduction

Many applications require parametric curves. For example, it is common to integrate a multivariate function along the boundary of a region. In order to integrate, this boundary must be parameterized. However, many curves are defined implicitly, and it is often difficult to obtain a parametric definition of the curve. However, if one has a good set of samples of the implicit curve, it is possible to approximate the parametric representation. We employ two interpolation methods, Lagrange interpolating polynomials and cubic splines to approximate parametric representations of the zero level curves of signed distance functions and compare the performance of these methods at different sampling densities.

Formulation

A parametric curve $P : \mathbb{R} \to \mathbb{R}^2$ is a continuous function that defines a path in 2D space. If $f(\mathbf{x}) = 0$ defines an implicit curve where $\mathbf{x}$ is a vector, the goal is to define P such that $|f(P(t))| < \epsilon$ for $t \in [a, b]$, for some $a, b \in \mathbb{R}$. Let S be an ordered list of sample points in $\mathbb{R}^2$ of an implicit curve, and the kth item in S be represented by $S(k)$. We can define P as an interpolation between the points in S. Specifically, we consider P to be a vector whose components are scalar functions of t:

$$P = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}$$

Let S_x and S_y be ordered lists of the x and y components of the points in S, respectively. We can construct $x(t)$ and $y(t)$ as interpolating functions of these x and y components and then combine them into a single vector to define P. The interpolating functions we use are cubic splines and Lagrange polynomials. In both cases, we consider our interpolating nodes for $x(t)$ to be $(0, S_x(0)), (1, S_x(1)), ..., (n, S_x(n))$. We consider $S_x(n) = S_x(0)$ to make the curve closed.

A cubic spline is constructed by defining a set of $n - 1$ cubic polynomials $x_{0..n-1}(t)$ such that

$$\begin{aligned} &x_k(k) = S_x(k) \\ &x'_k(k+1) = x'_{k+1}(k+1) \\ &x''_k(k+1) = x''_{k+1}(k+1) \\ &x'_0(0) = x''_0(0) = x'_{n-1}(n) = x''_{n-1}(n) = 0 \end{aligned}$$

We then define $x(t) = x_k(t)$ for $t \in [k, k+1]$. These conditions define a linear system of equations which can then be solved to obtain coefficients for each $x_k(t) = a_k t^3 + b_k t^2 + c_k t + d_k$. We use the built-in MATLAB cubic spline interpolation to create the spline. $y(t)$ can be constructed in a similar manner with nodes coming from S_y.

A Lagrange interpolating polynomial is constructed by the sum

$$x(t) = \sum_{k=0}^{n} S_x(k) L_{n,k}(t)$$

where

$$L_{n,k}(t) = \prod_{\substack{i=0 \\ i \neq k}}^{n} \frac{t-i}{k-i}$$

$y(t)$ can again be constructed with the same method using nodes from S_y.

We construct parametric curves using samples generated from signed distance functions. Let $D : \mathbb{R}^2 \to \mathbb{R}$ be a signed distance function. We can compute the average distance of a given parametric curve to the implicit curve defined by $D(x, y) = 0$ by

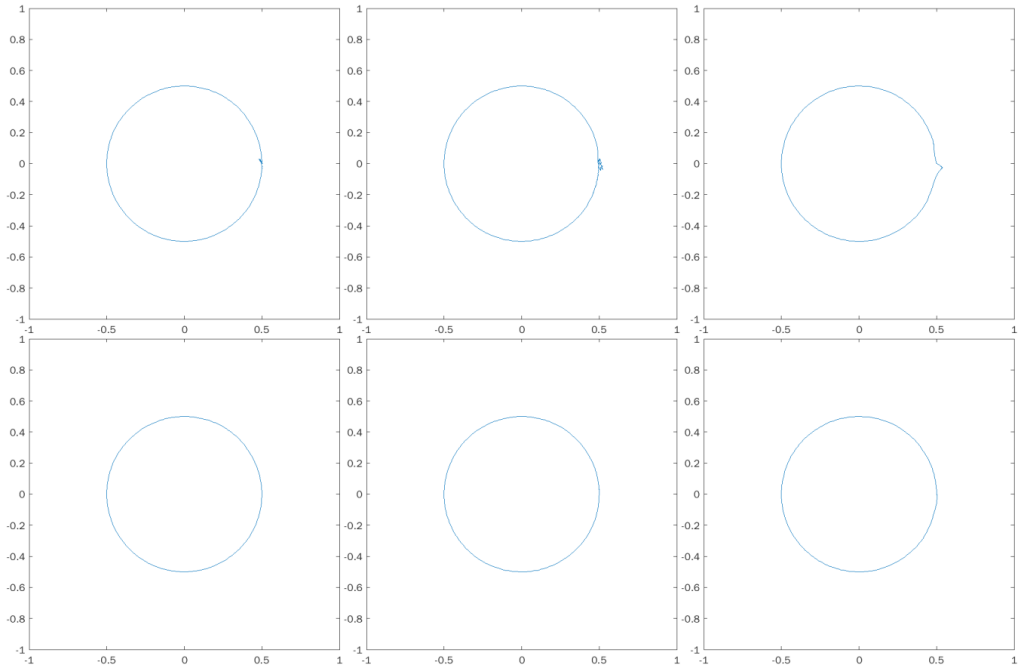
$$\frac{1}{n} \int_0^n |D(P(t))| dt$$

The smaller the average distance, the better the accuracy of the parametric curve.

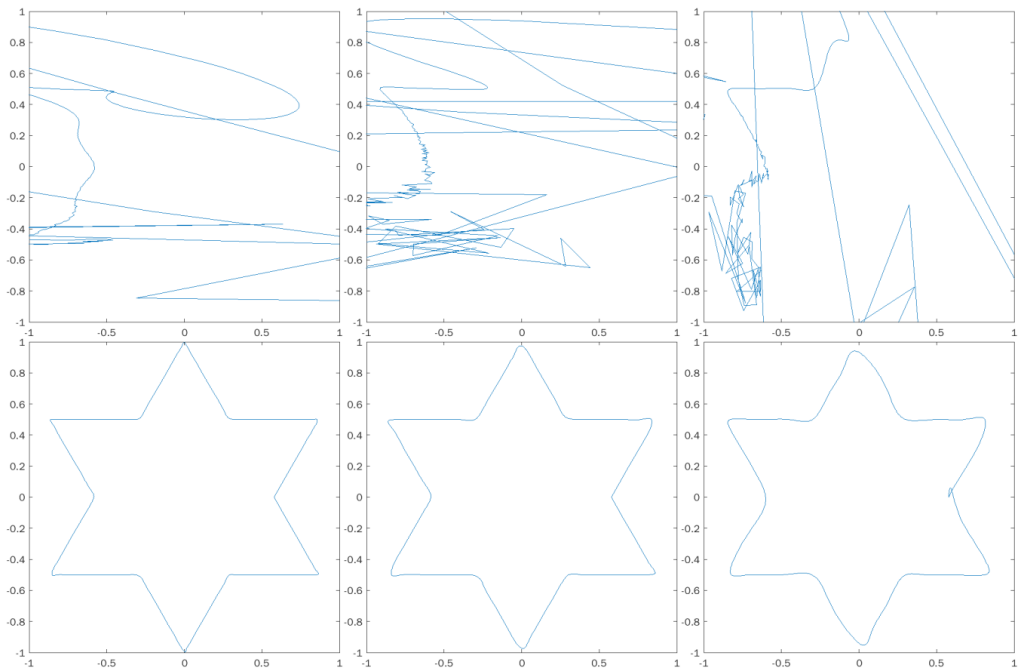
Results

Table 1: Average Euclidean Distance from Curve at Different Sample Densities

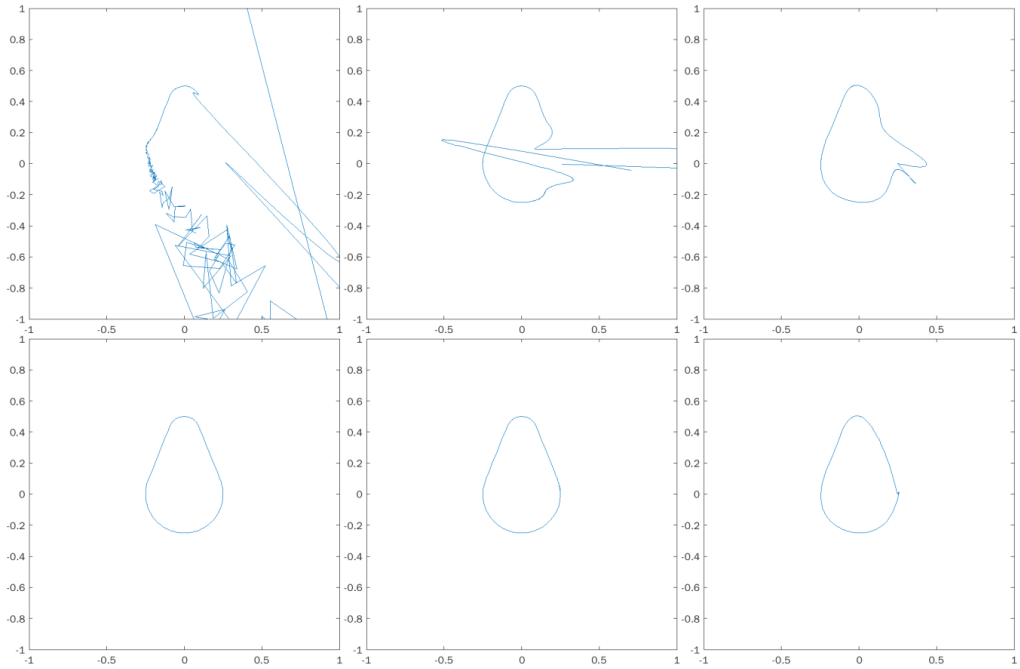
Shape	$s = 0.05$	$s = 0.1$	$s = 0.2$
Circle (Cubic)	4.7020e–05	6.9252e–04	0.0013
Circle (Lagrange)	0.0138	0.0692	0.0595
Hexagram (Cubic)	3.8315e–04	0.0013	0.0072
Hexagram (Lagrange)	2.2412e+19	2.6625e+10	6.2753e+06
Uneven Capsule (Cubic)	4.1746e–04	0.0031	0.0046
Uneven Capsule (Lagrange)	6.0158e+05	26.4659	0.3503



Comparison of methods for parameterizing a circle. Top: Lagrange Polynomial, Bottom: Cubic Spline. Left: $s = 0.05$, Middle: $s = 0.1$, Right: $s = 0.2$



Comparison of methods for parameterizing a hexagram. Top: Lagrange Polynomial, Bottom: Cubic Spline. Left: $s = 0.05$, Middle: $s = 0.1$, Right: $s = 0.2$



Comparison of methods for parameterizing an uneven capsule . Top: Lagrange Polynomial, Bottom: Cubic Spline. Left: $s = 0.05$, Middle: $s = 0.1$, Right: $s = 0.2$

In the tables and figures above, s refers to the approximate distance between consecutive samples. The samples were generated using the method described in Part B, using the 5-point midpoint formula to compute the gradients. It is apparent that cubic splines perform very well, particularly at high sample densities. On the other hand, Lagrange polynomials have very poor performance, particularly on the hexagram test. This makes sense due to the extreme oscillation in high-degree Lagrange polynomials, which is also known as the Runge phenomenon. Furthermore, a hexagram is a very pointy shape, for lack of a better term, which means it is difficult to approximate with smooth functions such as polynomials. Despite this difficulty, cubic splines are able to approximate the hexagram with very little error. Lagrange polynomials do, however, provide a decent approximation in the circle test, although not as good as cubic splines.

Conclusion

On the whole, cubic splines performed much better than Lagrange polynomials at all sampling densities. This is not unexpected, as the Runge phenomenon becomes increasingly prominent in Lagrange polynomials as the number of points increases. Interestingly, the Lagrange polynomials were able to approximate a circle reasonably well even at high sampling densities. It is worth investigating why this is the case. It is also worth investigating whether the performance of Lagrange polynomials could be improved by using, for example, arc-length parameterization. However, since

cubic splines do not suffer from the Runge phenomenon, and provide great accuracy when given good samples, they should be preferred in this application.

Part B

Introduction

Generating samples of implicit curves is an important step of many numerical procedures, such as boundary parameterization and finite element methods. These samples should ideally be evenly spaced along the curve with sampling density able to be determined by the user as appropriate for the application. While there are many approaches to sampling an implicit curve, one of the simplest is to frame the problem as a root-finding problem. This framing allows us to take advantage of the wealth of existing root finding methods in solving this problem. A suitable root-finding method is a generalized version of the Newton-Raphson method designed to find the roots of a multivariate scalar-valued function. This method requires that the gradient of the function that defines the curve is computed accurately in each iteration. While we would ideally have a closed-form expressions for each partial derivative of a given function, often the function that defines an implicit curve is complex and closed-form derivatives are difficult to obtain. Therefore, we employ three numerical methods to approximate the partial derivatives of the function and compare how they affect the quality of the generated samples.

Formulation

An implicit curve is defined as the set of points for which $f(x, y) = 0$ for some continuous function $f : \mathbb{R}^2 \to \mathbb{R}$. In this paper, we restrict our focus to closed, continuous implicit curves, particularly those defined by signed distance functions. The Newton-Raphson method of finding the roots of a real-valued function of a single variable is well known. Given an initial guess for a root x_0, perform the following iteration to approximate a nearby root:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

While this method generally works well for functions of a single variable, it is also possible to modify it to find the roots of a multivariate scalar-valued function as demonstrated by Meyer (2008). Let $\mathbf{x}$ be a vector, $f : \mathbb{R}^2 \to \mathbb{R}$, and let ∇f be defined at all points in the domain. Given an initial guess $\mathbf{x}_0$, the roots of f can be approximated by

$$\mathbf{x}_{i+1} = \mathbf{x}_i - f(\mathbf{x}_i)\frac{\nabla f(\mathbf{x}_i)}{\|\nabla f(\mathbf{x}_i)\|^2}$$

We use this iteration to generate the samples of the implicit curve. To improve the utility of our samples, we would like to generate them in counter-clockwise order around the curve rather than to generate them in a random order. To accomplish this, we first obtain a point $\mathbf{p}_0$ on the curve by the above generalized Newton-Raphson method. We then create a normalized vector that is perpendicular to $\nabla f(\mathbf{p}_0)$ by computing

$$\mathbf{t} = \frac{1}{\|\nabla f(\mathbf{p}_0)\|}\begin{bmatrix} -\frac{\partial f}{\partial y}(\mathbf{p}_0) \\ \frac{\partial f}{\partial x}(\mathbf{p}_0) \end{bmatrix}$$

and then computing $g = \mathbf{p}_0 + s \cdot \mathbf{t}$, where s is a user-defined step-size parameter that controls sampling density. We then iterate the generalized Newton-Raphson method with g as the initial guess until we obtain a new point on the surface $\mathbf{p}_1$. We repeat this process to generate $\mathbf{p}_2, \mathbf{p}_3$, and so on, until the distance from $\mathbf{p}_0$ of our generated sample is less than $\frac{1}{2}s$.

As stated in the introduction, we use numeric differentiation to approximate the gradient, as many implicit curves are generated from complicated functions that do not have simple expressions for their partial derivatives. We compare the performance of the forward difference, the central difference, and the five-point midpoint methods, which are given respectively here:

$$\frac{\partial f}{\partial x}(\mathbf{x}) \approx \frac{f(\mathbf{x}+h\hat{\mathbf{x}}) - f(\mathbf{x})}{h}$$
$$\frac{\partial f}{\partial x}(\mathbf{x}) \approx \frac{f(\mathbf{x}+h\hat{\mathbf{x}}) - f(\mathbf{x}-h\hat{\mathbf{x}})}{2h}$$
$$\frac{\partial f}{\partial x}(\mathbf{x}) \approx \frac{f(\mathbf{x}-2h\hat{\mathbf{x}}) - 8f(\mathbf{x}-h\hat{\mathbf{x}}) + 8f(\mathbf{x}+h\hat{\mathbf{x}}) - f(\mathbf{x}+2h\hat{\mathbf{x}})}{12h}$$

where h represents mesh size. In the experiments, the mesh size and the step size are the same value.

Results

Table 1. Results for Circle

Gradient Method	Step Size	Mean Distance Between Samples	Standard Deviation
Forward	0.05	0.0491	0.0062

Gradient Method	Step Size	Mean Distance Between Samples	Standard Deviation
Central	0.05	0.0491	0.0060
5-Point	0.05	0.0491	0.0060
Forward	0.1	0.0966	0.0127
Central	0.1	0.0961	0.0142
5-Point	0.1	0.0961	0.0142
Forward	0.2	0.1837	0.0297
Central	0.2	0.1837	0.0220
5-Point	0.2	0.1837	0.0221

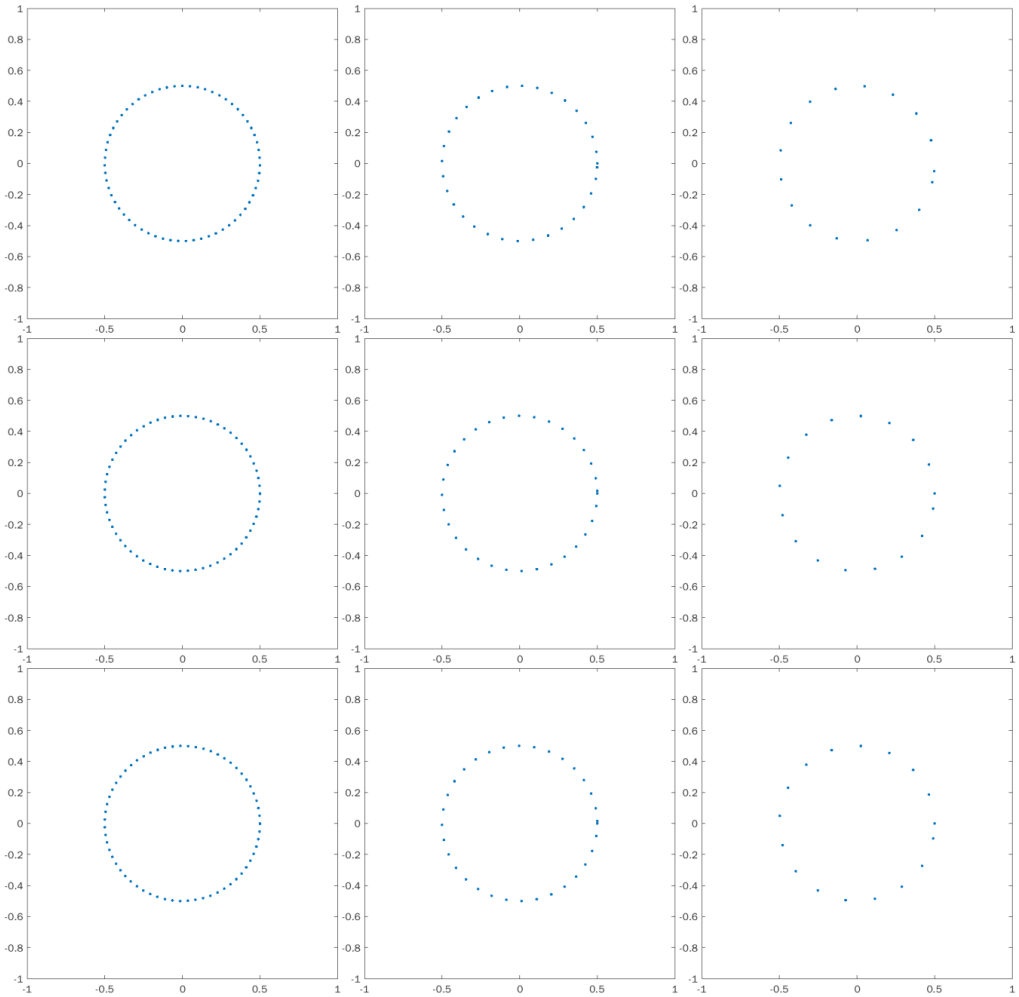
Table 2. Results for Hexagram

Gradient Method	Step Size	Mean Distance Between Samples	Standard Deviation
Forward	0.05	0.0558	0.1111
Central	0.05	0.0481	0.0057
5-Point	0.05	0.0479	0.0066
Forward	0.1	0.0985	0.0378
Central	0.1	0.0930	0.0166
5-Point	0.1	0.0918	0.0152
Forward	0.2	0.2011	0.1514
Central	0.2	0.1727	0.0419
5-Point	0.2	0.1677	0.0390

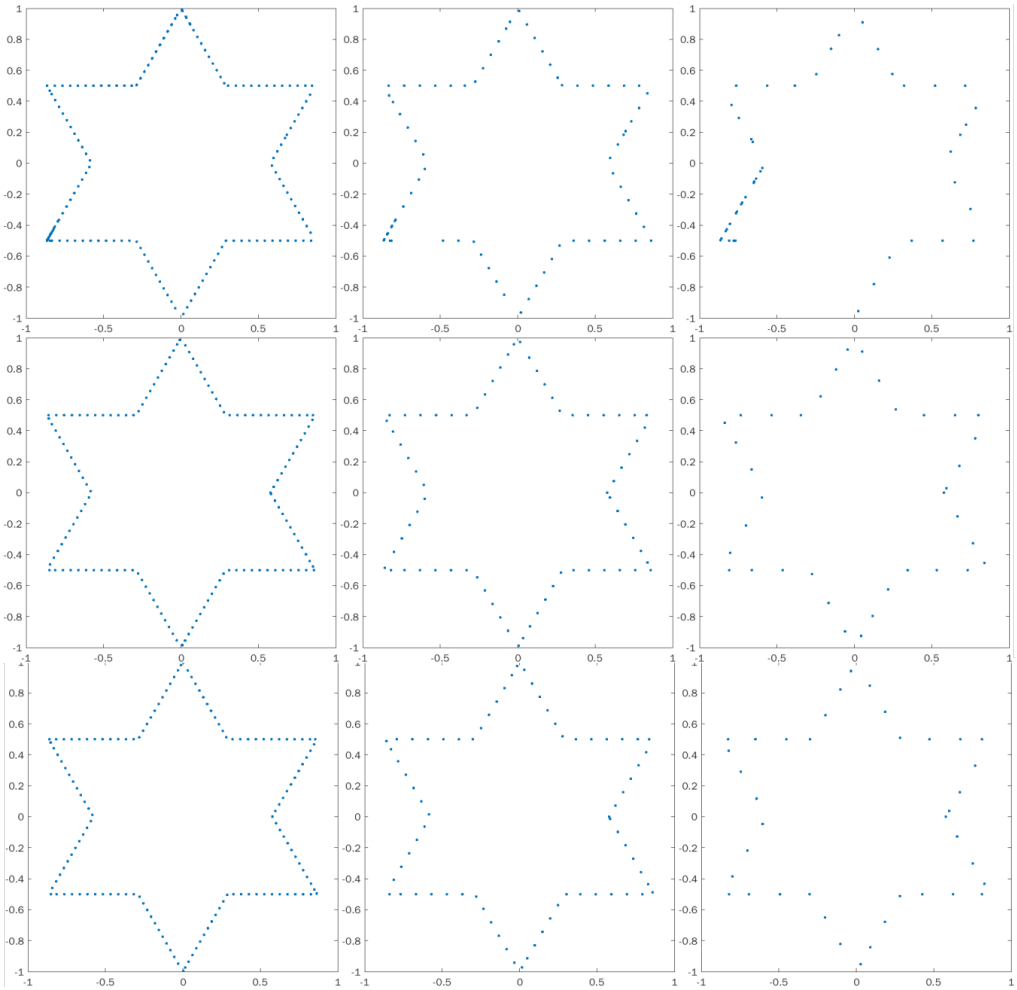
Table 3. Results for Uneven Capsule

Gradient Method	Step Size	Mean Distance Between Samples	Standard Deviation
Forward	0.05	0.0485	0.0051
Central	0.05	0.0484	0.0054
5-Point	0.05	0.0483	0.0056
Forward	0.1	0.0926	0.0138
Central	0.1	0.0926	0.0118
5-Point	0.1	0.0919	0.0135
Forward	0.2	0.1588	0.0530

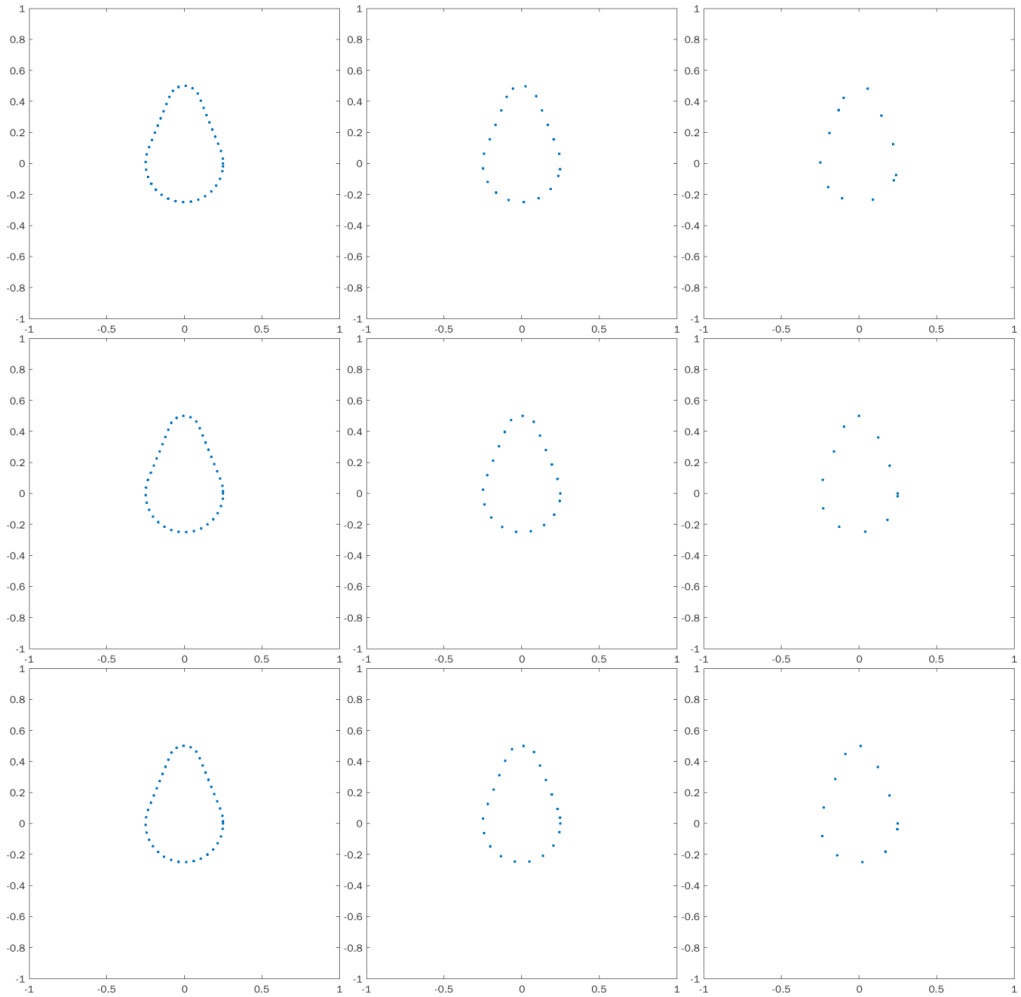
Gradient Method	**Step Size**	**Mean Distance Between Samples**	**Standard Deviation**
Central	0.2	0.1596	0.0495
5-Point	0.2	0.1600	0.0450



Comparison of methods for sampling a circle. Top: Forward Difference, Middle: Central Difference: Bottom: 5-Point Midpoint Difference. Left: $s = 0.05$, Middle: $s = 0.1$, Right: $s = 0.2$



Comparison of methods for sampling a hexagram. Top: Forward Difference, Middle: Central Difference: Bottom: 5-Point Midpoint Difference. Left: $s = 0.05$, Middle: $s = 0.1$, Right: $s = 0.2$



Comparison of methods for sampling an uneven capsule. Top: Forward Difference, Middle: Central Difference: Bottom: 5-Point Midpoint Difference. Left: $s = 0.05$, Middle: $s = 0.1$, Right: $s = 0.2$

The above tables show quantitatively how the method of numeric differentiation affects sample quality. In general, there is not much difference between each method, although forward difference tends to have more variation in distance between consecutive samples than either central difference or the 5-point midpoint method. In particular, forward difference has poor performance on the hexagram shape, as can be seen in the figures. The mean distance between consecutive samples seems to stay below the step size value, indicating that step size acts more as an upper bound than a parameter to directly control sample density. The samples do appear to be reasonably uniformly spaced, however, indicating that the presented method is suitable for sampling an implicit curve.

Conclusion

The five-point midpoint method and the central difference method performed well for sampling the curves and generated very qualitatively similar results. The forward

difference method performed decently when sampling the circle and the uneven capsule, but was poor for sampling the hexagram. This makes sense, as the forward difference is the least accurate of the three numeric differentiation methods, and the error accumulated as the generalized Newton-Raphson method was iterated. While the five-point midpoint method is theoretically more accurate than the central difference method, it appears that this increased accuracy did not matter for our application. The functions we tested were fairly smooth, however, so it would be worth testing noisy or high-frequency functions in the future. Regarding the Newton-Raphson method itself as a method of generating samples, it performs reasonably well, especially given its simplicity. However, pairing it with a relaxation step to ensure that samples are roughly equidistant from one another would likely improve sampling quality. Still, the method seems appropriate for applications without strict requirements on sample uniformity.

Bibliography

Meyer, M. (2008). *Dynamic particles for adaptive sampling of implicit surfaces.* (Doctoral dissertation, School of Computing, University of Utah).

Quilez, I. (n.d.). *2D SDFs.* Iquilezles.Org. Retrieved May 11, 2022, from https://iquilezles.org/articles/distfunctions2d/

MATLAB Code (Part A and Part B)

```
STEP_SIZE  = step_size();
MAX_POINTS = 20 / step_size();

% Set this value to whichever signed distance function you want to test
sdf = @(x) (sd_hexgram(x));

x0 = move_to_surface([1.0, 0.0], sdf);
samples = {x0};

while length(samples) < 2 ...
    || norm(samples{end} - samples{1}) > STEP_SIZE / 2 ...
    && length(samples) < MAX_POINTS
        s = move_to_surface(counter_clockwise(samples{end}, sdf), sdf);
        samples{end + 1} = s;
end

% Make the curve closed by appending the first point to the end of the
% samples
samples{end + 1} = samples{1};

X = 1:length(samples);
Y = 1:length(samples);

% Create S_x and S_y
for i = 1:length(samples)
    X(i) = samples{i}(1);
    Y(i) = samples{i}(2);
end

a = 0;
b = length(samples) - 1;

t_nodes = a:1:b;
v = a:STEP_SIZE:b;

N = length(t_nodes) - 1;
xlagrange_poly = polyfit(t_nodes, X, N);
ylagrange_poly = polyfit(t_nodes, Y, N);

xlagrange = @(t) (polyval(xlagrange_poly, t));
ylagrange = @(t) (polyval(ylagrange_poly, t));

xcubic = @(t) (interp1(t_nodes, X, t, "spline"));
ycubic = @(t) (interp1(t_nodes, Y, t, "spline"));

% Compute distances between consecutive samples
distances = [];
```

```
for i = 1:length(samples) - 1
    distances(end + 1) = norm(samples{i + 1} - samples{i});
end


fplot(xcubic, ycubic, [0, N], '-');
xlim([-1, 1]);
ylim([-1, 1]);
pbaspect([1 1 1]);
figure();

fplot(xlagrange, ylagrange, [0, N], '-');
xlim([-1, 1]);
ylim([-1, 1]);
pbaspect([1 1 1]);
figure();

plot(X, Y, '.');
xlim([-1, 1]);
ylim([-1, 1]);
pbaspect([1 1 1]);

disp("Mean distance");
display(mean(distances));
disp("Standard deviation");
display(std(distances));

dist_from_curve_spline = @(t) (abs(sdf([xcubic(t), ycubic(t)])));
dist_from_curve_lagrange = @(t) (abs(sdf([xlagrange(t), ylagrange(t)])));

avg_error_spline = integral(dist_from_curve_spline, 0, length(samples), ...
    ArrayValued=true) / length(samples);

avg_error_lagrange = integral(dist_from_curve_lagrange, 0, length(samples), ...
    ArrayValued=true) / length(samples);

disp("Average distance from curve (spline):");
disp(avg_error_spline);

disp("Average distance from curve (Lagrange):");
disp(avg_error_lagrange);

% Move the given point onto the level curve f(x,y) = 0
function p = move_to_surface(point, f)
    p = point;
    tolerance = 0.0001;

    % Iterate the modified version of Newton's method to find a close point
    % on the curve
```

```
    while abs(f(p)) > tolerance
        gr = grad(p, f);
        p  = p - f(p) * gr / norm(gr)^2;
    end
end

% Move approximately counter-clockwise along the level curve
% by moving along the line tangent to it
function p = counter_clockwise(point, f)
    g = grad(point, f);
    n = g / norm(g);
    tangent = [-n(2), n(1)];

    p = point + step_size() * tangent;
end

% Compute the gradient of the given function handle at the given point
% using the 5 point midpoint formula
function g = grad_5point(point, f)
    h = mesh_size();
    dx = [h, 0];
    dy = [0, h];

    dfdx = (f(point - 2*dx) ...
        - 8*f(point - dx) ...
        + 8*f(point + dx) ...
        - f(point + 2*dx) ...
        )/(12 * h);

    dfdy = (f(point - 2*dy) ...
        - 8*f(point - dy) ...
        + 8*f(point + dy) ...
        - f(point + 2*dy) ...
        )/(12 * h);

    g = [dfdx, dfdy];
end

% Compute the gradient of the given function handle at the given point
% using the central difference method
function g = grad_midpoint(point, f)
    h = mesh_size();
    dx = [h, 0];
    dy = [0, h];

    dfdx = (f(point + dx) - f(point - dx))/(2 * h);
    dfdy = (f(point + dy) - f(point - dy))/(2 * h);

    g = [dfdx, dfdy];
```

```
end

% Compute the gradient of the given function handle at the given point
% using the forward difference method
function g = grad_forward(point, f)
    h = mesh_size();
    dx = [h, 0];
    dy = [0, h];

    dfdx = (f(point + dx) - f(point))/h;
    dfdy = (f(point + dy) - f(point))/h;

    g = [dfdx, dfdy];
end

% Helper function to use a specific gradient function for all computations
function g = grad(point, f)
    g = grad_5point(point, f);
end

function y = clamp(x, a, b)
    y = min(max(x, a), b);
end

% Helper function to use a specific step size for all computations
function step_size = step_size()
    step_size = 0.2;
end

% Helper function to use a specific mesh size for all gradients
function mesh_size = mesh_size()
    mesh_size = 0.2;
end

% Credit goes to Inigo Quilez for designing these distance functions.
% https://iquilezles.org/articles/distfunctions2d/
function d = sd_hexgram(point)
    r = 0.5;
    k = [-0.5, 0.866025403, 0.5773502692, 1.7320508076];
    p = [abs(point(1)), abs(point(2))];
    p = p - (2.0 * min(dot([k(1), k(2)], p), 0.0) * [k(1), k(2)]);
    p = p - (2.0 * min(dot([k(2), k(1)], p), 0.0) * [k(2), k(1)]);
    p = p - [clamp(p(1), r*k(3), r*k(4)), r];

    d = norm(p) * sign(p(2));
end

function d = sd_circle(point)
    r = 0.5;
```

```
    d = norm(point) - r;
end

function d = sd_uneven_capsule(point)
    r1 = 0.25;
    r2 = 0.1;
    h  = 0.4;

    p = point;
    p(1) = abs(p(1));
    b = (r1 - r2)/h;
    a = sqrt(1.0 - b*b);
    k = dot(p, [-b, a]);
    if k < 0.0
        d = norm(p) - r1;
    elseif k > a*h
        d = norm(p - [0.0, h]) - r2;
    else
        d = dot(p, [a, b]) - r1;
    end
end
```