Old and new algorithms for computing Bernoulli numbers

David Harvey

University of New South Wales

25th September 2012, University of Ballarat

Bernoulli numbers

Rational numbers $B_0, B_1, \ldots$ defined by:

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

The first few Bernoulli numbers:

$$\begin{aligned}
B_0 &= 1 & B_1 &= -1/2 \\
B_2 &= 1/6 & B_3 &= 0 \\
B_4 &= -1/30 & B_5 &= 0 \\
B_6 &= 1/42 & B_7 &= 0 \\
B_8 &= -1/30 & B_9 &= 0 \\
B_{10} &= 5/66 & B_{11} &= 0 \\
B_{12} &= -691/2730 & B_{13} &= 0 \\
B_{14} &= 7/6 & B_{15} &= 0
\end{aligned}$$

Bernoulli numbers

Applications:

- Sums of the form $1^n + 2^n + \cdots + N^n$
- Euler–Maclaurin formula
- Connections to Riemann zeta function, e.g. Euler's formula:

$$\zeta(2n) = \frac{1}{1^{2n}} + \frac{1}{2^{2n}} + \frac{1}{3^{2n}} + \cdots = (-1)^{n+1} \frac{(2\pi)^{2n} B_{2n}}{2(2n)!}$$

- Arithmetic of cyclotomic fields, p-parts of class groups
- ...

Three algorithmic problems

I will discuss three algorithmic problems:

1. Given n, compute $B_0, B_1, \ldots, B_n$.
2. Given n, compute just B_n.
3. Given prime p, compute $B_0, B_2, \ldots, B_{p-3}$ modulo p.

How fast can we solve these problems?

How big is B_n?

Write B_{2n} as a reduced fraction

$$B_{2n} = \frac{N_{2n}}{D_{2n}}.$$

D_{2n} has $O(n)$ bits (von Staudt–Clausen theorem).

N_{2n} has $\Theta(n \log n)$ bits (Euler's formula and Stirling's formula).

Example:

$$B_{1000} = \frac{-18243\ldots(1769 \text{ digits})\ldots 78901}{2 \cdot 3 \cdot 5 \cdot 11 \cdot 41 \cdot 101 \cdot 251}.$$

Computing $B_0, B_1, \ldots, B_n$ — cubic algorithms

Equating coefficients in

$$\left(\frac{e^x - 1}{x}\right) \sum_{n \geq 0} \frac{B_n}{n!} x^n = 1$$

leads to the recursive formula

$$B_n = -\sum_{k=0}^{n-1} \frac{1}{k!(n-k+1)} B_k.$$

Given $B_0, \ldots, B_{n-1}$, we can compute B_n using $O(n)$ arithmetic operations.

But each B_k has $O(k \log k)$ bits, so we need $O(n^{2+o(1)})$ bit operations.

For computing $B_0, \ldots, B_n$, we need $O(n^{3+o(1)})$ bit operations.

Computing $B_0, B_1, \ldots, B_n$ — cubic algorithms

There are many other identities of this sort that lead to $O(n^{3+o(1)})$ algorithms for computing $B_0, \ldots, B_n$, for example:

- Seidel (1877)
- Ramanujan (1911)
- Knuth–Buckholtz (1967)
- Atkinson (1986)
- Akiyama–Tanigawa (1999)
- Brent–H. (2011)

They have various big-O constants and numerical stability properties.

Computing $B_0, B_1, \ldots, B_n$ — fast algorithms

But $B_0, \ldots, B_n$ only occupy $O(n^2 \log n)$ bits altogether.

Can we compute them all in only $O(n^{2+o(1)})$ bit operations?

Yes! (Brent–H., 2011)

There are several ways — I will sketch a particularly cute version.

Computing $B_0, B_1, \ldots, B_n$ — fast algorithms

Define the Tangent numbers $T_1, T_2, \ldots$ by

$$\tan x = \sum_{n \geq 1} \frac{T_n}{(2n-1)!} x^{2n-1}.$$

Here are the first few:

$$1, 2, 16, 272, 7936, 353792, \ldots$$

They are all *positive integers*, and it turns out that

$$T_n = \frac{(-1)^{n-1} 2^{2n} (2^{2n} - 1)}{2n} B_{2n}.$$

So it suffices to compute the first n Tangent numbers.

Computing $B_0, B_1, \ldots, B_n$ — fast algorithms

Watch for the non-overlapping coefficients:

$$9! \tan(10^{-3}) = 362.880120960048384019584007936\ldots$$

We can easily read off the Tangent numbers:

$$\begin{aligned} T_1 &= 362880/9! = 1 \\ T_2 &= 120960/(9!/3!) = 2 \\ T_3 &= 48384/(9!/5!) = 16 \\ T_4 &= 19584/(9!/7!) = 272 \\ T_5 &= 7936/(9!/9!) = 7936. \end{aligned}$$

In general, to compute $T_1, \ldots, T_n$, it suffices to compute $(2n-1)! \tan(10^{-p})$ to about $n^2 \log n$ significant digits, where $p = O(n \log n)$.

Computing $B_0, B_1, \ldots, B_n$ — fast algorithms

How do we compute $(2n-1)!\tan(10^{-p})$?

Like this:

$$9!\tan(10^{-3}) = 8!\frac{9!\sin(10^{-3})}{8!\cos(10^{-3})}$$

$$= 8!\frac{362.879939520003023999928000001\ldots}{40319.979840001679999944000001\ldots}$$

Writing down the required digits of $\sin(10^{-p})$ and $\cos(10^{-p})$ costs $O(n^2 \log^2 n)$ bit operations.

Then perform a single floating-point division with $O(n^2 \log n)$ digits of accuracy. (Some error analysis required.)

Assuming FFT-based algorithms, total cost is $O(n^2 \log^{2+o(1)} n)$ bit operations.

(Of course we work in base 2, not base 10!)

Computing B_n

What about computing a single B_n, without necessarily computing $B_0, \ldots, B_{n-1}$?

Since B_n has only $O(n \log n)$ bits, maybe we can do it in $O(n^{1+o(1)})$ bit operations?

Can we at least do better than $O(n^{2+o(1)})$?

Computing B_n

I'll discuss three algorithms:

- The zeta-function algorithm
  (rediscovered many times... Euler, Chowla–Hartung 1972, Fillebrown–Wilf 1992, Fee–Plouffe 1996, Kellner 2002, Cohen–Belabas–Daly (?))
- The multimodular algorithm (H. 2008)
- A new p-adic multimodular algorithm (H., 2012 — see arXiv)

Computing B_n — the zeta-function algorithm

Idea is to use Euler's formula

$$B_{2n} = (-1)^{n+1} \frac{2(2n)!}{(2\pi)^{2n}} \zeta(2n).$$

It suffices to compute $O(n \log n)$ significant bits of the right hand side.

It is known how to compute $n!$ and π to this precision in $O(n^{1+o(1)})$ bit operations.

But computing enough bits of $\zeta(2n)$ is more expensive, so we concentrate on this.

Computing B_n — the zeta-function algorithm

We use the Euler product:

$$\zeta(2n)^{-1} = \prod_{p \text{ prime}} (1 - p^{-2n})$$

It turns out that we need $O(n/\log n)$ primes to get enough bits, and we need to do $O(n^{1+o(1)})$ work per prime.

Total: $O(n^{2+o(1)})$ bit operations.

No better than computing all $B_0, \ldots, B_n$!

(Fillebrown gives the more precise bound $O(n^2 \log^{2+o(1)} n)$. Maybe this can be improved by a logarithmic factor?)

Computing B_n — the multimodular algorithm

Idea: compute $B_n \pmod p$ for many primes p and reconstruct using the Chinese Remainder Theorem.

Using fast interpolation algorithms, reconstruction step costs only $O(n^{1+o(1)})$ bit operations.

How many primes do we need? Since $\sum_{p<N} \log p \sim N$, we need all primes up to $O(n \log n)$, i.e. $O(n)$ primes.

The expensive part is computing $B_n \pmod p$ for each p. How do we do this?

Computing B_n — the multimodular algorithm

Assume $n \geq 2$, n even, and $p \geq 3$.

Define 'integralised' Bernoulli number

$$B_n^* = 2(1 - 2^n)B_n \in \mathbf{Z}.$$

Basic identity (a 'Voronoi congruence'):

$$\begin{aligned} B_n^* &= n \sum_{j=1}^{p-1} (-1)^j j^{n-1} \pmod{p} \\ &= n(-1^{n-1} + 2^{n-1} - \cdots + (p-1)^{n-1}) \pmod{p}. \end{aligned}$$

Straightforward evaluation scheme: compute in turn 1^{n-1}, 2^{n-1}, etc, all modulo p.

Computing B_n — the multimodular algorithm

Better: rearrange terms. Put $j = g^i$ for a generator g.

Example to compute $B_8 \pmod{23}$, using $g = 5$:

$$-1^7 - 5^7 + 2^7 + 10^7 + 4^7 + 20^7 + 8^7 - 17^7 + 16^7 - 11^7 - 9^7 \pmod{23}.$$

Can keep track of g^i, $g^{i(n-1)}$ with only $O(1)$ operations per term.

The sign of each term depends on the parity of g^i.

The sum for $g^{11}, \ldots, g^{22}$ is the same as for $g^0, \ldots, g^{10}$.

Complexity: $O(p^{1+o(1)})$ bit operations per prime.

Overall complexity to get B_n: still $O(n^{2+o(1)})$.

Computing B_n — the multimodular algorithm

Various tricks to squeeze out large constant factor... beyond the scope of this talk.

Current record for a single B_n is held by the multimodular algorithm.

Timing data from 2008 paper:

- B_{10^7}: 40 CPU hours using zeta function algorithm (Pari/GP).
- B_{10^7}: 10 CPU hours using multimodular algorithm.
- B_{10^8}: 1164 CPU hours using multimodular algorithm.

This data nicely illustrates quasi-quadratic asymptotics.

Wikipedia reports that Holoborodko recently used my implementation to compute $B_{2\times 10^8}$.

Computing B_n — the p-adic multimodular algorithm

While preparing for this talk, I discovered a new algorithm:

Theorem (H., 2012)

The Bernoulli number B_n may be computed in

$$O(n^{4/3+o(1)})$$

bit operations.

This is the first known *subquadratic* algorithm for this problem.

Computing B_n — the p-adic multimodular algorithm

Sketch: the Voronoi identity may be generalised to obtain information modulo *powers* of p:

$$B_n^* = \sum_{j=0}^{p-1} \sum_{k=0}^{s-1} \binom{n}{k+1} B_{k+1}^* p^k (-1)^j j^{n-k-1} \pmod{p^s}$$

(assuming $s \leq n$).

For $s = 1$ this simplifies to the previous formula.

There are ps terms, and we need to do arithmetic modulo p^s.

So the complexity of evaluating this sum (ignoring logarithmic factors) is roughly ps^2.

Computing B_n — the p-adic multimodular algorithm

Optimisation problem: select precision parameter s_p for each p.

Want to minimise the cost (roughly $\sum_p p s_p^2$) while ensuring that we collect enough bits (roughly $\sum_p s_p \log p$) to determine B_n.

Unfortunately this doesn't work — it still leads to overall cost $O(n^{2+o(1)})$.

(This remains true even if you throw in bits from $\zeta(n)$ to yield information at the prime at infinity.)

Computing B_n — the p-adic multimodular algorithm

But we can rewrite the identity like this:

$$B_n^* = \sum_{j=0}^{p-1} (-1)^j j^{n-s} p^{s-1} F(j/p) \pmod{p^s}$$

where

$$F(x) = \sum_{k=0}^{s-1} \binom{n}{k+1} B_{k+1}^* x^{s-1-k} \in \mathbf{Z}[x].$$

Note $F(x)$ is a polynomial of degree $s-1$, and is *independent* of p.

Select parameters $s = n^{2/3+o(1)}$, $M = n^{2/3+o(1)}$, $N = n^{1/3+o(1)}$.

Evaluate $F(j/p) \pmod{2^M}$ for every $p < N$, every $0 \le j < p$, *simultaneously* using a fast multipoint evaluation algorithm.

Then add everything up and use the Chinese Remainder Theorem!

Computing B_n — the p-adic multimodular algorithm

Actually one can interpolate between

- $O(n^{4/3+o(1)})$ time and $O(n^{4/3+o(1)})$ space, and
- $O(n^{3/2+o(1)})$ time and $O(n^{1+o(1)})$ space.

For what n do we beat the previous algorithms in practice?

No idea. I hope to find out soon...

Computing $B_0, \ldots, B_{p-3} \pmod{p}$ — motivation

The residues $B_0, B_2 \ldots, B_{p-3} \pmod{p}$ are closely related to the arithmetic of cyclotomic fields.

Let $K = \mathbf{Q}(\zeta_p)$ and let h be the class number of K.

Then

$$p \mid h \iff p \mid B_k \text{ for some } k = 0, 2, \ldots, p-3.$$

A prime p is *regular* if $B_k \not\equiv 0 \pmod{p}$ for all $k = 0, 2, \ldots, p-3$, i.e. $p \nmid h$.

Much of Kummer's work on Fermat's Last Theorem applies only to regular primes, essentially because one can then always take pth roots in the class group.

Computing $B_0, \ldots, B_{p-3} \pmod{p}$ — motivation

If p is not regular it is called *irregular*.

The smallest irregular prime is $p = 37$:

$$B_{32} = \frac{-7709321041217}{5100} = 0 \pmod{37}.$$

We call $(37, 32)$ an *irregular pair*.

Each irregular pair corresponds to a nontrivial eigenspace in the Galois action on the p-Sylow part of the class group of K.

The irregular pairs have been determined for all $p < 163\,577\,856$ (Buhler–H., 2011).

Computing $B_0, \ldots, B_{p-3} \pmod{p}$ — motivation

Once the irregular pairs for p are known, one can quickly test interesting conjectures, such as the Kummer–Vandiver conjecture.

This states that the class number of the maximal real subfield

$$K^+ = K \cap \mathbf{R} = \mathbf{Q}(\zeta_p + \zeta_p^{-1})$$

is not divisible by p.

Some people think it's true.

Some people think it's false.

(It's true for all $p < 163\,577\,856$.)

Computing $B_0, \ldots, B_{p-3} \pmod{p}$

Modern algorithms for computing $B_0, \ldots, B_{p-3} \pmod{p}$:

- Power series inversion
- Voronoi identity & Bluestein's trick
- Voronoi identity & Rader's trick

All have theoretical bit complexity $O(p \log^{2+o(1)} p)$.

These days we can only fight to improve the constant.

(Pre-1992 algorithms had complexity $O(p^{2+o(1)})$.)

Computing $B_0, \ldots, B_{p-3} \pmod{p}$ — series inversion

Idea: invert the power series

$$\frac{e^x - 1}{x} = 1 + \frac{x}{2!} + \frac{x^2}{3!} + \cdots + \frac{x^{p-3}}{(p-2)!} + O(x^{p-2})$$

in $\mathbf{F}_p[[x]]$.

Actually we only need coefficents of even index. Can save a factor of two by using for example

$$\frac{x^2}{\cosh x - 1} = -2 \sum_{n \geq 0} \frac{(2n-1)B_{2n}}{(2n)!} x^{2n}.$$

So we need to invert $(\cosh x - 1)/x^2$ to about $p/2$ terms.

Computing $B_0, \ldots, B_{p-3} \pmod{p}$ — series inversion

Series inversion can be reduced to polynomial multiplication via Newton's method.

So the basic question is: how to multiply polynomials of degree $O(p)$ in $\mathbf{F}_p[x]$?

Fastest way to multiply polynomials is via FFT.

But we can't perform an efficient FFT over $\mathbf{F}_p$, because usually $\mathbf{F}_p$ won't have sufficiently many roots of unity of smooth order.

Computing $B_0, \ldots, B_{p-3} \pmod{p}$ — series inversion

We can lift the multiplication problem to $\mathbf{Z}[x]$, and then do either:

- Use floating-point FFT to multiply over $\mathbf{R}[x]$, and round results to the nearest integer, or
- Choose a few word-sized primes q such that $\mathbf{F}_q$ supports efficient FFT, multiply in $\mathbf{F}_q[x]$ for each q, then reconstruct via CRT, or
- Some combination of the above.

Computing $B_0, \ldots, B_{p-3} \pmod{p}$ — series inversion

Suppose $f, g \in \mathbf{Z}[x]$, of degree p, with coefficients bounded by p.

Coefficients of $h = fg$ are bounded by p^3.

Example from 1992: $p \sim 10^6$, so coefficients of h have 60 bits.

- IEEE double-precision floating-point is only 53 bits!
  Doesn't quite fit.
- Could use two primes near 2^{32}.

Computing $B_0, \ldots, B_{p-3} \pmod{p}$ — series inversion

Let's cheat: represent coefficients in $(-p/2, p/2)$ instead of $[0, p)$.

Then statistically we expect coefficients of $h = fg$ to be bounded by $O(p^{2.5})$.

Try $p \sim 10^6$ again: now coefficents of h have 50 bits.

This fits into floating-point (just), but no longer provably correct.

Fast-forward to 2012: now perhaps $p \sim 10^9$.

Even using $O(p^{2.5})$ bound, coefficients of h need 75 bits.

Computing $B_0, \ldots, B_{p-3} \pmod p$ — Voronoi identity

Recall Voronoi identity:

$$\frac{B_n^*}{n} = \sum_{j=1}^{p-1} (-1)^j j^{n-1} \pmod p.$$

Again let's put $j = g^i$ for g a generator of $(\mathbf{Z}/p\mathbf{Z})^*$, and let x_i be the corresponding ± 1 sign for each i. Also write $\ell = n - 1$, and $b_\ell = B_n^*/n$. The sum becomes

$$b_\ell = \sum_{i=0}^{p-2} g^{\ell i} x_i.$$

We want to evaluate this for all (odd) $0 \leq \ell < p - 1$.

This is precisely a *discrete Fourier transform* of the sequence $(x_0, \ldots, x_{p-2})$ over $\mathbf{F}_p$.

Computing $B_0, \ldots, B_{p-3} \pmod p$ — Voronoi identity

Can't evaluate the DFT directly over $\mathbf{F}_p$ (same reason as before).

Idea: use *Bluestein's trick* to convert to polynomial multiplication.

Using the identity $\ell i = \binom{\ell}{2} + \binom{-i}{2} - \binom{\ell-i}{2}$, the DFT

$$b_\ell = \sum_{i=0}^{p-2} g^{\ell i} x_i$$

becomes

$$g^{-\binom{\ell}{2}} b_\ell = \sum_{i=0}^{p-2} g^{-\binom{\ell-i}{2}} \left(g^{\binom{-i}{2}} x_i \right).$$

The right hand side is now a convolution of $g^{-\binom{i}{2}}$ and $g^{\binom{-i}{2}} x_i$.

Computing $B_0, \ldots, B_{p-3} \pmod{p}$ — Voronoi identity

Resulting algorithm involves multiplying two polynomials of degree about $p/2$ with coefficients in $\mathbf{F}_p$.

We run into the same problem as before: for $p \sim 10^9$, the product still has 75-bit coefficients (even if allow cheating).

I will now sketch how Rader's trick mitigates this problem. This seems to new in the context of computing Bernoulli numbers.

Computing $B_0, \ldots, B_{p-3} \pmod{p}$ — Rader's trick

First, some symmetry implies we only need sum over odd terms:

$$b_\ell = 2 \sum_{\substack{i=0 \\ i \text{ odd}}}^{p-2} g^{\ell i} x_i.$$

For simplicity let us assume that $q = (p-1)/2$ is *prime*.

Then $(\mathbf{Z}/2q\mathbf{Z})^*$ is cyclic of order $q-1$.

In fact $(\mathbf{Z}/2q\mathbf{Z})^*$ consists of all odd $0 \leq i < p-1$, except for q.

So

$$b_\ell = 2g^{\ell q} x_q + 2 \sum_{i \in (\mathbf{Z}/2q\mathbf{Z})^*} g^{\ell i} x_i.$$

Computing $B_0, \ldots, B_{p-3} \pmod{p}$ — Rader's trick

Now h be a generator of $(\mathbf{Z}/2q\mathbf{Z})^*$. Put $i = h^{-s}$ and $\ell = h^t$.

Then the sum

$$y_\ell = \sum_{i \in (\mathbf{Z}/2q\mathbf{Z})^*} g^{\ell i} x_i$$

becomes

$$y_{h^t} = \sum_{s=0}^{q-2} g^{h^{t-s}} x_{h^{-s}}.$$

The right hand side is now a convolution of g^{h^s} and $x_{h^{-s}}$.

But notice that the values $x_{h^{-s}}$ are all ± 1!

Computing $B_0, \ldots, B_{p-3} \pmod{p}$ — Rader's trick

So we end up with a polynomial multiplication fg, where:

- Both have degree about $p/2$,
- The coefficients of f are 'random' looking integers in $[0, p)$,
- The coefficients of g are all ± 1.

Now the coefficients of $h = fg$ are bounded by about p^2.

Even for $p \sim 10^9$ this fits into 64 bits.

If we cheat, we get the bound $p^{1.5}$. For $p \sim 10^9$ this is about 45 bits — it even fits into IEEE double-precision floating-point.

Computing $B_0, \ldots, B_{p-3} \pmod{p}$ — timings

Historical comparison:

Year	Hardware	Algorithm	p	Time
1992	Motorola 68040	series inversion	10^6	200s
2012	Intel 'Westmere' (12 cores)	Rader's trick	10^9	25s

Current plan is to do this for all $p < 2^{30} \sim 10^9$.

Should cost about 1.5 million CPU hours.

Thank you!