Computing numerically with functions instead of numbers

Lloyd N. Trefethen
Mathematical Institute
University of Oxford
Oxford OX2 6GG, UK
trefethen@maths.ox.ac.uk

ABSTRACT

Science and engineering depend upon computation of functions such as flow fields, charge distributions, and quantum states. Ultimately, such computations require some kind of discretization, but in recent years it has become possible in many cases to hide the discretizations from the user. We present the Chebfun system for numerical computation with functions, which is based on a key idea: an analogy of floating-point arithmetic for functions rather than numbers.

1. INTRODUCTION

The oldest problem of computing is, how can we calculate mathematical quantities? As other aspects of computing have entered into every corner of our lives, mathematical computation has become a less conspicuous part of computer science, but it has not gone away. On the contrary, it is bigger than ever, the basis of much of science and engineering.

The mathematical objects of interest in science and engineering are not just individual numbers but *functions*. To make weather predictions, we simulate velocity and pressure and temperature distributions, which are multidimensional functions evolving in time. To design electronic devices, we compute electric and magnetic fields, which are also functions. Sometimes the physics of a problem is described by long-established differential equations such as the Maxwell or Schrödinger equations, but just because the equations are understood doesn't mean the problem is finished. It may still be a great challenge to solve the equations.

How do we calculate functions? The almost unavoidable answer is that they must be discretized in one way or another, so that derivatives, for example, may be replaced by finite differences. Numerical analysts and computational engineers are the experts at handling these discretizations.

As computers grow more powerful, however, a new possibility has come into play: hiding the discretizations away so that the scientist does not have to see them. This is not feasible yet for weather prediction, but for certain kinds of desktop computing, it is becoming a reality. This paper introduces the Chebfun software system, which has followed this vision from its inception in 2002. For functions of one variable, $f(x)$, the aim has been largely achieved, and progress is well underway for functions of two variables, $f(x, y)$.

Chebfun is built on an analogy. To work with real numbers on a computer, we typically approximate them to 16 digits by finite bit strings: *floating-point numbers,* with an associated concept of *rounding* at each step of a calculation. To work with functions, Chebfun approximates them to 16 digits by polynomials (or piecewise polynomials) of finite degree: Chebsyhev expansions, again with an associated concept of rounding. Thus the key to numerical computation with functions is the generalization of the ideas of floating-point approximation and rounding from numbers to functions.

2. A COMBINATORIAL EXPLOSION

Haven't discretizations in general, and floating-point numbers in particular, been rendered superfluous by the introduction of symbolic systems like Mathematica or Maple? It is worth taking a moment to explain why the answer is no, for this will help elucidate the basis of our algorithms for numerical computing with functions.

We begin with what looks like an encouraging observation: if x and y are rational numbers, then so are $x + y$, $x - y$, xy, and x/y (assuming $y \neq 0$). Since rational numbers can readily be represented on computers, this might seem to suggest that there is no need for floating-point arithmetic with its inexact process of rounding. If a computer works in rational arithmetic, no error is ever made, so it might seem that in principle, much of numerical computation could be carried out exactly.

The first obstacle we encounter is that not every interesting real number x is rational (think of the hypotenuse of a triangle). However, this alone is not a serious problem, as x can be approximated arbitrarily closely by rationals.

The bigger problem is that when we try to construct such approximations by practical algorithms, we run into combinatorial or exponential explosions. For example, suppose we wish to find a root of the polynomial

$$p(x) = x^5 - 2x^4 - 3x^3 + 3x^2 - 2x - 1.$$

We can approximate an answer to great accuracy by rational numbers if we take a few steps of Newton's method, taught in any introductory numerical analysis course. Let us do

The original version of this paper was published with the same title in *Mathematics in Computer Science 1* (2007), 9–19.

Table 1: Five steps of Newton's method in rational arithmetic to find a root of a quintic polynomial.

$$x^{(0)} = 0$$

$$x^{(1)} = -\frac{1}{2}$$

$$x^{(2)} = -\frac{22}{95}$$

$$x^{(3)} = -\frac{11414146527}{36151783550}$$

$$x^{(4)} = -\frac{437115663193076384403256764909499867587929989600855536}{1386343327900876161184081275583890033212689660909186 25}$$

$$x^{(5)} = -\frac{724391479176820176129001381878925973035003883604754393117804119434357926010580274469629922882064184585670017703551996316651611596346 3}{2297460237315758733339908166643200351477598472080210886600668747832494887509884519822479758228984471808467983259225717929917685478944 49}$$

$$\frac{4562735299921308664663139405767412052875538201240642484300698212354536105198706894715223176068754569028985198376505504345452967792 1}{1536221568972260935865495519518216876316931568370465908144002495419674804116675018139752278347161906687414800535564210785107754125 0}$$

this, beginning from the initial guess $x^{(0)} = 0$. The startling result is shown in Table 1.

There is a problem here! As approximations to an exact root of p, the rational numbers displayed in the table are accurate to approximately 0, 0, 1, 3, 6, and 12 digits, respectively; the number of useful digits doubles at each step thanks to the quadratic convergence of Newton's method. Yet the lengths of the numerators are 1, 1, 2, 10, 53, and 265 digits, expanding by a factor of about 5 at each step since the degree of p is 5. After three more steps we will have an answer $x^{(8)}$ accurate to 100 digits, but represented by numerator and denominator each about 33,125 digits long, and storing it will require 66 kilobytes. If we were so foolish as to try to take 20 steps of Newton's method in this mode, we would need 16 terabytes to store the result.

Such difficulties are ubiquitous. Rational computations, and symbolic computations in general, have a way of expanding exponentially. If nothing is done to counter this effect, computations grind to a halt because of excessive demands on computing time and memory. This is ultimately the reason why symbolic computing, though powerful when it works, plays such a circumscribed role in computational science. As an example with more of a flavor of functions rather than numbers, suppose we want to know the indefinite integral of the function

$$f(x) = e^x \cos^5(6x) \sin^6(5x).$$

This happens to be a function that can be integrated analytically, but the result is not simple. The Wolfram Mathematica Online Integrator produces an answer that consists of the expression

$$\frac{5e^x(24\sin(24x) + \cos(24x))}{295424}$$

plus twenty other terms of similar form, with denominators ranging from 512 to 3687424. Working with such expressions is unwieldy when it is possible at all. An indication of their curious status is that if I wanted to be confident that this long formula was right, the first thing I would do was see if it matched results from a numerical computation.

3. FLOATING-POINT ARITHMETIC

It is in the light of such examples that I would like to consider the standard alternative to rational arithmetic, namely floating-point arithmetic. As is well known, this is the idea of representing numbers on computers by, for example, 64-bit binary words containing 53 bits (≈ 16 digits) for a fraction and 11 for an exponent. (These parameters correspond to the IEEE double precision standard.) Konrad Zuse invented floating-point arithmetic in Germany before World War II, and the idea was developed by IBM and other manufacturers a few years later. The IEEE standardization came in the mid-1980s, and is beautifully summarized in the book by Overton [14]. For more up-to-date details, see [13].

There are two aspects to floating-point technology: a *representation* of real (and complex) numbers via a subset of the rationals, and a prescription for *rounded arithmetic*. These principles combine to halt the combinatorial explosion. Thus, for example, if two 53-bit numbers are multiplied, the mathematically exact result would require about 106 bits to be represented. Instead of accepting this, we round the result down to 53 bits again. More generally, most floating-point arithmetic systems adhere to the following principle: when an operation $+, -, \times, /$ is performed on two floating-point numbers, the output is the exactly correct result rounded to the nearest floating-point number, with ties broken by a well-defined rule. This implies that every floating-point operation is exact except for a small relative error:

$$\text{computed}(x * y) = (x * y)(1 + \varepsilon), \quad |\varepsilon| \le \varepsilon_{\text{mach}}. \qquad (1)$$

Here $*$ denotes one of the operations $+, -, \times, /$, and we are ignoring the possibilities of underflow or overflow. The IEEE double precision value of "machine epsilon" is $\varepsilon_{\text{mach}} = 2^{-53} \approx 1.1 \times 10^{-16}$.

Equation (1) implies an important corollary: when two floating-point numbers x and y are combined on the computer by an operation $*$, the result computed$(x*y)$ is exactly equal to $\tilde{x} * \tilde{y}$ for some two numbers $\tilde{x}$ and $\tilde{y}$ that are close to x and y in a relative sense:

$$\text{computed}(x * y) = \tilde{x} * \tilde{y}, \quad \frac{|x - \tilde{x}|}{|x|}, \frac{|y - \tilde{y}|}{|y|} \le \varepsilon_{\text{mach}}. \quad (2)$$

Numerical analysts say that the operations $+, -, \times, /$ are *backward stable*, delivering the exactly correct results for inputs that are slightly perturbed from their correct values in a relative sense. The same conclusion holds or nearly holds for good implementations of other elementary operations, often unary instead of binary, such as $\sqrt{}$, exp, or sin [13].

Floating-point arithmetic is not widely regarded as one of computer science's sexier topics. A common view is that it

is an ugly but necessary engineering compromise. We can't do arithmetic honestly, the idea goes, so we cheat a bit — unfortunate, but unavoidable, or as some have called it, a "Faustian bargain." In abandoning exact computation we sell our souls, and in return, we get some numbers.

I think one can take a more positive view. Floating-point arithmetic is an *algorithm,* no less than a general procedure for containing the combinatorial explosion. Consider the Newton iteration of Table 1 again, but now carried out in IEEE 16-digit arithmetic:

$$\begin{aligned}
x^{(0)} &= 0.00000000000000,\\
x^{(1)} &= -0.50000000000000,\\
x^{(2)} &= -0.33684210526316,\\
x^{(3)} &= -0.31572844839629,\\
x^{(4)} &= -0.31530116270328,\\
x^{(5)} &= -0.31530098645936,\\
x^{(6)} &= -0.31530098645933,\\
x^{(7)} &= -0.31530098645933,\\
x^{(8)} &= -0.31530098645933.
\end{aligned}$$

It's the same process as before, less startling without the exponential explosion, but far more useful. Of course, though these numbers are printed in decimal, what is really going on in the computer is binary. The exact value at the end, for example, is not the decimal number printed but

$$x^{(8)} = -0.01010000101101111001000 0\ldots \\ 1100000100111101010001111000 01_{\text{binary}}.$$

Abstractly speaking, when we compute with rational numbers, we might proceed like this:

> *Compute an exact result,*
> *then round it to a certain number of bits.*

The problem is that the exact result is often exponentially lengthy. Floating-point arithmetic represents an alternative idea:

> *Round the computation at every step,*
> *not just at the end.*

This strategy has proved overwhelmingly successful. At a stroke, combinatorial explosion ceases to be an issue. Moreover, so long as the computation is not numerically unstable in a sense understood thoroughly by numerical analysts, the final result will be accurate. This is what one observes in practice, and it is also the rigorous conclusion of theoretical analysis of thousands of algorithms investigated by generations of numerical analysts [11].

4. CHEBFUN

Chebfun is an open-source software system developed over the past decade at Oxford by myself and a succession of students and postdocs including Zachary Battles, Ásgeir Birkisson, Nick Hale, and Alex Townsend, as well as Toby Driscoll at the University of Delaware (a full list can be found in the Acknowledgments and at `www.chebfun.org`). The aim of Chebfun is to extend the ideas we have just discussed from numbers to functions. Specifically, Chebfun works with piecewise smooth real or complex functions defined on an interval $[a, b]$, which by default is $[-1, 1]$. A function is represented by an object known as a *chebfun.* (We write "Chebfun" as the name of the system and "chebfun" for a representation of an individual function.) If f and g are chebfuns, we can perform operations on them such as $+, -, \times, /$, as well as other operations like exp or sin. The intention is not that such computations will be exact. Instead, the aim is to achieve an analogue of (2) for functions,

$$\text{computed}(f * g) = \tilde{f} * \tilde{g}, \quad \frac{\|f - \tilde{f}\|}{\|f\|}, \frac{\|g - \tilde{g}\|}{\|g\|} \le C\varepsilon_{\text{mach}} \quad (3)$$

(again ignoring underflow and overflow), where C is a small constant, with a similar property for unary operations. Here $\|\cdot\|$ is a suitable norm such as $\|\cdot\|_\infty$. Thus the aim of Chebfun is *normwise backward stable computation of functions.* We shall say more about the significance of (3) in Section 6.

Chebfun is implemented in MATLAB, a language whose object-oriented capabilities enable one to overload operations such as $+$, $-$, $\times$, $/$, sin, and exp with appropriate alternatives. Some of the methods defined for chebfuns are as follows (this list is about one-third of the total):

`abs`	`csc`	`kron`	`real`
`acos`	`cumprod`	`legpoly`	`remez`
`airy`	`cumsum`	`length`	`roots`
`angle`	`diff`	`log`	`round`
`arclength`	`dirac`	`max`	`sec`
`asin`	`eq`	`mean`	`semilogy`
`atan`	`erf`	`min`	`sign`
`atanh`	`exp`	`minus`	`sin`
`besselj`	`feval`	`mod`	`sinh`
`bvp4c`	`find`	`norm`	`spline`
`ceil`	`floor`	`null`	`sqrt`
`chebpade`	`gmres`	`ode45`	`std`
`chebpoly`	`heaviside`	`pinv`	`sum`
`chebpolyplot`	`imag`	`plot`	`svd`
`cond`	`integral`	`plus`	`tanh`
`conj`	`interp1`	`poly`	`times`
`conv`	`inv`	`polyfit`	`transpose`
`cos`	`isequal`	`prod`	`var`
`cosh`	`isinf`	`qr`	`waterfall`
`cot`	`isnan`	`rank`	
`coth`	`jacpoly`	`rank`	

MATLAB (or Python) programmers will recognize many of these as standard commands. In MATLAB, such commands apply to discrete vectors, or sometimes matrices, but in Chebfun, they perform continuous analogues of the operations on chebfuns. Thus for example `log(f)` and `sinh(f)` deliver the logarithm and the hyperbolic sine of a chebfun `f`, respectively. More interestingly, `sum(f)` produces the definite integral of `f` from a to b (a scalar), the analogue for continuous functions of the sum of entries of a vector. Similarly, `cumsum(f)` produces the indefinite integral of `f` (a chebfun), `diff(f)` computes the derivative (another chebfun), and `roots(f)` finds the roots in the interval $[a, b]$ (a vector of length equal to the number of roots).

Mathematically, the basis of Chebfun — and the origin of its name — is piecewise Chebyshev expansions. Let T_j denote the Chebyshev polynomial $T_j(x) = \cos(j \cos^{-1} x)$, of degree j, which equioscillates between $j + 1$ extrema ± 1 on $[-1, 1]$. The Chebyshev series for any Hölder continuous $f \in C[-1, 1]$ is defined by [20]

$$f(x) = {\sum_{j=0}^{\infty}}' a_j T_j(x), \quad a_j = \frac{2}{\pi} \int_{-1}^{1} \frac{f(x) T_j(x)}{\sqrt{1 - x^2}}\, dx, \qquad (4)$$

where the prime indicates that the term with $j = 0$ is multiplied by 1/2. (These formulas can be derived using the change of variables $x = \cos\theta$ from the Fourier series for the 2π-periodic even function $f(\cos\theta)$. Chebyshev series are essentially the same as Fourier series, but for nonperiodic functions.) Chebfun is based on storing and manipulating coefficients $\{a_j\}$ for such expansions. Many of the algorithms make use of the equivalent information of samples $f(x_j)$ at *Chebyshev points,*

$$x_j = \cos\frac{j\pi}{n}, \quad 0 \le j \le n, \tag{5}$$

and one can go back and forth to the representation (4) as needed by means of the Fast Fourier Transform (FFT). Each chebfun has a fixed finite n chosen to be large enough for the representation, according to our best estimate, to be accurate in the local sense (3) to 16 digits. Given data $f_j = f(x_j)$ at the Chebyshev points (5), other values can be determined by the *barycentric interpolation formula* [17],

$$f(x) = \sum_{j=0}^{n} \frac{w_j}{x - x_j} f_j \Bigg/ \sum_{j=0}^{n} \frac{w_j}{x - x_j}\ , \tag{6}$$

where the weights $\{w_j\}$ are defined by

$$w_j = (-1)^j \delta_j, \quad \delta_j = \begin{cases} 1/2, & j = 0 \text{ or } j = n, \\ 1, & \text{otherwise.} \end{cases} \tag{7}$$

(If x happens to be exactly equal to some x_j, one bypasses (6) and sets $f(x) = f(x_j)$.) This method is known to be numerically stable, even for polynomial interpolation in millions of points [12].

If f is analytic on $[-1, 1]$, its Chebsyhev coefficients $\{a_j\}$ decrease exponentially [20]. If f is not analytic but still several times differentiable, they decrease at an algebraic rate determined by the number of derivatives. It is these properties of rapid convergence that Chebfun exploits to be a practical computational tool. Suppose a chebfun is to be constructed, for example by the constructor statement

```
f = chebfun(@(x) sin(x)).
```

What happens when this command is executed is that the system performs adaptive calculations to determine what degree of polynomial approximation is needed to represent $\sin(x)$ to about 15 digits of accuracy. The answer in this case turns out to be 13, so that our 15-digit approximation is actually

$$\begin{aligned} f(x) = \ & 0.88010117148987T_1(x) - 0.03912670796534T_3(x) \\ & + 0.00049951546042T_5(x) - 0.00000300465163T_7(x) \\ & + 0.00000001049850T_9(x) - 0.00000000002396T_{11}(x) \\ & + 0.00000000000004T_{13}(x) \end{aligned}$$

when represented in the well-behaved basis of Chebyshev polynomials $\{T_k\}$, or

$$\begin{aligned} f(x) = \ & 1.00000000000000x - 0.16666666666665x^3 \\ & + 0.00833333333314x^5 - 0.00019841269737x^7 \\ & + 0.00000275572913x^9 - 0.00000002504820x^{11} \\ & + 0.00000000015785x^{13} \end{aligned}$$

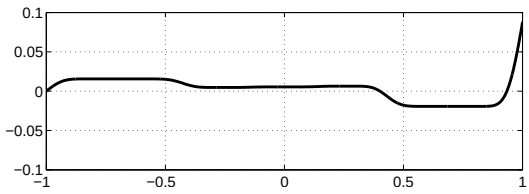
in the badly-behaved but more familiar basis of monomials. This is a rather short chebfun; more typically the length might be 50 or 200. For example, `chebfun(@(x) sin(50*x))` has length 90, and `chebfun(@(x) exp(-1./x.^2))` has length 219.

Having settled on representing functions by Chebyshev expansions and interpolants, we next face the question of how to implement mathematical operations such as those in the list above. This is a very interesting matter, and details of the many algorithms used in Chebfun can be found in [20] and the other references. For example, zeros of chebfuns are found by `roots` by a recursive subdivision of the interval combined with eigenvalue computations for Chebyshev "colleague matrices" [3], and global maxima and minima are determined by `max` and `min` by first finding zeros of the derivative. All these computations are fast and accurate even when the underlying polynomial representations have degrees in the thousands.

At the end of Section 2 we considered an indefinite integral. In Chebfun indefinite integration is carried out by the command `cumsum`, as mentioned above, and that example on the interval $[-1, 1]$ could go like this:

```
x = chebfun(@(x) x);
f = exp(x).*cos(6*x).^5.*sin(5*x).^6;
g = cumsum(f);
```

The chebfun g is produced in about 0.02 secs. on a desktop machine, a polynomial of degree 94 accurate to about 16 digits. Here is a plot:



5. TAMING THE EXPLOSION

As mentioned earlier, when two 53-bit numbers are multiplied, an exact result would normally require 106 bits, but floating-point arithmetic rounds this to 53. Chebfun implements an analogous compression for polynomial approximations of functions as opposed to binary approximations of numbers. For example, suppose x is the chebfun corresponding to the linear function x on $[-1, 1]$. If we execute the commands

```
f = sin(x),  g = cos(x),  h = f.*g,
```

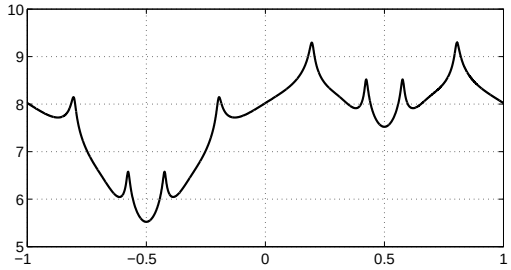
we find that the chebfuns `f` and `g` have degrees 13 and 14, respectively. One might expect their product to have degree 27, but in fact, `h` has degree only 17. This happens because at every step, the system automatically discards Chebyshev coefficients that are below machine precision — just as floating-point arithmetic discards bits below the 53rd. The degree grows only as the complexity of the functions involved genuinely grows, as measured on the scale of machine epsilon.

Here is an example to illustrate how this process contains the explosion of polynomial degrees. The program

```
f = chebfun(@(x) sin(pi*x));
s = f;
for j = 1:15
  f = (3/4)*(1 - 2*f.^4);
```

```
  s = s + f;
end
plot(s)
```

begins by constructing a chebfun `f` corresponding to the function $\sin(\pi x)$ on the interval $[-1,1]$, with degree 19. Then it takes fifteen steps of an iteration that raises the current `f` to the 4th power at each step. The result after a fraction of a second on a desktop computer is a rather complicated chebfun, of degree 3378, which looks like this:



The degree 3378 may seem high, but it is very low compared to what it would be if the fourth powers were computed without dropping small coefficients, namely $19\times4^{15} = 20{,}401{,}094{,}656$! Thus the complexity has been curtailed by a factor of millions, yet with little loss of accuracy. For example, the command `roots(s-8)` now takes less than a second to compute the twelve points $x \in [-1,1]$ with $s(x) = 8$:

```
-0.99293210741191
-0.81624993429017
-0.79888672972343
-0.20111327027657
-0.18375006570983
-0.00706789258810
 0.34669612041826
 0.40161707348210
 0.44226948963247
 0.55773051036753
 0.59838292651791
 0.65330387958174
```

These results are all correct except in the last digit.

Once one has a chebfun representation, further computations are easy. For example, `sum(s)` returns the definite integral `15.26548382582674` in a few thousandths of a second. The exact value is $15.26548382582674700943\ldots$.

6. NORMWISE BACKWARD STABILITY

Does Chebfun live up to the vision of an analogue for functions of floating-point arithmetic for numbers? In considering this question, a good starting point is the normwise backward stability condition (3), and in particular, it is productive to focus on two questions:

(I) How close does Chebfun come to achieving (3)?

(II) What are the implications of this condition?

The answer to (I) appears to be that Chebfun does satisfy (3), at least for the basic operations $+, -, \times, /$. This has not been proved formally and it is a project for the future to develop a rigorous theory. To explain how (3) can hold, let us consider the mode in which each chebfun is represented precisely by a finite Chebyshev series with floating-point coefficients (instead of values at Chebyshev points). The property (3) for $+$ and $-$ stems from the corresponding properties for addition and subtraction of floating-point numbers, together with the numerical stability of barycentric interpolation [12]. For multiplication, the argument is only slightly more complicated, since again the operation comes down to one of Chebyshev coefficients. The more challenging fundamental operation is division, for in this case the quotient f/g is sampled pointwise at various Chebyshev points and then a new Chebyshev series is constructed by the adaptive process used generally for chebfun construction. It is not known whether the current code contains safeguards enough to give a guarantee of (3), and this is a subject for investigation. In addition, it will be necessary to consider analogues of (3) for other Chebfun operations besides $+, -, \times, /$.

This brings us to (II), the question of the implications of (3). The easier part of the answer, at least for numerical analysts familiar with backward error analysis, is to understand exactly what the property (3) does and does not assert about numerical accuracy. A crucial fact is that the bound involves the global norms of the function f and g, not their values at particular points. For example, we may note that if two chebfuns f and g give $(f-g)(x) < 0$ at a point x, then from (3) we cannot conclude that $f(x) < g(x)$. We can conclude, however, that there are nearby chebfuns $\tilde{f}$ and $\tilde{g}$ with $\tilde{f}(x) < \tilde{g}(x)$. This is related to the "zero problem" that comes up in the theory of real computation [22]. It is well known that the problem of determining the sign of a difference of real numbers with guaranteed accuracy poses difficulties. However, Chebfun makes no claim to overcome these difficulties: the normwise condition (3) promises less.

Does it promise enough to be useful? What strings of computations in a system satisfying (3) at each step can be expected to be satisfactory? This is nothing less than the problem of *stability of Chebfun algorithms,* and it is a major topic for future research. Certainly there may be be applications where (3) is not enough to imply what one would like, typically for reasons related to the zero problem. For example, this may happen in some problems of geometry, where arbitrarily small coordinate errors may make the the difference between two bodies intersecting or not intersecting, or between convex and concave. On the other hand, generations of numerical analysts have found that such difficulties are by no means universal, that the backward stability condition (2) for floating-point arithmetic is sufficient to ensure success for many scientific computations. An aim of ours for the future will be to determine how far this conclusion carries over to condition (3) for chebfuns.

7. CHEBFUN SOFTWARE PROJECT

Chebfun began in 2002 as a few hundred lines of MATLAB code, written by Zachary Battles, for computing with global polynomial representations of smooth functions on $[-1,1]$, and this "core Chebfun" framework has been the setting for the discussion in this article. But in fact, the project has expanded greatly in the decade since then, both as a software effort and in its computational capabilities.

In terms of software, we have grown to an open-source project hosted on GitHub with currently about a dozen developers, most but not all based at Oxford. The code is written in MATLAB, which is a natural choice for this kind of work because of its vector and matrix operations,

although implementations of parts of core Chebfun have been produced by various people in other languages including Python, C, Julia, and Octave. To date there have been about 20,000 Chebfun downloads. We interact regularly with users through bug reports, help requests by email, and other communications, but we believe we are not alone among software projects in feeling that we have an inadequate understanding of who our users are and what they are doing.

In terms of capabilities, here are some of the developments beyond the core ideas emphasized in this article. The abbreviations ODE and PDE stand for ordinary and partial differential equations.

- piecewise smooth functions [15]
- periodic functions (Fourier not Chebyshev) [6]
- fast edge detection for determining breakpoints [15]
- infinite intervals $[a,\infty)$, $(-\infty,b]$, $(-\infty,\infty)$
- functions with poles and other singularities
- delta functions of arbitrary order
- Padé, Remez, CF rational approximations [7, 16, 21]
- fast Gauss and Gauss–Jacobi quadrature [8, 10]
- fast Chebyshev $\leftrightarrow$ Legendre conversions [9]
- continuous QR factorization, SVD, least-squares [1, 19]
- representation of linear operators [5]
- solution of linear ODEs [5]
- solution of integral equations [4]
- solution of eigenvalue problems [5]
- exponentials of linear operators [5]
- Fréchet derivatives via automatic differentiation [2]
- solution of nonlinear ODEs [2]
- PDEs in one space variable plus time
- Chebgui interface to ODE/PDE capabilities
- Chebfun2 extension to rectangles in 2D [18]

We shall not attempt to describe these developments, but here are a few comments. For solving ODE boundary value problems, whether scalars or systems and smooth or just piecewise smooth, Chebfun and its interface Chebgui have emerged as the most convenient and flexible tool in existence, making it possible to solve all kinds of problems with minimal effort with accuracy close to machine precision (these developments are due especially to Ásgeir Birkisson, Toby Driscoll, and Nick Hale) [2]. For computing quadrature nodes and weights, convolution, and conversion between Legendre and Chebyshev coefficient representations, Chebfun contains codes implementing new algorithms that represent the state of the art, enabling machine accuracy for even millions of points in seconds (these developments are due to Nick Hale and Alex Townsend [8, 9]). Extensions to multiple dimensions have begun with Alex Townsend's Chebfun2 code initially released in 2013 [18].

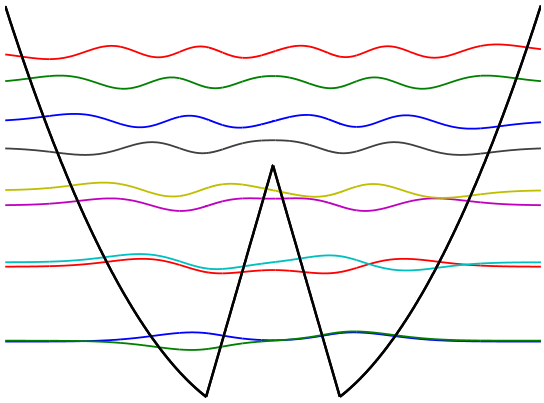


Figure 1: Schrödinger eigenstates computed by `quantumstates(V)`, where `V` is a chebfun representing a piecewise smooth potential function.

The best way to get a sense of the wide range of problems that can be solved by this kind of computing is to look at the collection of Chebfun Examples available online at the web site `www.chebfun.org`. Approaching 200 in number, the Examples are organized under headings that look like chapters of a numerical analysis textbook (optimization, quadrature, linear algebra, geometry, ...), with dozens of short discussions in each category of problems ranging from elementary to advanced.

Here is an example that gives a taste of Chebfun's ability to work with functions that are only piecewise smooth, and to solve ODE eigenvalue problems. The sequence

```
x = chebfun(@(x) x,[-2,2]);
V = max(x.^2/2,1-2*abs(x));
quantumstates(V)
```

produces the plot shown in Figure 1 as well as associated numerical output. The figure shows the first 10 eigenmodes of a Schrödinger operator $-h^2\partial^2 u/\partial x^2 + V(x)u(x)$ with the default value of Planck's constant $h = 0.1$. The potential function $V(x)$ consists of the parabola $x^2/2$ over the interval $[-2,2]$ maximized with a triangular barrier around $x = 0$, and it is represented by a piecewise-smooth Chebfun with four pieces. This kind of mathematics arises in any introductory quantum mechanics course; Chebfun makes exploring the dependence of eigenstates on potential functions almost effortless, yet with accuracy close to machine precision.

And here is an example that gives a taste of Chebfun-like computing on rectangles in 2D as implemented by Alex Townsend's extension Chebfun2. The sequence

```
f = chebfun2(@(x,y) exp(-(x.^2+y.^2))...
    .*sin(6*(2+x).*x).*sin(4*(3+x+y).*y));
contour(f)
```

defines and plots a chebfun2 representing an oscillatory function of x and y on the unit square $[-1,1]^2$. The command `max2` tells us its global maximum in a fraction of a second:

```
max2(f)
ans = 0.970892994917307
```

The algorithms underlying Chebfun2 are described in [18].

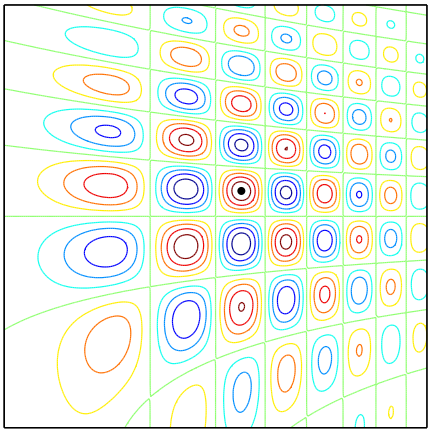


Figure 2: Two-dimensional extension of Chebfun: an oscillatory function represented by a chebfun2, with its maximum shown as a black dot.

8. CONCLUSION

Chebfun is being used by scientists and engineers around the world to solve 1D and some 2D numerical problems without having to think about the underlying discretizations. The Chebyshev technology it is built on is powerful, and it is hard to see any serious competition for this kind of high-accuracy representation of functions in one dimension.

At the same time, the deeper point of this article has been to put forward a vision that is not tied specifically to Chebyshev expansions or to other details of Chebfun. The vision is that by the use of adaptive high-accuracy numerical approximations of functions, computational systems can be built that "feel symbolic but run at the speed of numerics."

9. ACKNOWLEDGMENTS

In addition to the leaders mentioned at the beginning of Section 4, other contributors to the Chebfun project have included Anthony Austin, Folkmar Bornemann, Filomena di Tommaso, Pedro Gonnet, Stefan Güttel, Hrothgar, Mohsin Javed, Georges Klein, Hadrien Montanelli, Sheehan Olver, Ricardo Pachón, Rodrigo Platte, Mark Richardson, Joris Van Deun, Grady Wright, and Kuan Xu. It has been a fascinating experience working with these people over the past decade to rethink so much of discrete numerical mathematics in a continuous mode.

During 2008–2011 the Chebfun project was supported by the UK Engineering and Physical Sciences Council. Currently we are supported by MathWorks, Inc. and by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007–2013)/ERC grant agreement no. 291068. The views expressed in this article are not those of the ERC or the European Commission, and the European Union is not liable for any use that may be made of the information contained here.

10. REFERENCES

[1] Battles, Z. and Trefethen, L. N., An extension of MATLAB to continuous functions and operators, *SIAM J. Sci. Comp. 25* (2004), 1743–1770.

[2] Birkisson, Á. and Driscoll, T. A., Automatic Fréchet differentiation for the numerical solution of boundary-value problems, *ACM Trans. Math. Softw. 38* (2012), 26:1–28.

[3] Boyd, J. A., Computing zeros on a real interval through Chebyshev expansion and polynomial rootfinding, *SIAM J. Numer. Anal. 40* (2002), 1666–1682.

[4] Driscoll, T. A., Automatic spectral collocation for integral, integro-differential, and integrally reformulated differential equations, *J. Comp. Phys. 229* (2010), 5980–5998.

[5] Driscoll, T. A., Bornemann, F., and Trefethen, L. N., The Chebop system for automatic solution of differential equations, *BIT Numer. Math. 48* (2008), 701–723.

[6] Driscoll, T. A., Hale, N., and Trefethen, L. N., *Chebfun Guide,* Pafnuty Publications, Oxford, UK, 2014 (freely available at `www.chebfun.org`).

[7] Gonnet, P., Pachón, R. and Trefethen, L. N., Robust rational interpolation and least-squares, *Elect. Trans. Numer. Anal. 38* (2011), 146–167.

[8] Hale, N. and Townsend, A., Fast and accurate computation of Gauss–Legendre and Gauss–Jacobi quadrature nodes and weights, *SIAM J. Sci. Comput. 35* (2013), A652–A674.

[9] Hale, N. and Townsend, A., A fast, simple, and stable Chebyshev–Legendre transform using an asymptotic formula, *SIAM J. Sci. Comp. 36* (2014), A148–A167.

[10] Hale, N. and Trefethen, L. N., Chebfun and numerical quadrature, *Sci. China Math. 55* (2012), 1749–1760.

[11] Higham, N. J., *Accuracy and Stability of Numerical Algorithms,* 2nd ed., SIAM, 2002.

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

[13] J.-M. Muller et al., *Handbook of Floating-Point Arithmetic,* Birkhäuser, 2010.

[14] Overton, M. L. *Numerical Computing with IEEE Floating Point Arithmetic*, SIAM, 2001.

[15] Pachón, R., Platte, R. and Trefethen, L. N., Piecewise-smooth chebfuns, *IMA J. Numer. Anal. 30* (2010), 898–916.

[16] Pachón, R. and Trefethen, L. N., Barycentric-Remez algorithms for best polynomial approximation in the chebfun system, *BIT Numer. Math. 49* (2009), 721–741.

[17] Salzer, H. E., 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.

[18] Townsend, A. and Trefethen, L. N., An extension of Chebfun to two dimensions, *SIAM J. Sci. Comput. 35* (2013), C495–C518.

[19] Trefethen, L. N., Householder triangularization of a quasimatrix, *IMA J. Numer. Anal. 30* (2010), 887–897.

[20] Trefethen, L. N., *Approximation Theory and Approximation Practice,* SIAM, 2013.

[21] Van Deun, J. and Trefethen, L. N., A robust implementation of the Carathéodory–Fejér method for rational approximation, *BIT Numer. Math. 51* (2011), 1039–1050.

[22] Yap, C. K., Theory of real computation according to EGC, in *Reliable Implemention of Real Number Algorithms: Theory and Practice,* Lecture Notices in Comp. Sci. *5045* (2008), Springer-Verlag, pp. 193–237.