Unbounded Spigot Algorithms for the Digits of Pi

Jeremy Gibbons

1 INTRODUCTION.

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

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

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 reuse 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 execute inscrutably, delivering no output until the whole computation is completed.

However, the Rabinowitz–Wagon 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 for a given number of digits—specifically, 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, six 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 of 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 also show the same technique applied to other characterizations of π. No commitment need be made in advance to the number of digits to be computed; given enough memory, the programs will generate digits ad infinitum. Once more (necessarily, in fact, given the previous property), the programs 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 a nonrepeating sequence such as the digits of π indefinitely, so we have to allow arbitrary-precision arithmetic, or some other manifestation 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, our algorithms are simple to understand and admit almost as concise an implementation. As evidence to support the second claim, here is a (deliberately obscure) 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 foregoing 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 under discussion is written in Haskell [5], a lazy functional programming language. As a secondary point of this paper, we hope to convince the reader that such languages are excellent vehicles for expressing mathematical computations, certainly when compared with other general-purpose programming languages such as Java, C, and Pascal, and arguably even when compared with computer algebra systems such as 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 presented earlier has been compressed to compete with the C program for conciseness, so we do not argue that this particular one is easy to follow—but we do claim that the later Haskell programs are.

2 LAZY FUNCTIONAL PROGRAMMING IN HASKELL.

To aid the reader's understanding, we start with a brief (and necessarily incomplete) description of the concepts of functional programming (henceforth FP) and laziness and their manifestation in Haskell [5], the de facto standard lazy FP language. Further resources, including pointers to tutorials and free implementations for many platforms, can be found at the Haskell website [6].

FP is programming with *expressions* rather than *statements*. Everything is a value, and there are no assignments or other state-changing commands. Therefore, a pure FP language is *referentially transparent*: an expression may always be substituted for one with an equal value, without changing the meaning of the surrounding context. This makes reasoning in FP languages just like reasoning in high-school algebra.

Here is a simple Haskell program:

```
> square :: Integer -> Integer
> square x = x * x
```

Program text is marked with a "`>`" in the left-hand column, and comments are unmarked. This is a simple form of *literate programming*, in which the emphasis is placed on making the program easy for people rather than computers to read. Code and documentation are freely interspersed; indeed, the manuscript for this article is simultaneously an executable Haskell program.

The first line in the `square` program is a type declaration; the symbol "`::`" should be read "has type." Thus, `square` has type `Integer -> Integer`, that is, it is a function from `Integer`s to `Integer`s. (Type declarations in Haskell are nearly always optional, because they can be inferred, but we often specify them anyway for clarity.) The second line gives a definition, as an equation: in any context, a subexpression of the form `square x` for any `x` may be replaced safely by `x * x`.

Lists are central to FP. Haskell uses square brackets for lists, so `[1,2,3]` has three elements, and `[]` is the empty list. The operator "`:`" prefixes an element; so `[1,2,3] = 1 : (2 : (3 : []))`. For any type `a`, there is a corresponding type `[a]` of lists with elements drawn from type `a`, so, for example, `[1,2,3] :: [Integer]`. Haskell has a very convenient *list comprehension* notation, analogous to set comprehensions; for instance, `[ square x | x <- [1,2,3] ]` denotes the list of squares `[1,4,9]`.

For the "substitution of equals for equals" property to be universally valid, it is important not to evaluate expressions unless their values are needed. For example, consider the following Haskell program:

```
> three :: Integer -> Integer
> three x = 3

> nonsense :: Integer
> nonsense = 1 + nonsense
```

The first definition is of a function that ignores its argument `x` and always returns the integer `3`; the second is of a value of type `Integer`, but one whose evaluation never terminates. For substitutivity to hold, and in particular for `three nonsense` to evaluate to `3` as the equation suggests, it is important not to evaluate the function argument `nonsense` in the function application `three nonsense`. With *lazy evaluation*, in which evaluation is demand-driven, no expression is evaluated unless and until its value is needed to make further progress.

A useful by-product of lazy evaluation is the ability to handle infinite data structures: they are evaluated only as far as is necessary. We illustrate this with a definition of the infinite sequence of Fibonacci numbers:

```
> fibs :: [Integer]
> fibs = f (0,1) where f (a,b) = a : f (b,a+b)
```

(Here, `(0,1)` is a pair of `Integer`s, and the `where` clause introduces a local definition.) Evaluating `fibs` never terminates, of course, but computing a finite prefix of it does. For example, with

```
> take :: Integer -> [Integer] -> [Integer]
> take 0 xs         = []
> take (n+1) []     = []
> take (n+1) (x:xs) = x : take n xs
```

(so `take` takes two arguments, an `Integer` and a `[Integer]`, and returns another `[Integer]`), we have `take 10 fibs = [0,1,1,2,3,5,8,13,21,34]`, which terminates normally.

Functions may be *polymorphic*, defined for arbitrary types. Thus, the function `take` in fact works for lists of any element type, since elements are merely copied and not further analyzed. The most general type assignable to `take` is `Integer -> [a] -> [a]` for an arbitrary type `a`.

Because this is FP, functions are "first-class citizens" of the language, with all the rights of any other type. Among other things, they may be passed as arguments to and returned as results from *higher-order functions*. For example, with the definition

```
> map :: (a->b) -> [a] -> [b]
> map f []     = []
> map f (x:xs) = f x : map f xs
```

the list comprehension `[ square x | x <- [1,2,3] ]` is equal to `map square [1,2,3]`. Note that `map` and `take` are *curried*. In fact, the function type former `->` associates to the right, so the type of `map` is equivalent to `(a->b) -> ([a]->[b])`, which one might read as saying that `map` takes a function of type `a->b` and transforms it into one of type `[a]->[b]`.

3 RABINOWITZ AND WAGON'S SPIGOT ALGORITHM.

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

$$\pi = 2+\frac{1}{3}\left(2+\frac{2}{5}\left(2+\frac{3}{7}\left(\cdots\left(2+\frac{i}{2i+1}\left(\cdots\right)\right)\right)\right)\right) \quad . \tag{2}$$

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

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

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

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 $(0;9,9,9,\ldots)$ with a zero before the point and maximal digits afterwards represents 1. Accordingly, regular decimal 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 $(0;2,4,6,\ldots)$ with zero before the point and maximal digits afterwards represents 2. Accordingly, regular base $\mathcal{B}$ representations with zero before the point lie between 0 and 2. (We call the representations "regular" rather than "normal" 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, this 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 earlier, they underestimate 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. (Here, "$\lfloor x\rfloor$" denotes the greatest integer not larger than x.) Again, as noted, this does not mean that those n decimal digits are all correct digits of π, only that the n-digit decimal number produced is within 5×10^{-n} of the desired result.

4 STREAMING ALGORITHMS.

We turn now to streaming algorithms, by way of a simpler example than computing the digits of π. Consider the problem of converting a fraction in the interval $[0,1]$ from one base to another. We represent fractions as digit sequences, and for simplicity (but without loss of generality) we consider only infinite sequences. For this reason, we cannot consume all the input before producing 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 thus far. Based on that state, it will either produce another output, if that is possible given the available information, or consume another input if it is not. This pattern is captured by the following higher-order function:

```
> 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` is 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` of type `b->c`, `safe` of type `b->c->Bool`, `prod` of type `b->c->b` and `cons` of type `b->a->b`) 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, the output never terminates.

In the case of conversion from an infinite digit sequence in base $\mathcal{F}_m$ to an infinite sequence in base $\mathcal{F}_n$, clearly both the input and output elements are of type `Integer`. The state maintained is a pair (u,v) of `Rationals`, satisfying the "invariant" (that is, a property that is established before a loop commences, and is maintained by each iteration of that loop) that the original input

$$\frac{1}{m}\left(x_0+\frac{1}{m}\left(x_1+\cdots\right)\right)$$

is equal to

$$\frac{1}{n}\left(y_0+\frac{1}{n}\left(y_1+\cdots+\frac{1}{n}\left(y_{j-1}+v\times\left(u+\frac{1}{m}\left(x_i+\frac{1}{m}\left(x_{i+1}+\cdots\right)\right)\right)\right)\right)\right)$$

when i input terms $x_0,x_1,\ldots,x_{i-1}$ have been consumed and j 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 $(u-y/(nv),nv)$ when producing an additional output term y and to $(x+um,v/m)$ when consuming an additional input term x. The value of the remaining input

$$\frac{1}{m}\left(x_i+\frac{1}{m}\left(x_{i+1}+\cdots\right)\right)$$

ranges between 0 and 1, so the next output term is determined provided that nvu and $nv(u+1)$ have the same integer part or `floor`. This justifies the following streaming algorithm:

```
> 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')
>     (m',n')       = (fromInteger m, fromInteger n)
```

(Here, "`%`" constructs a `Rational` from two `Integer`s, "==" is the comparison operator, and the function `fromInteger` coerces `Integer`s to `Rational`s.)

For example, $1/e$ is $0.1002210112\ldots$ in base 3 and $0.2401164352\ldots$ in base 7. Therefore, applying the function `convert (3,7)` to the infinite list `[1,0,0,2,2,1,0,1,1,2...` should yield the infinite list `[2,4,0,1,1,6,4,3,5,2...`. The first few states through which execution of this conversion proceeds are illustrated in the following table:

input		1	0	0		2	2	1				
state	$\frac{0}{1},\frac{1}{1}$	$\frac{1}{1},\frac{1}{3}$	$\frac{3}{1},\frac{1}{9}$	$\frac{9}{1},\frac{1}{27}$	$\frac{9}{7},\frac{7}{27}$	$\frac{41}{7},\frac{7}{81}$	$\frac{137}{7},\frac{7}{243}$	$\frac{418}{7},\frac{7}{729}$	$\frac{10}{49},\frac{49}{729}$	$\frac{10}{49},\frac{343}{729}$	$\cdots$	
output				2					4	0		

The middle row shows consecutive values of the state (u,v). The upper row shows input digits consumed, above the state resulting from their consumption. The lower row shows output digits produced, below the state resulting from their production. Notice that outputs are produced precisely when the corresponding state (u,v), which is in the previous column, is safe; that is, when $\lfloor 7uv \rfloor = \lfloor 7(u+1)v \rfloor$. For example, the fourth state $(\frac{9}{1},\frac{1}{27})$ is safe, because $\lfloor 7 \times \frac{9}{1} \times \frac{1}{27} \rfloor = 2 = \lfloor 7 \times (\frac{9}{1}+1) \times \frac{1}{27} \rfloor$.

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 in numerous problems concerning *changes of data representation*, of which conversions between number formats are a representative example. Consequently, we have been calling such algorithms *metamorphisms*.

5 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 arises because the process of regularizing a number in base $\mathcal{B}$ proceeds from right to left, hence 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 that 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, for 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

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

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, and t with $qt - rs \neq 0$—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, and 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+r) / fromInteger (s*x+t)
> unit :: LFT
> unit = (1,0,0,1)
> comp :: LFT -> LFT -> LFT
> comp (q,r,s,t) (u,v,w,x) = (q*u+r*w, q*v+r*x, s*u+t*w, s*v+t*x)
```

(The first line introduces the abbreviation `LFT` for the type of four-tuples of `Integer`s.)

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 map the interval $[3,4]$ onto converging 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 also holds for any tail of the infinite composition.)

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 identity 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 it is safe to commit to it.

```
> 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  = (n == floor (extr z 4))
>   prod z n  = comp (10, -10*n, 0, 1) z
>   cons z z' = comp z z'
```

The definition of `lfts` uses a list comprehension, with generator `[1..]`, the infinite list of `Integer`s from `1` upwards. 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

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

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 program by making various simple optimizations. These include: unfolding intermediate definitions; exploiting the invariant that the

bottom left element s of every linear fractional transformation $\left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)$ be 0; constructing the input transformations in place; representing the sequence of remaining transformations simply by the index `k`; and simplifying away one of the divisions.

Another optimization that can be performed is the elimination of any factors common to all four entries resulting from a matrix multiplication. 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.)

6 MORE STREAMING ALGORITHMS FOR THE DIGITS OF π.

The expression (2) turns out not to be a very efficient one for computation. Each term shrinks the range by a factor of about a half, so more than three terms are required on average for every digit of the output. Better sequences are known; the book π *Unleashed* [1] presents many. We conclude this paper with two more applications of the streaming technique from section 4 to computing π, using the same approach but based on two of these different expressions.

Lambert's expression. Here is a more efficient expression for π, due to Lambert in 1770 [1, eq. (16.99)], that yields two decimal digits for every three terms:

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

Again, one can view this as an infinite composition of linear fractional transformations:

$$\pi = \left(4\div\right)\left(1+1^2\div\right)\left(3+2^2\div\right)\left(5+3^2\div\right)\cdots\left(2i-1+i^2\div\right)\cdots \quad .$$

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

$$\left(2i-1+i^2\div\right)\left(2i+1+(i+1)^2\div\right)\left(2i+3+(i+2)^2\div\right)\cdots \quad ,$$

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 thus 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*i, 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 = (n == floor ((q*x+2*r) % (s*x+2*t)))
>                          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)
```

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

$$\pi = 3+\frac{1\times 1}{3\times 4\times 5}\times\left(8+\frac{2\times 3}{3\times 7\times 8}\times\left(\cdots 5i-2+\frac{i(2i-1)}{3(3i+1)(3i+2)}\times\cdots\right)\right) \quad . \tag{6}$$

Once more, we can view this as an infinite composition of linear fractional transformations, namely,

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

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 rise to the following program:

```
> piG = stream next safe prod cons init lfts where
>   init                  = ((1,0,0,1), 1)
>   lfts                  = [let j = 3*(3*i+1)*(3*i+2)
>                             in (i*(2*i-1),j*(5*i-2),0,j) | i<-[1..]]
>   next ((q,r,s,t),i)    = div (q*x+5*r) (s*x+5*t) where x = 27*i-12
>   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)
```

(The `let` here is another form of local definition.)

A challenge. Gosper's series (6) yields more than one digit per term on average, since the scaling factors approach $\frac{2}{27}$ (which is less than $\frac{1}{10}$) from below. This suggests that we could dispense with the test altogether and strictly alternate between consumption and production, as expressed by the following conjecture. Eliminating the test would speed up the algorithm considerably.

Conjecture 1. *Define the following functions:*

$$\begin{array}{lcl} n(z,i) & = & \lfloor z(\frac{27i-12}{5})\rfloor \\ p((z,i),n) & = & (\left(\begin{smallmatrix}10 & -10n\\ 0 & 1\end{smallmatrix}\right)z, i) \\ c((z,i),z') & = & (zz', i+1) \\ s((z,i),n) & = & (n = \lfloor z(\frac{675i-216}{125})\rfloor) \quad . \end{array}$$

For $i = 1,2\ldots$, let

$$\begin{array}{lcl} x_i & = & \left(\begin{smallmatrix} i(2i-1) & 3(3i+1)(3i+2)(5i-2) \\ 0 & 3(3i+1)(3i+2)\end{smallmatrix}\right) \\ u_0 & = & (\left(\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right), 1) \\ u_i & = & p(v_i, y_i) \\ v_i & = & c(u_{i-1}, x_i) \\ y_i & = & n(v_i) \quad . \end{array}$$

Then $s(v_i, y_i)$ holds for all i.

If this conjecture holds, then, in the following program, every value taken by the variable `v` satisfies the condition `safe v (next v)`.

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

(Here, `next`, `prod`, `cons`, `init`, and `lfts`, as well as the predicate `safe` mentioned in the claim, are as in section 6.)

We have not been able to prove Conjecture 1, although we have verified it for the first thousand terms. Perhaps some diligent reader can provide enlightenment. If it is valid, then `piG2` does indeed produce the digits of π, and the optimizations outlined at the end of section 5 can be applied to `piG2`, yielding 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)
```

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

ACKNOWLEDGMENTS. Thanks are due to the anonymous referees, the Algebra of Programming research group at Oxford, Stan Wagon, Sue Gibbons, and especially to Christoph Haenel, who suggested Conjecture 1. Frank Taylor reported an error in the program `piG` and the statement of Conjecture 1. All of them have made suggestions that have improved the presentation of this paper.

REFERENCES.

[1] J. Arndt and C. Haenel, π *Unleashed*, 2nd ed., Springer-Verlag, Berlin, 2001.

[2] R. P. Brent, Fast multiple-precision evaluation of elementary functions, *J. ACM* **23** (1976) 242–251.

[3] J. Gibbons. Streaming representation-changers, in *Mathematics of Program Construction*, D. Kozen, ed., Springer-Verlag, Berlin, 2004 pp. 142–168.

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

[5] S. Peyton Jones, ed., *Haskell 98 Language and Libraries: The Revised Report*, Cambridge University Press, Cambridge, 2003.

[6] Haskell web site. `http://www.haskell.org/`.

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

[8] S. Rabinowitz and S. Wagon, A spigot algorithm for the digits of π, *Amer. Math. Monthly* **102** (1995) 195–203.

[9] E. Salamin, Computation of π using arithmetic-geometric mean, *Math. Comp.* **30** (1976) 565–570.

JEREMY GIBBONS is a University Lecturer in Software Engineering and Continuing Education at the University of Oxford. He obtained his doctorate at Oxford in 1991, then spent five years at the University of Auckland in New Zealand and three years at Oxford Brookes University, before returning to the University of Oxford. He now teaches on the professional postgraduate Software Engineering Programme (*http://www.softeng.ox.ac.uk/*). His research interests are in programming languages and methods, particularly in the functional and object-oriented paradigms, and in recurring patterns in the structure of computer programs.

Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK
jeremy.gibbons@comlab.ox.ac.uk
http://www.comlab.ox.ac.uk/jeremy.gibbons/

BRIEF DESCRIPTIVE SUMMARY. Rabinowitz and Wagon (in the April 1995 issue of the MONTHLY) 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. Rabinowitz and Wagon's algorithm is inherently *bounded*; it requires a commitment in advance to the number of digits to be computed. We propose some *streaming algorithms* based on the same and some similar characterizations of π, with the same incremental characteristics, but without requiring the prior bound.
jeremy.gibbons@comlab.ox.ac.uk