The BBP Algorithm for Pi

David H. Bailey*

September 17, 2006

1. Introduction

The "Bailey-Borwein-Plouffe" (BBP) algorithm for π is based on the BBP formula for π, which was discovered in 1995 and published in 1996 [3]:

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right). \tag{1}$$

This formula as it stands permits π to be computed fairly rapidly to any given precision (although it is not as efficient for that purpose as some other formulas that are now known [4, pg. 108–112]). But its remarkable property is that it permits one to calculate (after a fairly simple manipulation) hexadecimal or binary digits of π beginning at an arbitrary starting position. For example, ten hexadecimal digits π beginning at position one million can be computed in only five seconds on a 2006-era personal computer. The formula itself was found by a computer program, and almost certainly constitutes the first instance of a computer program finding a significant new formula for π.

It turns out that the existence of this formula has implications for the long-standing unsolved question of whether π is normal to commonly used number bases (a real number x is said to be *b-normal* if every m-long string of digits in the base-b expansion appears, in the limit, with frequency b^{-m}). Extending this line of reasoning recently yielded a proof of normality for class of explicit real numbers (although not yet including π) [4, pg. 148–156].

*Lawrence Berkeley National Laboratory, Berkeley, CA 94720, dhbailey@lbl.gov. Supported in part by the Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy, under contract number DE-AC02-05CH11231.

2. Computing binary digits of log 2

The BBP formula arose in 1995, when Peter Borwein and Simon Plouffe of Simon Fraser University observed that the following well-known formula for $\log 2$ permits one to calculate binary digits of $\log 2$ beginning at an arbitrary starting position:

$$\log 2 \;=\; \sum_{k=1}^{\infty} \frac{1}{k 2^k}. \tag{2}$$

This scheme is as follows. Suppose we wish to compute a few binary digits following the first d binary digits (i.e., a few binary digits beginning at position $d+1$). This is equivalent to calculating $\{2^d \log 2\}$, where $\{\cdot\}$ denotes fractional part. Thus we can write

$$\begin{aligned} \{2^d \log 2\} &= \left\{ \left\{ \sum_{k=1}^{d} \frac{2^{d-k}}{k} \right\} + \sum_{k=d+1}^{\infty} \frac{2^{d-k}}{k} \right\} \\ &= \left\{ \left\{ \sum_{k=1}^{d} \frac{2^{d-k} \bmod k}{k} \right\} + \sum_{k=d+1}^{\infty} \frac{2^{d-k}}{k} \right\}. \end{aligned} \tag{3}$$

We are justified in inserting "mod k" here in the numerator of the first summation, because we are only interested in the fractional part of the quotient when divided by k.

The first summation consists of d terms, each of which is a quotient of integers no larger than k, which can be divided and then summed using ordinary floating-point computer arithmetic. For the second summation, only a few terms need to be evaluated, because they quickly become sufficiently small that they can be ignored, to the accuracy of the floating-point arithmetic being used in the calculation.

A key observation is that the numerators of the first summation in equation (3), namely $2^{d-k} \bmod k$, can be calculated very rapidly by means of the binary algorithm for exponentiation, performed modulo k. The binary algorithm for exponentiation is merely the formal name for the observation that exponentiation can be economically performed by means of a factorization based on the binary expansion of the exponent. For example, we can write $3^{17} = ((((3^2)^2)^2)^2) \cdot 3$, thus producing the result in only 5 multiplications, instead of the usual 16. In our application, we need to obtain the exponentiation result modulo a positive integer k. This can be done by modifying the binary algorithm for exponentiation as follows:

Binary algorithm for exponentiation modulo k:

To compute $r = b^n \bmod k$, where r, b, n and k are positive integers: First set t to be the largest power of two such that $t \le n$, and set $r = 1$. Then

A: if $n \ge t$ then $r \leftarrow br \bmod k$; $\quad n \leftarrow n - t$; $\quad$ endif
$t \leftarrow t/2$
if $t \ge 1$ then $r \leftarrow r^2 \bmod k$; $\quad$ go to A; $\quad$ endif

Note that this algorithm is performed entirely with positive integers that do not exceed k^2 in size. Thus the entire scheme to compute binary digits of $\log 2$ beginning after the first d digits (for reasonable-sized d) can be performed using only standard-precision computer hardware, and with very little memory. For example, computing binary digits beginning at the millionth position can be done in a second or two on a 2006-era personal computer.

3. Computing hexadecimal digits of pi

As soon as Borwein and Plouffe discovered the scheme to compute binary digits of $\log 2$, they began seeking other mathematical constants that shared this property. To that end, Plouffe performed integer relation searches to see if a formula of this type existed for π. This was done using a computer program written by the present author, which implements Helaman Ferguson's "PSLQ" integer relation algorithm [4, pg. 230–234] using high-precision floating-point arithmetic. Ferguson's PSLQ algorithm was recently recognized as one of the 20th century's most important algorithms, and is widely used in the emerging discipline of "experimental mathematics" [2].

Plouffe's search succeeded, producing the BBP formula for π:

$$\pi \;=\; \sum_{k=0}^{\infty} \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right). \tag{4}$$

From this formula, one can derive an algorithm for computing digits of π at an arbitrary starting position that is very similar to the scheme just described for $\log 2$. We state this algorithm explicitly as follows:

BBP algorithm for π:

To compute the hexadecimal digits of π beginning after the first d hex digits (i.e., beginning at position $d+1$): Let $\{\cdot\}$ denote the fractional part as before. Given an integer $d > 0$, we can write, from formula (4),

$$\{16^d \pi\} \;=\; \{4\{16^d S_1\} - 2\{16^d S_4\} - \{16^d S_5\} - \{16^d S_6\}\}, \tag{5}$$

where

$$S_j = \sum_{k=0}^{\infty} \frac{1}{16^k(8k+j)}. \tag{6}$$

Note that

$$\begin{aligned} \{16^d S_j\} &= \left\{ \left\{ \sum_{k=0}^{d} \frac{16^{d-k}}{8k+j} \right\} + \sum_{k=d+1}^{\infty} \frac{16^{d-k}}{8k+j} \right\} \\ &= \left\{ \left\{ \sum_{k=0}^{d} \frac{16^{d-k} \bmod 8k+j}{8k+j} \right\} + \sum_{k=d+1}^{\infty} \frac{16^{d-k}}{8k+j} \right\}. \end{aligned} \tag{7}$$

Now apply the binary exponentiation algorithm to (7), in a similar way as described above in the scheme for $\log 2$, to compute $\{16^d S_j\}$ for $j = 1, 4, 5, 6$. Combine these four results, as shown in (5). Add or subtract integers, as necessary, to reduce the final result to within 0 and 1. The resulting fraction, when expressed in hexadecimal notation, gives the first few hex digits of π that follow position d (i.e., the first few hex digits starting at position $d+1$).

As an example, when $d = 1,000,000$, we compute

$$\begin{aligned} S_1 &= 0.181039533801436067853489346252\ldots \\ S_2 &= 0.776065549807807461372297594382\ldots \\ S_3 &= 0.362458564070574142068334335591\ldots \\ S_4 &= 0.386138673952014848001215186544\ldots \end{aligned}$$

Combining these as in (5) yields

$$\{16^d \pi\} = 0.423429797567540358599812674109\ldots \tag{8}$$

We can convert this fraction to hexadecimal form by repeatedly multiplying by 16, noting the fractional part and then subtracting the fractional part, thus yielding the sequence $6, 12, 6, 5, 14, 5, 2, 12, 11, 4, 5, 9, 3, 5, 0, 0, 5, 0, 12, 4, 11, 11, 1, \cdots$, which when translated to hexadecimal notation is `6C65E52CB459350050E4BB1`. Indeed, these are the first 24 hexadecimal digits of π beginning at position 1,000,001.

4. Some implementation details

How many digits are correct depends on the precision of the floating-point arithmetic used to perform the division and summation operations in

formulas (5) and (7). Using IEEE 64-bit floating-point arithmetic (available on almost all computers) typically yields 9 or more correct hex digits (for reasonable-sized d); using 128-bit floating-point arithmetic (available on some systems at least in software) typically yields 24 or more correct hex digits. 128-bit floating-point arithmetic was used in the calculation described in the previous section.

Note, however, that the exponentiation operation $16^{d-k} \bmod 8k+j$ in the numerator of (7) must be performed exactly. This means, at the least, that d must be limited to those values such that $(8d+6)^2$ is exactly representable in the integer or floating-point data format being used. The author has found that on most systems using IEEE 64-bit floating-point arithmetic, a straightforward implementation of the above scheme works properly so long as d is less than approximately 1.18×10^7. If 128-bit floating-point arithmetic can be employed, the limit is much higher, typically at least 10^{14}. The exponentiation operation can also be done using integer arithmetic, if available in long word lengths.

However the BBP algorithm is implemented, it is important to note that a result calculated at position d can be checked by repeating at position $d-1$ (or $d+1$), and verifying that the hex digits perfectly overlap with an offset of one, except possibly for a few trailing digits.

Run times for the BBP algorithm depend on the system and the arithmetic being used, and increase roughly linearly with the position d. However, even rather large values of d can be handled in modest run times—the case $d = 10,000,000$ requires only 51 seconds on the author's Apple G5 workstation. A value of 10^{15} is roughly the present-day limit of computational feasibility on large parallel systems (see next section).

As can be seen from the above, the BBP algorithm is quite simple, and can be implemented quite easily on present-day systems, using only standard computer arithmetic and only a minuscule amount of memory. Readers are encouraged to try writing their own computer programs; alternatively, sample C, Fortran-90 and *Mathematica* implementations are available at:
`http://www.experimentalmath.info/bbp-codes`

5. Large computations

Needless to say, the BBP algorithm for π has been implemented by numerous researchers. Table 1 gives some results known as of this writing. The first few entries were due to the present author, and were included in the original paper on the BBP formula [3]. In 1997, Fabrice Bellard of INRIA

Position	Hex Digits Beginning at This Position
10^6	26C65E52CB4593
10^7	17AF5863EFED8D
10^8	ECB840E21926EC
10^9	85895585A0428B
10^{10}	921C73C6838FB2
10^{11}	9C381872D27596
1.25×10^{12}	07E45733CC790B
2.5×10^{14}	E6216B069CB6C1

Table 1: Computed hexadecimal digits of π.

computed 152 binary digits of π starting at the trillionth binary digit position. The computation took 12 days on 20 workstations working in parallel over the Internet. His scheme is actually based on the following variant of (4):

$$\pi \;=\; 4\sum_{k=0}^{\infty}\frac{(-1)^k}{4^k(2k+1)} - \frac{1}{64}\sum_{k=0}^{\infty}\frac{(-1)^k}{1024^k}\left(\frac{32}{4k+1}+\frac{8}{4k+2}+\frac{1}{4k+3}\right)$$

This formula permits individual hex or binary digits of π to be calculated roughly 43% faster than with (4).

A year later, Colin Percival, then a 17-year-old student at Simon Fraser University, utilized a network of 25 machines to calculate binary digits in the neighborhood of position 5 trillion, and then in the neighborhood of 40 trillion. In September 2000, he found that the quadrillionth binary digit is "0," based on a computation that required 250 CPU-years of run time, carried out using 1,734 machines in 56 countries. This is the last entry in the table.

Since the original discovery of the BBP formula for π, similar BBP-type formulas have been found for quite a few other mathematical constants. An updated compendium of such formulas is available at [1]. Some additional information on the BBP formula and related mathematics is available in [4, Chap. 3].

References

[1] David H. Bailey, "A Compendium of BBP-Type Formulas," 2005, available at `http://crd.lbl.gov/~dhbailey/dhbpapers/bbp-formulas.pdf`.

[2] David H. Bailey, "Integer Relation Detection," *Computing in Science and Engineering*, Jan-Feb 2000, pg. 24-28.

[3] David H. Bailey, Peter B. Borwein and Simon Plouffe, "On the Rapid Computation of Various Polylogarithmic Constants," *Mathematics of Computation*, vol. 66, no. 218 (Apr 1997), pg. 903–913.

[4] Jonathan M. Borwein and David H. Bailey, *Mathematics by Experiment: Plausible Reasoning in the 21st Century*, AK Peters, Natick, MA, 2004.