Circular-Kernel-Based Curvature-Motion Algorithms

David Li | davidlxl@umich.edu

Selim Esedoglu (Faculty Advisor) | esedoglu@umich.edu

August 12, 2022

Abstract

Mean curvature flow is an area of interest in mathematics, computer science, material science, and other fields. Over the course of summer 2022, I developed single-circular and triple-circular algorithms for the evolution of 2-dimensional curves under mean curvature flow.[1] This document serves as a user manual for these algorithms. Some background is given at the beginning, followed by a concrete walkthrough. The order of accuracy for each algorithm is demonstrated at the end.

Contents

1 Introduction **1**

2 Methodology **1**

3 Walkthrough **1**

3.1 Initializing the Initial Level-Set Function 1

3.2 Compiling Source Code 3

3.3 Simulating the Evolution 3

4 Demonstration of Order of Convergence **5**

4.1 Benchmark Final Curve 6

4.2 Approximate Final Curve 8

4.3 Maximum Relative Error Versus Time Step Size 9

4.3.1 Single-Circular Algorithm 10

4.3.2 Triple-Circular Algorithm 11

A Performance Comparison Among Different Implementations **13**

1. The algorithm source code is written in C but is intended to be utilized with MATLAB. As of now, we have not decided on when to release the source code to the public, but we may be contacted by email.

1 Introduction

Our domain of interest is $D = [0,1] \times [0,1] \subset \mathbb{R}^2$. Let $\Sigma^0 \subset D$ denote the union of our initial curve of interest and its interior. The initial curve of interest is the boundary of Σ^0, and it is assumed to be closed, simple, and smooth. A concrete example is shown in subsection 3.1. We are interested in how the curve evolves over time under mean curvature flow.

2 Methodology

We refer to our method as the *level-set method*. Let $\phi^0 : D \to \mathbb{R}$ be our initial level-set function, which is assumed to be smooth. We choose the associated Σ of any level-set function ϕ to be $\{x \in D \mid \phi(x) \geqslant 0\}$. Thus our initial curve of interest consists of $\{x \in D \mid \phi^0(x) = 0\}$ (i.e. the "zero-level set of ϕ^0"). A concrete example is shown in subsection 3.1.

Let $T \in (0, \infty)$ be our evolution time of interest. Our method is to update the initial level-set function a certain number of times and then use the resulting updated level-set function to approximate the final curve. Let n be the number of updates. The size of each time step is then given by $\delta t = \frac{T}{n}$. Let ϕ^n denote the resulting level-set function after n updates and Σ^n denote the union of the associated curve and its interior. The approximate final curve at time T (i.e. the boundary of Σ^n) is then given by $\{x \in D \mid \phi^n(x) = 0\}$.

3 Walkthrough

A detailed walkthrough in MATLAB is given in this section to simulate the evolution of a specific curve (an ellipse) over a specific amount of time ($T = 0.01$).

3.1 Initializing the Initial Level-Set Function

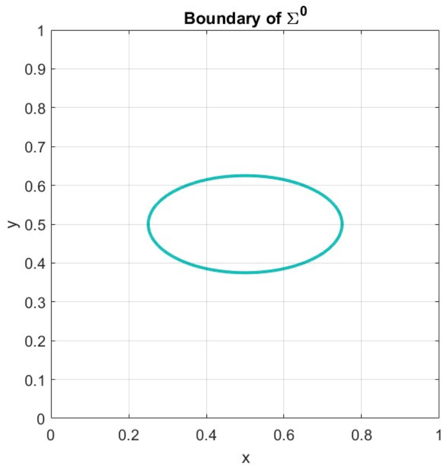
In this walkthrough, we will use an ellipse as our initial curve of interest. We let $a_0 = 0.25$ and $b_0 = 0.125$ such that the ellipse is defined by $\{(x,y) \in \mathbb{R}^2 \mid \frac{(x-0.5)^2}{a_0^2} + \frac{(y-0.5)^2}{b_0^2} = 1\} \subset D$. Note that the center of the ellipse is at $(0.5, 0.5)$. To initialize the corresponding variable (`phi_0`) in MATLAB, we also need to decide on the spatial resolution. Generally, a higher spatial resolution leads to a longer computation time but also a more accurate approximation. Typically, a spacing of $\frac{1}{1600}$ between two consecutive grid points would yield an excellent approximation of the final curve. Thus, we initialize the level-set function ϕ^0 as follows:

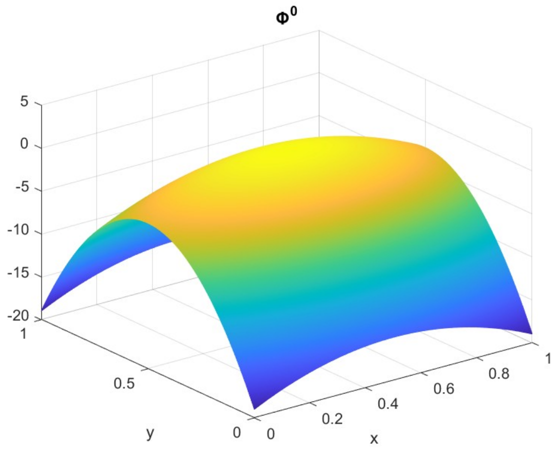
```
numOfPointsForXDimension = 1601;
```

```matlab
numOfPointsForYDimension = 1601;
a0 = 0.25; % initial a
b0 = 0.125; % initial b

[x, y] = meshgrid((0 : numOfPointsForXDimension - 1) / (numOfPointsForXDimension - 1), ...
                  (0 : numOfPointsForYDimension - 1) / (numOfPointsForYDimension - 1));

phi_0 = 1 - (((x - 0.5) / a0) .^ 2 + ((y - 0.5) / b0) .^ 2); % initial level-set function
```

We can visualize the initial curve and level-set function as follows:

```matlab
contour(x, y, phi_0, [0, 0], 'LineWidth', 2);
axis square;
title('Boundary of \Sigma^0');
xlabel('x');
ylabel('y');
grid on;

mesh(x, y, phi_0);
title(sprintf('%s^0', char(hex2dec('03A6'))));
xlabel('x');
ylabel('y');
grid on;
```





For the purpose of subsection 4.3, we define the `initializeLevelSetFcn` function as follows:

```matlab
function phi_0 = initializeLevelSetFcn(a0, b0, numOfPointsForXDimension, ...
                                                            numOfPointsForYDimension)

    [x, y] = meshgrid((0 : numOfPointsForXDimension - 1) / ...
                                                     (numOfPointsForXDimension - 1), ...
```

```
                     (0 : numOfPointsForYDimension - 1) / (numOfPointsForYDimension - 1));

    phi_0 = 1 - (((x - 0.5) / a0) .^ 2 + ((y - 0.5) / b0) .^ 2);

end
```

3.2 Compiling Source Code

Place the source-code files "updatePhiWithSingleCircularKernelWithImprovedBisectionWithNB.c" and "updatePhiWithTripleCircularKernelWithImprovedBisectionWithNB.c" in the current folder (which can be determined by the `pwd` command). Compile the source code using the `mex` command as follows:

```
mex updatePhiWithSingleCircularKernelWithImprovedBisectionWithNB.c;
mex updatePhiWithTripleCircularKernelWithImprovedBisectionWithNB.c;
```

3.3 Simulating the Evolution

We are interested in the resulting curve after the initial curve evolves for a time interval of $T = 0.01$. To perform the simulation, we need to decide on the number of updates. A larger number of updates leads to a longer computation time but also a more accurate approximation. For the sake of this walkthrough, we simulate using $n = 5$ updates. Thus the time step size is $\delta t = \frac{T}{n} = \frac{1}{500}$.

We can use either the `updatePhiWithSingleCircularKernelWithImprovedBisectionWithNB` or `updatePhiWithTripleCircularKernelWithImprovedBisectionWithNB` algorithm to perform the simulation. In either case, we need to decide on the values of certain parameters associated with the algorithm. The kernel radius r is immediately determined by the time step size (due to our mathematical derivations for developing the algorithms[2]): $r = \sqrt{2\delta t}$. The number of iterations, on the other hand, is more subjective. This parameter controls how many bisections are performed vertically for each pixel during each update. A larger value of this parameter leads to a longer computation time but also a more accurate approximation. Typically, a value of 40 is more than sufficient to yield an excellent approximation.

Using the `updatePhiWithSingleCircularKernelWithImprovedBisectionWithNB` algorithm (which runs faster than the other algorithm but is less accurate) as an example, we simulate the evolution and obtain the updated level-set function (`phi_curr`) as follows:

```
T = 0.01; % evolution time
numOfUpdates = 5; % number of updates with the single-circular kernel
delta_t = T / numOfUpdates; % time step size
```

2. Steven J. Ruuth, "Efficient algorithms for diffusion-generated motion by mean curvature" (PhD diss., University of British Columbia, 1996), 61–63, https://doi.org/http://dx.doi.org/10.14288/1.0079751, https://open.library.ubc.ca/collections/ubctheses/831/items/1.0079751.

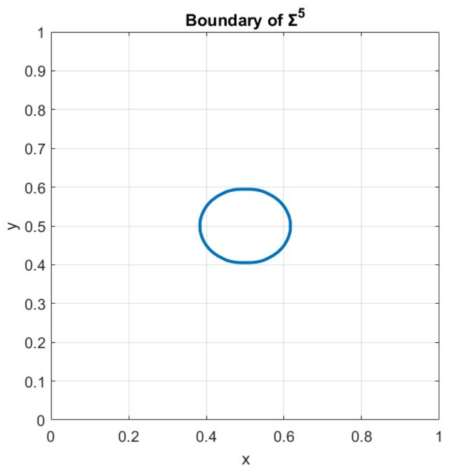
```
r = sqrt(2 * delta_t); %  kernel radius
numOfIterations = 40; % number of iterations for vertical bisection

phi_curr = updatePhiWithSingleCircularKernelWithImprovedBisectionWithNB(phi_0, r, ...
                                                                numOfIterations, numOfUpdates);
% alternative algorithm:
% phi_curr = updatePhiWithTripleCircularKernelWithImprovedBisectionWithNB(phi_0, r, ...
%                                                               numOfIterations, numOfUpdates);
```

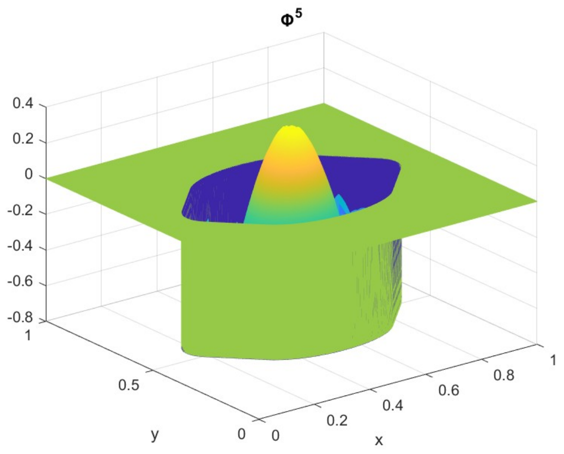
Now, even though the approximate final curve is theoretically obtained by taking the "zero-level set" of `phi_curr`, there is one more subtlety to consider in practice. Since we incorporate the feature of "narrow banding" in our algorithms to speed up the computation time, `phi_curr` is only valid in the vicinity of the initial curve. Thus, to obtain the approximate final curve, we need an additional step of filtering (since we know that the final curve should definitely be inside the initial ellipse due to the convex nature of an ellipse):

```
contourMatrix = contourc(x(1, :), y(:, 1), phi_curr, [0, 0]);
contourMatrix_filtered = contourMatrix(:, ((contourMatrix(1, :) - 0.5) / a0) .^ 2 + ...
                                          ((contourMatrix(2, :) - 0.5) / b0) .^ 2 < 1);
x_curr = contourMatrix_filtered(1, :); % x-coordinates of the final curve
y_curr = contourMatrix_filtered(2, :); % y-coordinates of the final curve
```

Finally, we can visualize the resulting final curve and updated level-set function as follows:

```
plot(x_curr, y_curr, 'LineWidth', 2);
axis([0, 1, 0, 1]);
axis square;
title(sprintf('Boundary of %s^{%d}', char(hex2dec('03A3')), numOfUpdates));
xlabel('x');
ylabel('y');
grid on;

mesh(x, y, phi_curr);
title(sprintf('%s^{%d}', char(hex2dec('03A6')), numOfUpdates));
xlabel('x');
ylabel('y');
grid on;
```





4 Demonstration of Order of Convergence

The single-circular algorithm has an order of convergence of 1 whereas the triple-circular algorithm has an order of convergence of 2. This means that, assuming the spatial resolution is sufficiently fine, as the number of updates becomes large, every time we increase the number of updates by a factor of α (which translates to decreasing the time step size by a factor of α), where $\alpha \in (1, \infty)$, the absolute difference between the distance from an arbitrary point on the initial curve to the "normal-direction corresponding point" on the exact final curve and the distance from the same point on the initial curve to the "normal-direction corresponding point" on the approximate final curve would decrease by at least a factor of α for the single-circular algorithm and by at least a factor of α^2 for the triple-circular algorithm.

To verify that our algorithms do indeed have the expected orders, we came up with two types of tests: single-ray test and multiray test. In both tests, we let the initial curve be the same ellipse that was defined in the last section and we use the same evolution time $T = 0.01$. In the single-ray test, we take the absolute difference between the distance from the rightmost point of the initial ellipse to the rightmost point of the exact final curve and the distance from the rightmost point of the initial ellipse to the rightmost point of the approximate final curve. Then we define the relative error to be the computed absolute difference divided by the distance from the rightmost point of the initial ellipse to the rightmost point of the exact final curve. We would then observe the behavior of this relative error with respect to the number of updates. In the multiray test, on the other hand, we take 36 such triples of points around the curves, compute 36 relative errors, and observe the behavior of the maximum of these relative errors with respect to the number of updates. Since the second test is both more challenging for our algorithms in theory and more meaningful in practice, we will demonstrate the order of convergence using the second test. The details of the single-ray tests, however, can be found in "singleRayTestingForSingleCircularKernel.mlx" and "singleRayTestingForTripleCircularKernel.mlx".

4.1 Benchmark Final Curve

Since there is no explicit formula for the exact curve at any time during the evolution under mean curvature flow, we need to come up with another algorithm that serves as the generator of the *benchmark final curve*. Such an algorithm is written in the source code "updateEllipseWithFirstOrderAccuracy.c". Note that the name may be slightly misleading for two reasons: 1) the "ellipse" refers to the fact that the input curve is an ellipse, which does not mean that the curve remains an ellipse throughout the evolution, and 2) just because an input curve is not an ellipse does not necessarily mean the algorithm can't compute a valid benchmark final curve for it. But since we are only interested in the specific test case where the input curve is an ellipse, I decided to name the algorithm this way. I also implemented "updateEllipseWithSecondOrderAccuracy.c" and "updateEllipseWithThirdOrderAccuracy.c", which account for the second-order and third-order time coefficients in each update, respectively, but these are developed mostly for pedagogical purposes (they can be used for verifying the expected order of convergence for a single update; see "singleRayTestingForSingleCircularKernel.mlx" and "singleRayTestingForTripleCircularKernel.mlx") and are not suitable for the purpose of demonstrating the order of convergence for the total error accumulated by multiple updates.

Place the "updateEllipseWithFirstOrderAccuracy.c" file in the current folder and compile it as usual:

```
mex updateEllipseWithFirstOrderAccuracy.c;
```

The benchmark final curve is then computed by the `calculateBenchmarkSolution()` function defined as follows:

```
function [x, y] = initializeEllipse(n, a, b)

    % n = number of distinct points on the ellipse (initial and final points are the same)
    % a and b are both between 0 and 0.5 such that
    %                                        ((x - 0.5) / a) ^ 2 + ((y - 0.5) / b) ^ 2 = 1

    t = (0 : n) / n * 2 * pi;

    x = 0.5 + a * cos(t);
    y = 0.5 + b * sin(t);

end

function [xNew, yNew, thetas] = calculateBenchmarkSolution(a0, b0, T, n, nt, T_reparam)

    % a0 = initial a
    % b0 = initial b
    % T = evolution time
    % n = num of distinct points on the ellipse
    % nt = num of time steps
    % T_reparam = num of time steps per reparameterization

    [x, y] = initializeEllipse(n, a0, b0);
    [xNew, yNew] = updateEllipseWithFirstOrderAccuracy(x, y, T, nt, T_reparam);

    xNew = xNew(1 : end - 1);
    yNew = yNew(1 : end - 1);
    thetas = atan2(yNew - 0.5, xNew - 0.5); % atan2() returns a value inside (-pi, pi]
    thetas(thetas == pi) = -pi; % convert the values into [-pi, pi)
    [thetas, indicesForSorting] = sort(thetas); % angles now go from -pi to pi ascendingly
    xNew = xNew(indicesForSorting);
    yNew = yNew(indicesForSorting);

end
```

Note that for the benchmark calculation, there is a stability condition regarding the relation between and `numOfPoints` and `numOfUpdates`. In general, assuming the stability of the combination of the current `numOfPoints` and `numOfUpdates`, for all $\alpha \in (1, \infty)$, whenever `numOfPoints` increases by a factor of α, a sufficient way to guarantee stability is to increase `numOfUpdates` by a factor of α^2. If we do not increase `numOfUpdates` by that much, the benchmark calculation may blow up. To prevent the significant elongation of the computation time when this happens, I included a mechanism in the source code that prints out the message "Something went horribly wrong!" and immediately terminates the calculation whenever it detects that the curve has extended outside the domain of interest $D = [0, 1] \times [0, 1]$ at any point during the evolution. If no such message is printed at the end of the calculation, the calculated benchmark final curve is most likely perfectly valid.

Another relatively minor point pertains to the value of `T_reparam`. This variable determines how many updates need to be performed before the program performs another arc-length reparameterization on the curve. If this is set equal to `0`, then the points on the curve may become overly "crowded" over time and thus blow up the calculation (and induce the "Something went horribly wrong!" message). However, as long as it is reasonably large, it is found not to be too significant for the benchmark calculation. In fact, keeping it equal to 1% of `numOfUpdates` appears to be perfectly acceptable.

For our test scenario of interest, the benchmark final curve is calculated by `calculateBenchmarkSolution(0.25, 0.125, 0.01, 1e2 * 64, 1e4 * 64 * 3, 10 * 64 * 3)`. A larger number of points used in the calculation leads to a higher accuracy of the benchmark final curve. But this solution, which is obtained by using 6400 points on the initial ellipse, turns out to be sufficiently accurate for the purpose of the demonstration of the order of convergence. For more details on how the "consecutive error" in benchmark distance behaves as `numOfPoints` increases, see "motionOfAnEllipse.mlx".

4.2 Approximate Final Curve

The approximate final curve is generated in two steps. First, either the single-circular or triple-circular algorithm is used to generate the updated level-set function `phi_curr`. Then, the `calculateApproximateSolution()` function is used to convert the updated level-set function to the approximate final curve:

```
function [xNew, yNew, thetas] = calculateApproximateSolution(a0, b0, phi_curr)

    % a0 = initial a
    % b0 = initial b
    % phi_curr = updated level-set function

    numOfPointsForXDimension = size(phi_curr, 2);
    numOfPointsForYDimension = size(phi_curr, 1);

    [x, y] = meshgrid( ...
                   (0 : numOfPointsForXDimension - 1) / (numOfPointsForXDimension - 1), ...
                   (0 : numOfPointsForYDimension - 1) / (numOfPointsForYDimension - 1));
    contourMatrix = contourc(x(1, :), y(:, 1), phi_curr, [0, 0]);
    contourMatrix_filtered = contourMatrix(:, ((contourMatrix(1, :) - 0.5) / a0) .^ 2 + ...
                                              ((contourMatrix(2, :) - 0.5) / b0) .^ 2 < 1);
    xNew = contourMatrix_filtered(1, :);
    yNew = contourMatrix_filtered(2, :);

    xNew = xNew(1 : end - 1);
    yNew = yNew(1 : end - 1);
    thetas = atan2(yNew - 0.5, xNew - 0.5); % atan2() returns a value inside (-pi, pi]
    thetas(thetas == pi) = -pi; % convert the values into [-pi, pi)
    [thetas, indicesForSorting] = sort(thetas); % angles now go from -pi to pi ascendingly
    xNew = xNew(indicesForSorting);
    yNew = yNew(indicesForSorting);

end
```

4.3 Maximum Relative Error Versus Time Step Size

There is an additional subtlety in the calculation of the maximum relative error that we have not discussed. To calculate a single relative error along a single ray, we need to calculate two distances: the distance from the intersection of the ray and the initial curve to the intersection of the ray to the benchmark final curve and the distance from the intersection of the ray and the initial curve to the intersection of the ray to the approximate final curve. But there is no guarantee that any of the intersection points is precisely stored in the coordinates data of any curve, since a curve contains infinitely many points in theory. Therefore, an additional logic is needed to approximate each intersection point. We choose the logic to be linear interpolation. The actual implementation is somewhat tricky, and, in the interest of space, is not shown here. However, the implementation details can be found in the `calculateDistanceOnOneRay()` function in "multirayTestingForSingleCircularKernel.mlx".

With the help of the `calculateDistanceOnOneRay()` function, the maximum relative error is calculated as follows:

```matlab
function maxRelativeDistanceError = calculateMaxRelativeDistanceError(a0, b0, ...
                                              xNew1, yNew1, thetas1, xNew2, yNew2, thetas2)

    % xNew1, yNew1, and thetas1 are assumed to be for the benchmark solution, whereas
    % xNew2, yNew2, and thetas2 are assumed to be for the approximate solution

    for theta = -pi : 2 * pi / 36 : (pi - 2 * pi / 36)

        d1 = calculateDistanceOnOneRay(a0, b0, xNew1, yNew1, thetas1, theta);
        d2 = calculateDistanceOnOneRay(a0, b0, xNew2, yNew2, thetas2, theta);

        if theta == -pi
            maxRelativeDistanceError = abs(d1 - d2) / d1;
        else
            maxRelativeDistanceError = max(maxRelativeDistanceError, abs(d1 - d2) / d1);
        end

    end

end
```

Now we are ready to show how the maximum relative error depends on the number of updates n for each algorithm. Recall that the number of updates n is related to the time step size δt via $\delta t = \frac{T}{n}$, and $T = 0.01$ is fixed for our test scenario.

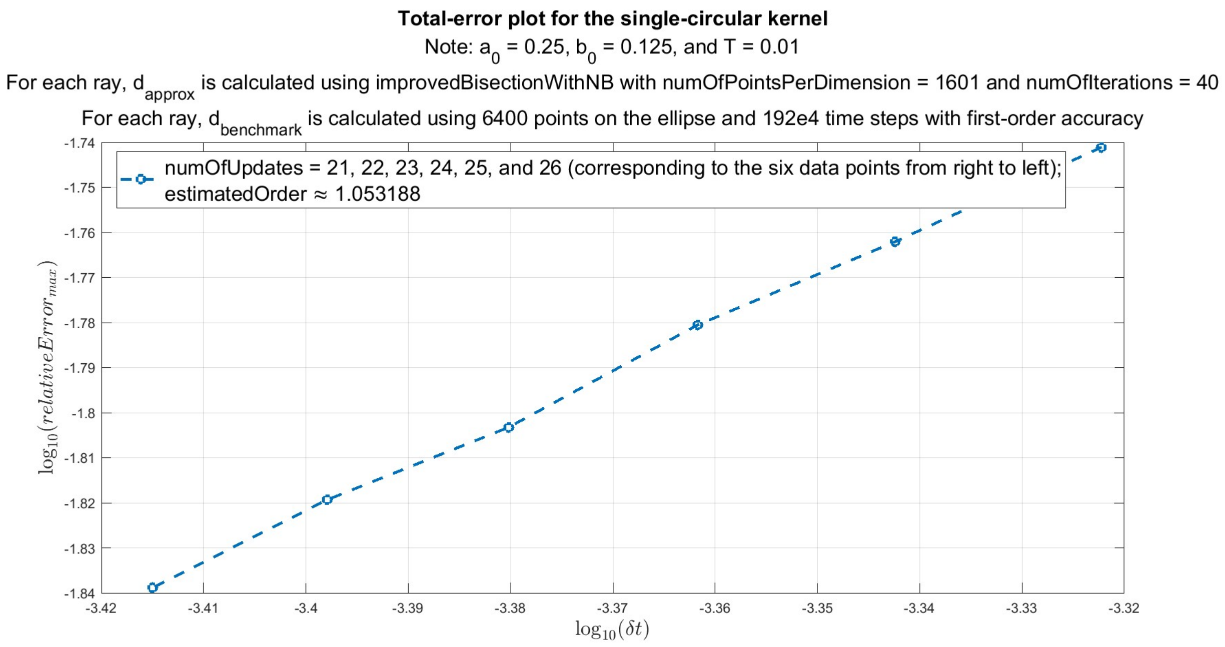
4.3.1 Single-Circular Algorithm

```
T = 0.01; % evolution time
maxRelativeErrors = zeros(6, 1);
delta_ts = zeros(6, 1);

numOfPointsForXDimension = 1601;
numOfPointsForYDimension = 1601;
a0 = 0.25; % initial a
b0 = 0.125; % initial b
[xNew1, yNew1, thetas1] = ...
                 calculateBenchmarkSolution(a0, b0, T, 1e2 * 64, 1e4 * 64 * 3, 10 * 64 * 3);

phi_0 = initializeLevelSetFcn(a0, b0, numOfPointsForXDimension, numOfPointsForYDimension);

numsOfUpdates = [21; 22; 23; 24; 25; 26];

for k = 1 : 6

    numOfUpdates = numsOfUpdates(k); % number of updates with the single-circular kernel
    delta_t = T / numOfUpdates; % time step size
    delta_ts(k) = delta_t;
    r = sqrt(2 * delta_t); % radius of the single-circular kernel
    numOfIterations = 40; % number of iterations for vertical bisection

    phi_curr = updatePhiWithSingleCircularKernelWithImprovedBisectionWithNB(phi_0, r, ...
                                                                numOfIterations, numOfUpdates);

    [xNew2, yNew2, thetas2] = calculateApproximateSolution(a0, b0, phi_curr);
    maxRelativeErrors(k) = calculateMaxRelativeDistanceError(a0, b0, ...
                                                 xNew1, yNew1, thetas1, xNew2, yNew2, thetas2);

end
```

The variable `maxRelativeErrors` stores the six maximum relative errors associated with the six time step sizes stored in `delta_ts`. The order of convergence can be estimated by converting them to the logarithmic scale and taking the slope of the best-fit line for the data:

```
coeffs = polyfit(log10(maxRelativeErrors), log10(maxRelativeErrors), 1);
estimatedOrder = coeffs(1);
textForEstimatedOrder = sprintf('estimatedOrder %s %f', char(hex2dec('2248')), ...
                                                                           estimatedOrder);
```

```

plot(log10(delta_ts), log10(maxRelativeErrors), 'o--', 'LineWidth', 2);
grid on;
xlabel('$\log_{10}(\delta t)$', 'Interpreter', 'latex', 'FontSize', 15);
ylabel('$\log_{10}(relativeError_{max})$', 'Interpreter', 'latex', 'FontSize', 15);
legend({"numOfUpdates = 21, 22, 23, 24, 25, and 26 " + ...
        "(corresponding to the six data points from right to left);" + ...
        newline + textForEstimatedOrder}, 'FontSize', 15, 'Location', 'best');
title('Total-error plot for the single-circular kernel', ...
      {"Note: a_0 = 0.25, b_0 = 0.125, and T = 0.01" + newline + ...
       "For each ray, d_{approx} is calculated using improvedBisectionWithNB with" + ...
                            "numOfPointsPerDimension = 1601 and numOfIterations = 40" + ...
       newline + "For each ray, d_{benchmark} is calculated using 6400 points " + ...
                      "on the ellipse and 192e4 time steps with first-order accuracy"}, ...
      'FontSize', 15);
```

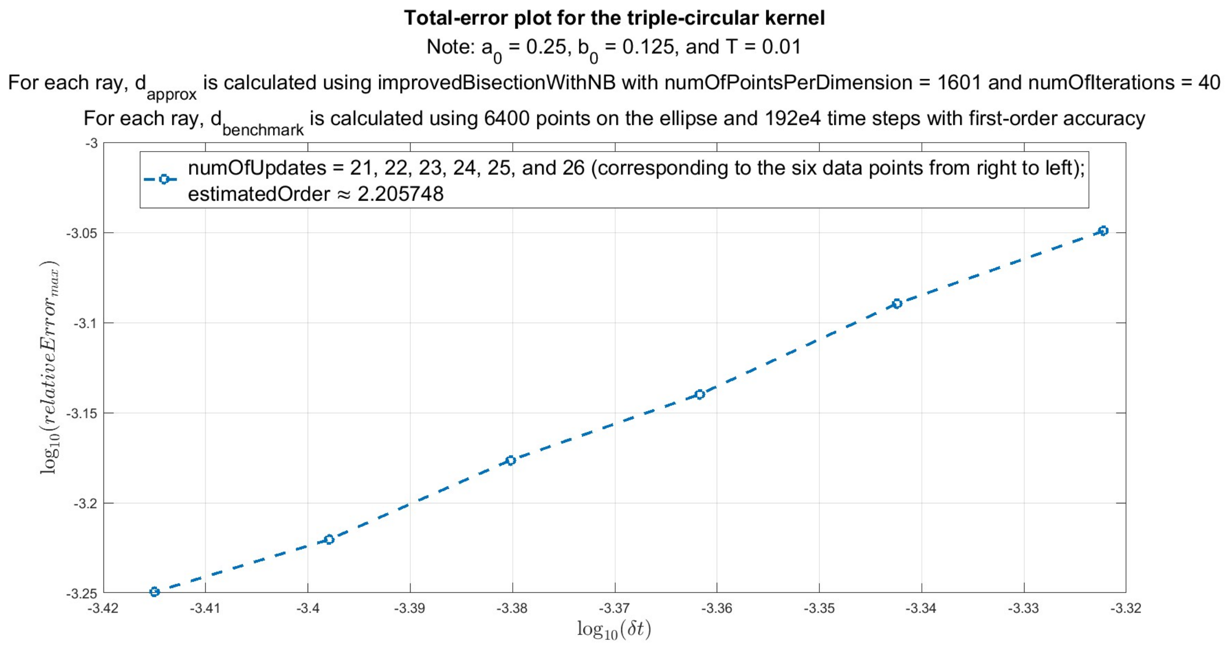


Six of the data points we looked at are shown in the plot. Taking into account the other data points that are not shown, we conclude that the "consecutive estimated order" (calculated by two consecutive data points) starts off being much larger than 1 and then generally stays around 1. Note that by the time `numOfUpdates` reaches 21, the maximum relative error has already decreased to less than 1.82%.

4.3.2 Triple-Circular Algorithm

The result for the triple-circular algorithm can be obtained in very much the same way as the one for the single-circular algorithm. The MATLAB code given in the last subsubsection can be reused with a couple of changes (i.e. changing `updatePhiWithSingleCircularKernelWithImprovedBisectionWithNB` to `updatePhiWithTripleCircularKernelWithImprovedBisectionWithNB` and changing `'Total-error plot for the single-circular kernel'` to `'Total-error plot for the triple-circular kernel'`). Thus, in the interest of space, we will just present the result.

Again, six of the data points are shown in the plot below. Taking into account the other data points that are not shown, we conclude that the pattern for the "consecutive estimated order" is similar to that in the single-circular case; it starts off being much larger than 2 and then generally stays around 2. Note that by the time `numOfUpdates` reaches 21, the maximum relative error has already decreased to less than 0.1%.



References

Ruuth, Steven J. “Efficient algorithms for diffusion-generated motion by mean curvature.” PhD diss., University of British Columbia, 1996. https://doi.org/http://dx.doi.org/10.14288/1.0079751. https://open.library.ubc.ca/collections/ubctheses/831/items/1.0079751.

Appendix

A Performance Comparison Among Different Implementations

For the single-circular algorithm, prior to the birth of "updatePhiWithSingleCircularKernelWithImprovedBisectionWithNB.c", I developed other implementations of the same algorithm: "updatePhiWithSingleCircularKernelWithSorting.c", "updatePhiWithSingleCircularKernelWithBisection.c", and "updatePhiWithSingleCircularKernelWithImprovedBisection.c". In fact, our recommended "updatePhiWithSingleCircularKernelWithImprovedBisectionWithNB.c" is essentially an upgraded version of the latter two.

For the triple-circular algorithm, prior to the birth of "updatePhiWithTripleCircularKernelWithImprovedBisectionWithNB.c", I developed "updatePhiWithTripleCircularKernelWithSorting.c". However, it turns out that for the triple-circular kernel, the sorting-based implementation is not able to achieve an accuracy that is nearly similar to the improved-bisection-with-narrow-banding-based one does given similar parameters that are used in the single-circular case. While I have confidence in the correctness of my sorting-based implementation, this implementation may not be very useful practically, as it may require large values of the "number of query points" parameters and thus significantly longer, if not impractically longer, computation time to obtain a solution of an acceptable accuracy.

The table below compares the performance among the four single-circular implementations and the one triple-circular implementation that is based on improved bisection with narrow banding. The MATLAB source code for the comparisons can be found in "comparisonAmongCircularKernelImplementations.mlx". Note that the same scenario ($a_0 = 0.25$, $b_0 = 0.125$, and $T = 0.01$) is used to generate the results, and the number of updates ($n = 21$) is the same for all five implementations.

Implementation	**Computation Time**	**Max Relative Error**
Sorting-Based Single-Circular Kernel	24 min 46 s	1.8165%
Bisection-Based Single-Circular Kernel	2 h 51 min 8 s	1.8152%
Improved-Bisection-Based Single-Circular Kernel	2 h 40 min 32 s	1.8153%
Improved-Narrow-Band-Bisection-Based Single-Circular Kernel	24 min 26 s	1.8152%
Improved-Narrow-Band-Bisection-Based Triple-Circular Kernel	2 h 2 min 50 s	0.089289%