Efficiently computing Bernoulli numbers using FLINT

David J. A. Howden - URSS Project, Summer 2007. Supervisor: Dr William Hart

THE UNIVERSITY OF WARWICK

Introduction

Named after the Swiss mathematician Jackob Bernoulli (1654-1705), the Bernoulli numbers B_n are a sequence of signed rational numbers that can be identified by the following identity

$$\frac{x}{e^x-1} \equiv \sum_{n=0}^{\infty} \frac{B_n x^n}{n!}.$$

In simpler terms, they come from the coefficients of the Taylor expansion of $x/(e^x-1)$, and as such can be calculated recursively by setting $B_0 = 1$, and then using the relation

$$\binom{k+1}{1}B_k + \binom{k+1}{2}B_{k-1} + \ldots + \binom{k+1}{k}B_1 + B_0 = 0.$$

Bernoulli numbers are seen as central to many ideas and structures in algebraic number theory. For instance, Euler used them to express the sums of equal powers of consecutive integers. Bernoulli numbers even featured in early attempts to solve Fermat's Last Theorem.

Using machines to generate Bernoulli numbers is an old problem dating back to 1842 when Ada Byron constructed an algorithm which would allow Charles Babbage's 'Analytical Engine' to compute them. This makes the Bernoulli numbers the subject of one of the first computer programs ever conceived. This problem continues today, where there are numerous efforts by different groups of mathematicians to compute Bernoulli numbers to increasing limits.

Here we will concentrate on computing Bernoulli numbers modulo a prime number p.

Theory - Reducing to Polynomial Multiplication

In order to compute the Bernoulli numbers modulo a prime number p, we will followed a method outlined by David Harvey. The algorithm is split into two parts. The first part involves expressing Bernoulli numbers in terms of distributions on $\mathbf{Z}_p$ which is taken from Lang's "Cyclotomic Fields" (chapter 2, Theorem 2.3), and can be written as follows

$$B_k/k = \frac{1}{1-g^k} \sum_{x \in \mathbf{Z}/p\mathbf{Z}} x^{k-1} h(x) \mod p$$

where g is a generator of $(\mathbf{Z}/p\mathbf{Z})^*$, and where h is defined as

$$h(x) = \left\{\frac{x}{p}\right\} - g\left\{\frac{g^{-1}x}{p}\right\} + \frac{g-1}{2}.$$

Here $\{\cdot\}$ denotes the fractional part. By substituting $x = g^j$, using the fact that $h(g^j)/g^j$ has period $(p-1)/2$ as a function of j, and since we are only interested in even k, we have

$$B_{2k} = \frac{4k}{1-g^{2k}} \sum_{j=0}^{(p-3)/2} g^{2jk} \frac{h(g^j)}{g^j} \mod p.$$

The sum on the right is a number-theoretic transform of length $(p-1)/2$. The second part of the algorithm involves evaluating this transform by using *Bluestein's trick*, which states that any transform of the form

$$b_k = \sum_{j=0}^{(p-3)/2} g^{2jk} a_j$$

can be rewritten using the identity $2jk = k^2 + j^2 - (k-j)^2$ as

$$b_k = g^{k^2} \sum_{j=0}^{(p-3)/2} c_{k-j} d_j$$

where $c_j = g^{-j^2}$ and $d_j = g^{j^2} a_j$. This last sum is a convolution, and so it tantamount to computing the product of the polynomials

$$F(X) = \sum_{j=-(p-3)/2}^{(p-3)/2} c_j X^j \text{ and } G(X) = \sum_{j=0}^{(p-3)/2} d_j X^j.$$

In fact one checks that $c_{j+(p-1)/2} = (-1)^{(p-1)/2} c_j$, so

$$F(X) = 1 + \left(1 + (-1)^{(p-1)/2} X^{-(p-1)/2}\right) \sum_{j=1}^{(p-3)/2} c_j X^j$$

and this observation reduces the problem to multiplying two polynomials of length $(p-1)/2$.

Polynomials in FLINT

FLINT is an open source library coded in C which provides various number theory algorithms and structures. In particular, FLINT implements multi-precision integer polynomials and is currently faster than all other available libraries for polynomial operations.

There are several separate FLINT modules, which allow for ordinary polynomial multiplication tuned for different inputs.

- 'Multi-precision Integer Polynomials' or 'mpz_poly'. Polynomials with arbitrarily large coefficients, which are automatically resized if necessary.
- 'Flat Multi-precision Integer Polynomials' or 'fmpz_poly'. Polynomials with arbitrarily large coefficients that are fixed in size. These polynomials are recommended when the the size of the output coefficients is known. Since there is no coefficient resizing in any operations, this module provides performance improvements over the 'mpz_poly' polynomial module.

Since the two input polynomials are represented mod p, the size of each of the coefficients in the polynomials is less than the size of p. Hence either the 'mpz_poly' or 'fmpz_poly' can be used to implement the Bernoulli algorithm.

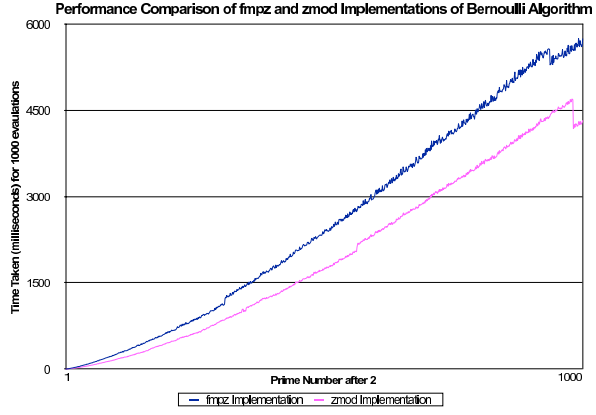
However, we can improve on this. Since we only need the coefficients of the resulting polynomial mod p, a new FLINT polynomial module, 'zmod_poly', which handles polynomials modulo a prime number should give some performance increase by computing the modulo arithmetic within the multiplication instead of after it.

Results

I implemented 'mpz_poly', 'fmpz_poly' and 'zmod_poly' versions of the algorithm described earlier along with the new FLINT module 'zmod_poly', and found that

- The 'fmpz_poly' implementation is twice as fast as the 'mpz_poly' implementation.
- The implementation using the new module which I have contributed, 'zmod_poly', out performs the 'fmpz_poly' implementation by 20% in general.

The following graph demonstrates the performance difference between the 'fmpz_poly' and 'zmod_poly' implementations. The code has been profiled by computing the Bernoulli numbers for the first 1000 prime numbers (beginning with 3), each done 1000 times and totalling the time taken for each.



Conclusions and Future Directions

The 'zmod_poly' implementation of the algorithm to find the Bernoulli numbers mod p out performs the other FLINT polynomial module implementations, and 'zmod_poly' should in most cases outperform 'fmpz_poly' and 'mpz_poly' for polynomial arithmetic mod p.

The Bernoulli algorithm is limited by the size of the prime number p, which must be less than than 18446744073709551615 on a 64-bit computer. The FLINT module 'zmod_poly' could be extended to included arbitrarily large primes, and hence compute Bernoulli numbers for arbitrarily large p.