Math.comput.sci. 1 (2007), 9–19

1661-8270/010009-11, *published online* October 23, 2007
DOI 10.1007/s11786-007-0001-y

Mathematics in Computer Science

Computing Numerically with Functions Instead of Numbers

Lloyd N. Trefethen

For Richard Brent on his 60^{th} *birthday*

Abstract. Symbolic computation with functions of a real variable suffers from combinatorial explosion of memory and computation time. The alternative `chebfun` system for such computations is described, based on Chebyshev expansions and barycentric interpolation.

Mathematics Subject Classification (2000). Primary 41A10; Secondary 68W30.

Keywords. Chebfun, Chebyshev series.

1. Rational arithmetic and a combinatorial explosion

The arithmetic operations $+, -, \times, /$ map rational inputs to rational outputs. Thus in principle, as is well known, much of numerical computation could be carried out exactly on a computer.

For example, suppose we wish to find a root of the quintic polynomial

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

The answer won't be a rational number, but we can approach it very fast by rational numbers using Newton's method. If the initial guess is $x^{(0)} = 0$, here is what we find:

$$x^{(0)} = 0\,,$$
$$x^{(1)} = -\frac{1}{2}\,,$$
$$x^{(2)} = -\frac{22}{95}\,,$$
$$x^{(3)} = -\frac{11414146527}{36151783550}\,,$$

$$x^{(4)} = -\frac{437115663193076384403256764909499867587929989600855536}{13863433279008761611840812755838900332126896609091862 5},$$

$$x^{(5)} = -\frac{724391479176820176129001381878925973035003883604754393117804119434357926010580274469629 9}{229746023731575873333990816664320035147759847208021088660066874783249488750988451982247 97}$$

$$\frac{228820641845856700177035519963166516115963436345627352999213086646631394057674 1052875538}{582289844718084679815362215689722609358654953259225717929917685478944951951821687631 6931}$$

$$\frac{201240642484300698212354536105198706894715223176068754569028985198376505504345452967792 1}{568370465908144002495419674804116675018139752278347161906687414800535564210785107754125 0}.$$

There is a problem here! As approximations to an exact root of p, these rational numbers are accurate to approximately 0, 0, 1, 3, 6, and 12 digits, respectively; the number of good 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 33125 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, we would need 16 terabytes to store the result.

This difficulty is a familiar one. Rational computations, like 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.

2. Floating-point arithmetic

It is in this context that I would like to consider floating-point arithmetic. As is well known, this is the idea of representing numbers on computers by, for example (in the IEEE double precision standard), 64-bit binary words containing 53 bits (≈ 16 digits) for a fraction and 11 for an exponent. Konrad Zuse invented floating-point arithmetic in Germany before and during 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.

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 stop the combinatorial explosion. Thus for example, if two 53-bit numbers are multiplied, the result would typically require about 106 bits to be represented exactly. 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 should be the exactly correct result rounded to the nearest floating-point number. This implies that every floating-point operation is exact except for a small relative error:

$$\text{computed}(x * y) = (x * y)(1 + \varepsilon), \quad |\varepsilon| \le \varepsilon_{\text{machine}}. \tag{2.1}$$

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 $\varepsilon_{\text{machine}} = 2^{-53} \approx 1.1 \times 10^{-16}$ [11].

Equation (2.1) implies an important corollary:

$$\text{computed}(x * y) = \tilde{x} * \tilde{y}\,, \quad \frac{|x-\tilde{x}|}{|x|}, \frac{|y-\tilde{y}|}{|y|} \le \varepsilon_{\text{machine}}\,. \tag{2.2}$$

Thus each of the fundamental operations is *backward stable*, delivering the exactly correct result for inputs that are slightly perturbed in a relative sense. The same conclusion often holds for good implementations of other fundamental operations, often unary instead of binary, such as $\sqrt{\ }$, exp, or sin.

Floating-point arithmetic is not generally regarded as one of science's sexier topics. A widespread view is that it is an ugly though necessary engineering compromise. We can't do real 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 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. Incidentally, though the numbers above 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

$$x^{(8)} = -0.0101000010110111100100001100000100111101010001111000 01_{\text{binary}}\,.$$

Abstractly speaking, whenever 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 [8].

If there is a single essential reason for this good behavior, it is the phenomenon of backward stability, encapsulated for a single operation $*$ in the condition (2). More broadly, backward stability is the property that the solution obtained at the end of a floating-point computation is the exactly (or nearly exactly) correct solution for slightly perturbed data. This is a subtle and powerful idea, and it takes some getting used to. If a system obeys condition (2) but no more, then a subtraction of two nearly equal numbers x and y, for example, might yield a result that doesn't even have the right sign. This may seem bizarre, and in fact, computers that adhere to the stronger condition (1) will not produce this sign anomaly in a single subtraction. They may do so in a sequence of several operations, however. It is a well established principle in numerical analysis that although backward stability is not all one might wish for ideally, it is both realistic to achieve and powerful enough to guarantee accuracy of the ultimate solution in a wide range of computations. It is the "right" model for much of practical numerical computation. The definitive reference on these matters is Higham's *Accuracy and Stability of Numerical Algorithms* [8]; in earlier decades the key references were the two books by Wilkinson [14, 15].

The ideas we have been discussing are the basis of the whole field of computational science and played a part in the development of almost all the technology that surrounds us. Jet planes, mobile phones, automobiles and office buildings are all designed with floating-point arithmetic.

3. The chebfun system

My former student Zachary Battles and I have implemented a system whose aim is to extend these ideas from numbers to functions [1–3]. Specifically, our system works with smooth real or complex functions defined on $[-1, 1]$. An object of this kind in our representation is called a *chebfun*. If f and g are chebfuns, we can perform operations on them such as $+, -, \times, /$, as well as other operations like exp or sin. (For f/g, it is assumed that g is bounded away from 0.) The intention is not that such computations will be exact. Instead the aim is to achieve the analogue of (2.2),

$$\text{computed}(f * g) = \tilde{f} * \tilde{g}\,, \quad \frac{\|f - \tilde{f}\|}{\|f\|}, \frac{\|g - \tilde{g}\|}{\|g\|} \le C\varepsilon_{\text{machine}} \tag{3.1}$$

(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 the chebfun system is *normwise backward stable computation of functions.* We shall say more about the significance of (3.1) in §5.

The chebfun system is a class implemented in the language MATLAB. MATLAB is object-oriented, enabling programmers to overload standard operations such as +, −, ×, /, sin, and exp with appropriate alternatives. The operators defined for chebfuns are as follows:

`abs`	`display`	`ldivide`	`ne`	`real`	`sum`
`angle`	`end`	`length`	`norm`	`rescale`	`svd`
`chebfun`	`eq`	`log`	`null`	`roots`	`tan`
`chebpoly`	`erf`	`log10`	`pinv`	`semilogy`	`tanh`
`cond`	`erfc`	`log2`	`plot`	`sign`	`times`
`conj`	`erfcx`	`max`	`plus`	`sin`	`transpose`
`cos`	`erfinv`	`mean`	`poly`	`sinh`	`uminus`
`cosh`	`exp`	`min`	`power`	`size`	`uplus`
`ctranspose`	`horzcat`	`minus`	`prod`	`sqrt`	`var`
`cumprod`	`imag`	`mldivide`	`qr`	`std`	`vertcat`
`cumsum`	`introots`	`mrdivide`	`rank`	`subsasgn`	
`diff`	`isempty`	`mtimes`	`rdivide`	`subsref`	

All of these are standard MATLAB commands except `chebpoly`, `introots`, and the `chebfun` constructor itself. In MATLAB, such commands apply to discrete vectors, or sometimes matrices, but in the chebfun system, they perform 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 -1 to 1 (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), and `introots`(f) finds its roots in the interval $[-1, 1]$ (a vector of length equal to the number of roots, if any).

Mathematically, the basis of the system is Chebyshev expansions. Let T_j denote the Chebyshev polynomial of degree j, $T_j(x) = \cos(j \cos^{-1} x)$, 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 [10, 12]

$$f(x) = {\sum_{j=0}^{\infty}}' a_j T_j(x)\,, \qquad a_j = \frac{2}{\pi} \int_{-1}^{1} \frac{f(x) T_j(x)}{\sqrt{1 - x^2}}\, dx\,, \tag{3.2}$$

where the prime indicates that the term with $j = 0$ is multiplied by $1/2$. (These formulas can be derived from the transplantation to $x = \cos\theta$ of the Fourier series for the 2π-periodic even function $f(\cos\theta)$.) The chebfun system could have been built on storing and manipulating coefficients $\{a_j\}$ for such expansions. As it happens, it is built on the equivalent information of samples of f at *Chebyshev points*,

$$x_j = \cos\frac{j\pi}{n}\,, \quad 0 \le j \le n\,; \tag{3.3}$$

we go back and forth to the representation (3.2) where convenient by means of the Fast Fourier Transform (FFT). Each chebfun has a fixed finite n chosen to be

"sufficiently large." Given data $f_j = f(x_j)$ at the Chebyshev points (3.3), other values are determined by Salzer's *barycentric interpolation formula,*

$$f(x) \;=\; \sum_{j=0}^{n} \frac{w_j}{x - x_j} f_j \;\Bigg/ \sum_{j=0}^{n} \frac{w_j}{x - x_j}\;, \tag{3.4}$$

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

$$w_j = (-1)^j \delta_j\,, \qquad \delta_j = \begin{cases} 1/2, & j = 0 \text{ or } j = n\,, \\ 1, & \text{otherwise}\,. \end{cases} \tag{3.5}$$

This method is known to be numerically stable [9].

If f is analytic on $[-1, 1]$, its Chebsyhev coefficients $\{a_j\}$ decrease exponentially. 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 the chebfun system exploits to be a practical computational tool. Suppose a chebfun is to be constructed, for example by the constructor statement

```
f = chebfun('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

$$\begin{aligned} f(x) = {} & 0.88010117148987x - 0.03912670796534x^3 + 0.00049951546042x^5 \\ & - 0.00000300465163x^7 + 0.00000001049850x^9 - 0.00000000002396x^{11} \\ & + 0.00000000000004x^{13}\,. \end{aligned}$$

This is a rather short chebfun; more typically the length might be 50 or 200. For example, `f = chebfun('sin(50*x)')` gives a chebfun of length 143, and `f = chebfun('exp(1./x.^2)')` gives a chebfun of length 198.

Having settled on representing functions by Chebyshev expansions and interpolants, we next face the question of how to implement mathematical operations such as those summarized in the list of commands below (3.1). This is a very interesting matter, and details of the various algorithms used in the chebfun system can be found in [1] and [2]. For example, the integral of `sum` is calculated using the FFT, a process equivalent to Clenshaw–Curtis quadrature [7]; zeros of chebfuns are found by `roots` by a recursive subdivision of the interval combined with eigenvalue computations for Chebyshev companion matrices [5, 6]; and global maxima and minima are located 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 orders in the thousands.

4. Taming the combinatorial explosion

As mentioned earlier, when two 53-bit numbers are multiplied, an exact result would typically require 106 bits, but floating-point arithmetic rounds this to 53. The chebfun system 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. If we execute the commands

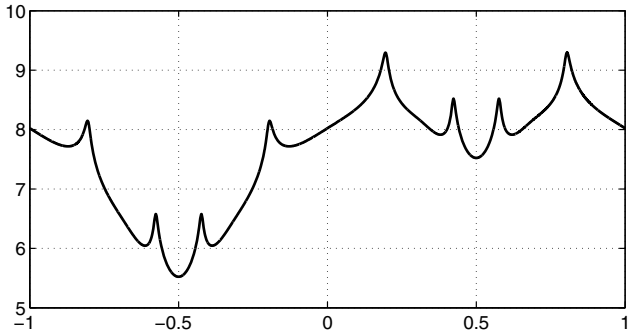
$$f = \sin(X)\,, \quad g = \cos(X)\,, \quad h = f.*g\,,$$

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 may contain the explosion of polynomial degrees. The MATLAB program

```
f = chebfun('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 $\sin(\pi x)$, 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 about half a second on my workstation is a rather complicated chebfun, of degree 3400, which looks like this:



The degree 3400 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 $19 \times 4^{15} = 20{,}401{,}094{,}656$! Thus the complexity has been curtailed drastically, yet with little loss of accuracy. In fact, the command `introots(s-8)` now takes about 8 seconds to compute the twelve values in $[-1, 1]$ at which $s = 8$:

```
-0.99293210741191
-0.81624993429018
-0.79888672972343
-0.20111327027657
-0.18375006570983
-0.00706789258810
 0.34669612041826
 0.40161707348209
 0.44226948963247
 0.55773051036753
 0.59838292651791
 0.65330387958174
```

Applying the 15-step iteration to these numbers in ordinary floating-point arithmetic gives the results

```
8.00000000000002
7.99999999999992
7.99999999999993
7.99999999999993
8.00000000000009
7.99999999999999
8.00000000000001
7.99999999999997
7.99999999999998
7.99999999999998
7.99999999999997
8.00000000000001
```

The fact that these numbers are so close to 8 reveals that the chebfun computation has retained close to machine accuracy throughout.

What is the integral of s? The command `sum(s)` prints `15.26548382582675` in less than one-hundredth of of a second. All of these digits are correct, for the exact answer is $15.2654838258267470094\ldots$. This result was supplied to me by Rob Corless based on 200-digit numerical calculations in Maple and confirmed by Thomas Schmelzer using a different method implemented in Mathematica.

5. Normwise backward stability and condition (3.1)

There is no doubt that the chebfun system can do remarkable things. Computing the integral of the function shown in the figure of the last section is a good example, a difficult calculation carried out successfully in a fraction of a second, and the reader is encouraged to download the system from [3] and explore other examples. One would like to go beyond examples, however, and develop a rigorous and general analysis of the prospects for a system like this. A good starting point would be the normwise backward stability condition (3.1), and in particular, we believe it is productive to focus on two questions:

(I) How close does the chebfun system come to achieving (3.1)?

(II) What are the implications of this condition?

The answer to (I) appears to be that the chebfun system does satisfy (3.1), at least for the basic operations $+, -, \times, /$. This has not been proved formally and it is a project for us in the near future to carry out a proof, making minor modifications in the code as necessary to make this possible. To explain how (3.1) can hold, let us imagine as a slight simplification that each chebfun is represented precisely by a finite Chebyshev series with floating-point coefficients (instead of values at Chebyshev points). The property (3.1) for $+$ and $-$ appears to follow from the corresponding properties for addition and subtraction of floating-point numbers, together with the numerical stability of barycentric interpolation [9]. 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 clear that the current code contains safeguards enough to give a guarantee of (3.1), but if not, we believe this will be achievable with small modifications.

It will also be important to consider analogues of (3.1) for other chebfun operations besides $+, -, \times, /$. These will have to be addressed on a case-by-case basis, but it appears that in most cases such analogues will hold.

This brings us to (II), the question of the implications of (3.1). The easier part of the answer, at least for numerical analysts familiar with backward error analysis, is to understand exactly what the property (3.1) 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. Returning to the problem of sign anomalies discussed at the end of §2, for example, we may note that if two chebfuns f and g give $(f - g)(x) < 0$ at a point x, then from (3.1) 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 throughout the theory of real computation [17]. It is well known that the problem of determining the sign of a difference of real numbers with guaranteed accuracy poses difficulties. However, the chebfun system makes no claim to overcome these difficulties: the normwise condition (3.1) promises less.

Does it promise enough to be useful? What strings of computations in a system satisfying (3.1) 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 (3.1) 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. The aim of the field

known as Exact Geometric Computation is to delineate problems that face such challenges and to find ways to overcome them [16]. On the other hand, generations of numerical analysts have found that such difficulties are by no means universal, that the backward stability condition (2.2) for floating-point arithmetic is sufficient to ensure success for many scientific computations. In the future our aim will be to determine how far this conclusion carries over to condition (3.1) for chebfuns.

6. Discussion

The chebfun class is a powerful system for dealing with smooth functions on $[-1, 1]$, and Zachary Battles' successor Ricardo Pachón is in the process of extending it to more realistic situations such as piecewise-continuous functions on arbitrary intervals and functions with poles. With further research and program development we hope soon to prove that the system does indeed live up to the model (3.1), as discussed in §5. The deeper point of this brief article, however, is to put forward a vision that is not tied specifically to Chebyshev expansions or to other details of the chebfun system. The vision is that a good deal of what is normally regarded as "symbolic" computing can be done numerically, with a potentially vast gain in computer time and memory. And what does "numerically" ultimately mean? It means pruning an algebraic representation at every step rather than just once at the end of all the steps.

Acknowledgements

I am grateful to Zachary Battles, Richard Brent, Ricardo Pachón, and Chee Yap for their advice, and to Rob Corless and Thomas Schmelzer for their Maple and Mathematica wizardry in the example of §4.

References

[1] Z. Battles, *Numerical linear algebra for continuous functions*, DPhil thesis, Oxford University Computing Laboratory, 2006.

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

[3] Z. Battles and L. N. Trefethen, chebfun software, available at `http://www.comlab.ox.ac.uk/chebfun/`.

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

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

[6] D. Day and L. Romero, *Roots of polynomials expressed in terms of orthogonal polynomials*, SIAM J. Numer. Anal. 43 (2005), 1969–1987.

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

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

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

[10] J. C. Mason and D. C. Handscomb, *Chebyshev Polynomials*, Chapman and Hall/ CRC, 2003.

[11] M. L. Overton, *Numerical computing with IEEE floating point arithmetic*, SIAM, 2001.

[12] T. J. Rivlin, *The Chebyshev polynomials: from approximation theory to algebra and number theory,* 2nd ed., Wiley, 1990.

[13] 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.

[14] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*, Prentice-Hall, 1963.

[15] J. H. Wilkinson, *The Algebraic Eigenvalue Problem,* Clarendon Press, 1965.

[16] C. K. Yap, On guaranteed accuracy computation, Chapter 12 in F. Chen and D. Wang (eds.), *Geometric Computation*, World Scientific, 2004, pp. 322–373.

[17] C. K. Yap, Theory of real computation according to EGC, In: P. Hertling, C. H. Hoffmann, W. Luther, and N. Revol (eds.), *Reliable Implementation of Real Number Algorithms: Theory and Practice*, Lecture Notes in Computer Science, Springer, 2007 (to appear).

Lloyd N. Trefethen
Oxford University Computing Laboratory
Wolfson Bldg., Parks Road
Oxford OX1 3QD
United Kingdom
e-mail: LNT@comlab.ox.ac.uk

Received: November 20, 2006.
Revised: January 17, 2007.
Accepted: June 5, 2007.