Calculating π to a million digits or more - a discursive approach

Samir Chettri

July 8, 2013

1 Why π?

Despite the hype explicit in the title of a recent book – "Pi: A Biography of the World's Most Mysterious Number" [2], there are many reasons to study this number and compute it to a very large number of digits. Here is why :

1. The number has been studied by a pantheon of mathematical greats including Newton, Euler, Ramanujan et al. Retracing their mathematical steps is pleasurable.
2. Computing the digits to (say) the millionth digit could help test the hardware providing the computational power.
3. It provides a good education in and applications of various aspects of computer science and mathematics.

The emphasis in these notes is pedagogical. Specifically, this document discusses a powerful technique due to Borwein, Bailey and Plouffe (BBP) [1] for calculating *any* digit of π in roughly linear time without computing the intervening digits. Remarkably, an individual with a high school grounding in mathematics and calculus (as is currently offered in many schools) will be able to follow the arguments. Certainly most students at the end of their first year in a Computer Science or Electrical Engineering degree program will find the text eminently comprehensible. Moreover, with rudimentary skills in a computer language such as MATLAB or Python, code may be written to perform the calculations.

2 Some mathematics - computer mathematics

Understanding how a computer works with numbers is important and will be used in future sections. The presentation that follows is minimalistic with just enough detail to understand the main arguments of this presentation.

2.1 Decimal numbers

Decimal numbers are the workhorse of science and engineering. This is not a tutorial on the decimal number system in any detail, rather definitions and key concepts are discussed – the intent being to get to to the destination with just the necessary amount of knowledge. While there will be occasions when a circuitous route will be taken, it will always be with the goal in mind.

Definition 2.1. *The* radix *or* base *of a number system is the number of symbols used to represent numbers.*

The radix of the decimal system is 10 because it uses the symbols $0, 1, 2, \ldots 9$ to represent a number. For octal, the radix is 8, for binary (next subsection) it is 2 $(0, 1)$ etc. There is also the notion of a *radix point* that permits separation between integers and fractions.

Any exposition on decimal numbers include the units, tenths, hundredths, thousandths etc. places. For example the decimal number 7942 may be viewed in the following manner.

7	9	4	2
$\times$	$\times$	$\times$	$\times$
10^3	10^2	10^1	10^0
$10^3 \times 7$	$10^2 \times 9$	$10^1 \times 4$	$10^0 \times 2$

The above table emphasizes the notion that the number is $1000 \times 7 + 100 \times 9 + 10 \times 4 + 1 \times 2$ or $7000 + 900 + 40 + 2 = 7942$. Sometimes, in order to make things clear a number will have its radix as a subscript to distinguish between other radices, e.g., $6325_{10} \neq 6325_8$.

2.2 Binary numbers

The radix/base of the binary system of numbers is 2 and the symbols employed are $\{0, 1\}$, also known as binary digits or *bits*. What does a binary number look like? It must use the symbols $0, 1$ and these symbols only. For example a binary number is 1101. Again, to help disambiguation from the decimal number 1101 (one thousand one hundred and one) it may written as 1101_2.

Humans use decimal numbers in everyday calculation, yet computers are most efficient when using binary since the state of many of the devices used in modern machines may be "on/off" or 0/1. This begs the question – if humans like decimal[1] and computers prefer binary – how do we convert from one to the other and vice–versa?

The following example will make clear how to convert a binary number (say, 1111100000110_2) into decimal. The binary number is written below along with its associated powers of two. The

[1]This is perhaps not universal since the ancient Babylonians used a base-60 number system. The continued use of hours, minutes and seconds in measuring angles is another example.

left–most bit, 1, also known as the most significant bit (MSB) is associated with 2^{12}, the next bit to the right (also a 1) is associated with 2^{11} and so on until we reach the least significant bit (LSB) which is associated with $2^0 = 1$. Notice the similarity (and difference) in binary representation and its association with powers of two and the decimal representation and the association with powers of ten.

1	1	1	1	1	0	0	0	0	0	1	1	0
$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
4096	2048	1024	512	256	0	0	0	0	0	4	2	0

To obtain the decimal number from the binary we just have to follow the multiplication pattern above, i.e., $2^{12} \times 1 + 2^{11} \times 1 + 2^{10} \times 1 + 2^9 \times 1 + 2^8 \times 1 + 2^7 \times 0 + 2^6 \times 0 + 2^5 \times 0 + 2^4 \times 0 + 2^3 \times 0 + 2^2 \times 1 + 2^1 \times 1 + 2^0 \times 0 = 4096 + 2048 + 1024 + 512 + 256 + 4 + 2 = 7942$. Hence, $1111100000110_2 = 7942_{10}$. this process of binary to decimal conversion is cometimes called *decoding*.

The reverse process, i.e., decimal to binary conversion, also known as *encoding*, involves repeated division. Since the answer is already known 7942_{10} will be converted. The table below illustrates the process with repeated division by two with the last line showing remainders (either 0 or 1). The results are obtained in reverse order, i.e., the LSB first and MSB last.

7942	3971	1985	992	496	248	124	62	31	15	7	3	1
$\div$	$\div$	$\div$	$\div$	$\div$	$\div$	$\div$	$\div$	$\div$	$\div$	$\div$	$\div$	$\div$
2	2	2	2	2	2	2	2	2	2	2	2	2
0	1	1	0	0	0	0	0	1	1	1	1	1

Below is a MATLAB function, `MyDec2BinConvert`, that takes a positive decimal integer and converts it to a binary representation using the process described above. As explained, the output is reversed though the function may be modified to print it in conventional order. Although MATLAB has its own converter (`de2bi`), the code `MyDec2BinConvert` will become important in a later part of the document, so an understanding of the coding of the conversion process is crucial.

```matlab
function [ ] = MyDec2BinConvert(n)
% Obtain the binary representation of a decimal integer

% Number of bits needed to represent the number n
% ndigit = floor(log2(n)) + 1 - this may be used for
% loop control as opposed to the while loop below.

while (n > 0)
    if (mod(n,2) == 1)
        disp('1')
    else
        disp('0')
    end

    n = floor(n/2);
end
```

Binary addition and multiplication[2] follows very simple rules. The tables show addition (left) and multiplication (right)

+	0	1	
0	0	1	
1	1	0	$\rightarrow$ carry 1

$\times$	0	1
0	0	0
1	0	1

For binary addition of ones the result is 0 with a carry over of 1 as indicated in the table.

In subsequent sections multiplication of binary numbers by two will be needed - there is a special structure to this multiplication.

$$\begin{array}{lcccccccccl}
 & a_1 & a_2 & a_3 & a_4 & a_5 & & \dots & a_n & & \\
 & \times & & & & & & 1 & 0 & & \\
\hline
 & & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & (n \text{ zeros}) \\
 & a_1 & a_2 & a_3 & a_4 & a_5 & & \dots & a_n & & \\
\hline
 & a_1 & a_2 & a_3 & a_4 & a_5 & & \dots & a_n & 0 &
\end{array}$$

So multiplication by $2_{10} = 10_2$ leads to a left shift of the entire number with a 0 added to the end, i.e., the MSB of the first number is a_1 and it is multiplied by 2^{n-1}, whereas in the result the MSB is a_1 and it is multiplied by 2^n. The LSB is a_n in the first number and 0 in the result.

When multiplying mixed binary numbers (i.e., the number has integer and fractional parts) with two, the binary point moves one point to the right[3].

Example: Multiply this binary number $11.\underline{0010}\ \underline{0100}\ \underline{0011}\ \underline{1111}\ \underline{0110}\ \underline{1010}\ \underline{1000}\ \underline{1000}$ by 2^7.

This result is $110010\ 010.\underline{0}\ \underline{0011}\ \underline{1111}\ \underline{0110}\ \underline{1010}\ \underline{1000}\ \underline{1000}$, i.e., the binary radix point moves seven places to the right.

2.3 Hexadecimal numbers

Consider any four–bit number $XXXX_2$ where each bit could be either 0 or 1. The total number of numbers that can be represented by four bits is $2^4 = 16$ and the largest number representable

[2]Subtraction and division are also easy, but not needed in this document
[3]Similar to multiplying decimal numbers by ten

is $2^4 - 1 = 15$. In the hexadecimal ("hex," meaning six and "decimal," meaning ten) system the radix or base is sixteen and its symbols are $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$. A table showing conversions between hex, decimal and binary is below.

Decimal	Hexadecimal	Binary	Decimal	Hexadecimal	Binary
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	10	A	1010
3	3	0011	11	B	1011
4	4	0100	12	C	1100
5	5	0101	13	D	1101
6	6	0110	14	E	1110
7	7	0111	15	F	1111

Hexadecimal converts readily to binary, and vice–versa, with no need for multiplications or other mathematical operations. Consider $7E0_{16}$. Simply use the conversion table provided above to convert each hexadecimal number to its binary equivalent, e.g.,

Hex	7	E	0
Binary	0111	1110	0000

For the reverse process group the binary numbers in tetrads from the LSB adding zeros, if necessary, at the MSB, and follow that by a mapping from the table above. For example, 10011100000_2 may be grouped as $\underline{100}$ $\underline{1110}$ $\underline{0000}$ and the conversion to $4E0_{16}$ follows trivially.

Grouped binary	100	1110	0000
Augmented binary	$\underline{0}100$	1110	0000
Hex	4	E	0

Mixed numbers are also easily converted from binary to hex and vice versa. The long binary number $11.\underline{0010}\ \underline{0100}\ \underline{0011}\ \underline{1111}\ \underline{0110}\ \underline{1010}\ \underline{1000}\ \underline{1000}_2$ is represented as 3.243F6A88 in base sixteen. The reverse process is left as an exercise for the reader.

Conversions from hex to decimal and decimal to hex follow the methodology outlined for binary→decimal/decimal→binary, i.e., repeated use of factors of sixteen instead of two. For example the hex number $3BA7_{16} = 15,721_{10}$. Details are shown below - note that the numbers in the last row are added together to give the final result. The MATLAB function `hex2dec` may be used for this purpose.

3	B	A	7
$\times$	$\times$	$\times$	$\times$
16^3	16^2	16^1	16^0
3×4096	11×256	10×16	7×1

As with binary→decimal conversions, the binary→hexadecimal conversion requires repeated division by sixteen while the remainders provide the decimal digits which are to be converted to hex. Also, the hex number is output in reverse order, i.e., the most significant hex digit is on the right and the least significant hex digit is on the left.

15271	954	59	3
$\div$	$\div$	$\div$	$\div$
16	16	16	16
7	10	11	3
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
7	A	B	3

3 Some mathematics - number theory

This section serves to introduce some useful elementary number theory in a self contained manner. As with the rest of the document, the goal is to extract the most useful concepts efficiently. For more details the reader may wish to refer to the vast corpus of textbooks on the subject.

3.1 The mod function

The $\pmod{}$ function is the workhorse of number theory yet only involves simple mathematical concepts. But first the basis of the $\pmod{}$ – divisibility.

Definition: $\forall a, b, k \in \mathbb{Z}$ [4], a divides b (or b is divisible by a) if and only if $b = ka$. We write this as $a|b$ where ’“|” is taken to mean “divides” or “is divisible by.” There are two important facts about divisibility that are needed.

Fact 3.1. *$\forall a, b, c \in \mathbb{Z}$, if $a|b$ and $a|c$ then $a|(b + c)$.*

The proof is easy. Let $k, l \in \mathbb{Z}$ then $b = ka$ and $c = la$. Adding $b + c = (k + l)a$ or by definition $a|(b + c)$ since $k + l \in \mathbb{Z}$.

Fact 3.2. *$\forall a, b, c, j, k \in \mathbb{Z}$, if $a|b$ and $a|c$ then $a|(jb + kc)$.*

It is easy to show that $a|jb$ and $a|kc$. Let $jb = l$ and $kc = m$ ($l \in \mathbb{Z}$ and $m \in \mathbb{Z}$). So $a|l$ and $a|m$. Using Fact 3.1 we know $a|(l + m)$ or $a|(jb + kc)$.

[4] The notation $\mathbb{Z}$ is an abbreviation for all integers, i .e., $\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$. In the language of set theory, $\mathbb{Z}$, is the set of all integers and $\in$ means “belongs to.” Thus $a \in \mathbb{Z}$ is shorthand for the statement “a is an integer.”

The floor of $x \in \mathbb{R}$, denoted $\lfloor x \rfloor$ is the greatest integer less than or equal to x. For example $\lfloor 3.14159 \rfloor = 3$ and $\lfloor -3.14159 \rfloor = -4$.

Definition 3.1. *$\forall a, b, n \in \mathbb{Z}, a \equiv b \pmod{m}$ means that b is the remainder when a is divided by m. Another way of saying this is $m|(a-b)$, i.e., m divides $(a-b)$.*

The $\equiv$ sign is sometimes replaced by the $=$ sign so we might write $a = b \pmod{m}$.We will use this in preference to $\equiv$.

Example $7 = 3 \pmod{4}$ is clearly true. What about $-1 = 3 \pmod{4}$? This may shown to be true by rewriting $4|(-1-3) = 4|(-4)$.

One may generalize the above example by asking what numbers $a \in \mathbb{Z}$ when divided by 4 will give a remainder of 3. Again, using the definition, the numbers in the series $\ldots, -5, -1, 3, 7, 11, \ldots$ all have this property.

Property 3.1. *$\forall a_1, b_1, a_2, b_2, n \in \mathbb{Z}$, if $a_1 = b_1 \pmod{n}$ and $a_2 = b_2 \pmod{n}$, then $(a_1 + a_2) = (b_1 + b_2) \pmod{n}$ or $n|(a_1 - b_1 + a_2 - b_2)$.*

From the definition $n|(a_1 - b_1)$ and $n|(a_2 - b_2)$. Using Fact 3.1 on divisibility we have $n|(a_1 - b_1 + a_2 - b_2)$.

Property 3.2. *$\forall a_1, b_1, a_2, b_2, n \in \mathbb{Z}$, if $a_1 = b_1 \pmod{n}$ and $a_2 = b_2 \pmod{n}$, then $(a_1a_2) = (b_1b_2) \pmod{n}$.*

Now $n|(a_1 - b_1)$ and $n|(a_2 - b_2)$. Using Fact 3.2 on divisibility (linear combinations) we have $n|(a_2(a_1 - b_1) + b_1(a_2 - b_2)$. Simplifying, $n|(a_1a_2 - b_1b_2)$ which by definition gives $(a_1a_2) = (b_1b_2) \pmod{n}$.

Multiplying, $a_1a_2 = (b_1 \pmod{n})(b_2 \pmod{n}) = b_1b_2 \pmod{n}$. What this is saying is, if we have $a = b \pmod{n}$ and b can be factorized into b_1, b_2 then $b_1b_2 \pmod{n} = (b_1 \pmod{n})(b_2 \pmod{n})$. This property permits modulo operations on very large integers by performing consecutive modulo operations on its factors.

Property 3.3. *If $a = b \pmod{n}$, then $b = a \pmod{n}$.*

By definition $n|(b-a)$, but it is also true that $n|(-1)(a-b)$ or $n|(b-a) \implies b = a \pmod{n}$.

Example Previously we considered $7 = 3 \pmod{4}$ which can be read as 7 divided by 4 returns a remainder of 3. Using Property 3.3 we also have $3 = 7 \pmod{4}$ which says that 3 is the remainder when 7 is divided by 4. We will use this latter form, i.e., $b = a \pmod{n}$ means that b is the remainder when a is divided by n.

3.2 Successive Squaring

Doing remainders using mod seems easy enough until the numbers get very large. Consider the following example.

Example Compute 7^{10} mod 93 and 7^{365} mod 93.

We use MATLAB for this problem.

Listing 1: Illustration of mod in MATLAB

```
>> mod(7^10, 93)

ans =

    25
```

Noting that $7^{10} = 282475249$, a remainder of 25 is obtained if this number is divided (long division) by 93. The reader is encouraged to do this. But what about the latter problem?

```
>> y = mod(7^365, 93)

y =

   NaN
```

MATLAB says that it "Not a Number". Further insight may be obtained as below:

```
>> 7^365

ans =

   Inf
```

MATLAB says the number is too large to calculate in IEEE arithmetic. Again, a determined human should be able to write the answer by repeated multiplication followed by division, but does this mean that the problem is unsolvable by computer and are we fated to have humans do these types of calculations by hand? Let us take the scenic route to answering that question.

Begin with the following series of calculations: $7^{10} \pmod{93} = 7^7 \pmod{93} \times 7^3 \pmod{93} = 7^7 \pmod{93} \times 343 \pmod{93} = 7^7 \pmod{93} \times 64 \pmod{93} = 7^7 64 \pmod{93}$. The calculation to the right of the first $=$ sign is a consequence of Property 3.2; now $7^3 = 343$ which accounts for $64 = 343 \pmod{93}$ but by Property 3.3 we can write the final result.

The entire computation may be carried out but with some details hidden so as not to make this exposition longer,

$$\begin{aligned}
7^{10} \pmod{93} &= 7^7 \times 7^3 \pmod{93} \\
&= 7^7 \times 343 \pmod{93} \\
&= 7^7 \times 64 \pmod{93} \\
&= 7^6 \times 7 \times 64 \pmod{93} \\
&= 7^6 \times 76 \pmod{93} \\
&= 7^5 \times 67 \pmod{93} \\
&= 7^4 \times 4 \pmod{93} \\
&= 7^3 \times 28 \pmod{93} \\
&= 7^2 \times 196 \pmod{93} \\
&= 490 \pmod{93}
\end{aligned}$$

Performing the final division gives a remainder of 25, exactly what MATLAB calculated. In principle $7^{365} \pmod{93}$ could be computed as above. However, one crucial observation will help make computations quicker.

Consider $7^{13} \pmod{93} = 7^8 \times 7^4 \times 7^1 \pmod{93} = 19$. Now create a table as follows

$$\begin{array}{ccccccccc}
7^1 & & & & & = & 7 & = & 7 \pmod{93} \\
7^2 & = & (7^1)^2 & = & 7^2 & = & 49 & = & 49 \pmod{93} \\
7^4 & = & (7^2)^2 & = & (49)^2 & = & 2401 & = & 76 \pmod{93} \\
7^8 & = & (7^4)^2 & = & (76)^2 & = & 5776 & = & 10 \pmod{93}
\end{array}$$

Note that only 7^8, 7^4, and 7^1 are needed to perform the computation, but in order to get to 7^8 we need to calculate 7^4. Only three multiplications involving squaring the remainders $7, 49$ and 76 are required. The mod function is also used four times.

To continue,

$$\begin{aligned}
7^{13} \bmod 93 &= 7^8 \times 7^4 \times 7^1 \bmod 93 \\
&= (7^8 \bmod 93) \times (7^4 \bmod 93) \times (7^1 \bmod 93) \\
&= (10 \bmod 93) \times (76 \bmod 93) \times (7 \bmod 93) \quad \text{(substitute from above table)} \\
&= (10 \times 76 \bmod 93) \times (7 \bmod 93) \\
&= (16 \bmod 93) \times (7 \bmod 93) \\
&= 19
\end{aligned}$$

Since we use the powers–of–two calculations above, we see that there are only three multiplications and two applications of mod. Thus the total calculation is 6 multiplications and 5 mod applications.

Example Show that calculating $7^{13} \bmod 93$ would take twelve multiplications and numerous uses of mod.

We can show that the number of multiplications to compute $a^k \bmod n$ is between $\lceil \log_2 k \rceil$ and $2\lceil \log_2 k \rceil$, whereas, the direct method would take k multiplications. For k large enough $2\lceil \log_2 k \rceil < k$, proving the superiority of the method of successive squaring.

For clarity note that the powers of two that sum up to 13 in the above calculation could explicitly be multiplied by **1** or **0**, i.e., $13 = \mathbf{1}\times 2^3 + \mathbf{1}\times 2^2 + \mathbf{0}\times 2^1 + \mathbf{1}\times 2^0 = 8+4+0+1$. Hence it should be evident that the binary expansion of 13_{10} is [1 1 0 1] . If we read this string from left to right we can clearly see which powers of 2 sum to 13 (i.e., $2^3, 2^2, 2^0$) and in in particular $2^1 = 2$ is not used in the fast calculation of $7^{13} \bmod 93$.

The naive method of squaring and the method of successive squares can be compared using the following programs/functions.

```
function [NaivePower] = NaiveSquaresModulo(a, n, m)
%NaiveSquaresModulo This just loops around n times computing
%a^n mod m

NaivePower = 1;
for i = 1:n

    NaivePower = mod(NaivePower*mod(a,m), m);

end
```

The method of successive squares is a more complicated code but still quite brief:

```
 1 function [ SSPower ] = SSquares( a, n, m )
 2 % SSquares Use the method of successive squares for powers
 3 % modulo n. Detailed explanation in the text of this document
 4 
 5 prod = 1;
 6 
 7 while (n > 0)
 8     if (mod(n,2) == 1)
 9         prod = mod(prod*a,m);
10     end
11     a = mod(a^2,m);
12     n = floor(n/2);
13 end
14 SSPower = prod;
15 
16 end
```

The reader should note that lines 8, 9 and 13 are identical with the previously presented `MyDec2BinConvert` code and that this must be the case since the exponent n is being broken into its powers of two.

Running `SSquares` and `NaiveSquaresModulo` gives,

```
>> disp([NaiveSquaresModulo(7,13,93) SSquares(7,13,93)])
```

```
    19     19
```

i.e, each code produces the right answer.

The acid test comes upon comparing both for very large powers by running, `TimeSSquaresNaiveSquares`, a test script.

```
% This script obtains CPU times of NaiveSquaresModulo and
% SSquares. SSquares is the fast method of performing squares
% modulo a number (method of successive squaring).

% Time to compute SSquares
 t = cputime;
SSquares(12,1000000000,29);
eSSquares = cputime-t

% Time to compute SSquares
t = cputime;
NaiveSquaresModulo(12,1000000000,29);
eNaiveSquares = cputime-t

sprintf('Successive Squares = %f, Naive method = %f', ...
           eSSquares, eNaiveSquares)
```

Running the program provides times in seconds.

```
>> TimeSSquaresNaiveSquares

ans =

Succesive Squares = 0.000000, Naive method = 54.085547
```

The results are remarkable - the method of successive squares is hugely faster than the naive method. Even when calculating a number to the billionth power, the clock tick in MATLAB does not have enough resolution to measure the time.

4 Some mathematics - calculus

While calculus may be forbidding for some readers, it is absolutely essential in order to understand key concepts in subsequent calculations. Series (McLaurin, geometric) are required, followed by elementary integral calculus. The three mathematical sections (computer mathematics, number theory and calculus) are combined to describe an algorithm to calculate, first, $\ln 2$ and subsequently π.

4.1 Series

The Taylor series[5] of $f(x)$ about a is defined as

$$\begin{aligned} f(x) &= f(a) + (x-a)f^{(1)}(a) + \frac{(x-a)^2}{2!}f^{(2)}(a) + \frac{(x-a)^3}{3!}f^{(3)}(a) + \ldots + \frac{(x-a)^n}{n!}f^{(n)}(a) + \ldots \\ &= \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k \qquad (1) \end{aligned}$$

where $f^{(n)}(a)$ is the n^{th} derivative of $f(x)$ evaluated at $x = a$. The McLaurin series is the Taylor series of $f(x)$ about $a = 0$:

$$\begin{aligned} f(x) &= f(0) + xf^{(1)}(0) + \frac{x^2}{2!}f^{(2)}(0) + \frac{x^3}{3!}f^{(3)}(0) + \ldots + \frac{x^n}{n!}f^{(n)}(0) + \ldots \\ &= \sum_{k=0}^{\infty} \frac{f^{(k)}(0)}{k!}x^k \end{aligned}$$

Example: What is the McLaurin series of $f(x) = \ln(1 + x)$? First obtain the derivatives of the function and substitute in Equation (1)

$$\begin{array}{lclclcl} f^{(1)}(x) &=& \frac{1}{1+x} &\Longrightarrow& f^{(1)}(0) &=& 1 \\ f^{(2)}(x) &=& \frac{-1}{(1+x)^2} &\Longrightarrow& f^{(2)}(0) &=& -1 \\ f^{(3)}(x) &=& \frac{x}{(1+x)^3} &\Longrightarrow& f^{(3)}(0) &=& 2 \\ f^{(4)}(x) &=& \frac{-6}{(1+x)^4} &\Longrightarrow& f^{(4)}(0) &=& -6 \\ f^{(5)}(x) &=& \frac{24}{(1+x)^5} &\Longrightarrow& f^{(5)}(0) &=& -24 \\ \vdots && \vdots && \vdots && \end{array}$$

$$\begin{aligned} \ln(1+x) &= 0 + x - \frac{1}{2}x^2 + \frac{1}{3}x^3 - \frac{1}{4}x^4 + \frac{1}{5}x^5 - \ldots \\ &= \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n}x^n \end{aligned}$$

Exercise: Write a MATLAB program to evaluate $\ln(1 + x)\,|_{x=0.5}$ and check the results on a hand held calculator. This will provide practice in evaluating sums on the computer.[6]

The geometric series is starts with a constant first term and multiplies it repeatedly by x - called the common ratio. Each term is added to the sum of the previous terms. There is a

[5]This definition does not discuss *radius of convergence* for both the Taylor and McLaurin series. This is an important point, but note that all the functions in this document will satisfy standard convergence criteria.

[6]Note that calculators use a variety or series solutions (or variants thereof) to calculate mathematical functions

closed form solution to the series as shown below:

$$\begin{aligned}
S_n &= ax^0 + ax^1 + ax^2 + ax^3 + \ldots + ax^{n-1} \\
xS_n &= \phantom{ax^0 +{}} ax^1 + ax^2 + ax^3 + \ldots + ax^{n-1} + ax^n \\
S_n - xS_n &= ax^0 - ax^n \\
S_n &= \frac{a(1-x^n)}{1-x}
\end{aligned}$$

Assuming $|x| < 1$, the limit as $n \to \infty$

$$S_\infty = \sum_{i=0}^{\infty} ax^i = ax^0 + ax^1 + ax^2 + \ldots = \frac{a}{1-x} = S$$

The geometric series shows up frequently in many areas of science, engineering and computer science.

Consider the geometric series with $a = 1$. Then

$$\begin{aligned}
\frac{x^{k-1}}{1-x^8} &= \sum_{n=0}^{\infty} x^{k-1+8n} \\
\int_0^{\frac{1}{\sqrt{2}}} \frac{x^{k-1}}{1-x^8} &= \int_0^{\frac{1}{\sqrt{2}}} (\sum_{n=0}^{\infty} x^{k-1+8n}) dx \\
&= \sum_{n=0}^{\infty} (\frac{x^{k+8n}}{k+8n})\Big|_0^{\frac{1}{\sqrt{2}}} \text{ (exchanging order of integration and summation)} \\
&= \sum_{n=0}^{\infty} \frac{(\frac{1}{\sqrt{2}})^{k+8n}}{k+8n}
\end{aligned}$$

$$\therefore \int_0^{\frac{1}{\sqrt{2}}} \frac{x^{k-1}}{1-x^8} = \sum_{n=0}^{\infty} \frac{1}{2^{k/2} 16^n (8n+k)} \tag{2}$$

Exercise: Repeatedly use Equation (2) to prove that

$$I = \int_0^{\frac{1}{\sqrt{2}}} \frac{4\sqrt{2} - 8x^3 - 4\sqrt{2}x^4 - 8x^5}{1-x^8} dx = \sum_{n=0}^{\infty} \frac{1}{16^n} \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right) \tag{3}$$

The right hand side of Equation (3) can be numerically evaluated using `BBPSeriesPi`

```
function [ SumValBBP ] = BBPSeriesPi(n)

%BBPSeriesPi Sum the BBP series for pi to n terms
%   The formula may be found in "The BBP Algorithm for Pi", by
%   David Bailey, published 17 Sep. 2006. This article is online.
```

```
SumValBBP = 0;

for i = 0:n

    t1 = 8*i;
    t = 4/(t1+1) - 2/(t1+4)- 1/(t1+5)- 1/(t1+6);
    SumValBBP = SumValBBP + t/16^i;

end
```

This code produces the following output

```
>> disp([BBPSeriesPi(1); BBPSeriesPi(3); BBPSeriesPi(10)])
   3.141422466422466
   3.141592457567436
   3.141592653589793
```

The last number indicates that the series might be approaching the value of π – the trick is to prove it mathematically rather than through numerical experiments.

Start by substituting $y = x\sqrt{2}$ in the integral in Equation (3). This gives

$$I = 16\int_0^1 \frac{4\sqrt{2} - 2\sqrt{2}y^3 - \sqrt{2}y^4 - \sqrt{2}y^5}{16 - y^8}\frac{dy}{\sqrt{2}} = 16\int_0^1 \frac{4 - 2y^3 - y^4 - y^5}{16 - y^8}dy \tag{4}$$

The common factors in the numerator and denominator are $y^2 + 2$ and $y^2 + 2y + 2$ which simplifies the integral to

$$I = \int_0^1 \frac{16y - 16}{(y^2 - 2y + 1)(y^2 - 2)}dy$$

Using a partial fraction decomposition leads to two simpler integrals

$$I = \int_0^1 \frac{4y}{y^2 - 2}dy - \int_0^1 \frac{4y - 8}{y^2 - 2y + 2}dy = I_1 + I_2.$$

Exercise Work through the algebra to show that the partial fraction expansion is correct, i.e.,

$$\frac{4y}{y^2 - 2} + \frac{4y - 8}{y^2 - 2y + 2} = \frac{16y - 16}{(y^2 - 2y + 1)(y^2 - 2)}.$$

$$I_2 = -\int_0^1 \frac{4(y - 2)}{(y - 1)^2 + 1}dy = -\int_{-1}^0 \frac{4u - 4}{u^2 + 1}du \quad \text{(substitute u = y - 1)}$$

Hence,

$$I_2 = \int_{-1}^0 \frac{4}{u^2 + 1}du - \int_{-1}^0 \frac{4u}{u^2 + 1}du = I_3 + I_4.$$

Note that I_3 is a well known antiderivative of $\tan^{-1} u$[7] with an extraneous factor of four.

$$I_4 = -2\int_{-1}^{0} \frac{2u}{u^2+1}du = -2\int_{2}^{1} \frac{dt}{t} \quad (\text{put } u^2+1=t) = -2\,\ln t|_2^1 = 2\ln 2.$$

So I_2 has been evaluated.

It remains to evaluate I_1 which can be achieved by substituting $y^2 - 2 = t$ to get a somewhat familiar form

$$I_1 = \int_0^1 \frac{4y}{y^2-2}dy = 2\int_{-2}^{-1} \frac{dt}{t},$$

which reminds us of I_4. A further substitution of $p = -\frac{1}{t}$ makes I_1 amenable to integration

$$I_1 = -2\int_{\frac{1}{2}}^{1} \frac{p}{dp} = -2\ln 2.$$

Therefore, $I = I_1 + I_2 = I_1 + I_3 + I_4 = -2\ln 2 + \int_{-1}^{0} \frac{4}{u^2+1}du + 2\ln 2 = 4\,\tan^{-1} u\big|_{-1}^{0} = 0 - 4\frac{\pi}{4} = \pi.$

So through a series of substitutions and manipulations we have

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

The numerical value of the LHS was evaluated using MATLAB (i.e., `3.141592653589793`) and it is now proven mathematically.

5 Putting it all together

This paper has introduced the reader to number systems (decimal, binary, hexadecimal), elementary number theory - culminating in the method of Successive Squares and the derivation of a formula for π through simple integral calculus. Now it is time to put it all together to calculate π to (say) the millionth hexadecimal digit *without* calculating any of the intervening digits. Again, in order to explain the concepts, we take a detour through an easier problem.

5.1 Binary digits of $\ln 2$

We know from our discussion of McLaurin series that

$$\ln(1+x) \quad = \quad \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n} x^n$$

[7] Easily obtained from tables or in countless elementary calculus books

Substitute $x = -\frac{1}{2}$ to get the well known infinite series

$$\ln 2 = \sum_{n=1}^{\infty} \frac{1}{n2^n}.$$

Consider $d \geq 1$, then

$$\begin{aligned}(2^d \ln 2) \bmod 1 &= \left(\sum_{n=1}^{\infty} \frac{2^{d-n}}{n}\right) \bmod 1 \\ &= \left(\sum_{n=1}^{d} \frac{2^{d-n}}{n} + \sum_{n=d+1}^{\infty} \frac{2^{d-n}}{n}\right) \bmod 1\end{aligned}$$

Read Equations (6) from left to right: Multiplying $\ln 2$ by 2^d is equivalent to moving the binary point of the binary representation of $\ln 2$ by d places[8]. Taking mod 1 of the result is equivalent to keeping the remainder and discarding everything before the binary point. Equivalently the series representation of $\ln 2$ should be multiplied by 2^d and the remainder taken. Subsequently the sum in question is divided into two parts – a "head" sum and a "tail" sum. Each term in the head sum is typically greater than unity while it is guaranteed that each term in the tail sum is less than unity. Thus the equation becomes

$$(2^d \ln 2) \bmod 1 = \left(\sum_{n=1}^{d} \left(\frac{2^{d-n}}{n}\right) \bmod 1 + \sum_{n=d+1}^{\infty} \frac{2^{d-n}}{n}\right) \bmod 1 \tag{6}$$

What would happen if the numerator in the head sum were replaced by $2^{d-n} \bmod n$? Effectively, one would obtain the remainder of 2^{d-n} when it is divided by n and this is precisely what is required, i.e., we want to drop all terms before the binary point.

$$(2^d \ln 2) \bmod 1 = \left(\sum_{n=1}^{d} \left(\frac{2^{d-n} \bmod n}{n}\right) \bmod 1 + \sum_{n=d+1}^{\infty} \frac{2^{d-n}}{n}\right) \bmod 1 \tag{7}$$

A formal proof of the above can be accomplished by the following argument

$$\frac{a}{b} \bmod 1 = \left(\lfloor \tfrac{a}{b} \rfloor + \frac{a \bmod b}{b}\right) \bmod 1 = \left(\frac{a \bmod b}{b}\right) \bmod 1$$

where $\lfloor \frac{a}{b} \rfloor$ represents the integer part of $\frac{a}{b}$, but this is always an integer ≥ 0 whereas $\frac{a \bmod b}{b}$ is the part of $\frac{a}{b}$ after the decimal point which is what was required in the first place.

This might seem like hair splitting but the specific formulation

$$\left(\frac{2^{d-n}}{n}\right) \bmod 1 = \left(\frac{2^{d-n} \bmod n}{n}\right) \bmod 1$$

[8]See section on binary arithmetic

permits invocation of the successive squares algorithm for the right hand side where the numerator never exceeds n whereas the left hand side involves high powers of two followed by a division by n.

Back to Equation (7). The head sum will have d terms (say $d = 10^6$) each involving large powers of 2, but by using the succesive squares algorithm the numerator will never get very large and each term in the sum can be calculated very quickly. The tail sum involves ever larger numbers (i.e., $\frac{1}{n2^{n-d}}, n > d$) in the denominator and so need not go to ∞, only a few terms will suffice – upto the machine accuracy to be precise.

And that in a nutshell is the algorithm to calculate the $d + 1^{th}$ binary digit of $\ln 2$. This can easily be coded as shown in function `Log2Binary` below. Note that this presumes the existence of a code to convert fractional decimal numbers to binary `Fr_dec2bin`.[9]

```
function [ BinDigitsLog2 ] = Log2Binary( StartDigit )
% Compute the binary digit of log(2) starting at StartDigit.
% Details are in David H. Bailey, "The BBP Algorithm for Pi",
% Sep. 17, 2006

% First the head sum

tmpsum1 = 0;

for i = 1:StartDigit
 SSPower = SSquares(2, StartDigit-i, i);
 tmpsum1  = mod(tmpsum1 + SSPower/i,1);
end

% Now the tail sum. Error is proportional to 1/(j*2^(j-digit))

tmpsum2 = 0;

for j = StartDigit+1:StartDigit+15
    tmpsum2 = tmpsum2 + 1/(j*2^(j-StartDigit));
end

tmpsum = mod(tmpsum1+tmpsum2,1);

%Convert to binary and display binary numbers {0 1} from
%StartDigit digit onward
[BinDigitsLog2, str_Fr, Fr_dec] = Fr_dec2bin (tmpsum);

end
```

[9] I obtained `Fr_dec2bin` from the internet.

The code below, `Log2Tests`, is used to time each invocation of `Log2Binary` as well as to show one subtle aid in debugging any code of its nature.

```
% Log2Tests.m
% Compute how long it takes to calculate the StartDigit bit in
% the binary representation of log(2). StartDigit numbers
% considered: 8, 100, 1000, 1^4, 10^5, 10^6, 10^6+1, 10^7.
% Use the previously written function Log2Binary. Computing at
% 10^6 and 10^6 + 1 is a way of debugging the code
%
% NOTE: When 8 (or d) is input to Log2Binary, what is
% returned is the (d+1)th binary digit.
%
% Use t=cputime; compute binary digits of log(2); cputime-t

StartDigitList = [8 100 1000 10^4 10^5 10^6 10^6+1 10^7];

size(StartDigitList);

j             = 1;
EndT          = zeros(size(StartDigitList));

for i = StartDigitList

    StartT              = cputime;
    tmp                 = Log2Binary(i);
    EndT(j)             = cputime - StartT;

    str = sprintf('%d %8d %s %d', j, i+1, tmp(1:18), EndT(j));
    disp(str);

    j = j + 1;

end

% Produce plots

SubsetArr = [1 2 3 4 5 6 8];
plot(log10(StartDigitList(SubsetArr)),EndT(SubsetArr),'b-o')
title('Time to compute ln(2)')
xlabel('(d+1)th digit of ln(2) - log scale')
ylabel('Time (sec)')
```

Runing `Log2Tests` (which runs `Log2Binary`) produces results that can be analyzed for run times. Consider the output for the 9^{th} binary digit of π – it is a zero and the 10^{th}, 11^{th} and 12^{th}, are all ones. Now the decimal expansion of $\ln 2 = 0.6931471805$ which gives, upon conversion,

$\ln 2 = 0.10110001\underline{0111}0010$ clearly identifying the 9th through the 12th bits. Note that the output for 10^6 and $10^6 + 1$ are identical except that the bits of the latter are shifted to the left by one. This fact can be used as a debugging tool. Also note that the code does not just produce the $(d+1)^{th}$ digit of $\ln 2$ but several additional digits that depend on the precision of the floating point arithmetic implementation (typically 64- or 32-bit).

```
>> Log2Tests
1         9 0.0111001000010111 0
2       101 0.0011111100101111 0
3      1001 0.0011111111010101 0
4     10001 0.0101101101100000 6.240040e-002
5    100001 0.0100101100111111 2.964019e-001
6   1000001 0.1010100100100011 3.510023e+000
7   1000002 0.0101001001000111 3.432022e+000
8  10000001 0.1011100101100110 3.878185e+001
```

The graph (Figure 1) shows the times taken by `Log2Binary` to obtain the $(d+1)^{th}$ of $\ln 2$ where $d \in \{8 \ \ 10^2 \ \ 10^3 \ \ 10^4 \ \ 10^5 \ \ 10^6 \ \ 10^6+1 \ \ 10^7\}$. Note that the x–axis is scaled by $\log_{10}$, hence the graph does not look like a straight line – however examination of the data (above) shows the linear trend quite clearly. One would anticipate that the 100–millionth digit of $\ln 2$ would take somewhat over six hours.

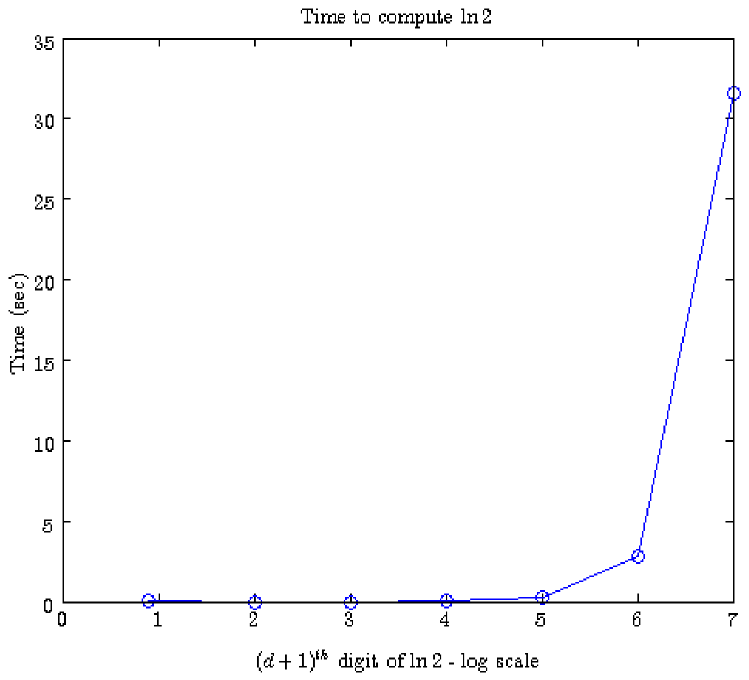

Figure 1: Time taken to calculate the $(d+1)^{th}$ bit of $\ln 2$

5.2 Finally – The BBP formula for π

The previously derived formula for π is repeated below for convenience.

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

This formula was first derived by Bailey, Borwein and Plouffe (BBP) [1] in an entirely different manner using considerably more complex techniques than used here. Subsequently BBP also provided an elementary version of the proof – the version in this paper provides details missing in the original publications.

To make things simpler write Equation (8) as

$$\pi \quad = \quad 4S_1 - 2S_4 - 1S_5 - 1S_6$$

where,

$$S_j = \sum_{i=0}^{\infty} \frac{1}{16^n(8n+j)}, \quad j \in \{1 \;\; 4 \;\; 5 \;\; 6\}$$

To get the $(d+1)^{th}$ hexadecimal digit of π simply multiply Equation (8) by 16^d, followed by mod1, to get

$$\begin{aligned} 16^d\pi \bmod 1 \quad &= \quad (4S_1 - 2S_4 - 1S_5 - 1S_6) \bmod 1 \\ &= \quad \Big(4 \times (16^d S_1) \bmod 1 \,-\, 2 \times (16^d S_4) \bmod 1 \\ &\qquad\quad -(16^d S_5) \bmod 1 \,-(16^d S_1) \bmod 1\Big) \bmod 1 \end{aligned} \tag{9}$$

It should be evident from our computations with $\ln 2$ that each term in Equation (9) should be broken into a head sum and a tail sum. Examine S_1 in detail.

$$\begin{aligned} (16^d S_1) \bmod 1 \quad &= \quad \left(\left(\sum_{n=0}^{d} \frac{16^{d-n}}{8n+1}\right) \bmod 1 + \sum_{n=d+1}^{\infty} \frac{16^{d-n}}{8n+1}\right) \bmod 1 \\ &= \quad \left(\sum_{n=0}^{d}\left(\frac{16^{d-n} \bmod 8n+1}{8n+1}\right) \bmod 1 + \sum_{n=d+1}^{\infty} \frac{16^{d-n}}{8n+1}\right) \bmod 1 \qquad (10) \\ &\quad\;\; \text{or more generally} \quad (j \in \{1 \;\; 4 \;\; 5 \;\; 6\}) \\ (16^d S_j) \bmod 1 \quad &= \quad \left(\sum_{n=0}^{d}\left(\frac{16^{d-n} \bmod 8n+j}{8n+j}\right) \bmod 1 + \sum_{n=d+1}^{\infty} \frac{16^{d-n}}{8n+j}\right) \bmod 1 \qquad (11) \end{aligned}$$

There are four such sums, the one described above for S_1 and three others for S_4, S_5 and S_6. At the end of the calculation of the sums, the number will be converted to hexadecimal and the first digit will be the required one.

5.3 What to do?

Since this write up culminates in an explanation of the BBP algorithm to calculate any digit of π the reader should write MATLAB programs to perform this using `Log2Binary` as a guide. The decimal to hexadecimal code is easy to write and there should be a separate function for this.

It should be clear that calculating all the hexadecimal digits of π upto d is an easily parallelizable problem. Simply have one computer calculate the digits of π from 10^6–$10^6 + M_1$, the second may calculate from $10^6 + M_1 + 1$–$10^6 + M_2$ and so on. M_1, M_2 etc. may be optimized so that each computer spends approximately the same time doing its computations.

6 Conclusion

This document introduces the reader to the elementary mathematics needed to calculate π to any arbitrarily large digit. The exposition attempts to show the connection between number theory, calculus and computer arithmetic in the solution of this problem. Take–home points are

1. There exist *linear*–time algorithms to calculate *any* hexadecimal digit of π without computing the intervening digits
2. The computational (CPU) resources required for this problem are modest – a simple home computer will do
3. The main memory requirements are similarly modest
4. The problem is very parallelizable – so all the digits from the first through the d^{th} may be calculated on multiple computers with only a small amount of additional code
5. The algorithm is simple to understand and easy to program

It is important to note that the BBP algorithm is far from being the fastest π algorithm – however these other methods require considerably more complex mathematics (e.g., Fast Fourier Transforms) and programming techniques.

References

[1] D. H. Bailey. On the rapid computation of various polylogarithmic constants. *Mathematics of Computation*, 66(218):903–913, 1997.

[2] A. S. Posamenter and I. Lehman. *Pi: A Biography of the World's Most Mysterious Number*. Prometheus Books, Amherst, NY, 2004.