6. Spigot Algorithms

The *spigot algorithm*, developed by Stanley Rabinowitz and Stanley Wagon [95], is a recent and elegant method of calculating π. The algorithm is ideal for running on a personal computer.

1. The spigot algorithm starts producing digits of π right from start-up and thereafter churns them out at regular intervals. Under all the other methods, π has to be calculated completely in a buffer and is only output at the end, all at once. By contrast, with the spigot algorithm the π digits trickle out one at a time. It is possible to watch it at work, so it is well suited for online demonstrations on the Internet, for example. The CD-ROM which goes with this book contains the Java applet `spigot/pispigot.htm`, which shows the algorithm in action. You only need to load it into a standard Java-capable browser.
2. The algorithm works with nice small integers; even for 15,000 digits of π the values of its variables do not climb to more than 32 bits (including plus and minus signs), so that the `long` C data type is sufficient on standard 16-and 32-bit compilers. This means there are no problems with rounding, deletion or truncation, which can make life miserable with other algorithms.
3. Implementation of the spigot algorithm does not require any extraneous software, such as, for example, a high precision library. Everything you need will be found in any standard C compiler.
4. The spigot algorithm is surprisingly fast. Although the time it requires is of quadratic order, so that it cannot compare with high-performance algorithms such as the Gauss AGM algorithm (see page 91), nevertheless it regularly outdoes algorithms that are based on arctan series (see page 69).
5. The mathematics behind the spigot algorithm is simple.

6. The spigot algorithm can be written in only a few lines of source code. The shortest π programs are based on it. A proof of this has already been provided above (see page 37).

6.1 The spigot algorithm in detail

The starting point is the following simple structured series for π:

$$\pi = 2 + \frac{1}{3}(2 + \frac{2}{5}(2 + \frac{3}{7}(2 + \cdots))) \tag{6.1}$$

It can be derived from the Leibniz series (16.59) without great effort if one just uses the Euler transformation [76, p. 255]. However, we will not perform that step now.

The series (6.1) can be conceived of as a number in a number system with a variable base. Normally we encounter only numbers with a fixed base, which is usually base 10. Every place of such a number must be multiplied with a value which is higher than the value of the next digit to the right, by a constant factor known as the "base".

But occasionally we come across numbers in which this factor is not constant, e.g. the numbers which correspond to the expression "2 weeks, 3 days, 4 hours and 5 minutes". Because the ratio of weeks to days = 1 : 7, days to hours = 1 : 24 and hours to minutes = 1 : 60, when converting this number to a decimal number (with the unit "weeks"), three different factors have to be considered, namely 1/7, 1/24 and 1/60. Hence, to answer the question how many weeks are in the example, we have to calculate the following:

$$2 + \frac{1}{7}(3 + \frac{1}{24}(4 + \frac{1}{60}(5))) \tag{6.2}$$

Now compare this expression with the right-hand side of the π series (6.1). You will see the same arrangement of brackets and also the different bases. On the other hand, in the π series all the numerals are equal, i.e. = 2, whereas in the week example they are different, i.e. = 2, 3, 4 and 5; the π series is also infinite, whereas the example terminates after the fourth digit.

Back to π. The task which the spigot algorithm performs consists simply of converting the π series (6.1) to our base 10 number system, i.e. to this form:

$$\pi = 3.1415\ldots = 3 + \frac{1}{10}(1 + \frac{1}{10}(4 + \frac{1}{10}(1 + \frac{1}{10}(5 + \cdots)))) \tag{6.3}$$

Such a task is known in arithmetic as *radix conversion* and functions – in the present case – as follows:

On every step one π decimal place is calculated. To do this, first of all all the digits in the number to be converted are multiplied by 10 (the new base). Then, starting from the right, every decimal place is divided by the previous base $(2i + 1)/i$ which applies to this decimal place. On every division, the remainder is retained and the integer quotient is carried over to the next decimal place. The most recently calculated carry-over is the new decimal place of π.

The question arises as to how many terms in the π series (6.1) really have to be carried over in order to obtain n decimal places of π, including the digit 3 before the decimal point. In their article, Rabinowitz and Wagon specify the value $\lfloor 10n/3 \rfloor$, where the notation $\lfloor x \rfloor$ in the customary manner refers to the largest integer $\leq x$, i.e. it means $\lfloor 10/3 \rfloor = 3$ and $\lfloor 3 \rfloor = 3$. They even "prove" that this value is "correct". Unfortunately, it is not correct, as they could have seen from the values $n = 1$ and $n = 32$. We are taking the liberty of correcting the mistake and take one more decimal place, i.e. $\lfloor 10n/3 + 1 \rfloor$ decimal places, in the (tested) supposition that we are then correct in every case.

Before we can begin to programme, we must explain the single real complication in the algorithm.

When the base of the π series (6.1) is changed, it is possible for a digit pair $= 10$ to occur. It can thus happen that at some point a 10 occurs at position p, i.e. $3.1415 \ldots (p)(10)$. The 1 in the 10 is an unresolved carry-over and must be added to the previous digit i.e. $3.1415 \ldots (p + 1)0$. It can even happen that in front of such a 10 one or more 9s have been calculated, so that those places too must be corrected. The $3.1415 \ldots (p)99 \ldots 9(10)$ must then be transformed into $3.1415 \ldots (p + 1)00 \ldots 00$.

This complication means that the π program cannot release the digits it has calculated immediately, but must store them in a buffer until the next digit or the next few digits has/have been calculated. When a new digit arrives, one or more digits will still be held in the buffer. The first of these will definitely be < 9 and the others, if there are any, exactly $= 9$. In this way we have the following situation in the buffer at the point when the new digit q arrives:

$$\underbrace{p99\ldots9} \longleftarrow q$$

There are now three possibilities:

1. $q < 9$: it turns out not to have been necessary to retain the value in the buffer. p and any succeeding $99\ldots9$ can be output as they are. q becomes the next decimal digit of p.
2. $q = 9$: no decision has been made so the number of 9s held in the buffer is increased by 1 as q is added to it.
3. $q = 10$: it turns out to have been necessary to store the intermediate values in the buffer because a 1 must now be added to the sequence of digits held in the buffer. As a result, p is increased by 1, and all the 9s held temporarily become 0s. All these decimal places are now ready and can be output, but the 1 in the 10 is stored temporarily, i.e. p is set to 0.

The complication thus arises because the π series (6.1) is not unique in relation to the second positions. For example, $\frac{2}{3}$ can be represented in two ways, namely by $0+\frac{1}{3}(0+\frac{2}{5}(2+\frac{3}{7}(3+\cdots)))$ and by $0+\frac{1}{3}(2)$. There are series for π which are unique, but they are much more long-winded to calculate than (6.1).

6.2 Sequence of operations

We have now assembled all the elements required to describe the sequence of operations in the spigot algorithm.

The spigot algorithm calculates the first n decimal places of π. It works with a field $a[0], a[1], \ldots, a[N]$ consisting of $N+1 = \lfloor(10n)/3\rfloor+1$ integers, where $\lfloor x \rfloor$ signifies the largest integer $\leq x$. Moreover, two variables p and q are used to record the first and the current provisional decimal place, also a numerator *nines* for the number of temporary nines.

Initialisation: set $p = 0$ and $nines = 0$.
For $i = 0, 1, 2, \ldots, N$, set $a[i] = 2$.
Iteration: repeat until n decimal places have been output.

- *Multiply with the new base.* Multiply each $a[i]$ by 10.
- *Normalise.* Beginning on the right, from $i = N$ to $i = 1$, divide $a[i]$ by $(2i+1)$ to obtain a quotient q and a remainder r. Replace $a[i]$ with r. Multiply q by i and add the result (the carry-over) to the element $a[i-1]$.

- *Calculation of the next provisional digit of* π. The left-hand digit $a[0]$ is post-processed. It is divided by 10. The division remainder replaces $a[0]$, while the quotient q produces the next provisional digit of π.
- *Correct the old provisional digits.* If q is neither 9 nor 10, then the first provisional digit up to now, p, and the *nines* which succeed it are confirmed and output. The new first provisional digit p now becomes $= q$, and *nines* is set to $= 0$.
  If $q = 9$, then out of the provisional nines only the number *nines* is incremented by 1; no digits are output.
  If $q = 10$, the first provisional digit up to now, p, is increased by 1 and output. The provisional *nines* become zeros; they are likewise output. The new first provisional digit, p, becomes $= 0$ (that is the 1st digit of q). *nines* is reset to 0.

A small improvement can be achieved if one initialises the first provisional digit of p with a negative value and then intercepts this value during output. This means that the output begins immediately with 314... instead of initially with a zero.

The following C function `spigot()` is described in the sequence of operations below.

```
/*
* function
*       void spigot(digits)
* Spigot program for pi
* 1 digit per loop
*/

#include <stdio.h>
#include <stdlib.h>

void spigot(int digits)
{
   int   i, nines = 0;
   int   q,                   /* next prelim. digit          */
         p = -1;              /* previous prelim. digit      */
   int   len = 10*digits+3+1; /* len: One more than R+W      */
   int   *a;                  /* array pointer               */

   a = malloc(len*sizeof(*a));
   for (i=0; i < len; ++i)    /* Init a[] with 2's           */
      a[i] = 2;
   while (digits >= 0)
   {                        /* Compensate for the very first digit */
      q = 0;
      for(i=len; --i >= 1; )
      {
         q += 10L * a[i];    /* q = carry + 10*a[i]          */
         a[i] = q % (i+i+1); /* a[i] := q % (2i+1)           */
         q /= (i+i+1);           /* carry := floor(q,2i+1)*i         */
         q *= i;
      }
                                 /* first digit                      */
      q += 10L * a[0];           /* q := carry + 10 * a[0]           */
      a[0] = q % 10;             /* a[0] = q mod 10                  */
      q /= 10;                   /* q : next prelim digit            */
      if (q == 9)
         ++nines;                /* q == 9: increment no of 9's      */
      else
      {                          /* q != 9: print prelim. digits     */
         if (p >= 0)
            printf("%011d", p + q/10);   /* p : prev. prel. digit */
         if (digits < nines)     /* adjust digits to print          */
            nines = digits;
         digits -= (nines+1);
         while (--nines >= 0)    /* print 9's or 0's                 */
            printf(q == 10? "0" : "9");
         nines = 0;
         p = (q == 10 ? 0 : q);  /* set previous prelim. digit      */
      }
   }
   free(a);
   return;
}
```

At the end of the much-cited article [95] by Rabinowitz and Wagon there is a PASCAL program which evidently was not written by the authors themselves but by a student, to whom they express their thanks. This program has also been typed out and tested by some readers of this book. They had various problems with it, for example, on 16-bit PASCAL compilers the program stops working from $n > 262$ because an integer overflow occurs at that point, or else at $n = 1$ and $n = 32$ it prints an incorrect last digit because the series length is too short, or else in many cases of n it outputs fewer than n digits because variable `nines` at the end of the program is not yet 0. Our above program attempts to avoid these weaknesses.

6.3 A faster variant

Two improvements can be made to the spigot algorithm which make it considerably shorter and quicker to run.

First of all, instead of π, 1000π is calculated, and hence the series

$$1000\pi = 2000 + \frac{1}{3}(2000 + \frac{2}{5}(2000 + \frac{3}{7}(2000 + \cdots))) \qquad (6.4)$$

is used. The conversion is thus not performed in base 10 but in base 10,000, so that on every pass the program produces four decimal places instead of only one.

This trick not only has the effect that the entire program becomes 4 times faster, but an even more important effect is that the "complication" mentioned above becomes a lot simpler. The faster variant always waits only exactly 1 position (consisting of 4 digits) instead of having a variable number of places. This is sufficient for the first approx. 50,000 digits of π, as up to that point there is no case in which more than one such 4-digit chain has to be placed in a buffer. This is only necessary if 4 zeros occur one after the other in a position that is divisible by 4, and the first time that this occurs in π is at digit position 54,936.

The second improvement is really a textbook tip which, however, was evidently forgotten in the original formulation of the spigot algorithm. Namely, with a radix conversion, after each result place, one can shorten the remainder that still has to be converted, by the number of bits of the result place. Hence after each calculation of one place consisting of 4 decimal digits, the length of the `f[]` field can be reduced by $\lfloor 10\cdot 4/3+1\rfloor = 14$ places. This improvement speeds up the program by an additional factor of 2.

Here is a C program for the faster variant. It is an expanded version of the mini-program shown on page 37.

```
/*
* Spigot program for pi to NDIGITS decimals
* 4 digits per loop
* Expanded version
* Thanks to Dik T. Winter and Achim Flammenkamp.
*/

#include <stdio.h>
#include <stdlib.h>

#define NDIGITS 15000          /* max. digits to compute  */
#define LEN   (NDIGITS/4+1)*14 /* nec. array length       */

long a[LEN];                   /* array of 4 digit-decimals*/
long b;                        /* nominator prev. base    */
long c = LEN;                  /* index                   */
long d;                        /* accumulator and carry   */
long e = 0;                    /* save prev. 4 digits     */
long f = 10000;                /* new base, 4 dec. digits */
long g;                        /* denom prev. base        */
long h = 0;                    /* init switch             */

int main(void)
{
   for ( ; (b=c-=14) > 0; )    /* outer loop:4 digits/loop*/
   {
      for (; --b > 0; )        /* inner loop: radix conv  */
      {
            d *= b;                 /* acc *= nom. prev base    */
            if (h == 0)
               d += 2000 * f;       /* first outer loop         */
            else
               d += a[b] * f;       /* non-first outer loop     */
            g=b+b-1;                /* denom prev. base         */
            a[b] = d % g;
            d /= g;                 /* save carry               */
         }
         h = printf("%04ld", e+d/f);  /* print prev 4 digits */
         d = e = d % f;             /* save current 4 digits    */
                                    /* assure a small enough d */
      }
      return 0;
   }
```

On the basis of the `NDIGITS` definition, this program calculates exactly 15,000 decimal places of π. There is no reason why it has to stop at this limit if one is not concerned about the portability of the program and compliance with ANSI C. As most C compilers around ignore an *overflow* in the evaluation of integer expressions which depend on plus and minus signs and there is sufficient "breathing space" in the array length, this program can normally be expanded to calculate twice as many π positions (using `#define NDIGITS 32500`). This number was still a world record 40 years ago. Even more decimal places can be calculated with floating point variables and arithmetic.

But however large a data range you select, even with this variant of the spigot algorithm, i.e. where each place consists of only 4 π digits and only one such place is held in the buffer, it is still not possible to get past π digit position 54932. This can only be achieved by removing at least one of these limiting conditions; the easiest one to remove is the first.

6.4 Spigot algorithm for *e*

The spigot algorithm is clearly not limited to the calculation of π.

Let us consider the transcendental number $e = 2.7182\ldots$. Its series expansion which is the equivalent of the above π series (6.1) goes as follows:

$$e = 1 + \frac{1}{1}(1 + \frac{1}{2}(1 + \frac{1}{3}(1 + \cdots))) \qquad (6.5)$$

Here, as in the π series, different bases occur, but this time they are all of the type that their numerator = 1. This means that when calculating the carry-over, there is no need for a multiplication operation.

More important, however, is the fact that the e series (6.5) is unique, so that the complication which occurs with π does not occur here. A spigot program for the decimal places of e is therefore simpler.

Here is a 138-character long program for the calculation of e to 15,000 places in the style of the mini-π program shown on page 37:

```
/* note: N=15000, LEN=87700 >= 1.4*N*log10(N), 84700=LEN-N/5 */
a[87700],b,c=87700,d,e=1e4,f=1e5,h;
main(){for(;b=c--,b>84700;h=printf("%05d",e+d/f),e=d%=f)
for(;--b;d+=f*(h?a[b]:e),a[b]=d%b,d/=b);}
```

Back on page 36 we promised to tell you how Lievaart's "obfuscated" program works. Well, it too works with the spigot algorithm for e.