Computing π in ELI

Wai-Mee Ching

April 10, 2019

Computing π, i.e. the length of the circumference of the unit circle (radius 1), has been a fascination for mathematicians from ancient times. On March 14, 2019, Emma H. Iwao of Google computed π to 31.4 trillion decimal digits, the current record. We would like to compute π using the programming language ELI, thus illustrate basic rules, facilities and coding styles of ELI. No previous experience with ELI is assumed here. Hence, this essay can be looked upon as a brief peek into ELI through three different ways to compute . ELI is an interpretive array programming language system based on the classical array language APL; it uses ASCII characters and is freely available on Windows, Mac OS and Linux (`http://fastarray.appspot.com/`). Two main features of ELI/APL are:

- ELI has 80 monadic (i.e. having a right argument only) and dyadic (i.e. having a left and a right argument) **primitive** (i.e. provided by the system) functions together with 6 **operators** (they apply to primitive functions), each one of them is represented symbolically by one or two ASCII characters. For example, `a*b` is *a multiply b*, while `a*.b` is *a to the power of b*, and `a<-b` is *the assign function.*
- A **line** of ELI code executes from **right** to **left** with all functions having **equal precedence** while an argument to a function inside a pair of parenthesis is evaluated first before the execution of the function involved. For example, we have

```
      3*2+4
18
      (3*2)+4
10
```

In fact, one of the monadic primitive functions in ELI is π, `@x` is π multiply by x:

```
      @1
3.141592654
      @0.25
0.7853981634
```

However, we are interested in computing π without invoking that primitive function. One way to compute π without using any formula is to use the Monte Carlo method. The basic idea is to throw a huge number of random dots into the 1 by 1 unit square and calculate the ratio of the number of dots falling within the quarter unit circle over the total number of dots to get $\pi/4$. The details of this computation has been worked out in §4.2.2 of the tutorial **Introduction to Programming with Arrays using ELI** available at the Document section of ELI site. The result there is 0.783 for after throwing 10,000 random dots, i.e. = 3.132, not accurate enough.

The usual way for computing π is by summing up of items in the following series

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - ...$$

Let us get the first 20 terms in ELI:

```
      sign<-20#1 _1
      sign
1 _1 1 _1 1 _1 1 _1 1 _1 1 _1 1 _1 1 _1 1 _1 1 _1
      _1+2*!20
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
      sign*_1+2*!20
1 _3 5 _7 9 _11 13 _15 17 _19 21 _23 25 _27 29 _31 33 _35 37 _39

      %sign*_1+2*!20
1 _0.3333333333 0.2 _0.1428571429 0.1111111111 _0.09090909091 0.07692307692 _0.06666666667 0.05882352941
      _0.05263157895 0.04761904762 _0.04347826087 0.04 _0.03703703704 0.03448275862 _0.03225806452
      0.0303030303 _0.02857142857 0.02702702703 _0.02564102564
      +/%sign*_1+2*!20
0.7729059517
```

Note that i) _1 is negative 1; ii) # is the dyadic primitive function **reshape** which replicates the right argument b up to a (the left argument) elements; iii) is !x the interval function which generates a vector of integers from 1 to x ; iv) dyadic * is the multiplication function, 2 multiplies to each element of the right argument, and the monadic % is the inverse function which inverts each element of the right argument; v) +/ is the **reduce** operator applied to the add function +, i.e. sums up the vector on its right side.

We would like to save this piece of code into a user defined function. A **defined function** f in ELI has a name (in contrast to a primitive function being denoted by a symbol), a right argument or a left-right argument pair (or no argument at all). For a simple defined function, we can write it in **short-function form** which is a line of code preceded by the function name and enclosed in a pair of curly brackets where x is assumed to be the right argument, y is the left argument if present, and z or value of the last expression is the return result of the function. In our case, we have

```
       {pi: 4*+/%(x#1 _1)*_1+2*!x}
pi
       pi 100
3.131592904
       pi 1000
3.140592654
```

where the right argument x is the number of items to sum up and the result is π. We see that we need to sum up 1000 items of the series to get the second decimal digit of π to be correct. This formula which has been around since year 1400 is of very slow convergence.

Currently, the most efficient way to compute π is the Bailey-Borwein-Plouffe formula (BBP, see Mark Braveman: *Computing with real numbers, from Archimedes to Turing and beyond*, p251 in [1]):

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

which was last improved in 1995. Let us write a short function `f` with `f(k)` replacing each individual item in the summation above after `k` substitutes for `x`.

```
      {f:(%16*.x)*(4%1+8*x)-(2%4+8*x)+(%5+8*x)+%6+8*x}
f
      v<-0,!9
      v
0 1 2 3 4 5 6 7 8 9
      f"v
3.133333333 0.008089133089 0.0001649239241 5.067220854E_6 1.878929009E_7 7.767751215E_9
      3.447932931E_10 1.609187716E_11 7.795702954E_13 3.88711526E_14
      +/f"v
3.141592654
```

We note that because ELI executes from right to left, ordinary arithmetic expression `a-b-c-d` becomes `a-(b+c+d)` in ELI. `f"v` is the **each** operator `"` applied to the defined function `f` resulting in a derived function `f"` which applies `f` to each item in `v`; sums up the resulting items ends up in π. We see that the BBP formula converges very fast: a summation of 10 terms is already identical to the primitive function `@1` to the tenth decimal digit. In fact, floating-point in ELI uses double precision 64-bits IEEE 754 standard, so we can only get 15 meaningful decimal digits. The default in ELI for printing floating point is 10 decimal digits. This can be changed to 15 by resetting the system variable `[]PP`:

```
     []PP<-15
     +/f"v
3.14159265358979
     @1
3.14159265358979
```

Therefore the above computation is accurate up to 15 decimal digits. Still, there are ways to compute π to more decimal digits using the BBP formula. To do so, we must replace `v` above by a parameter x which represents the number of items in the BBP formula for summation. We have

```
    {BBP: []IO<-0; +/f"!x}
BBP
```

where the system function `[]IO`, called **index origin**, changes the result of !10 from 1. . . 10 to 0. . . 9. Now we have

```
    BBP 10
3.141592654
```

We may use this function to device a way to compute π up to a thousand decimal digits, but beyond that it is a challenge to compute more decimal digits of π accurately as it would involve the question of storing intermediate results.

References

[1] M. Pitici, ed., The Best Writing on Mathematics 2014, Princeton Univ. Press, 2014.