Perils of Subtraction: a New Language for an Old Algorithm

by Donald B. McIntyre

APL and the Classical Method for Computing Pi

The symbol *pi*, π, was chosen in 1706 [12; 3] to represent the length of the *perimeter* of a circle with unit diameter. The classical way of obtaining its value was given by Archimedes about 240BC, making this one of the oldest known algorithms [1; 2; 13; 16; 17; 18]. An inscribed hexagon has six sides, each of length one half; consequently pi is greater than 3, which is the perimeter of the hexagon. A circumscribed hexagon provides an upper bound.

Doubling the number of sides of the polygon gives a closer approximation, and successive doublings, with a larger and larger number of smaller and smaller sides, converges on the true value for the perimeter of the circle. The length of a new side, after doubling the number, is found by two applications of Pythagoras' theorem, which was well-known to Archimedes. If the length of the old chord is s, then the length of the new chord is given by:

$$\sqrt{\{1/2\ (1 - \sqrt{(1 - s^2)})\}}$$

The side of the 12-sided figure resulting from doubling is given in APL as follows:

```
      y← 0.5                    Side of the inscribed hexagon
      (0.5 × 1 - (1 - Y*2)*0.5)*0.5
0.25889

      ∇ Z← CH Y                 Function defining the new chord length
[1]   Z←(0.5×1-(1-Y*2)*0.5)*0.5
      ∇

      CH CH 0.5                 Length of the new side after 2 doublings
0.130526
```

With 8 doublings the approximation to pi is:

```
    6 ×(2×2×2×2×2×2×2×2)× CH CH CH CH CH CH CH CH 0.5
3.14159
```

Using APL notation for exponentiation:

```
      6 ×(2*8) × CH CH CH CH CH CH CH CH 0.5
```

We need an equivalent operation for repeated application of *CH*. Direct Definition (introduced to APL in 1976 [9; 14]) is concise, and leads naturally to Iverson's recent extension called J [10; 11; 15]. *CH* is written as follows. A colon separates the name of the function from the body, and reference to arguments is by the symbols α and ω:

```
CH:(0.5×1-(1-ω*2)*0.5)*0.5
```

The recursive function *C* applies *CH* to the right argument the number of times given by the left argument. The body, divided into three parts by two colons, should be read *execute the left-hand expression unless the proposition given in the middle expression is true, in which case execute the right-hand expression*:

```
      C:(α-1) C CH ω : α=0 : ω
      2 C .5
0.130526
```

After eight doublings the estimate of pi is:

```
      PI:6 × (2*ω) × ω C .5
      PI 8
3.14159
```

From APL to J

To convert from APL to J, replace three APL symbols by ASCII equivalents:

```
Y← 0.5                                  In APL
(0.5×1-(1-Y*2)*0.5)*0.5
y=. 0.5                                 In J
(0.5*1-(1-y^2)^0.5)^0.5
%: 0.5*1- %: 1-y^2                      Rewrite without parentheses:
                                        %: is square root
%:-: 1- %: 1- *:y                       -: is half and -. is 1-
%:-:-.%:-.*:y
```

Now define a verb (function) to compute the length of a new chord:

```
ch=. 3 : '%:-:-.%:-.*: y.'
```

This is an *explicit* definition, because `y.` represents the (right) argument. `ch` is a verb because `3` is the sum of possible valences (`0` for a noun, `1` for an adverb, and `2` for a conjunction).

```
      ch 0.5
0.258819
```

With a left argument of 13 this automatically translates to *tacit* definition:

```
     13 : '%:-:-.%:-.*: y.'
[: %: [: -: [: -. [: %: [: -. *:
```

The result is a *train* of 11 verbs, parsed from right to left as a sequence of *forks*. In this case a tacit definition can also be written using the *atop* conjunction (@):

```
ch=. %:@-:@-.@%:@-.@*:
```

One Verb Under Another

Before taking the next step, consider a simpler example. First of all straight execution:

```
     %: +/ *: 3 4      The square root of the sum of squares (Pythagoras).
5
```

Using names helps us generalize. The conjunction [. returns its left argument (a verb).

```
     f=. %: [. g=. +/ [. h=. *:
     f@g@h 3 4                    Conjunctions compose a new verb
5
     (] -: f@h) y=. i.4           These match; i.e. f is the inverse of h.
1
     g&.h 3 4                     g under h; i.e. apply f to the result of g@h
5
```

Another example. Find the shape of each boxed item (> opens a box and < closes it):

```
(<@$@> -: $&.>) 1 2 3; i.2 4      Match. Shape under open.
```

%: is the inverse of *:, so we rewrite ch as follows — the parentheses are not needed

```
ch=. %: @(-:@-.@%:@-.)@ *:
ch=. (-:@-.@%:@-.)&.*:
ch=. -:@(-.@%:@-.)&.*:
```

As -: is its own inverse, write:

```
     ch=. -:@(%:&.-.)&.*:          Parentheses are now required.
     ch 0.5
0.258819
```

The Power Conjunction with a Gerund: a New Way for Iteration

An elegant feature of J is that, just as we write (2^y), so we can write (ch^:y); thus:

```
      pi=. 3 : '6*(2^y.)*(ch^:y.) 0.5'
      pi 8
  3.14159
```

The tacit form is more subtle: one verb (v1) gives the number of times ch is to be applied, while another (v2) gives the value of ch's initial argument. The two are tied together to form a *gerund*. In this case v2 always returns 0.5.

```
  ch^:(v1`v2) y    ↔  ch^:(v1 y) (v2 y)      The definition
   g=. ]`(0.5"_)   The rank conjunction converts a noun into a verb of given rank
   pi=. 6: * 2&^ * ch^:g
   pi"0 i.9        The rank conjunction applies pi to the rank-0 cells of i.9
3 3.10583 3.13263 3.13935 3.14103 3.14145 3.14156 3.14158 3.14159
```

But Now for the Problem!

Unfortunately the results of further doublings now become worse and worse [15, pp.563-564]. After 28 doublings — the number of sides is 1,610,612,736 — the computed value of pi is zero!

```
      pi"0 [ 24 25 26 27 28
  3.18198 3.3541 4.24264 6 0
```

To understand what goes so wrong, we must know how numbers are stored and manipulated.

How Numbers are Stored in an IBM System/360 and its Descendants

Floating-point numbers have three parts: a normalized fraction; an exponent (characteristic); and a sign. In the IBM System/360 and its descendants the leftmost byte contains the sign (counting from the left, this is bit 0) and the exponent (bits 1-7).

The fraction (3 bytes in short form and 7 in long) is expressed in *hexadecimal digits*, each a group of 4 bits (a *nibble*). The radix point is to the left of the fraction, which is normalized by shifting it until the first significant hexadecimal digit occupies bits 8-11. Because normalization is by *hexadecimal* digits, bits 8-10 may all be zero.

In a decimal fraction, the value assigned to a digit is determined by the place-value of the digit's position. Immediately to the right of a decimal point we have tenths; so in a hexadecimal fraction that is where we have sixteenths. The value corresponding to a digit represented by bits 8-11 is illustrated by the following examples:

```
bit    8    9    10    11
       ------------------
       0    0    0     1        1/16
       0    0    1     0        2/16          1/8
       0    1    0     0        4/16          1/4
       1    0    0     0        8/16          1/2
       1    1    1     1       15/16
```

Each time the fraction is shifted one hexadecimal digit right or left during normalization, the exponent changes to record the shift. The exponent is a 7-bit binary number. It is the power, **n**, to which the base (**16**) must be raised so that the product of the fraction and $\mathbf{16^n}$ equals the absolute value of the number represented. If the exponent is 0, the values of the fraction are multiplied by $16^0 = 1$ and are unchanged. If the exponent is 1, the values of the fraction are multiplied by $16^1 = 16$ and become 1 2 4 8 and 15. If the exponent is 2, the values of the fraction are multiplied by $16^2 = 256$ and become 16 32 64 128 and 240.

In order to provide for negative exponents, 64 (binary 100 0000) is added to all exponents. This is called *excess-64* representation.

```
411c  ↔0100 0001 0001 1100 ↔ 16^1 (1/16^1 + 12/16^2)  ↔  1.75

c21d5a ↔1100 0010 0001 1101 0101 1010
c21d5a ↔ -(16^2) (1/16^1 + 13/16^2 +5/16^3+10/16^4)  ↔  -29.3516
  -(16^2)*(1 13 5 10 +/ .% 16^1 2 3 4)    Executing in J
_29.3516
```

Translate from hexadecimal floating-point representation to decimal:

```
dfh=. '0123456789abcdef'&i.              Decimal from hexadecimal
bfh=. [: , 2 2 2 2"_ #: dfh              Binary from hexadecimal
fs360=. _1: ^ {.@bfh                     Floating-point 360 sign
  g=. #.@}.@bfh@(2&{.)
fe360=. 16"_ ^ g - 64"_                  Floating-point 360 exponent
  f=. 16"_ ^ >:@i.@#
ff360=. (+/ .% f)@dfh@(2&}.) f.          Floating-point 360 fraction (Hook)
fp360=. fs360 * fe360 * ff360
```

It is convenient to convert trailing blanks to zeros. `zfb` uses *item amend* to convert any blanks to zeros. First laminate a row of 0s to make a table of two rows. Select from row 1 unless there is a blank, in which case select from row 0. Parentheses

are not needed; they merely draw the reader's attention to the verb to which the adverb *amend* (}) applies.

```
   zfb=. [: (~:&' '@{:)} '0'&,:        Zeros from blanks
   dfh360=. [: fp360 [: zfb 16&{.      Decimal from 360 hexadecimal
```

Because the blanks are trailing, the *fit* conjunction (!.) can control the padding:

```
   pad=. 16&({.!.'0')        !. modifies {. by specifying the fill element
   dfh360=. fp360@pad
   dfh360 'c21d5a'
_29.3516
```

To deal with a list of hexadecimal strings, append a blank character to define the *fret* and use the *cut* conjunction (;.); for example:

```
   <;._2 '428 424 c21d5a '
```

428	424	c21d5a

```
   ];._2 '428 424 c21d5a '
428
424
c21d5a
   pad;._2@(,&' ')'428 424 c21d5a'
4280000000000000
4240000000000000
c21d5a0000000000
```

Incorporate this in the verb `dfh360` (parentheses are required):

```
   dfh360=. ,@(fp360"1@table@pad;._2@(,&' '))
   dfh360=. [: , [: fp360"1 pad;._2@(,&' ')
   dfh360 '41c 414 411 c11 411c c21d5a'
12 4 1 _1 1.75 _29.3516
```

How Numbers are Stored in a PC

On the PC, J uses the *double extended* form of 64 bits: the sign is bit 0, the exponent bits 1-11, and the fraction bits 12-63 [7]. Unlike 360 architecture, both exponent and fraction are binary. Because normalized binary numbers always begin with a 1, there is no need to store it. As usual, the exponent keeps a record of shifts made by the fraction, but in this case the count is binary. 1023 is added to allow for negative exponents.

```
   <:2^10                    #:1023
1023                     1 1 1 1 1 1 1 1 1 1
```

Evaluation of the 64-bit binary string is therefore:

```
(_1^sign)*(2^(2#.exp)-1023)*1+(2#. fr) % 2^# fr
```

s, e, and f extract sign, exponent, and fraction respectively from the binary representation.

```
s=. _1: ^ {.  [.   e=. [: }. 12&{.   [.   f=. 12"_ }. ]
    exp=. [: 2&^ #. - 1023"_          Note that the dyad #. is 2&#.
    fr=. [: >: #. % 2&^@#
    dfbj=. (s * exp@e * fr@f) f.      Decimal from binary: J on PC
    dfhj=. dfbj@bfh                   Decimal from hexadecimal representation
    dfhj '400921f9'                   Compare ir o.1 below
3.14159
```

Internal representation is obtained by the *foreign conjunction* `!:`. When used as `3!:3`, the result is given in hexadecimal notation. Because J, like APL, stores information about the *personality* of data along with the values, we drop the prefix (length is 4 + rank):

```
       ir=. 3!:3 }.~ 4: + #@$      Internal representation
       ir 0.25                  ]y=. ir %1 2 4
00000000                    00000000
0000d03f                    0000f03f
                            00000000
       ir o.1               0000e03f
182d4454                    00000000
fb210940                    0000d03f
```

Each atom is now represented by a pair of rows. Isolate these with the *infix* adverb (\) and then ravel them. Here are examples of how *infix* works:

```
  _2 <\ y
```

00000000 0000f03f	00000000 0000e03f	00000000 0000d03f

```
  _2 ,\ y
0000000000000f03f
0000000000000e03f
0000000000000d03f
```

```
    hx=. (_2: ,\ ])@ir
    bi=. <"_1          Box items
    bi 8 2$ , hx 0.25
```

00	00	00	00	00	00	d0	3f

Although this is how bytes are stored, it is convenient to reverse the order:

```
    |. bi 8 2$ , hx 0.25
```

3f	d0	00	00	00	00	00	00

```
      reverse=. ;@|.@bi@(8 2&$)@,"1
      hex=. reverse@hx
      hex % 1 2 4
3ff0000000000000      ↔     1.0
3fe0000000000000      ↔     0.5
3fd0000000000000      ↔     0.25
```

How Numbers are Subtracted Inside a PC

Floating-point representation has the advantage that a great range of magnitudes can be represented with the same number of significant figures. When floating-point numbers are multiplied or divided, the fractions and the exponents are separately combined. When they are added or subtracted, however, the fractional part of the smaller number is shifted to the right until its increased exponent matches that of the larger number. Because this happens in a register of fixed size, the smaller number loses its rightmost digits as they "drop off the end". If the shift is great enough, *all* precision is lost!

The critical part of the algorithm is therefore the operation $(1 - s^2)$. As the square of the new chord becomes smaller and smaller, it is shifted further and further to the right to match the constant 1 (hex representation `3ff0`). This is shown dramatically by displaying the square of the next chord, in both hexadecimal and decimal notation, at each iteration:

```
fmt=. [: 1&": [: (37#10)&(#:!.0) 1e37&*
n=. ": ,. i.29 [ h=. hex *: c [ c=. ch^:(i.29) 0.5
ch2=. fmt *: c [ pi=. ": ,. p [ p=. 6*(2^i.29)*c
  n ,.' ',. h ,.' ',. ch2,.' ',. pi

 0 3fd0000000000000 25000000000000000000000000000000000000 3
 1 3fb126145e9ecd57 06698729810778069000000000000000000000 3.10583
 2 3f91722b8b740eb0 01703708685546584000000000000000000000 3.13263
 3 3f71855b44e5d93f 00427756931309480800000000000000000000 3.13935
 4 3f518a29d563ba00 00107053838069826020000000000000000000 3.14103
 5 3f318b5da3b24800 00026770626181715770000000000000000000 3.14145
 6 3f118baa99e90fff 00006693104521909851000000000000000000 3.14156
 7 3ef18bbdd7a0ffff 00001673304129945484200000000000000000 3.14158
 8 3ed18bc2a711ffff 00000418327782469862580000000000000000 3.14159
 9 3eb18bc3daf00000 00000104582054993862520000000000000000 3.14159
10 3e918bc427e00000 00000026145520581888350000000000000000 3.14159
11 3e718bc43b000001 00000006536380570132395000000000000000 3.14159
12 3e518bc440000001 00000001634095170288674000000000000000 3.14159
13 3e318bc440000001 00000000408523792572168500000000000000 3.14159
14 3e118bc440000001 00000000102130948143042120000000000000 3.14159
15 3df18bc440000001 00000000025532737035760530000000000000 3.14159
16 3dd18bc500000000 00000000006383188422276475000000000000 3.14159
17 3db18bc400000001 00000000001595795717790338200000000000 3.14159
18 3d918bd000000000 00000000000398953092783926800000000000 3.14161
19 3d718bc000000000 00000000000099736885417200920000000000 3.14159
20 3d518bfffffffff 00000000000024935609133081000000000000 3.14167
21 3d318bfffffffff 00000000000062339022832702500000000000 3.14167
22 3d11900000000001 00000000000001559863349598345200000000 3.14307
```

```
23 3cf1c00000000000 0000000000000039412917374193050000000 3.15981
24 3cd1fffffffffff 0000000000000009992007221626406000000 3.18198
25 3cb4000000000001 0000000000000002775557561562891000000 3.3541
26 3ca0000000000001 0000000000000001110223024625156400000 4.24264
27 3c90000000000000 0000000000000000555111512312578200000 6
28 0000000000000000 0000000000000000000000000000000000000 0
```

The apparent precision of numbers like `5.551115123125782e_1` is very misleading.

```
      5.551115123125782e_17 ↔ dfhj '3c9'
```

The fraction has lost ALL significant figures, and the result comes from the exponent alone. With 360 architecture the result would be zero, but the PC adds an implied `1` to the fraction before multiplying by the decoded exponent. At the next iteration, however, the exponent too is zero.

STSC APL*PLUS PC System 5.0

```
 n        *: next chord          Pi
23   3.8857805861880482e_15   3.13748
24   9.9920072216264070e_16   3.18198
25   2.2204460492503132e_16   3.00000
26   5.5511151231257832e_17   3.00000
27   0                        0
```

```
3.8857805861880482e_15 ↔ (16^_11)*1 1 8 +/ .% 16^1 2 3
9.9920072216264070e_16 ↔ (16^_12)* 4 8 +/ .% 16 ^1 2
2.2204460492503132e_16 ↔ (16^_12)* 1* 16^_1
5.5511151231257832e_17 ↔ (16^_13)* 4*16^_1
```

VS APL on an IBM 4341 Computer (360 Architecture — includes a guard digit):

```
 n        *: next chord          Hex     Digits     Pi
23   3.8927194800919541e_15   351188      4      3.1403
24   9.7144514654701180e_16   344600      2      3.1375
25   2.3592239273284575e_16   341100      2      3.0923
26   5.5511151231257810e_17   334000      1      3.0000
27   1.3877787807814456e_17   331000      1      3.0000
28   0                        000000      0      0
```

```
3.8927194800919541e_15 ↔ (16^_11)*1 1 8 8+/ .%16^>:i.4
9.7144514654701180e_16 ↔ (16^_12)*4 6 +/ .% 16^1 2
2.3592239273284575e_16 ↔ (16^_12)*1 1 +/ .% 16^1 2
5.5511151231257810e_17 ↔ (16^_13)* 4%16^1
1.3877787807814456e_17 ↔ (16^_13)* 1%16^1
```

Acknowledgements

I dedicate this study to Professor Winifred Asprey, Emeritus Professor of Mathematics, Vassar College, in whose honour the Computer Centre at Vassar is named. In 1967, when Vassar received the first IBM Model 30 installed at a customer site, Dr Asprey told me that this algorithm had been run (in Fortran) as a demonstration for the Trustees of the College. To everyone's embarrassment, the computer reported the value of pi to be 3. I suggested running the program in double precision. This was duly done, only to find that the value was still 3 — to a greater number of decimal places.

I included some of this material in a paper at the 1982 I.P. Sharp Users Conference; in my Banquet Address at APL83; and elsewhere as an ACM Distinguished Lecturer. Because in 1991 several features used in the present paper had not yet been implemented in J, I omitted analysis of the algorithm's failure from my paper in the special issue of the *IBM Systems Journal* commemorating the 25th Anniversary of APL [15, pp.563-564].

Roger Hui and Ken Iverson have most generously given help and advice on countless occasions. I started this study in 1964 by reading Iverson's *A Programming Language* [8] and Falkoff, Iverson, and Sussenguth's *A Formal Description of System/360* [4]. I am particularly grateful to all these and to the late Dr J.W. Bergquist of IBM.

References

[1] Beckmann, Petr. *A History of π (pi).* The Golem Press (1977) 4th Ed. pp.184-189.

[2] Dorn, W.S. and D.D. McCracken. *Numerical Methods with Fortran IV Case Studies.* John Wiley & Sons, New York (1972) pp.106-120.

[3] Eves, Howard. *An Introduction to the History of Mathematics, with cultural connections by Jamie H. Eves.* Saunders College Publishing, Philadelphia (1990) 6th Edition. pp.117-118.

[4] Falkoff, A.D., K.E. Iverson, and E.H. Sussenguth. *A Formal Description of System/360.* IBM Systems Journal, Vol. 3, Number 3 (1964) pp.198-263.

[5] Forsythe, George. *Pitfalls in Computation.* American Mathematical Monthly, Vol.77, Number 9 (1970) pp.931-956.

[6] Greenblatt, M.H. *The "Legal" Value of π, and Some Related Mathematical Anomalies.* American Scientist, Vol. 53 (December 1965) pp.427A-434A.

[7] Hennessy, J.L. and D.A. Patterson. *The IEEE 754-1985 floating-point standard.* In: *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann Publishers, Inc. (1990) Appendix A

[8] Iverson, Kenneth E. *A Programming Language.* John Wiley and Sons, New York (1962) 286pp.

[9] Iverson, Kenneth E. *Elementary Analysis.* APL Press, Swarthmore, Pennsylvania (1976) 219pp. ISBN 0-917326-01-6. Preliminary edition *Elementary Functions.* IBM Corporation (1974).

[10] Iverson, Kenneth E. *A Personal View of APL.* IBM Systems Journal, Vol. 30, Number 4 (1991) pp.582-593.

[11] Iverson, Kenneth E. *J: Introduction and Dictionary.* Iverson Software, Inc., Toronto (1994) ISBN 1-895721-08-3.

[12] Jones, William. *Synopsis Palmariorum Matheseos, or a New Introduction to the Mathematics.* London (1706) pp. 263.

[13] Maddox, John. *False Calculation of Pi by Experiment.* Nature, Vol.370, No.6488 (4 Aug 1994) pp.323

[14] McIntyre, Donald B. *Experience with Direct Definition One-Liners in Writing APL Applications.* An APL Users Meeting, Proceedings. I.P. Sharp Associates Ltd., Toronto (1978), pp.281-297.

[15] McIntyre, Donald B. *Language as an Intellectual Tool: From Hieroglyphics to APL.* IBM Systems Journal, Vol. 30, Number 4 (1991) pp.554-581.

[16] Schepler, H.C. *The Chronology of Pi.* Mathematics Magazine (1950), pp.165-170, 216-228, 279-283.

[17] Smith, David. *An Historical Piece of Pi.* In: *Interface: Calculus and the Computer.* Saunders College Publishing, Philadelphia (1984) 2nd Edition. pp.63-75 and *Instructor's Manual* pp.23-25.

[18] Wrench, J.W. *The Evolution of Extended Decimal Approximations to Pi.* The Mathematics Teacher, Vol. 53 (1960) pp. 644-650.