Exact polynomial multiplication using approximate FFT

Richard J. Fateman
University of California
Berkeley, CA 94720-1776

May 4, 2005

Abstract

It is well-recognized in the computer algebra systems community that some version of the Fast Fourier Transform (FFT) can be used for multiplying polynomials, and in theory is fast, at least for "large enough" polynomials. The version used, sometimes call NTT for "number theoretic transform" does not use floating-point arithmetic, but "modular arithmetic" where integer operations are followed by reducing the result to a remainder (modulo the prime). Yet it appears that no major general-purpose computer algebra system uses the FFT for polynomial multiplication. The builders of these systems may view the FFT as too optimized for rare special cases, or perhaps too hard to program. In this brief paper we point out how to use an off-the-shelf floating-point FFT program to produce EXACT answers to polynomial multiplication problems for arbitrary-precision coefficient polynomials.

1 Introduction

We assume the reader has been exposed to the FFT or NTT as a technique for multiplying polynomials (or perhaps very long integers), and have heard of the Chinese Remainder Theorem. For background, see for example, Gathen and Gerhard [6], or (especially) Knuth [3]. A longer version of this paper with more details and alternatives to the FFT is available from the author [4]. Here we cut to the chase in order to make a recommendation for other system builders considering the FFT.

Given that the floating-point FFT gives approximate answers, can we use it to produce EXACT answers to multiplication problems for arbitrary-precision coefficient polynomials?

An example: consider a case in which the answer just cannot be represented, e.g. the input includes the number r=123,456,789,123,456,789. Note that r requires 56 bits to represent to full accuracy, so it seems we cannot use the double-precision (53 bit) floating-point FFT for a polynomial with this coefficient, much less getting the right answer which includes multiples of r or even r^2.

NOT SO!

Consider the primes $m_0 = 33554393$, $m_1 = 33444383$, and $m_2 = 33554371$, all less than 2^{25}, so their pairwise products fit comfortably[1] in 52 bits. We can represent $r \bmod m_0 = 27733531$ exactly as a double-float, and similarly for $r \bmod m_1 = 7390453$ and $r \bmod m_2 = 5709040$. We can use these numbers in three separate *floating-point FFTs*, and combine them using the Chinese Remainder Algorithm. For example, if we are squaring $(1 + 123456789123456789x)$, we would instead square $(1 + 27733531x)$, $(1 + 5709040x)$, etc. As long as these computations can be done exactly in a floating-point FFT, we are safe.

How did we know that three FFTs were needed? We compute a bound for the coefficients in the product of two polynomials. Based on this, as well as the comfort bound (see below) we can tell how many moduluar images we need. We can then modularly reduce the inputs so that the answer's coefficients can be represented

[1] How comfortably they must fit is an important question addressed shortly.

correctly in the floating-point fraction. These can then be combined to get the correct answer by the Chinese Remainder Algorithm [6].

One coefficient bound for the answer is easy to compute. For polynomials p_1 and p_2, the bound is maxcoef(p_1)·maxcoef(p_2)·(1+min(deg(p_1),deg(p_2))). (Proof left as an exercise: consider squaring a polynomial with coefficients that are all 1).

How comfortable do we need to be? The longer version [4] of this paper includes some analysis, but one can get a good idea of how this might work with some experiments on our particular FFT program, translated into Lisp from "Numerical Recipes". The experiment is simple: Choose a particular sequence size, and see what the largest k is such that an input polynomial with random coefficients, but whose coefficient bound is 2^k can be squared by the two FFT and one inverse FFT method, yielding the *exact* answer. The FFT is using only double-precision floats. This is what we found:

size	k
32	23
64	23
128	21
256	20
512	20
1024	19
2048	19
4096	18
8000	17
16000	17

For example, if you have a polynomial p of length 2048, whose coefficients are random integers less than 2^{19}, you can multiply p by p and get all the coefficients in the *exact* answer p^2 by running the floating-point FFT procedure we tested, and rounding the coefficients in the answer to the nearest integer. (Preview for fans of the Chinese Remainder Algorithm: If p has coefficients bounded by 2^{38} you will have to run the process twice, having reduced p's coefficients in two different ways to numbers less than 2^{19}).

2 Four ways to use an FFT to multiply polynomials

There are three common approaches, and one more which we suggest below. There may be others.

1. Use higher precision floating-point arithmetic, like doubled-double (quadruple) or double it yet again. This is not a general solution for arbitrary inputs because exponent overflow and truncation must be controlled, and these approaches provide a particular maximum precision. That is, fractions and exponents don't grow as in the GMP software, although they are fast for the precision they deliver. Along the same line, however, we can implement the FFT using GMP style bigfloat arbitrary precision arithmetic. In this case the number of bits needed in the answer tells us how much accuracy must be retained in the FFT which then can be directly related to the size and accuracy in the particular implementation of the FFT. Thus if you wish to perform the previous $p \times p$ example of length 2048, but the coefficients in p are bounded by 2^{59}, you would need an FFT that can deliver 40 more bits (or 93 total, since double-precision already provides 53). Note that the precise implementation of the GMP FFT affects the accuracy.

2. Find a bound B for the largest coefficient in the answer. Identify a finite field of size at least $2B$ which also contains a value ω which is a primitive 2^nth root of unity, where 2^n exceeds the degree of the answer. Computing two forward transforms and one reverse transform gives us the product.

3. Do the FFT arithmetic itself in a number of finite fields whose moduli, when multiplied together, exceed $2B$. The Chinese Remainder Theorem (Garner's algorithm) allows these pieces to be put together, in

a final step. (There are numerous minor variations [6], some of which can make substantial timing differences. A floating-point FFT is much faster.)

4. (We think this idea is new, and the point of this short note.) We do the arithmetic not in a single finite field, or a collection of fields each with an ω or over a high-precision floating-point domain. Rather, we use the existing highly-tuned floating-point computations familiar in numerical scientific computation, repeatedly. The innovation is that the *inputs only* have been reduced modulo selected primes m_0, m_1, $q_2, \cdots$, (e.g. each less than 2^{19} if the inputs are 1024 term polynomials) The arithmetic is done rapidly *in floating-point* and then rounded to integers. We know these integers are exactly correct, even when reduced modulo those same primes m_0, m_1, $m_2, \cdots$. The resulting exact images can be collected via the Chinese Remainder Theorem Garner's algorithm) in a final step, the only one which requires "bignum" arithmetic.

The big win here is that any time a better numerical FFT is available on whatever computer we are using, we can use it. The FFTW website shows how intense this development activity is.

3 Conclusion

Actually, we find that FFT methods are potentially faster than other methods only for quite large univariate polynomials (degree 1,000 or so with rather small coefficients), at least when compared to other methods implemented carefully. Classical or Karatsuba-style multiplication is a good bet. We would like to encourage users of an NTT, to try a floating-point FFT instead. More justification of this recommendation, as well as benchmark comparisons to other polynomial multiplication methods, and complete program listings, can be found in a longer version of this paper [4].

References

[1] D. Bailey, ARPREC. http://crd.lbl.gov/ dhbailey/mpdist/

[2] The Gnu MP Home page. http://swox.com/gmp/

[3] D. E. Knuth, *The Art of Computer Programming*, volume 2. (1969, 1981). Addison-Wesley.

[4] R. Fateman, "Is FFT multiplication of arbitrary-precision polynomials practical?," draft, 5/2005.

[5] R. Fateman, "Can you save time in multiplying polynomials by encoding them as integers?", draft, 2004

[6] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*, Cambridge Univ. Press 1999, 2003.

[7] William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling. *Numerical Recipes in Fortran* (various editions), lisp version online at www.library.cornell.edu/nr/cornell_only/othlangs/lisp.1ed/kdo/readme.htm.

[8] V. Shoup. NTL, (Number Theory Library) `http://shoup.net/ntl/`.