Unbounded Spigot Algorithms for the Digits of π

Jeremy Gibbons

24th May 2004

Abstract

Rabinowitz and Wagon (in *American Mathematical Monthly* 102(3):195–203, 1995) present a *spigot algorithm* for computing the digits of π. A spigot algorithm yields its outputs incrementally, and does not reuse them after producing them. Their algorithm is inherently *bounded*; it requires a commitment in advance to the number of digits to be computed, and in fact might still produce an incorrect last few digits. We propose some *streaming algorithms* based on the same and some other characterizations of π, with the same incremental characteristics but without requiring the prior bound.

1 Introduction

Rabinowitz and Wagon [8] present a 'remarkable' algorithm for computing the decimal digits of π, based on the expansion

$$\pi = \sum_{i=0}^{\infty} \frac{(i!)^2 2^{i+1}}{(2i+1)!} \tag{1}$$

Their algorithm uses only bounded integer arithmetic, and is surprisingly efficient. Moreover, it admits extremely concise implementations; witness, for example, the following (deliberately obfuscated) C program due to Dik Winter and Achim Flammenkamp [1, p. 37], which produces the first 15,000 decimal digits of π:

```
a[52514],b,c=52514,d,e,f=1e4,g,h;main(){for(;b=c-=14;h=printf("%04d",
e+d/f))for(e=d%=f;g=--b*2;d/=g)d=d*b+f*(h?a[b]:f/5),a[b]=d%--g;}
```

Rabinowitz and Wagon call their algorithm a *spigot algorithm*, because it yields digits incrementally, and does not use digits after they have been computed. The digits drip out one by one, as if from a leaky tap. In contrast, most algorithms for computing the digits of π deliver no output until the whole computation is completed.

However, their algorithm has its weaknesses. In particular, the computation is inherently bounded: one has to commit in advance to computing a certain number of digits. Based on this commitment, the computation proceeds on an appropriate finite prefix of the infinite series (1). In fact, it is essentially impossible to determine in advance how big that finite prefix should be — a computation that terminates with nines for the last few digits

of the output is inconclusive, because there may be a 'carry' from the first few truncated terms. Rabinowitz and Wagon suggest that 'in practice, one might ask for, say, 6 extra digits, reducing the odds of this problem to one in a million' [8, p. 197], a not entirely satisfactory recommendation. Indeed, the implementation printed at the end of their paper is not quite right [1, p. 82], sometimes printing an incorrect last digit because the finite approximation to the infinite series is one term too short.

We propose a different algorithm, based on the same series (1) for π but avoiding these problems; we show the same technique applied to other series too. No commitment need be made in advance to the number of digits to be computed; the programs will generate digits ad infinitum. Once more (necessarily, in fact, given the previous property), they are spigot algorithms in Rabinowitz and Wagon's sense: they yield digits incrementally, and do not reuse them after producing them. Of course, no algorithm using a bounded amount of memory can generate digits of π, a non-repeating sequence, indefinitely, so we have to allow arbitrary-precision arithmetic (or some other use of dynamic memory allocation). Like Rabinowitz and Wagon's algorithm, our proposals are not competitive with state-of-the-art *arithmetic-geometric mean* algorithms for computing π [2, 9]. Nevertheless, the algorithms are simple to understand and admit almost as concise an implementation. As evidence to support the second claim, here is a program that will generate as many digits of π as memory will allow:

```
pi = g(1,0,1,1,3,3)where
  g(q,r,t,k,n,l) = if 4*q+r-t<n*t
    then n:g(10*q,10*(r-n*t),t,k,div(10*(3*q+r))t-10*n,l)
    else g(q*k,(2*q+r)*l,t*l,k+1,div(q*(7*k+2)+r*l)(t*l),l+2)
```

The remainder of this paper provides a justification for the above program, and some others like it.

These algorithms exhibit a pattern that we call *streaming* [3]. Informally, a streaming algorithm consumes a (potentially infinite) sequence of inputs, and generates a (possibly infinite) sequence of outputs, maintaining some state as it goes; based on the current state, at each step there is a choice between producing an element of the output and consuming an element of the input. Streaming seems to be a common pattern for various kinds of *representation changers*, including several data compression and number conversion algorithms.

The program above is written in Haskell [6], the de facto standard lazy functional programming language. As a secondary point of this paper, we hope to convince the reader that Haskell is an excellent vehicle for expressing mathematical computations, certainly when compared to other general-purpose programming languages like Java, C and Pascal, and even when compared to computer algebra systems like Mathematica. In particular, a lazy language allows direct computations with infinite data structures, which require some kind of indirect representation in most other languages. The Haskell program above has been compressed to compete with the earlier C program for conciseness, so we do not argue that it is easy to follow; but we do claim that the later programs are.

2 Rabinowitz and Wagon's spigot algorithm

Rabinowitz and Wagon's algorithm is based on the expression (1) for π, which expands out to

$$\pi \;=\; 2 + \frac{1}{3}\Big(2 + \frac{2}{5}\Big(2 + \frac{3}{7}\Big(\cdots\Big(2 + \frac{i}{2i+1}\Big(\cdots\Big)\Big)\Big)\Big)\Big) \tag{2}$$

This expression for π can be derived from the well-known Leibniz series

$$\frac{\pi}{4} \;=\; \sum_{i=0}^{\infty} \frac{(-1)^i}{2i+1} \tag{3}$$

using Euler's convergence-accelerating transform, among several other methods [7]. One can view expression (2) as representing a number $(2; 2, 2, 2, \ldots)$ in a mixed-radix base $\mathcal{B} = (\frac{1}{3}, \frac{2}{5}, \frac{3}{7}, \ldots)$, in the same way that the usual decimal expansion

$$\pi = 3 + \frac{1}{10}\Big(1 + \frac{1}{10}\Big(4 + \frac{1}{10}\Big(1 + \frac{1}{10}\Big(5 + \cdots\Big)\Big)\Big)\Big)$$

represents $(3; 1, 4, 1, 5, \ldots)$ in the fixed-radix base $(\frac{1}{10}, \frac{1}{10}, \frac{1}{10}, \ldots)$. The task of computing the decimal digits of π is then simply a matter of converting from base $\mathcal{B}$ to base ten.

We consider *regular* representations. For a regular representation in decimal, every digit after the decimal point is in the range $[0, 9]$; the decimal number with a zero before the point and maximal fractional digits $(0; 9, 9, 9, \ldots)$ represents 1, and so regular representations with a zero before the point lie between 0 and 1. By analogy, for a regular representation in base $\mathcal{B}$, the digit in position i after the point (the first after the point being in position 1) is in the range $[0, 2i]$; the number with zero before the point and maximal digits $(0; 2, 4, 6, \ldots)$ represents 2, so regular representations with zero before the point lie between 0 and 2. (We call the representations 'regularized' rather than 'normalized' because they are not unique.)

Conversion from base $\mathcal{B}$ to decimal proceeds as one might expect. The integer part of the input becomes the integer part of the output. The fractional part of the input is multiplied by ten; the integer part of this becomes the first output digit after the decimal point, and the fractional part is retained. This is again multiplied by ten; the integer part of this becomes the second output digit after the point; and so on.

Multiplying a number in base $\mathcal{B}$ by ten is achieved simply by multiplying each digit by ten. However, that yields an irregular result, because some of the resulting digits may be too big. Regularization proceeds from right to left, reducing each digit as necessary and propagating any carry leftwards.

The only remaining problem is in computing the integer part of a number $(a_0; a_1, a_2, a_3)$ in base $\mathcal{B}$. This is either a_0 or $a_0 + 1$, depending on whether the remainder $(0; a_1, a_2, a_3)$ is in $[0, 1)$ or $[1, 2)$. (In principle, the remainder could equal 2; but in practice, that cannot happen in the computation of an irrational such as π.) Therefore Rabinowitz and Wagon's algorithm temporarily buffers any nines that are produced, until it is clear whether or not there could be a carry that would invalidate them.

This whole conversion is performed on a *finite* number $(2; 2, 2, 2, \ldots, 2)$ in base $\mathcal{B}$ — necessarily, as regularization proceeds from right to left. Rabinowitz and Wagon provide

a bound on the number of base $\mathcal{B}$ digits needed to yield a given number of decimal digits. In fact, as mentioned above, they underestimated by one in some cases: $\lfloor 10n/3 \rfloor$ digits is usually sufficient, but sometimes $\lfloor 10n/3 \rfloor + 1$ input digits is necessary (and sufficient) for n decimal digits. Again, as noted above, this does not mean that those n decimal digits are all correct, only that the n-digit decimal number produced is within 5×10^{-n} of π.

3 Streaming algorithms

We turn now to streaming algorithms, by way of a simpler example. Consider the problem of converting a fraction in $[0, 1]$ from one base to another. We will represent fractions as digit sequences, and for simplicity but without loss of generality we will consider only infinite sequences. For this reason, we cannot consume all the input before generating any output; we must alternate between consumption and production. The computation will therefore maintain some state depending on the inputs consumed and the outputs produced so far; based on that state, it will either produce another output, if that is possible, or consume another input if it is not. This pattern is captured by the following program.

```
> stream :: (b->c) -> (b->c->Bool) -> (b->c->b) -> (b->a->b) ->
>           b -> [a] -> [c]
> stream next safe prod cons z (x:xs)
>   = if safe z y then y : stream next safe prod cons (prod z y) (x:xs)
>                 else stream next safe prod cons (cons z x) xs
>     where y = next z
```

This defines a function `stream` taking six arguments. The result of applying `stream` will be an infinite list of output terms, each of type `c`. The last argument `x:xs` is a list of input terms, each of type `a`; the first element or 'head' is `x`, and the infinite remainder or 'tail' is `xs`. The penultimate argument `z` is the state, of type `b`. The other four arguments (`next`, `safe`, `prod` and `cons`) are all functions. From the state `z` the function produces a provisional output term `y = next z` of type `c`. If `y` is `safe` to commit to from the current state `z` (whatever input terms may come next), then it is produced, and the state adjusted accordingly using `prod`; otherwise, the next term `x` of the input is `cons`umed into the state. This process continues indefinitely: the input is assumed never to run out, and if the process is productive then the output will never terminate.

In the case of conversion from an infinite digit sequence in base $\mathcal{M}$ to an infinite sequence in base $\mathcal{N}$, clearly the input and output elements will both be of type `Integer`. The state maintained is a pair (u, v) of `Rational`s, satisfying the invariant that the original input

$$\frac{1}{\mathcal{M}}\Big(x_0 + \frac{1}{\mathcal{M}}\Big(x_1 + \cdots\Big)\Big)$$

is equal to

$$\frac{1}{\mathcal{N}}\Big(y_0 + \frac{1}{\mathcal{N}}\Big(y_1 + \cdots + \frac{1}{\mathcal{N}}\Big(y_{j-1} + v \times \Big(u + \frac{1}{\mathcal{M}}\Big(x_i + \frac{1}{\mathcal{M}}\Big(x_{i+1} + \cdots\Big)\Big)\Big)\Big)\Big)\Big)\Big)$$

when input terms $x_0, x_1, \ldots, x_{i-1}$ have been consumed and output terms $y_0, y_1, \ldots, y_{j-1}$ have been produced. Initially i and j are zero, so the invariant is established with $u = 0$

and $v = 1$. In order to maintain the invariant, the state (u, v) should be transformed to $(u - \frac{y}{\mathcal{N}\,v}, \mathcal{N}\,v)$ when producing an extra output term y, and to $(x + u\,\mathcal{M}, \frac{v}{\mathcal{M}})$ when consuming an extra input term x. The value of the remaining input

$$\frac{1}{\mathcal{M}}\Big(x_i + \frac{1}{\mathcal{M}}\Big(x_{i+1} + \cdots\Big)\Big)$$

ranges between 0 and 1, so the next output term is determined provided that $\mathcal{N}\,v\,u$ and $\mathcal{N}\,v\,(u+1)$ have the same integer part or `floor`. This gives us the following program.

```
> convert :: (Integer,Integer) -> [Integer] -> [Integer]
> convert (m,n) xs = stream next safe prod cons init xs
>   where
>     init         = (0%1, 1%1)
>     next (u,v)   = floor (u*v*n)
>     safe (u,v) y = (y == floor ((u+1)*v*n))
>     prod (u,v) y = (u - fromInteger y/(v*n), v*n)
>     cons (u,v) x = (fromInteger x + u*m, v/m)
```

(Here, `%` constructs a `Rational` from two `Integer`s, and `fromInteger` coerces `Integer`s to `Rational`s.)

This paper is not the place to make a more formal justification for the correctness of this program, although it is not hard to establish from the invariant stated. Nevertheless, it is possible to *derive* the streaming program from a specification expressed in terms of independent operations for expanding and collapsing digit sequences, using a general theory of such algorithms [3]. (In the general case, either the input or the output or both may be finite; we have stuck to the simple case of necessarily infinite lists here, because that is all that is needed for computing the digits of π.) We have found this pattern of computation cropping up in numerous problems concerning *changes of data representation*, of which conversions between number formats are a representative example, so we have been calling such algorithms *metamorphisms*.

4 A streaming algorithm for the digits of π

The main problem with Rabinowitz and Wagon's spigot algorithm is that it is bounded: one must make a commitment in advance to the number of terms of the series (1) to use. This commitment in turn arises because the process of regularizing a number in base $\mathcal{B}$ proceeds from right to left, so works only for finite numbers in that base.

It turns out that there is a rather simple *streaming algorithm* for regularizing infinite numbers in base $\mathcal{B}$. This means we can make Rabinowitz and Wagon's algorithm unbounded: there is no longer any need to make a prior commitment to a particular finite prefix of the expansion of π. However, we will not say any more about this approach: there is a more direct way of computing the digits of π from the expression (2), to which we now turn.

One can view the expansion (2) as the composition

$$\pi \;=\; \Big(2 + \frac{1}{3}\times\Big)\Big(2 + \frac{2}{5}\times\Big)\Big(2 + \frac{3}{7}\times\Big)\cdots\Big(2 + \frac{i}{2i+1}\times\Big)\cdots \tag{4}$$

of an infinite series of *linear fractional transformations* or *Möbius transformations*. These are functions taking x to $\frac{qx+r}{sx+t}$ for integers q, r, s, t — that is, yielding a ratio of integer-coefficient linear transformations of x. Such a transformation can be represented by the four coefficients q, r, s, t, and if they are arranged as a matrix $\left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)$ then function composition corresponds to matrix multiplication.

```
> type LFT = (Integer, Integer, Integer, Integer)

> extr :: LFT -> Integer -> Rational
> extr (q,r,s,t) x = (fromInteger q * x + fromInteger r) /
>                    (fromInteger s * x + fromInteger t)

> unit :: LFT
> unit = (1,0,0,1)

> comp :: LFT -> LFT -> LFT
> comp (q,r,s,t) (q',r',s',t') = (q*q'+r*s',q*r'+r*t',s*q'+t*s',s*r'+t*t')
```

The infinite composition of transformations in (4) converges, in the following sense. Although the products of finite prefixes of the composition have coefficients that grow without bound, the transformations represented by these products focus the interval $[3, 4]$ onto ever-decreasing subintervals of itself. (This is easy to see, as each term is a monotonic transformation, reduces the width of an interval by at least a factor of two, and maps $[3, 4]$ onto a subinterval of itself. Indeed, the same holds for any tail of the infinite composition too.)

Therefore, Equation (4) can be thought of as the representation of some real number, and computing the decimal digits of this number is a change of representation, effectable by a streaming algorithm. The streaming process maintains as its state an additional linear fractional transformation, representing the required function from the inputs yet to be consumed to the outputs yet to be produced. This state is initially the unit matrix; consumption of another input term is matrix multiplication; production of a digit n is multiplication by $\left(\begin{smallmatrix} 10 & -10n \\ 0 & 1 \end{smallmatrix}\right)$, the inverse of the linear fractional transformation taking x to $n + \frac{x}{10}$. If the current state is the transformation z, then the next digit to be produced lies somewhere in the image under z of the interval $[3, 4]$; if the two endpoints of this image have the same integer part, then that next digit is completely determined and safe to commit to.

```
> pi = stream next safe prod cons init lfts where
>   init      = unit
>   lfts      = [(k, 4*k+2, 0, 2*k+1) | k<-[1..]]
>   next z    = floor (extr z 3)
>   safe z n  = floor (extr z 4) == n
>   prod z n  = comp (10, -10*n, 0, 1) z
>   cons z z' = comp z z'
```

The definition of `lfts` above uses a *list comprehension*, analogous to a Zermelo–Fraenkel set comprehension. The list consists of the expression `(k, 4*k+2, 0, 2*k+1)` to the left

of the vertical bar, evaluated for each value of `k` from 1 upwards; the first few terms are

$$\left[\begin{pmatrix} 1 & 6 \\ 0 & 3 \end{pmatrix}, \begin{pmatrix} 2 & 10 \\ 0 & 5 \end{pmatrix}, \begin{pmatrix} 3 & 14 \\ 0 & 7 \end{pmatrix}, \ldots\right]$$

For example, the first term $\left(\begin{smallmatrix} 1 & 6 \\ 0 & 3 \end{smallmatrix}\right)$ represents the transformation taking x to $\frac{1\times x+6}{0\times x+3}$, or $2+\frac{1}{3}x$.

The condensed program shown in Section 1 can be obtained from this one by making various simple optimizations: unfolding intermediate definitions; exploiting the invariant that the bottom left element s of every linear fractional transformation $\left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)$ is 0; constructing the input transformations in place; representing the sequence of remaining transformations simply by the number `k`; and simplifying away one of the divisions.

Another optimization that can be performed is, after each matrix multiplication, to eliminate any factors common to all four entries of the product; this optimization is valid, since linear fractional transformations are invariant under scaling of the matrix, and helpful, as it keeps the numbers small. (In fact, it is better still to perform this cancellation less frequently than every iteration.)

5 More streaming algorithms for the digits of π

The expression (2) turns out not to be a very efficient one for computation; each term focusses the range by a factor of about a half, so more than three terms are required for every digit of the output. Better sequences are known; the book π *Unleashed* [1] presents many. We conclude with two more applications of the streaming technique from Section 3 to computing π, using the same approach but based on two of these different series.

5.1 Lambert's series

Here is a more efficient series for π, due to Lambert in 1770 [1, equation (16.99)] and yielding two decimal digits every three terms:

$$\pi = \cfrac{4}{1+\cfrac{1^2}{3+\cfrac{2^2}{5+\cfrac{3^2}{7+\cdots}}}} \tag{5}$$

One can again view this as an infinite composition of linear fraction transformations:

$$\pi = \Big(4\div\Big)\Big(1+1^2\div\Big)\Big(3+2^2\div\Big)\Big(5+3^2\div\Big)\cdots\Big(2i-1+i^2\div\Big)\cdots$$

After consuming i terms of the input, the remaining terms represent the composition

$$\Big(2i-1+i^2\div\Big)\Big(2i+1+(i+1)^2\div\Big)\Big(2i+3+(i+2)^2\div\Big)\cdots$$

which denotes a value in the range $[2i-1, 2i-1+\frac{i}{2}]$. As before, we subject this infinite sequence to a streaming process; this time, however, we maintain a state consisting of not just a linear fractional transformation $\left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)$, but also the number i of terms consumed so far (needed in order to determine the next digit to produce, which lies between $\left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)(2i-1)$ and $\left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)(2i-1+\frac{i}{2})$).

This reasoning justifies the following program:

```
> piL = stream next safe prod cons init lfts where
>   init                 = ((0,4,1,0), 1)
>   lfts                 = [(2*i-1, i^2, 1, 0) | i<-[1..]]
>   next ((q,r,s,t),i)   = floor ((q*x+r) % (s*x+t)) where x=2*i-1
>   safe ((q,r,s,t),i) n = floor ((q*x+2*r) % (s*x+2*t)) == n where x=5*i-2
>   prod (z,i) n         = (comp (10, -10*n, 0, 1) z,i)
>   cons (z,i) z'        = (comp z z', i+1)
```

5.2 Gosper's series

An even more efficient series for π, yielding more than one decimal digit per term, is due to Gosper [4]:

$$\pi \;=\; 3+\frac{1\times 1}{3\times 4\times 5}\times(8+\frac{2\times 3}{3\times 7\times 8}\times(\cdots 5i-2+\frac{i(2i-1)}{3(3i+1)(3i+2)}\times\cdots)) \qquad (6)$$

One can again view this as an infinite composition of linear fraction transformations:

$$\pi \;=\; \left(3+\frac{1\times 1}{3\times 4\times 5}\times\right)\left(8+\frac{2\times 3}{3\times 7\times 8}\times\right)\cdots\left(5i-2+\frac{i(2i-1)}{3(3i+1)(3i+2)}\times\right)\cdots$$

It is not hard to show that, after consuming $i-1$ terms of the input, the remaining terms denote a value in the range $[\frac{27}{5}i-\frac{12}{5}, \frac{27}{5}i-\frac{2^3 3^3}{5^3}]$, which gives the following program:

```
> piG = stream next safe prod cons init lfts where
>   init                 = ((1,0,0,1), 0)
>   lfts                 = [let u = 3*(3*i+1)*(3*i+2)
>                           in (i*(2*i-1),u*(5*i-2),0,u) | i<-[1..]]
>   next ((q,r,s,t),i)   = div (q*x+5*r) (s*x+5*t) where x = 15+27*i
>   safe ((q,r,s,t),i) n = n == div (q*x+125*r) (s*x+125*t) where x=675*i-216
>   prod (z,i) n         = (comp (10, -10*n, 0, 1) z,i)
>   cons (z,i) z'        = (comp z z', i+1)
```

5.3 A challenge

Christoph Haenel [5] suggests that Gosper's series is even better than suggested above: since it yields at least one digit per term, one could dispense with the test altogether, and simply alternate between consumption and production, giving the following program.

```
> piG2 = process next prod cons init lfts
> process next prod cons z (x:xs)
>   = y : process next prod cons (prod z' y) xs
>     where z' = cons z x
>           y = next z'
```

where `next`, `prod`, `cons`, `init` and `lfts` are as in Section 5.2. The proof that this simplification is valid (that is, that `safe z' (next z')` holds for each `z'` encountered) eludes the author at present, but perhaps some diligent reader can provide enlightenment. If it is valid, the optimizations outlined at the end of Section 4 can be applied, giving the following program:

```
piG3 = g(1,180,60,2) where
  g(q,r,t,i) = let (u,y)=(3*(3*i+1)*(3*i+2),div(q*(27*i-12)+5*r)(5*t))
               in y : g(10*q*i*(2*i-1),10*u*(q*(5*i-2)+r-y*t),t*u,i+1)
```

which is of comparable length to the compressed program given in Section 1, but approximately five times faster.

Acknowledgements

Thanks are due to the Algebra of Programming research group at Oxford, Stan Wagon and especially to Christoph Haenel; all of them have made suggestions that have improved the presentation of this paper.

References

[1] Jörg Arndt and Christoph Haenel. π *Unleashed*. Springer-Verlag, second edition, 2000.

[2] Richard P. Brent. Fast multiple-precision evaluation of elementary functions. *Journal of the ACM*, 23(2):242–251, April 1976.

[3] Jeremy Gibbons. Metamorphisms and streaming algorithms. In *Mathematics of Program Construction*, Lecture Notes in Computer Science. Springer Verlag, July 2004.

[4] R. W. Gosper. Acceleration of series. Technical Report AIM-304, AI Laboratory, MIT, March 1974. `ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-304.pdf`.

[5] Christoph Haenel. Personal communication. Email message, January 2004.

[6] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[7] J. C. R. Li et al. Solutions to Problem E854: A series for π. *American Mathematical Monthly*, 56(9):633–635, November 1949.

[8] Stanley Rabinowitz and Stan Wagon. A spigot algorithm for the digits of π. *American Mathematical Monthly*, 102(3):195–203, 1995.

[9] Eugene Salamin. Computation of π using arithmetic-geometric mean. *Mathematics of Computation*, 30(135):565–570, July 1976.