J. Approx. Softw., Vol. 2, No. 2 (2025) 1

Journal of Approximation Software

doi: 10.13135/3103-1935/11442 – ISSN: 3103–1935

CONIS: A C++ Package for Conic-Preserving Interpolatory Subdivision

N. Bügel [1], L. Romani [2,⋆], and J. Kosinka [3]

[1] *CERN, European Organization for Nuclear Research, Geneva, Switzerland*
[2] *Dipartimento di Matematica, Alma Mater Studiorum Università di Bologna, Italy*
[3] *Bernoulli Institute, University of Groningen, the Netherlands*

Received: 19/12/2024 – Published: 04/09/2025
Communicated by: R. Cavoretto

Abstract

We present a C++ package, named CONIS, designed to compute and visualize the planar curves generated via the subdivision scheme proposed in [3]. We describe the software package, provide instructions to use it, and showcase the package using several numerical tests.

Keywords: curve subdivision, interpolation, conic reproduction (MSC2020: 65D17, 68U07)

1 Introduction

In this paper, we provide a C++ package for computing and visualizing the planar curves generated via the subdivision scheme proposed in [3]. The subdivision scheme under discussion takes as input a planar polygon $\mathbf{P}^{(0)}$ with vertices $\mathbf{p}_i^{(0)}$, $i \in I_0$, as well as the (unit) normals $\mathbf{n}_i^{(0)}$, $i \in I_0$, at these vertices, and constructs a sequence of refined polygons $\{\mathbf{P}^{(k)}\}_{k>0}$ such that:

- the vertices of $\mathbf{P}^{(k-1)}$ are also vertices of $\mathbf{P}^{(k)}$ for any $k > 1$, which guarantees that the vertices of $\mathbf{P}^{(0)}$ are also vertices of $\mathbf{P}^{(k)}$ for any $k > 0$, i.e., the subdivision scheme is *interpolatory*;
- all types of *conic sections are reproduced* whenever the vertices of $\mathbf{P}^{(0)}$ and the associated normals are arbitrarily sampled from them;
- a visually pleasing, smooth curve without unwanted oscillations is generated in the limit of the subdivision process.

⋆ Corresponding author: lucia.romani@unibo.it

To clarify how the subdivision scheme works, we provide a detailed description of all the key operations to be performed at each subdivision step to map the coarser polygon $\mathbf{P}^{(k-1)} = \{\mathbf{p}_i^{(k-1)}, i \in I_{k-1}\}$ equipped with normals $\mathbf{N}^{(k-1)} = \{\mathbf{n}_i^{(k-1)}, i \in I_{k-1}\}$, to the refined polygon $\mathbf{P}^{(k)} = \{\mathbf{p}_i^{(k)}, i \in I_k\}$ and its normals $\mathbf{N}^{(k)} = \{\mathbf{n}_i^{(k)}, i \in I_k\}$. As we are concerned with a binary subdivision scheme, we consider index sets I_{k-1}, I_k such that $|I_k| = 2|I_{k-1}|$. In other words, at each subdivision step we double the number of vertices (and hence of normals too). This means that, for each $i \in I_{k-1}$, we need a rule to compute both $\mathbf{p}_{2i}^{(k)}$, $\mathbf{n}_{2i}^{(k)}$ and $\mathbf{p}_{2i+1}^{(k)}$, $\mathbf{n}_{2i+1}^{(k)}$. Their computation is described by first assuming that $\mathbf{P}^{(k-1)}$ forms a convex polygon.

1.1 Pseudocode

For each $i \in I_{k-1}$:

1. Set $\mathbf{p}_{2i}^{(k)} = \mathbf{p}_i^{(k-1)}$ and $\mathbf{n}_{2i}^{(k)} = \mathbf{n}_i^{(k-1)}$.

2. Define the local neighbourhood of the edge $\mathbf{p}_i^{(k-1)}\mathbf{p}_{i+1}^{(k-1)}$ by considering the 4 consecutive point-normal pairs

$$(\mathbf{p}_{i-1}^{(k-1)}, \mathbf{n}_{i-1}^{(k-1)}),\ (\mathbf{p}_i^{(k-1)}, \mathbf{n}_i^{(k-1)}),\ (\mathbf{p}_{i+1}^{(k-1)}, \mathbf{n}_{i+1}^{(k-1)}),\ (\mathbf{p}_{i+2}^{(k-1)}, \mathbf{n}_{i+2}^{(k-1)}). \tag{1}$$

   For each $j \in \{i-1, i, i+1, i+2\}$, let

$$\mathbf{p}_j^{(k-1)} = [x_j, y_j]^T, \quad \mathbf{n}_j^{(k-1)} = [n_j^x, n_j^y]^T,$$

   and construct the matrix

$$\mathbf{A}' = \begin{bmatrix}
x_{i-1}^2 & y_{i-1}^2 & 2x_{i-1}y_{i-1} & 2x_{i-1} & 2y_{i-1} & 1 & 0 & 0 & 0 & 0 \\
x_i^2 & y_i^2 & 2x_iy_i & 2x_i & 2y_i & 1 & 0 & 0 & 0 & 0 \\
x_{i+1}^2 & y_{i+1}^2 & 2x_{i+1}y_{i+1} & 2x_{i+1} & 2y_{i+1} & 1 & 0 & 0 & 0 & 0 \\
x_{i+2}^2 & y_{i+2}^2 & 2x_{i+2}y_{i+2} & 2x_{i+2} & 2y_{i+2} & 1 & 0 & 0 & 0 & 0 \\
2x_{i-1} & 0 & 2y_{i-1} & 2 & 0 & 0 & -n_{i-1}^x & 0 & 0 & 0 \\
0 & 2y_{i-1} & 2x_{i-1} & 0 & 2 & 0 & -n_{i-1}^y & 0 & 0 & 0 \\
2x_i & 0 & 2y_i & 2 & 0 & 0 & 0 & -n_i^x & 0 & 0 \\
0 & 2y_i & 2x_i & 0 & 2 & 0 & 0 & -n_i^y & 0 & 0 \\
2x_{i+1} & 0 & 2y_{i+1} & 2 & 0 & 0 & 0 & 0 & -n_{i+1}^x & 0 \\
0 & 2y_{i+1} & 2x_{i+1} & 0 & 2 & 0 & 0 & 0 & -n_{i+1}^y & 0 \\
2x_{i+2} & 0 & 2y_{i+2} & 2 & 0 & 0 & 0 & 0 & 0 & -n_{i+2}^x \\
0 & 2y_{i+2} & 2x_{i+2} & 0 & 2 & 0 & 0 & 0 & 0 & -n_{i+2}^y
\end{bmatrix}_{12\times 10}. \tag{2}$$

   Then let w_e^v and w_e^n be large positive numbers (e.g., $w_e^v = 10^5$, $w_e^n = 10^5$), whereas let w_o^n be a smaller positive value (e.g., $w_o^n = 1$). After defining $w_o^v = \tau w_o^n$ with $\tau > 1$ (e.g. $\tau = 10$), build the 12×12 diagonal matrix

$$\mathbf{W} = \mathrm{diag}(w_o^v, w_e^v, w_e^v, w_o^v, w_o^n, w_o^n, w_e^n, w_e^n, w_e^n, w_e^n, w_o^n, w_o^n)$$

   and compute the right-singular vector $\mathbf{q}'$ of $\mathbf{WA}'$ that corresponds to its smallest singular value. Note that $\mathbf{q}'$ is a vector in $\mathbb{R}^{10}$ and the first 6 of its entries provide the coefficients $q_{20}, q_{02}, q_{11}, q_{10}, q_{01}, q_{00}$ of the conic

$$\mathscr{C}: \quad f(x,y) = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}^T \begin{bmatrix} q_{20} & q_{11} & q_{10} \\ q_{11} & q_{02} & q_{01} \\ q_{10} & q_{01} & q_{00} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = 0 \tag{3}$$

which best fits the 4 point-normal pairs in (1), whereas the last 4 entries of $\mathbf{q}'$ determine the scaling factors $\alpha_{i-1}, \alpha_i, \alpha_{i+1}, \alpha_{i+2}$ for the normals $\mathbf{n}_{i-1}^{(k-1)}, \mathbf{n}_i^{(k-1)}, \mathbf{n}_{i+1}^{(k-1)}, \mathbf{n}_{i+2}^{(k-1)}$.

3. For each edge given by vertices $\mathbf{p}_i^{(k-1)} = [x_i, y_i]^T$, $\mathbf{p}_{i+1}^{(k-1)} = [x_{i+1}, y_{i+1}]^T$, compute

$$\mathbf{m}_{2i+1} = \frac{\mathbf{p}_i^{(k-1)} + \mathbf{p}_{i+1}^{(k-1)}}{2}, \tag{4}$$

$$\mathbf{n}_i^{\text{edge}} = (\mathbf{p}_{i+1}^{(k-1)} - \mathbf{p}_i^{(k-1)})^{\perp} = \begin{bmatrix} y_i - y_{i+1} \\ x_{i+1} - x_i \end{bmatrix}, \tag{5}$$

and then set the so-called *edge vertex* as

$$\mathbf{p}_{2i+1}^{(k)} = \begin{cases} \mathbf{l}(t_1), & \text{if } \|\mathbf{m}_{2i+1} - \mathbf{l}(t_1)\|_2 \leq \|\mathbf{m}_{2i+1} - \mathbf{l}(t_2)\|_2, \\ \mathbf{l}(t_2), & \text{otherwise,} \end{cases} \tag{6}$$

where

$$\mathbf{l}(t) = \mathbf{m}_{2i+1} + t\,\mathbf{n}_i^{\text{edge}} = \left[\tfrac{x_i + x_{i+1}}{2} + t\,(y_i - y_{i+1}),\ \tfrac{y_i + y_{i+1}}{2} + t\,(x_{i+1} - x_i) \right]^T, \quad t \in \mathbb{R}. \tag{7}$$

The real parameters t_1 and t_2 appearing in (6) are the two solutions of the quadratic equation

$$\begin{bmatrix} (x_i + x_{i+1})/2 + t\,(y_i - y_{i+1}) \\ (y_i + y_{i+1})/2 + t\,(x_{i+1} - x_i) \\ 1 \end{bmatrix}^T \begin{bmatrix} q_{20} & q_{11} & q_{10} \\ q_{11} & q_{02} & q_{01} \\ q_{10} & q_{01} & q_{00} \end{bmatrix} \begin{bmatrix} (x_i + x_{i+1})/2 + t\,(y_i - y_{i+1}) \\ (y_i + y_{i+1})/2 + t\,(x_{i+1} - x_i) \\ 1 \end{bmatrix} = 0 \tag{8}$$

in the unknown t, which identifies the (two) intersections between the line $\mathbf{l}(t)$ and the conic $\mathscr{C}$ previously computed in (3).

4. Finally select $\mathbf{n}_{2i+1}^{(k)}$ as the normal sampled from the said conic $\mathscr{C}$ at the point $\mathbf{p}_{2i+1}^{(k)}$.

We now describe the refinement step in case $\mathbf{P}^{(k-1)}$ is a non-convex polygon. Following the idea proposed by Romani et al. [2], we start splitting the control polygon $\mathbf{P}^{(k-1)}$ into a collection of globally convex sub-polygons. The splitting takes place by introducing the inflection point

$$\mathbf{p}_{\text{infl}} = \mathbf{m}_{2i+1}, \tag{9}$$

whenever the points $\mathbf{p}_{i-1}^{(k-1)}$ and $\mathbf{p}_{i+2}^{(k-1)}$ lie on different half planes with respect to the edge $\mathbf{p}_i^{(k-1)}\mathbf{p}_{i+1}^{(k-1)}$. As our method uses both points and normals to find the best conic fit, the normal $\mathbf{n}_{\text{infl}}$ at the inflection point $\mathbf{p}_{\text{infl}}$ also needs to be set appropriately as it influences the resulting curve. In particular, we define

$$\mathbf{n}_{\text{infl}} = \begin{cases} \mathbf{n}_{\text{infl}}^0 & \text{if } \angle \mathbf{n}_{\text{infl}}^0 \mathbf{n}_i^{\text{edge}} < \angle \mathbf{n}_{\text{infl}}^1 \mathbf{n}_i^{\text{edge}}, \\ \mathbf{n}_{\text{infl}}^1 & \text{otherwise,} \end{cases} \tag{10}$$

where

$$\mathbf{n}_{\text{infl}}^0 = (1 - \gamma_0)\,\frac{\mathbf{n}_i^{\text{edge}}}{\|\mathbf{n}_i^{\text{edge}}\|_2} + \gamma_0\,\frac{\mathbf{p}_{i+1}^{(k-1)} - \mathbf{p}_i^{(k-1)}}{\|\mathbf{p}_{i+1}^{(k-1)} - \mathbf{p}_i^{(k-1)}\|_2} \quad \text{with} \quad \gamma_0 = \frac{1}{2} - \left| \frac{\phi_i^{(k-1)}}{\pi} - \frac{1}{2} \right|,$$

and

$$\mathbf{n}_{\text{infl}}^{1} = (1-\gamma_1)\,\frac{\mathbf{n}_i^{\text{edge}}}{\|\mathbf{n}_i^{\text{edge}}\|_2} + \gamma_1\,\frac{\mathbf{p}_{i+1}^{(k-1)} - \mathbf{p}_i^{(k-1)}}{\|\mathbf{p}_{i+1}^{(k-1)} - \mathbf{p}_i^{(k-1)}\|_2} \quad \text{with} \quad \gamma_1 = \frac{1}{2} - \left|\frac{\phi_{i+1}^{(k-1)}}{\pi} - \frac{1}{2}\right|.$$

In the above formulas, $\phi_i^{(k-1)}$ is the angle between $\mathbf{p}_{i-1}^{(k-1)}\mathbf{p}_i^{(k-1)}$ and $\mathbf{p}_i^{(k-1)}\mathbf{p}_{i+1}^{(k-1)}$ whereas $\phi_{i+1}^{(k-1)}$ is the angle between $\mathbf{p}_i^{(k-1)}\mathbf{p}_{i+1}^{(k-1)}$ and $\mathbf{p}_{i+1}^{(k-1)}\mathbf{p}_{i+2}^{(k-1)}$.

We emphasize that once the inflection points have been inserted, a modified polygon $\mathbf{P}^{(k-1)}$ is obtained and instructions in Steps 1–4 are repeated to refine it. However, when considering the edge with endpoints $\mathbf{p}_i^{(k-1)}\,\mathbf{p}_{\text{infl}}$, some changes are required to define its new edge vertex. Specifically, the local neighbourhood of this edge must be modified by considering the 4 consecutive point-normal pairs

$$(\mathbf{p}_{i-2}^{(k-1)}, \mathbf{n}_{i-2}^{(k-1)}),\ (\mathbf{p}_{i-1}^{(k-1)}, \mathbf{n}_{i-1}^{(k-1)}),\ (\mathbf{p}_i^{(k-1)}, \mathbf{n}_i^{(k-1)}),\ (\mathbf{p}_{\text{infl}}, \mathbf{n}_{\text{infl}}),$$

and the weight matrix $\mathbf{W}$ collecting the weights assigned to each vertex and to each normal in such a local neighbourhood becomes

$$\mathbf{W} = \text{diag}(w_o^v, w_o^v, w_e^v, w_e^v, w_o^n, w_o^n, w_o^n, w_o^n, w_e^n, w_e^n, w_e^n, w_e^n).$$

Similarly, the definition of the new edge vertex for the edge with endpoints $\mathbf{p}_{\text{infl}}\,\mathbf{p}_{i+1}^{(k-1)}$ is modified by considering as local neighbourhood the 4 consecutive point-normal pairs

$$(\mathbf{p}_{\text{infl}}, \mathbf{n}_{\text{infl}}),\ (\mathbf{p}_{i+1}^{(k-1)}, \mathbf{n}_{i+1}^{(k-1)}),\ (\mathbf{p}_{i+2}^{(k-1)}, \mathbf{n}_{i+2}^{(k-1)}),\ (\mathbf{p}_{i+3}^{(k-1)}, \mathbf{n}_{i+3}^{(k-1)})$$

and the corresponding weight matrix $\mathbf{W}$ becomes

$$\mathbf{W} = \text{diag}(w_e^v, w_e^v, w_o^v, w_o^v, w_e^n, w_e^n, w_e^n, w_e^n, w_o^n, w_o^n, w_o^n, w_o^n).$$

These changes are due to the fact that, after inserting the inflection point, the three points $\mathbf{p}_i^{(k-1)}, \mathbf{p}_{\text{infl}}, \mathbf{p}_{i+1}^{(k-1)}$ are collinear and thus, if used all together to define the local neighbourhood, they give rise to a singular conic.

2 Description of the code

After the above summary of the main theoretical results published in [3], we now describe the code implemented in C++ and made available at

https://github.com/BugelNiels/conic-subdivision.

For the design of the framework, we mainly focussed on making the individual components simple, reusable, and testable. Additional care was taken to make the implementation as numerically accurate as possible while keeping it performant. Finally, the GUI was designed to allow for easy testing of various approaches.

The software consists of two main components: the core library and the GUI. The core library (Section 2.1) contains all the necessary code for defining curves and performing actions such as conic subdivisions on them. The GUI component (Section 2.2) provides a visual interactive program on top of the core library. By design, the core library does not depend on the GUI and can be built as a stand-alone library. We now discuss these two components in detail.

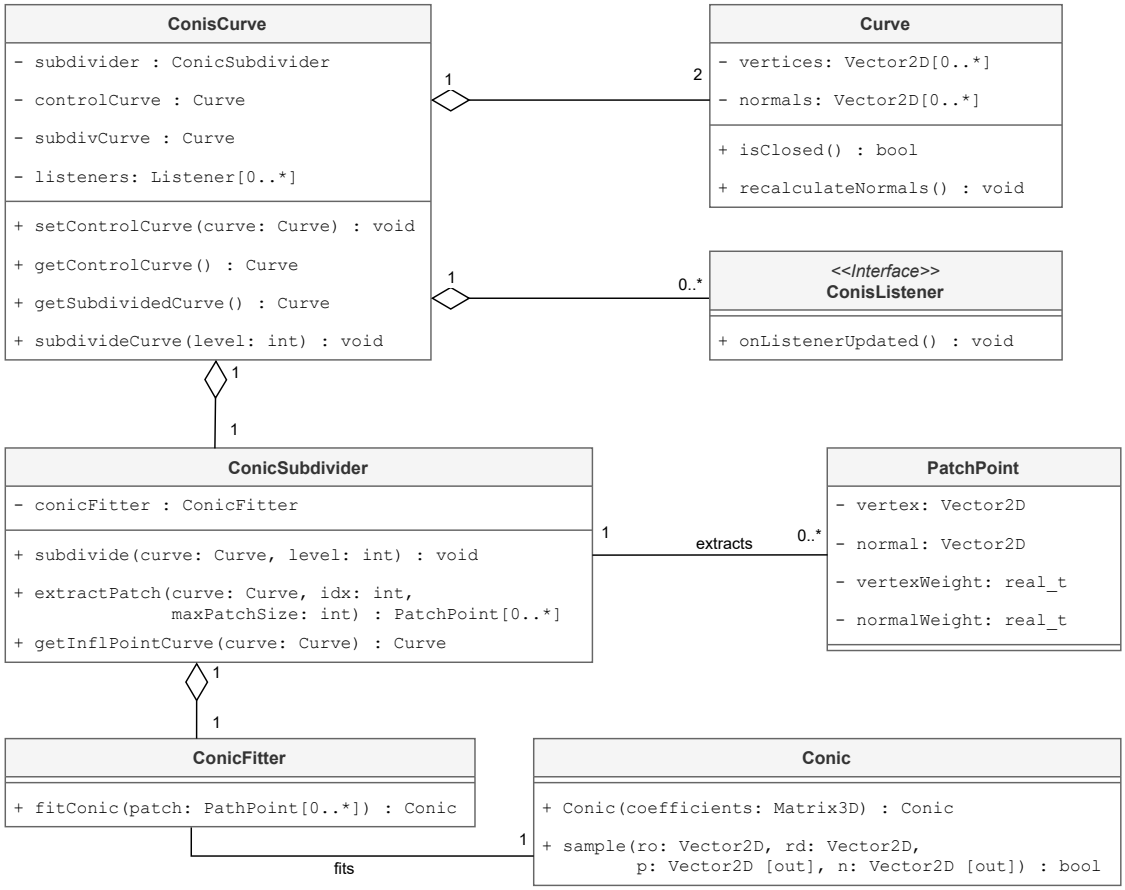


Figure 1: A simplified UML diagram of the CONIS core library.

2.1 Core Library

The basic layout of the core library can be seen in Figure 1. The primary interface for the library is the `ConisCurve` class. This class contains three main items relating to subdivision:

- a control curve that can be manipulated;
- a subdivider that can subdivide the control curve;
- a subdivided curve to store the result of the subdivision process in.

The `ConisCurve` contains a set of listeners to implement the Observer pattern. Any time a notable change happens to either the control or subdivided curve, all the listeners will be notified. This allows us to easily attach a GUI to this library without the library having knowledge of a GUI. This has the added benefit that a command-line interface would be trivial to add.

At the core of the framework is the `Curve` class. It represents a collection of point-normal pairs and facilitates different operations to manipulate these points and normals. It is completely decoupled from conics or the subdivision process and can be used independently from these. Similarly, the `Conic` class is solely a container for the conic coefficients, facilitating different operations on these such as finding the intersection between a ray and said conic via the `sample` method. The `Conic` class is also independent and decoupled from the `Curve` class and the subdivision/fitting processes.

The conic fitting process (cf. Step 2 in Section 1.1) happens in `ConicFitter`. This class takes a collection of point-normal pairs (and their weights) as input and produces the best-fitting conic using singular value decomposition. The subdivision process happens through the `ConicSubdivider` class which takes a `Curve` and a number of subdivision levels as input. The input is used to perform the subdivision process *in-place*, i.e. the provided curve argument will contain the subdivided curve once the process is finished. To reduce the number of memory allocations during the subdivision process, the `ConicSubdivider` uses double-buffering. At each subdivision step, the subdivision result is stored in `bufferCurve` which is then used as the input to the next subdivision step, effectively swapping the buffers. Before the subdivision process starts, the initial buffer is chosen (and populated) such that the final result always ends up back in the provided `Curve` argument. The capacity of the `bufferCurve` can be maintained between multiple subdivisions so that this buffer does not have to be reallocated. If a non-convex curve is provided as input, the necessary inflection points are inserted according to Equations (9) and (10) prior to starting the subdivision process.

Code 1: Recursive subdivision method

```
void ConicSubdivider::subdivideRecursive(Curve &controlCurve,
                                         Curve &subdivCurve,
                                         const int level) {
    // base case
    if (level == 0) {
        return;
    }
    int n = controlCurve.numPoints() * 2 - 1;
    if (controlCurve.isClosed()) {
        n += 1;
    }
    subdivCurve.getVertices().resize(n);
    subdivCurve.getNormals().resize(n);
    // set old vertex points and normals (cf. Step 1)
    for (int i = 0; i < n; i += 2) {
        subdivCurve.setVertex(i, controlCurve.getVertex(i / 2));
        subdivCurve.setNormal(i, controlCurve.getNormal(i / 2));
    }
    // set new edge points (cf. Step 3)
    for (int i = 1; i < n; i += 2) {
        edgePoint(controlCurve, subdivCurve, i);
    }
    // Update the indices of the inflection points
    for (int &inflIdx: inflPointIndices_) {
        inflIdx *= 2;
    }
    // Double buffering: switch the curves
    subdivideRecursive(subdivCurve, controlCurve, level - 1);
}
```

Subdivision is implemented using recursion as this makes the usage of double-buffering trivial. This can be seen in Code 1. Note that we keep track of the indices of the inflection points to be able to easily extract convex parts. That said, a single subdivision is relatively simple: copy all the old vertex points to a new curve and then insert all the edge points between these old vertices. The `edgePoint` method inserts a new point-normal pair at index `i` by first extracting a patch of point-normal pairs surrounding the edge in question, then fitting a conic to this patch (using `ConicFitter`), and finally sampling a point from this conic (using `Conic::sample`). A shortened version of the `edgePoint` method can be seen in Code 2; cf. Step 3 in Section 1.1. It is important to note that executing `resize` on a `std::vector` will not shrink existing memory allocations [4]. This means that the buffer will only ever grow in size. Consequently, the first full subdivision process on a curve will incur a few allocations on the buffer, but in subsequent executions the buffer will not need re-allocations. The exception to this is when the number of subdivision steps grows or the control curve gains extra vertices.

Code 2: Method for calculating edge point.

```
void ConicSubdivider::edgePoint(const Curve &controlCurve, Curve &subdivCurve, const int i) {
    const int n = subdivCurve.numPoints();
    const int prevIdx = (i - 1 + n) % n;
    const int nextIdx = (i + 1) % n;

    // Construct origin and direction of the ray we use to intersect the found conic
    const Vector2DD origin = (subdivCurve.getVertex(prevIdx) + subdivCurve.getVertex(nextIdx))
     / 2.0;
    Vector2DD dir = subdivCurve.getVertex(prevIdx) - subdivCurve.getVertex(nextIdx);
    // rotate the line segment counterclockwise 90 degree to get the normal of it
    // Note that dir is not normalized! This is to prevent introducing further rounding errors
    .
    // The conic solver finds a multiple of the normal, so the length does not matter
    dir = {dir.y(), -dir.x()};

    // i/2 as we extract the patch from the control curve (while we're currently in the index
    space of the subdiv curve)
    std::vector<PatchPoint> patchPoints = extractPatch(controlCurve, i / 2, settings_.
    patchSize);
    const Conic conic = fitter_.fitConic(patchPoints);
    Vector2DD sampledPoint;
    Vector2DD sampledNormal;
    bool valid = conic.sample(origin, dir, sampledPoint, sampledNormal);
    if (!valid) {
        // The full code tries to expand the neighbourhood here before giving up
        // No valid conic found, set to midpoint and its normal
        sampledPoint = origin;
        sampledNormal = dir;
    }
    subdivCurve.setVertex(i, sampledPoint);
    subdivCurve.setNormal(i, sampledNormal);
}
```

2.2 GUI

The GUI sits on top of the core library and is responsible for creating an interactive visualization of the curves for the user. The GUI is implemented using Qt and the visualisation of the curve(s) is done via OpenGL. To ensure the view is updated when required, the GUI implements the `Listener` interface. Each time the `onListenerUpdated` method is triggered, it will update the curve buffers in OpenGL and render them.

The rendering of the curves is done via OpenGL shaders, where the control curve and the subdivided curve use different shaders. The most important distinction between the two is the geometry shader for the subdivided curve, which is used to draw the curvature combs to visualise the curvature. To do this, we use osculating circles to determine the curvature at each vertex. The idea is as follows: for three consecutive vertices $\mathbf{p}_{i-1}^{(k)}, \mathbf{p}_i^{(k)}, \mathbf{p}_{i+1}^{(k)}$, construct a circle of radius R_i that passes through them. The curvature κ_i at $\mathbf{p}_i^{(k)}$ is then calculated as

$$\kappa_i = \frac{1}{R_i}.$$

The resulting curvature comb is then drawn from $\mathbf{p}_i^{(k)}$ to $\mathbf{p}_i^{(k)} + \lambda_s \cdot \kappa_i \cdot \mathbf{n}_i^{(k)}$, where $\mathbf{n}_i^{(k)}$ is the unit normal of $\mathbf{p}_i^{(k)}$ and λ_s is a global scaling factor to control the final size of the curvature combs. The shader code for finding the circle and returning the corresponding curvature can be seen in Code 3. This example uses `dvec` and `double` as the additional precision is useful at higher subdivision levels.

Code 3: Curvature calculation in shader

```
double calcCurvatureCircular(dvec2 a, dvec2 b, dvec2 c) {
    dvec2 ab = a - b;
    dvec2 cb = c - b;
    dvec2 ac = a - c;

    double denom = (dot(ab, ab) * dot(cb, cb) * dot(ac, ac));
    if (denom == 0.0) return 0.0;
    dvec3 t = cross(dvec3(ab, 0.0), dvec3(cb, 0.0));
    return sqrt(dot(t, t) / denom);
}
```

2.3 Numerical Accuracy

As subdivision is a recursive process, numerical errors are amplified at every subdivision step. As finding the best-fitting conic relies on solving a system of equations using least squares fitting, the method is more sensitive to numerical errors compared to schemes with fixed rules, such as those based on B-splines. The most trivial way to improve precision is to use a high-precision data type. In our framework, we use the data type `real_t` defined as a `long double`. This data type is an extended-precision format [1] equivalent to an 80-bit floating point on (most) x86 processors. It also happens to be the highest-precision data type that the Eigen library [6] supports at the time of writing. While other types such as `__float128` exist, these are not supported by the Eigen library and are typically significantly slower due to lack of hardware support. While using a high-precision data type is the easiest improvement we can make to obtain more accurate results, it is not sufficient when going to higher subdivision levels ($k > 10$). Although most practical scenarios do not require going above $k = 6$ subdivision steps, we use a number of techniques for improving numerical accuracy, allowing the method to produce smooth curves at even higher subdivision levels.

We start by observing that the efforts for improving precision should be focused on the edge point calculations. After all, the vertex points are copied directly from the previous subdivision step without any calculations. As we are using a library for the conic fitting process, the improvements to be done here are limited. However, finding the conic is only one part of the process; once a conic has been found, we need to sample a point-normal pair from it. This process can be divided into two parts: finding the intersection point (Step 3) and finding the normal at said point (Step 4).

The intersection point is found through a ray-conic intersection by solving the quadratic equation described in (8) which can be shortly rewritten as $at^2 + 2bt + c = 0$ where

$$a = \tilde{\mathbf{n}}_i^T \mathbf{Q} \tilde{\mathbf{n}}_i, \quad b = \tilde{\mathbf{n}}_i^T \mathbf{Q} \tilde{\mathbf{m}}_{2i+1}, \quad c = \tilde{\mathbf{m}}_{2i+1}^T \mathbf{Q} \tilde{\mathbf{m}}_{2i+1},$$

with

$$\mathbf{Q} = \begin{bmatrix} q_{20} & q_{11} & q_{10} \\ q_{11} & q_{02} & q_{01} \\ q_{10} & q_{01} & q_{00} \end{bmatrix},$$

and $\tilde{\mathbf{m}}_{2i+1}$ and $\tilde{\mathbf{n}}_i$ denoting $\mathbf{m}_{2i+1}$ and $\mathbf{n}_i^{\text{edge}}$ in 3D homogeneous coordinates, respectively.

In our implementation, owing to the symmetry of the conic coefficient matrix, we thus compute

$$t_{1,2} = \frac{-b \pm \sqrt{b^2 - ac}}{a}. \tag{11}$$

If $a \to 0$, the equation becomes linear and we obtain only one solution: $t = \frac{-c}{2b}$. When there are two solutions, we pick the value closest to 0. Instead of comparing t_1 and t_2 to determine

which is the smallest, it can be deduced from (11) that b can be used to determine whether t_1 or t_2 will be smaller. As such, only one of the values needs to be evaluated and t can be calculated directly using

$$t = \begin{cases} \frac{-b+\sqrt{b^2-ac}}{a}, & \text{if } b < 0, \\ \frac{-b-\sqrt{b^2-ac}}{a}, & \text{otherwise.} \end{cases} \tag{12}$$

The discriminant calculation can suffer from floating point errors. To reduce these errors, we use Kahan's algorithm [7] to calculate $b^2 - ac$. The calculation leverages fused multiply-add (`fmal`) instructions to reduce floating errors that occur when adding a small value to a significantly larger one, ensuring the result remains precise and accurate. The implementation of this can be seen in Code 4. Further error correction techniques, such as those used by the Racket maths library [8], could be implemented to improve the accuracy even further if needed.

Code 4: Implementation of Kahan's method to compute $ab - cd$ with a small error.

```
static real_t diffOfProducts(const real_t a, const real_t b, const real_t c, const real_t d) {
    const real_t w = d * c;
    const real_t e = fmal(-d, c, w);
    const real_t f = fmal(a, b, -w);
    return f + e;
}
```

Once the intersection point is obtained, the normal of the conic is sampled at said point (Code 5). Similar to the intersection calculation, we make use of fused multiply-add instructions to reduce the floating errors. Note that the matrix described in (2) allows for a unique scaling factor for each normal. As such, one final important trick we use here is to skip the normalization of the sampled normal to prevent needlessly introducing additional numerical errors. It turns out that skipping this normal normalization leads to a significant increase in the numerical accuracy of our method as we highlight in Section 3.

Code 5: Calculating the normal of a conic with coefficients `Q_` at a given point `p`.

```
Vector2DD Conic::conicNormal(const Vector2DD &p) const {
    real_t xn = fmal(Q_(0, 0), p.x(), fmal(Q_(0, 1), p.y(), Q_(0, 2)));
    real_t yn = fmal(Q_(1, 0), p.x(), fmal(Q_(1, 1), p.y(), Q_(1, 2)));
    return {xn, yn};
}
```

3 Code usage and numerical tests

In this section, we show how to use the C++ package CONIS (Section 3.1) and present several numerical tests and results, focusing on conic reproduction (Section 3.2) and highlighting the impact of the techniques described in Section 2.3.

Note that the figures in the original paper were constructed by exporting the subdivision curves from the CONIS framework and visualizing them using MATLAB. All the curve figures in this paper were generated using the CONIS framework directly.

3.1 Code Usage

As the CONIS framework has a clear separation between the core library and the GUI, it is relatively straightforward to set up a short demo program using just the core library. For example, Code 6 demonstrates a simple method for loading a curve from a file, subdividing it a few times, and writing it back to a different file. The input file format follows a structure similar

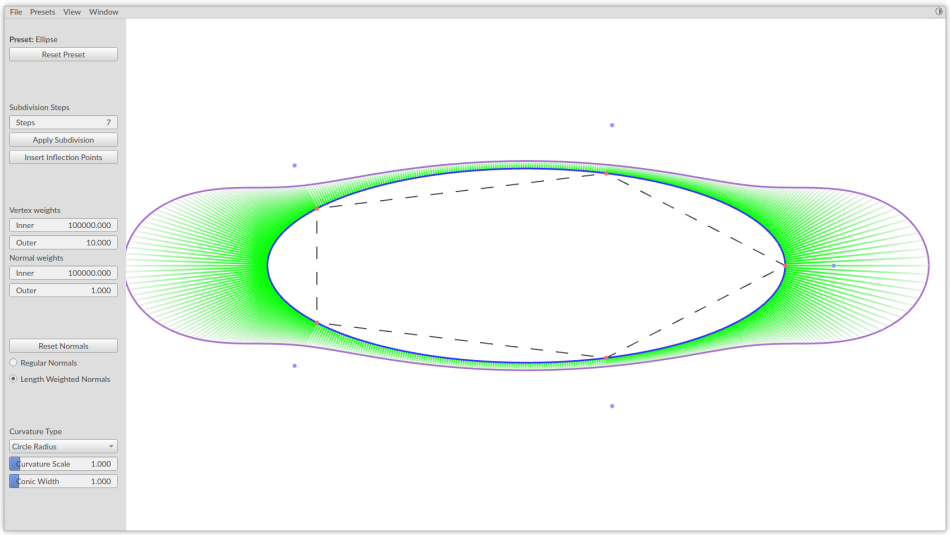


Figure 2: GUI of the CONIS framework.

to wavefront `.obj` files. It is simply an ordered collection of vertices optionally followed by another ordered collection of the associated normals. Each vertex and normal entry resides on a separate line and has the format `<prefix> x y`. For vertices the prefix `v` is used, while for normals the prefix `vn` is used. For the output, the same file format can be used, or it is also possible to only output an ordered list of vertices without any prefix.

Code 6: Basic usage of the CONIS core library.

```
using namespace conis::core

void demo(std::string& inputPath, std::string& outputPath, int subdivLevels) {
    // Setup
    SubdivisionSettings subdivSettings; // This will use the default settings
    ConicSubdivider subdivider(subdivSettings);
    CurveLoader loader;
    CurveSaver saver;

    // Load, subdivide and save
    Curve curve = loader.loadCurveFromFile(inputPath);
    subdivider.subdivide(curve, subdivLevels);
    saver.saveCurveWithNormals(outputPath, curve);
}
```

To get the full framework including the GUI up and running, there is a script called `build.sh` to allow for easy compilation, testing and running of the application. For example, running `./build.sh --run --test` compiles the program, runs the unit tests and starts the full application. The GUI comes with a number of knobs and dials to test various settings for the subdivision process, this can be seen in Figure 2. Additional test settings are available in the GUI when the program is compiled in debug mode.

3.2 Numerical tests and conic reproduction

We assess the numerical precision of the method in two ways. First, the smoothness of a subdivided curve can be visually assessed through the usage of curvature combs. Second, we present a series of tests implemented using GoogleTest [5] that (1) sample point/normals from a given conic, (2) construct a subdivided curve using our method, and (3) calculate the error between the points on the subdivided curve and the original conic. As conic sections

Conic	Error Metric	$k=4$	$k=8$	$k=12$
$x^2+y^2=25$	$E^{(k)}$	$1.97\cdot10^{-18}$	$4.70\cdot10^{-17}$	$1.57\cdot10^{-14}$
	$E^{(k)}_{\max}$	$8.67\cdot10^{-18}$	$1.35\cdot10^{-15}$	$1.04\cdot10^{-13}$
$4x^2+9y^2=36$	$E^{(k)}$	$1.37\cdot10^{-18}$	$6.54\cdot10^{-17}$	$3.46\cdot10^{-15}$
	$E^{(k)}_{\max}$	$6.17\cdot10^{-18}$	$1.10\cdot10^{-15}$	$1.53\cdot10^{-13}$
$4x^2-y^2=4$	$E^{(k)}$	$1.14\cdot10^{-17}$	$4.67\cdot10^{-15}$	$4.55\cdot10^{-13}$
	$E^{(k)}_{\max}$	$5.97\cdot10^{-16}$	$3.54\cdot10^{-13}$	$2.31\cdot10^{-12}$
$y=x^2$	$E^{(k)}$	$3.54\cdot10^{-16}$	$4.05\cdot10^{-15}$	$3.83\cdot10^{-13}$
	$E^{(k)}_{\max}$	$3.53\cdot10^{-15}$	$1.13\cdot10^{-12}$	$8.09\cdot10^{-12}$

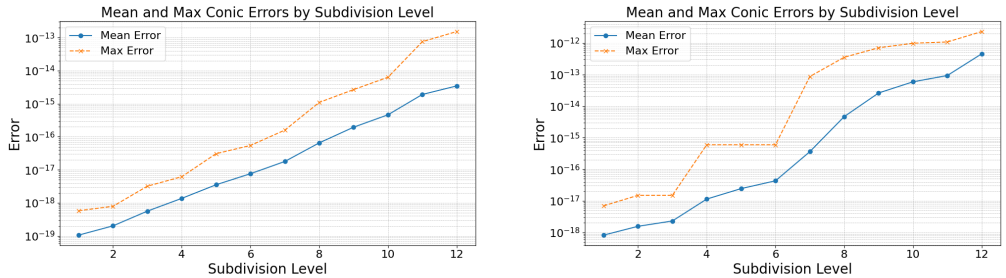
Table 1: Conic reproduction error for a circle, ellipse, hyperbola and parabola at increasing subdivision levels k.

are implicit functions of the form $f(x,y)=0$ (Equation 3), we can use the output of $f(x,y)$ to construct an error function. By taking the absolute value and normalizing the resulting value, we construct an error metric that is independent of the curve density and is proportional to the defined conic. It should be noted that this error metric is only meaningful when all control point-normal pairs are sampled from a single conic. For a curve at a given subdivision level k, we construct what is essentially the mean absolute error over n_k samples (vertices $\mathbf{p}_i^{(k)}=[x_i,y_i]^T$, $i\in I_k$ of the subdivided curve):

$$E^{(k)}=\frac{1}{n_k}\sum_{i=1}^{n_k}\left|\frac{f(x_i,y_i)}{\nabla f(x_i,y_i)}\right|,\quad n_k=|I_k|. \tag{13}$$

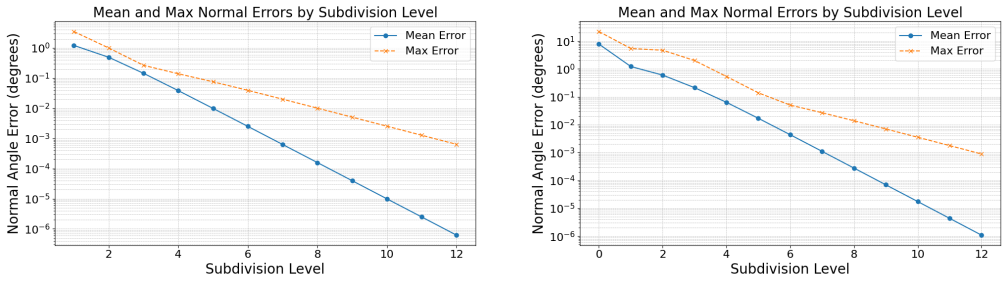
Similarly, we also define the maximum absolute error to catch any outliers:

$$E^{(k)}_{\max}=\max_{i=1}^{n_k}\left|\frac{f(x_i,y_i)}{\nabla f(x_i,y_i)}\right|,\quad n_k=|I_k|. \tag{14}$$



(a) $4x^2+9y^2=36$ (b) $4x^2-y^2=4$

Figure 3: Curve precision error $E^{(k)}$ and $E^{(k)}_{\max}$ obtained using **FINAL** for increasing subdivision levels up to $k=12$ on an ellipse (a) and hyperbola (b).



(a) $4x^2 + 9y^2 = 36$ (b) $4x^2 - y^2 = 4$

Figure 4: Mean and maximum normal angle error (in degrees) obtained using **FINAL** for increasing subdivision levels up to $k = 12$ on an ellipse (a) and hyperbola (b).

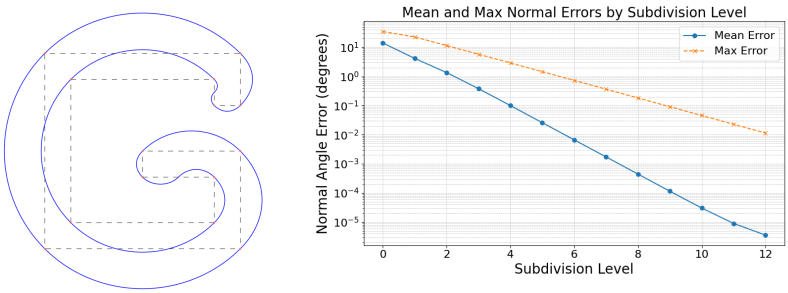
Using these metrics, we investigate the behaviour of the three different implementations:

- the baseline implementation **BASE** using `doubles` and no accuracy improvements,
- **NORM_SKIP** which is the same as the baseline but without the normalization of the normals,
- **FINAL** which uses `long doubles` and sampling accuracy improvements on top of **NORM_SKIP**.

The baseline implementation uses `doubles` instead of `floats` as the latter are too inaccurate to make meaningful comparisons. In Table 1, the results of **FINAL** with four different conic types show that even at high subdivision levels, the error between the original and reproduced conic remains extremely small. In Figure 3, we can see in more detail how $E^{(k)}$ and $E^{(k)}_{\max}$ evolve as the number k of subdivision steps grows. As expected, the error grows exponentially as the subdivision levels increase, but both $E^{(k)}$ and $E^{(k)}_{\max}$ remain reasonably low even for $k = 12$.

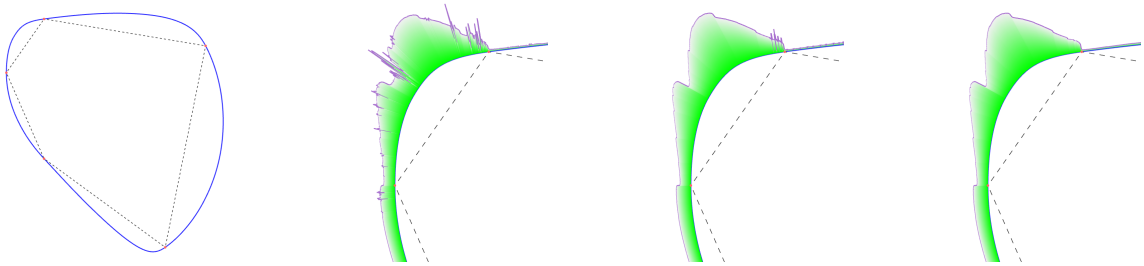
In Step 1 of Section 1.1, the normals are copied between subdivision steps. This could lead to a situation where the final normals of the subdivided curve do not accurately represent the underlying geometry. To test this, we use the same curves used for Table 1 and compute the angle differences between the normals obtained from the subdivision scheme (namely $\mathbf{n}^{(k)}_i$, $i \in I_k$) and the normals calculated based on the curve geometry (i.e., estimated from the points $\mathbf{p}^{(k)}_i$, $i \in I_k$). From Figure 4 it can be seen that both of these errors are relatively small and decrease further as k increases. For a more arbitrary non-convex curve, Figure 5 shows similar results. It should be noted that it is unlikely for the error to be exactly 0 as this depends on how one calculates the “true" normals of a curve. This means that for sufficiently large values of k, the normals of the control points are equivalent to the normals of the subdivided curve, demonstrating that our scheme interpolates the input normals in the limit.

Finally, Figure 6 shows the behaviour of the different implementations using curvature combs. These results were obtained from the curve seen in Figure 6 (a) using a subdivision level of $k = 10$. All of the different techniques to reduce numerical errors contribute to obtaining a smooth subdivided curve, but the most impactful optimisation is the removal of the normal normalization. Interestingly, when all points lie on a single conic section, enabling the normal normalization can improve the accuracy. **FINAL** with normalization then obtains e.g. $E^{(k)} = 3.47 \cdot 10^{-16}$ and $E^{(k)}_{\max} = 1.37 \cdot 10^{-14}$ for the parabola $y = x^2$ at $k = 12$. However, for any more general curve, it is evident that the normalization step is crucial for obtaining a smooth curve. The remaining fluctuations in the curvature combs are the result of how the scheme



(a) (b)

Figure 5: Mean and maximum normal angle error (in degrees) obtained using **FINAL** for increasing subdivision levels up to $k = 12$ on a non-convex curve. (a) Reference control polygon and subdivision curve. (b) Normal angle error plot.



(a) (b) (c) (d)

Figure 6: Smoothness of the subdivided curves for the various implementations shown through curvature combs. (a) Reference control polygon and subdivision curve. (b) **BASE** implementation at $k = 10$. (c) **NORM_SKIP** implementation at $k = 10$. (d) **FINAL** implementation at $k = 10$.

works on this particular set of point-normal pairs and not of numerical accuracy problems. As such, despite the method being very sensitive to numerical errors, the CONIS implementation is capable of producing highly accurate results even at higher subdivision levels.

4 Conclusions

We have provided an overview of the CONIS framework, the software used to implement the conic-preserving interpolatory subdivision scheme presented in [3]. The framework was written in C++ using Eigen, Qt, and OpenGL. The framework has a simple decoupled design consisting of a stand-alone core library, with a GUI sitting on top. Due to the approximating nature of the conic fitting process, the implementation uses several techniques to reduce numerical errors. Through a number of numerical tests and demos, we show that the implementation produces smooth curves up to high ($k > 10$) subdivision levels. Additionally, we provide numerical evidence demonstrating that the scheme interpolates normals for sufficiently high values of k. A possible future improvement to the CONIS framework would be to extend the existing unit test suites and use this to improve the robustness of the implementation.

Acknowledgments

Lucia Romani is a member of GNCS-INdAM (Gruppo Nazionale Calcolo Scientifico - Istituto Nazionale di Alta Matematica). Her research has been done within the Italian Network on Approximation and the thematic group on "Approximation Theory and Applications" of the Italian Mathematical Union.

Declaration of interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754-2008, (2008), 1–70. URL http://dx.doi.org/10.1109/IEEESTD.2008.4610935

[2] G. Albrecht and L. Romani. *Convexity preserving interpolatory subdivision with conic precision*. Applied Mathematics and Computation, vol. 219, 8, (2012), 4049–4066. ISSN 0096-3003. URL http://dx.doi.org/https://doi.org/10.1016/j.amc.2012.10.048

[3] N. Bügel, L. Romani, and J. Kosinka. *A point-normal interpolatory subdivision scheme preserving conics*. Computer Aided Geometric Design, vol. 111, (2024), 102347. ISSN 0167-8396. URL http://dx.doi.org/10.1016/j.cagd.2024.102347

[4] cppreference.com. *std::vector<>::resize*. https://en.cppreference.com/w/cpp/container/vector/resize. Accessed November 23, 2024

[5] Google LLC. *GoogleTest*. https://github.com/google/googletest. Accessed November 23, 2024

[6] G. Guennebaud, B. Jacob et al. *Customizing Eigen: Custom Scalar Types*. https://eigen.tuxfamily.org/dox/TopicCustomizing_CustomScalar.html. Accessed November 23, 2024

[7] C.-P. Jeannerod, N. Louvet, and J.-M. Muller. *Further analysis of Kahan's algorithm for the accurate computation of 2 x 2 determinants*. Mathematics of Computation, vol. 82. URL http://dx.doi.org/10.1090/S0025-5718-2013-02679-8

[8] N. Toronto, J. A. Søgaard et al. *Racket Math Library*. https://docs.racket-lang.org/math/. Accessed November 23, 2024