

The calculation of e to many significant digits

By A. H. J. Sale*

The interesting problem of accurate evaluation of the number e is discussed, and a new technique proposed which evaluates the digits of e (to any base) in sequence. The technique is easily programmed in a high-level language, and an ALGOL 60 algorithm is given.

(First received November 1967, and in revised form April 1968)

The calculation of the transcendental numbers e and π to a ridiculously high number of significant digits has always been a hobby of mathematicians and computer programmers. There appear to be only three good reasons for ever computing these numbers: to investigate properties of the digital sequence produced; to illustrate the accuracy that the digital principle of operation makes possible, whether it is normally utilised or not; and as a programming exercise. The calculation of e is in fact an excellent example for students to program, and almost always involves temporary storage, loops, and an iterative or counting termination. The answer is, of course, very easily checked.

Normally e is evaluated from the simple and well-known infinite series:

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \dots$$

This series has a fast convergence and is easily computed, as the terms may be easily computed from the previous term, as shown by the table below:

$$\begin{aligned} e_0 &:= 1 \\ e_1 &:= e_0 + e_0/1 \\ e_2 &:= e_1 + (e_0/1)/2 \\ e_3 &:= e_2 + ((e_0/1)/2)/3 \end{aligned}$$

Of course a computer word cannot usually accommodate more than about 10–15 decimal digits, and for more accuracy some sort of multiple-word arithmetic would have to be planned. It will also be obvious that as the terms get smaller, the truncation error when they are added to the successive approximants for e will become larger.

It is possible to reduce this error quite simply, by evaluating the series from the other end. This requires that the series be first truncated, for an infinite number of terms is unacceptable. This is a change of programming technique: first the required number of terms is computed, and then the series evaluated—instead of an iterative loop with a testing exit. Examination of the regrouped series for e , given below, shows what is the well-known method of rewriting polynomials

in a slightly different guise:

$$e = 1 + 1\left(1 + \frac{1}{2}\left(1 + \frac{1}{3}\left(1 + \frac{1}{4}\left(1 + \dots\left(1 + \frac{1}{n}\dots\right)\right)\right)\right)\right)$$

This is probably the fastest method of evaluating e , given a suitable multiple-word division routine, since each addition is only of 1, and not therefore of multiple-length accuracy.

Further examination of the e -series in its nested evaluation form shows that yet a third distinct method of evaluation is possible. This method has the advantage that it can be simply and economically implemented in a high-level language, such as ALGOL 60 or FORTRAN. (As an incidental benefit it can then be easily used as a demonstration program.) The method also has the characteristic that it produces the digits for e one-by-one, instead of a final answer with the entire value. In a slow computer it can be arranged that the computation time is entirely or partially masked by the output delay, and an effective speed increase can be demonstrated. This approach has been very effective on the IBM 1620, which, being a character-oriented machine, is also ideally suited to the normal method of evaluation.

The method depends on the fact that except for the first two terms of the series, each successive term, and their sum, is less than 1. Suppose then that a truncated portion of the series beyond the first two terms be multiplied by 10: it will then consist of an integer part (in this case 7) and a fractional remainder series. The integer can be removed (and possibly printed), and the process repeated, giving successively the fractional digits of e . This process is most easily understood from the example:

$$\begin{aligned} e &= 2 + \left[\frac{1}{2} \left(1 + \frac{1}{3} \left(1 + \frac{1}{4} \left(1 + \frac{1}{5} \left(1 + \frac{1}{6} \left(1 + \dots \right) \right) \right) \right) \right) \right] \\ &= 2 + \frac{1}{10} \left[\frac{1}{2} \left(10 + \frac{1}{3} \left(10 + \frac{1}{4} \left(10 + \frac{1}{5} \right) \right) \right) \right] \end{aligned}$$

* University of Natal, King George V Avenue, Durban, South Africa.

Calculation of e

$$\begin{aligned}
 & \left(10 + \frac{1}{6}\left(10 + \dots \right)\right)\right)\right) \\
 = & 2 + \frac{1}{10} \left[7 + \frac{1}{2} \left(0 + \frac{1}{3} \left(1 + \frac{1}{4} \left(0 + \frac{1}{5} \right. \right. \right. \right. \\
 & \left. \left. \left. \left. \left(1 + \frac{1}{6} \left(5 + \dots \right) \right) \right) \right) \right) \right) \right] \\
 = & 2.7 + \frac{1}{100} \left[\frac{1}{2} \left(0 + \frac{1}{3} \left(10 + \frac{1}{4} \left(0 + \frac{1}{5} \right. \right. \right. \right. \right. \\
 & \left. \left. \left. \left. \left(10 + \frac{1}{6} \left(50 + \dots \right) \right) \right) \right) \right) \right) \right] \\
 = & 2.7 + \frac{1}{100} \left[1 + \frac{1}{2} \left(1 + \frac{1}{3} \left(1 + \frac{1}{4} \left(3 + \frac{1}{5} \right. \right. \right. \right. \right. \\
 & \left. \left. \left. \left. \left(4 + \frac{1}{6} \left(2 + \dots \right) \right) \right) \right) \right) \right) \right]
 \end{aligned}$$

This, of course, neatly sidesteps all problems of calculating e in binary notation and then being faced with a conversion problem to decimal notation. It is very simple to calculate e then to any desired base, for example to base 7, or base 16, by performing the multiplication of the fractional part of the series by the desired base. For binary computers a base of the form 2^n can of course lead to extremely simple multiplication by shifting. An important practical point is that if the language precision permits, the multiplication (for decimal output), may be by 10 or 1000, thus producing two or three digits at a time, increasing the efficiency. Typically: if a maximum integer size of 32767 is specified, the algorithm allows the series to include all terms up to $1/3276!$, using a base of 10.

This method is interesting, and may in principle be employed in evaluating other series, for example $\sin(x)$. This, however, occasionally gives a negative digit, indicating corrections to the previous digits, and thus the digits are not necessarily correct as they are computed.

A high degree of machine independence is also achieved, since only integer arithmetic is involved in the evaluations. This is therefore an alternative to the procedure given by Naur (1967) to illustrate the concept of machine environmental enquiries.

It is also likely that the highest accuracies for e can be obtained this way, at least theoretically, for if the memory size of a computer is k words then the available accuracy in the normal method grows proportionally to k , while with this method it grows as $k!$.

References

SHANKS, D., and WRENCH, J. W., JR. (1962). Calculation of π to 100,000 decimals, *Math. Comput.*, Vol. 16, p. 76.
 YARBROUGH, L. (1967). Precision calculations of e and π constants, *Comm. ACM*, Vol. 10, no. 9, p. 537.
 NAUR, P. (1967). Machine Dependent Programming in Common Languages, *Nordisk Tidsskrift for Informationsbehandling* Vol. 7, no. 2, pp. 123-131.

The ALGOL 60 algorithm given below is self-contained, and evaluates n decimal digits of e by this method. It estimates the number of terms required in the series to give $n + 1$ correct digits by using Stirling's approximation for $m!$, increasing m from an initial value of 4 until $m! > 10 \uparrow (n + 1)$. It will be obvious that this arithmetic, even if performed in real mode, is very likely to overflow the dynamic range of a real number. The procedure therefore uses a logarithmic evaluation and test. In practice it selects 73 terms of the series as sufficient for 100 decimal digits.

The algorithm

```

procedure ecalculation (n, d);
value n;
integer n;
integer array d;
comment This procedure for calculating the transcendental
number e to n correct decimal places uses only integer
arithmetic, except for estimating the required series
length. The digits of the result are placed in the array
d, the array element d[0] containing entier(e), and the
subsequent elements the following digits. These digits
are individually calculated and may be printed one-by-
one within the for statement labelled 'sweep'.
begin integer m;
real test;
m := 4;
test := (n + 1) × 2.30258509;
loop: m := m + 1;
if m × (ln(m) - 1.0) + 0.5 × ln(6.2831852 × m)
< test then go to loop;
begin integer i, j, carry, temp;
integer array coef [2 : m];
for j := 2 step 1 until m do coef [j] := 1;
d [0] := 2;
sweep: for i := 1 step 1 until n do begin
carry := 0;
for j := m step -1 until 2 do begin
temp := coef [j] × 10 + carry;
carry := temp ÷ j;
coef [j] := temp - carry × j
end of digit generation;
d [i] := carry
end having calculated n digits
end deleting declarations
end of ecalculation;
    
```