MATH 371, FA 2019

Numerical Methods

Project: Curvature Flow

Student Information

Name Ziqiao Ma (Martin)

UMID 71224904

Contact

Email marstin@umich.edu

November 14, 2019

Electrical Engineering and Computer Science

University of Michigan

Ann Arbor, MI

Contents

List of Tables

1 Project Introduction

Many Scientific studies and engineering projects involve **curvature flow**.

We would say that a family of surfaces evolves under (mean) curvature flow, if the normal component of the velocity of which a point on the surface moves is given by the mean curvature of the surface [1]. Usually, the process is numerically indirect, by introducing successive evolutions to curve surfaces. The purpose of curvature flow is to smooth, or eliminate noise on the curve.

In our project, the process starts with a predefined initial 2D-curve. By using the **Euler Method** and differentiation using **Fast Fourier Transform** and **Inverse Fast Fourier Transform**, we de-noise the surface of the curve as time-step increases. Eventually, a smooth curve, *i.e.* a circle in this case, is expected.

Curvature flow serves as a basic model for several phenomena such as evolution of fluid interfaces in droplet/bubble flows and grain coarsening in materials. It is also used as a tool for solving problems in engineering such as image segmentation [2].

2 Theoretic Background

2.1 Differentiation with FFT and IFFT

Using FFT and IFFT, we can evaluate differentiation. Given a function f, we denote $g = f'$.

First, compute $\hat{f}$ using FFT and Trapezoidal rule:

$$\{\hat{f}\}_{j=-M}^{M-1} = \frac{1}{2\pi}\int_o^{2\pi} f(x)e^{-ijx}\,dx$$

Then, set $\hat{g_k} = (ik)\hat{f_k}$, for all $k = -M, \cdots, M-1$.

Since $f(x) = \sum_{k=-M}^{M-1} \hat{f}_k e^{ikx}$,

$$\begin{aligned} g(x) = f'(x) &= \sum_{k=-M}^{M-1} \hat{f}_k (ik) e^{ikx} \\ &= \sum_{k=-M}^{M-1} \hat{g_k} e^{ikx} \end{aligned}$$

Based on this, compute $g(x) = \sum_{k=-M}^{M-1} \hat{g_k} e^{ikx}$ using IFFT.

The complexity of this method is theoretically $O(M \log M)$, where we discretize $[0, 2\pi)$ into $2M$ pieces.

2.2 Parametrized Curve

Defined on a finite-dimensional vector space V and an interval $I \subset \mathbb{R}$, a curve $\mathcal{C} \subset V$ is parametrized by a continuous, surjective, and locally injective map $\gamma : I \to \mathcal{C}$.

In this project, we consider a closed curve that is evolving in time. whose coordinates $\mathbf{x}$ are parametrized by α. It could be represented in the following form:

$$\mathbf{x}(\alpha, 0) = \begin{bmatrix} x_1(\alpha, t) \\ x_2(\alpha, t) \end{bmatrix}, \text{ where } \mathbf{x}(\alpha, 0) = (4 + \cos 3\alpha) \begin{bmatrix} \cos\alpha \\ \sin\alpha \end{bmatrix}$$

where $\alpha \in I = [0, 2\pi)$, and t denotes time, represented using time-steps.

We use notations as follows:

$$x_{i\alpha} = \frac{\partial x_i}{\partial \alpha}, \ i = 1, 2$$

$$|\mathbf{x}_\alpha| = \sqrt{x_{1\alpha}^2 + x_{2\alpha}^2}$$

The curvature c of a curve is given by

$$c(\alpha) = \frac{x_{2\alpha} x_{1\alpha\alpha} - x_{1\alpha} x_{2\alpha\alpha}}{|\mathbf{x}_\alpha|^3}$$

The normal vector $\mathbf{n}$ of a curve is given by

$$\mathbf{n}(\alpha) = \frac{1}{|\mathbf{x}_\alpha|} \begin{bmatrix} x_{2\alpha} \\ -x_{1\alpha} \end{bmatrix}$$

The length of the curve at time t is given as

$$L(t) = \int_0^{2\pi} |\mathbf{x}_\alpha| \, d\alpha$$

2.3 Euler Method

The governing equation for the curvature flow is given as

$$\dot{\mathbf{x}} = \frac{d\mathbf{x}}{dr} = c(\alpha)\mathbf{n}(\alpha)$$

where c is the curvature of the curve and $\mathbf{n}$ is the normal vector of the curve.

To solve the non-linear ODEs, apply Euler Method. The iterative equation is given by

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} + \Delta t \cdot \dot{\mathbf{x}}$$

where Δt is the time elapse of each time-step.

3 Matlab Implementation

The Matlab codes for the implementation of the project is as follows. Codes are also available on GitHub: https://github.com/Mars-tin/Curvature-Flow-with-Euler-Method.git

Differentiation with FFT and DFFT:

```
function g = DIFT(f, M)
        f_fft = fftshift(fft(f,M));
        g_fft = zeros(1, M);
        for k = -M/2 : M/2-1
            g_fft(k+M/2+1) = f_fft(k+M/2+1) * (1i * k);
        end
        g_fft = fftshift(g_fft);
        g = real(ifft(g_fft, M));
end
```

Computing the normal vector:

```
function n = NORM(x, M)
    x_1a = DIFT(x(1,:), M);
    x_2a = DIFT(x(2,:), M);
    x_a = sqrt(x_1a.^2 + x_2a.^2);
    n1 = x_2a ./ x_a;
    n2 = -x_1a ./ x_a;
    n = [n1; n2];
end
```

Computing the curvature:

```
function kappa = CURV(x, M)
    x_1a = DIFT(x(1,:), M);
    x_2a = DIFT(x(2,:), M);
    x_1aa = DIFT(x_1a, M);
    x_2aa = DIFT(x_2a, M);
    x_a = sqrt(x_1a.^2 + x_2a.^2);
```

```
    kappa = (x_2a .* x_1aa - x_1a .* x_2aa) ./ x_a.^3;
end
```

Computing the length:

```
function len = LENGTH(x, M)
    x_1a = DIFT(x(1,:), M);
    x_2a = DIFT(x(2,:), M);
    x_a = sqrt(x_1a.^2 + x_2a.^2);
    len = (sum(x_a) - x_a(1)/2 - x_a(M)/2) *2*pi/M;
end
```

Computing the x-dot:

```
function x_dot = XDOT(x, M)
    x_1a = DIFT(x(1,:), M);
    x_2a = DIFT(x(2,:), M);
    x_1aa = DIFT(x_1a, M);
    x_2aa = DIFT(x_2a, M);
    x_a = sqrt(x_1a.^2 + x_2a.^2);
    kappa = (x_2a .* x_1aa - x_1a .* x_2aa) ./ x_a.^3;
    n1 = x_2a ./ x_a;
    n2 = -x_1a ./ x_a;
    x_dot1 = n1 .* kappa;
    x_dot2 = n2 .* kappa;
    x_dot = [x_dot1; x_dot2];
end
```

Plot the figure at some time-step:

```
function PLOTNORM(x,n)
    plot(x(1,:), x(2,:));
    axis equal
    hold on
    quiver(x(1,:), x(2,:), n(1,:), n(2,:));
```

```
    hold off
end
```

The main driving function:

```
function CIRCULIZE(M, N, t0, tf)
    x1_ = @(a) (4+cos(3*a)).*cos(a);
    x2_ = @(a) (4+cos(3*a)).*sin(a);
    h = 2*pi/M;
    a = linspace(0, 2*pi-h, M);
    x = [x1_(a); x2_(a)];
    j = (tf - t0)/N;
    for i = 0 : N
        x_dot = XDOT(x, M);
        if rem(i, N/(tf-t0)) == 0
            PLOTNORM(x, x_dot);
            pause;
        end
        x = x + x_dot * j;
    end
end
```

Evaluate CPU time:

```
function time = CPUTIME(M, N, t0, tf)
    x1_ = @(a) (4+cos(3*a)).*cos(a);
    x2_ = @(a) (4+cos(3*a)).*sin(a);
    h = 2*pi/M;
    a = linspace(0, 2*pi-h, M);
    x = [x1_(a); x2_(a)];
    j = (tf - t0)/N;
    time = zeros(1, N+1);
    for i = 0 : N
        f = @() EULERMETHOD(x, M, j);
        time(i+1) = timeit(f);
```

```
        x = EULERMETHOD(x, M, j);
    end
end

function x = EULERMETHOD(x, M, j)
    x_dot = XDOT(x, M);
    x = x + x_dot * j;
end
```

Finding Error:

```
function err = ERROR(M, N, t0, tf)
    x1_ = @(a) (4+cos(3*a)).*cos(a);
    x2_ = @(a) (4+cos(3*a)).*sin(a);

    h = 2*pi/M;
    a = linspace(0, 2*pi-h, M);
    x = [x1_(a); x2_(a)];

    j = (tf - t0)/N;

    for i = 0 : N
        x_dot = XDOT(x, M);
        if i == N
            err = LENGTH(x, M);
        end
        x = x + x_dot * j;
    end
end
```

3 Visualization of Curve Evolution

This section is oriented to Question 1.

To simulate using $N = 100$ and $M = 32$, simulate the evolution of the curve of time $t \in [0, 4]$ until $t = 4$, we run the `Matlab` code: `CIRCULIZE(32, 100, 0, 4)`. The results are represented as follows (with normal vector on).

At $t = 0s$, *i.e.* the initial curve, the plot is given as:

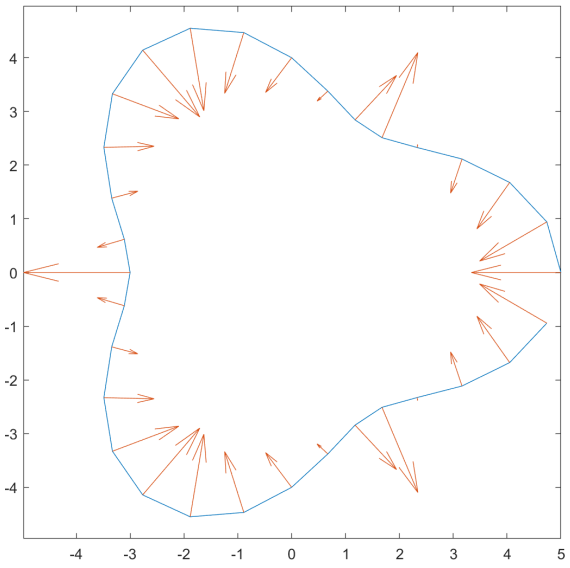

Figure 1: Evolution of the curve, at $t = 0s$

At $t = 1s$, the plot is given as:

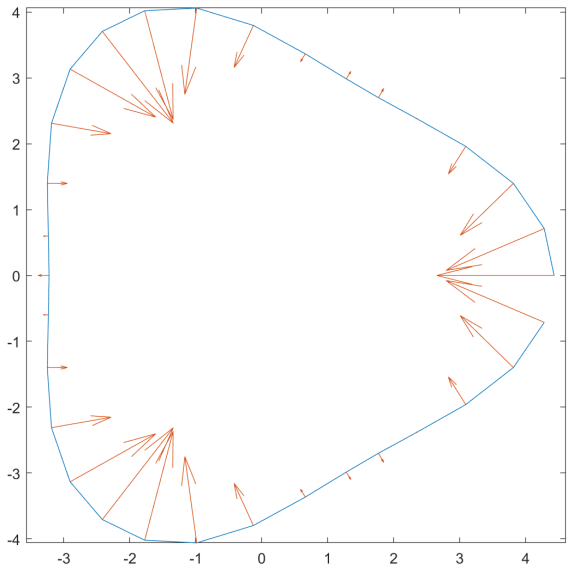

Figure 2: Evolution of the curve, at $t = 1s$

At $t = 2s$, the plot is given as:

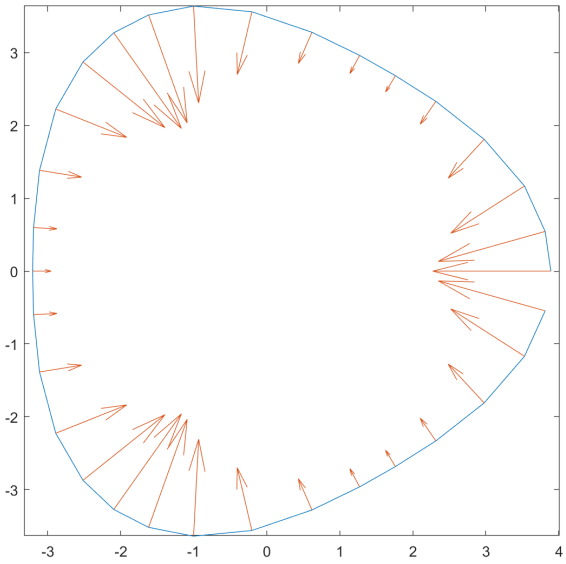

Figure 3: Evolution of the curve, at $t = 2s$

At $t = 3s$, the plot is given as:

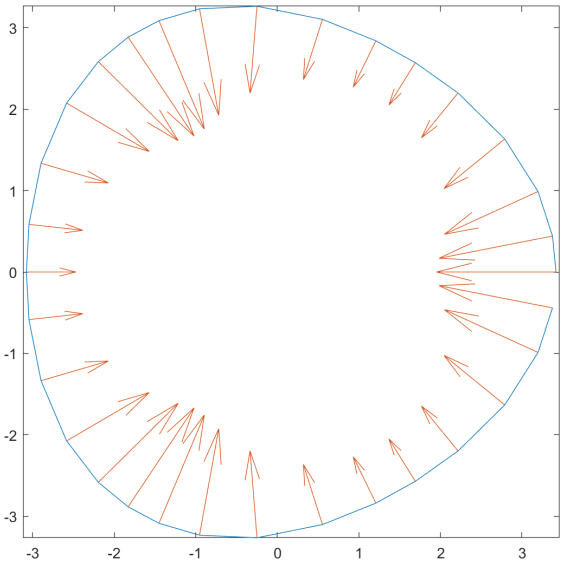

Figure 4: Evolution of the curve, at $t = 3s$

At $t = 4s$, *i.e.* the final state, the plot is given as:

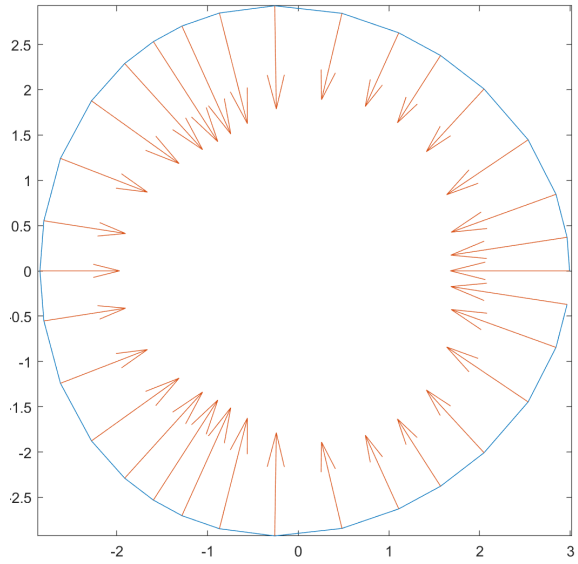

Figure 5: Evolution of the curve, at $t = 4s$

In order to show the evolution process, we sketch all these figures in the same plot:

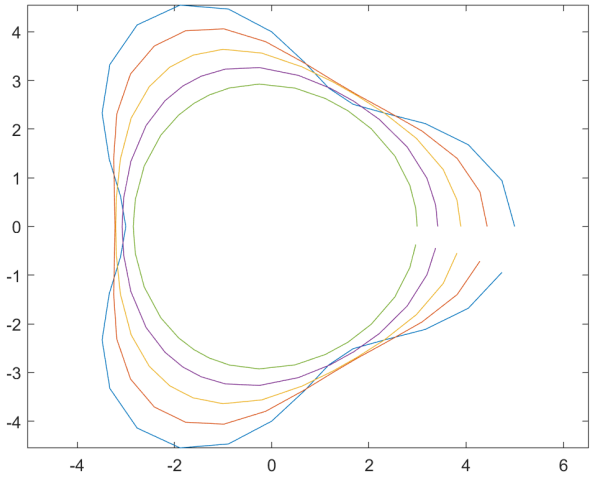

Figure 6: Evolution of the curves

From the graphs above, we observe and conclude that

1. The method is successful (*i.e.* converges) under the set of parameters, and a smooth circular curve is obtained at the end of the method (Figure 5).

2. Since we moved the points orthogonal to the curve at a speed proportional to the curvature, the method is curve shortening, *i.e.* **Euclidean shortening flow**. As the method processes, the length and area of the curve is decreasing.

4 Complexity Analysis

This section is oriented to Question 2.

In this section, we would like to analyse the time complexity of the method by measuring the CPU time per time-step.

In order to avoid other unrelated functions, like plotting a curve, from affecting the runtime, the Euler iterative equation is taken out and is measured with `timeit` every time-step. After recording all times, we calculate the CPU time per time-step by taking an average.

Still fix $N = 100$, the CPU time per time-step for $M = 16, 32, 64, 128$ and 256 are tabulated as follows, by running `CPUTIME(M, 100, 0, 4)`.

M	$\frac{\text{CPU Time}}{\text{Time-Step}}$ $[s/0.04s]$
16	3.67×10^{-5}
32	4.96×10^{-5}
64	6.14×10^{-5}
128	9.68×10^{-5}
256	1.67×10^{-4}

Table 1: CPU per time-step for different values of $M \in [16, 256]$

From the data above, it looks as if the complexity is linear, *i.e.* $O(M)$. This is because as the value of M doubles, the CPU time gets around 1.5 times larger. However, when M is small, the value of its logarithm $\log M$ is close to constant. Hence, we need to verify with much larger value of M.

M	$\frac{\text{CPU Time}}{\text{Time-Step}}$ $[s/0.04s]$	M	$\frac{\text{CPU Time}}{\text{Time-Step}}$ $[s/0.04s]$
8	3.23×10^{-4}	2048	9.68×10^{-4}
16	3.67×10^{-4}	4096	0.0020
32	4.96×10^{-4}	8192	0.0039
64	6.14×10^{-4}	16384	0.0074
128	9.68×10^{-4}	32768	0.0155
256	1.67×10^{-4}	65536	0.0354
512	2.95×10^{-4}	131072	0.0778
1024	5.28×10^{-4}	262144	0.1654

Table 2: CPU per time-step for different values of $M \in [8, 262144]$

In order to see whether linear model or loglinear model fits the above data more, I set up two statistical models `y=a*x+b` and `y=a*x*ln(x+b)`, and fit the data with `OriginLab`. The results are as follows.

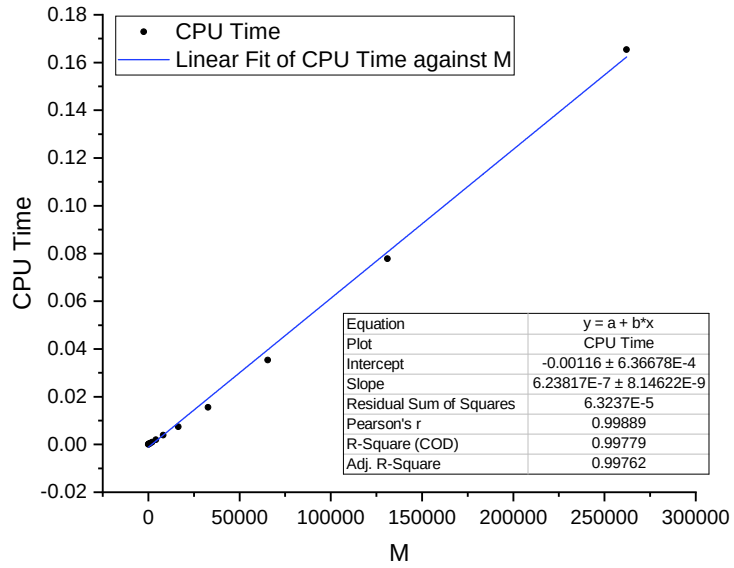

Figure 7: Linear Fit of CPU Time against M

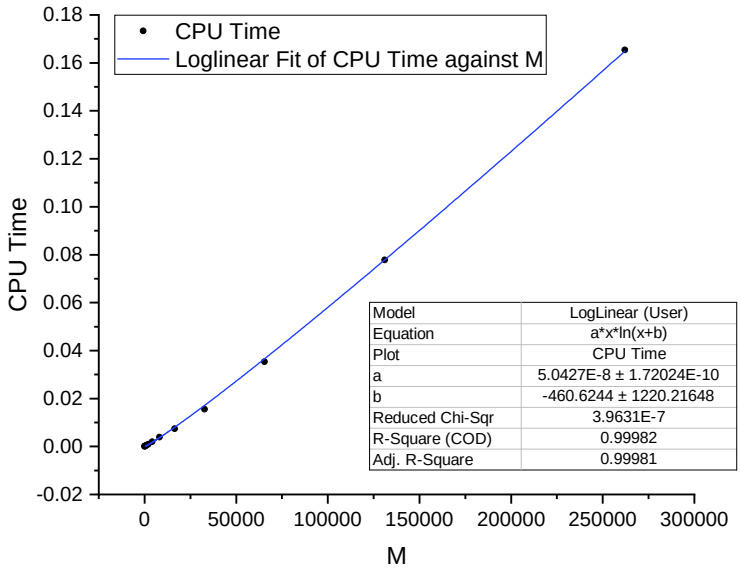

Figure 8: Loglinear Fit of CPU Time against M

From the above graphs, we can observe that the loglinear model performs a better fit of the data. Plus, compare the adjusted R^2, we found $R^2_{\text{linear}} = 0.99762 < 0.99981 = R^2_{\text{loglinear}}$, which indicates that the loglinear model is a better model. Hence, we conclude that the complexity of the method is $O(M \log M)$.

5 Accuracy Analysis

This section is oriented to Question 3.

To evaluate the order of accuracy, we need to select a norm of the curve. In this project, we select the length of the curve as the norm.

Fix $M = 64$, we run the code `ERROR(64, N, 0, 4)`, for $N = 100, 200, 400, 800, 1600$. The results are tabulated as follows.

N	$L(4)$
100	5.07×10^{17}
200	3.74×10^{16}
400	18.1689
800	18.1654
1600	18.1637

Table 3: The length $L(4)$ of the curve at different values of $N \in [100, 1600]$

Choose $N = 1600$ as standard value, calculated the differences as the errors. The errors are tabulated as follows.

N	error
100	5.07×10^{17}
200	3.74×10^{16}
400	5.20×10^{-3}
800	1.70×10^{-3}

Table 4: The error of the curve at different values of $N \in [100, 800]$

We cannot deduce the order of accuracy based on this data, because for $N = 100, 200$, the time-step is too large for the forward Euler Method to be stable, and the resulting curve blows up, which is indicated by the huge value of length.

In order to get the order of accuracy, we may give up the first 2 sets of unstable data and extend the reference value of N to 25600. The results are tabulated as follows.

N	$L(4)$	N	$L(4)$
400	18.1689	6400	18.1624
800	18.1654	12800	18.1622
1600	18.1637	25600	18.1621
3200	18.1629	51200	18.1621

Table 5: The length $L(4)$ of the curve at different values of $N \in [400, 25600]$

Notice that for $N = 51200$, $L(4) = 18.1621$, which is the same as that of $N = 25600$. This means that the accuracy is beyond `Matlab`. Hence, choose $N = 25600$ as standard value, calculated the differences as the errors. The errors are tabulated as follows.

N	error	N	error
400	6.80×10^{-3}	3200	8.00×10^{-4}
800	3.30×10^{-3}	6400	3.00×10^{-4}
1600	1.60×10^{-3}	12800	1.00×10^{-4}

Table 6: The error of the curve at different values of $N \in [400, 25600]$

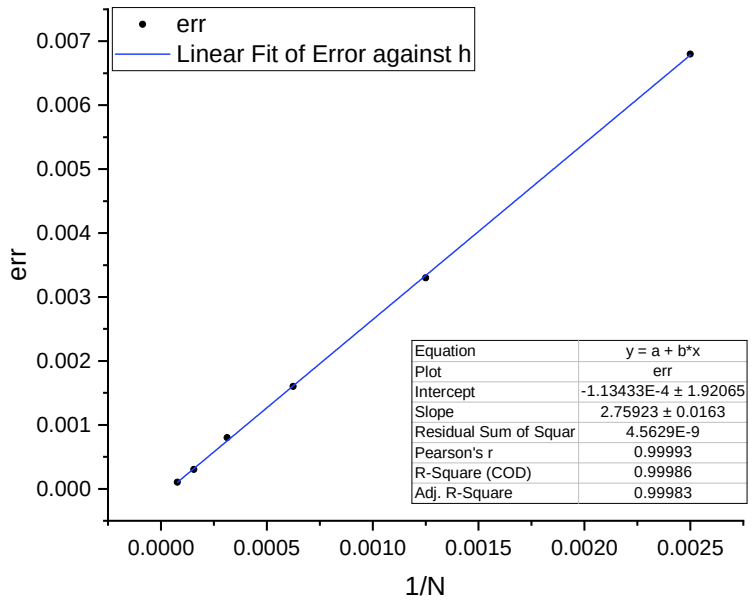

Figure 9: Linear fit of error against h

From the above table and graph, we see that as the number of time-steps (N) doubles, the error reduced by half. Hence, the rate of convergence is linear, *i.e.* $O(h)$.

6 Stability Analysis

This section is oriented to Question 4.

To simulate using $N = 100$ and $M = 128$, simulate the evolution of the curve of time $t \in [0, 4]$ until $t = 4$, we run the `Matlab` code: `CIRCULIZE(128, 100, 0, 4)`. The results are represented as follows (with normal vector on).

At $t = 0s$, *i.e.* the initial curve, the plot is given as:

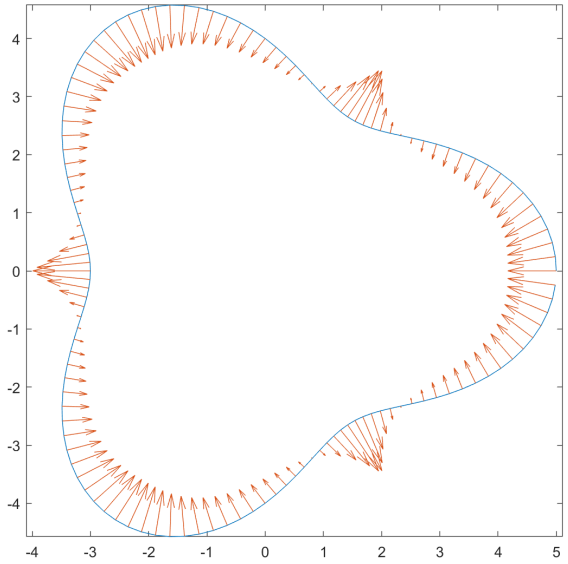

Figure 10: Evolution of the curve, at $t = 0s$

At $t = 1s$, the plot is given as:

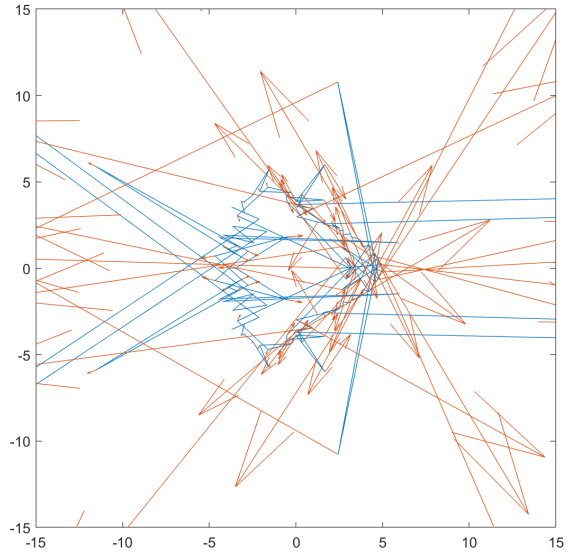

Figure 11: Evolution of the curve, at $t = 1s$

At $t = 2s$, the plot is given as:

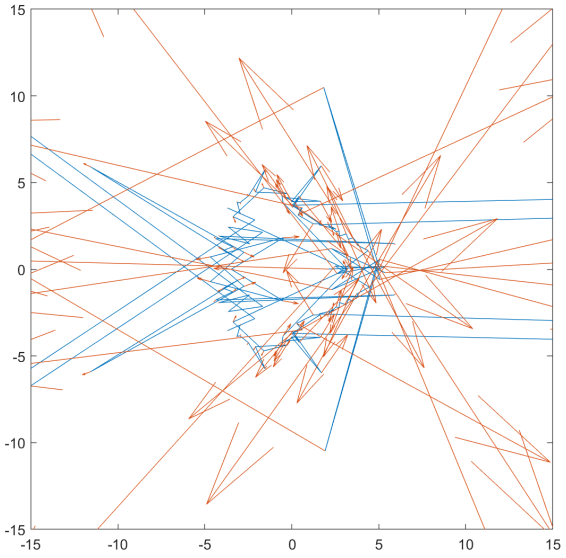

Figure 12: Evolution of the curve, at $t = 2s$

At $t = 3s$, the plot is given as:

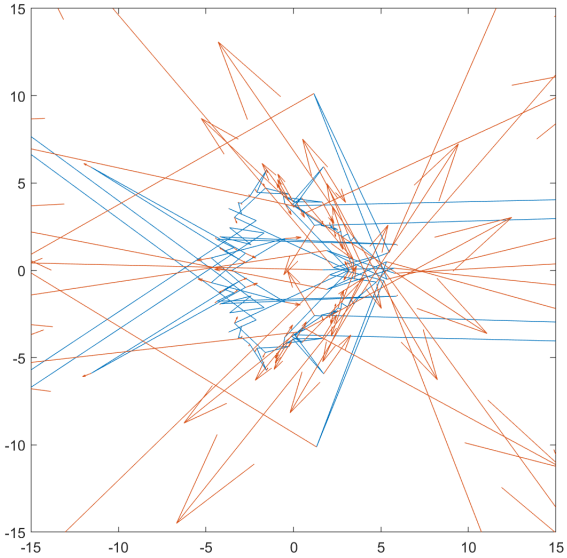

Figure 13: Evolution of the curve, at $t = 3s$

At $t = 4s$, the plot is given as:

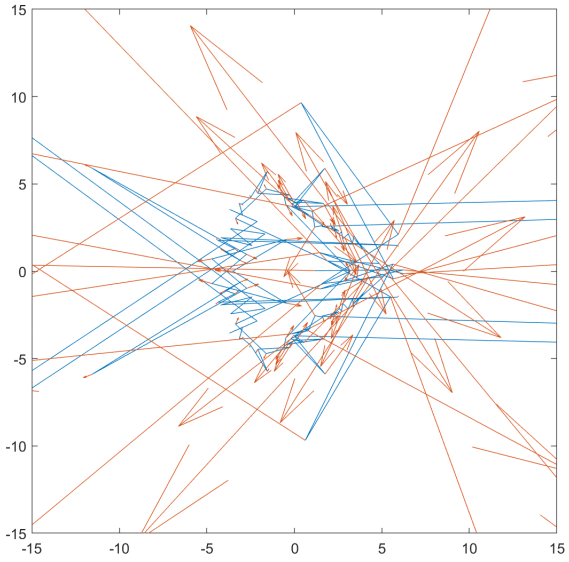

Figure 14: Evolution of the curve, at $t = 4s$

In order to show the evolution process, we sketch all these figures in the same plot:

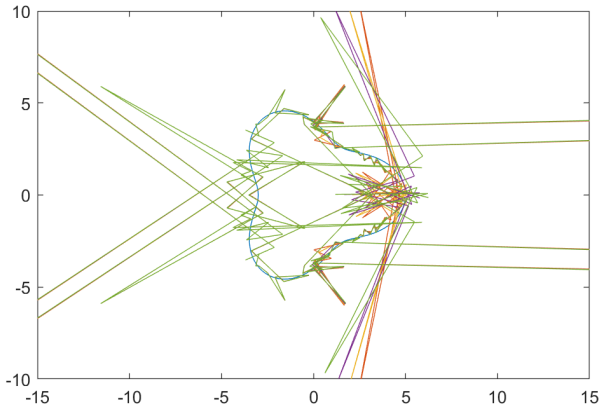

Figure 15: Evolution of the curves

From the graphs above, we observe and conclude that:

1. The method is failed (*i.e.* diverges) under the set of parameters, the plot lost shape from Figure 11.

2. The reason for this is that the time step chosen does not satisfy the stability criteria of the forward Euler method.

   The stability criteria for this ODE is an inequality that

$$\frac{1}{N} = \Delta t < f(M)$$

where $f(M)$ is monotonic decreasing function of M.

As $N = 100$, the time-step $\Delta t = 0.04s$. When the $M = 32$ is small, $\frac{1}{N} = \Delta t < f(M)$ is satisfied, so the method converges. However, when $M = 128$ is larger, $\frac{1}{N} = \Delta t < f(M)$ is no longer true, so the method diverges and failed.

In order to make the method work under $M = 128$, we can increase the value of N, to decrease the value of time-step. Take $N = 1600$, and run `CURCULIZE(128, 1600, 0, 4)`, we obtain:

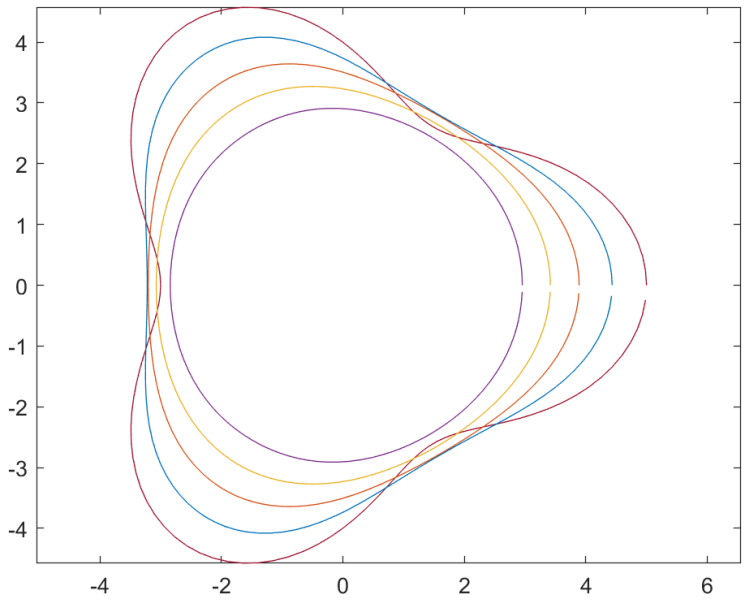

Figure 16: Evolution of the curves

7 Conclusion

In our project, We use the **Euler Method** and differentiation using **Fast Fourier Transform** and **Inverse Fast Fourier Transform**, to we de-noise a predefined initial 2D-curve surface and obtain a smooth curve as output.

Investigating the method, we verified that the time complexity is loglinear, which is $O(M \log M)$, and that the rate of convergence is first order, which is $O(h)$.

At last, we observed that the method is not unconditionally stable. For larger values of M, one may need larger value of N correspondingly to ensure that the method converges.

8 References

[1] Mean Curvature Flow. [Online; accessed 10-Nov-2019] `https://en.wikipedia.org/wiki/Mean_curvature_flow`

[2] S. Veerapanen."project.pdf"(2019). University of Michigan, Ann Arbor. [Online; accessed 5-Nov-2019]. `https://www.dropbox.com/s/zj4tnxu7rgvuvp7/project.pdf?dl=0`