Developing Mathematical Software for Efficient Representation of Functions and Numerical Integration: julia and ApproxFun

Dion Ho

February 11, 2019

Numerical integration, otherwise known as quadrature, denotes a set of algorithms in which a definite integral is approximated, rather than determined exactly. These algorithms are used when direct (exact) integration of the function is difficult or impossible.

We experimented using the Newton-Cotes rules, Lagrange polynomial interpolation, Chebyshev polynomial interpolation, and Taylor polynomial approximation, to perform numerical integration. These algorithms approximate the integrand with an *approximation polynomial*. The approximation polynomial is integrated (which is trivially easy) to attain an approximation of the actual definite integral. While the Newton-Cotes rules appear to bypass the formation of an approximation polynomial, they are in fact equivalent to Lagrange polynomial interpolation (due to theorem 1 which is elaborated below).

1 Polynomial Interpolation

With the exception of Taylor polynomial approximation, all the algorithms experimented with use polynomial interpolation to create an *interpolating polynomial* which is used as the approximation polynomial. In contrast, the Taylor polynomial based algorithm performs a Taylor expansion at the midpoint of the integration interval or spline (see §4 for an elaboration on splines). By definition, polynomial interpolation creates an interpolating polynomial which intersects the actual function at specific points, called *interpolation nodes*.

Lagrange polynomial interpolation performs the interpolation by solving a system of linear equations. For example, given 3 interpolation nodes: (1,3), (5,4), (2,0), we can construct a second-order interpolating polynomial.

Let the interpolating polynomial be ax^2+bx+c. Therefore, we have the following system of linear equations:

$$a(1)^2 + b(1) + c = 3,$$
$$a(5)^2 + b(5) + c = 4,$$
$$a(2)^2 + b(2) + c = 0.$$

Solving the system of linear equations, we find $a = \frac{13}{12}, b = \frac{-25}{4}, c = \frac{49}{6}$. Therefore, the interpolating polynomial is $\frac{13}{12}x^2 + \frac{-25}{4}x + \frac{49}{6}$. Note that while there may exist a third-order interpolating polynomial which intersects the integrand at the 3 nodes, interpolating polynomial generally refers to the *interpolating polynomial of the lowest possible order*; i.e. the interpolating polynomial which order is one less than the number of nodes.

A method of solving the system of linear equations is to use matrices. With reference to the system of linear equations above, let

$$V = \begin{pmatrix} 1 & 1 & 1 \\ 25 & 5 & 1 \\ 4 & 2 & 1 \end{pmatrix} \text{ and } Y = \begin{pmatrix} 3 \\ 4 \\ 0 \end{pmatrix}.$$

V is the Vandermonde matrix containing the coefficients of unknowns a, b, c, and Y is the vector of y-values.

$$V^{-1}Y = \begin{pmatrix} \frac{13}{12} \\ \frac{-25}{4} \\ \frac{49}{6} \end{pmatrix}$$

provides the values of the coefficients of the interpolating polynomial.

Theorem 1. *For all natural numbers $n \geq 2$, for all choices of n unique nodes, the lowest order interpolating polynomial is unique.*

Proof. Given any $n \geq 2$ unique nodes: $(x_1, y_1), (x_2, y_2) \ldots, (x_n, y_n)$, we can form the following system of linear equations:

$$a_n(x_1)^{n-1} + a_{n-1}(x_1)^{n-2} + \ldots + a_1 = y_1,$$
$$a_n(x_2)^{n-1} + a_{n-1}(x_2)^{n-2} + \ldots + a_1 = y_2,$$
$$\vdots \tag{1}$$
$$a_n(x_n)^{n-1} + a_{n-1}(x_n)^{n-2} + \ldots + a_1 = y_n. \tag{2}$$

Therefore, we can form matrix V_1 and vector Y_1, each of which may be real or complex valued,

$$V_1 = \begin{pmatrix} {x_1}^{n-1} & {x_1}^{n-2} & \ldots & 1 \\ {x_2}^{n-1} & {x_2}^{n-2} & \ldots & 1 \\ \vdots & \vdots & \vdots & \vdots \\ {x_n}^{n-1} & {x_n}^{n-2} & \ldots & 1 \end{pmatrix} \text{ and } Y_1 = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}.$$

${V_1}^{-1}Y_1 = (a_n \quad a_{n-1} \quad \dots \quad a_1)^T$ where $a_n, a_{n-1}, \dots, a_1$ are coefficients of an interpolating polynomial, P_1.

Consider a second polynomial, P_2, with coefficients $b_n, b_{n-1}, \dots, b_1$, which intersects the function at the same n nodes. The coefficients $b_n, b_{n-1}, \dots, b_1$ must fulfil the system of linear equations (2). Therefore, we can form matrix V_2 and vector Y_2 where $V_2 = V_1$ and $Y_2 = Y_1$ and ${V_2}^{-1}Y_2 = (b_n \quad b_{n-1} \quad \dots \quad b_1)^T$. That P_1 and P_2 are interpolating polynomials of the lowest order implies that V_1 and V_2 are square matrices. The inverse of a square matrix, if it exists, is unique. Therefore, $V_1^{-1} = V_2^{-1}$, which implies that $V_1^{-1}Y_1 = V_2^{-1}Y_2$ which in turn implies that $P_1 = P_2$. □

The implication of theorem 1 is regardless of how the coefficients of the approximation polynomial are calculated (see §2) the resultant coefficients will be identical if and only if the interpolation nodes are identical.

Therefore, all that differs between different polynomial interpolation methods is the choice of interpolation nodes.

2 Choice of Interpolation Nodes

The intuitive choice of interpolation nodes is equispaced nodes. Standard Lagrange polynomial interpolation uses equispaced nodes. Moreover, the Newton-Cotes rules use equispace nodes (which is why they are equivalent to Lagrange polynomial interpolation by Theorem 1).

Equispaced nodes however face the problem of the Runge phenomenon. The Runge phenomenon is the phenomenon of an increase in the order of the interpolating polynomial resulting in a *decrease* in the accuracy of the approximation. The Runge phenomenon manifests as oscillations at the extreme ends of the interpolating polynomial which deviate significantly from the actual function (see figure 1).

Trefethen [6] provides a solution to the Runge phenomenon: use the roots of Jacobi polynomials as interpolation nodes. These nodes are clustered about the extreme ends of the actual function (see figure 2). In particular, Trefethen focuses on the Legendre and Chebyshev polynomials (special Jacobi polynomials), he writes "[For] polynomial interpolation in Legendre or Chebyshev points, $\|f - p_N\| = \mathcal{O}(\text{constant}^{-N})$ if f is analytic (for some constant greater than 1)" (p. 264). This implies that the Runge phenomenon will not occur. Trefethen [6] also provides an asymptotic description of the Lebesgue constant, Λ_N, for equispaced, Legendre and Chebyshev nodes (a smaller Lebesgue constant indicates that the nodes result in a better polynomial approximation).

$$\text{Equispaced points: } \Lambda_N \sim 2^N/eN\log N$$
$$\text{Legendre points: } \Lambda_N \sim \text{const}\sqrt{N}$$
$$\text{Chebyshev points: } \Lambda_N \sim \text{const}\log N$$

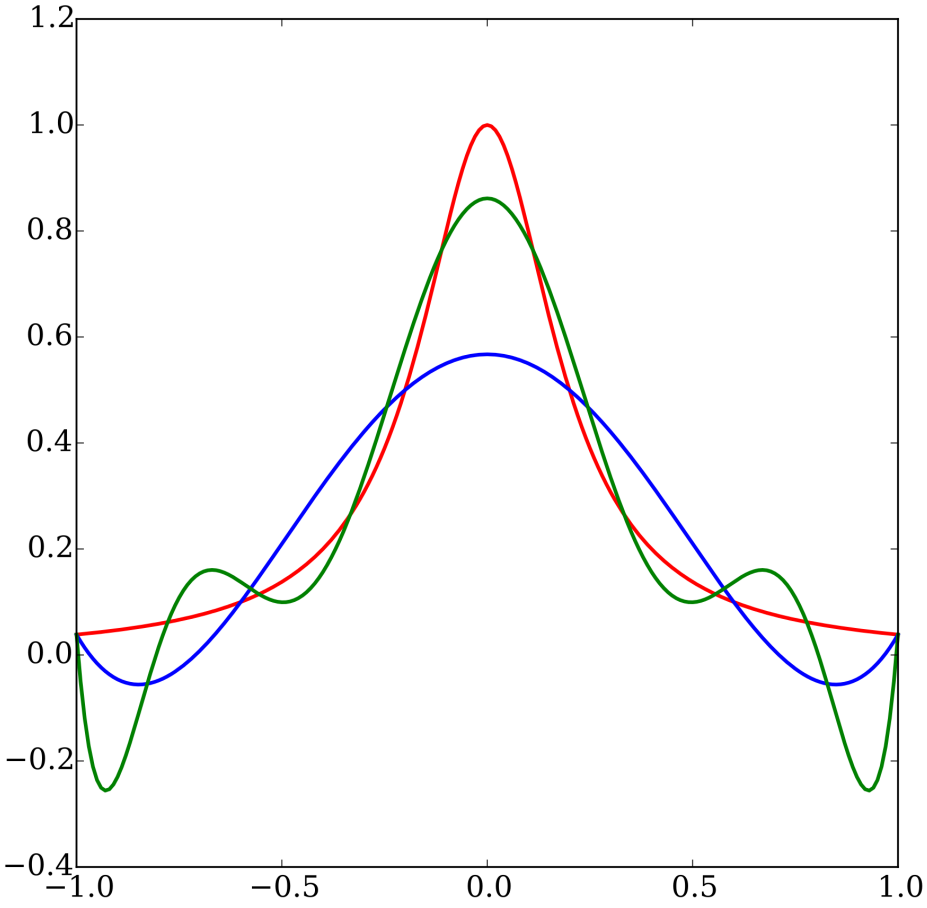


Figure 1: The red curve is the function to be approximated (Runge's function). The green curve is of higher order than the blue curve, yet it is a worse approximation due to the oscillations at the extreme ends. Taken from `https://commons.wikimedia.org/wiki/File:Runge_phenomenon.svg`.

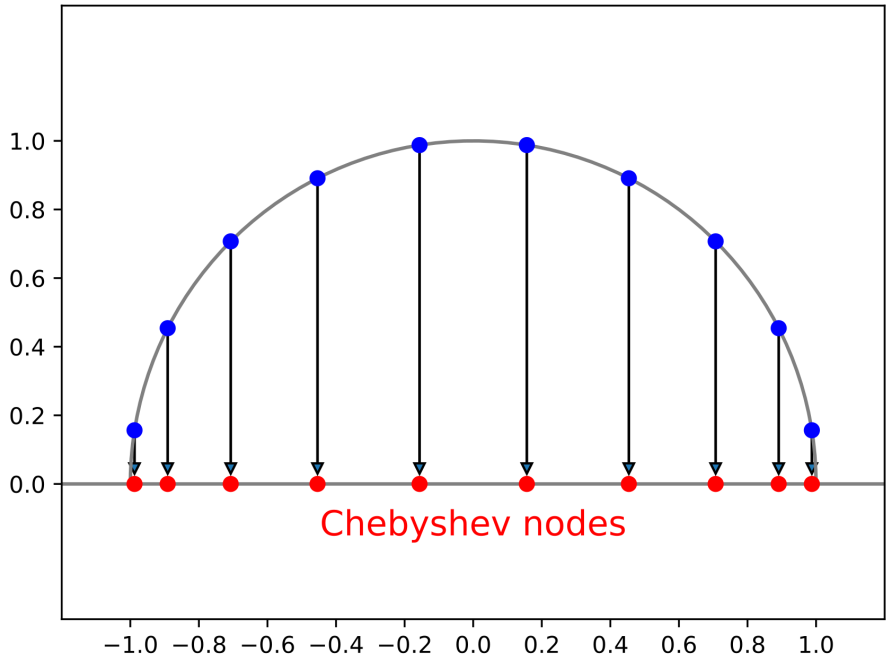


Figure 2: Chebyshev nodes are equally spaced around a semicircle. Notice how the nodes cluster about the extreme ends on the x-axis. Taken from `https://commons.wikimedia.org/wiki/File:Chebyshev-nodes-by-projection.svg`.

This indicates that amongst the three nodes distributions, Chebyshev nodes are the best for polynomial approximation and equispaced nodes are the worst. In addition, Teukolsky, Press, et al. [5] state: "The Chebyshev approximation is very nearly the same polynomial as that holy grail of approximating polynomials [which is] the minimax polynomial. [The minimax polynomial] (among all polynomials of the same degree) has the smallest maximum deviation from the true function $f(x)$. The minimax polynomial is very difficult to find; the Chebyshev approximating polynomial is almost identical and is very easy to compute!" (p. 192).

The roots of Chebyshev polynomials of the first kind, which are the Chebyshev nodes, are extremely easy to compute as compared to the roots of other Jacobi polynomials. For $N \in \mathbb{N}$, the Chebyshev nodes $x_1, x_2, \ldots x_N$ are given by the formula

$$x_k = \cos\left(\frac{2k-1}{2N}\pi\right) \text{ for } k = 1, 2, \ldots N. \tag{3}$$

Chebyshev nodes for other intervals can be found through linear transformation.

A Chebyshev polynomial approximation is of the form

$$g_N(x) = \sum_{k=1}^{N} c_k T_k(x) \text{ where } T_k \text{ is a Chebyshev polynomial of the first-kind.}$$

If g_N is a polynomial approximation of the function f, then we hope that, for an appropriate definition of the limit of a function,

$$\lim_{N\to\infty} g_N = f.$$

One possible definition for the limit of a function is the *pointwise limit*

$$\forall x \in \text{dom}(f), \lim_{N\to\infty} g_N(x) = f(x).$$

Teukolsky, Press, et al. [5] provide a formula to calculate the coefficients c_k for a Chebyshev polynomial approximation of function f:

$$c_k = \frac{2}{N} \sum_{k=1}^{N} f\left[\cos\left(\frac{2k-1}{2N}\pi\right)\right] \cos\left(\frac{2k-1}{2N}\pi\right) \tag{4}$$

Though, given theorem 1, a system of linear equations can be solved with Chebyshev nodes to attain the Chebyshev polynomial approximation. It is expected that the specialised method (4) is more computationally efficient.

3 ApproxFun

There are two components of ApproxFun which are significant to this project: its Chebyshev-based numerical integration algorithm, and its adaptive algorithm. ApproxFun's adaptive algorithm will be discussed in detail in §7.

ApproxFun uses Chebyshev zero points (given in equation 3) as its interpolation nodes. There is a second variant of Chebyshev nodes called *Chebyshev extreme points*, they will be elaborated upon in §7.1. To use ApproxFun to integrate function f from y to z, one runs the code: *Fun(f,y..z)*. The output will be a *Fun*, which represents the Chebyshev approximation polynomial, with $n \in \mathbb{N}$ number of coefficients specified in the Chebyshev space provided $(y..z)$. The number of coefficients is determined by ApproxFun's adaptive algorithm; there should be enough coefficients for the approximation to be machine precise (approximation error of around 10^{-15}).

3.1 Re-expressing a Fun in Chebyshev space as a standard polynomial

The polynomial produced by ApproxFun is in the form $c_0T_0 + c_1T_1 + \ldots$ where T_i are Chebyshev polynomials of the first kind and $c_0, c_1, \ldots$ are the coefficients produced by ApproxFun from left to right. To re-express this polynomial as a standard

polynomial, we first need to expand the T_i's and second, transform the polynomial into standard space $[-1, 1]$. For example, a *Fun* which specifies coefficients 1,2,3 in Chebyshev space $[0, 1]$ is equivalent to the polynomial $1+2(2x-1)+3(2(2x-1)^2-1) = 2 - 20x + 24x^2$.

3.2 Manual Interpolation

To perform Chebyshev polynomial approximation but bypass the the adaptive algorithm, one must first create the space by running the code: S = *Chebyshev(y..z)*, for an integration from y to z. Next, generate the Chebyshev nodes within space S by running the code: *points(S,n)*. This generates $n \in \mathbb{N}$ number of Chebyshev nodes. Values $f_1, f_2 \ldots, f_n$ must be evaluated at each node. Create the array $v = [f_1, f_2, \ldots, f_n]$. Finally, run the code: *Fun(S,ApproxFun.transform(S,v))* to generate the *Fun* with n number of coefficients (equal to the number of Chebyshev nodes generated).

The wrapper function *mcheb* which we coded allows for more convenient Chebyshev-based numerical integration with manual interpolation (details in §5.1). Further information on ApproxFun, as well as the download link, can be found at `https://github.com/JuliaApproximation/ApproxFun.jl`.

4 Splines, Composition, and Accuracy

There are two main methods by which the accuracy of a polynomial approximation can be improved. The first method is to increase the order of the approximating polynomial (though the Runge phenomenon may be a concern). The second method is to use splines.

Taken from Wouter Den Haan [3]: "The idea about splines is to split up the domain into different regions and to use a different polynomial for each region. This would be a good strategy if, the function can only be approximated well with a polynomial of a very high order over the entire domain, but can be approximated well with a sequence of low-order polynomials for different parts of the domain." (p. vii). The strategy detailed by Wouter Den Haan is called *composition*.

4.1 Composite Newton-Cotes rules

Definition 2. The four basic Newton-Cotes rules are:

Trapezoidal rule: $\int_b^a f(x)\,\mathrm{d}x \approx \frac{a-b}{2}(f_a + f_b)$.

Simpson's rule: $\int_b^a f(x)\,\mathrm{d}x \approx \frac{a-b}{3}(f_a + 4f_c + f_b)$.

Simpson's $\frac{3}{8}$ rule: $\int_b^a f(x)\,\mathrm{d}x \approx \frac{3(a-b)}{8}(f_a + 3f_c + 3f_d + f_b)$.

Boole's rule: $\int_b^a f(x)\,\mathrm{d}x \approx \frac{2(a-b)}{45}(7f_a + 32f_c + 12f_d + 32f_e + 7f_b)$. [4]

Given identical nodes, the use of these rules to perform numerical integration is more computationally efficient than the use of standard Lagrange interpolation. The problem with the use of the Newton-Cotes rules is that the number of interpolation nodes each rule uses is fixed to $2, 3, 4, 5$ respectively. Therefore, the only way to use the Newton-Cotes rules to approximate a complicated integrand to high accuracy is to perform composition, which results in the Composite Newton-Cotes rules.

Definition 3. Let $n \in \mathbb{N}$ denote the number of *divisions* of the integration interval $[b, a]$. If there are n divisions, then the integration interval is divided equally into intervals (splines) $[b, x_k], [x_k, x_{2k}], \ldots, [x_{kn-k}, x_{kn}], [x_{kn}, a]$, where $\forall i \in \mathbb{N}^0, i \leq kn + k$,

$$x_i = b + \left(\frac{i}{k}\right)\left(\frac{a-b}{n+1}\right) \text{ and}$$

k is the number of function evaluations in the rule minus one (e.g. for Simpson's rule, $k = 2$) which is constant for each rule. It is evident that $x_0 = b$ and $x_{kn+k} = a$. It is possible for the splines to be of different sizes, though that possibility is not accounted for in this definition.

Using definitions 2 and 3, the four composite Newton-Cotes rules for n divisions can be derived for all $n \in \mathbb{N}$. For simplicity, $f(x_i)$ will be denoted f_i instead of f_{x_i}. Each composite rule will be stated in two forms. The first form is simply a summation all the splines. The second form is a simplification of the first such that fewer function evaluations are needed in total, which results in greater computational efficiency.

1. **Composite Trapezoidal rule:**

$$\begin{aligned}\int_b^a f(x)\,\mathrm{d}x &\approx \frac{a-b}{2}(f_a + f_b) \\ &= \sum_{i=1}^{n+1} \frac{x_i - x_{i-1}}{2}(f_i + f_{i-1}). \end{aligned} \tag{5}$$

For example, if $n = 3$, then

$$\int_b^a f(x)\,\mathrm{d}x \approx \frac{x_1 - b}{2}(f_1 + f_b) + \frac{x_2 - x_1}{2}(f_2 + f_1) + \frac{x_3 - x_2}{2}(f_3 + f_2) + \frac{a - x_3}{2}(f_a + f_3).$$

An equivalent form is

$$\int_b^a f(x)\,\mathrm{d}x \approx \frac{x_1 - b}{2}(f_b) + \sum_{i=1}^{n} \frac{x_{i+1} - x_{i-1}}{2}(f_i) + \frac{a - x_n}{2}(f_a). \tag{6}$$

If the integration interval $[b, a]$ is divided equally, then the length of each spline is constant. Therefore, we can let the length of each spline be denoted h such that $\forall i \in \mathbb{N}^0, h = x_{i+1} - x_i$.

Substituting $h = x_{i+1} - x_i$ into equations 5 and 6,

$$(5) : \int_b^a f(x)\,\mathrm{d}x \approx \frac{h}{2}\sum_{i=1}^{n+1}(f_i + f_{i-1}) \text{ and}$$
$$(6) : \int_b^a f(x)\,\mathrm{d}x \approx \frac{h}{2}(f_b) + h\sum_{i=1}^{n}(f_i) + \frac{h}{2}(f_a).$$

2. **Composite Simpson's rule:**

$$\int_b^a f(x)\,\mathrm{d}x \approx \frac{a - b}{3}(f_a + 4f_c + f_b) = \sum_{i=1}^{n+1} \frac{x_{2i} - x_{2i-2}}{3}(f_{2i} + 4f_{2i-1} + f_{2i-2}). \tag{7}$$

Equivalently,

$$\int_b^a f(x)\,\mathrm{d}x \approx \frac{x_2 - b}{3}(f_b) + \frac{4}{3}\sum_{i=1}^{n+1}(x_{2i} - x_{2i-2})(f_{2i-1}) + \frac{1}{3}\sum_{i=1}^{n}(x_{2i+2} - x_{2i-2})(f_{2i}) + \frac{a - x_{2n}}{3}(f_a). \tag{8}$$

If the division is equal, then let $h = x_{2i+2} - x_{2i}$. Therefore,

$$(7) : \int_b^a f(x)\,\mathrm{d}x \approx \frac{h}{3}\sum_{i=1}^{n+1}(f_{2i} + 4f_{2i-1} + f_{2i-2}) \text{ and}$$
$$(8) : \int_b^a f(x)\,\mathrm{d}x \approx \frac{h}{3}(f_b) + \frac{4h}{3}\sum_{i=1}^{n+1}(f_{2i-1}) + \frac{2h}{3}\sum_{i=1}^{n}(f_{2i}) + \frac{h}{3}(f_a).$$

3. **Composite Simpson's $\frac{3}{8}$ rule:**

$$\int_b^a f(x)\,\mathrm{d}x \approx \frac{3(a-b)}{8}(f_a + 3f_c + 3f_d + f_b)$$
$$= \sum_{i=1}^{n+1} \frac{3(x_{3i} - x_{3i-3})}{8}(f_{3i} + 3f_{3i-1} + 3f_{3i-2} + f_{3i-3}). \tag{9}$$

Equivalently,

$$\int_b^a f(x)\,\mathrm{d}x \approx \frac{3(x_3 - b)}{8}(f_b) + \frac{9}{8}\sum_{i=1}^{n+1}(x_{3i} - x_{3i-3})(f_{3i-1} + f_{3i-2})$$
$$+ \frac{3}{8}\sum_{i=1}^{n}(x_{3i+3} - x_{3i-3})(f_{3i}) + \frac{3(a - x_{3n})}{8}(f_a). \tag{10}$$

If the division is equal, then let $h = x_{3i+3} - x_{3i}$. Therefore,

$$(9) : \int_b^a f(x)\,\mathrm{d}x \approx \frac{3h}{8}\sum_{i=1}^{n+1}(f_{3i} + 3f_{3i-1} + 3f_{3i-2} + f_{3i-3}) \text{ and}$$
$$(10) : \int_b^a f(x)\,\mathrm{d}x \approx \frac{3h}{8}(f_b) + \frac{9h}{8}\sum_{i=1}^{n+1}(f_{3i-1} + f_{3i-2}) + \frac{3h}{4}\sum_{i=1}^{n}(f_{3i}) + \frac{3h}{8}(f_a).$$

4. **Composite Boole's rule:**

$$\int_b^a f(x)\,\mathrm{d}x \approx \frac{2(a-b)}{45}(7f_a + 32f_c + 12f_d + 32f_e + 7f_b)$$
$$= \sum_{i=1}^{n+1} \frac{2(x_{4i} - x_{4i-4})}{45}(7f_{4i} + 32f_{4i-1} + 12f_{4i-2} + 32f_{4i-3} + 7f_{4i-4}). \tag{11}$$

Equivalently,

$$\int_b^a f(x)\,\mathrm{d}x \approx \frac{14(x_4 - b)}{45}(f_b) + \frac{8}{45}\sum_{i=1}^{n+1}(x_{4i} - x_{4i-4})(8f_{4i-1} + 3f_{4i-2} + 8f_{4i-3})$$
$$+ \frac{14}{45}\sum_{i=1}^{n}(x_{4i+4} - x_{4i-4})(f_{4i}) + \frac{14(a - x_{4n})}{45}(f_a). \tag{12}$$

If the division is equal, then let $h = x_{4i+4} - x_{4i}$. Therefore,

$$(11) : \int_b^a f(x)\,\mathrm{d}x \approx \frac{2h}{45}\sum_{i=1}^{n+1}(7f_{4i} + 32f_{4i-1} + 12f_{4i-2} + 32f_{4i-3} + 7f_{4i-4}) \text{ and}$$

$$(12) : \int_b^a f(x)\,\mathrm{d}x \approx \frac{14h}{45}(f_b) + \frac{8h}{45}\sum_{i=1}^{n+1}(8f_{4i-1} + 3f_{4i-2} + 8f_{4i-3}) + \frac{28h}{45}\sum_{i=1}^{n}(f_{4i}) + \frac{14h}{45}(f_a).$$

The total number of function evaluations in the second form of each composite rule, E, is given by the formula $E = (k+1)(n+1) - n = kn + k + 1 = k(n+1) + 1$.

The second form of each composite Newton-Cotes rule has been implemented in the numerical integration algorithms we coded (see §5).

4.2 Relationships between the number of splines and the accuracy of the numerical integration

Splines can also be used to improve the accuracy of other numerical integration algorithms. In fact, splines double-up as a solution to the Runge phenomenon. We investigated the relationship between the number of splines used and the accuracy of the numerical integration for Newton-Cotes rules, Taylor polynomial, and Chebyshev polynomial based algorithms. We discovered that for some functions, Newton-Cotes rules and Taylor polynomial based algorithms demonstrate the relationship

$$E = \frac{e^a}{n^b} \qquad \Leftrightarrow \qquad \log(E) = a + b\log(n), \tag{13}$$

where a is a constant, n is the number of splines, b is the number of function evaluations (or one more than the order of the Taylor polynomial), and E is the error of the numerical integration defined by $E = |\text{actual value} - \text{numerical integration value}|$.

We have named each of our graphs "[function denotation][r value in function][algorithm denotation]". The first function we experimented with is

$$s[r] : \sin(10^r x).$$

The algorithms are denoted "tay[ω]" for Taylor polynomial with order ω (if no number is specified, order is 15 by default), "b" denotes Boole's rule, "s38" denotes Simpson's $\frac{3}{8}$ rule, and "s" denotes Simpson's rule. As an example, the graph "s2s38" denotes that the function $\sin(10^2 x)$ and Simpson's $\frac{3}{8}$ rule were used.

Figure 3 shows the graph named "s2s38". Apart from a few data points to the extreme left, the graph appears to adhere strictly to relationship (13); we performed linear regression using RStudio and found a p-value which is smaller than machine precision, and an adjusted R-squared value of 0.9985. In addition, we found that $a = 3.96$ and $b = 4$. Figure 4 shows the another graph named "s5tay11", for which $a = 102$ and $b = 12$.

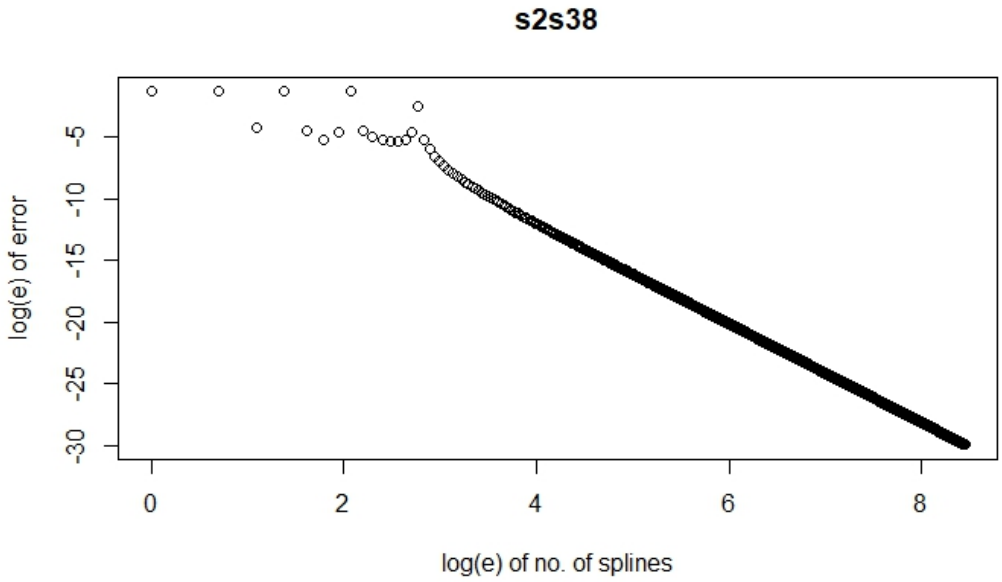


Figure 3: The function $\sin(10^2 x)$ and Simpson's $\frac{3}{8}$ rule were used. $a = 3.96$ and $b = 4$.

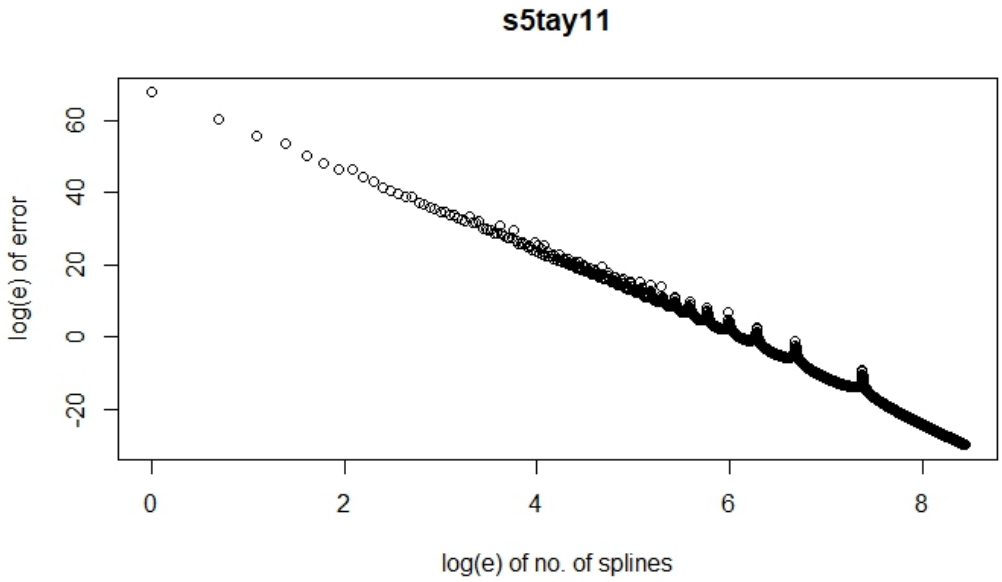


Figure 4: The function $\sin(10^5 x)$ and a Taylor polynomial with order 11 were used. $a = 102$ and $b = 12$.

Name of graph	b (nearest integer)	a (3s.f.)
s1s	4	-8.74
s1s38	4	-9.55
s1b	6	-15.2
s2s	4	3.96
s2s38	4	3.96
s2b	6	6.57
s3s	4	11.9
s3s38	4	10.9
s3b	6	19.3
s4s	4	20.2
s4s38	4	19.2
s4b	6	33.1
s4tay	16	97.3
s5b	6	46.3
s5tay	16	131

Table 1: a and b values for relationship (13)

Table 1 shows the values of b and a for functions s[r]. As expected, the b values equal the number of function evaluations in the algorithm (or one more than the order of the Taylor polynomial). The a values tend to be larger as the function becomes more oscillatory (r is larger). In fact, for each algorithm, the a value increases linearly as r in $\sin(10^r x)$ increases. Figure 5 shows the linear relationship between a and r, with gradient equal to 15.0 (3s.f.). The linear relationship and relationship (13) are significant as given a single data point (a, r), they allow us to estimate the number of splines, n, such that error, $E \leq T$ for any $T \in \mathbb{R}^+$, for any non-zero real number r where $\sin(10^r x)$ is the function to be numerically integrated with respect to x.

Unfortunately, these relationships break down as the function to be numerically integrated becomes more complicated: the pattern of oscillation shows greater variation. For the function

$$\text{ss[r]: } \sin\left(10^r x + \frac{\sin(10^r x)}{10}\right),$$

the relationships still hold, albeit with more discrepancies. Figure 6 shows that relationship (13) holds firmly for error less than approximately e^{-14}. In fact, when error is small, the linear relationship between a and r holds firmly as well. Yet, bewteen figure 3 and figure 6, the latter has a significantly larger percentage of data points which do not adhere to relationship (13).

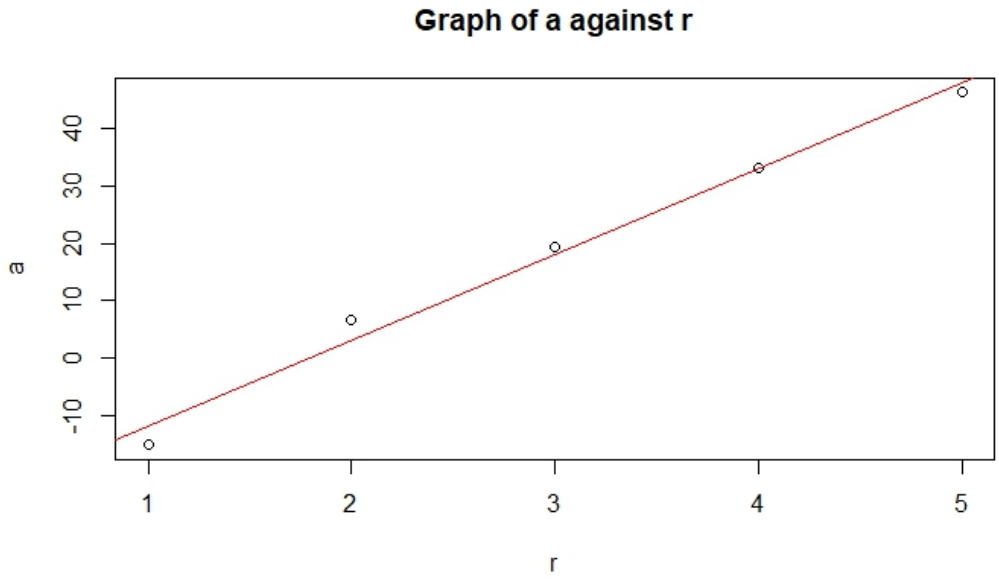


Figure 5: The linear relationship between constant a and r in $\sin(10^r x)$. Gradient is equal to 15.0 (3s.f.).

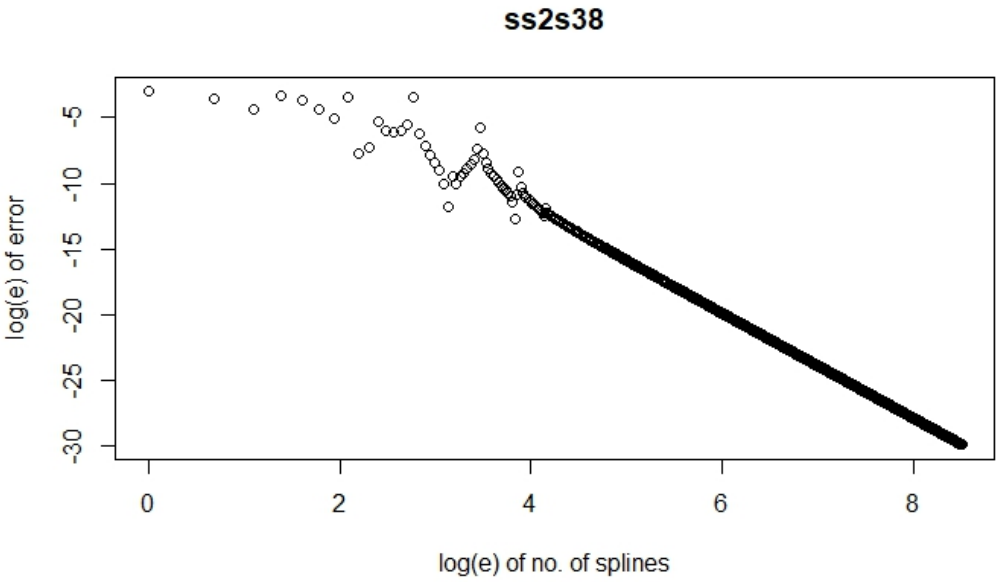


Figure 6: The function $\sin\left(10^2 x + \frac{\sin(10^2 x)}{10}\right)$ and Simpson's $\frac{3}{8}$ rule were used. $a = 4.02$ and $b = 4$.

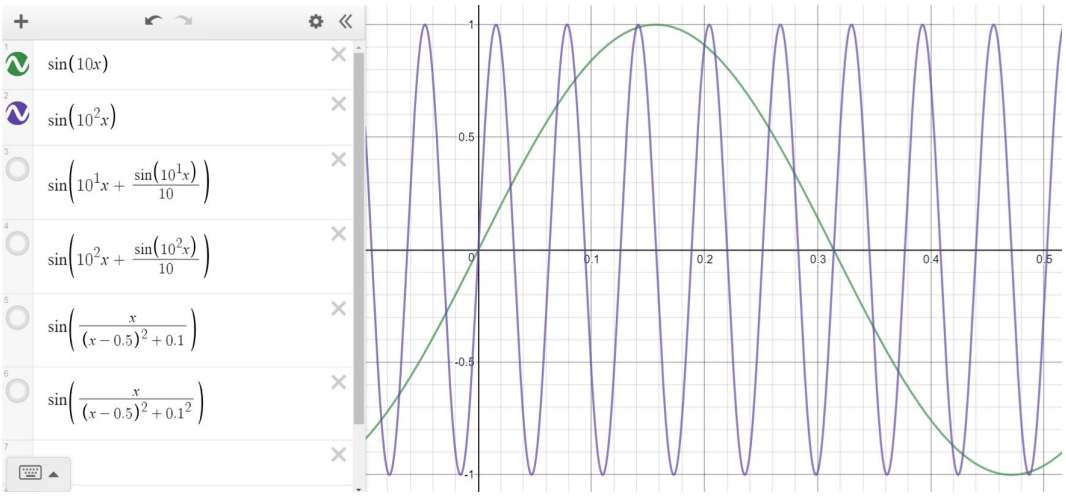


Figure 7: Graph of the function $s[r]$ for $r = 1, 2$.

The function

$$\text{am[r]: } \sin\left(\frac{x}{(x-0.5)^2 + 0.1^r}\right)$$

is much more complicated than previous functions because its behaviour ranges from that of $\sin(4x)$ at $x = 0$ or $x = 1$ to that of $\sin(10^r x)$ at $x = 0.5$. When we used the function "am[r]", relationship (13) no longer held for some numerical integration algorithms as demonstrated in figure 12. Notably, for some numerical integration algorithms, for sufficiently small values of r, for extremely small error values, the data points still adhere to relationship (13). This is shown in figure 13. Nonetheless, we cannot reliably estimate the number of splines necessary to bring the error below a certain value unless prior testing has shown that relationship (13) and the linear relationship between a and r hold for the function to be numerically integrated.

The figures 7, 8, 9 show the graphs for $s[r], ss[r], am[r]$ respectively for $r = 1, 2$. Figure 10 shows all of the graphs together. For larger values of r the graph is barely visible. For example, figure 11 shows the graphs for $ss[r]$ and $am[r]$ for $r = 3$.

When we used Chebyshev polynomial based algorithms, relationship (13) did not hold for every function experimented with. Figure 14 shows the results from experimentation with function $\sin(10^2 x)$ and Chebyshev polynomial approximations with order 15.

4.3 Conclusion on the use of splines and on Taylor polynomial based algorithms

Our experimentation with the various numerical integration algorithms led us to two conclusions. First, we decided to dismiss the use of Taylor polynomials for numerical integration for two reasons.

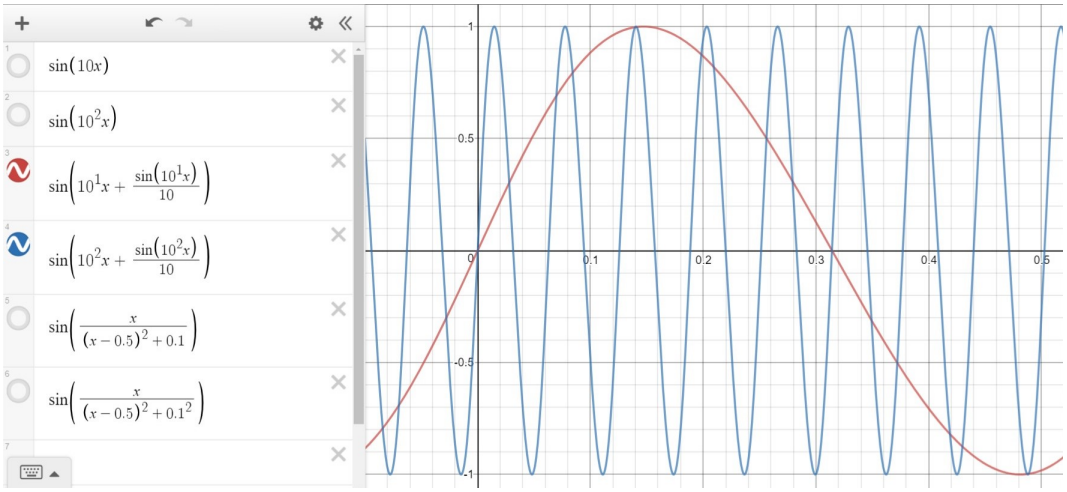


Figure 8: Graph of the function $ss[r]$ for $r = 1, 2$.

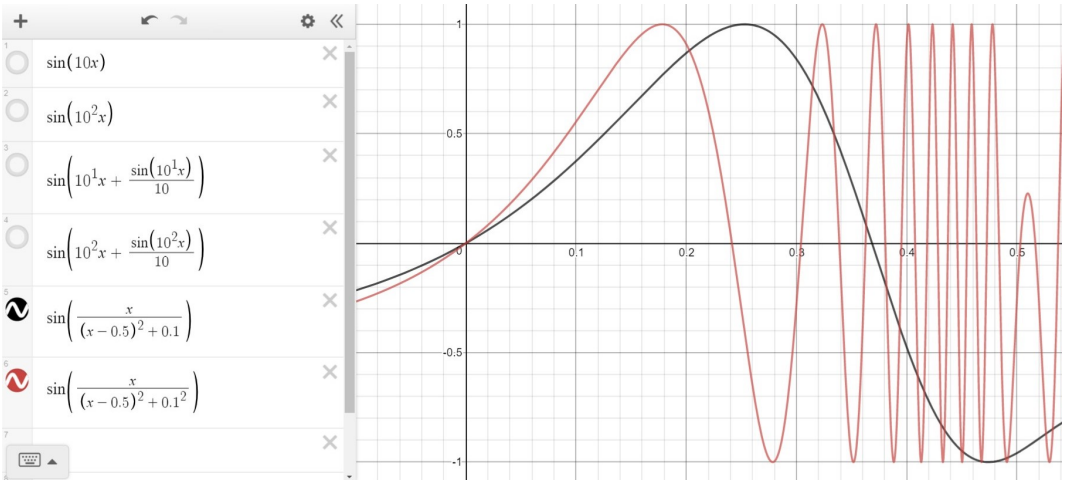


Figure 9: Graph of the function $am[r]$ for $r = 1, 2$.

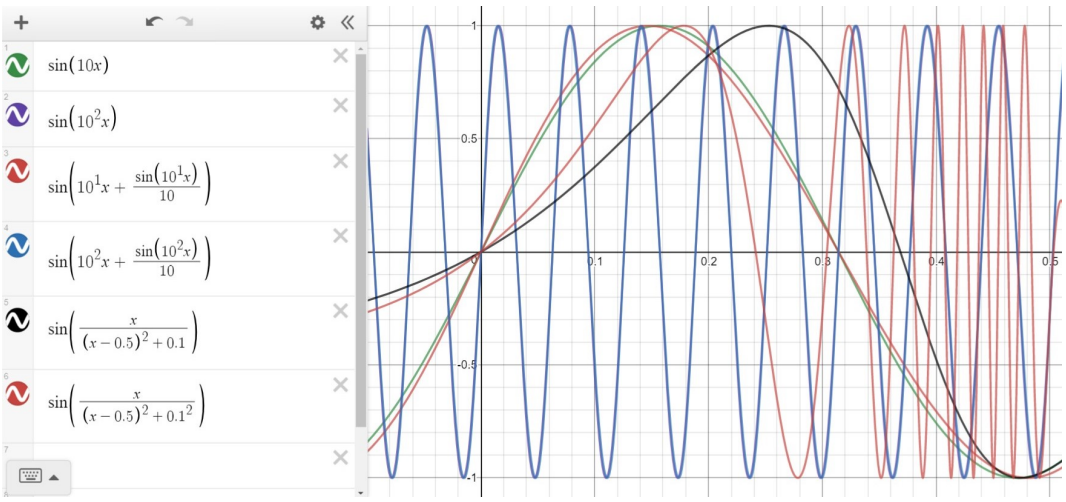


Figure 10: Graphs of the functions $s[r], ss[r], am[r]$ for $r = 1, 2$.

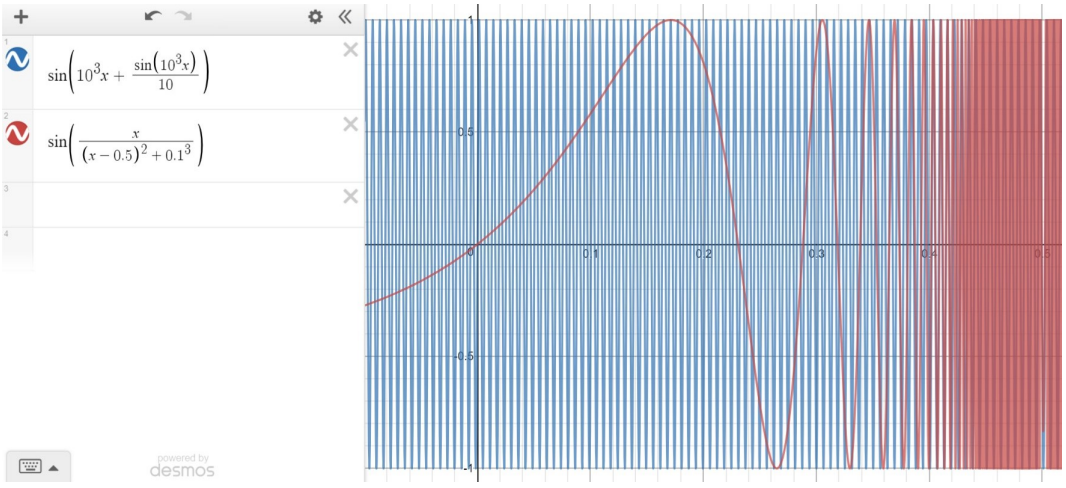


Figure 11: Graphs of the functions $ss[r], am[r]$ for $r = 3$.

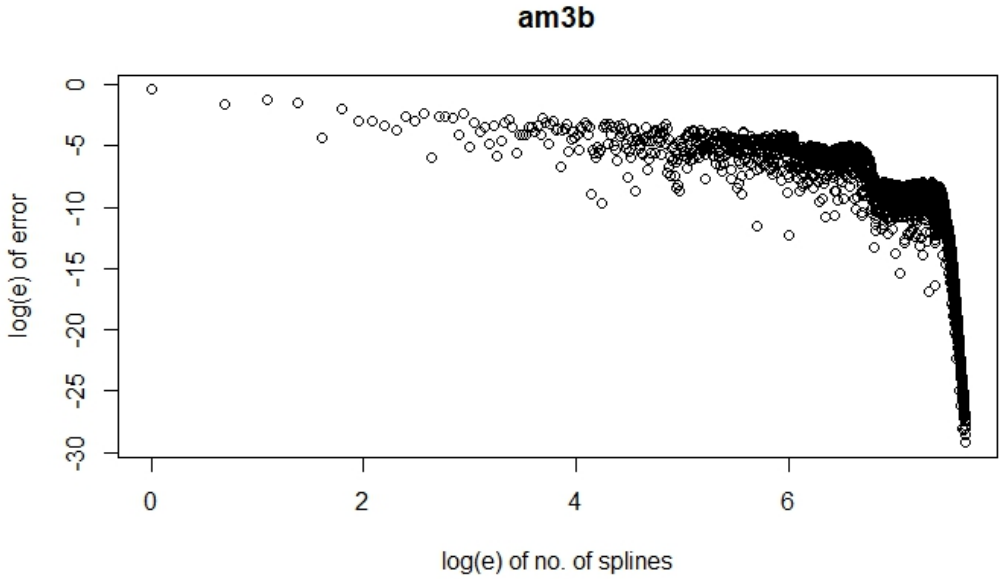


Figure 12: The function $\sin\left(\frac{x}{(x-0.5)^2+0.1^3}\right)$ and Boole's rule were used. The graph is no longer linear.

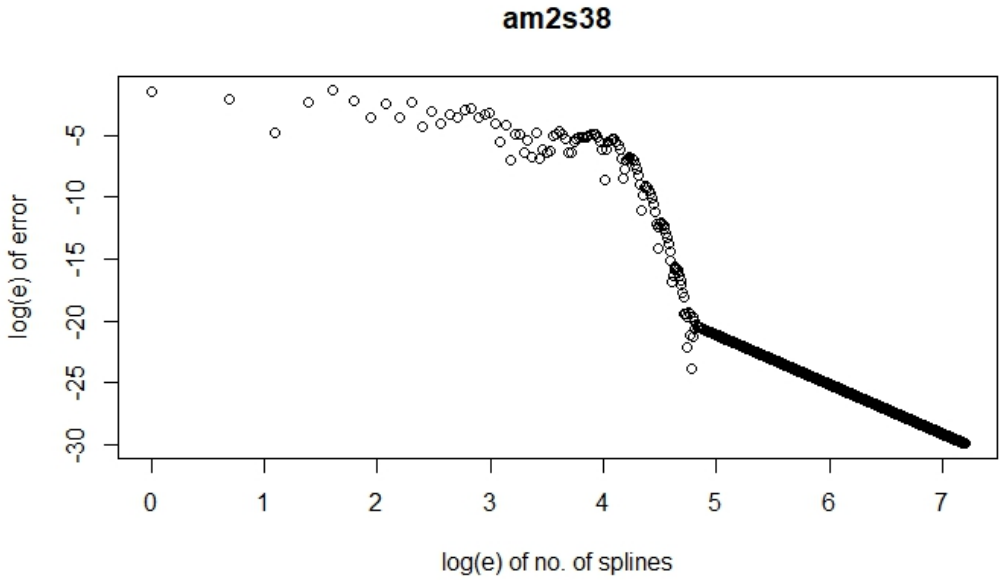


Figure 13: The function $\sin\left(\frac{x}{(x-0.5)^2+0.1^2}\right)$ and Simpson's $\frac{3}{8}$ rule were used. The graph is mostly non-linear.

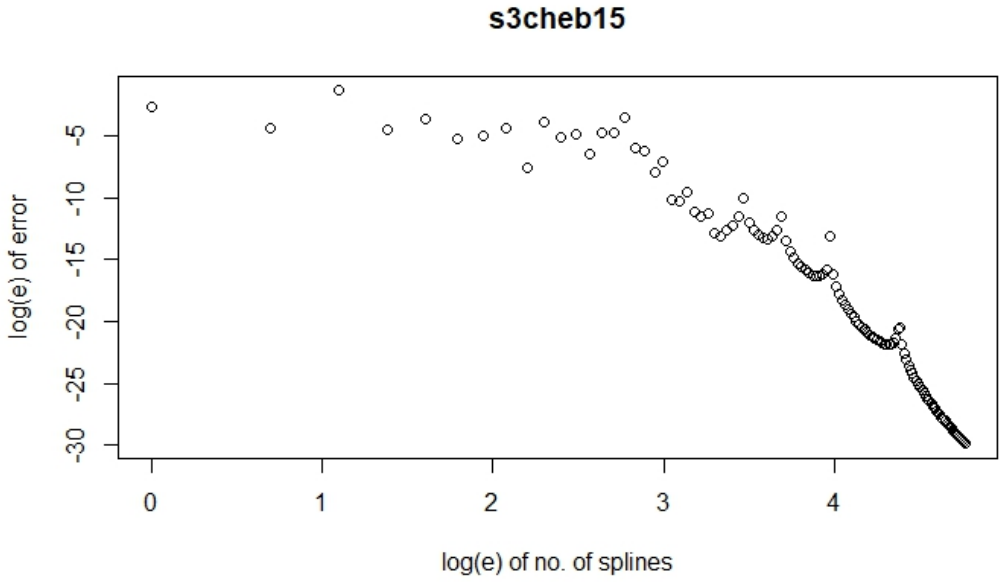


Figure 14: The function $\sin(10^2 x)$ and a Chebyshev polynomial approximation with order 15 were used. The graph is non-linear.

With reference again to table 1, notice that when Taylor polynomial based algorithms are used, the value of a is massive. This manifests as a massive error when the number of splines is small. With reference to figure 4, the error is approximately e^{60} when no composition is performed. Graphically, this is because, outside of a small range from the point of Taylor expansion, a Taylor polynomial approximation diverges significantly from the function; an example of such divergence is shown in figure 15. Obviously, this is a problem for numerical integration.

Moreover, Taylor expansion relies upon differentiation, this is opposed to polynomial interpolation which simply needs to evaluate the function. Differentiation is much more computationally expensive than evaluation. Moreover, some functions are finitely-differentiable or undifferentiable altogether. Therefore, polynomial interpolation based algorithms are more versatile and more computationally efficient.

Our second conclusion is that the use of splines is an effective, and possibly predictable, method of increasing the accuracy of a numerical integration. Unlike our formation of splines by equal division of the integration interval, splines are usually created by an adaptive algorithm. Also, the standard method to increase the accuracy of a numerical integration is to increase the order of the approximating polynomial. More research is required to compare these standard methods to our methods involving the creation and use of splines.

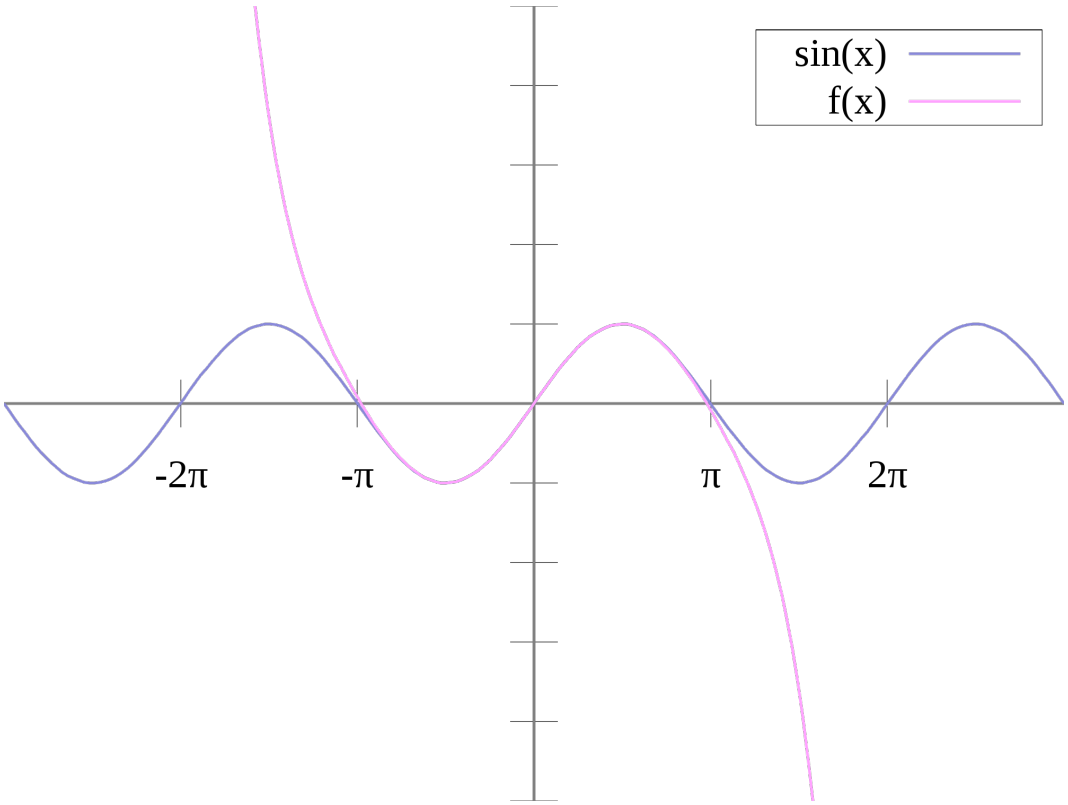


Figure 15: The Taylor polynomial approximation of order 7 diverges significantly from the function for $-\pi > x$ and $\pi < x$. The Taylor expansion is centred on the origin. Taken from `https://commons.wikimedia.org/wiki/File:Taylorsine.svg`.

5 Our algorithms and ApproxFun wrapper functions

We coded seven numerical integration algorithms in the Julia programming language. Four of the algorithms: "trapezoid", "simpson", "simpson38", "boole", are based on the Newton-Cotes rules. Their syntax is *[algorithm](f::function, y::number, z::number, d::Int64)* for integrate function f from y to z. d is an optional argument for the number of divisions of the integration interval, which is one less than the number of splines. If d is unspecified, d is set to 0 by default.

The algorithm "tayloridea" uses Taylor polynomials. Its syntax is *tayloridea(f::function, y::number, z::number, n::Int64, d::Int64)*, where the additional argument n is the order of the approximating polynomial used in each spline. For "tayloridea", the highest value n can be is 15 due to limitations of code.

Finally, the last two algorithms are "fullm" and "linear". "fullm" allows for any interpolation nodes to be specified; Lagrange interpolation is performed with these nodes. Be careful with the use of "fullm", for not every distribution of interpolation nodes will result in an approximating polynomial which convergences uniformly to the function as the number of nodes approach infinity; the Runge phenomenon is one example of divergence (see §2). "linear" performs standard Lagrange interpolation (see §1 for more details). "linear" has identical syntax to "tayloridea". "fullm" requires an array argument which will be elaborated upon below.

Six of the seven algorithms are able to accept an array argument in lieu of arguments y, z and d. "fullm" requires an array argument and has the syntax *fullm(f::Function, y::Number, z::Number, C::Array)*. The array argument, usually denoted C, demarcates the splines to be used in the numerical integration; for "fullm", C is an array of interpolation nodes. For the Newton-Cotes rules based algorithms, the syntax for the use of an array argument is *[algorithm](f::function, C::Array)*. As an example, the code *simpson38*$(x \to x^2, 0, 1, 4)$ is equivalent to *simpson38*$(x \to x^2, [0, 0.2, 0.4, 0.6, 0.8, 1])$. The use of array arguments is especially important for contour integration (see §5.2).

5.1 Efficient use of ApproxFun and its wrapper functions

Two primary wrapper functions were coded for ApproxFun: "mcheb" and "completecheb". "mcheb" syntax is *mcheb(f::function, y::number, z::number, n::Int64, d::Int64)*, identical to that of "tayloridea". "mcheb" bypasses ApproxFun's in-built adaptive algorithm. Therefore, if we know both the order of the Chebyshev approximation polynomial and the number of splines which are necessary to attain the desired accuracy, "mcheb" is more computationally efficient than ApproxFun. Examples of situations in which "mcheb" is more computationally efficient than ApproxFun will be given below.

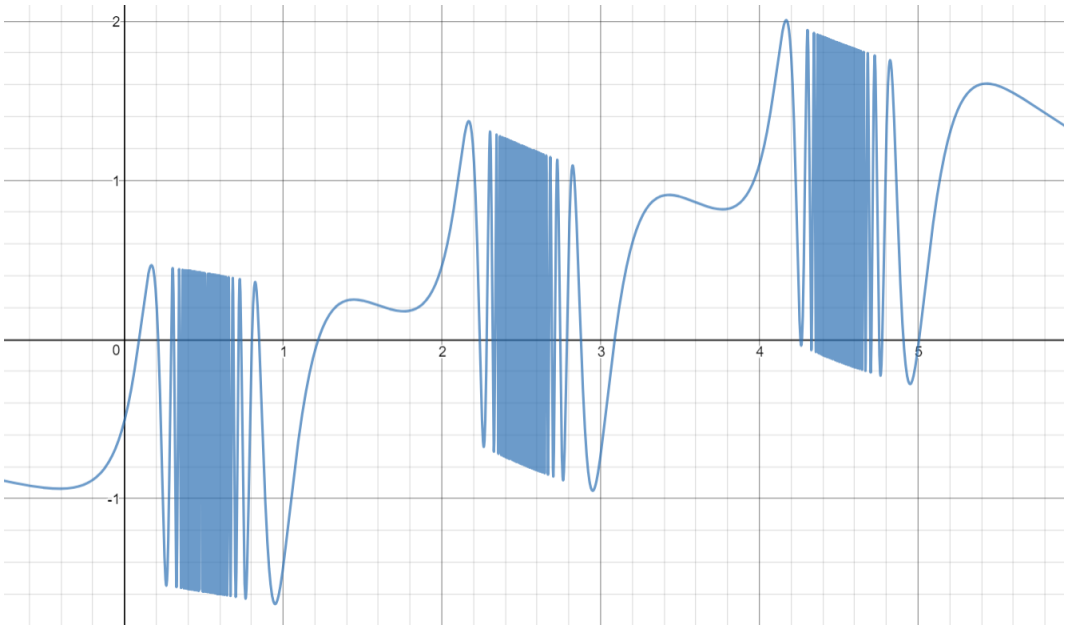


Figure 16: Function "am4,5,6" (14) graphed in Desmos (`https://www.desmos.com`).

We will be using the following two functions:

$$\text{am4,5,6:}\quad \sum_{r=0}^{2} \sin\left(\frac{x-2\times r}{(x-2\times r-0.5)^2+0.1^{r+4}}\right) \tag{14}$$

$$\text{am2,4,6:}\quad \sum_{r=0}^{2} \sin\left(\frac{x-2\times r}{(x-2\times r-0.5)^2+0.1^{2r+2}}\right). \tag{15}$$

Figures 16 and 17 show the graph of the respective functions. These functions consist of simple parts and highly oscillatory parts, each of which can be split up to form splines. Attempting to numerically integrate the entire integration interval, which is the method denoted "AdaptallN", requires an approximating polynomial with a massively high order. "AdaptallN" accuracy will be shown to be abyssmal as the highest order polynomial ApproxFun's adaptive algorithm can generate is a polynomial of order 2097151, and even it could not attain the desired accuracy. On the other hand, using the adaptive algorithm on every spline, which is the method denoted "AdaptallC", is computationally expensive.

Our solution is to re-use the approximating polynomial's order and bypass the adaptive algorithm when possible. Consider, for example, that we have three splines, each of which contains a highly oscillatory part of the graph. Assume we require an approximating polynomial of order 1000000, 1500000 and 2000000 to accurately approximate the respective splines. We can instead use an approximating polynomial of order 2000000 to approximate each of the three splines and ensure we attain at least

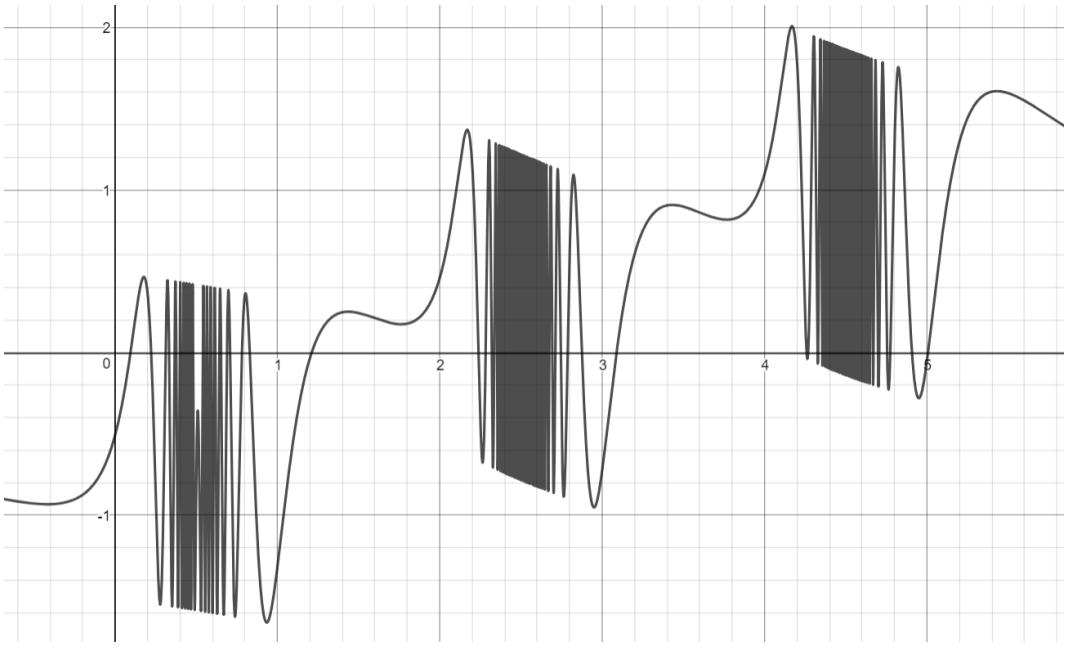


Figure 17: Function "am2,4,6" (15) graphed in Desmos (`https://www.desmos.com`).

the desired accuracy. Therefore, if there are splines which are similarly difficult to numerically integrate, and we can identify the spline which is most difficult, or representative of the difficulty, to numerically integrate, we can use the adaptive algorithm on only that spline and re-use the approximating polynomial's order to numerically integrate similar splines. This idea underlies the wrapper function "completecheb". "completecheb" has the syntax *completecheb(f::Function,C::Array,A::Array, X)*. Array C demarcates the splines to be used in the numerical integration. Array A specifies the *positions* (from left to right) of the splines on which to use the adaptive algorithm. X can be an array of the positions of the splines which approximating polynomial's order will be re-used for the remaining splines. If X is a multi-element array, the average order of the approximating polynomials will be used. Alternatively, X can be an integer to specify the order to be used for the approximating polynomials. Two examples:

1. *completecheb*$(f, [0, 0.5, 0.75, 1], [1], [4])$ means the spline $[0, 0.5]$ will be numerically integrated with the use of the adaptive algorithm. The spline $[0.75, 1]$ will also be numerically integrated with the use of the adaptive algorithm, moreover, the order of its approximating polynomial will be re-used to numerically integrate the spline $[0.5, 0.75]$. The output is the sum of the numerical integration of each spline.

2. *completecheb*$(f, [0, 0.5, 0.75, 1], [1, 3], 100)$ means the splines $[0, 0.5]$ and $[0.75, 1]$ will be numerically integrated with the use of the adaptive algorithm. The last

Graph to approximate: am2,4,6				
Method	**Median Time**	**Error (Compared to AdaptallC)**	**Method Description**	
AdaptallC	4.619s	NA	Use of adaptive algorithm for every spline.	
AdaptallN	3.996s	0.000245	Normal usage of ApproxFun; this is unlikely to be usable for contour integration.	
Adapt1	**4.445s**	**8.88*10^-16**	**Use of adaptive algorithm for the splines with highly oscillatory parts only.**	**BEST**
Adapt2	10.779s	1.42*10^-14	Use the highest order necessary for every spline.	
Adapt3	7.742s	1.51*10^-14	Use the highest order necessary for every splines with highly oscillatory parts, and the highest order necessary for every remaining splines.	

Graph to approximate: am4,5,6				
Method	**Median Time**	**Error (Compared to AdaptallC)**	**Method Description**	
AdaptallC	9.511s	NA	Use of adaptive algorithm for every spline.	
AdaptallN	4.725s	0.000922	Normal usage of ApproxFun; this is unlikely to be usable for contour integration.	
Adapt1	9.682s	0	Use of adaptive algorithm for the splines with highly oscillatory parts only.	
Adapt2	12.842s	1.78*10^-15	Use the highest order necessary for every spline.	
Adapt3	**9.135s**	**0**	**Use the highest order necessary for every splines with highly oscillatory parts, and the highest order necessary for every remaining splines.**	**BEST**

Figure 18: The results of the comparison between "completecheb" and ApproxFun

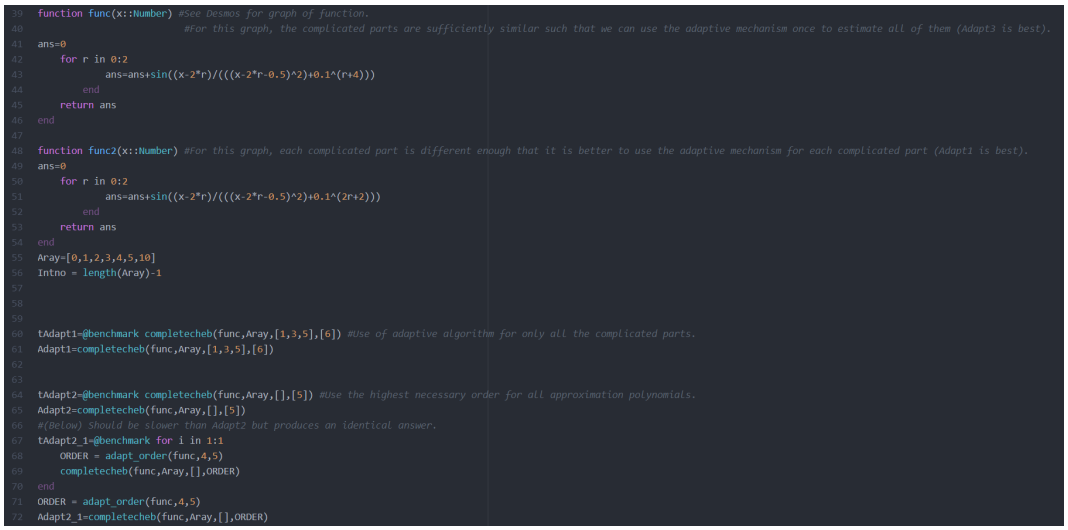
spline $[0.5, 0.75]$ will be numerically integrated using a Chebyshev polynomial approximation with order 100. The output is the sum of the numerical integration of each spline.

One peripheral wrapper function we coded for ApproxFun is "adapt_order", it outputs the order of the approximating polynomial generated by ApproxFun's adaptive algorithm for a particular numerical integration. "adapt_order" has the syntax *adapt_order(f::Function,y::Number, z::Number)* for integrate function f from y to z.

We tested "completecheb" against ApproxFun using the BenchmarkTools software package in Julia (see `https://github.com/JuliaCI/BenchmarkTools.jl` for software details and the download link). The results we attained is detailed in figure 18. The code we ran for the tests is detailed in figures 20 and **??**. For function "am4,5,6" (14), the highly oscillatory parts were similar, and the non-oscillatory parts were also similar, therefore, "completecheb" performed entirely as expected. For function "am2,4,6" (15), the highly oscillatory parts were sufficiently different that it was better to use the adaptive algorithm for those splines.

5.2 Contour Integration

Suppose we wish to use numerical integration to approximate $\int_C \frac{1}{z}\,\mathrm{d}z$, where z is a complex variable and C is the upper half of the counterclockwise unit circle $|z = 1|$



```
function func(x::Number) #See Desmos for graph of function.
                         #For this graph, the complicated parts are sufficiently similar such that we can use the adaptive mechanism once to estimate all of them (Adapt3 is best).
ans=0
    for r in 0:2
            ans=ans+sin((x-2*r)/(((x-2*r-0.5)^2)+0.1^(r+4)))
        end
    return ans
end

function func2(x::Number) #For this graph, each complicated part is different enough that it is better to use the adaptive mechanism for each complicated part (Adapt1 is best).
ans=0
    for r in 0:2
            ans=ans+sin((x-2*r)/(((x-2*r-0.5)^2)+0.1^(2r+2)))
        end
    return ans
end
Aray=[0,1,2,3,4,5,10]
Intno = length(Aray)-1



tAdapt1=@benchmark completecheb(func,Aray,[1,3,5],[6]) #Use of adaptive algorithm for only all the complicated parts.
Adapt1=completecheb(func,Aray,[1,3,5],[6])


tAdapt2=@benchmark completecheb(func,Aray,[],[5]) #Use the highest necessary order for all approximation polynomials.
Adapt2=completecheb(func,Aray,[],[5])
#(Below) Should be slower than Adapt2 but produces an identical answer.
tAdapt2_1=@benchmark for i in 1:1
    ORDER = adapt_order(func,4,5)
    completecheb(func,Aray,[],ORDER)
end
ORDER = adapt_order(func,4,5)
Adapt2_1=completecheb(func,Aray,[],ORDER)
```

Figure 19: Part 1 of the code we ran to test "completecheb" against ApproxFun.

```
ORDER = adapt_order(func,4,5)
Adapt2_1=completecheb(func,Aray,[],ORDER)


Adapt3=0
#Use the highest order necessary for all the complicated parts and
#the highest order necessary for all the simple parts.
tAdapt3=@benchmark for i in 1:1
    Adapt3=0
    ORDERC = adapt_order(func,Aray[5],Aray[6])
    ORDERS = adapt_order(func,Aray[6],Aray[7])
    for r in 0:2
        Adapt3=Adapt3+completecheb(func,[Aray[2r+1],Aray[2r+2]],[],ORDERC)
    end
    for r in 0:2
        Adapt3=Adapt3+completecheb(func,[Aray[2r+2],Aray[2r+3]],[],ORDERS)
    end
end
ORDERC = adapt_order(func,Aray[5],Aray[6])
ORDERS = adapt_order(func,Aray[6],Aray[7])
for r in 0:2
    Adapt3=Adapt3+completecheb(func,[Aray[2r+1],Aray[2r+2]],[],ORDERC)
end
for r in 0:2
    Adapt3=Adapt3+completecheb(func,[Aray[2r+2],Aray[2r+3]],[],ORDERS)
end


tAdaptallC=@benchmark completecheb(func,Aray,[1,2,3,4,5,6,7],[]) #Use of adaptive algorithm on every integration interval.
AdaptallC=completecheb(func,Aray,[1,2,3,4,5,6,7],[])


tAdaptallN=@benchmark AdaptallN=sum(Fun(func,Aray[1]..Aray[end])) #Normal usage of ApproxFun; this is unlikely to be usable for contour integration.
AdaptallN=sum(Fun(func,Aray[1]..Aray[end]))
```

Figure 20: Part 2 of the code we ran to test "completecheb" against ApproxFun.

Exact value	-3.141592654i
Chebyshev nodes	-3.141610132i
Linear-spaced nodes	-3.141878418i

Table 2: Results for line segment based contour integration with six nodes distributed about each line segment.

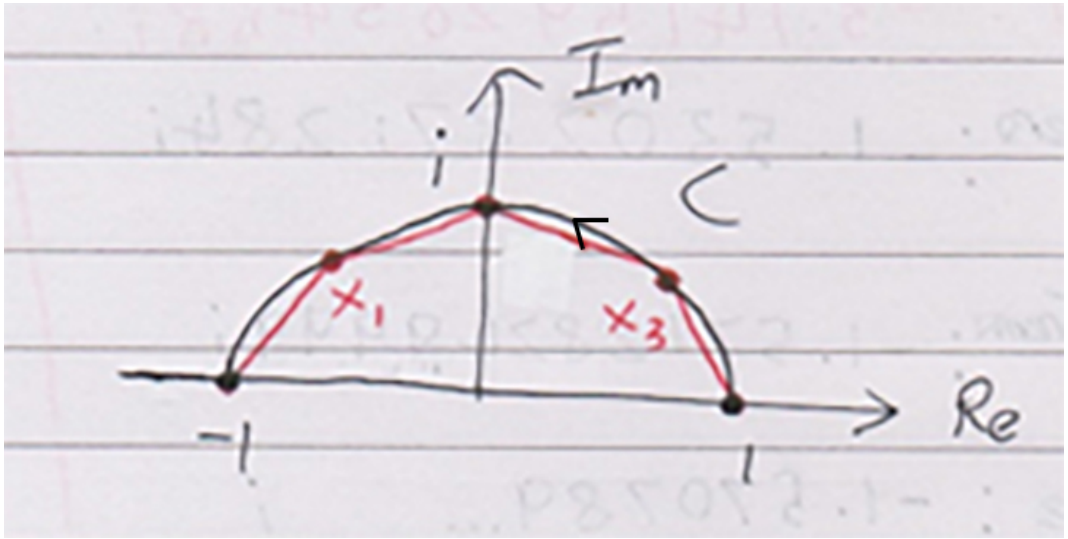


Figure 21: Contour, C, approximated by four line segments. Image is for illustration purposes only, it is not drawn accurately.

such that $\text{Im}(z) \geq 0$. This can be accomplished either by the use of line segments, or by contour interpolation. We coded our algorithms to accomodate both methods. The actual value of the integration is $-\frac{\pi}{2}i$, where i is the imaginary constant.

We can approximate the contour, C, with line segments. Figure 21 shows C approximated by four line segments. An array argument can be input into any of the aforementioned algorithms or wrapper functions, with each line segment considered as one spline. An example of the code for the numerical integration is $mcheb(z \to \frac{1}{z}, [-1, x_1, i, x_3, 1], 5)$, where an order of five requires six interpolation nodes. Table 2 shows the results for line segment based contour integration with six nodes distributed about each line segment.

Alternatively, we can perform contour interpolation. If we take the Chebyshev nodes on contour, C, as shown in figure 22, we can perform numerical integration as usual. We can take linear-spaced nodes on C as well. While ApproxFun does not support contour interpolation, we can use the algorithm "fullm" to perform Lagrange interpolation through the nodes and numerically integrate using a complex Chebyshev approximation polynomial (see §5 for details on algorithm "fullm", and see §1 for details on Lagrange interpolation). Table 3 shows the results for con-

Exact value	-3.141592654i
Chebyshev nodes	-3.085115443i
Linear-spaced nodes	-3.158259736i

Table 3: Results for contour interpolation based integration with 24 nodes distributed about the integration interval.

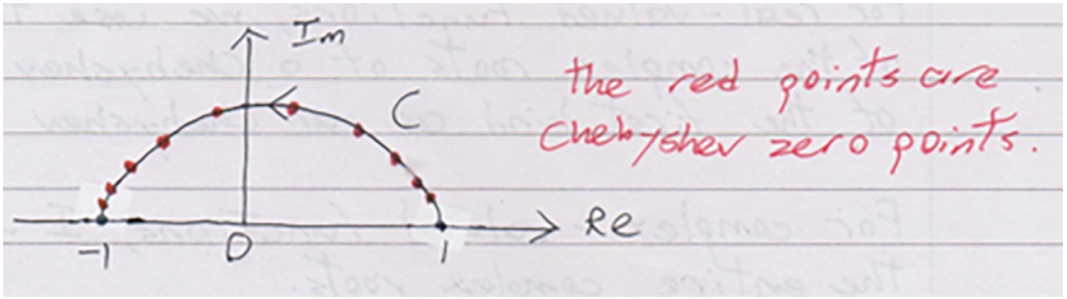


Figure 22: Contour, C, with Chebyshev zero points (standard Chebyshev nodes) distributed about it. Image is for illustration purposes only, it is not drawn accurately.

tour interpolation based integration with 24 nodes distributed about the integration interval.

Contrary to our expectations, the use of a mere four line segments produced significantly more accurate results than contour interpolation based integration. Further research can be done to explain the results attained. Nonetheless, we will use line segments to perform numerical integration on a contour.

6 The time-dependent linear Schrodinger equation

The time-dependent linear Schrodinger equation is a complex partial differential equation used to model quantum systems with wavefunctions which evolve with time. We aim to plot the solution to the equation for a variety of initial conditions.

The problem is given as follows:

$$\text{Partial Differential Equation, PDE: } [\partial_t + i(-i\partial_x)^2]q(x,t) = q_t - iq_{xx} = 0 \tag{16}$$

$$\text{Initial Condition, IC: } q(x,0) = q_0(x) \tag{17}$$

$$\text{Boundary Condition, BC: } q_x(0,t) + \beta q(0,t) = h(t). \tag{18}$$

From the PDE (16) and IC (17), we derived

$$\text{Global Relation, GR: } 0 = \hat{q_0}(-\lambda) - e^{i\lambda^2\tau}\hat{q}(-\lambda;\tau) - F(-\lambda;0)$$

and, by applying Fokas method, the Ehrenpreis Form [2],

$$\text{EF: } 2\pi q(x,t) = \int_{-\infty}^{\infty} e^{i\lambda x - i\lambda^2 t}\hat{q_0}(\lambda)\,\mathrm{d}x - \int_{\partial D^+} e^{i\lambda x - i\lambda^2 t}F(\lambda;0)\,\mathrm{d}\lambda$$

where

$$\begin{aligned}
&\partial D^+ = -i(-\infty,0]\cup[0,\infty),\\
&\hat{q}_0(\lambda) = \int_0^{\infty} e^{-i\lambda x}q_0(x)\,\mathrm{d}x, \quad \forall\lambda\in\mathrm{clos}(\mathbb{C}^-),\\
&F(\lambda;0) = \int_0^{\tau} e^{i\lambda^2 s}q_x(0,s)\,\mathrm{d}s + i\lambda\int_0^{\tau} e^{i\lambda^2 s}q(0,s)\,\mathrm{d}s \quad \forall\tau\in[t,T],\forall\lambda\in\mathbb{C}.
\end{aligned}$$

We aim to re-express $q(x,t)$ in terms of known quantities.

6.1 Obtain a formula for $F(\lambda;0)$

$$F(\lambda;0) = \int_0^{\tau} e^{i\lambda^2 s}q_x(0,s)\,\mathrm{d}s + i\lambda\int_0^{\tau} e^{i\lambda^2 s}q(0,s)\,\mathrm{d}s$$

We aim to re-express $F(\lambda;0)$ in terms of β, h, q_0, τ and $\hat{q}(\cdot;\tau)$ by forming substitutions for $\int_0^{\tau} e^{i\lambda^2 s}q_x(0,s)\,\mathrm{d}s$ and $i\lambda\int_0^{\tau} e^{i\lambda^2 s}q(0,s)\,\mathrm{d}s$ denoted Q_x and Q respectively. Start by performing a *time-transform* of BC (18):

$$\int_0^{\tau} e^{i\lambda^2 s}h(s)\,\mathrm{d}s = \int_0^{\tau} e^{i\lambda^2 s}q_x(0,s)\,\mathrm{d}s + \beta\int_0^{\tau} e^{i\lambda^2 s}q(0,s)\,\mathrm{d}s \tag{19}$$

Substitute $-\lambda$ into $F(\lambda;0)$:

$$\begin{aligned}
F(-\lambda;0) &= \int_0^{\tau} e^{i(-\lambda)^2 s}q_x(0,s)\,\mathrm{d}s + i(-\lambda)\int_0^{\tau} e^{i(-\lambda)^2 s}q(0,s)\,\mathrm{d}s\\
\implies F(-\lambda;0) &= \int_0^{\tau} e^{i\lambda^2 s}q_x(0,s)\,\mathrm{d}s - i\lambda\int_0^{\tau} e^{i\lambda^2 s}q(0,s)\,\mathrm{d}s
\end{aligned} \tag{20}$$

Form an equation for Q by taking (19) - (20):

$$\begin{aligned}
&\int_0^{\tau} e^{i\lambda^2 s}h(s)\,\mathrm{d}s - F(-\lambda;0) = (\beta+i\lambda)\int_0^{\tau} e^{i\lambda^2 s}q(0,s)\,\mathrm{d}s\\
\implies &\int_0^{\tau} e^{i\lambda^2 s}q(0,s) = \frac{1}{\beta+i\lambda}\left(\int_0^{\tau} e^{i\lambda^2 s}h(s)\,\mathrm{d}s - F(-\lambda;0)\right)\\
\implies &Q = \frac{1}{\beta+i\lambda}\left(\int_0^{\tau} e^{i\lambda^2 s}h(s)\,\mathrm{d}s - F(-\lambda;0)\right)
\end{aligned} \tag{21}$$

Form an equation for Q_x by substituting (21) into (20):

$$\begin{aligned}
&\int_0^{\tau} e^{i\lambda^2 s}q_x(0,s)\,\mathrm{d}s = \frac{i\lambda}{\beta+i\lambda}\left(\int_0^{\tau} e^{i\lambda^2 s}h(s)\,\mathrm{d}s - F(-\lambda;0)\right) + F(-\lambda;0)\\
\implies &\int_0^{\tau} e^{i\lambda^2 s}q_x(0,s)\,\mathrm{d}s = \frac{i\lambda}{\beta+i\lambda}\int_0^{\tau} e^{i\lambda^2 s}h(s)\,\mathrm{d}s + \frac{\beta}{\beta+i\lambda}F(-\lambda;0)\\
\implies &Q_x = \frac{i\lambda}{\beta+i\lambda}\int_0^{\tau} e^{i\lambda^2 s}h(s)\,\mathrm{d}s + \frac{\beta}{\beta+i\lambda}F(-\lambda;0)
\end{aligned} \tag{22}$$

Substitute (22) and (21) into $F(\lambda;0)$:

$$\begin{aligned}
F(\lambda;0) &= \frac{i\lambda}{\beta+i\lambda}\int_0^\tau e^{i\lambda^2 s}h(s)\,\mathrm{d}s + \frac{\beta}{\beta+i\lambda}F(-\lambda;0) \\
&\quad + \frac{i\lambda}{\beta+i\lambda}\left(\int_0^\tau e^{i\lambda^2 s}h(s)\,\mathrm{d}s - F(-\lambda;0)\right) \\
&= \frac{2i\lambda}{\beta+i\lambda}\int_0^\tau e^{i\lambda^2 s}h(s)\,\mathrm{d}s + \frac{\beta-i\lambda}{\beta+i\lambda}F(-\lambda;0) \\
F(\lambda;0) &= \frac{2i\lambda}{\beta+i\lambda}\int_0^\tau e^{i\lambda^2 s}h(s)\,\mathrm{d}s + \frac{\beta-i\lambda}{\beta+i\lambda}(\hat{q_0}(-\lambda) - e^{i\lambda^2\tau}\hat{q}(-\lambda;\tau)) \quad \text{(Using GR)}
\end{aligned}$$

In terms of β, h, q_0, τ and $\hat{q}(.;\tau)$,

$$F(\lambda;0) = \frac{2i\lambda}{\beta+i\lambda}\int_0^\tau e^{i\lambda^2 s}h(s)\,\mathrm{d}s + \frac{\beta-i\lambda}{\beta+i\lambda}\hat{q_0}(-\lambda) - \frac{\beta-i\lambda}{\beta+i\lambda}(e^{i\lambda^2\tau})\hat{q}(-\lambda;\tau)$$

6.2 Substitute $F(\lambda;0)$ into EF

$$\begin{aligned}
&\text{EF: } 2\pi q(x,t) = \int_{-\infty}^{\infty} e^{i\lambda x - i\lambda^2 t}\hat{q_0}(\lambda)\,\mathrm{d}\lambda \\
&-\int_{\partial D^+} e^{i\lambda x - i\lambda^2 t}\left(\frac{2i\lambda}{\beta+i\lambda}\int_0^\tau e^{i\lambda^2 s}h(s)\,\mathrm{d}s + \frac{\beta-i\lambda}{\beta+i\lambda}\hat{q_0}(-\lambda) - \frac{\beta-i\lambda}{\beta+i\lambda}(e^{i\lambda^2\tau})\hat{q}(-\lambda;\tau)\right)\mathrm{d}\lambda
\end{aligned}$$

Focusing on the second term of EF,

$$\begin{aligned}
&\int_{\partial D^+} e^{i\lambda x - i\lambda^2 t}\left(\frac{2i\lambda}{\beta+i\lambda}\int_0^\tau e^{i\lambda^2 s}h(s)\,\mathrm{d}s + \frac{\beta-i\lambda}{\beta+i\lambda}\hat{q_0}(-\lambda) - \frac{\beta-i\lambda}{\beta+i\lambda}(e^{i\lambda^2\tau})\hat{q}(-\lambda;\tau)\right)\mathrm{d}\lambda \\
&= \int_{\partial D^+}\frac{2i\lambda}{\beta+i\lambda}(e^{i\lambda x - i\lambda^2 t})\int_0^\tau e^{i\lambda^2 s}h(s)\,\mathrm{d}s\,\mathrm{d}\lambda \\
&\quad + \int_{\partial D^+}\frac{\beta-i\lambda}{\beta+i\lambda}(e^{i\lambda x - i\lambda^2 t})\hat{q_0}(-\lambda)\,\mathrm{d}\lambda \\
&\quad - \int_{\partial D^+}\frac{\beta-i\lambda}{\beta+i\lambda}(e^{i\lambda x - i\lambda^2 t})(e^{i\lambda^2\tau})\hat{q}(-\lambda;\tau)\,\mathrm{d}\lambda
\end{aligned}$$

Therefore, EF:

$$\begin{aligned}
2\pi q(x,t) &= \int_{-\infty}^{\infty} e^{i\lambda x - i\lambda^2 t}\hat{q_0}(\lambda)\,\mathrm{d}\lambda \\
&\quad + \int_{\partial D^+}\frac{2i\lambda}{\beta+i\lambda}(e^{i\lambda x - i\lambda^2 t})\int_0^\tau e^{i\lambda^2 s}h(s)\,\mathrm{d}s\,\mathrm{d}\lambda \\
&\quad + \int_{\partial D^+}\frac{\beta-i\lambda}{\beta+i\lambda}(e^{i\lambda x - i\lambda^2 t})\hat{q_0}(-\lambda)\,\mathrm{d}\lambda \\
&\quad - \int_{\partial D^+}\frac{\beta-i\lambda}{\beta+i\lambda}(e^{i\lambda x - i\lambda^2 t})(e^{i\lambda^2\tau})\hat{q}(-\lambda;\tau)\,\mathrm{d}\lambda \qquad (23)
\end{aligned}$$

With $q(x,t)$ in this form, we will be able to plot it for various x, t, and initial conditions q_0.

6.3 Argue that terms in EF which depend explicitly upon $\hat{q}(\cdot;\tau)$ can be replaced by other formulations.

In EF, the first three terms are known, they will be denoted K. Therefore, EF:

$$2\pi q(x,t) = K - \int_{\partial D^+} \frac{\beta - i\lambda}{\beta + i\lambda} e^{i\lambda x - i\lambda^2 t} e^{i\lambda^2 \tau} \hat{q}(-\lambda;\tau)\,\mathrm{d}\lambda$$

We aim to show that

$$I(\lambda;x,t,\tau) = \left(\int_{\partial D^+} \frac{\beta - i\lambda}{\beta + i\lambda} e^{i\lambda x} e^{i\lambda^2(\tau - t)} \hat{q}(-\lambda;\tau)\,\mathrm{d}\lambda \right) \to 0 \text{ as } \lambda \to \infty \text{ and}$$

$I(\lambda;x,t,\tau)$ is analytic (defined) on all of $\mathrm{clos}(D^+)$, given $\lambda \in \mathrm{clos}(\mathbb{C}^+)$.

EF is undefined when $\lambda = i\beta$. Therefore, to ensure $I(\lambda;x,t,\tau)$ is analytic on all of $\mathrm{clos}(D^+)$, β must not be in the closed fourth quadrant. There are, however, two exceptions to this restriction on β: $\beta = 0$ and $\beta \to \infty$ are both allowed.

$$\text{If } \beta = 0, \frac{\beta - i\lambda}{\beta + i\lambda} = \frac{-i\lambda}{i\lambda} = -1.$$
$$\text{If } \beta \to \infty, \frac{\beta - i\lambda}{\beta + i\lambda} \to \frac{\beta}{\beta} = 1.$$

To show that $I(\lambda;x,t,\tau)$ decays, we argue that since $\lambda \in \mathrm{clos}(\mathbb{C}^+)$ and $x \in [0,\infty)$, $\mathrm{Re}(i\lambda x) \leq 0$, $e^{i\lambda x} = \mathcal{O}(1)$. Furthermore, since $\lambda^2 \in \{z \in \mathbb{C} : Im(z) \geq 0\}$, $t \in [0,T]$ and $\tau \in [t,T]$, $(\tau - t) \geq 0$ which implies that $\mathrm{Re}(i\lambda^2(\tau - t)) \leq 0$ which in turn implies that $e^{i\lambda^2(\tau - t)} = \mathcal{O}(1)$. Next,

$$\frac{\beta - i\lambda}{\beta + i\lambda} = -1 + \frac{2\beta}{\beta + i\lambda} \implies \frac{\beta - i\lambda}{\beta + i\lambda} \to -1 \text{ as } \lambda \to \infty.$$

We state without proof that $\hat{q}(-\lambda;\tau)$ decays as λ becomes large. Therefore, $I(\lambda;x,t,\tau) \to 0$ as $\lambda \to \infty$, and if $\beta < 0$, then $I(\lambda;x,t,\tau)$ is analytic (defined) on all of $\mathrm{clos}(D^+)$. This statement can be proved using integration by parts and the generalised Riemann-Lebesgue lemma [1]. The implication is that we can replace $I(\lambda;x,t,\tau)$ with 0 when plotting $q(x,t)$.

6.4 Types of initial conditions: function q_0

We require the properties of every function q_0 to the following:

1.
$$\begin{cases} q_0(x) > 0 \text{ if } 1 < x < 3 \\ q_0(x) = 0 \text{ otherwise.} \end{cases}$$

2.
$$\int_1^3 q_0(x)\,\mathrm{d}x = 1.$$

The following are the four types of q_0 we will use:
Discontinuous:

$$q_0 = \begin{cases} 0.5 \text{ if } 1 < x < 3 \\ 0 \text{ otherwise.} \end{cases}$$

Continuous but non-differentiable:

$$q_0 = \begin{cases} -|x-2|+1 \text{ if } 1 < x < 3 \\ 0 \text{ otherwise.} \end{cases}$$

Continuous and n-differentiable:

$$q_0 = \begin{cases} a[(x-1)(x-3)]^{n+1} \text{ if } 1 < x < 3 \\ 0 \text{ otherwise.} \end{cases}$$

where a is the normalising constant such that $a\int_1^3 q_0\,\mathrm{d}x = 1$.
Continuous and infinitely-differentiable (mollifier function):

$$q_0 = \begin{cases} be^{-\frac{1}{1-(x-2)^2}} \text{ if } 1 < x < 3 \\ 0 \text{ otherwise.} \end{cases}$$

where b is the normalising constant such that $b\int_1^3 q_0\,\mathrm{d}x = 1$.

Constants a and b will be evaluated using numerical integration.

6.5 Determining the integration interval size for $q(x,t)$

One final obstacle we face in plotting the graph of the solution to the time-dependent linear Schrodinger equation is that ApproxFun cannot perform numerical integration for an infinitely-sized integration interval. Instead, we will have to use a sufficiently large integration interval.

To decide on an integration interval which is sufficiently large, we calibrated $q(x,0)$ based on its initial conditions. Our initial condition states that $q(x,0) = q_0(x)$. This implies that

$$\int_1^3 q(x,0)\,\mathrm{d}x = \int_1^3 q_0(x)\,\mathrm{d}x = 1.$$

If we set the integration intervals within $q(x,t)$ to $[-m,m]$ and $-i[-m,0]\cup[0,m]$, we can code a loop to increase the value of m until

$$\int_1^3 \tilde{q}(x,0)\,\mathrm{d}x \approx 1$$

where

$$\begin{aligned}2\pi\tilde{q}(x,t) = &\int_{-m}^{m} e^{i\lambda x - i\lambda^2 t}\hat{q_0}(\lambda)\,\mathrm{d}\lambda \\ &+ \int_{\partial D_m^+} \frac{2i\lambda}{\beta + i\lambda}(e^{i\lambda x - i\lambda^2 t})\int_0^\tau e^{i\lambda^2 s}h(s)\,\mathrm{d}s\,\mathrm{d}\lambda \\ &+ \int_{\partial D_m^+} \frac{\beta - i\lambda}{\beta + i\lambda}(e^{i\lambda x - i\lambda^2 t})\hat{q_0}(-\lambda)\,\mathrm{d}\lambda \\ &- \int_{\partial D_m^+} \frac{\beta - i\lambda}{\beta + i\lambda}(e^{i\lambda x - i\lambda^2 t})(e^{i\lambda^2\tau})\hat{q}(-\lambda;\tau)\,\mathrm{d}\lambda.\end{aligned}$$

The following are the values of m we attained for the various types of q_0 for an arbitrary tolerance of 10^{-5}:

Discontinuous: No value was attained as the limit of ApproxFun's adaptive algorithm was reached.

Continuous but non-differentiable: 67

Continuous and 1-differentiable: 69

Continuous and 4-differentiable: 21

Continuous and 7-differentiable: 18

Continuous and infinitely-differentiable (mollifier function): 52

We expected $q(x,t)$ to be easier to approximate, in the sense that the m value required for good accuracy is lower, as q_0 became smoother. Therefore, we expected the value of m to decrease as the function is differentiable more times. Our predictions were mostly accurate. The one major anomaly is the mollifier function which severely underperformed, for reasons yet unknown.

Nonetheless, with the values of m mostly acquired, we are ready to plot $q(x,t)$. The syntax for $q(x,t)$ is *q(x::Number, t::Number, m::Number, q0k::Int64, a::Number, n::Int64)* where both a and n are optional arguments. The purpose of arguments x, t and m should be evident. Argument $q0k$ indicates which initial condition function, q_0, to use:

$$\begin{aligned}q0k = 1 &\implies \text{use the discontinuous function for } q_0 \\ q0k = 2 &\implies \text{use the continuous but non-differentiable function for } q_0 \\ q0k = 3 &\implies \text{use the continuous and n-differentiable for } q_0 \\ q0k = 4 &\implies \text{use the mollifier function for } q_0.\end{aligned}$$

Argument n is relevant only if $q0k = 3$; it specifies the number of times q_0 can be differentiated. If $q0k = 3$ but n is unspecified, $n = 1$ by default. Argument a is the normalising constant which ensures $a\int_1^3 q_0\,\mathrm{d}x = 1$. If a is unspecified, the algorithm will automatically calculate the value of a based on the function q_0 specified, which will increase computation time.

6.6 Graphs of the solution to the time-dependent linear Schrodinger equation

The solution to the time-dependent linear Schrodinger equation is the wave function, $q(x,t)$. $|q(x,t)|^2$ is a probability distribution which models the probability of finding the particle at position x, at time t. We will plot $|q(x,t)|^2$ with $\beta = 0$ and $h(s) = 0$ (see 23), which implies that the total energy in the quantum system is conserved. The basic properties we expect from our plot is for the initial peak to flatten and spread out. We also expect to see a waveform with frequency which decreases over time, since there is finite energy in the system (especially since total energy is conserved) and that energy is being spread over an increasing range of values of x. Figures 23, 24 and 25 show the graphs of $|q(x,t)|^2$ for q_0 is the mollifer function, 7-differentiable, and non-differentiable respectively. All three graphs have $x \in [0, 10]$ for various t values.

The graphs demonstrate the properties we expected of them. Therefore, we believe our algorithm for $q(x,t)$, together with the algorithm "completecheb" which it relies on, are successful.

7 ApproxFun's adaptive algorithm

ApproxFun's adaptive algorithm runs a four-step process:

1. A *Fun* is generated which represents the Chebyshev approximation polynomial.
2. The accuracy of the *Fun* is estimated, if its accuracy is less than a threshold, redo step 1 but with a higher order approximating polynomial. Otherwise, proceed to step 3.
3. Shave off some coefficients from the *Fun*.
4. Output the *Fun*.

The adaptive algorithm increases the number of coefficients (order plus one) in the approximating polynomial by a scale factor of two each time, starting from 16. If the *Fun* with 16 coefficients is insufficiently accurate, it is discarded and a *Fun* with 32 coefficients is generated next, if necessary, a *Fun* with 64 coefficients is generated after that and so on. If a *Fun* with 2^{20} coefficients is still insufficiently accurate, ApproxFun will simply output a *Fun* with 2^{21} coefficients. Therefore, if ApproxFun displays the warning "Maximum number of coefficients 1048577 reached in constructing Fun", one should check the accuracy of the output *Fun*, for ApproxFun did not.

We currently do not understand the algorithms by which the accuracy of the *Fun* is estimated, and the number of coefficients to shave off is determined. Nonetheless, it is clear that steps 1 and 2 comprise the majority of the adaptive algorithm's runtime as many *Fun*s are generated, tested, and for all but one *Fun*, discarded.

It is possible to optimise the loop between these two steps by re-using function evaluations from a discarded *Fun* for the next *Fun*. This is what we aim to do.

7.1 Re-use of function values

We restate the formula (3) to generate the Chebyshev nodes, $x_1, x_2, \ldots, x_N$:

$$x_k = \cos\left(\frac{2k-1}{2N}\pi\right) \text{ for } k = 1, 2, \ldots N.$$

Notice that $\forall k \in \mathbb{N}, k \leq N$, $2k-1$ is odd. Consider $\frac{b(2k-1)}{b(2N)}$ for some $b \in \mathbb{N}$. If we let $Nb = M$, then $\frac{b(2k-1)}{b(2N)} = \frac{b(2k-1)}{2M}$. Realise that if b is odd, then $b(2k-1)$ is also odd, and

$$x_k = \cos\left(\frac{2j-1}{2M}\pi\right) \text{ for } j = 1, 2, \ldots M,$$

such that $\forall k, \exists j : 2j - 1 = b(2k-1)$. The implication is that for any $b \in \mathbb{N} : b$ is odd, a distribution of N Chebyshev nodes and a distribution of Nb Chebyshev nodes will have all N Chebyshev nodes coincide with a node in the distributon of Nb Chebyshev nodes. Since we need to evaluate the function at all N Chebyshev nodes to create a Chebyshev polynomial approximation with N coefficients (see 3.2), if we then wish to create a Chebyshev polynomial approximation with Nb coefficients, we can re-use all N function evaluations which would save us up to $\frac{100}{b}$ percent of computation time.

In fact, there exists a second variant of Chebyshev nodes called "Chebyshev extreme points" [6], as opposed to the standard "Chebyshev zero points" which ApproxFun uses. The formula to generate Chebyshev extreme points, $x_1, x_2, \ldots x_N$, is:

$$x_k = \cos\left(\frac{k}{N}\pi\right) \text{ for } k = 0, 1, \ldots N.$$

The significance of Chebyshev extreme points is that, with them, we can re-use all N function evaluations to create a Chebyshev polynomial approximation with Nc coefficients where $c \in \mathbb{N} : c$ is even. Therefore, we can re-use even more function evaluations and save more computation time.

We implemented our "re-use of function values" idea in ApproxFun's "constructor.jl" file which contains the code for its adaptive algorithm. One significant change we made was to remove the scale factor of two between steps 1 and 2, for that prevented function evaluations from being re-used; we left ApproxFun to continue using Chebyshev zero points. Instead, we made the adaptive algorithm loop through the following sequence of integers for number of coefficients in the *Fun*:

$$(16, 32, 80, 128, 240, 512, 1200, 2048, 3600, 8192, 18000,$$
$$32768, 54000, 131072, 270000, 524288, 810000).$$

If a *Fun* with 810000 coefficients is still insufficiently accurate, the edited ApproxFun will output a *Fun* with 2430000 coefficients. Other details of our improvements are omitted from this report.

Tests of the edited ApproxFun indicate the "re-use of function values" code improvements do speed up ApproxFun's adaptive algorithm. Figure 26 shows the results of our tests. Note that "Median Pure Time using logn", highlighted in green, are the results for our tests on step 1 of the adaptive algorithm alone; figure 27 shows the code we ran to generate these results. "Median ApproxFun Time" are the results for our tests on ApproxFun as a whole. The two sets of results should be similar, but they are not. In fact, there are numerous anomalies in the results which we cannot yet explain (details omitted from this report). Significantly more testing and research is necessary, especially since there are important lines of code in ApproxFun's adaptive algorithm which we have yet to understand. Nonetheless, we consider our "re-use of function values" code improvements to ApproxFun to be promising.

Acknowledgments

I will like to thank my project supervisor, Professor David Smith, for his invaluable help and support.

References

[1] M. J. Ablowitz and A. S. Fokas, *Complex variables*, Cambridge Texts in Applied Mathematics, Cambridge University Press, 1997.

[2] A. S. Fokas, *A unified approach to boundary value problems*, CBMS-SIAM, 2008.

[3] Wouter Den Haan, http://econ.lse.ac.uk/staff/wdenhaan/numerical/, filename: functionapproximation.pdf, last modified: 2011-08-29 16:02.

[4] J.H. Mathews and K.D. Fink, *Numerical methods using matlab*, Featured Titles for Numerical Analysis Series.

[5] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical recipes in c (2nd ed.): The art of scientific computing*, pp. 190–195, Cambridge University Press, New York, NY, USA, 1992.

[6] L.N. Trefethen, *Finite difference and spectral methods for ordinary and partial differential equations*, pp. 260–268, The author, 1996.

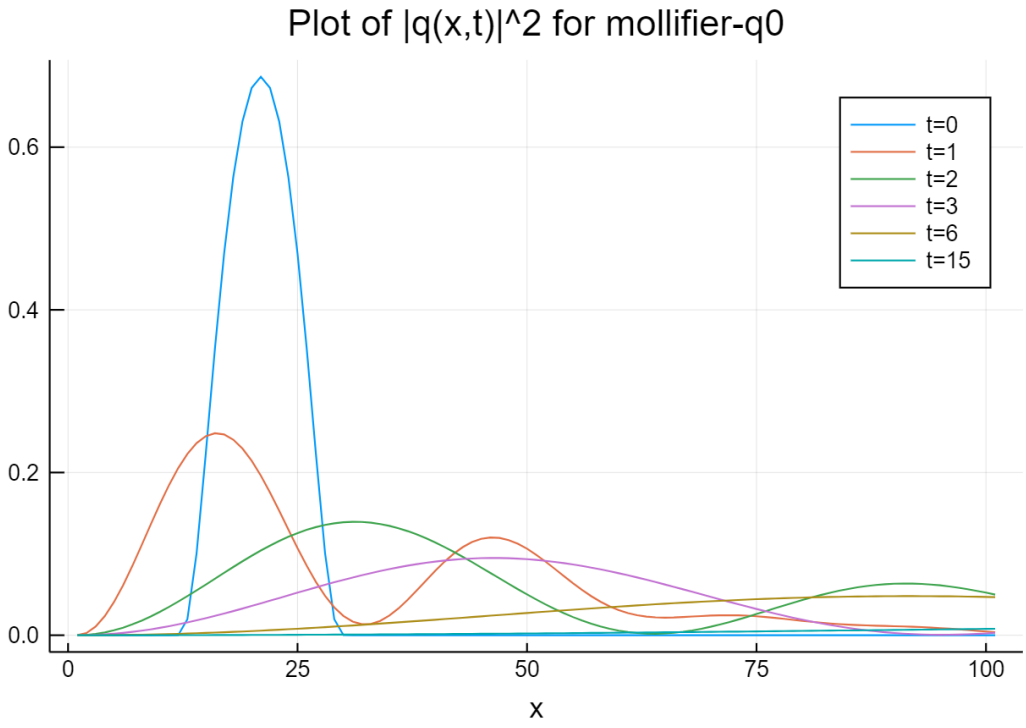


Figure 23: Plot of $|q(x,t)|^2$ for q_0 is the mollifer function, for $x \in [0, 10]$, for various t values.

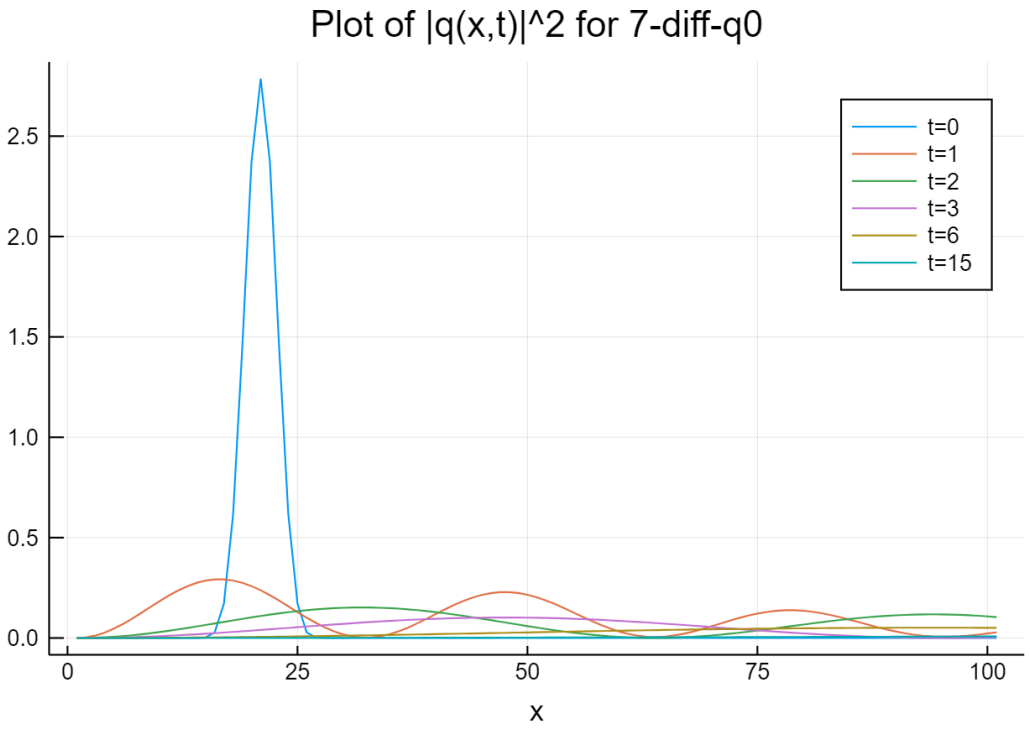


Figure 24: Plot of $|q(x,t)|^2$ for q_0 is 7-differentiable, for $x \in [0, 10]$, for various t values.

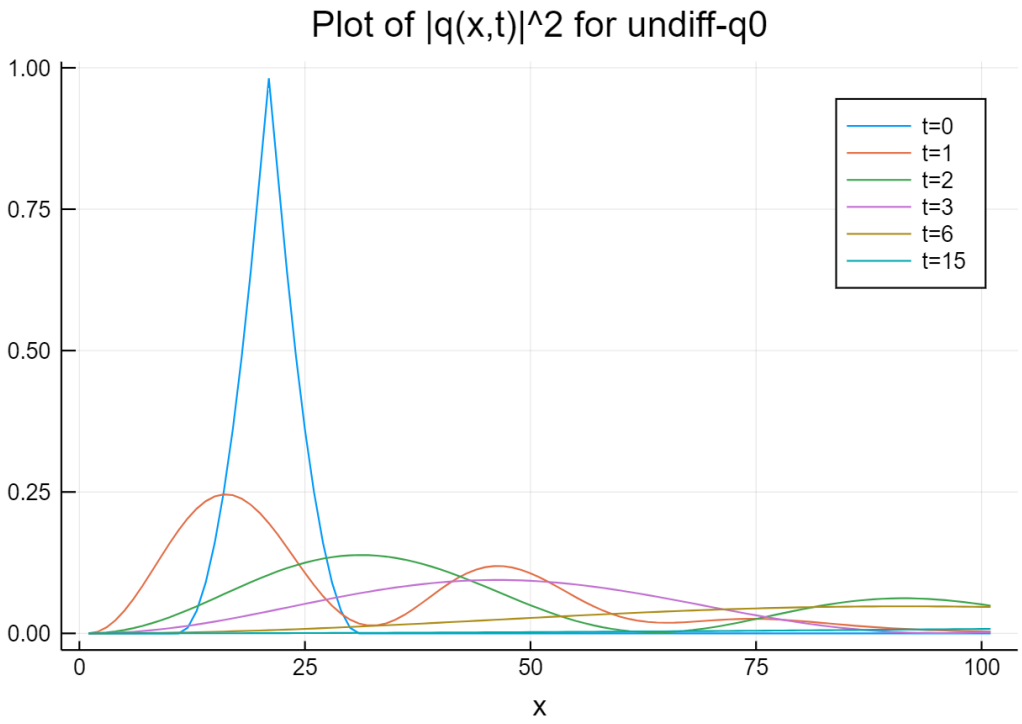


Figure 25: Plot of $|q(x,t)|^2$ for q_0 is non-differentiable, for $x \in [0, 10]$, for various t values.

FL=l->completecheb(x->sin(l*x/(((x-0.5)^2)+0.1^4) + sin(x/(((x-0.5)^2)+0.1^4))/10),[0,1],[],ORDER)
f=l->(sin((10^4)*FL(l)))^4 0..1

Method	Median Pure Time using logn	Median ApproxFun Time	Memory usage	
Reuse points	162s	164s*	21.88GiB / 22.37GiB	ApproxFun value: 0.3634998271386595
Do not reuse points	202s	423s*	27.13GiB / 55.68GiB	ApproxFun value: 0.3634998271386725
		* Final order used is identical: 117		

x->sin((10^5)x) 0..1

Median time taken for f(0.5): (454*10^-4)ms

Method	Median Pure Time using logn	Median ApproxFun Time	Memory usage	
Reuse points	2.139s	2.583s	295.13MiB / 291.82MiB	ApproxFun value: 1.9993608073639 31*10^-5
Do not reuse points	3.176s	2.766s**	850.32MiB / 320.11MiB	ApproxFun value: 1.999360807546028*10^-5
		** 4194288 total function evaluations instead of 4286176		
		These use logn = 2^x for x in 4:20 instead of the edited logn.		

Figure 26: Results of the comparison between the edited ApproxFun (which re-uses function evaluations) against the original ApproxFun (which does not re-use function evaluations).

```
#Option 1 for f; logn = [16, 32, 80, 128, 240, 512, 1200, 2048, 3600, 8192, 18000, 32768, 54000, 131072, 270000, 524288, 810000, 810000*3]
f=x->sin((10^5)x)
#Option 2 for f; logn = [16, 32, 80, 128, 240]
ORDER=2^20 #Gotten from adapt_order based on "(Original) constructor.jl"
FL=l->completecheb(x->sin(x/(((x-0.5)^2)+0.1^4) + l*sin(x/(((x-0.5)^2)+0.1^4))/10),[0,1],[],ORDER)
f=l->(sin((10^4)*FL(1)))^4
d=Chebyshev(0..1)
### Define f via one of the two options, change logn if need be, then run the following code.
logn = [16, 32, 80, 128, 240, 512, 1200, 2048, 3600, 8192, 18000, 32768, 54000, 131072, 270000, 524288, 810000, 810000*3]
Time_Reuse = @benchmark  for Placeholder in 1:1
    REUSE_PTS=[]
for i in logn[1:end]
pts=points(d,i)
RP = length(REUSE_PTS)
    if RP == 0
        REUSE_PTS = v = f.(pts)
    else
        Lpts = length(pts)
        Factor = Lpts/RP
        if Factor%1 == 0 && Factor%2 == 1 #"Factor%2 == 1" is faster than "isodd". Becareful of factor = 1, though that should not happen.
            REUSE = convert(Int64,Factor/2+0.5):convert(Int64,Factor):convert(Int64,(RP*2-1)*Factor/2+0.5)
            pts[REUSE].=NaN
            Slct = x->selective(f,x)
            v = Slct.(pts)
            setindex!(v,REUSE_PTS,REUSE)
            REUSE_PTS = v
        else
        v = f.(pts)
        end
    end
    Fun(d,transform(d,v)), REUSE_PTS
end
end
T = real(eltype(domain(d)))
Time_NoReuse = @benchmark for i in logn[1:end]
pts=points(d,i)
Fun(d,transform(d,T[f(x...) for x in pts]))
end
```

Figure 27: Code by which the results for “Median Pure Time using logn” (shown in figure 26) were generated.