Notes on Chebyshev Series and Computer Algebra

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

January 30, 2014

Abstract

A typical computer algebra system (CAS) represents formulas by (what amounts to) data structures known as expression trees. By approximating the function represented by a formula in one variable as a univariate polynomial of some maximum degree d presumed to be "near enough" on some interval, the function can be represented as an array of $d+1$ floating-point coefficients. An alternative representation is as a set of values of the function at $d+1$ special points (interpolation values). In either case, one can compute in this realm of "functions" further by combining such arrays, and one can convert between these two representations surprisingly efficiently.

By taking advantage of this transformation and other special properties of Chebyshev series, a number of facilities may be programmed especially neatly. These have been nicely pursued by the Chebfun project at Oxford University [4]. Combining these ideas and reimplementing with computer algebra systems, we gain yet additional facilities.

1 Introduction

It is the premise of computer algebra systems (CAS) that one can gain insight by dealing with mathematical functions by manipulating their formulas as data structures. In some circumstances this has decided advantages over the traditional scientific computing model in which one sees formulas solely as source text in programming languages, where they can only be evaluated numerically.

The usual CAS represents formulas by (what amounts to) expression trees, although it is easy enough to represent the simple case of a univariate polynomial of degree d as an array of $d+1$ coefficients any traditional numerical language.

If an expression is not so simple as an explicit polynomial it is plausible nevertheless to represent it via an *approximating polynomial*. This can be done by storing coefficients of the power or monomial basis: $\{1, x, x^2 ...\}$ or using some other set of basis functions. The errors in the approximation appear from inaccuracies in the coefficients and/or dropping terms of too-high a degree and/or arithmetic round-off in evaluation.

Another possibility for representation of an expression is to store a set of (perhaps approximate) values of the function at specified (perhaps approximately) points. From this one could reduce the situation to the previous problem since one can use this information to determine polynomial coefficients. For our discussion here, this often turns out to be unnecessary, and furthermore, not even a good idea. The monomial basis may be an uncomfortable representation for a function, since it may require large coefficients or high precision even for relatively simple functions.

We can use the evaluation points more or less directly for interpolation of values of the function at intermediate points, or use some other basis. Since a typical concrete representation may be of arrays of numbers, almost any programming language is a plausible host for manipulating these representations. By translating from one representation (or basis) to the another, we can take advantage of particular properties that are most easily observed in 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 retains some of the flavor of approximation by truncated Taylor or Laurent series, a popular feature in CAS, but extended to another, non-monomial basis.

By taking advantage of special features of a particular basis set, namely Chebyshev polynomials, a number of facilities may be programmed especially neatly. We describe some of these features in this paper.

The usual (prior) model of Chebyshev series in CAS provisions has been to use them almost at the terminus of a computation: as the approximation which is then converted almost directly into a floating-point number. Thus the idea of carrying an encoding of a function as a Chebyshev series through a sequence of operations in the midst of a CAS is somewhat novel. Moreover, in a CAS, the option of using exact rational arithmetic or arbitrarily higher-precision floating-point numbers is always at the ready.

2 Inspiration

These notes were prompted by viewing a paper on the Chebfun Project [4] headed by Nick Trefethen at Oxford University.

Partly to have fun understanding the implementation decisions needed for Chebfun itself, and partly to see how these ideas might be extended in the context of a CAS, we implemented some of the basic Chebfun features in Lisp or 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 could be added, especially those dealing more carefully with piecewise approximation. The idea behind Chebfun's "splitting" is that a useful representation of a piecewise continuous function should be one in which the discontinuities are located and then separate chebfun approximations used for each section. The alternative, namely to attempt to mimick a discontinuity by a single high-degree polynomial is, to say the least, unrewarding.

3 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 carry on an extended sequence of computations maintaining that representation via an array 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 apt choice. We point out (to persons familiar with CAS) that a CAS typically already supports a related feature: an approximation expansion, but in another basis (the monomial basis) through truncated power series (in Maxima, this is called Taylor series). Chebyshev series have some key advantages which are exploited by the Chebfun Project [4], whose documentation includes highly-readable introductory and reference material on application and theory of these ideas.

The Chebfun project is philosophically interesting in that it generalizes facilities built to support conventional "numerical" computation in Matlab[3]. It implements appropriate features in a new space of "functional" computation. This is somewhat akin to the leap from numeric to symbolic computing, but not using algebraic trees; rather, it uses Chebyshev series as its function representation.

Here are some of the Chebfun features and innovations.

(a) Starting from some other presentation of a formula, numerically approximating the coefficients of the Chebyshev series by double-float numbers using fast and accurate methods related to the Fast Fourier Transform (FFT).

(b) Applying some cleverness deciding which terms can be neglected and truncated so as to bound the degree of the series. A series that is relatively compact promotes more efficient computation.

(c) Building a nice user interface in a popular interactive computer system (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 that paper, set the related programs aside. Nevertheless we explained some cute methods for computing Chebyshev series from different forms, using Macsyma or similar programs. Some of these reappear in the programs provided with this paper.

4 Enter, Computer Algebra

What can be done using the insights of the Chebfun Project and the tools in a computer algebra system?

4.1 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 trigonometric, exponential and logarithmic components as well as polynomials and tables or lists of floating-point numbers) and so they can be passed around, converted to Chebyshev form (tables) when appropriate and converted back to ordinary-form polynomials, as well as evaluated and manipulated in the Chebfun form. 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 construct and deploy a variety of tools in a CAS to compute the Chebyshev coefficients for an expression. A few lines of code allow us to exploit orthogonality properties, or evaluation at special points. Some of these calculations can be done using exact rational arithmetic, or using (arbitrarily) higher-precision floating-point arithmetic with little or no additional effort by the programmer.

3. Finally, arbitrary-precision software floating-point numbers are smoothly integrated into the system so it is possible to build into applications the option of using higher accuracy at any point in the computation. 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 that one encounters substantially longer 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.

4.2 Specialization or Generalization

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 numerical objects available to the Matlab user. Additionally there are also extensions or new operations on these objects: 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 loss of information: it is an approximation. Furthermore,

the Chebyshev array is restricted to 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 expressions pretty much with abandon, one can meaningful add Chebfuns only if they agree in various particulars: same sole variable, same range, presumably approximating a continuous function.

A positive aspect of implementing Chebfuns in a CAS is that all the ideas promoted in the (Matlab-based) Chebfun Project can be mimicked in the CAS, and yet one can revert to a more-general representation as a fall-back. This can be done alternatively by linking a CAS to Matlab, but perhaps with some inconveniences[1].

In the case where the graph of a (perhaps quite complicated) function is rather smooth on [-1,1], the Chebyshev approximation may have only a few terms. By using such a Chebfun version we can quickly (say) plot the function, find zeros, and manipulate it in various ways to rather high accuracy. There is no guarantee that we will *always* win with Chebfuns since we may encounter nasty functions (example: step functions or perhaps something like $x\sin(1/x)$) whose Chebyshev expansion will require many terms and/or will ultimately not be very accurate somewhere. Functions which wiggle frantically cannot be accurately represented with low-degree polynomials. A recipe for dealing with such issues exists: Breaking such functions up into piecewise approximations can overcome some problems of this nature. For example, a trick to ease zero-finding: if the Chebfun is of high degree, split the interval into pieces and approximate each piece by a lower degree polynomial.

4.3 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, and how to allow operations. Certainly anything that the Matlab system can do on Chebfuns is fair game. Using a tested design from the Chebfun team simplifies the task considerably! One can also follow their design extension for piecewise functions.

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, with restrictions on the range and domain of G, 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 either an error, or an expression which can be encapsulated and carried around symbolically but not further simplified until (for example) values as constants or CS are given for x and y.

Frankly, the CS context is a very limited slice of a CAS, limited but useful: a CS is a univariate polynomial, and one further limited[2] 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] then it may be 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[3] 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

[1] assuming one has a Matlab license, for example.

[2] 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).

[3] We use the lambda n otation from logic (or Lisp) to denote the variable(s) in an expression to be "bound" when a functional form is applied. that is, instead of defining `f(x):=cos(x)` and then referring to `f`, we use the anonymous functional form `lambda(x),cos(x)`.

expressions with parts that look like $K(x)+K(y)$, or perhaps $K(x)+K(y)$ with $-1 \le x \le 1$ and $-1 \le y \le 1$. 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.

Ideally we can set a specification so a package can do such arithmetic on CS objects as makes sense, yet avoiding nonsensical "extensions" to the approach.

5 Sidelight: Expressions or functions?

This is a topic of concern in the world of CAS building, and is a human-factors question regarding modes of (human) thought: 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. The lambda-expressions we've shown above have a long history in logic (and in the Lisp programming language). In Lisp, a function we might define as `f(x):=x*(x+1)` can be expressed without using up a name like "f". In its parenthesized prefix notation Lisp 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, for the moment] 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)` or more explicitly, `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 judging correctly 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[4] 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

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

aspect of Chebshev approximation, and so much else about this topic has been well studied in the classical literature.

6 What to store?

The Chebfun Project design is set to store an array of values of the CS at a set of Gauss-Lobatto points (GLP): certain points in the interval [-1,1]. In particular, for n terms, these points are $\cos(k\pi/(n-1))$ for $0 \le k \le n-1$. (There are other choices possible, including points half-way between these chosen above). These particular sets of evaluation points 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[5].

We have one set of programs that manipulates CS based on a data representation that stores the Chebyshev coefficients (CC). Another set of programs uses GLP. These representations can be rapidly 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 must 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[6].

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 [4] 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 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 algorithmic approach, while inefficient for small cases, also lead to vastly more efficient large-scale 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.)

7 Implementation Notes

1. Using an FFT[7] for DCT indeed speeds things up considerably over the naive summation formula. Writing a specific fast DCT program[8] 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

[5] 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. The DCT is often implemented in a suite of programs based on fast Fourier transforms

[6] Actually, they have to be represented at the same GLP set, 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.

[7] 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.

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

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 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 [2]. Readers interested in pursuing this aspect of this paper may consider yet alternative DCTs[9]. 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, a program may run faster with an algorithm that is $O(n^2)$ rather than $O(n \log n)$ operations for a length-n FFT. The "slow" cosine transform can, at least for short cases, produce useful *exact answers in terms of symbolic (algebraic) roots and rational numbers.*

2. We have overly simplified the calculation heuristics for how many terms are needed to represent a function. Our 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 certainly important to look at two or more coefficients since common functions may be even or odd, and therefore have alternate coefficients being zero. We could implement the more elaborate code in Chebfun, but we leave that to future developers.

3. We 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. It is of course possible to accurately track the extensions of the Chebfun Project by gettin a copy of Matlab and Chebfun! (This is discounting the educational or fun aspects of revisiting implementations).

5. We believe that many functions that we supply, while formally correct, 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 computes $\cos(3\pi/4)$ as $-\frac{1}{\sqrt{2}}$, and can continue working with expressions of that kind, but not if a numerical value for π is first inserted).

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

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][10] Some of the programs now insert this "Horner" conversion, routinely.

8 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. Exact numbers and certainly symbolic entries 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. It should be clear that the constants `t1` and `t2` can precomputed. Less obviously it can be improved 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.

9 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 (even sometimes non-numeric) argument types. These Maxima programs can be sped up by compiling into Lisp and then into 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, but this requires some data conversion if we are to actually re-use the results in the Maxima context.

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

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 Maxima's 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 more 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 and 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. From the applications in the Chebfun guide, it seems that this is not a common operation, in any case.

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 are potentially still under development, and so may grow or shrink; 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.

10 Examples

After loading the chebfun files for Maxima, we can do the following:

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

This results in the following expression:

```
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)
```

This is not meant to be examined by human eyes, but if we replace all "small" values by 0 it looks somewhat better. For this purpose We defined a program `stz` which converts small numbers to zeros.

```
[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 CS 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 carried should expand automatically as necessary (if the design is successful). We have tried to emulate the Chebfun Project heuristics. e.g. let

```
 five: onepowcs[5]
```

`length(five)` is 6 as expected.

`length(cc(five,five))` is 26, as expected. The CS approximation should be good for values in [-1,1], but in this case appears to be good elsewhere too.

```
  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 with a Chebyshev series, though whether this is meaningful may depend on having functions with appropriate domains.

Illustrating `chebplus` and `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)`, heuristically at least. 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[11].

Other operations on CS 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 CS 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)
```

[11] A referee pointed out that the term Chebyshev norm in conventional mathematical analysis is the uniform norm or sup norm. We didn't write the program that way, but could easily do so.

11 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, rational numbers, high-precision numbers, 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] FFTW: Fastest Fourier Transform in the West, `http://fftw.org`

[3] Mathworks, Matlab System, `http://www.mathworks.com/products/matlab`

[4] L. N. Trefethen and others, Chebfun Version 4.2, The Chebfun Development Team, `http://www.chebfun.org/` 2011

12 Appendix: Some Programs

There are a collection of experimental programs in files as indicated in the sections on Status and Implementation Notes.

We also experimented with the use of the DCT for multiplying polynomials (with floating-point coefficients). [12]

We include in this section a listing of the file called `mincheby.mac` for reference:

```
/* Chebyshev series: a small set of programs to get you started */

/* Convert analytic function lambda([x]...expression in x) to Chebyshev series
  The result will be a list of double-float Chebyshev coefficients.
  We use a Discrete Cosine Transform, (DCT) and also a fastDCT implemented
  by FFT.   */

load("c:/cheby/dct.mac")$
 /*defines various programs, arbitrary size slow DCT.
   load("c:/cheby/algo749formax.o")$
/* defines the functions fdct, fdcti using lisp programs. */

ispowerof2(n):=is(integerp(n) and (n>0) and n= 2^?round(?log(n,2))) ;

tocheby2(f,n):=
/* n need not be a power of 2 for this program, but
   if n IS a power of 2, it uses a fast transform method.
   The program assumes that f, which is the name of a function or perhaps
   looks like lambda([r],...), evaluates to a real number at each relevant point. */
 block([dc:if ispowerof2(n) then fdct(glpf(f,n)) else DCT(glpf(f,n)),
```

[12] It has been well-known that this could be done via FFTs but the programs we've used have not taken advantage of the fact that the inputs and outputs are real-valued; the DCT makes that assumption. The speedup for DCT is modest but noticeable. An alternative which came to light, of pasting the two polynomial inputs into the real and imaginary parts of the input, makes the FFT quite close to the DCT in speed and space.

```
       in:sqrt(4.0d0/n)],
       apply('chebseries,makelist(dc[i]*in,i,1,n) ))$

tocheby2slow(f,n):=
/* n need not be a power of 2 for this program,
   f need not evaluate to a real number (but if symbolic,
   expect a large expression!). */
 block([dc: DCT(glpf(f,n)),
       in:sqrt(4.0d0/n)],
       apply('chebseries,makelist(dc[i]*in,i,1,n) ))$


fromcheby(CS,x):= if numberp(x) then eval_ts(CS,?float(x,1.0d0))
                        else  fcheb(CS,x)$

/* Evaluates a CS at a point x, ordinarily a double-float.  BUT..  If
 x is a symbol, this will convert Chebyshev series to an expression in
 the power basis, i.e. polynomial in x. Better to say
 fromcheby(a,rat(x)) which will simplify the answer to a polynomial in
 x.  (if CS include parameters, this still works.)  This algorithm is
 based on the Clenshaw recurrence, also see Fox & Mayers, Computing
 Methods, Chapter 7 ex 4. */

fcheb(c,x):=
block([keepfloat:true,
  bj2:0,bj1:0,bj0:0,z:reverse(args(c)),tx:2*x],
  while z#[] do
    (bj2:bj1,bj1:bj0,
     bj0:tx*bj1-bj2+first(z),
     z:rest(z)),
  1/2*(bj0-bj2))$

/* Compiling fcheb makes it run about 13X faster.  The Lisp version of
 this program, eval_ts, runs about 5X times faster on top of that.
  Compiling eval_ts gives about another 3X times faster, or in total,
 200X.  The code for eval_ts is in algo749formax.lisp */

/* Since some of these DCT programs produce tiny residual numbers, it
 is handy, while eye-balling them, to convert such numbers to zeros. */

small_to_zero(expr,eps):=
   /* Replaces all numbers smaller in absolute value than eps by zero. */
  fullmap(lambda([x], if numberp(x) and abs(x)<eps then 0.0d0 else x),
             expr)$

/*shorthand for above, using a fixed epsilon that we found useful*/
stz(z):=small_to_zero(z,1.0d-9)$

/* Chebnorm computes the sum-of-squares "size" of a CS*/

chebnorm(r):=block([s:0], for i in args(r) do s:s+i^2, s)$
```

```
/* Given a Chebyshev series, return another one in which we may have
 removed terms that don't matter much; that is, we chop off trailing
 coeffs which were small relative to larger coeffs.  An interesting
 design question is how to deal with CS that are constants,
 e.g. chebseries(2.0d0) is effectively the number 1.0d0. That is, a
 number is a special case of a CS. We don't, for now.*/

trimcs(a,relerr):=
 block([max: apply(max,args(a)),
        min: apply(min,args(a)),
        r: reverse(args(a)),
        bigmaginv,mm],
  if (mm: float(max(abs(max),abs(min))))=0.0
     then return (chebseries(0.0)) else bigmaginv:1/mm,
 while (r#[] and abs(first(r))*bigmaginv < relerr) do r:rest(r),
 return(apply('chebseries,reverse(r))))$

goodenoughCS(a,relerr):= /* is this Chebyshev series good enough? */
 block([max: apply(max,a:args(a)),
        min: apply(min,a),
la:length(a),tail,
        bigmaginv],
 tail:inpart(a,[la-1,la]), /* extract the last two items */
 bigmaginv: relerr/max(abs(max),abs(min)),
 /*display(bigmaginv), */
  /* test to see that last 2 coeffs are relatively small */
 is (abs(tail[1]) < bigmaginv)
     and
    (abs(tail[2]) < bigmaginv))$

/* a simple formula (Thacher, 1964) sufficient to convert from x^r to Cheby.
    onepowcs[r] is the chebseries for x^r */

onepowcs(z):=onepowcs[z]$
onepowcs[r]:=apply('chebseries,makelist(
   if evenp(r-q)then 2^(1-r)*binomial(r,(r-q)/2) else 0,
                     q,0,r))$

/*if powers(a,b,c,d) is an encoding of a+b*x+c*x^2+d*x^3,
  with "x" understood, then converting
  from power basis [exactly!] to chebshev basis is */

power2cheby(p):=
 block([pa,ans,n:length(p)],
  local(pa,ans),
  pa[i]:=inpart(p,i+1),
  ans[i]:=0,
  for r:0 thru n-1 do /* for each power r*/
   for q:0 thru r do (
     if evenp(r-q)then
```

```
          ans[q]:ans[q]+pa[r]*2^(1-r)*binomial(r,(r-q)/2)),
  apply ('chebseries, makelist(ans[i],i,0,n-1)))$

cheby2power(c):=  /* convert to power basis list, not so efficient perhaps */
block([f:fromcheby(c,rat(zz))],
apply('powers,makelist(ratcoeff(f,zz,i),i,0,length(c)-1)))$

/* this version of tocheby takes an expression and tries to
   figure out how many terms to include.  It requires
   that you use a function, e.g. tocheby(lambda([x],x^2)).
   It will try up to 512 terms.
   See cheb-misc.mac for even simpler to use:  cheb(x^2)  */

tocheby(f):=
 block([n:8,trial,ratprint:false],
    trial:tocheby2(f,n),
   while (n<=toomany) and not( goodenoughCS(trial,relerr)) do
    trial:tocheby2(f,n:2*n),
    if n>=toomany then (print("Chebyshev encoding requires more than",
                    toomany, " terms for error",relerr),
     trial)
   else trimcs(trial,relerr))$

round2n(h):=2^ceiling(log(h)/log(2.0d0))$

toomany:512$
relerr:1.0e-13$


/*  required for ... collecting ...  and polyp.
   produces exact results for exact polynomials .

poly2cheb(e,x):= if polynomialp(e,[x]) then
     (e:rat(e,x),
   power2cheby( for i:0 thru hipow(e,x) collect ratcoef(e,x,i)))
else tocheby(e)$

*/

poly2cheb(e,x):= if polynomialp(e,[x]) then
      block([h:[],re:rat(e,x)],
    for i:0 thru hipow(re,x) do h:cons(ratcoef(re,x,i),h),
    power2cheby(reverse(h)))
    else cheby(e)$

/* trials
stz(tocheby(lambda([r],cos(5*acos(r))))) ;  works
stz(tocheby(lambda([r],cos(10*acos(r))))) ;  works
stz(tocheby(lambda([r],cos(11*acos(r))))) ;  doesn't work. too much float error
*/
```