Approximations to Rotation Matrices and Their Derivatives

David Eberly, Geometric Tools, Redmond WA 98052
https://www.geometrictools.com/

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Created: August 12, 2020

Contents

1 An Equation for a Rotation Matrix **3**

2 Equations for the Derivatives of the Rotation Matrix **3**

3 The Singularity at Zero **4**

4 Floating-Point Issues in Function Evaluation **5**

4.1 Evaluation of $\sin(t)$ 6

4.2 Evaluation of $\cos(t)$ 6

4.3 Evaluation of $\alpha(t)$ 7

4.4 Evaluation of $\beta(t)$ 7

4.5 Evaluation of $\gamma(t)$ 11

4.6 Evaluation of $\delta(t)$ 14

5 Minimax Approximations **16**

5.1 Remez Algorithm 17

5.1.1 Chebyshev Nodes 17

5.1.2 The Linear System to Determine Coefficients 17

5.1.3 Newton Polynomials 17

5.1.4 Solving the Linear System 19

5.1.5 Updating the t-Nodes 19

5.1.6 Source Code 20

5.2 Modified Remez Algorithm . . . 22
5.2.1 Modifications that Work Generally . . . 22
5.2.2 Modifications for the Rotation-Coefficient Functions . . . 23
5.2.3 Computing the Exact Sign of a Power Series . . . 25
5.2.4 Source Code . . . 30
6 Polynomial Coefficients **30**

1 An Equation for a Rotation Matrix

A 3×3 rotation matrix R can be represented by $R = \exp(S)$ for a skew-symmetric matrix

$$S = \begin{bmatrix} 0 & -s_2 & s_1 \\ s_2 & 0 & -s_0 \\ -s_1 & s_0 & 0 \end{bmatrix} = \mathrm{Skew}(\mathbf{s}) \tag{1}$$

where the right-most equality defines the function $\mathrm{Skew}(\mathbf{s})$ with $\mathbf{s} = (s_0, s_1, s_2)$. This is a 3-parameter representation. When $\mathbf{s} = (0, 0, 0)$, the rotation matrix is the identity matrix I. Define

$$t = |\mathbf{s}| = \sqrt{s_0^2 + s_1^2 + s_2^2}, \quad \alpha(t) = \frac{\sin(t)}{t}, \quad \beta(t) = \frac{1 - \cos(t)}{t^2} \tag{2}$$

for $\mathbf{s} \neq \mathbf{0}$. The rotation matrix is

$$R(\mathbf{s}) = I + \alpha(t)S + \beta(t)S^2 = \begin{bmatrix} 1 - \beta(s_1^2 + s_2^2) & -\alpha s_2 + \beta s_0 s_1 & +\alpha s_1 + \beta s_0 s_2 \\ +\alpha s_2 + \beta s_0 s_1 & 1 - \beta(s_0^2 + s_2^2) & -\alpha s_0 + \beta s_1 s_2 \\ -\alpha s_1 + \beta s_0 s_2 & +\alpha s_0 + \beta s_1 s_2 & 1 - \beta(s_0^2 + s_1^2) \end{bmatrix} \tag{3}$$

where the notation $R(\mathbf{s})$ indicates that the rotation matrix is parameterized by the components of $\mathbf{s}$. Note that when $\mathbf{s} \neq 0$, a unit-length rotation axis is $\mathbf{u} = \mathbf{s}/t$. In this case define $U = \mathrm{Skew}(\mathbf{u})$; the rotation matrix is provided by the more common equation $R = I + \sin(t)U + (1 - \cos(t))U^2$. The form involving S is typically encountered when using Lie groups and Lie algebras [1]. It is also useful for computing derivatives of rotation matrices in optimization problems where the functions have rotation parameters.

2 Equations for the Derivatives of the Rotation Matrix

Define

$$\gamma(t) = \frac{-\alpha'(t)}{t} = \frac{\sin(t) - t\cos(t)}{t^3}, \quad \delta(t) = \frac{-\beta'(t)}{t} = \frac{2(1 - \cos(t)) - t\sin(t)}{t^4} \tag{4}$$

The minus signs in front of $\alpha'(t)$ and $\beta'(t)$ are chosen so that $\alpha(t)$, $\beta(t)$, $\gamma(t)$ and $\delta(t)$ have removable singularities that are all positive numbers and $\alpha'(t)/t$, $\beta'(t)/t$, $\gamma'(t)/t$ and $\delta'(t)/t$ have removable singularities at $t = 0$ which are all negative. The first-order partial derivative of t, $\alpha(t)$ and $\beta(t)$ with respect to s_k are

$$\frac{\partial t}{\partial s_k} = \frac{s_k}{t}, \quad \frac{\partial \alpha(t)}{\partial s_k} = \alpha'(t)\frac{\partial t}{\partial s_k} = -s_k\gamma(t), \quad \frac{\partial \beta(t)}{\partial s_k} = \beta'(t)\frac{\partial t}{\partial s_k} = -s_k\delta(t) \tag{5}$$

The first-order partial derivative of S with respect to s_k is the matrix E_k; these are

$$E_0 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}, \quad E_1 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix}, \quad E_2 = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{6}$$

The first-order partial derivative of $R(\mathbf{s})$ with respect to s_k is

$$\begin{aligned} \frac{\partial R}{\partial s_k} &= \alpha \frac{\partial S}{\partial s_k} + \frac{\partial \alpha}{\partial s_k} S + \beta \left(S \frac{\partial S}{\partial s_k} + \frac{\partial S}{\partial s_k} S \right) + \frac{\partial \beta}{\partial s_k} S^2 \\ &= \alpha E_k + \beta \left(S E_k + E_k S \right) - s_k \left(\gamma S + \delta S^2 \right) \end{aligned} \tag{7}$$

These are

$$\frac{\partial R}{\partial s_0} = \begin{bmatrix} \delta s_0 \left(s_1^2 + s_2^2 \right) & \beta s_1 + s_0(\gamma s_2 - \delta s_0 s_1) & \beta s_2 - s_0(\gamma s_1 + \delta s_0 s_2) \\ \beta s_1 - s_0(\gamma s_2 + \delta s_0 s_1) & \delta s_0 \left(s_0^2 + s_2^2 \right) - 2\beta s_0 & s_0(\gamma s_0 - \delta s_1 s_2) - \alpha \\ s_2 \left(\beta - \delta s_0^2 \right) + \gamma s_0 s_1 & \alpha - s_0(\gamma s_0 + \delta s_1 s_2) & \delta s_0 \left(s_0^2 + s_1^2 \right) - 2\beta s_0 \end{bmatrix} \tag{8}$$

and

$$\frac{\partial R}{\partial s_1} = \begin{bmatrix} \delta s_1 \left(s_1^2 + s_2^2 \right) - 2\beta s_1 & \beta s_0 + s_1(\gamma s_2 - \delta s_0 s_1) & \alpha - s_1(\gamma s_1 + \delta s_0 s_2) \\ \beta s_0 - s_1(\gamma s_2 + \delta s_0 s_1) & \delta s_1 \left(s_0^2 + s_2^2 \right) & s_2 \left(\beta - \delta s_1^2 \right) + \gamma s_0 s_1 \\ s_1(\gamma s_1 - \delta s_0 s_2) - \alpha & \beta s_2 - s_1(\gamma s_0 + \delta s_1 s_2) & \delta s_1 \left(s_0^2 + s_1^2 \right) - 2\beta s_1 \end{bmatrix} \tag{9}$$

and

$$\frac{\partial R}{\partial s_2} = \begin{bmatrix} \delta s_2 \left(s_1^2 + s_2^2 \right) - 2\beta s_2 & s_2(\gamma s_2 - \delta s_0 s_1) - \alpha & \beta s_0 - s_2(\gamma s_1 + \delta s_0 s_2) \\ \alpha - s_2(\gamma s_2 + \delta s_0 s_1) & \delta s_2 \left(s_0^2 + s_2^2 \right) - 2\beta s_2 & \beta s_1 + s_2(\gamma s_0 - \delta s_1 s_2) \\ \beta s_0 + s_2(\gamma s_1 - \delta s_0 s_2) & \beta s_1 - s_2(\gamma s_0 + \delta s_1 s_2) & \delta s_2 \left(s_0^2 + s_1^2 \right) \end{bmatrix} \tag{10}$$

3 The Singularity at Zero

The functions $\alpha(t)$, $\beta(t)$, $\gamma(t)$ and $\delta(t)$ have removable singularities at $t = 0$, easily seen using the Maclaurin series for the functions,

$$\begin{aligned} \alpha(t) &= \textstyle\sum_{i=0}^{\infty} (-1)^i \frac{1}{(2i+1)!} t^{2i}, & \beta(t) &= \textstyle\sum_{i=0}^{\infty} (-1)^i \frac{1}{(2i+2)!} t^{2i} \\ \gamma(t) &= \textstyle\sum_{i=0}^{\infty} (-1)^i \frac{2(i+1)}{(2i+3)!} t^{2i}, & \delta(t) &= \textstyle\sum_{i=0}^{\infty} (-1)^i \frac{2(i+1)}{(2i+4)!} t^{2i} \end{aligned} \tag{11}$$

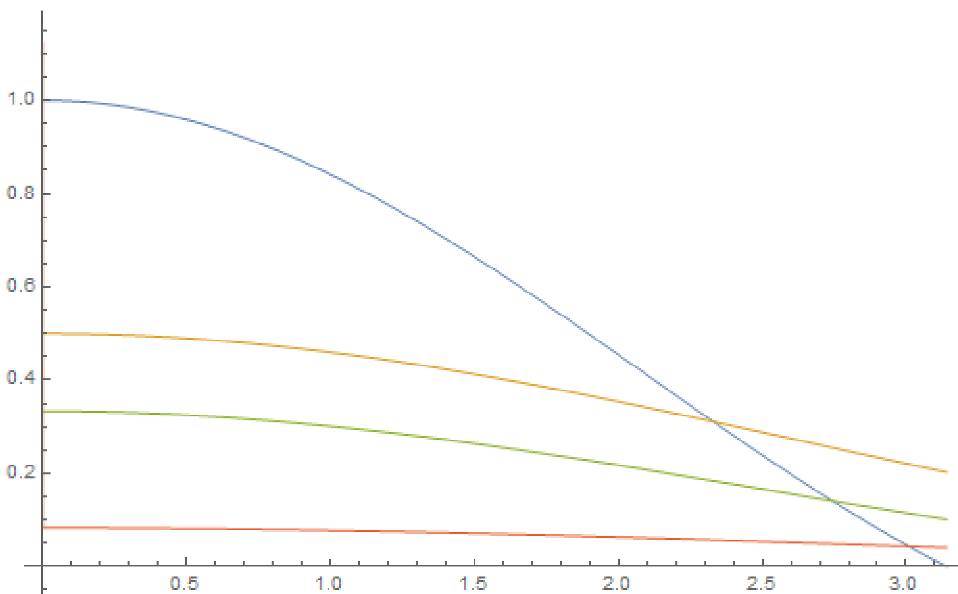
All series have only even powers of t. Evaluating the series at $t = 0$ produces $\alpha(0) = 1$, $\beta(0) = 1/2$, $\gamma(0) = 1/3$ and $\delta(0) = 1/12$. The derivatives of the functions divided by t will be used in computing approximating polynomials. The series for these are listed next, each having only odd powers of t,

$$\begin{aligned} \alpha'(t)/t &= \textstyle\sum_{i=0}^{\infty} (-1)^{i+1} \frac{2(i+1)}{(2i+3)!} t^{2i}, & \beta'(t)/t &= \textstyle\sum_{i=0}^{\infty} (-1)^{i+1} \frac{2(i+1)}{(2i+4)!} t^{2i} \\ \gamma'(t)/t &= \textstyle\sum_{i=0}^{\infty} (-1)^{i+1} \frac{4(i+1)(i+2)}{(2i+5)!} t^{2i}, & \delta'(t)/t &= \textstyle\sum_{i=0}^{\infty} (-1)^{i+1} \frac{4(i+1)(i+2)}{(2i+6)!} t^{2i} \end{aligned} \tag{12}$$

The ratios have removable singularities at $t = 0$. The values are $-1/3$, $-1/12$, $-1/15$ and $-1/90$, respectively.

The graphs of the four function is shown in Figure 1.

Figure 1. The graphs of $\alpha(t)$ [blue], $\beta(t)$ [yellow], $\gamma(t)$ [green] and $\delta(t)$ [red] for $t \in [0, \pi]$. The plots were drawn with Mathematica [8].



When computing with floating-point arithmetic, the functions can be evaluated directly using equations (2) and (4). The C++ Standard Library functions `std::sin` and `std::cos` are called in the evaluation. However, the singularity at zero leads to some unexpected floating-point behavior. This is the topic of the next section.

Alternatively, the functions can be evaluated using Maclaurin polynomials. If rational arithmetic is used to obtain a precision larger than that of `double`. For t near π, the degree of the Maclaurin polynomials will be so large that the performance suffers from the extremely large number of bits required to represent the polynomial exactly.

Finally, minimax polynomial approximations can be used for a rotation estimate that can be computed rapidly. For high-degree approximations, the floating-point computations to producing the polynomial coefficients is significant enough to cause accuracy problems. The algorithm uses bisection to find roots of functions and derivatives, where the functions have local extrema nearly zero. Bisection is a common topic covered in a course on numerical methods, but it is usually not pointed out that it depends strongly on knowing the exact sign of the function at the various t-values. At first glance, this appears to be an impossible task for functions defined as Maclaurin series. It turns out it is possible for non-zero series with alternating signs and terms that decrease in magnitude. In the last section, I discuss such an approach that uses rational arithmetic to determine the exact signs of power series expressions.

4 Floating-Point Issues in Function Evaluation

When computing the direct equations $\alpha(t)$, $\beta(t)$, $\gamma(t)$ or $\delta(t)$ with floating-point arithmetic, numerical problems invariably occur for values t near 0. To simplify the discussion, only $t \geq 0$ is considered.

The typical mathematical suggestion is to use a Taylor polynomial when t is nearly 0. The degree must be specified and the t-cutoff, say, $\hat{t}$, must be specified. The direct implementation is evaluated when $t > \hat{t}$ and the Taylor polynomial is evaluated when $0 \le t \le \hat{t}$. Moreover, a numerical analysis must be provided to understand the error bounds for the approximations. As it turns out, Taylor polynomials can be avoided $\alpha(t)$ and $\beta(t)$. For $\gamma(t)$ and $\delta(t)$, a Taylor polynomial or other approximating polynomial can be used. Implementations of the functions can be made robust for $t > 0$ when computing with floating-point arithmetic.

The experiments mentioned here are for float numbers. Similar experiments can be performed using double with the same floating-point issues that occur with float. For testing reproducibility, the unsigned integer encodings for the t-values are listed.

4.1 Evaluation of $\sin(t)$

As a real-valued function of a real variable $t \in [0, \pi/2]$, $\sin(t)$ is strictly increasing because its derivative is $\cos(t)$ which is negative for $t \in [0, \pi/2)$. Similarly, for $t \in [\pi/2, \pi]$, $\sin(t)$ is strictly decreasing.

As a float-valued function of a float variable t, std::sin(t) is strictly increasing for $t \in [0, t_0]$ where $t_0 = 4.43632947 * 10^{-4}$ (0x39e89768), nondecreasing for $t \in [t_0, t_1]$ where $t_1 = 1.57079637 \doteq \pi/2$ (0x3fc90fdb), nonincreasing for $t \in [t_1, t_2]$ where $t_2 = 1.99999952$ (03ffffffc) and strictly decreasing for $t \in [t_2, t_3]$ where $t_3 = 3.14159274 \doteq \pi$ (0x40490fdb). The function is piecewise constant on $[0, \pi]$ with 1,064,929,526 pieces on $[0, \pi/2]$ and 6,309,924 pieces on $[\pi/2, \pi]$ for a total of 1,071,239,449 pieces. The last piece of the first interval and the first piece of the second interval both have function value 1, so the pieces merge into one piece.

The std::sin(t) function is one of the functions recommended by the IEEE Standard 754™-2008 [3] to be *correctly rounded*. To say a function $z = f(t)$ is correctly rounded means that an input floating-point number t leads to an output floating-point number z that is the closest floating-point number to the theoretical result, where the definition of *closest* depends on the specified rounding mode. Even though std::sin(t) is piecewise constant, the function values are correctly rounded.

Observe that the domain of std::sin(t) for $t \in [0, \pi)$ has 1,078,530,011 (0x40490fdb) float numbers. The interval $[0, \pi/2)$ has 1,070,141,403 (0x3fc90fdb) float numbers, a count that is much larger than that of the count $8{,}388{,}608 = 2^{23}$ (0x00800000) of float numbers in the interval $[\pi/2, \pi)$. Moreover, $[0, \pi/2)$ contains zero and the subnormal numbers, a total of 2^{24} numbers, which is twice as many float numbers as in the interval $[\pi/2, \pi)$. If you need a dense set of float t-values to evaluate std::sin(t), consider reduction to the interval $t \in [0, \pi/2]$ using $\sin(t) = \sin(\pi - t)$ for $t \in [\pi/2, \pi]$.

4.2 Evaluation of $\cos(t)$

As a real-valued function of a real variable $t \in [0, \pi]$, $\cos(t)$ is strictly decreasing because its derivative is $-\sin(t)$ which is negative for $t \in (0, \pi)$.

As a float-valued function of a float variable t, std::cos(t) is nonincreasing for $t \in [0, t_0]$ where $t_0 = 0.999999642$ (0x3f7ffffa), strictly decreasing for $t \in [t_0, t_1]$ where $t_1 = 2.88907409$ (0x4038e697) and nonincreasing for $t \in [t_1, t_2]$ where $t_2 = 3.14159274 \doteq \pi$ (0x40490fdb). The function is piecewise constant on $[0, \pi/2]$ with 12,500,635 pieces and 7,861,537 pieces on $[\pi/2, \pi]$ for a total of 20,362,171 pieces. The last piece of the first interval and the first piece of the second interval both have function value $-4.37113883 * 10^{-8}$, so the pieces merge into one piece.

The std::cos(t) function is one of the functions recommended by the IEEE Standard 754™-2008 [3] to be *correctly rounded*. Even though std::cos(t) is piecewise constant, the function values are correctly rounded.

The density of the floating-point numbers in $[0, \pi]$ was mentioned in the previous section for std::sin(t). The same idea applies here. If you need a dense set of float t-values to evaluate std::cos(t), consider reduction to the interval $t \in [0, \pi/2]$ using $\cos(\pi - t) = -\cos(t)$ for $t \in [\pi/2, \pi]$.

4.3 Evaluation of $\alpha(t)$

As a real-valued function of a real-valued variable t, $\alpha(t)$ is strictly decreasing for $t \in [0, \pi]$. The derivative is $\alpha'(t) = (t\cos(t) - \sin(t))/t^2$. Sketching the graphs of $\tan(t)$ and t for $t \in (0, \pi/2)$, one can see that $\tan(t) > t$ in which case $\alpha'(t) < 0$ on the interval. Similarly, one can see that for $t \in (\pi/2, \pi)$, $t > \tan(t)$ in which case $\alpha'(t) < 0$ on the interval. Finally, $\alpha'(\pi/2) = -2/\pi^2 < 0$. Therefore, $\alpha'(t) < 0$ for all $t \in (0, \pi)$ which implies $\alpha(t)$ is indeed strictly decreasing.

The direct implementation for $\alpha(t)$ is shown in Listing 1 for type float.

Listing 1. The direct implementation of $\alpha(t)$ for $t \in [0, \pi]$ for type float.

```
float Alpha(float t)
{
    if (t > 0.0f) { return std::sin(t) / t; }
    else { return 1.0f; }
}
```

The function Alpha(t) is the floating-point version of $\alpha(t)$. It is piecewise constant with 14,012,441 pieces. The function is strictly decreasing on $[t_0, \pi]$, where $t_0 = 1.99999809$ (0x3ffffff0) and Alpha(t0) $= 0.454649568$. On the interval $[0, t_0]$, the differences in function values between consecutive pieces are $5.96046448 * 10^{-8}$ (occurring 237480 times), $5.96046448 * 10^{-8}$ (occurring 3802736 times), $-8.94069672 * 10^{-8}$ (occurring 276 times) and $-1.19209290 * 10^{-7}$ (occurring 684802 times). The maximum absolute error is $1.19209290 * 10^{-7}$.

The floating-point function S(t) = std::sin(t) and I(t) = t (the identity function) are both correctly rounded functions. However, the ratio Alpha(t) = S(t)/I(t) is not a correctly rounded function. Regardless, the floating-point behavior of Alpha(t) is acceptable because of the small errors at the jumps.

4.4 Evaluation of $\beta(t)$

The direct implementation of $\beta(t)$ is shown in Listing 2 for type float.

Listing 2. The direct implementation of $\beta(t)$ for $t \in [0, \pi]$ are not robust when computing using floating-point arithmetic.

```
float Beta(float t)
{
    if (t > 0.0f) { return (1.0f - std::cos(t)) / t / t; }
    else { return 0.5f; }
}
```

The reason to use two divisions by t is to avoid not-a-number (NaN) when t values are such that t^2 is less than the minimum subnormal and is rounded to 0. For such t values, the numerator $1 - \cos(t)$ evaluates to floating-point 0, so $(1 - \cos(t))/t^2$ is an indeterminate of the form 0/0 which is displayed as `-nan(ind)`.

The direct implementation has multiple problems when computing with floating-point arithmetic. Experiments were performed using Microsoft Visual Studio 2019 version 16.6.5. However, any IEEE-compliant implementation for `std::cos` will produce the same results, so any compiler should generate the same results. In the experiments, the default rounding mode is used: round-to-nearest-ties-to-even.

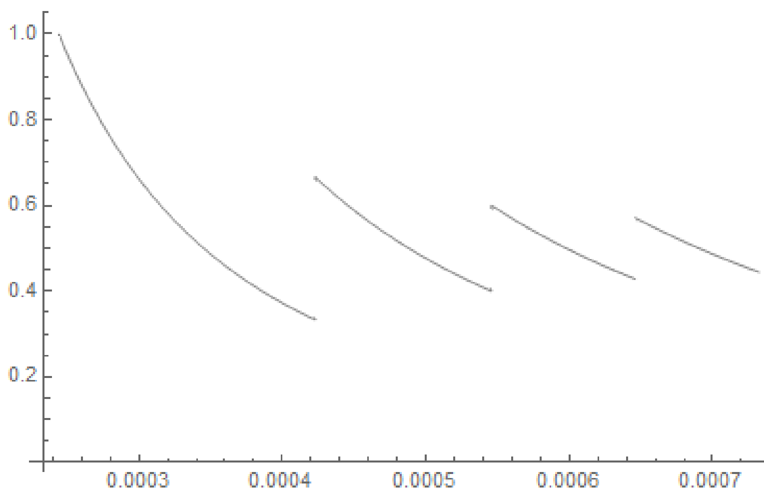
Table 1 shows evaluation results for the `float` version of $\beta(t)$.

Table 1. Evaluations of the `float` version of $\beta(t)$ from Listing 2.

	t input	encoding	$\beta(t)$ output
#1	0	0x00000000	0.5
#2	$[2^{-149}, 2.44140625 * 10^{-4}]$	[0x00000001, 0x39800000]	0
#3	$[2.44140654 * 10^{-4}, 4.22863959 * 10^{-4}]$	[0x39800001, 0x39ddb3d7]	0.999999762 ↘ 0.333333343
#4	$[4.22863988 * 10^{-4}, 5.45914925 * 10^{-4}]$	[0x39ddb3d8, 0x3a0f1bbc]	0.666666627 ↘ 0.400000066
#5	$[5.45915042 * 10^{-4}, 6.45935361 * 10^{-4}]$	[0x3a0f1bbd, 0x3a2953fd]	0.600000024 ↘ 0.428571463
#6	$[6.45935419 * 10^{-4}, 7.32421817 * 10^{-4}]$	[0x3a2953fe, 0x3a400000]	0.571428478 ↘ 0.444444448

The graph of $\beta(t)$ for $t \in [0, \pi]$ is shown in Figure 1. Although the graph appears to show a smooth function, the samples are sparse enough that the floating-point behavior for t near 0 are not evident. The graph of $\beta(t)$, computed using `float` arithmetic for small t, is shown in Figure 2.

Figure 2. The graph of $\beta(t)$ for small t. The plot was drawn by choosing 8193 samples of $\beta(t)$ for $t \in [2.44141 * 10^{-4}, 7.32421875 * 10^{-4}]$ using the `float` implementation and then allowing Mathematica [8] to draw a high-resolution polyline connecting the samples.



At this resolution for t near 0, the floating-point behavior is evident.

The t-inputs in row #2 of Table 1 lead to $\beta(t) = 0$ because the numerator still evaluates to 0.0f but t * t evaluates to a positive floating-point number, in which case the division is of the form $0/p$ for $p > 0$ and the result is 0.

The t-inputs in row #3 of Table 1 lead to $\beta(t)$ decreasing from 0.999999762 (approximately 1) at the left t-endpoint to 0.333333343 (approximately 1/3) at the right t-endpoint. This behavior is certainly not expected, because $\beta(t)$ has a theoretical maximum of 1/2. The reason for the behavior is that the floating-point output of the numerator $1 - \cos(t)$ is $5.96046448 * 10^{-8}$ (0x33880000) for the entire set of t-inputs of row #2; that is, the function is effectively of the form K_1/t^2 for a positive constant K_1. The actual value of $1 - \cos(t)$ for the encoding 0x39800001 is approximately twice that of the floating-point value, which is why an expected number of approximately 1/2 becomes approximately 1.

A discontinuity appears where $\beta(t)$ jumps from (approximately) 1/3 to (approximately) 2/3 at the t-values mentioned in row #4 of Table 1. This happens because the floating-point output of the numerator $1 - \cos(t)$ decreases from the previous $5.96046448 * 10^{-8}$ at $t = 4.22863959 * 10^{-4}$ (0x39ddb3d7) to $1.19209290 * 10^{-7}$ at $t = 4.22863988 * 10^{-4}$ (0x39ddb3d8).

The pattern repeats as t increases, leading to 7,712,437 segments of the form $H_j(t) = K_j/t^2$ where the $K_j > 0$ decrease as j increases. Each segment has the properties that $H_j'(t) < 0$ (the function is strictly decreasing) and $H_j''(t) > 0$ (the function is concave up). The theoretical function $\beta(t)$ has the properties that $\beta'(t) < 0$ (the function is strictly decreasing) and $\beta''(t) < 0$ (the function is concave down). The function collection of segments $H_j(t)$ is not an accurate approximation to $\beta(t)$ for small t, but the discontinuities for large t are small enough that for float calculations, the corresponding segments produce reasonable floating-point approximations.

Each of the $H_j(t)$ segments has domain consisting of 2 or more floating-point t-values. For t_k sufficiently large, the floating-point samples $\beta(t_k)$ are decreasing; that is, each segment has a single t-value for its domain. This behavior starts at $t = 0.999999344$ (0x3f7ffff5) with a β-value of 0.459697723.

The analysis indicates that depending on the values of t chosen in an application, the resulting floating-point rounding can lead to extremely inaccurate results. It is possible to use Taylor polynomials at $t = 0$ for an approximation, but it is necessary to select a switching point $\hat{t} > 0$ for which the Taylor polynomial should be used when $t \in [0, \hat{t}]$ and the equation $(1 - \cos(t))/t/t$ should be used when $t \in [\hat{t}, \pi]$.

It is not immediately clear what degree of the Taylor polynomial or what $\hat{t}$ should be for $\beta(t)$. An implication in the last section of [1] is to choose degree 6: $p(t) = 1/2! - t^2/4! + t^4/6! - t^4/8!$. If $\hat{t} = 1$ is chosen so that the Taylor polynomial skips all the $H_j(t)$ with domains containing at least 2 float numbers, the maximum of $|\beta(t) - p(t)|$ for $t \in [0, 1]$ occurs at $t = 1$ with value $2.73497 * 10^{-7}$. For smaller maximum error, you will need to choose $\hat{t}$ smaller than 1, keeping in mind how many $H_j(t)$ you want to skip by the approximation and how many you want to keep by evaluating the direct implementation of $\beta(t)$.

Smoothness at $\hat{t}$ might be important in your application. Let $q(t) = 1/2! - t^2/4! + q_4 t^4 + q_6 t^6$. This polynomial satisfies the conditions $q(0) = \beta(0)$, $q'(0) = \beta'(0)$ and $q''(0) = \beta''(0)$. If we also require C^1 continuity at $\hat{t} = 1$, where $q(1) = \beta(1)$ and $q'(1) = \beta'(1)$, then q_4 and q_6 are solutions to a linear system of equations. The solutions are $q_4 \doteq 1.38861746 * 10^{-3}$ and $q_6 \doteq 2.42566583 * 10^{-5}$. The maximum of $|\beta(t) - q(t)|$ for $t \in [0, 1]$ occurs at $t \doteq 0.706781$ with value $1.69002 * 10^{-8}$.

A similar analysis was performed for double to demonstrate that a larger precision does not eliminate the qualitative floating-point behavior. Table 2 lists transition t-values.

Table 2. Transition t-values for the double version of $\beta(t)$.

t-value	encoding
$t_0 = 2^{-1074}$	0x0000000000000001
$t_1 = 1.0536712127723507 * 10^{-8}$	0x3e46a09e667f3bcc
$t_2 = 1.0536712127723509 * 10^{-8}$	0x3e46a09e667f3bcd
$t_3 = 1.8250120749944284 * 10^{-8}$	0x3e53988e1409212e
$t_4 = 1.8250120749944287 * 10^{-8}$	0x3e53988e1409212f
$t_5 = 2.3560804576936208 * 10^{-8}$	0x3e594c583ada5b52
$t_6 = 2.3560804576936212 * 10^{-8}$	0x3e594c583ada5b53
$t_7 = 2.7877519926234643 * 10^{-8}$	0x3e5deeea11683f49
$t_8 = 2.7877519926234646 * 10^{-8}$	0x3e5deeea11683f4a
$t_9 = 3.1610136383170521 * 10^{-8}$	0x3e60f876ccdf6cd9

Table 3 shows evaluation results for the double version of $\beta(t)$.

Table 3. Evaluations of the double version of $\beta(t)$.

	t input	$\beta(t)$ output
#1	0	0.5
#2	$[t_0, t_1]$	0
#3	$[t_2, t_3]$	0.99999999999999989 ↘ 0.33333333333333343
#4	$[t_4, t_5]$	0.66666666666666652 ↘ 0.40000000000000002
#5	$[t_6, t_7]$	0.59999999999999998 ↘ 0.42857142857142860
#6	$[t_8, t_9]$	0.57142857142857129 ↘ 0.44444444444444453

The function $\beta(t)$ in floating-point terms is a sequence of segments $H_j(t) = K_j/t^2$. For j small, each domain includes 2 or more double numbers. For j sufficiently large, each domain is a single double number.

A Taylor polynomial can be used for small t, but as indicated previously, a numerical analysis must be performed to know what degree is required for accuracy and what choice of the switching point $\hat{t} > 0$ avoids the large discontinuity gaps. It is not sufficient to choose a small switching point arbitrarily; for example, you might say $\hat{t} = 10^{-6}$ is small enough. The large discontinuity gaps occur starting at $2.44140654 * 10^{-4}$ and occur regularly. How many gaps will you allow? For a specified degree, the larger you must choose $\hat{t}$ to avoid these, the larger the approximation error by the Taylor polynomial.

Based on the well-behaved floating-point nature of $\alpha(t)$, a robust implementation for $(1-\cos(t))/t^2$ uses the identity $(1 - \cos(t))/t^2 = (\sin(t/2)/(t/2))^2/2 = (\alpha(t/2))^2/2$ as shown in Listing 3.

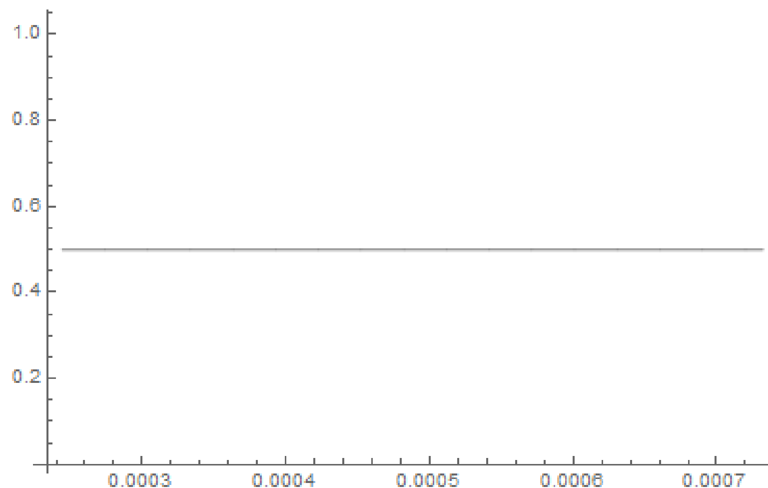
Listing 3. A robust implementation of $\beta(t) = (1 - \cos(t))/t^2$ for float using a trigonometric identity.

```
float Beta(float t)
{
    float tHalf = 0.5f * t;
    if (tHalf > 0.0f)
    {
        float alpha = Alpha(tHalf)
        return 0.5f * alpha * alpha;
    }
    else
    {
        return 0.5f;
    }
}
```

The computation of halfT is performed before the if-test to handle properly the case when $t = 2^{-129}$, the smallest subnormal t. The multiplication of this t by half causes rounding to zero when the default rounding mode is enabled; that is, tHalf is 0 for smallest subnormal t. For all other subnormals and normals in $[0, \pi]$, tHalf is positive.

Figure 3 shows the graph for the robust implementation of Beta(t).

Figure 3. The graph of the robust implementation of $\beta(t)$ for small t. The plot was drawn by choosing 8193 samples of $\beta(t)$ for $t \in [2.44141 * 10^{-4}, 7.32421875 * 10^{-4}]$ using the float implementation and then allowing Mathematica [8] to draw a high-resolution polyline connecting the samples.



At this resolution for t near 0, the robust floating-point behavior is evident. Compare this to the graph of the nonrobust implementation shown in Figure 2.

4.5 Evaluation of $\gamma(t)$

The direct implementation of $\gamma(t)$ is shown in Listing 4.

Listing 4. The direct implementation of $\gamma(t)$ for $t \in [0, \pi]$.

```
float Gamma(float t)
{
    if (t > 0.0f)
    {
        return (std::sin(t) - t * std::cos(t)) / t / t / t;
    }
    else
    {
        return 1.0f / 3.0f;
    }
}
```

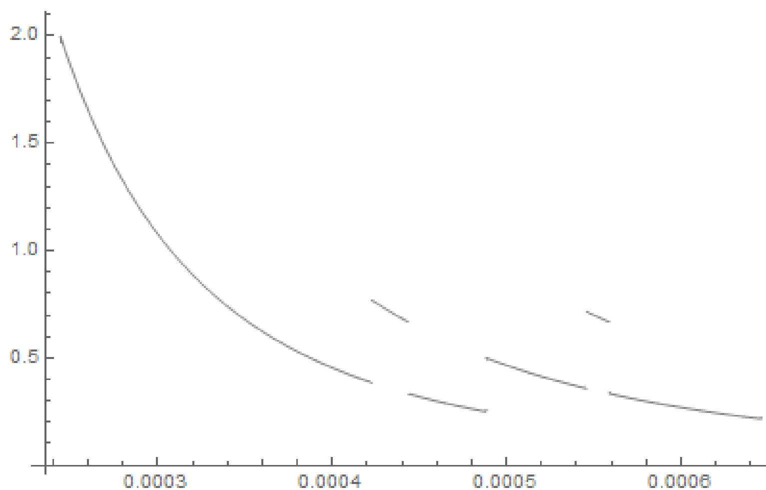
The same floating-point problems with the direct implementation of $\beta(t)$ occur with the direct implementation of $\gamma(t)$ for small t. Table 4 shows evaluation results for the float version of $\gamma(t)$.

Table 4. Evaluations of the float version of $\gamma(t)$ from Listing 4.

	t input	encoding	$\gamma(t)$ output
#1	0	0x00000000	0.333333343
#2	$[2^{-149}, 2.44140625 * 10^{-4}]$	[0x00000001, 0x39800000]	0
#3	$[2.44140654 * 10^{-4}, 4.22863959 * 10^{-4}]$	[0x39800001, 0x39ddb3d7]	1.99999928 ↘ 0.384900212
#4	$[4.22863988 * 10^{-4}, 4.43632947 * 10^{-4}]$	[0x39ddb3d8, 0x39e89768]	0.769800246 ↘ 0.666666687
#5	$[4.43632976 * 10^{-4}, 4.88281250 * 10^{-4}]$	[0x39e89769, 0x3a000000]	0.333333284 ↘ 0.250000000
#6	$[4.88281308 * 10^{-4}, 5.45914983 * 10^{-4}]$	[0x3a000001, 0x3a0f1bbc]	0.499999281 ↘ 0.357771009
#7	$[5.45915042 * 10^{-4}, 5.58942498 * 10^{-4}]$	[0x3a0f1bbd, 0x3a1285ff]	0.715541720 ↘ 0.666666687
#8	$[5.58942556 * 10^{-4}, 6.45935361 * 10^{-4}]$	[0x3a128600, 0x3a2953fd]	0.333333254 ↘ 0.215979710

Figure 4 shows the graph of the direct implementation of $\gamma(t)$ for small t.

Figure 4. The graph of $\gamma(t)$ for small t. The plot was drawn by choosing 8193 samples of $\gamma(t)$ for $t \in [2.44140625 * 10^{-4}, 6.45935361 * 10^{-4}]$ using the float implementation and then allowing Mathematica [8] to draw a high-resolution polyline connecting the samples.



At this resolution for t near 0, the floating-point behavior is evident. Using real-arithmetic, the maximum value of $\gamma(t)$ is 1/3. The floating-point values reach 2.0, and the inverse-square segments are not a good approximation to the actual function.

An attempt at robustness replaces $(\sin(t) - t\cos(t))/t^3$ by $(\alpha(t) - \cos(t))/t^2$ in the direct implementation, as shown in Listing 5.

Listing 5. An attempt at a robust implementation of $\gamma(t)$ for float.

```
float Gamma(float t)
{
    if (t > 0.0f)
    {
        float alpha = std::sin(t) / t;
        return (alpha - std::cos(t)) / t / t;
    }
    else
    {
        return 1.0f / 3.0f;
    }
}
```

This eliminates some of the large discontinuity gaps but not all. An analysis of the gaps shows that the maximum discontinuity is 0.381571442 at $t = 5.58942556{*}10^{-4}$. The gaps decrease in size to $1.57899857{*}10^{-2}$ at $t = 1.94289908{*}10^{-3}$. The next gap does not occur until much later, $8.34465027{*}10^{-7}$ at $t = 0.394411683$. The gaps remain on the order of 10^{-7} from this point on.

The gaps of size $1.57899857 * 10^{-2}$ and larger can be avoided by using a polynomial approximation. For example, choose a polynomial $P(t) = 1/3 - (1/30)t^2 + p^2 t^4 + p_3 t^6$ such that $P(0) = \gamma(0) = 1/3$, $P'(0) = \gamma'(0) = 0$, $P''(0) = \gamma''(0) = -1/15$, $P(\hat{t}) = \gamma(\hat{t})$ and $P'(\hat{t}) = \gamma'(\hat{t})$. The last two constraints are selected so that the real-valued piecewise function built from $P(t)$ and $\gamma(t)$ has C^1 continuity. The constraints have solution $p_2 = 1.19047014 * 10^{-3}$ and $p_3 = -2.19680528 * 10^{-5}$. Listing 6 shows the implementation.

Listing 6. A robust implementation of $\gamma(t)$ for float using a polynomial approximation for $t \in [0, \hat{t}]$, where $\hat{t} = 0.394411683$.

```
float GammaPolynomial(float t)
{
    std::array<float, 4> constexpr p =
    {
        1.0f / 3.0f,
        -1.0f / 30.0f,
        1.19047014e-3f,
        -2.19680528e-5f
    };

    float const tSqr = t * t;
    return p[0] + tSqr * (p[1] + tSqr * (p[2] + tSqr * p[3]));
}

float Gamma(float t)
{
    float constexpr tHat = 0.394411683f;

    if (t > tHat)
    {
        float alpha = std::sin(t) / t;
        return (alpha - std::cos(t)) / t / t;
    }
    else
    {
        return GammaPolynomial(t);
    }
}
```

4.6 Evaluation of $\delta(t)$

The direct implementation of $\delta(t)$ is shown in Listing 7.

Listing 7. The direct implementation of $\delta(t)$ for $t \in [0, \pi]$.

```
float Delta(float t)
{
    if (t > 0.0f)
    {
        return (2.0f * (1.0f - std::cos(t)) - t * std::sin(t)) / t / t / t / t;
    }
    else
    {
        return 1.0f / 12.0f;
    }
}
```

The floating-point problems are more severe, with Delta(t) returning -inf when $t = 2.64697828 * 10^{-23}$ (0x1a000001). The previous function values were 0 except at $t = 0$ where it is $1/12$. The computation of the numerator n has rounding errors that lead to n = -1.01e-45#DEN, a negative small subnormal. The number n/t = -5.29395529e-3, the number (n/t)/t = -1.99999952, the number ((n/t)/t)/t = -7.55578367e+22 and the number (((n/t)/t)/t)/t = -inf.

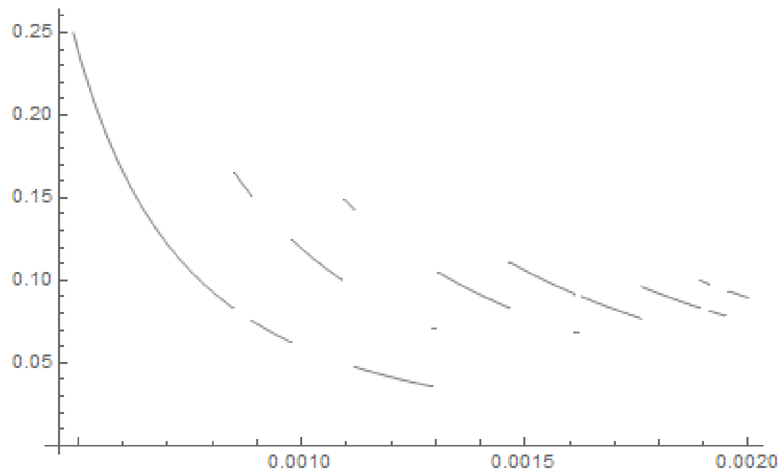
Differentiating the equation $\beta(t) = (\alpha(t/2))^2/2$ leads to $\beta'(t) = \alpha(t/2)\alpha'(t/2)/2$. A more robust implementation of $\delta(t)$ is shown in Listing 8.

Listing 8. An attempt at a robust implementation of $\delta(t)$ for type float.

```
float Delta(float t)
{
    float tHalf = 0.5f * t;
    if (tHalf > 0.0f)
    {
        float aHalf = std::sin(tHalf) / tHalf;
        float aderHalf = (aHalf - std::cos(tHalf)) / tHalf;
        return 0.5f * aHalf * aderHalf / t;
    }
    else
    {
        return 1.0f / 12.0f;
    }
}
```

The same floating-point problems with $\beta(t)$ occur with $\delta(t)$ for small t, as shown in Figure 5.

Figure 5. The graph of $\delta(t)$ for small t. The plot was drawn by choosing 8193 samples of $\delta(t)$ for $t \in [4.43632947 * 10^{-4}, 2.0 * 10^{-3}]$ using the float implementation and then allowing Mathematica [8] to draw a high-resolution polyline connecting the samples.



At this resolution for t near 0, the floating-point behavior is evident. Using real-arithmetic, the maximum value of $\delta(t)$ is $1/12$. The graph shows that for small t, nearly all the function values are larger than that.

Similar to the robust implementation of $\gamma(t)$, we can use a polynomial approximation to $\delta(t)$ for $t \in [0, \hat{t}]$ for a specified degree and for a switching point $\hat{t}$. Choosing the same gap error as that of $\gamma(t)$, $8.34465027 * 10^{-7}$, the smallest t-value at which this error is achieved is $t = 0.2616350353$. However, use the same $\hat{t}$ as that of $\gamma(t)$ so that the implementation for $\delta(t)$ can be simplified using a single switching point. The polynomial approximation is $Q(t) = 1/12 - t^2/180 + q_2 t^4 + q_3 q_6$ such that $Q(0) = \delta(0) = 1/12$, $Q'(0) = \delta'(0) = 0$, $Q''(0) = \delta''(0) = -1/90$, $Q(\hat{t}) = \delta(\hat{t})$ and $Q'(\hat{t}) = \delta'(\hat{t})$. The last two constraints are selected so that the real-valued piecewise function built from $Q(t)$ and $\delta(t)$ has C^1 continuity. The constraints have solution $q_2 = 1.48809020 * 10^{-4}$ and $q_3 = -2.19810423 * 10^{-6}$. Listing 9 shows the code.

Listing 9. A robust implementation of $\delta(t)$ for float using a polynomial approximation for $t \in [0, \hat{t}]$, where $\hat{t} = 0.394411683$.

```
float DeltaPolynomial(float t)
{
    std::array<float, 4> constexpr q =
    {
        1.0f / 12.0f,
        -1.0f / 180.0f,
        1.48809020e-4f,
        -2.19810423e-6f
    };

    float const tSqr = t * t;
    return q[0] + tSqr * (q[1] + tSqr * (q[2] + tSqr * q[3]));
}

float Delta(float t)
{
    float constexpr tHatHalf = 0.5f * 0.394411683f;

    float tHalf = 0.5f * t;
    if (tHalf > tHatHalf)
    {
        // delta(t) = -beta'(t)/t = -0.5*alpha(t/2)*alpha'(t/2)/t = -0.25*alpha(t/2)*gamma(t/2)
        float aHalf = std::sin(tHalf) / tHalf;
        float aderHalf = (aHalf - std::cos(tHalf)) / tHalf / tHalf;
        return 0.5f * aHalf * aderHalf / t;
    }
    else
    {
        return DeltaPolynomial(t);
    }
}
```

5 Minimax Approximations

An alternative approach to evaluating $\alpha(t)$, $\beta(t)$, $\gamma(t)$ and $\delta(t)$ is based on polynomial minimax approximations. This allows for fast computation with a reasonable accuracy and is easily implemented in hardware such as on an FPGA.

Each of our functions is defined on $[0, \pi]$ with the understanding that at $t = 0$, the removable-singularity values are used. Let $f(t)$ be one of our functions. The approximation is specified to have a degree bound n. The polynomial minimax approximation is the polynomial $p(t)$ defined on $[0, \pi]$ and of degree at most n for which $\max_{t \in [0,\pi]} |f(t) - p(t)|$ is minimized over all polynomials defined on $[0, \pi]$ and having degree at most n. Let the minimum error be denoted ε. The minimax polynomial has the property that there must be $n + 2$ points $0 \le t_0 < \cdots < t_{n+1} \le \pi$ such that $f(t_i) - p(t_i) = (-1)^i \varepsilon$.

5.1 Remez Algorithm

The *Remez algorithm* can be used to construct the coefficients of $p(t) = \sum_{i=0}^{n} p_i t^i$ numerically to approximate the function $f(t)$ for $t \in [a,b]$. A sketch of the algorithm is presented in [7]. In particular, the section entitled *Detailed discussion* lists the steps to follow. The description was too sparse in details for my liking, but I was able to use other sources of information to implement the basic algorithm. The details here are intended to help write an implementation. The theoretical basis for the algorithm is not discussed.

5.1.1 Chebyshev Nodes

The algorithm starts with initial guesses for the t_i for $0 \le i \le n+1$. For function domain $[-1,1]$, the standard choices are the Chebyshev nodes [4] in $(-1,1)$ and the endpoints,

$$\begin{aligned} t_0 &= -1 \\ t_k &= \cos\left(\tfrac{(2(n-k)+1)\pi}{2n}\right), \quad 1 \le k \le n \\ t_n &= +1 \end{aligned} \tag{13}$$

For function domain $[a,b]$, the Chebyshev nodes are transformed to

$$\begin{aligned} t_0 &= a \\ t_k &= \tfrac{a+b}{2} + \tfrac{b-a}{2}\cos\left(\tfrac{(2(n-k)+1)\pi}{2n}\right), \quad 1 \le k \le n \\ t_n &= b \end{aligned} \tag{14}$$

5.1.2 The Linear System to Determine Coefficients

The main idea is to update the t_k iteratively. A linear system of $n+2$ equations in the $n+2$ unknowns p_i for $0 \le i \le n$ and ε is formulated,

$$\sum_{i=0}^{n} p_i t_k^i + (-1)^k e = f(t_k), \quad 0 \le k \le n+1 \tag{15}$$

The solution to the linear system provides the coefficients for the approximating polynomial $p(t)$ and an estimate e of the minimax error. A general linear system solver, say, one that uses Gaussian elimination, is $O(n^3)$ in time. However, it is possible to solve the linear system of equation (15) in $O(n^2)$ time. In practice, the degree n is small, so the computation time using floating-point arithmetic is irrelevant on modern computers. If one were to use rational arithmetic instead, even for small degrees the time savings is significant; that said, the rational solution requires an enormous number of bits of precision. You would need a rational multiplication of two m-bit numbers that uses Fast Fourier transforms, $O(m \log m)$, rather than the standard $O(m^2)$ multiplication algorithm. Regardless, the linear system solving described next uses Newton polynomials to obtain the $O(n^2)$ asymptotic behavior.

5.1.3 Newton Polynomials

A detailed discussion of Newton polynomials is found at [6]. Only the information relative to this document is described here.

Given a set of $m+1$ data points $\{(t_k, g_k)\}_{k=0}^{m}$ where the t_k are strictly increasing, the Newton polynomial that interpolates the data points is of the form

$$N(t) = \sum_{i=0}^{m} a_i N_j(t) = \prod_{j=0}^{i-1}(t - t_j) \tag{16}$$

where $N_0(t) = 1$ and $N_j(t) = \prod_{j=0}^{i-1}(t - t_j)$ for $i \geq 1$. The coefficients a_i are forward divided differences [5],

$$\begin{array}{lcllcl} a_0 & = & [g_0] & = & g_0 \\ a_1 & = & [g_0, g_1] & = & (g_1 - g_0) \,/\, (t_1 - t_0) \\ & & & \vdots & \\ a_i & = & [g_0, \ldots, g_i] & = & ([g_1, \ldots, g_i] - [g_0, \ldots, g_{i-1}]) \,/\, (t_i - t_0) \end{array} \tag{17}$$

where the last equation defines the recursion for $2 \leq i \leq m$. These equations can be formulated as a linear system with lower-triangular coefficient matrix,

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 & (t_1 - t_0) & 0 & \cdots & 0 \\ 1 & (t_2 - t_0) & (t_2 - t_0)(t_2 - t_1) & \cdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 1 & (t_m - t_0) & (t_m - t_0)(t_m - t_1) & \cdots & \prod_{i=0}^{m-1}(t_m - t_i) \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} = \begin{bmatrix} g_0 \\ g_1 \\ g_2 \\ \vdots \\ g_m \end{bmatrix} \tag{18}$$

The solution is obtained by forward substitution, which is equivalent to $\sum_{i=0}^{j} a_i n_i(t_j) = g_j$ for $0 \leq j \leq m$. Listing 10 contains an implementation for solving the system.

Listing 10. Solving for a_i in the linear system of equation (18). The input and output arrays all have $m+1$ elements.

```
template <typename Real>
void SolveNewton(std::vector<Real> const& tNode, std::vector<Real> const& g, std::vector<Real>& a)
{
    size_t const mp1 = tNode.size();
    for (size_t i = 0; i < mp1; ++i)
    {
        a[i] = g[i];
        for (size_t j = 0; j < i; ++j)
        {
            a[i] -= a[j];
            a[i] /= tNode[i] - tNode[j];
        }
    }
}
```

Evaluation of a Newton polynomial is similar to the Horner's method, a nested evaluation. For example, consider $m = 3$. Horner's method for evaluation a standard polynomial is

$$P(t) = p_0 + p_1 t + p_2 t^2 + p_3 t^3 = p_0 + t(p_1 + t(p_2 + t p_3)) \tag{19}$$

The Newton polynomial is

$$\begin{aligned} N(t) &= a_0 + a_1(t - t_0) + a_2(t - t_0)(t - t_1) + a_3(t - t_0)(t - t_1)(t - t_2) \\ &= a_0 + (t - t_0)(a_1 + (t - t_1)(a_2 + (t - t_2)a_3)) \end{aligned} \tag{20}$$

The evaluation starts with a_3, multiplies by $(t-t_2)$, adds a_2, and so on. Listing 11 contains an implementation for evaluating a Newton polynomial.

Listing 11. Evaluating a Newton polynomial. The input arrays have $m + 1$ elements.

```
template <typename Real>
Real EvaluateNewton(Real t, std::vector<Real> const& tNode, std::vector<Real> const& a)
{
    size_t index = a.size() - 1;  // index is degree m
    Real result = a[index--];  // result = a[m], index is now m-1
    for (size_t i = 1; i < a.size(); ++i, --index)
    {
        result = a[index] + (t - tNode[index]) * result;
    }
    return result;
}
```

5.1.4 Solving the Linear System

The application to the Remez algorithm is as follows. Let $u(t)$ be the degree n Newton polynomial that interpolates $p(t)$ at the nodes t_0 through t_n; note that the node t_{n+1} is not included in the interpolation. Let $v(t)$ be the degree n Newton polynomial that interpolates $(-1)^k$ for $0 \le k \le n$. The linear combination is a Newton polynomial of degree n,

$$p(t) = u(t) - v(t)e \tag{21}$$

The e-term is obtained by solving the last linear equation at the node t_{n+1},

$$e = \frac{u(t_{n+1}) - f(t_{n+1})}{v(t_{n+1}) - (-1)^{n+1}} \tag{22}$$

5.1.5 Updating the t-Nodes

The current t_k values must be updated to provide better estimates of the nodes at which the minimax error is attained. Generally, if t_k is in an interval (z_0, z_1) where $f(t)-p(t) > 0$ and $f(z_0)-p(z_0) = f(z_1)-p(z_1) = 0$, it is replaced by $\bar{t}_k$ that is the location of the local maximum of $f(t) - p(t)$ on (z_0, z_1). Similarly, if t_k is in an interval (z_0, z_1) where $f(t) - p(t) < 0$ and $f(z_0) - p(z_0) = f(z_1) - p(z_1) = 0$, it is replaced by $\bar{t}_k$ that is the location of the local minimum of $f(t) - p(t)$ on (z_0, z_1).

To locate the critical points $\bar{t}_k$, the section *Detailed discussion* of [7] mentions: "No high precision is required here, the standard *line search* with a couple of *quadratic fits* should suffice." To support the claim, a reference to a book on linear and nonlinear programming is provided. This might be justified for small degrees, but it turns out that this is not the case for larger degrees. The maximum of $|f(t) - p(t)|$ becomes very small. Floating-point rounding errors in evaluating $f(t)$ and $p(t)$ become significant enough to cause inaccuracies in locating the extreme points using a quadratic-fit line search. For reference on quadratic-fit line searches,

see [2]. The approach I use involves bisection, not quadratic-fit line searches. This appears to be more robust, but precision problems can still occur when the signs of the function values at the root-bounding interval endpoints are misclassified because of rounding errors. I address this problem for a modified Remez algorithm that is used for the rotation-coefficient functions described in this document.

Given that the Chebyshev nodes provide good initial guesses for the t_k, the local extrema we seek, $\bar{t}_k$, are nearby the t_k. I use the t_k as endpoints to root-bounding intervals for $E(t) = f(t) - p(t)$. If $E(t_k)$ and $E(t_{k+1})$ have opposite signs, then bisection is used to find $z_k \in (t_k, t_{k+1})$ for which $E(z_k)$ is zero (within floating-point rounding error). I then find a root $\bar{t}_k$ to $E'(t) = 0$ on the interval (z_k, z_{k+1}), once again using bisection. The end intervals $[a, t_1]$ and $[t_n, b]$ might not have points where $E'(t)$ is zero, so instead a quadratic-fit line search is used for those intervals.

Given the new $\bar{t}_k$, the linear system is solved once again followed by the update of the nodes. The number of iterations is a user-defined parameter, but a test for sign oscillations of the errors at the nodes is applied for each iteration. If the signs fail to oscillate as $(-1)^k$, the algorithm is terminated and the current estimated coefficients are returned by the process.

5.1.6 Source Code

The source code for the Remez algorithm is found in the file RemezAlgorithm.h. An illustration of its use is shown in Listing 12.

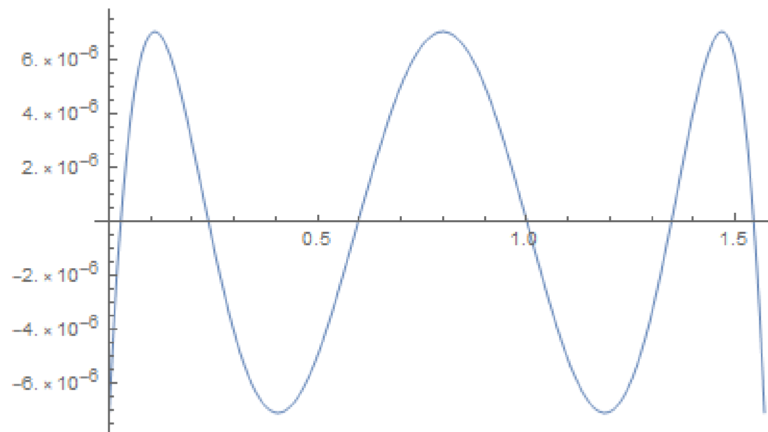
Listing 12. A sample application to illustrate computing the

```
#include <Mathematics/RemezAlgorithm.h>
using namespace gte;

void ApproximateSinDegree5()
{
    auto F = [](double const& x)
    {
        return std::sin(x);
    };

    auto FDer = [](double const& x)
    {
        return std::cos(x);
    };

    double const xMin = 0.0;
    double const xMax = GTE_C_HALF_PI;
    size_t const degree = 5;
    size_t const maxRemezIterations = 16;
    size_t const maxBisectionIterations = 1048;
    size_t const maxBracketIterations = 128;
    RemezAlgorithm<double> remez;

    auto iterations = remez.Execute(F, FDer, xMin, xMax, degree,
        maxRemezIterations, maxBisectionIterations, maxBracketIterations);
    auto const& p = remez.GetCoefficients();
    double estimatedMaximumError = remez.GetEstimatedMaxError();
    auto const& x = remez.GetXNodes();
    auto const& error = remez.GetErrors();

    // iterations = 16
    // p[0] = 7.0685186758729533e-06
    // p[1] = 0.99968986443393670
    // p[2] = 0.0021937161709613094
    // p[3] = -0.17223886508803649
```

```
    // p[4] = 0.0060973836732878166
    // p[5] = 0.0057217240548524534
    // estimatedMaximumError = -7.0685186758729533e-06
    // x[0] = 0.0000000000000000
    // x[1] = 0.10950063957513409
    // x[2] = 0.40467937702524381
    // x[3] = 0.79996961817349699
    // x[4] = 1.1880777522163142
    // x[5] = 1.4686862883722980
    // x[6] = 1.5707963267948966
    // error[0] = -7.0685186758729533e-06
    // error[1] = 7.0685186758651097e-06
    // error[2] = -7.0685186758789875e-06
    // error[3] = 7.0685186759344987e-06
    // error[4] = -7.0685186757124541e-06
    // error[5] = 7.0685186759344987e-06
    // error[6] = -7.0685186761565433e-06
}
```

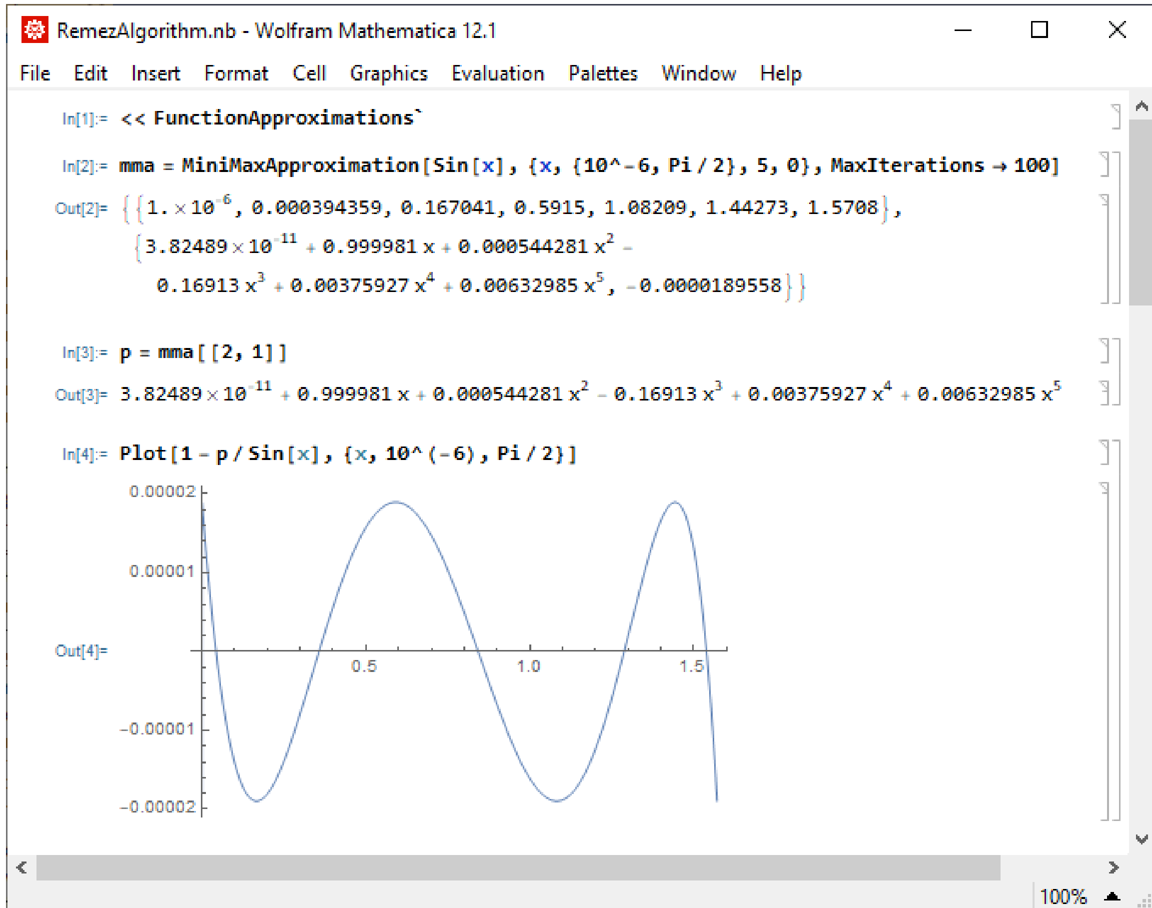
Figure 6 shows the graph of the difference $E(x) = \sin(x) - p(x)$. The local maxima are equioscillatory within floating-point rounding errors.

Figure 6. The graph of the difference between $\sin(x)$ and the minimax polynomial of degree 5 that approximates it. The plot was drawn with Mathematica [8].



As a side note, I used the Mathematica function MiniMaxApproximation uses relative error rather than absolute error; that is, instead of minimizing the maximum error for $|f(x) - p(x)|$ over all polynomials of a specified degree, Mathematica minimizes the maximum error for $|1 - p(x)/f(x)|$. This is problematic when $f(x)$ has roots on the interval of interest, which $\sin(x)$ does at $x = 0$. I chose the interval to be $[10^{-6}, \pi/2]$ to avoid the root. I also increased the maximum number of iterations to avoid a warning about incomplete convergence. The output is shown in Figure 7.

Figure 7. Mathematica [8] uses relative error for the minimax approximation rather than absolute error.



5.2 Modified Remez Algorithm

The Taylor series for the four rotation-coefficient functions all have even powers of t. The minimax polynomials $p(t)$ are constrained to have only even power terms. Moreover, if $f(t)$ is a rotation-coefficient function, it is required that $p(0) = f(0)$ and $p(\pi) = f(\pi)$.

5.2.1 Modifications that Work Generally

I modified the Remez algorithm as follows. The Chebyshev nodes are computed as the initial guesses for t_k, but then I overwrite the end values to obtain $t_0 = 0$ and $t_n = \pi$. The linear system of equations is modified

to

$$\begin{array}{lcl}\sum_{i=0}^{n} p_i t_0^{2i} & = & f(t_0) \\ \sum_{i=0}^{n} p_i t_k^{2i} + (-1)^{k+1} e & = & f(t_k), \quad 1 \leq k \leq n \\ \sum_{i=0}^{m} p_i t_n^{2i} & = & f(t_n)\end{array} \tag{23}$$

The first equation is equivalent to $p_0 = f(0)$ and the last equation is equivalent to $\sum_{i=0}^{n} p_i \pi^{2i} = f(\pi)$. Both equations do not have an error term on the left-hand side.

The computation of $u(t)$ to fit the $f(t_k)$ is effectively the same as for the standard Remez algorithm, except that n is half the degree of the polynomial and the variable is effectively $s = t^2$ with $s_k = t_k^2$.

The computation of $v(t)$ is slightly different. In the standard Remez algorithm, $v(t)$ fit the alternating signs $\{1, -1, \ldots, (-1)^{n+1}\}$. In the modified Remez algorithm, $v(t)$ must fit n of the $n+1$ error coefficients in the linear system. Choosing the first n error coefficients does not work because the last equation does not have an e-term to solve for, as was the case in generating equation (22). Instead, I choose the n error coefficients to come from all but the equation $k = \lfloor (n+2)/2 \rfloor$. This kth equation is then used to solve for e.

Updating the t_k values is the same as that for the standard Remez algorithm, but keeping in mind the polynomial terms are all even powers of t.

Bisection is used to locate the roots of $E(t) = f(t) - p(t)$ for the current polynomial approximation $p(t)$. These roots are used for bounding the roots of $E'(t) = f'(t) - p'(t)$. Bisection is also used to locate these roots which become the new t_k.

5.2.2 Modifications for the Rotation-Coefficient Functions

The general modifications discussed previously are theoretically based, treating $f(t)$ and $p(t)$ as differentiable functions of a real variable. In practice, though, floating-point computations of $f(t)$ and $p(t)$ have rounding errors that can cause the bisections to fail.

It is important to note that the bisection algorithm is formulated in terms of real-valued functions. If $g(t)$ is a continuous real-valued function on the domain $[t_0, t_1]$ with $g(t_0)g(t_1) < 0$, then $g(t)$ must have at least one root in (t_0, t_1). The bisection algorithm is summarized in the following steps:

1. Compute $\sigma_0 = \text{Sign}(g(t_0))$.
2. If $\sigma_0 = 0$, then t_0 is a root of $g(t)$ and the algorithm terminates.
3. Compute $\sigma_1 = \text{Sign}(g(t_1))$.
4. If $\sigma_1 = 0$, then t_1 is a root of $g(t)$ and the algorithm terminates.
5. It is known that $\sigma_0 \sigma_1 < 0$.
6. Compute $t_m = (t_0 + t_1)/2$ and $\sigma_m = \text{Sign}(g(t_m))$.
7. If $\sigma_m = 0$, then t_m is a root of $g(t)$ and the algorithm terminates.
8. If $\sigma_m = \sigma_0$, replace t_0 by t_m. Otherwise, $\sigma_m = \sigma_1$ so replace t_1 by t_m. Go to step 6 and repeat.

For real arithmetic, the loop formed by steps 6, 7 and 8 is infinite. In practice, the maximum number of iterations can be specified or a tolerance can be specified for how close $g(t)$ is to zero call t a root. When computing with floating-point arithmetic where the precision is finite, an alternative that avoids specifying either the maximum number of iterations or the tolerance. As the length of the subinterval containing a root decreases, eventually it will have no interior floating-point numbers due to the finite precision. When this happens, the bisection is terminated. Listing 13 shows code for this.

Listing 13. Bisection when computing with floating-point arithmetic. The assumption is that $g(t)$ is a finite floating-point number for all floating-point t in its domain. The type T is either float or double.

```
template <typename T>
void Bisection(std::function<T(T)> const& g, T t0, T t1, T& tRoot, T& gAtRoot)
{
    T g0 = g(t0);
    int sign0 = (g0 > 0 ? 1 : (g0 < 0 ? -1 : 0));
    if (sign0 == 0)
    {
        tRoot = t0;
        gAtRoot = 0;
        return;
    }

    T g1 = g(t1);
    int sign1 = (g1 > 0 ? 1 : (g1 < 0 ? -1 : 0));
    if (sign1 == 0)
    {
        tRoot = t1;
        gAtRoot = 0;
        return;
    }

    for (;;)
    {
        T tm = (t0 + t1) / 2;
        T gm = g(tm);
        if (gm == 0 || tm == t0 || tm == t1)
        {
            // This is the best we can do with fixed-precision floating-point arithmetic.
            tRoot = tm;
            gAtRoot = gm;
            return;
        }

        int signm = (gm > 0 ? 1 : (gm < 0 ? -1 : 0));
        if (signm == sign0)
        {
            t0 = tm;
        }
        else
        {
            t1 = tm;
        }
    }
}
```

The loop of Bisection is guaranteed to terminate when T is a floating-point type. One might be tempted to conclude that this is the most robust implementation possible for bisection. It is not, however. At a root r, $g(r) = 0$. For t near the root, $g(t)$ is near 0. Floating-point rounding errors in the evaluation of $g(t)$ can cause a misclassification of $\text{Sign}(g(t))$. This can lead to spurious roots—and many of them, even when you know theoretically $g(t)$ has a unique root on the interval.

The way to avoid this, if possible, is to implement a sign-testing function specifically for $g(t)$ that is accurate. For example, interval arithmetic can be used to determine the sign if it is 1 or -1. If the interval arithmetic has

no conclusive result about the sign (it might be 0), then additional logic can be used that takes advantage of knowledge of the structure of $g(t)$. A mixture of these two approaches is how I use bisection for the rotation-coefficient functions.

5.2.3 Computing the Exact Sign of a Power Series

The rotation-coefficient functions have power series representations provided by equation (11). Their derivatives have power series representations provided by equation (12). The idea for computing the exact sign of one of these power series is illustrated for $\alpha(t) = \sin(t)/t$. The same idea applies to the other functions. It is necessary to use rational arithmetic in the sign-test computations. I use the BSRational class in the Geometric Tools code for this purpose.

Bisection is used to compute the roots of $E(t) = \alpha(t) - p(t)$ and to compute the roots of $E'(t) = \alpha'(t) - p'(t)$ for the current polynomial $p(t) = \sum_{i=0}^{n} p_i t^{2i}$ and its derivative $p'(t) = 2t \sum_{i=0}^{n-1} (i+1) p_{i+1} t^{2i}$. The function $E'(t)$ has a removable singularity at $t = 0$ for which $E'(0) = 0$. We can instead compute the sign of $D(t) = E'(t)/(2t)$. Define $A(t)$ to be the Taylor polynomial of degree n of $E(t)$ and define $B(t)$ to be the Taylor polynomial of degree $n-1$ of $D(t)$,

$$A(t) = \sum_{i=0}^{n} \left(\frac{(-1)^i}{(2i+1)!} - p_i \right) t^{2i} = \sum_{i=0}^{n} a_i t^{2i} \tag{24}$$

where the last equality defines the coefficients a_i, and

$$B(t) = \sum_{i=0}^{n-1} (i+1) \left(\frac{(-1)^{i+1}}{(2i+3)!} - p_{i+1} \right) t^{2i} = \sum_{i=0}^{n-1} b_i t^{2i} \tag{25}$$

where the last equality defines the coefficients b_i. The $E(t)$ and $D(t)$ functions are the sum of these polynomials plus remainder series,

$$\begin{aligned} E(t) &= A(t) + \textstyle\sum_{i=n+1}^{\infty} \frac{(-1)^i}{(2i+1)!} t^{2i} \\ &= A(t) + (-1)^{n+1} \textstyle\sum_{j=0}^{\infty} \frac{(-1)^j}{(2(n+1+j)+1)!} t^{2(n+1+j)} \\ &= A(t) + (-1)^{n+1} \textstyle\sum_{j=0}^{\infty} (-1)^j r_j(t) \\ &= A(t) + (-1)^{n+1} R(t) \end{aligned} \tag{26}$$

where the last two equalities define $R(t)$ and $r_j(t)$, and

$$\begin{aligned} D(t) &= B(t) + \textstyle\sum_{i=n}^{\infty} (i+1) \frac{(-1)^{i+1}}{(2i+3)!} t^{2i} \\ &= B(t) + (-1)^{n+1} \textstyle\sum_{j=0}^{\infty} \frac{(-1)^j}{(2(n+j)+3)!} t^{2(n+j)} \\ &= B(t) + (-1)^{n+1} \textstyle\sum_{j=0}^{\infty} (-1)^j s_j(t) \\ &= B(t) + (-1)^{n+1} S(t) \end{aligned} \tag{27}$$

where the last two equalities define $S(t)$ and $s_j(t)$. The functions $R(t)$ and $S(t)$ are alternating series whose terms are decreasing in absolute value and have a limit of zero as j increases. At a specified t, as long as $E(t)$ is not zero, the alternating condition allows us to determine the exact sign of $E(t)$ using only a finite number

of terms of the remainder. The same is true for $D(t)$. Of course it is possible that the number of terms is significantly large that the computational time is too much for current computers to handle. However, a limit can be placed on the number of terms, after which a best guess is returned for the sign. For the tool I wrote to determine the minimax polynomials for the rotation-coefficient functions, that limit was small (32) and never exceeded.

Consider the expansion for $E(t)$ in equation (26). The function $R(t) = \sum_{j=0}^{\infty}(-1)^j r_j(t)$ for $t \in [0, \pi]$ is such that $r_j(t) > 0$, $r_{j+1}(t) < r_j(t)$ for all $j \geq 0$, and $\lim_{j\to\infty} r_j(t) = 0$. These conditions ensure $R(t) > 0$. Define $R_m(t) = \sum_{j=m}^{\infty}(-1)^{j-,} r_j(t)$ for $m \geq 0$; then $R_0(t) = R(t)$ and $R_m(t) > 0$ for all $m \geq 0$.

If $A(t) \geq 0$ and n is odd, then $E(t) = A(t) + R(t) > 0$, so we know the correct sign of $E(t)$. If $A(t) \leq 0$ and n is even, then $E(t) = A(t) - R(t) < 0$, so we also know the correct sign of $E(t)$.

If $A(t) \geq 0$ and n is even, the sign of $E(t) = A(t) - R(t)$ cannot be inferred immediately. Define $A_0(t) = A(t)$ and $A_{m+1}(t) = A_m(t) - (-1)^m r_m(t)$ for $m \geq 0$. Rewrite $E(t) = (A(t) - r_0(t)) + R_1(t) = A_1(t) + R_1(t)$. If $A_1(t) \geq 0$, then $E(t) > 0$. If $A_1(t) < 0$, then rewrite $E(t) = (A(t) - r_0(t) + r_1(t)) - R_2(t) = A_2(t) - R_2(t)$. If $A_2(t) \leq 0$, then $E(t) < 0$. If $A_2(t) > 0$, we can repeat the pattern of subtracting and adding remainder-series terms from $A(t)$ and testing the appropriate signs. This is repeated until the sign is determined.

If $A(t) \leq 0$ and n is odd, the sign of $E(t) = A(t) + R(t)$ cannot be inferred immediately. Now define $A_0(t) = A(t)$ and $A_{m+1}(t) = A_m(t) + (-1)^m r_m(t)$ for $m \geq 0$. Rewrite $E(t) = (A(t) + r_0(t)) - R_1(t) = A_1(t) - R_t(t)$. If $A_1(t) \leq 0$, then $E(t) > 0$. If $A_1(t) > 0$, then rewrite $E(t) = (A(t) + r_0(t) - r_1(t)) + R_2(t) = A_2(t) + R_2(t)$. If $A_2(t) \geq 0$, then $E(t) > 0$. If $A_2(t) < 0$, we can repeat the pattern of adding and subtracting remainder-series terms from $A(t)$ and testing the appropriate signs. This is repeated until the sign is determined.

As noted for practice, a maximum number of iterations should be specified so that when exceeded, the sign is reported as unknown. Rational arithmetic can be expensive. To reduce the cost of sign testing by amortization, floating-point interval arithmetic is used first in hopes of determining the sign without using rational arithmetic. Listing 14 contains pseudocode for the algorithm.

Listing 14. Pseudocode for computing the sign of $E(t)$. The coefficients of $A(t)$ must be computed once, but the sign testing can be performed for multiple t-values.

```
void CommputeACoefficients(size_t n, std::vector<double> const& p,
    std::vector<Rational>& a, Rational& twoNplus3Factorial)
{
    std::vector<Rational> factorial(n + 2);
    factorial[0] = 1;  // (2i+1)! at i = 0
    for (size_t i = 1, im1 = 0; i <= n + 1; ++i, ++im1)
    {
        factorial[i] = factorial[im1] * Rational((2 * i) * (2 * i + 1));
    }
    twoNplus3Factorial = factorial.back();

    a.resize(n + 1);
    Rational negOnePow = 1;
    for (size_t i = 0; i <= mDegree; ++i)
    {
        a[i] = negOnePow / factorial[i] - Rational(p[i]);
        negOnePow.SetSign(-negOnePow.GetSign());
    }
}
FPInterval<double> GetIntervalE(size_t n, double t, std::vector<double> const& p)
{
    if (t > 0.0)
    {
        // Compute an interval containing sin(t)/t.
        auto saveMode = std::fegetround();
```

```
        std::fesetround(FE_DOWNWARD);
        double sinTDown = std::sin(t);
        std::fesetround(FE_UPWARD);
        double sinTUp = std::sin(t);
        std::fesetround(saveMode);
        FPInterval<double> iSin(std::min(sinTDown, sinTUp), std::max(sinTDown, sinTUp));
        FPInterval<double> iT(t);
        FPInterval<double> iF = iSin / iT;

        // Copute an interval containing p(t).
        FPInterval<double> iTSqr = iT * iT;
        size_t index = n;
        FPInterval<double> iIndex(static_cast<double>(index));
        FPInterval<double> iP(p[index--]);
        for (size_t i = 1; i < p.size(); ++i, --index)
        {
            FPInterval<double> iPCoefficient(p[index]);
            iResult = iPCoefficient + iTSqr * iP;
        }

        // Compute an interval containing E(t).
        FPInterval<double> iE = iF - iP;
        return iE;
    }
    else
    {
        // sin(t)/t and p(t) are equal at t = 0, so E(t) = 0
        return FPInterval<double>(0.0);
    }
}

int GetSignE(size_t n, Rational t, std::vector<double> const& p,
    std::vector<Rational> const& a, Rational twoNplus3Factorial)
{
    // Attempt to classify the sign using floating-point arithmetic.
    FPInterval<double> iE = GetIntervalE(n, t, p);
    if (iE[0] > 0.0)
    {
        return +1;
    }

    if (iE[1] < 0.0)
    {
        return -1;
    }

    // The initial sign depends on whether n is odd or even.
    int sign = (n & 1 ? +1 : -1);

    // Compute A(t).
    Rational tSqr = t * t;
    Rational A = a[n];
    for (size_t i = 0, index = n - 1; i <= n - 1; ++i, --index)
    {
        A = a[index] + A * tSqr;
    }

    if (sign * A.GetSign() >= 0)
    {
        return sign;
    }

    // Compute tPow = t^{2n+2}.
    Rational tPow = tSqr;
    for (size_t i = 1; i <= n; ++i)
    {
        tPow *= tSqr;
    }

    // Apply the add/subtract pattern.
    size_t maxIterations = 32;
    Rational factor = tPow / twoNplus3Factorial;
```

```
    for (size_t j = 1; j <= maxIterations; j += 2)
    {
        Rational term = factor;
        term.SetSign(sign);
        A += term;
        if (sign * A.GetSign() <= 0)
        {
            return -sign;
        }

        term *= tSqr / Rational((2 * (n + j) + 2) * (2 * (n + j) + 3));
        term.SetSign(-sign);
        A += term;
        if (sign * A.GetSign() >= 0)
        {
            return sign;
        }
    }

    // Return an invalid integer to indicate the maximum number of iterations
    // has been exceeded. This code is not reached by the Geometric Tools
    // application that generates the polynomial approximations.
    return std::numeric_limits<int>::max();
}
```

A similar algorithm can be formulated for $D(t)$, $B(t)$ and $S(t)$. Listing 15 contains pseudocode for the algorithm.

Listing 15. Pseudocode for computing the sign of $D(t)$. The coefficients of $B(t)$ must be computed once, but the sign testing can be performed for multiple t-values.

```
void CommputeBCoefficients(size_t n, std::vector<double> const& p,
    std::vector<Rational>& b, Rational& twoNplus3Factorial)
{
    std::vector<Rational> factorial(n + 1);
    factorial[0] = 6;  // (2i+3)! at i = 0
    for (size_t i = 1, im1 = 0; i <= n; ++i, ++im1)
    {
        factorial[i] = factorial[im1] * Rational((2 * i + 2) * (2 * i + 3));
    }
    twoNplus3Factorial = factorial.back();

    b.resize(n);
    Rational negOnePow = -1;
    for (size_t i = 0; i < n; ++i)
    {
        b[i] = Rational(i + 1) * (negOnePow / factorial[i] - Rational(p[i]));
        negOnePow.SetSign(-negOnePow.GetSign());
    }
}

FPInterval<double> GetIntervalD(size_t n, double t, std::vector<double> const& p)
{
    if (t > 0.0)
    {
        // Compute an interval containing F'(t)/(2t).
        auto saveMode = std::fegetround();
        std::fesetround(FE_DOWNWARD);
        double cosTDown = std::cos(t);
        double sinTDown = std::sin(t);
        std::fesetround(FE_UPWARD);
        double cosTUp = std::cos(t);
        double sinTUp = std::sin(t);
        std::fesetround(saveMode);
        FPInterval<double> iCos(std::min(cosTDown, cosTUp), std::max(cosTDown, cosTUp));
        FPInterval<double> iSin(std::min(sinTDown, sinTUp), std::max(sinTDown, sinTUp));
```

```
        FPInterval<double> iT(t);
        FPInterval<double> iTSqr = iT * iT;
        FPInterval<double> iTCube = iT * iTSqr;
        FPInterval<double> iG = FPInterval<double>(0.5) * (iT * iCos - iSin) / iTCube;

        // Compute an interval containing p'(t)/(2t).
        FPInterval<double> iQ = GetIntervalQ(t);
        size_t index = n;
        FPInterval<double> iIndex(static_cast<double>(index));
        FPInterval<double> iQ = iIndex * FPInterval<double>(p[index--]);
        for (size_t i = 2; i < p.size(); ++i, --index)
        {
            iIndex = FPInterval<double>(static_cast<double>(index));
            FPInterval<double> iPCoefficient(p[index]);
            iQ = iIndex * iPCoefficient + iTSqr * iQ;
        }

        // Compute an interval containing D(t).
        FPInterval<double> iD = iG - iQ;
        return iD;
    }
    else
    {
        FPInterval<double> iG = FPInterval<double>(-1.0) / Interval(6.0);
        FPInterval<double> iQ = FPInterval<double>(p[1]);
        FPInterval<double> iD = iG - iQ;
        return iD;
    }
}

int GetSignD(size_t n, Rational t, std::vector<Rational> const& b,
    Rational twoNplus3Factorial)
{
    // The initial sign depends on whether n is odd or even.
    int sign = (n & 1 ? +1 : -1);

    // Compute A(t).
    Rational tSqr = t * t;
    Rational B = b[n - 1];
    for (size_t i = 0, index = n - 2; i <= n - 2; ++i, --index)
    {
        B = b[index] + B * tSqr;
    }

    if (sign * B.GetSign() >= 0)
    {
        return sign;
    }

    // Compute tPow = t^{2n}.
    Rational tPow = tSqr;
    for (size_t i = 1; i < n; ++i)
    {
        tPow *= tSqr;
    }

    // Apply the add/subtract pattern.
    size_t maxIterations = 32;
    Rational factor = tPow / twoNplus3Factorial;
    for (size_t j = 1; j <= maxIterations; j += 2)
    {
        Rational term = Rational(n + j) * factor;
        term.SetSign(sign);
        B += term;
        if (sign * B.GetSign() <= 0)
        {
            return -sign;
        }

        factor *= tSqr / Rational((2 * (n + j) + 2) * (2 * (n + j) + 3));
        term = Rational(n + j + 1) * factor;
        term.SetSign(-sign);
```

```
        B += term;
        if (sign * B.GetSign() >= 0)
        {
            return sign;
        }
    }

    // Return an invalid integer to indicate the maximum number of iterations
    // has been exceeded. This code is not reached by the Geometric Tools
    // application that generates the polynomial approximations.
    return std::numeric_limits<int>::max();
}
```

5.2.4 Source Code

The source code for the modified Remez algorithm applied to the rotation-coefficient functions is found in the tool GeometricTools/GTE/Tools/RotationApproximation. Links to the files are

RotationApproximationMain.cpp
RotationApproximationConsole.h
RotationApproximationConsole.cpp
RemezConstrained.h
RemezConstrained.cpp
RemezRotC0.h
RemezRotC0.cpp
RemezRotC1.h
RemezRotC1.cpp
RemezRotC2.h
RemezRotC2.cpp
RemezRotC3.h
RemezRotC3.cpp

The code was factored to use virtual functions that are specific to each rotation-coefficient function. The function $\alpha(t)$ is handled by RemezRotC0, the function $\beta(t)$ is handled by RemezRotC1, the function $\gamma(t)$ is handled by RemezRotC2 and the function $\delta(t)$ is handled by RemezRotC3. The goal was to share as much of the abstract details of the modified Remez algorithm into the base class RemezConstrained for code sharing.

The output of the tools consists of files name RotC?Info.txt that contains polynomial coefficients, the estimated maximum error, t-nodes and the corresponding $E(t)$ values. The degrees of fitting are $2 \leq n \leq 8$. The files RotC?GTL.txt contain Geometric Tools source code for the polynomial coefficients and maximum errors. The contents of these files were copied to RotationEstimate.h. The file also contains the functions for polynomial evaluations. And it contains source code for the rotation approximation of equation (3) and the rotation derivative approximations of equations (8), (9) and (10).

6 Polynomial Coefficients

The polynomial coeficients, t-nodes and associated errors $E(t)$ are displayed in tables in this section. Graphs of $E(t)$ were drawn using Mathematica [8].

Table 5. The polynomial coefficients, t-nodes and associated errors are shown here for $\alpha(t)$ for $t \in [0, \pi]$.

n	coefficients	t-nodes	errors
2	p[0] = +1.00000000000000000e+00		
	p[1] = -1.58971650732578684e-01	t[1] = +1.41696926221368713e+00	e[1] = -6.96563711867481672e-03
	p[2] = +5.84121356311684790e-03	t[2] = +2.77555038886482652e+00	e[2] = +6.96563711867501101e-03
3	p[0] = +1.00000000000000000e+00		
	p[1] = -1.66218398161274539e-01	t[1] = +1.06502458526975197e+00	e[1] = -2.23795060895759512e-04
	p[2] = +8.06129151017077016e-03	t[2] = +2.20850330064692724e+00	e[2] = +2.23795060895704001e-04
	p[3] = -1.50545944866583496e-04	t[3] = +2.94751550168847576e+00	e[3] = -2.23795060895801146e-04
4	p[0] = +1.00000000000000000e+00		
	p[1] = -1.66651290458553397e-01	t[1] = +8.53329526831219320e-01	e[1] = -4.86700964341668652e-06
	p[2] = +8.31836205080888937e-03	t[2] = +1.81734904413201592e+00	e[2] = +4.86700964330566421e-06
	p[3] = -1.93853969255209339e-04	t[3] = +2.55138186398592381e+00	e[3] = -4.86700964347219767e-06
	p[4] = +2.19921657358978346e-06	t[4] = +3.02055091107866058e+00	e[4] = +4.86700964307668071e-06
5	p[0] = +1.00000000000000000e+00		
	p[1] = -1.66666320608302304e-01	t[1] = +7.11696754226058381e-01	e[1] = -7.56547114955097300e-08
	p[2] = +8.33284074932796014e-03	t[2] = +1.53815790605168079e+00	e[2] = +7.56547114955097300e-08
	p[3] = -1.98184457544372085e-04	t[3] = +2.21700105040400874e+00	e[3] = -7.56547113289762763e-08
	p[4] = +2.70931602688878442e-06	t[4] = +2.73401164736507774e+00	e[4] = +7.56547116065320324e-08
	p[5] = -2.07033154672609224e-08	t[5] = +3.05865332776585674e+00	e[5] = -7.56547110861149896e-08
6	p[0] = +1.00000000000000000e+00		
	p[1] = -1.66666661172424985e-01	t[1] = +6.10295625428356914e-01	e[1] = -8.79391559571729431e-10
	p[2] = +8.33332258782319701e-03	t[2] = +1.33069662324980786e+00	e[2] = +8.79391448549426968e-10
	p[3] = -1.98405693280704135e-04	t[3] = +1.94839244797982092e+00	e[3] = -8.79391670594031893e-10
	p[4] = +2.75362742468406608e-06	t[4] = +2.45943689797937992e+00	e[4] = +8.79391670594031893e-10
	p[5] = -2.47308402190765123e-08	t[5] = +2.84310043249426592e+00	e[5] = -8.79391726105183125e-10
	p[6] = +1.36149932075244694e-10	t[6] = +3.08098593292690204e+00	e[6] = +8.79391684471819701e-10
7	p[0] = +1.00000000000000000e+00		
	p[1] = -1.66666666601880786e-01	t[1] = +5.34129297621524390e-01	e[1] = -7.91977594616355418e-12
	p[2] = +8.33333316679120591e-03	t[2] = +1.17158405134060084e+00	e[2] = +7.91966492386109167e-12
	p[3] = -1.98412553530683797e-04	t[3] = +1.72691263228428449e+00	e[3] = -7.91988696846601670e-12
	p[4] = +2.75567210003238900e-06	t[4] = +2.21943896819618747e+00	e[4] = +7.91988696846601670e-12
	p[5] = -2.50388692626200884e-08	t[5] = +2.61775904512782720e+00	e[5] = -7.91974819058793855e-12
	p[6] = +1.58972932135933544e-10	t[6] = +2.91360618222484735e+00	e[6] = +7.91972043501232292e-12
	p[7] = -6.61111627233688785e-13	t[7] = +3.09519274229929309e+00	e[7] = -7.91996156157548370e-12
8	p[0] = +1.00000000000000000e+00		
	p[1] = -1.66666666666648478e-01	t[1] = +1.71206551449520239e-01	e[1] = -4.44089209850062616e-16
	p[2] = +8.33333333318112164e-03	t[2] = +4.60075592255305033e-01	e[2] = +4.44089209850062616e-16
	p[3] = -1.98412698077537775e-04	t[3] = +8.57669717404237031e-01	e[3] = -4.44089209850062616e-16
	p[4] = +2.75573162083557394e-06	t[4] = +1.32506964372557778e+00	e[4] = +4.44089209850062616e-16
	p[5] = -2.50519743096581360e-08	t[5] = +1.81652300986421600e+00	e[5] = -3.33066907387546962e-16
	p[6] = +1.60558314470477309e-10	t[6] = +2.28392293618555620e+00	e[6] = +6.10622663543836097e-16
	p[7] = -7.60488921303402553e-13	t[7] = +2.68151706133448808e+00	e[7] = -2.49800180540660222e-16
	p[8] = +2.52255089807125025e-15	t[8] = +2.97038610214027310e+00	e[8] = +6.80011602582908381e-16

Figure 8. Graphs of $E(t)$ for $\alpha(t)$ and various degree minimax polynomials.

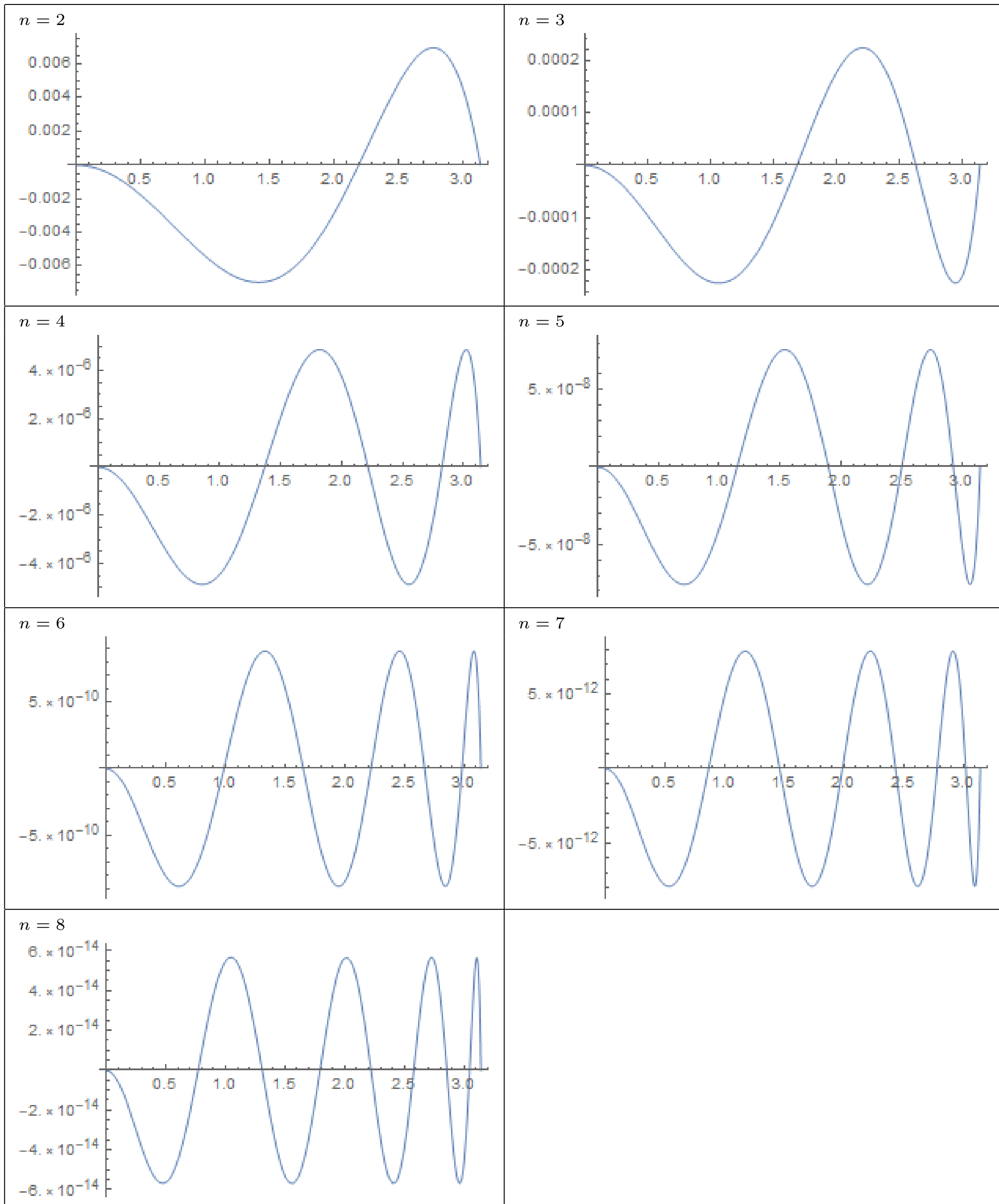


Table 6. The polynomial coefficients, t-nodes and associated errors are shown here for $\beta(t)$ for $t \in [0, \pi]$.

n	coefficients	t-nodes	errors
2	p[0] = +5.00000000000000000e-01		
	p[1] = -4.06593520914583922e-02	t[1] = +1.42276011523893997e+00	e[1] = -9.21190101505375836e-04
	p[2] = +1.06698549928666312e-03	t[2] = +2.77869585441082823e+00	e[2] = +9.21190101505375836e-04
3	p[0] = +5.00000000000000000e-01		
	p[1] = -4.16202835017619524e-02	t[1] = +1.06695024281444484e+00	e[1] = -2.32512618063007714e-05
	p[2] = +1.36087417563353699e-03	t[2] = +2.21077941596084315e+00	e[2] = +2.32512618062452603e-05
	p[3] = -1.99122437404000405e-05	t[3] = +2.94823245637554709e+00	e[3] = -2.32512618062730159e-05
4	p[0] = +5.00000000000000000e-01		
	p[1] = -4.16653520191245796e-02	t[1] = +8.54095869743815461e-01	e[1] = -4.16931608848702950e-07
	p[2] = +1.38761160375298095e-03	t[2] = +1.81851777200555631e+00	e[2] = +4.16931608848702950e-07
	p[3] = -2.44138380330618480e-05	t[3] = +2.55221790349626687e+00	e[3] = -4.16931608793191799e-07
	p[4] = +2.28499434819148172e-07	t[4] = +3.02076977142233449e+00	e[4] = +4.16931608820947375e-07
5	p[0] = +5.00000000000000000e-01		
	p[1] = -4.16666414534321572e-02	t[1] = +7.12046211488670533e-01	e[1] = -5.51778872592834091e-09
	p[2] = +1.38885303988537192e-03	t[2] = +1.53965237463759808e+00	e[2] = +5.51778867041718968e-09
	p[3] = -2.47850001122705350e-05	t[3] = +2.21757467564157373e+00	e[3] = -5.51778867041718968e-09
	p[4] = +2.72207208413898425e-07	t[4] = +2.73435222407546519e+00	e[4] = +5.51778867041718968e-09
	p[5] = -1.77358008600681907e-09	t[5] = +3.05873499545067862e+00	e[5] = -5.51778875368391653e-09
6	p[0] = +5.00000000000000000e-01		
	p[1] = -4.16666663178411334e-02	t[1] = +6.10472432234793638e-01	e[1] = -5.58657009541718708e-11
	p[2] = +1.38888820709641924e-03	t[2] = +1.33104460641283406e+00	e[2] = +5.58655344207181770e-11
	p[3] = -2.48011431705518285e-05	t[3] = +1.94872960419772490e+00	e[3] = -5.58655344207181770e-11
	p[4] = +2.75439902962340229e-07	t[4] = +2.45971981532091011e+00	e[4] = +5.58655899318694082e-11
	p[5] = -2.06736081122602257e-09	t[5] = +2.84334786702351305e+00	e[5] = -5.58657009541718708e-11
	p[6] = +9.93003618302030503e-12	t[6] = +3.08062222494501992e+00	e[6] = +5.58654511539913301e-11
7	p[0] = +5.00000000000000000e-01		
	p[1] = -4.16666666664263635e-02	t[1] = +2.10446803619233513e-01	e[1] = -7.16093850883225969e-15
	p[2] = +1.38888888750799658e-03	t[2] = +5.61107910590008752e-01	e[2] = +7.16093850883225969e-15
	p[3] = -2.48015851902670717e-05	t[3] = +1.03355234196907242e+00	e[3] = -7.16093850883225969e-15
	p[4] = +2.75571871163332658e-07	t[4] = +1.57079632679489656e+00	e[4] = +7.16093850883225969e-15
	p[5] = -2.08727380201649381e-09	t[5] = +2.10804031162072114e+00	e[5] = -7.16093850883225969e-15
	p[6] = +1.14076763269827225e-11	t[6] = +2.58048474299978459e+00	e[6] = +7.16093850883225969e-15
	p[7] = -4.28619236995285237e-14	t[7] = +2.93114584997056005e+00	e[7] = -7.16093850883225969e-15
8	p[0] = +5.00000000000000000e-01		
	p[1] = -4.16666666666571719e-02	t[1] = +1.71206551449520239e-01	e[1] = +7.21644966006351751e-16
	p[2] = +1.38888888885105744e-03	t[2] = +4.60075592255305033e-01	e[2] = -7.21644966006351751e-16
	p[3] = -2.48015872513761947e-05	t[3] = +8.57669717404237031e-01	e[3] = +7.21644966006351751e-16
	p[4] = +2.75573160474227648e-07	t[4] = +1.32506964372557778e+00	e[4] = -7.21644966006351751e-16
	p[5] = -2.08766469798137579e-09	t[5] = +1.81652300986421600e+00	e[5] = +7.21644966006351751e-16
	p[6] = +1.14685460418668139e-11	t[6] = +2.28392293618555620e+00	e[6] = -7.21644966006351751e-16
	p[7] = -4.75415775440997119e-14	t[7] = +2.68151706133448808e+00	e[7] = +6.66133814775093924e-16
	p[8] = +1.40555891469552795e-16	t[8] = +2.97038610214027310e+00	e[8] = -7.21644966006351751e-16

Figure 9. Graphs of $E(t)$ for $\beta(t)$ and various degree minimax polynomials.

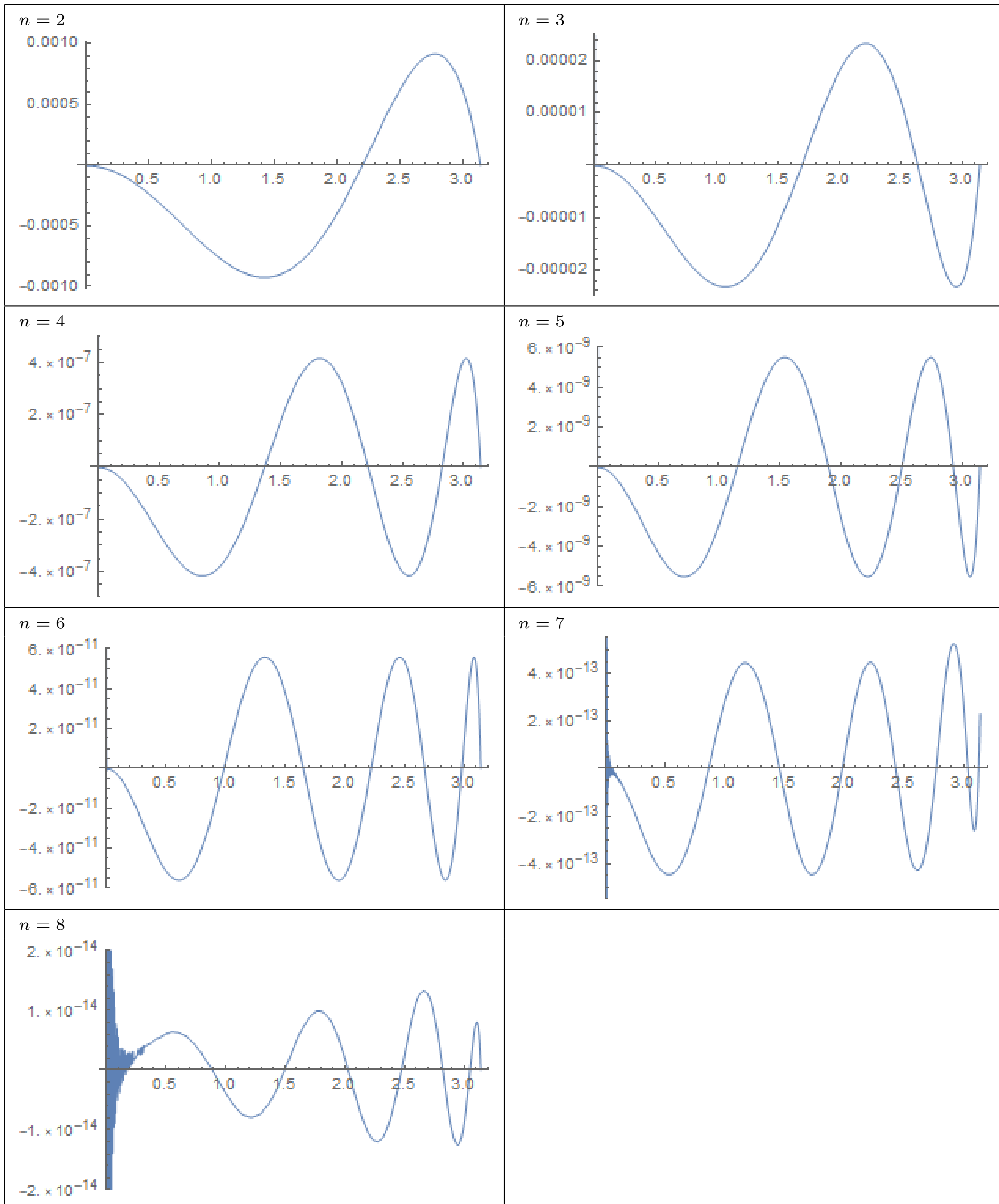


Table 7. The polynomial coefficients, t-nodes and associated errors are shown here for $\gamma(t)$ for $t \in [0, \pi]$.

n	coefficients	t-nodes	errors
2	p[0] = +3.33333333333333315e-01		
	p[1] = -3.24417271573718483e-02	t[1] = +1.42223031101916009e+00	e[1] = -8.14615084602177131e-04
	p[2] = +9.05201583387763454e-04	t[2] = +2.77840834490346023e+00	e[2] = +8.14615084602288153e-04
3	p[0] = +3.33333333333333315e-01		
	p[1] = -3.32912781805089902e-02	t[1] = +1.06680106617810866e+00	e[1] = -2.10750257848557609e-05
	p[2] = +1.16506615743456146e-03	t[2] = +2.21060290328081877e+00	e[2] = +2.10750257848002498e-05
	p[3] = -1.76083105011587047e-05	t[3] = +2.94817679453449877e+00	e[3] = -2.10750257847169831e-05
4	p[0] = +3.33333333333333315e-01		
	p[1] = -3.33321218985461534e-02	t[1] = +8.54044430959940870e-01	e[1] = -3.84148385823568361e-07
	p[2] = +1.18929901553194335e-03	t[2] = +1.81843926051814986e+00	e[2] = +3.84148386128879693e-07
	p[3] = -2.16884239911580259e-05	t[3] = +2.55216167857938370e+00	e[3] = -3.84148386128879693e-07
	p[4] = +2.07111898922214621e-07	t[4] = +3.02075495492954182e+00	e[4] = +3.84148386073368542e-07
5	p[0] = +3.33333333333333315e-01		
	p[1] = -3.33333098285273563e-02	t[1] = +7.12025844238255212e-01	e[1] = -5.14359671521802397e-09
	p[2] = +1.19044276839748377e-03	t[2] = +1.53872651514969183e+00	e[2] = +5.14359665970687274e-09
	p[3] = -2.20303898188601926e-05	t[3] = +2.21754025356027462e+00	e[3] = -5.14359657644014590e-09
	p[4] = +2.47382309397892291e-07	t[4] = +2.73433189396261778e+00	e[4] = +5.14359654868457028e-09
	p[5] = -1.63412179599052932e-09	t[5] = +3.05873013020570461e+00	e[5] = -5.14359664582908493e-09
6	p[0] = +3.33333333333333315e-01		
	p[1] = -3.33333330053029661e-02	t[1] = +6.10462474431522573e-01	e[1] = -5.25335885903643884e-11
	p[2] = +1.19047554930589209e-03	t[2] = +1.33102494758831069e+00	e[2] = +5.25333665457594634e-11
	p[3] = -2.20454376925152508e-05	t[3] = +1.94870868266697173e+00	e[3] = -5.25333110346082321e-11
	p[4] = +2.50395723787030737e-07	t[4] = +2.45970209711377397e+00	e[4] = +5.25333665457594634e-11
	p[5] = -1.90797721719554658e-09	t[5] = +2.84339912512876136e+00	e[5] = -5.25333110346082321e-11
	p[6] = +9.25661051509749896e-12	t[6] = +3.08035750612466508e+00	e[6] = +5.25333943013350790e-11
7	p[0] = +3.33333333333333315e-01		
	p[1] = -3.33333333331133561e-02	t[1] = +2.10446803619233513e-01	e[1] = -7.71605002114483796e-15
	p[2] = +1.19047618918715682e-03	t[2] = +5.61107910590008752e-01	e[2] = +7.71605002114483796e-15
	p[3] = -2.20458533943125258e-05	t[3] = +1.03355234196907242e+00	e[3] = -7.71605002114483796e-15
	p[4] = +2.50519837811549507e-07	t[4] = +1.57079632679489656e+00	e[4] = +7.77156117237609578e-15
	p[5] = -1.92670551155064303e-09	t[5] = +2.10804031162072114e+00	e[5] = -7.74380559676046687e-15
	p[6] = +1.06463697865186991e-11	t[6] = +2.58048474299978459e+00	e[6] = +7.71605002114483796e-15
	p[7] = -4.03135292145519115e-14	t[7] = +2.93114584997056005e+00	e[7] = -7.74380559676046687e-15
8	p[0] = +3.33333333333333315e-01		
	p[1] = -3.33333333333034956e-02	t[1] = +1.71206551449520239e-01	e[1] = +2.27595720048157091e-15
	p[2] = +1.19047619036920628e-03	t[2] = +4.60075592255305033e-01	e[2] = -2.27595720048157091e-15
	p[3] = -2.20458552540489507e-05	t[3] = +8.57669717404237031e-01	e[3] = +2.27595720048157091e-15
	p[4] = +2.50521015434838418e-07	t[4] = +1.32506964372557778e+00	e[4] = -2.27595720048157091e-15
	p[5] = -1.92706504721931338e-09	t[5] = +1.81652300986421600e+00	e[5] = +2.27595720048157091e-15
	p[6] = +1.07026043656398707e-11	t[6] = +2.28392293618555620e+00	e[6] = -2.27595720048157091e-15
	p[7] = -4.46498739610373537e-14	t[7] = +2.68151706133448808e+00	e[7] = +2.27595720048157091e-15
	p[8] = +1.30526089083317312e-16	t[8] = +2.97038610214027310e+00	e[8] = -2.23432383705812754e-15

Figure 10. Graphs of $E(t)$ for $\gamma(t)$ and various degree minimax polynomials.

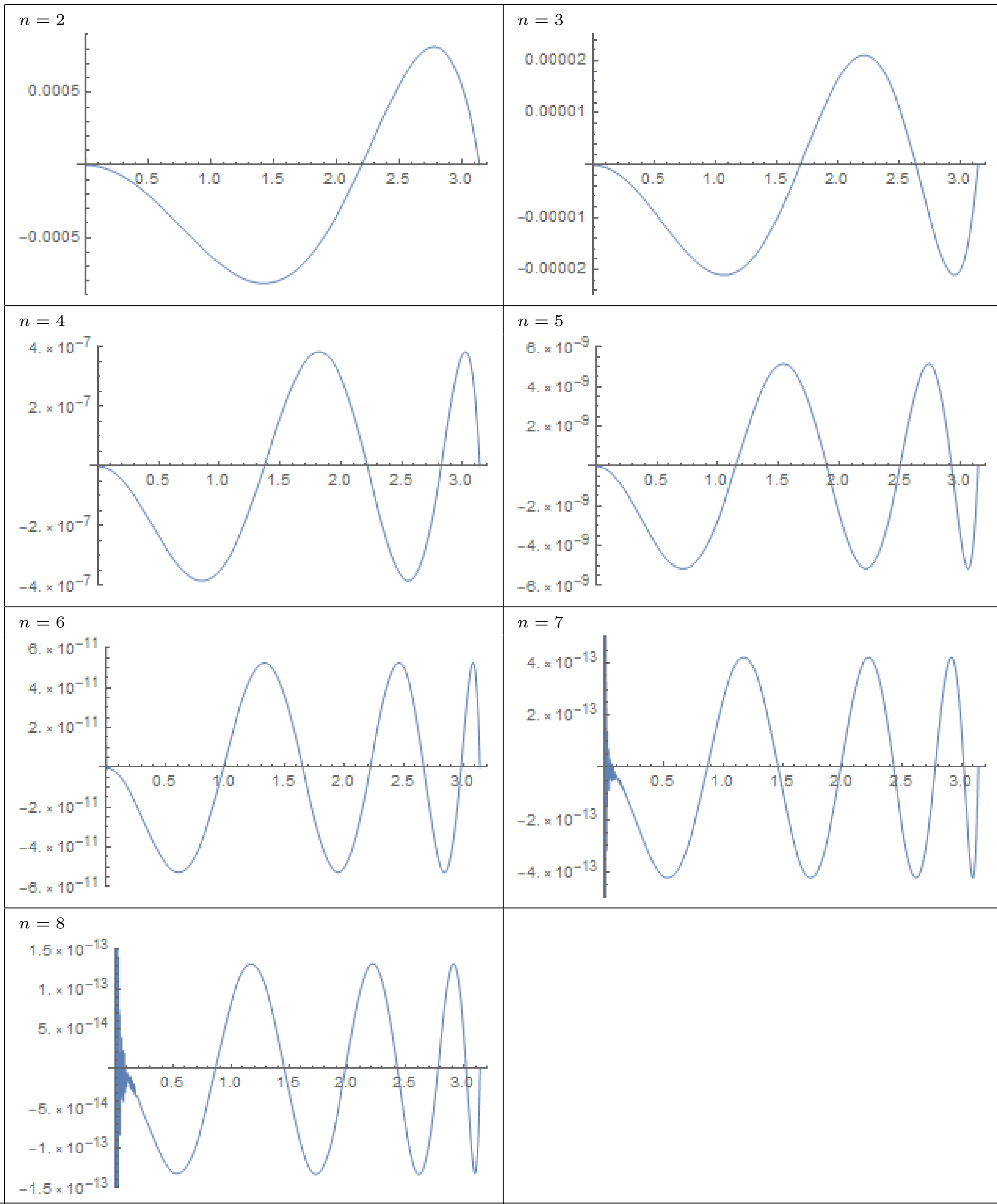
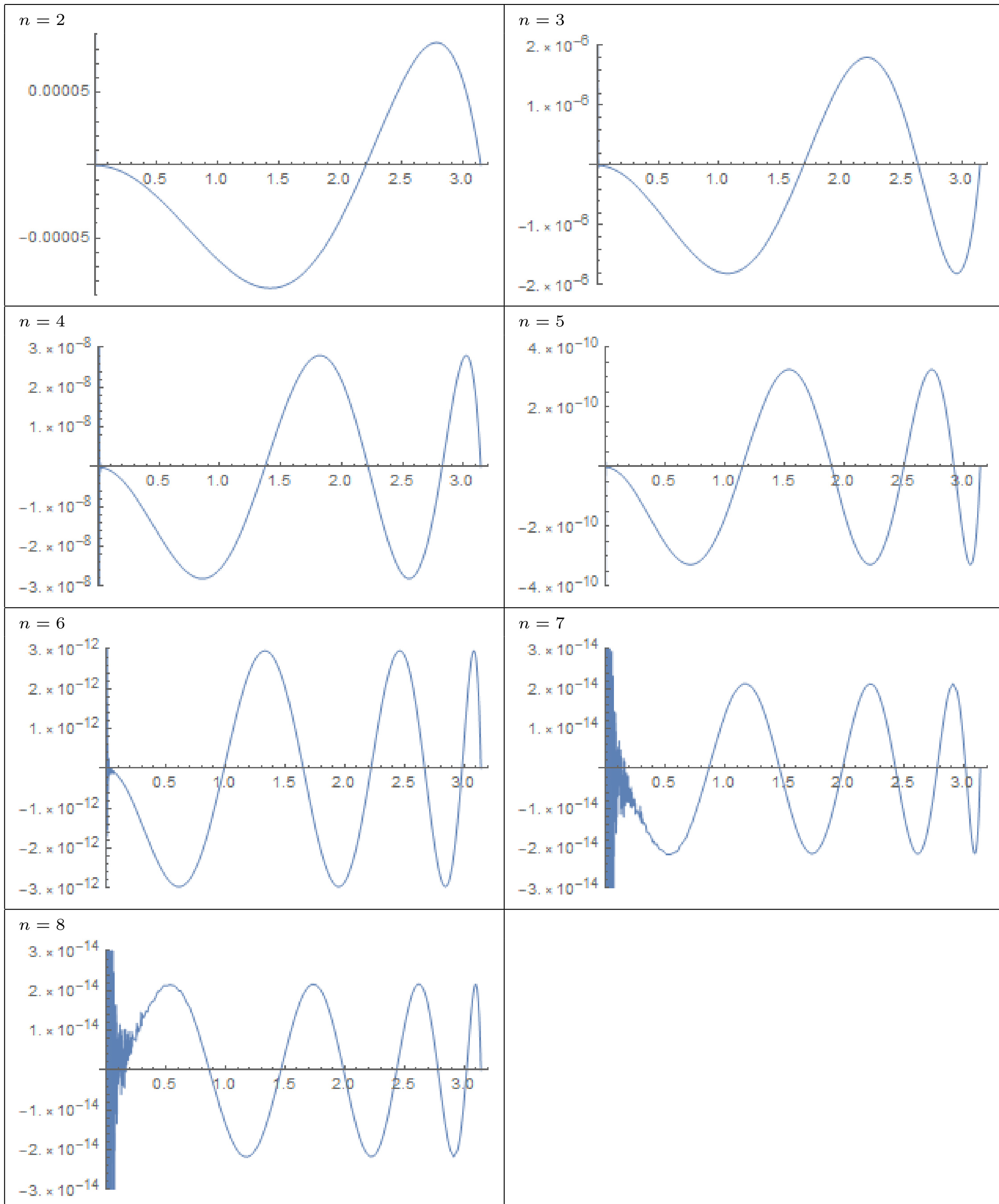


Table 8. The polynomial coefficients, t-nodes and associated errors are shown here for $\delta(t)$ for $t \in [0, \pi]$.

n	coefficients	t-nodes	errors
2	p[0] = +8.33333333333333287e-02		
	p[1] = -5.46357009138465424e-03	t[1] = +1.42605999087888513e+00	e[1] = -8.46120368888786389e-05
	p[2] = +1.19638433962248889e-04	t[2] = +2.78046287136283787e+00	e[2] = +8.46120368888855778e-05
3	p[0] = +8.33333333333333287e-02		
	p[1] = -5.55196372993948303e-03	t[1] = +1.06817028719641804e+00	e[1] = -1.80519731848849396e-06
	p[2] = +1.46646667516630680e-04	t[2] = +2.21221668479315792e+00	e[2] = +1.80519731859257737e-06
	p[3] = -1.82905866698780768e-06	t[3] = +2.94868366244126356e+00	e[3] = -1.80519731859951627e-06
4	p[0] = +8.33333333333333287e-02		
	p[1] = -5.55546733314307706e-03	t[1] = +8.54617646375245066e-01	e[1] = -2.80161038951343144e-08
	p[2] = +1.48723933698110248e-04	t[2] = +1.81931270563229708e+00	e[2] = +2.80161039506454657e-08
	p[3] = -2.17865651989456709e-06	t[3] = +2.55278594674443404e+00	e[3] = -2.80161039367676779e-08
	p[4] = +1.77408035681006169e-08	t[4] = +3.02091841327438537e+00	e[4] = +2.80161039367676779e-08
5	p[0] = +8.33333333333333287e-02		
	p[1] = -5.55555406357728914e-03	t[1] = +7.12296212146430685e-01	e[1] = -3.26754151513952706e-10
	p[2] = +1.48807404153008735e-04	t[2] = +1.54086677889246326e+00	e[2] = +3.26753749058106280e-10
	p[3] = -2.20360578108261882e-06	t[3] = +2.21798657499795304e+00	e[3] = -3.26753742119212376e-10
	p[4] = +2.06782449582308932e-08	t[4] = +2.73459656227776815e+00	e[4] = +3.26753755997000184e-10
	p[5] = -1.19178562817913197e-10	t[5] = +3.05878482724521739e+00	e[5] = -3.26753749058106280e-10
6	p[0] = +8.33333333333333287e-02		
	p[1] = -5.55555555324832757e-03	t[1] = +2.64726913948481579e-01	e[1] = -1.37140299116822462e-13
	p[2] = +1.48809514798423797e-04	t[2] = +6.98108645491121060e-01	e[2] = +1.37140299116822462e-13
	p[3] = -2.20457622072950518e-06	t[3] = +1.26434916557872890e+00	e[3] = -1.37140299116822462e-13
	p[4] = +2.08728631685852690e-08	t[4] = +1.87724348801106444e+00	e[4] = +1.37126421329014647e-13
	p[5] = -1.36888190776165574e-10	t[5] = +2.44348400809867261e+00	e[5] = -1.37133360222918554e-13
	p[6] = +5.99292681875750821e-13	t[6] = +2.87686573964131131e+00	e[6] = +1.37133360222918554e-13
7	p[0] = +8.33333333333333287e-02		
	p[1] = -5.55555555528319030e-03	t[1] = +2.10446803619233513e-01	e[1] = +3.20715676238592096e-14
	p[2] = +1.48809523101214977e-04	t[2] = +5.61107910590008752e-01	e[2] = -3.20715676238592096e-14
	p[3] = -2.20458493798151629e-06	t[3] = +1.03355234196907242e+00	e[3] = +3.20715676238592096e-14
	p[4] = +2.08765224186559757e-08	t[4] = +1.57079632679489656e+00	e[4] = -3.20715676238592096e-14
	p[5] = -1.37600800115177215e-10	t[5] = +2.10804031162072114e+00	e[5] = +3.20715676238592096e-14
	p[6] = +6.63762129016229865e-13	t[6] = +2.58048474299978459e+00	e[6] = -3.20785065177631168e-14
	p[7] = -2.19044013684859942e-15	t[7] = +2.93114584997056005e+00	e[7] = +3.20715676238592096e-14
8	p[0] = +8.33333333333333287e-02		
	p[1] = -5.55555555501025672e-03	t[1] = +1.71206551449520239e-01	e[1] = +4.77673456344973602e-14
	p[2] = +1.48809521898935978e-04	t[2] = +4.60075592255305033e-01	e[2] = -4.77673456344973602e-14
	p[3] = -2.20458342827337994e-06	t[3] = +8.57669717404237031e-01	e[3] = +4.77673456344973602e-14
	p[4] = +2.08757075326674457e-08	t[4] = +1.32506964372557778e+00	e[4] = -4.77673456344973602e-14
	p[5] = -1.37379825035843510e-10	t[5] = +1.81652300986421600e+00	e[5] = +4.77673456344973602e-14
	p[6] = +6.32209097599974706e-13	t[6] = +2.28392293618555620e+00	e[6] = -4.77742845284012674e-14
	p[7] = +7.39204014316007136e-17	t[7] = +2.68151706133448808e+00	e[7] = +4.77673456344973602e-14
	p[8] = -6.43236558920699052e-17	t[8] = +2.97038610214027310e+00	e[8] = -4.77673456344973602e-14

Figure 11. Graphs of $E(t)$ for $\delta(t)$ and various degree minimax polynomials.



References

[1] Ethan Eade. Lie Groups for Computer Vision.
http://ethaneade.com/lie_groups.pdf, 2014.

[2] Katta G. Murty. Line search algorithms.
http://www-personal.umich.edu/~murty/611/611slides9.pdf.
IOE 611 Lecture slides, unknown publication date.

[3] IEEE Computer Society and Microprocessor Standards Association.
IEEE 754™-2008 - IEEE Standard for Floating-Point Arithmetic.
https://standards.ieee.org/standard/754-2008.html.

[4] Wikipedia. Chebyshev nodes.
https://en.wikipedia.org/wiki/Chebyshev_nodes.
accessed August 10, 2020.

[5] Wikipedia. Divided differences.
https://en.wikipedia.org/wiki/Divided_differences.
accessed August 10, 2020.

[6] Wikipedia. Newton polynomial.
https://en.wikipedia.org/wiki/Newton_polynomial.
accessed August 10, 2020.

[7] Wikipedia. Remez algorithm.
https://en.wikipedia.org/wiki/Remez_algorithm.
accessed August 10, 2020.

[8] Wolfram Research, Inc. *Mathematica 12.1.1.0*. Wolfram Research, Inc., Champaign, Illinois, 2020.