The Bailey-Borwein-Plouffe (BBP) Formula

Jonathan Cast
Christine Whitman

April 20, 2006

1 Introduction

It used to be thought that computing the nth digit of π requires computing the first $n-1$ digits. However, in 1995, David Bailey, Peter Borwein, and Simon Plouffe demonstrated that it is possible to directly calculate the nth digit (in certain bases) of several transcendental constants in time proportional to $n \ln^2 n$ and space proportional to $\ln n$.

Here are some examples:

$$\begin{aligned}
\pi^2 &= \frac{1}{8}\sum_{k=0}^{\infty}\frac{1}{64^k}\left[\frac{144}{(6k+1)^2} - \frac{216}{(6K+2)^2} - \frac{72}{(6k+3)^2} - \frac{54}{(6k+4)^2} + \frac{9}{(6k+5)^2}\right] \\
\tan^{-1}\frac{1}{3} &= \sum_{k=0}^{\infty}\frac{1}{16^k}\left[\frac{1}{8k+1} - \frac{1}{8k+2} - \frac{1/2}{8k+4} - \frac{1/4}{8k+5}\right]
\end{aligned}$$

2. Formula

We will be focusing on the formula for π known as the BBP formula:

$$\pi = \sum_{k=0}^{\infty}\frac{1}{16^k}\left[\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\right]$$

This formula allows you to directly calculate the hexadecimal (base 16) digits of π beginning at any arbitrary starting position n without having to calculate the first $n-1$ digits; it also allows the use of smaller data types to compute the first n digits, increasing the speed of the computation. Unless, of course, you actually want all of the first n digits; then, this algorithm is slower than other, older, computations.

The following formula will be used in the proof:

$$\begin{aligned}
&\int_0^{1/\sqrt{2}} \frac{x^{j-1}}{1-x^8}\mathrm{d}x \\
&= \int_0^{1/\sqrt{2}} \sum_{k=0}^{\infty} x^{8k+j-1}\,\mathrm{d}x \\
&= \frac{1}{2^{j/2}}\sum_{k=0}^{\infty}\frac{1}{16^k(8k+j)}
\end{aligned}$$

Now, the BBP formula for π can be written (using the above) as

$$\int_0^{1/\sqrt{2}} \frac{4\sqrt{2} - 8x^3 - f\sqrt{2}x^4 - 8x^5}{1-x^8}\mathrm{d}x$$

$$
\begin{aligned}
\text{(Substitute } y = \sqrt{2}x\text{)} &= \int_0^1 \frac{16(4-2y-y^4-y^5)}{16-y^8}\mathrm{d}y \\
&= \int_0^1 \frac{16(y-1)}{(y^2-2)(y^2-27+1)}\mathrm{d}y \\
&= \int_0^1 \frac{4y}{y^2-2}\mathrm{d}y - \int_0^1 \frac{4y-8}{y^2-2y+2}\mathrm{d}y \\
&= \left[2\ln(y^2-2)\right]_0^1 - 4\left[\tan^{-1}(1-y) + 2\ln((y-1)^2+1)\right]_0^1 \\
&= \left[2\ln(-1) - 2\ln(-2)\right] - \left[\left(4\tan^{-1}0 + 2\ln 1\right) - \left(4\tan^{-1}1 + 2\ln 2\right)\right] \\
&= 2\ln(-1) - 2\ln(-2) + 4\tan^{-1}0 - 2\ln 1 + 4\tan^{-1}1 + 2\ln 2 \\
&= 0 + 2\ln 2 - 2\ln 2 + 2 - 2 + 4\tan^{-1}1 \\
&= 4\tan^{-1}1 \\
&= 4\frac{\pi}{4} \\
&= \pi
\end{aligned}
$$

3. Calculation

Note that we can calculate the n hexadecimal digit of any number x by calculating $16^n x$ modulo 1 (that is, discarding any carry into the one's place), and using the hexadecimal digits from the fractional part.

To compute the nth hexadecimal digit of π using this formula, we first have to compute the sums of the terms:

$$
\begin{aligned}
A &= \sum_{k=0}^{\infty} \frac{1}{16^k(8k+1)} \\
B &= \sum_{k=0}^{\infty} \frac{1}{16^k(8k+4)} \\
C &= \sum_{k=0}^{\infty} \frac{1}{16^k(8k+5)} \\
D &= \sum_{k=0}^{\infty} \frac{1}{16^k(8k+6)}
\end{aligned}
$$

And then calculate $\pi = 4A - 2B - C - D$. This last computation can be done using floating point calculation modulo 1, and thus in constant time.

To calculate one of the constants above, we use the identity:

$$
\begin{aligned}
& 16^n \sum_{k=0}^{\infty} \frac{1}{16^k p(k)} \bmod 1 \\
= & \sum_{k=0}^{\infty} \frac{16^{n-k}}{p(k)} \bmod 1 \\
= & \sum_{k=0}^{n} \frac{16^{n-k} \bmod p(k)}{p(k)} \bmod 1 + \sum_{k=n+1}^{\infty} \frac{16^{n-k}}{p(k)} \bmod 1
\end{aligned}
$$

The second sum is $\leq \sum_{k=1}^{\infty} 16^{-k}$; we only need enough terms to make the sum of the remainder less than $\frac{1}{2}16^{-1}$. Note that $\sum_{k=l+1}^{\infty} 16^{-k} = \frac{1}{16}\sum_{k=l}^{\infty} 16^{-k}$; that is, dropping terms makes the sum decrease exponentially. Thus, if we take the first two terms, we are guaranteed that the sum of the remaining terms will be sufficiently low to guarantee the accuracy of the computed digits; the sum of the first two terms can be computed using regular floating-point arithmetic. For the first sum, the terms are quotients of

expressions whose maximums go to ∞ as $n \to \infty$. Thus using floating point arithmetic to calculate them will lose most of the digits of these terms, leaving a fairly inaccurate quotient with very little accuracy. Instead, we compute the numerator and denominator *exactly*, using infinite-precision integer arithmetic, and the use floating-point arithmetic to perform the division and add up the terms. Since fixed-precision floating-point operations take constant time and constant space, we do this part of the computation in linear time and constant space. Calculating $p(k)$ requires a constant number of arithmetic operations, which take logarithmic time and space in k. Thus, since we don't have to keep the values of $p(k)$ around between terms, we can do the total computation in time proportional to $n \ln n$ and space proportional to $\ln n$. The problem is calculating the numerators; since the numbers are taken modulo $p(k)$, we need at worst space proportional to $\ln k$, but a naive computation of the exponent would have time proportional to $(n-k)\ln k$, giving a total time (for the entire sum) proportional to $\frac{1}{2}n^2 \ln n$. Consequently, we need to present an algorithm for the exponent with time proportional to $\ln(n-k)\ln k$, giving total time of $n \ln^2 n$.

To compute $x^n \bmod c$, we use the following algorithm; this leaves the result in s.

$s := 1$
$t_0 := t_1 := t := \inf\{2^k | n \leq 2 \cdot 2^k - 1\}$
$m := n$
Loop :
  If $m \geq t_0$:
    $s := x \cdot s \bmod c$
    $m := m - 1$
  If $t_0 > 1$:
    $s := s \cdot s \bmod c$
    $t_1 := t_1/2$
  $t_0 := t_0/2$
While $t_0 \geq 1$

An easy induction shows that we have the following invariants:

$$\begin{aligned} s^{t_1} x^m &\equiv x^n \pmod{c} \\ 0 \leq m &\leq 2t_0 - 1 \\ 0 \leq s &< c \end{aligned}$$

When the algorithm finishes, we have the following additional conditions:

$$\begin{aligned} t_1 &= 1 \\ t_0 &= 1/2 \end{aligned}$$

From this it follows that after the algorithm is finished we have

$$0 \leq m \leq 2(1/2) - 1 = 1 - 1 = 0, \text{ so } m = 0; \text{ and}$$

$$s^1 x^0 \equiv x^n \pmod{c} \Rightarrow \equiv x^n \pmod{c}, \text{ and } 0 \leq s < c, \text{ so } s = x^n \bmod c.$$

Thus, the algorithm gives the correct answer. Clearly, it uses $O(\ln t)$ multiplications, and $t = O(\ln n)$, so the number of multiplications is $O(\ln n)$, so since the multiplications take time $O(\ln c) = O(\ln n)$, so the total timeis $O(\ln^2 n)$. Thus the total time to compute the nth digit of π is $O(n \ln^2 n)$.