CHEBFUN AND NUMERICAL QUADRATURE

NICHOLAS HALE* AND LLOYD N. TREFETHEN†

Abstract. Chebfun is a Matlab-based software system that overloads Matlab's discrete operations for vectors and matrices to analogous continuous operations for functions and operators. We begin by describing Chebfun's fast capabilities for Clenshaw–Curtis and also Gauss–Legendre, –Jacobi, –Hermite, and –Laguerre quadrature, based on algorithms of Waldvogel and Glaser, Liu, and Rokhlin. Then we consider how such methods can be applied to quadrature problems including 2D integrals over rectangles, fractional derivatives and integrals, functions defined on unbounded intervals, and the fast computation of weights for barycentric interpolation.

Key words. Chebfun, Clenshaw–Curtis quadrature, Gauss quadrature, barycentric interpolation formula, Riemann–Liouville integral, fractional calculus

AMS subject classifications. 65D32, 41A55

1. Introduction. One of the fundamental problems in numerical mathematics is *quadrature*, the approximate evaluation of an integral such as

$$I = \int_{-1}^{1} w(x) f(x)\, dx, \tag{1.1}$$

where f is a continuous function on $[-1, 1]$ and w is a *weight function* which we take to be positive and continuous on $(-1, 1)$, though perhaps approaching 0 or ∞ as $x \to \pm 1$. The starting point of almost every quadrature algorithm is the notion of an $(n+1)$-point quadrature formula,

$$I_n = \sum_{k=0}^{n} w_k f(s_j), \tag{1.2}$$

where $s_0, \ldots, s_n$ are a set of *nodes* in $[-1, 1]$ and $w_0, \ldots, w_n$ are a set of *weights*. The reason for using the parameter n for a quadrature formula of $n+1$ points is that usually, the weights $\{w_k\}$ are chosen according to the principle that the approximation should be exact if f is a polynomial of degree at most n, i.e., $I_n = I$ for $f \in \mathcal{P}_n$.

Chebfun [22], which is an open-source software system built on Matlab, contains implementations of what we believe are the best available algorithms for computing many families of quadrature nodes and weights, particularly the algorithms of Waldvogel [23] and Glaser, Liu, and Rokhlin [7]. The purpose of this paper is twofold: first, to call attention to these remarkable algorithms and their Chebfun implementations, and second, to show how Gauss–Jacobi formulas in particular lead to very flexible computation of integrals in the Chebfun environment, including an application to the computation of fractional derivatives and integrals. Finally, following an observation of Wang and Xiang at Central South University in China [24], we show how Chebfun takes advantage of these methods to enable barycentric interpolation in Legendre and related points even on grids of sizes in the millions.

*Oxford University Mathematical Institute, 24–29 St Giles, Oxford OX1 3LB, UK, `hale@maths.ox.ac.uk`. Supported by MathWorks, Inc. and by Award No. KUK-C1-013-04, made by the King Abdullah University of Science and Technology (KAUST).

†Oxford University Mathematical Institute, 24–29 St Giles, Oxford OX1 3LB, UK, `trefethen@maths.ox.ac.uk`. Supported by the European Research Council.

We set the stage with a quick illustration. The following Chebfun computation computes three Gauss–Legendre quadrature nodes and the corresponding weights. The name `legpts` comes from "Legendre points", another term for Gauss–Legendre quadrature nodes, since these nodes are the roots of the Legendre polynomial P_{n+1}.

```
>> [s,w] = legpts(3)
s =
  -0.774596669241483
                   0
   0.774596669241483
w =
   0.555555555555556   0.888888888888889   0.555555555555556
```

If we change 3 to 10000, the same command computes nodes and weights for the 10000-point Gauss–Legendre quadrature rule. Here is the time required for this computation on a 2010 desktop machine:

```
>> tic, [s,w] = legpts(10000); toc
Elapsed time is 0.268227 seconds.
```

2. Gauss and Clenshaw–Curtis quadrature. As is well known and described in many books, the nodes and weights for Gauss quadrature are determined by the condition that the formula should have maximal polynomial order, namely $I = I_n$ whenever f is a polynomial of degree at most $2n + 1$. If the weight function is a constant, $w(x) = 1$, this is the case of Gauss–Legendre quadrature. An alternative for $w(x) = 1$ is Clenshaw–Curtis quadrature, in which the nodes are the Chebyshev points $s_j = \cos(j\pi/n)$ [3]. Here the polynomial order is only n, but as explained in [13] and [19], this large difference in polynomial order often makes little difference in practice.

Chebfun, which represents functions to machine precision by polynomial or piecewise polynomial interpolation in Chebyshev points, uses Clenshaw–Curtis quadrature as its basic integration tool. The $(n + 1)$-point Clenshaw–Curtis quadrature approximant I_n is readily computed in $O(n \log n)$ operations by converting the data to a Chebyshev series and applying the Fast Fourier Transform (FFT) [6], and the constant implicit in the "O" is very small. Alternatively, Chebfun enables one to compute explicit Clenshaw–Curtis nodes and weights with the `chebpts` command, in analogy to the example shown above:

```
>> [s,w] = chebpts(3)
s =
  -1
   0
   1
w =
   0.333333333333333   1.333333333333333   0.333333333333333

>> tic, [s,w] = chebpts(10000); toc
Elapsed time is 0.003027 seconds.
```

This fast computation makes use of a Chebfun implementation of an $O(n \log n)$ algorithm published by Waldvogel [23], which determines the weights explicitly, again by use of the FFT.

Note that although the Chebfun computation of Gauss–Legendre quadrature

nodes and weights is very fast, for Clenshaw–Curtis it is even faster. For smooth functions at least, Clenshaw–Curtis quadrature is a powerful tool for any application.

Sometimes, however, one wants to work with Gauss formulas, with their optimal order of polynomial accuracy. Here there are several familiar choices for the weight function in (1.1):

Gauss–Legendre: $w(x) = 1$,

Gauss–Chebyshev: $w(x) = (1 - x^2)^{-1/2}$,

Gauss–Jacobi: $w(x) = (1 - x)^{\alpha}(1 + x)^{\beta}$, $\alpha, \beta > -1$.

Both Gauss–Legendre and Gauss–Chebyshev are special cases of Gauss–Jacobi. In Chebfun, Gauss–Jacobi nodes and weights are available through the command `jacpts`. Chebyshev, Legendre, and Jacobi polynomials are also available through `chebpoly`, `legpoly`, and `jacpoly`. There are analogous commands for Gauss–Hermite quadrature on $(-\infty, \infty)$ and Gauss–Laguerre quadrature on $[0, \infty)$, as summarized in Table 2.1.

TABLE 2.1
Orthogonal polynomial and Gauss quadrature capabilities in Chebfun. The domains listed are defaults, which are automatically scaled to other intervals such as $[a, b]$ as appropriate.

Name	Domain	Weight function $w(x)$	Orthogonal polynomials	Nodes and weights
Legendre	$[-1, 1]$	1	`legpoly`	`legpts`
Chebyshev	$[-1, 1]$	$(1 - x^2)^{-1/2}$	`chebpoly`	`chebpts`
Jacobi	$[-1, 1]$	$(1 - x)^{\alpha}(1 + x)^{\beta}$	`jacpoly`	`jacpts`
Hermite	$(-\infty, \infty)$	$\exp(-x^2/2)$	`hermpoly`	`hermpts`
Laguerre	$[0, \infty)$	$\exp(-x)$	`lagpoly`	`lagpts`

3. Golub–Welsch and Glasier–Liu–Rokhlin algorithms. A famous algorithm for computing Gauss quadrature nodes and weights was introduced by Golub and Welsch (GW) in 1969 [8]. This algorithm reduces the problem to a real symmetric tridiagonal eigenvalue problem, which can be solved in principle in $O(n^2)$ time. Thanks to the powerful and numerically stable algorithms that have been developed for calculating matrix eigenvalues, this leads to an accurate and effective way of computing quadrature nodes and weights that has been the standard for two generations. An unfortunate feature is that Matlab's black-box eigenvalue solver does not take advantage of the tridiagonal structure, so the operation count worsens from $O(n^2)$ to $O(n^3)$.

Whether $O(n^2)$ or $O(n^3)$, the operation count of the GW algorithm is too high for this method to be effective when n is in the thousands or higher. For many applications this hardly matters, since often one does not need a high-order Gauss formula, but it imposes an unfortunate limit on our numerical explorations, especially when one considers that Clenshaw–Curtis formulas are applicable in $O(n \log n)$ time. Consequently it was a striking advance when Glaser, Liu, and Rokhlin (GLR) introduced an algorithm that computes Gauss quadrature nodes and weights in $O(n)$ operations [7]. The GLR algorithm calculates the nodes and weights one at a time, hopping from each node to the next by an ingenious method involving a 30-term Taylor series and a few steps of Newton iteration. The work is just $O(1)$ operations

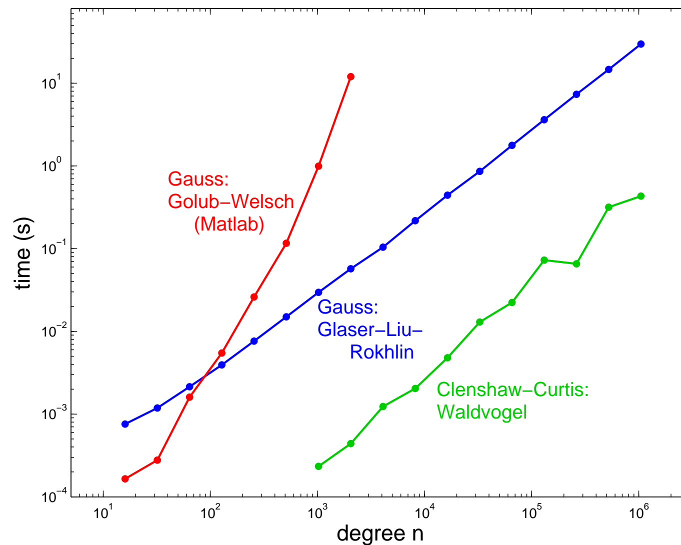

FIG. 3.1. *Comparison of Chebfun timings for computation of Gauss–Legendre nodes and weights by the Glaser–Liu–Rokhlin and Golub–Welsch algorithms. The third curve shows that computing Clenshaw–Curtis nodes and weights, by the algorithm of Waldvogel, is even faster.*

per node, and that is why the overall operation count is $O(n)$. This speed is rather startling when one considers that merely evaluating a Legendre polynomial at a point requires $O(n)$ operations.

Chebfun contains implementations of the GLR algorithm for all the classes of polynomials listed in Table 2.1, whose uniformly high efficiency is summarized by the following experiment, which arbitrarily takes parameters $\alpha = 2$ and $\beta = 3$ for Gauss–Jacobi.

```
>> tic
>> [s,w] = legpts(1000);
>> [s,w] = chebpts(1000);
>> [s,w] = jacpts(1000,2,3);
>> [s,w] = hermpts(1000);
>> [s,w] = lagpts(1000);
>> toc
Elapsed time is 0.598015 seconds.
```

For the basic case of Gauss–Legendre quadrature, Figure 3.1 shows Chebfun timings for the GLR and Matlab GW algorithms as a function of n. (Chebfun's default for larger values of n is to use the GLR algorithm, but GW is available by setting a flag.) It is clear that GW becomes impractical once n is in the thousands, whereas GLR can be used even for n in the millions. The plot also shows data for Clenshaw–Curtis nodes and weights, a reminder that this problem remains simpler than Gauss quadrature.

4. Chebfun quadrature for smooth functions. The aim of Chebfun is to compute with functions of a real variable in a manner that has "the feel of symbolics but the speed of numerics." For smooth functions, this is achieved by representing

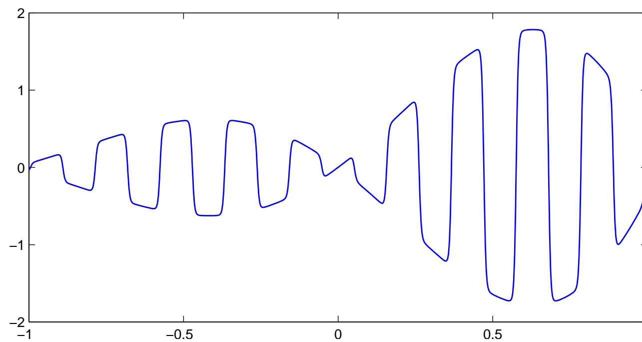

FIG. 4.1. *An example of a smooth chebfun represented by a single global polynomial interpolant through Chebyshev points. The function is* $f(x) = e^x \sin(3x)\tanh(5\cos(30x))$*, and the polynomial is of degree* 3209*.*

functions by polynomial and piecewise polynomial interpolants, and by overloading familiar Matlab commands for discrete vectors to their natural analogues for functions. For example, consider the function

$$f(x) = e^x \sin(3x)\tanh(5\cos(30x))$$

defined on the interval $[-1, 1]$. The following commands construct a chebfun of f and produce the plot shown in Figure 4.1:

```
>> x = chebfun('x');
>> f = exp(x).*sin(3*x).*tanh(5*cos(30*x));
>> plot(f)
```

The Chebfun representation consists of a global polynomial of degree 3209, an interpolant through 3210 Chebyshev points, and this degree has been determined adaptively to achieve approximately machine precision.

```
>> length(f)
ans =   3210
```

For details of the underlying approximation theory, see [21]. Once the chebfun has been constructed, all kinds of operations can be performed, each relying on a Chebfun implementation of an appropriate numerical algorithm, typically at high speed with accuracy close to machine precision. For example, the maximum of f is computed by differentiating f, finding the roots of f', and evaluating f at those roots:

```
>> max(f)
ans = 1.782604429158422
```

Computing the 2-norm of f, defined as the square root of the integral of $|f(x)|^2$, entails the computation of an integral by Clenshaw–Curtis quadrature:

```
>> norm(f)
ans = 1.250542878304186
```

Chebfun integrals arise in other contexts too. For example, the `sum` command determines the integral over the domain of definition:

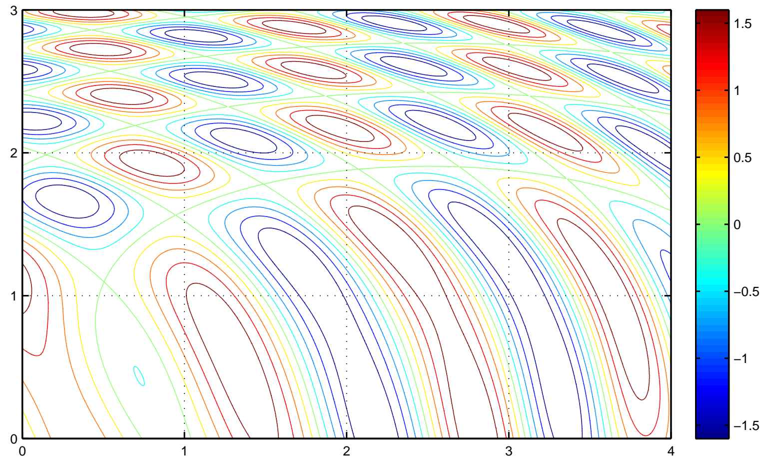

Fig. 5.1. *Chebfun is at heart a one-dimensional tool, but it can be applied to quadrature over a rectangle by taking a product. The example integrand plotted here is* $f(x, y) = \sin(5x + 2y) + \sin(x^2 + y^3)$.

```
>> sum(f)
ans = -0.017790593076879
```

The transpose symbol `'` is used in the usual Matlab fashion to signal the computation of an inner product, the continuous analogue of a vector inner product, again realized via an integral evaluated by Clenshaw–Curtis quadrature:

```
>> f'*exp(f)
ans = 2.149796850732142
```

Calculations like these happen in milliseconds, and the results are typically accurate in all but perhaps the final digit. Comparisons between Chebfun and specialized adaptive quadrature routines have indicated that Chebfun is generally as reliable and accurate as specialized software and runs at a comparable speed, being slower by a factor in the range 1–10 [1, 10]. In [1] it was noted that a particularly impressive competitor in such comparisons was the code `coteda` by Espelid [4], a precursor of Espelid's higher-order code `da2glob` [5]. A difference between Chebfun and the usual approach to adaptive quadrature is that Chebfun attempts adaptively to resolve the function itself before integrating it, whereas most quadrature algorithms adapt on the integral rather than the function. This difference is the basic reason why Chebfun tends to lie at the high-reliability, low-speed end of the range. Gonnet, however, has recommended adaptive quadrature based on resolving the function rather than the interval, and his algorithms presented in [9] share Chebfun's characteristics of high reliability at some cost in speed. Algorithm 3 of [9] has subsequently been developed into the code `quadcc` in Octave as of version 3.4 [12].

5. Chebfun quadrature over a rectangle. We like to think that Chebfun can do "almost anything in one dimension." It is a longstanding ambition of the Chebfun team to move to two or three dimensions, but so far, this project lies mainly in the future.

For the specific problem of quadrature over a rectangle, however, one can use a

product of two copies of any 1D quadrature method to get results. In particular, Chebfun can be used in this way, and the result are often quite reasonable, especially for smooth functions.

For example, Figure 5.1 shows a contour plot of the function

$$f(x, y) = \sin(5x + 2y) + \sin(x^2 + y^3)$$

over the rectangle $0 \le x \le 4$, $0 \le y \le 3$. The integral of f over this region can be computed by Chebfun with the following commands:

```
>> tic, f = @(x,y) sin(5*x+2*y)+sin(x.^2+y.^3);
>> Iy = @(y) sum(chebfun(@(x) f(x,y),x([1 end])));
>> I = sum(chebfun(@(y) Iy(y),y([1 end]),'vectorize')), toc
I = 0.862836879410888
Elapsed time is 0.814382 seconds.
```

This result is probably accurate in all but perhaps the final digit, since it agrees with the following computation by Matlab's `dblquad`:

```
>> tic, I = dblquad(f,0,4,0,3,1e-11,@quadl), toc
I = 0.862836879410889
Elapsed time is 24.46229 seconds.
```

In this example DBLQUAD appears much slower than tensor product Chebfun, but we make no claims about Chebfun performance for 2D integrals in general. The conclusion reached in [1], based on a collection of computations of this kind, was that Chebfun is typically about 15 times slower for integrals over rectangles than a tensor product of integrations by the routine `coteda`.

6. Chebfun quadrature for functions with endpoint singularities. Chebfun's representation of functions is actually more general than has been indicated so far in this paper. Chebfun can also work with functions with algebraic endpoint or interior algebraic singularities, which it treats by representing a function by a concatenation of pieces of the form $(x - a)^\alpha (b - x)^\beta p(x)$, where p is a polynomial and α and β may be be negative or positive, fractional or integer. These methods originate with the contribution of Richardson [15], and the exponents α and β can be specified by the user or determined automatically. Once found, exponents are adjusted in the appropriate way in further computations. For example, if the chebfun representing a function f has a singularity with an exponent α at some point, then the command `sqrt(f)` produces a chebfun with an exponent $\alpha/2$ at the same point.

This is where more specialized quadrature formulas come into the calculation. To integrate one of the functions just described, which may have point singularities involving arbitrary exponents, the mathematically ideal tool is Gauss–Jacobi quadrature, with parameters α and β chosen in accordance with the singularities at one or both ends of each subinterval. This is exactly the tool used by Chebfun, with nodes and weights computed on the fly by the GLR algorithm. The result is great accuracy and flexibility in computing integrals. For example, here we make a chebfun of the gamma function $\Gamma(x)$ on $[-4.5, 4.5]$, plotted in Figure 6.1a. Chebfun automatically determines that there are simple poles at $-4, -3, -2, -1, 0$ and splits the domain into six pieces.

```
>> g = chebfun(@gamma,[-4.5,4.5],'blowup','on','splitting','on');
```

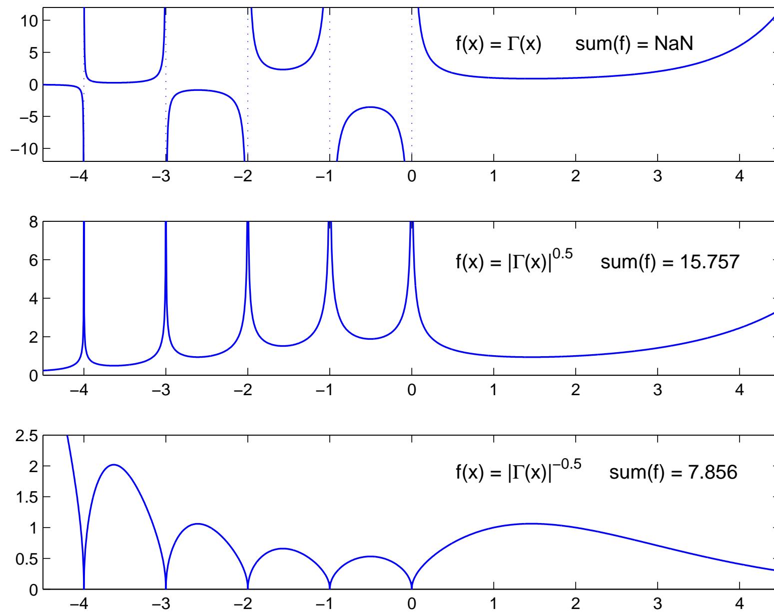

FIG. 6.1. *Chebfun represents functions with point singularities in a piecewise fashion, where each piece consists of a product of a Chebyshev interpolant and singular terms at one or both ends. Integrals of such functions are then computed by Gauss–Jacobi quadrature with appropriate exponents.*

If Chebfun is asked to compute the integral, it adds up the integrals from each of the six pieces. Three of the contributions are $+\infty$ and three are $-\infty$, so the result is Not-a-Number:

```
>> sum(g)
ans = NaN
```

If $\Gamma(x)$ is replaced by $|\Gamma(x)|$, all six pieces agree in sign, and we get infinity:

```
>> absg = abs(g);
>> sum(absg)
ans = Inf
```

True numerical results appear as soon as one weakens the singularities, so that the integrand becomes integrable:

```
>> sum(absg.^-.5)
ans = 7.855535000849889

>> sum(absg.^.5)
ans = 15.756773863531844

>> sum(absg.^.99)
ans = 5.511556606477695e+02

>> sum(absg.^.9999)
ans = 5.417630420657751e+04
```

To obtain these numbers, Chebfun has silently adjusted the exponents and then ap-

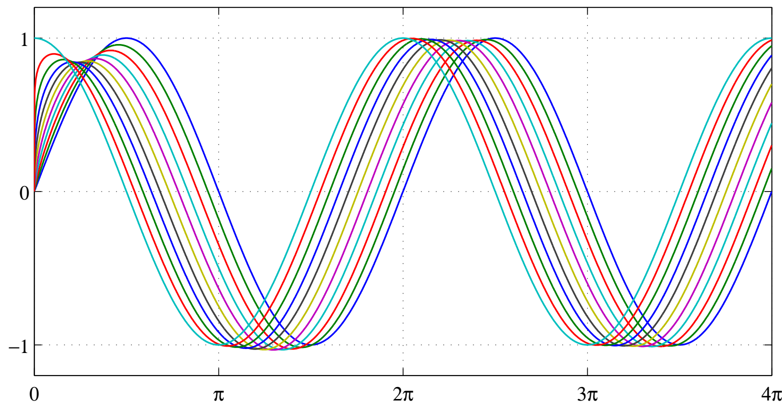

Fig. 7.1. *The function* $\sin x$ *on* $[0, 4\pi]$ *together with its derivatives of fractional orders* $0.1, 0.2, \ldots, 1$. *These computations rely on Chebfun's Gauss–Jacobi quadrature operations for sampling functions defined by the integral* (6.1).

plied the corresponding Gauss–Jacobi formulas. Chebfuns for $|\Gamma(x)|^{1/2}$ and $|\Gamma(x)|^{-1/2}$ are also plotted in Figure 6.1.

7. Fractional derivatives and integrals. An established idea of analysis is the notion of *fractional calculus*, the study of derivatives and integrals of fractional rather than integral order. Suppose, for example, that u is a continuous function defined on an interval $[a, b]$, and consider the *Riemann–Liouville integral*

$$I^{(\nu)}u(x) = \frac{1}{\Gamma(\nu)} \int_a^x (x-s)^{\nu-1} u(s)\, ds. \tag{7.1}$$

If ν is an integer, (7.1) gives the νth indefinite integral of u, and one may take the same formula as a definition of a νth-order fractional integral for arbitrary $\nu > 0$. By differentiating the result one or more times, we may extend the same definitions to fractional differentiation operators. There is also an alternative notion of the fractional derivative, due to Caputo, in which one differentiates first and then integrates. Ideas like this have far-reaching generalizations which are described in various places including [14] and [17].

The integrand of (7.1) has a singularity of type $x^{\nu-1}$, which makes it a perfect candidate for evaluation by Chebfun's Gauss–Jacobi quadrature capabilities as described in the last section. In Chebfun, the `diff` command computes derivatives, an overload of Matlab's `diff` for finite differences. If one specifies a non-integer order to `diff`, Chebfun applies (7.1) (or a Caputo alternative if the flag `'caputo'` is given) to compute the fractional derivative. The same fractional-order functionality is also accessible through the indefinite integral command `cumsum`.

Figure 7.1 illustrates fractional differentiation by ploting $\sin x$ on the interval $[0, 4\pi]$ together with its derivatives of orders $\alpha = 0.1, 0.2, \ldots, 1$. Each curve is approximately a translation of $\sin x$ by a distance $\pi\alpha/2$ to the left, but these translations are only approximate because of effects of the boundary at $x = 0$. Unlike integer-order differentiation and (up to constants) integer-order integration, fractional-order differentiation and integration are non-local operations.

We should say a word to clarify the Chebfun computations involved in implementing these operations of fractional calculus. To evaluate (7.1) at a single point x,

the system constructs a chebfun with an appropriate singularity at one end and then integrates it by Gauss–Jacobi quadrature. Producing results like those of Figure 7.1, however, requires the computation of new chebfuns representing (7.1) as a function of x. This is done by first fixing the singularity appropriately, then constructing a chebfun adaptively in the standard fashion by sampling the function on finer and finer grids until convergence to 15 or 16 digits is achieved.

Although nothing mathematically deep is going on here, computations like these would be sufficiently complicated without Chebfun that it is rare to see a figure like Figure 7.1 in which numerically evaluated fractional derivatives or integrals are plotted. Indeed, we do not know of any such figures in the literature. This suggests that Chebfun offers entirely new possibilities for practical explorations of fractional integrals and derivatives.

8. Functions defined on unbounded intervals. Chebfun also contains algorithms for representing functions, and integrating them, on unbounded integrals of the form $[a, \infty)$, $(-\infty, b]$, or $(-\infty, \infty)$. These were implemented by Rodrigo Platte around 2008, and like the other features described here, they have not been presented in published form before. Chebfun treats unbounded intervals by applying nonlinear changes of variables to reduce them to $[-1, 1]$. The software makes it possible to utilize quite arbitrary maps, but by default, the maps are rational functions of the form $(cx + d)/(ex + f)$. In practice, functions can be represented so long as they approach zero, or a constant, at a reasonably rapid rate as x approaches the infinite limits.

This leads to a quite efficient Chebfun capability for quadrature on infinite intervals, ultimately achieved by applying Chebfun's standard methods to the transplant on $[-1, 1]$. For example, here is the integral of e^{-x} from 0 to ∞:

```
>> f = chebfun('exp(-x)',[0,inf]);
>> length(f)
ans =  41
>> sum(f)
ans = 1.000000000000000
```

Here is the result for the more complicated function $e^{-x}\sin(100x)$:

```
>> f = chebfun('sin(100*x).*exp(-x)',[0,inf]);
>> length(f)
ans = 6403
>> sum(f)
ans = 0.009999000099987
```

9. Barycentric nodes and weights. In this final section we describe Chebfun's capabilities for computing barycentric interpolation weights associated with Legendre or more generally Gauss–Jacobi points quickly even for large n, based on the GLR algorithm and an observation of Wang and Xiang [24].

Suppose $s_0, \ldots, s_n$ are a set of $n+1$ distinct points, and $f_0, \ldots, f_n$ are a set of data given at these points. Then it is well known that the unique polynomial interpolant of degree at most $\leq n$ through these data is given by the *barycentric interpolation formula* [2, 21]:

$$p(x) = \sum_{j=0}^{n} \frac{v_j f_j}{x - s_j} \Bigg/ \sum_{j=0}^{n} \frac{v_j}{x - s_j}, \tag{9.1}$$

with the special case $p(x) = f_j$ if $x = s_j$ for some j, where the *barycentric weights* $\{v_j\}$ are defined by

$$v_j = \frac{C}{\prod_{k \neq j}(s_j - s_k)}, \tag{9.2}$$

and the constant C can be chosen arbitrarily since it cancels in the numerator and denominator of (9.1). An equivalent formula is

$$v_j = \frac{C}{\ell'(s_j)}, \tag{9.3}$$

where ℓ is the node polynomial

$$\ell(x) = \prod_{j=0}^{n}(x - s_j) \tag{9.4}$$

[21, Chap. 5]. The formula (9.1) is not just mathematically correct, but the basis of a fast and numerically stable numerical algorithm, at least when the interpolation points are distributed with suitable clustering near ± 1, as proved by Higham [11].

If $\{s_j\}$ are Chebyshev points, then (9.2) reduces to a simple form involving barycentric weights ± 1 of alternating sign, or $\pm 1/2$ for $j = 0$ and n [16]. If $\{s_j\}$ are Legendre or more generally Gauss–Jacobi points, on the other hand, no simple formula for the barycentric weights is known. However, Wang and Xiang have observed that they are related to the quadrature weights as follows [24, Thm. 3.1]:

$$v_j = (-1)^j \sqrt{(1 - s_j^2)w_j}. \tag{9.5}$$

One can derive this formula from (9.3) and the fact that the quadrature weights for the Gauss–Jacobi formula with parameters α, β are given by

$$w_j = \frac{C^{(\alpha,\beta)}}{(1 - s_j^2)[\ell'(s_j)]^2} \tag{9.6}$$

[18, eq. (15.3.1)], [25], where $C^{(\alpha,\beta)}$ is a constant. Chebfun uses these results to compute quadrature weights $\{w_j\}$ and barycentric weights $\{v_j\}$ as follows. First, the GLR algorithm returns derivatives $\ell'(s_j)$ at the quadrature nodes in $O(n)$ operations. The weights are then computed from (9.3) and (9.6). Thus we immediately get an $O(n)$ algorithm for barycentric interpolation in Legendre or Gauss–Jacobi points. Chebfun returns both sets of weights when `legpts` or `jacpts` is invoked with an additional argument, like this:

```
>> [s,w,v] = legpts(3)
s =
  -0.774596669241483
                   0
   0.774596669241483
w =
   0.555555555555556   0.888888888888889   0.555555555555556
v =
   0.500000000000000
  -1.000000000000000
   0.500000000000000
```

Increasing 3 to 10000, as in the opening example of this paper, gives barycentric weights with no additional computing time.

```
>> tic, [s,w,v] = legpts(10000); toc
Elapsed time is 0.263366 seconds.
```

Suppose, for example, one wished to evaluate at $x = 0$ the polynomial interpolant in 10000 Legendre points to $f(x) = (1+1000x^2)^{-1}$. One could proceed like this, using Chebfun's `bary` command with explicit third and fourth arguments for barycentric nodes and weights:

```
>> f = 1./(1+1000*s.^2);
>> p0 = bary(0,f,s,v)
p0 = 0.999999999999998
```

This is very close to the correct answer, which would match the value 1 to many more than 16 digits of precision. Chebfun also generalizes these computations to barycentric interpolation by Hermite and Laguerre polynomials (`hermpts`, `lagpts`), for which it is possible to show a similar relation between the quadrature weights $\{w_j\}$ and the node polynomial $\ell'(s_j)$.

REFERENCES

[1] P. Assheton, *Comparing Chebfun to Adaptive Quadrature Software,* thesis, MSc in Mathematical Modelling and Scientific Computing, Oxford University, 2008.

[2] J.-P. Berrut and L. N. Trefethen, Barycentric Lagrange interpolation, *SIAM Rev.* 46 (2004), 501–517.

[3] C. W. Clenshaw and A. R. Curtis, A method for numerical integration on an automatic computer, *Numer. Math.* 2 (1960), 197–205.

[4] T. O. Espelid, Doubly adaptive quadrature routines based on Newton–Cotes rules, *BIT Numer. Math.* 43 (2003), 319–337.

[5] T. O. Espelid, Extended doubly adaptive quadrature routines, Tech. Rep. 266, Dept. of Informatics, U. of Bergen.

[6] W. M. Gentleman, Implementing Clenshaw–Curtis quadrature I and II, *J. ACM* 15 (1972), 337–346.

[7] A. Glaser, X. Liu and V. Rokhlin, A fast algorithm for the calculation of the roots of special functions, *SIAM J. Sci. Comp.* 29 (2007), 1420–1438.

[8] G. H. Golub and J. H. Welsch, Calculation of Gauss quadrature rules, *Math. Comp.* 23 (1969), 221–230.

[9] P. Gonnet, Increasing the reliability of adaptive quadrature using explicit interpolants, *ACM Trans. Math. Softw.* 37 (2010), 26:2–26:32.

[10] P. Gonnet, Battery test of Chebfun as an integrator, `http://www.maths.ox.ac.uk/chebfun/examples/quad`, 2010.

[11] N. J. Higham, The numerical stability of barycentric Lagrange interpolation, *IMA J. Numer. Anal.* 24 (2004), 547–556.

[12] Octave software, `http://www.octave.org/`.

[13] H. O'Hara and F. J. Smith, Error estimation in the Clenshaw–Curtis quadrature formula, *Comput. J.* 11 (1968), 213–219.

[14] K. B. Oldham and J. Spanier, *The Fractional Calculus: Integrations and Differentiations of Arbitrary Order*, Academic Press, 1974.

[15] M. Richardson, *Approximating Divergent Functions in the Chebfun System,* thesis, MSc in Mathematical Modelling and Scientific Computing, Oxford University, 2009.

[16] H. E. Salzer, Lagrangian interpolation at the Chebyshev points $x_{n,\nu} = \cos(\nu\pi/n)$, $\nu = 0(1)n$; some unnoted advantages, *Computer J.* 15 (1972), 156–159.

[17] S. G. Samko, A. A. Kilbas, and O. I. Marichev, *Fractional Integrals and Derivatives,* Gordon and Breach, 1993.

[18] G. Szegő, *Orthogonal Polynomials,* Amer. Math. Soc., 1939.

[19] L. N. Trefethen, Is Gauss quadrature better than Clenshaw–Curtis? *SIAM Rev.* 50 (2008), 67–87.

[20] L. N. Trefethen, Six myths of polynomial interpolation and quadrature, *Maths. Today* 47 (2011), 184–188.

[21] L. N. Trefethen, *Approximation Theory and Approximation Practice,* SIAM, to appear in 2013.

[22] L. N. Trefethen and others, Chebfun Version 4.0, 2011, `http://www.maths.ox.ac.uk/chebfun/`.

[23] J. Waldvogel, Fast construction of the Fejér and Clenshaw–Curtis quadrature rules, *BIT Numer. Math.* 46 (2006), 195–202.

[24] H. Wang and S. Xiang, On the convergence rates of Legendre approximation, *Math. Comp.* 81 (2012), 861–877.

[25] C. Winston, On mechanical quadratures formulae involving the classical orthogonal polynomials, *Ann. Math.* 35 (1934), 658–677.