Notes on Chebyshev Series and Computer Algebra

Richard J. Fateman
Computer Science Division
Electrical Engineering and Computer Sciences
University of California at Berkeley

September 26, 2011

Abstract

It is the premise of computer algebra systems (CAS) that we can gain insight by dealing with mathematical functions by manipulating their formulas as data structures. In some circumstances this has decided advantages over using formulas solely for evaluation and then manipulating or perhaps plotting the resulting numbers, as is done by FORTRAN and other typical conventional numerical programming languages. The usual CAS typically represents formulas by (what amounts to) trees, although it is easy to represent a polynomial as an array of coefficients, even in a numerical language. If an expression is not so simple as a polynomial, it is plausible nevertheless to represent the expression as an array of floating-point numbers in which is encoded a polynomial which approximates that function, or perhaps by storing the coefficients of an expansion in terms of some set of other basis functions. Another possibility is to store a set of values of the function at specified points by which one can, by interpolation, approximate values in-between those points. In either case, one can compute further by combining such arrays, and by translating from one representation to the other, take advantage of particular properties of each. It is possible to differentiate, integrate, and perform other operations by composition, as well as arithmetic, on these arrays. This kind of symbolic-numeric approximate algebraic computing has some of the flavor of approximation by truncated Taylor or Laurent series, a popular feature in CAS.

By taking advantage of special features of a particular basis set, namely Chebyshev polynomials, a number of facilities may be programmed especially neatly.

1 Inspiration

These notes were prompted by viewing a paper on the The Chebfun Project [2] headed by Nick Trefethen at Oxford University. Partly to see what could be done with these ideas in the context of a computer algebra system, and partly to have fun understanding the implementation decisions needed for Chebfun itself, we implemented some of the basic features in Maxima (open-source Macsyma). Chebfun itself has moved on to more elaborate features which we have not included. Should the initial work here engender interest, such extensions may be added.

2 The Chebfun Idea

The basic idea is to use an encoding of a mathematical function by a series of coefficients corresponding to an expansion in orthonormal polynomials (Chebyshev series or for brevity, CS), and compute with that sequence of coefficients as an approximate model for the function. The concept is probably comfortable to anyone who has studied Chebyshev approximations, and this brief paper does not attempt to detail why this is an especially useful choice. We point out (to persons familiar with computer algebra systems) that a CAS typically supports an approximation expansion, but in another basis (the power basis) through truncated power series. Chebyshev series are advantageous in several significant ways some of which are capitalized upon by the Chebfun Project. This project is interesting in that it expands upon numerous ideas from more conventional "numerical" computation and provides implementations of appropriate generalizations in this new space of "functional" computation. It also provides a framework for representing and performing Chebyshev series manipulations from classical numerical analysis. Here are some of the Chebfun innovations.

(a) Numerically approximating the coefficients by double-float numbers using fast and accurate methods related to the Fast Fourier Transform (FFT).

(b) Applying some cleverness in truncating (probably) useless low-order terms so as to keep the series compact, even though operation upon operation would ordinarily seem to make the series grow in length.

(c) Building a nice user interface in a popular interactive computer system (in particular, Matlab),

(d) Demonstrating neat applications including solution of algebraic and differential equations, quadrature and rootfinding.

In some sense, the process is analogous to floating-point arithmetic (FPA). Just as we can make great use of FPA to approximate real numbers, but with a computationally more convenient finite-bit-length approximation, we can make use of CS to approximate real *functions* of a single variable on a finite interval by the use of a collection of floating-point coefficients. Just as we can accumulate large libraries of useful functions based on FPA, we can hope to accumulate a set of tools for a variety of CS applications.

Our own interest in Chebyshev series goes back at least to 1989 [1], but we neglected all of (a)-(d), and after writing the paper, set the program aside. That paper does, nevertheless explain some cute methods for computing Chebyshev series from different forms, using Macsyma or similar programs.

3 Enter, Computer Algebra

It seemed time to re-examine what could be done using the insights of the Chebfun Project and the tools in a computer algebra system.

3.1 Advantages for a CAS doing Chebfun-like computations

There are three obvious advantages for a CAS:

1. A CAS has a built-in symbolic representation for any algebraic expression (expressions including polynomials and tables or lists of floating-point numbers) and so they can be passed around, converted to Chebyshev form (tables) and converted back, evaluated and manipulated. In some cases it is important that expressions be evaluated to higher-than-normal precision. Arbitrary-precision numerical operations are generally supported by a CAS, as are tables of high-precision numbers.

2. It is easy to provide alternative expansion tools for finding Chebyshev coefficients, including special tricks (e.g. for polynomial denominators), a more general Taylor series method, exact integrals using orthogonality properties, and evaluation at specified points. The last of these is the basic numerical technique used in most situations.

3. Finally, the availability of arbitrary-precision software floats, so more accuracy is easily specified. This provides another avenue for exploration of more precise representation, more bits rather than (or in addition to) more terms. Arbitrary-precision arithmetic generally also banishes numerical exponent overflow. The tradeoff is for substantially slower computing times. Many routines, not just basic arithmetic, may be available in arbitrary precision. For instance, the underlying technique of computing the Discrete Cosine Transforms (DCT) maps to a FFT. In our programs, a version of the FFT has, conveniently, been implemented to use arbitrary precision arithmetic, if so requested.

3.2 Specialization or Generalization

Starting with a CAS, there is also a rather different perspective from the Chebfun Project. Since the original Chebfun is implemented using Matlab (and does not use its symbolic toolkit), the notion of a "function represented by a table which is its expansion in the Chebyshev basis" is an extension of the realm of objects typically available to the Matlab user. For an example of a new operation in the extension in this new kind of object, one can compute a derivative of a Chebfun.

Compare this to the CAS version, where a function can already be represented symbolically, perhaps in several ways. Computing a derivative is already provided. The approximation into a set of Chebyshev coefficients (or the equivalent) is, strictly speaking, a downgrading of the information: it is an approximation. Furthermore, it works for functions of only a single real variable, and it works on a pre-set interval, initially [-1,1]. While a CAS can combine its ordinary functions with abandon, one can add Chebfuns only if they agree various particulars.

The major positive side of using Chebfuns in a CAS is that the full formula for an arbitrary function in a CAS can be costly to manipulate and evaluate, even if its graph on [-1,1] is be rather smooth. In that case, the Chebyshev approximation will have only a few terms, and by using a Chebfun version we can quickly (say) plot the function, find zeros, and manipulate it in various ways to rather high accuracy. This does not guarantee that we will always win with Chebfuns since we may encounter nasty functions (step functions or perhaps something like $x \sin(1/x)$) whose Chebyshev expansion will require many terms and/or will not be very accurate. Functions which wiggle frantically are not going to be easy to represent with low-degree polynomials. Breaking such functions up into piecewise approximations can overcome some problems of this nature.

3.3 Generic arithmetic or not?

Since Chebyshev series are a new kind of mathematical object in a CAS, there are programming issues that must be addressed. In this case there are a few roughly comparable choices for representation, any of which would likely be feasible. but it remains to be decided what should be implemented as allowable operations. Certainly anything that the Matlab system can do on Chebfuns is fair game.

At first glance, we can support a number of operations: For F,G each a CS of compatible variable and length, we can more-or-less automatically compute F+G, F*G, F(0.5), F(G), i.e, composition, as well as mixed forms, like F+1.

At second glance, we must figure out what, if anything, to do with other expressions that could easily be typed in, such as F+x+y. This is probably an error.

Frankly, the CS context seems inappropriate for absorbing so much of the rest of a CAS: a numerical CS is a univariate polynomial, and one limited[1] in precision, degree, and domain.

Let's re-examine a notion dropped casually into the discussion a few paragraphs back and consider operator calculus such as composition. If F and G are each considered as functions mapping a domain [-1,1] to a range [-1,1] perhaps, it is useful to be able to compose them as F(G). In the same sense that one can combine cos and cube-root one can combine Chebyshev series. They can be combined to form a new function[2] or operator `lambda((x),cos(x^(1/3))`, or a different operator, `lambda((x),cos(x)*(x^(1/3))`, or yet a different operator, `lambda((x),cos(x)^(1/3))`.

Maxima does not currently have built-in tools to combine, in an operator sense, such functions defined as `lambda()` expressions, though of course such facilities might be programmed in almost any symbolic system.

To restate this, if K is a CS, or alternatively an abstract function, `lambda((x)...)`, K represents a function of a single variable on a finite range. It may be sensible to "do symbolic arithmetic" and construct expressions with parts that look like $K(x)+K(y)$, or perhaps $K(x)+K(y)$ with $-1 \leq x \leq 1$ and $-1 \leq y \leq 1$.

[1]Except for the case of a CS representation of an exact polynomials —when rational arithmetic may be used for computing coefficients—the approximation generally suffers from truncation error (finite number of terms) and roundoff error (in the coefficient values).

[2]We use the lambda notation from logic (or Lisp) to denote the variable(s) in an expression to be "bound" when a functional form is applied.

It does not make sense to add together $K + x$, except as a kind of mathematical pun where x is shorthand for the function `lambda((x),x)`.

Similarly we can arithmetically combine K with scalars, with 3 being a shorthand for `lambda((x) 3)`. then: $3 * K$ or $3 + K$ are new operators.

Drawing the line between a package that does such arithmetic on CS objects that makes sense, and avoids contradictory or nonsensical operations can be ticklish.

4 Sidelight: Expressions or functions?

This is a topic of concern in the world of CAS building. What makes people comfortable?

For example, we could talk about the function cos, and thus make sense of `integral(cos,{-1,1})`. Alternatively we could introduce a dummy parameter and render the same computation on expressions: `integral(cos(t),{t,-1,1})`. It may seem redundant to mention `t` twice, but it is not. This becomes more revealing if the function has other parameters. For example, consider `f(x):=exp(x^2/t)`. We can still talk about `integral(f,{-1,1})`. But given an expression, are we dealing with the integral with respect to `x` or `t` or for that matter, yet some other variable not even mentioned in the integrand? It seems that we need the extra `x` in `integral(exp(x^2/t), {x,-1,1})`. (Or perhaps `t`.)

Sometimes it is convenient to "functionize" an expression, as we have done with the definition of f, above, but without naming it. Such anonymous functions have a long history in logic (and in the Lisp programming language) and are known there as lambda expressions. Thus if you wish to refer to a function `f(x):=x*(x+1)` without using up a name like "f", Lisp, in its parenthesized prefix notation, would let you do this: `(lambda(x)(* x (+ x 1)))`. This can be said in Matlab as `@(x)(x*(x+1))`. It can be said in Mathematica as `Function[{x},x*(x+1)]`. Also in Mathematica's syntactic shorthand, `#(#+1)&`. Finally, in Maxima the format is `lambda([x],x*(x+1))`.

What does this have to do with CS? Firstly, should a CS have within itself some variable name? [no] Or are we encoding a lambda-expression? [yes] If we compute a set of coefficents q based on, say, an approximation to cosine, it seems natural to `cheby_apply(q,1.0)` and get 0.5403, about the same as cos(1.0). In the example section we define syntax so that `q@1.0` works. We do not (currently, at least) allow `q(1.0)` or for that matter, `q(z)` where `z` is just a symbol, but `q@z` is fine, and we get some polynomial in `z`.

It also seems plausible to do a conversion to a CS this way: `cheb(cos)` `cheb(cos,n)` where n is some integer related to the number of terms desired. It is superfluous to indicate a "variable" in the argument to `cheb` as in `cheb(cos(x))` since the result simply has one anonymous variable and so `cheb(cos(x))` would be quite indistinguishable from `cheb(cos(z))`

The key to Chebfun efficiency is guessing how many terms are sufficient. For a nasty function like `f(x):=sin(1/x)`, there is no number of terms that will give a really good fit, since the computation of the series coefficients is based on sampling of f. The precise values of f near zero are based on the happenstance of evaluation of floating-point numbers, and so repeatedly doubling the number of sample points does not converge to a smooth fit.

For nicer functions[3] there is a kind of limit to the frequency of the signal, so to speak. $\cos(20x)$ will need more terms than $\cos(10x)$, but computing the coefficients beyond the point when even and odd coefficients are both below 1.0e-15 is not worthwhile. For a step function, the really effective way of dealing with the discontinuity is to break the function into two sections, as is done in a later version of Chebfun.

A function like $x^{100} + 1$ seems to require terms up to the 100th Chebyshev coefficient, but not much (actually, none) beyond that. Yet consider that this function is hardly different from g(x):=1 on most of [-1,1], and so can be approximated to some degree of satisfaction by a polynomial of lower degree. This aspect of Chebshev approximation, and so much else about this topic has been well studied in the classical literature.

[3]There are much more specific characterizations than "nicer". We are avoiding the technical discussion necessary to make this notion definite.

5 What to store?

The Chebfun Project design is set to store an array of values of the CS at some set of Gauss-Lobatto points (GLP), certain points in the interval [-1,1]. In particular, for n terms, $\cos(k\pi/(n-1))$ for $0 \le k \le n-1$. (There are other choices possible, including points half-way between these chosen above). These can be converted by a Discrete Cosine Transform (so-called DCT-I) into the coefficients in a Chebyshev series; these are convertible back by an inverse DCT, which is the same program with the result multiplied by a constant[4].

We have one set of programs manipulates Chebshev series based on a data representation that stores the Chebyshev coefficients (CC). Another set of programs uses evaluation points. These representations can be easily interconverted by the DCT and its inverse. Our primary reason for using (or at least periodically computing) CCs is that we need them to make a determination that after some certain number of terms, the remaining terms (probably) make a negligible contribution to the value of the function. This is not obvious from the function values at GLP, and so computations that logically suggest trimming terms depend on computing the CCs.

What operations can we do with the GLP representation? Lots. For example, we can add or multiply two CS in the GLP representation in linear time: just add or multiply pointwise the two arrays[5].

There may be other material stored with the CS, such as "what computations got us to the function" and perhaps both the GLP and CC could be stored.

We refer the reader elsewhere [2] for a discussion of the many kinds of computations that can be considered, and the nature of functions that can be appropriately modeled by a Chebyshev expansion.

Interestingly, there is a history (since the mid-1960s) in the CAS community of performing polynomial computations more rapidly by taking images of polynomials formed by modular and evaluation homomorphisms, performing operations on the images, and then re-assembling the results back to polynomials. This lead to vastly more efficient computation of polynomial greatest-common-divisor and related algorithms. Approximate evaluation of more general functions except perhaps in the context of Taylor series, was not an area of activity.

The Chebfun web site discusses further Matlab programs which include facilities for functions being translated or scaled away from the interval [-1,1], or pasted together piecewise, so that discontinuous functions may be split into intervals that are tractable, but manipulated collectively. (We have written programs to provide this basic facility too. However, we have not yet written an efficient splitting algorithm – one which would determine breakoints for these alternative representations. This involves detecting "edges" in a function.)

6 Implementation Notes

1. Using an FFT[6] for DCT indeed speeds things up considerably over the naive summation formula. Writing a specific fast DCT program[7] gives us a factor of about 8 in our experiments over the mapping to an FFT. Given that we are probably only interested in DCTs of sizes 2^n for $2 \le n \le 9$ we can use a well-known optimization technique: that is to generate specific code for each of these fixed-size DCTs, essentially unrolling all loops. For example, we hand-coded an 8-point DCT and found it about 5 times faster than the more general DCT on 8 points. We "automatically generated" a 32-point DCT by instrumenting our Lisp DCT to emit code instead of performing the operations and found the code ran about 4.5 times faster. By doing more operations essentially in registers rather than arrays, the speed can be improved yet more. The number of tricks that can be played here, trying to

[4] Any of the DCT versions and their inverses can conveniently be expressed as a row-by-matrix multiplication; the fast DCT can be thought of as a way to rapidly multiply that row by a specially structured matrix.

[5] Actually, I lied. They have to be represented at the same GLP, and if they are not, then there are at least 2 plausible techniques, namely interpolation and using a DCT, changing the number of terms, and then an inverse.

[6] If we use FFT code, the DCT of size n can be mapped in linear time to an FFT of size $4n$ where half the entries are zero and half of the remaining entries are the same as the original, but in reverse order, with some normalization multiplier.

[7] ACM Algorithm 749, translated from FORTRAN to Lisp

benefit from particular memory and arithmetic architectures is almost unlimited. There are so many implementations of the FFT it is reasonably to simply rely on someone else doing a bang-up job of it. The Chebfun people presumably use the DCT in Matlab, which we believe is some version from FFTW. Readers interested in pursuing this aspect of this paper may consider yet alternative DCTs[8]. We have written an interface to FFTW available from Lisp, which has a feature that any length FFT, (or for our purposes, a DCT), not just power-of-two or product-of-primes size, can be computed.

Some of the versions (of the FFT), especially in those written in a kind of generic-arithmetic Lisp, have the neat option of also allowing arbitrary precision, or other extended arithmetic.

Speed is not everything: for small cases not a power of 2, I don't even feel bad if the program uses $O(n^2)$ rather than $O(n \log n)$ operations for a length-n FFT. The slow cosine transform can, at least for short cases, produce exact answers in terms of symbolic (algebraic) roots and rational numbers.

2. I am overly simplifying the calculation heuristics for how many terms are needed to represent a function. My program looks at several of the trailing coefficients. If they are relatively small compared to the largest of the other coefficients, it truncates the series there. It is important to look at two or more coefficients since common functions may be even or odd, and therefore have half zero coefficients.

3. I have not dealt at all with NaN or Infinities, either those that might occur in the input data from evaluation of a function, or from calculations of the transform.

4. There are new extensions (cf Chebfun Project software in Matlab) that are possible and some of these may be enhanced by symbolic capabilities. There are others that seem to make sense only as functions implemented as arrays of double-floats, and CAS features are irrelevant. A more productive use of our time or potential users of such facilities would seem to suggest that users get a copy of Matlab and Chebfun! (This is discounting the educational or fun aspects of revisiting implementations).

5. Many functions that we supply could be done in a cleverer fashion. This might reduce roundoff accumulation, or speed up the production of coefficients compared to evaluation/DCT. For example, conversion of rational functions can be done by partial fraction expansion, rootfinding, and many special cases. There are different methods of implementation in the CC or GLP representations of basic operations, including even multiplication and addition.

6. To some extent our programs are slowed down by the availability of full generic (arbitrary precision, rational, etc.) arithmetic facilities. Thus they may be slower than necessary for some users. As compensation, with a few declarations we can compile them specifically for double-precision floats to be faster (while also less general).

7. On the other hand, for some programs we assumed that they made sense only in double-precision and just dropped in a double-float value for π, especially in the fastDCT. In these cases a user could complain of prematurely approximating calculations: a lack of exactness or generality. (Maxima knows that $\cos(3\pi/4)$ is $-\frac{1}{\sqrt{2}}$, and can continue working with expressions of that kind, but not if a numerical value for π is first inserted).

8. In converting to a Chebyshev form, it is important that a function be evaluated accurately at the GL points. If it is not, it will appear to be a slightly different function, perhaps one whose Chebyshev coefficient series is much longer. In our experiments, we tried converting $(4x-5)^2$, a simple-enough function, to a Chebyshev series matching our usual criterion, having a minimal number of terms. To our dismay we got a series with hundreds of terms, all but the first three of size about 10^{-15}. Our program to cut off terms was fooled into thinking these terms were significant. Converting the original expression to the equivalent, "Horner" form $x\ (16\,x-40)+25$ leads to a more stable evaluation, and conversion to Chebyshev form returns three terms: [66, -40, 8][9] Some of the programs now insert this "Horner" conversion, routinely.

[8]Google for FFTPACK5, FFTW, djbfft, MPFR, Numerical Recipes

[9]To be more precise, chebseries $(66.0, -40.0, 7.999999999999999)$.

7 Neat symbolic manipulations

While some functions are written to produce exact Chebyshev coefficients when given polynomial inputs, and may work even for polynomials with symbolic entries, these are probably only of occasional pedagogical use. They certainly will not survive a round-trip through a fast FFT-based DCT. Yet not everything needs to be done via DCT or fast DCT, and sometimes it can be illuminating to deal with symbolic entries. Consider an expression $a * T_5(x)$ where the x is implicit. This is a Chebyshev series `[0,0,0,0,0,`a`]`. Its derivative is the surprisingly regular expression `[10`a`,0,10`a`,0,10`a`]`. The square of `[0,0,0,`b`]` is `[`b^2`,0,0,0,0,0,`$b^2/2$`]`. While not usually advisable, even the discrete cosine transform can be symbolically and exactly computed:

$$\mathrm{DCT}([a,b,c]) \;=\; \left[\frac{c+b+a}{\sqrt{3}}, -\frac{c-a}{2}, \frac{c-2\,b+a}{2\,\sqrt{3}}\right].$$

The explicit expression for a DCT of [a,b,c,d] is clumsier, and probably not worth displaying *per se* but with a little juggling, a *program* for an arbitrary 4-term transform can be produced. One such program looks like this:

```
block([t1,t2,t3],
      t1 : pi/8,
      t2 : cos(t1),
      t3 : sin(t1),
      [(d+c+b+a)/2,          -(t2*d+t3*c-t3*b-t2*a)/2,
        sqrt(2)*(d-c-b+a)/4, -(t3*d-t2*c+t2*b-t3*a)/2])
```

This little program is not optimal with respect to multiplications. Consider the two subexpressions R=t2*d+t3*c and S=t3*d+t2*c. This uses 4 multiplications and 2 additions. We can do it using only 3 multiplications and 5 additions. (z:=d*(t2+t3), R:=(c-d)*t3+z, S:=-(c+d)*t2+z). This trick can be used twice in the program above.

8 Status

The programs are currently divided into a few files; while some were initially written in Lisp, most programs were rewritten in the Maxima language so as to be more general, allowing for different (non-numeric) argument types. These can be compiled into Lisp and thence machine instructions. However, writing in Lisp is sometimes a good deal more direct, especially if we know that given the particular context, the only sensible data types are numeric, and among those numeric types, the only one that will be used is (real) machine double-float.

Of course the temptation exists to link the Maxima or Lisp system to Matlab and just use Chebfun.

The file mincheb.mac is the minimum setup, for converting to and from chebyshev form, and trimming coefficients. it is 2 pages long, and has some useful utility functions.

The file dct.mac provides the discrete cosine transform and inverse, both "slow" and "fast" (using the FFT). It is one page, although an additional half-page provides alternative somewhat faster versions of Maxima code to set up the call to the FFT. The versions using the FFT are obsoleted by the next file, though.

The file algo749.lisp is a lisp-language version of the discrete cosine transform from Collected Algorithms of the ACM, Algorithm 749. This was created by first automatically translating the original FORTRAN into Lisp with the program f2cl. We then edited the result so as to use idiomatic Lisp, and in the process, vastly shortened the f2cl output. In simple tests on 16 and 64-point transforms, it appears to be about 10 times faster than the previous version in dct.mac which is first obligated to set up a rearrangement of the terms and then call the built-in FFT on a larger array.

The file cheb-arith.mac shows how to do addition, multiplication of CS, and composition of a CS with another, as well as application of a (non-CS) function to a CS to produce a CS. About 1.5 pages.

The file cheb-integ.mac provides (indefinite) integration and differentiation. 40 lines.

The file cheb-bary.mac provides barycentric interpolation and also some arithmetic for operating on lists of evaluation points rather than chebyshev coefficients. This really doesn't fit into the rest of the system – in fact it is using a different cosine transform based on another set of evaluation points (off by 1/2 from the ones used elsewhere).

The file cheb-extend.mac provides tools for creating and manipulating generalized Chebyshev series on a finite interval [a,b] other than [1,1], as well as piecewise representations. For example, the absolute value function on [-3,3] is `gencheby(-3,0.0, chebseries(3.0,-1.5), 0.0,3, chebseries(3.0,1.5))`. This is simply suggesting how we might incorporate in our model the piecewise advanced version of Chebfun. The file cheb-misc.mac contains a few functions that are useful if the user prefers not to write or see "anonymous functions" as lambda expressions, and just uses `cheby(sin(x)+x)` rather than `tocheby(lambda([x],sin(x)+x))`.

All the files may grow or shrink, and additional files may be started as appropriate.

Many of the functions work as expected. Very little timing data has been collected.

There are several arbitrary limits set that might be altered. In particular: the acceptable relative error for truncation of a CS is `relerr` set to 1.0e-13. The largest number of coefficients is arbitrarily set at 512. The number of trailing items tested to see if an approximation is good enough is 2.

9 Examples

```
s: cheb(sin(x)); -->

chebseries(7.8062556418956319*10^-17,0.88010117148987,-1.9009829523078689*10^-17,
-0.039126707965337,2.4590084002406301*10^-17,4.9951546042242053*10^-4,
2.6187632824244736*10^-18,-3.0046516348858354*10^-6,-1.349298206681369*10^-17,
1.0498500309169361*10^-8,1.9452934283922738*10^-17,-2.3960209521740189*10^-11)

which is easier to eye-ball if we replace all ``small'' values by 0
[stz means small-to-zero]:

stz(%);

chebseries(0,0.88010117148987,0,-0.039126707965337,0,4.9951546042242053*10^-4,
0,-3.0046516348858354*10^-6,0,1.0498500309169361*10^-8,0,0)

Here we define a notation for application of a series to a point. (We
could include this automatically in the Chebyshev package, but we
hesitate to use up a symbol that might be used for other operators.)
In Maxima, the point could be symbolic.  It is preferable to convert
``contagiously'' to a rational form so all the arithmetic consequences
of the conversion will be simplified -- so we use  rat(z).

infix("@"); -->
"@"(a,b):=fromcheby(a,b)

cheb(sin(x))@0.345 -->
0.33819667724783

which compares to
sin(0.345) -->
0.33819667724779
```

For a symbolic polynomial (power-basis) version of the series, try this:

```
stz(cheb(sin(x)@rat(z)) -->
2.4535254550261953*10^-8*z^11+2.7550880291605769*10^-6*z^9
-1.9841231276078824*10^-4*z^7+
0.0083333332209776*z^5-0.16666666665263*z^3+0.9999999999995*z
```

Forms for exact series for powers of a variable are stored in an array (the array is expanded as needed); This shows that exact coefficients are possible too. Here is the series for x^10 {or z^10, etc}.

```
onepowcs[10] -->

chebseries(63/128,0,105/256,0,15/64,0,45/512,0,5/256,0,1/512)
```

To illustrate the difference between the simplified rat() and ‘‘just plugged in’’ forms:

```
onepowcs[4]@z; -->
z*(2*z*(z^2/2+3/8)-z/4)-z^2/2
```

compare to

```
onepowcs[4]@rat(z); -->
z^4
```

We can integrate a series and then apply it.

```
stz(chebint(onepowcs[4])@rat(z)); -->
0.2*z^5
```

We can compose a series with another, using the sin(x) series s from above:

```
cc(s,s)@0.3 -->
 0.29123754560999
```

compares to

```
sin(sin(0.3) -->
0.29123754560994
```

The number of terms should expand as necessary (we hope).
e.g.
let
 five: onepowcs[5]

length(five) is 6.
length(cc(five,five)) is 26.

```
cc(five,five)@2.0 -->
3.3554432028975487*10^7
```

compared to the exact answer

```
3.355443200*10^7
```

It is also possible to compose an arbitrary function of one variable which a Chebyshev series, though whether this is meaningful may depend on having functions with appropriate domains.

Illustrating chebplus chebdiff for addition and differentiation, respectively, we can show that sin(x)+ d/dx(cos(x)) is close to zero:

```
hh:  stz(chebplus(cheb(sin(x)),chebdiff(cheb(cos(x))))); -->
chebseries(0,0,0,0,0,0,0,0,0,0,0,0)
```

We would ordinarily drop the terms less than some relative error, relerr. The default relerr is set to 1.0e-13. This is usually done by trimcs:

```
trimcs(hh,relerr); -->
 chebseries(0.0)
```

Or consider

```
chebplus( chebtimes(cheb(sin(x)),cheb(sin(x))),
          chebtimes(cheb(cos(x)),cheb(cos(x)))); -->
chebseries(2.0).
```

Here's another way to tell that diff(cos(x),x)=-sin(x), probably. Observe that

```
chebnorm(chebplus(cheb(sin(x)),chebdiff(cheb(cos(x))))); -->
8.8714401421864493*10^-27
```

This norm is a sum of squares of the coefficients.

Other operations on Chebyshev series include chebtimes, chebsubtr, chebmin, chebpower, for multiplication, subtraction, negation, and powers. Since chebpower works for real number exponents, a/b can be computed by chebtimes(a,chebpower(b,-1)).

It is also possible to embed chebseries inside other expressions and evaluate them this way:

```
E: q(  onepowcs[10 + r(onepowcs[5])  ) -->
q(r(chebseries(0,5/8,0,5/16,0,1/16))+
  chebseries(63/128,0,105/256,0,15/64,0,45/512,0,5/256,0,1/512))

Allfromcheby(E, x) -->  q(r(x^5)+x^10)
```

10 Conclusions

Considering the pleasant collection of ideas and programs constituting the Chebfun suite of programs, we see that mimicking them in a computer algebra system can provide some additional facilities. While the core of the programs we wrote can use (for example) highly-tuned library programs for DCT (discrete cosine transform), we can also introduce and maintain, in some aspects, symbolic parameters, and a fairly natural high-level interface.

References

[1] T. Einwohner and R. Fateman, "A MACSYMA Package for the Generation and Manipulation of Chebyshev Series (Extended Abstract)" Proc. 1989 ISSAC, ACM 0-89791-325-6/89/0007/0180 p180-185.

[2] Chebfun Project, `http://www2.maths.ox.ac.uk/chebfun` which contains links to new and old literature.