Special Series Summation with Arbitrary Precision

S. Kamal Abdali

The Algol 60 code of the algorithm as listed in the cited article appears below after the publication information.

---

procedure *series* (*places*, *terms*, *base*, *digit*, *sgn*, *numerator*, *denominator*, *num*0, *denom*0);
    value *places*, *terms*, *base*; **integer** *places*, *terms*, *base*, *sgn*, *num*0, *denom*0;
    integer array *digit*; **integer procedure** *numerator*, *denominator*;
comment Programs for very precise summation of series are conventionally written in machine language and employ multi-precision routines to perform arithmetic on especially defined multiword registers. The present algorithm requires only integer arithmetic and can be implemented in any algebraic language. It is applicable to series in which the ratios of successive terms can be expressed as quotients of given integers or integer functions of term positions.

    The sum of a given series is computed to a given number of places, *places*, in a specified base for representation, *base*. The number of terms needed, *terms*, should be calculated outside the procedure. Procedures *numerator* and *denominator* are to be obtained from the fraction ith term/$(i-1)$-th term, expressed as a ratio of two integer functions of i. (That fraction should preferably be reduced to its lowest terms.) *num*0 and *denom*0 are the integer numerator and denominator of the 0th term. The outputs of the procedure are the sign of the result, *sgn*, the integer part, *digit*[0], and the digits of the fractional part, *digit*[1], $\cdots$, *digit*[*places*].

    For example, one way to compute $\sin 0.6 = .6 - .6^3/3! + 6^5/5! - \cdots$ correct to 1000 decimal places is to call *series* with the parameter values: *terms* = 226, *num*0 = 3, *denom*0 = 5, (and since ith term/$(i-1)$-th term $= -.6^2/2i(2i+1)$) *numerator*(i) $= -9$ and *denominator*(i) $= 50i(2i+1)$. By taking *base* = 100000 and *places* = 200, five decimal digits of the result will be obtained per word of the array *digit*.

    The use of a large *base* (and, consequently, smaller *places*) results in faster computation, as the number of operations is proportional to (*places* × *terms*) for large values of *terms* and *places*. However, the intermediate products (*base* × *num*[i] × *coef*[i]) (and *coef*[i] can almost

equal *denom*[*i*]) should not exceed the largest number representable by an integer variable. Also within this limit should be the product of *base* and the integer portion of the result;

begin

    integer *i, j, k, l*;   **integer array** *num*[−1:*terms*], *denom*, *coef*[0:*terms*];

    comment Express the series by the expression

$$\frac{n_0}{d_0}\left(c_0 + \frac{n_1}{d_1}\left(c_1 + \cdots + \frac{n_t}{d_t}(c_t)\cdots\right)\right) \tag{1}$$

    where n_i and d_i are positive and c_i are ± 1. (For short, *n, d, c* and *t* in (1) stand for *num*, *denom*, *coef* and *terms*, respectively);

    num[−1] := 1;   *num*[0] := *abs*(*num*0);   *denom*[0] := *abs*(*denom*[0]);

    coef[0] := *sign*(*num*0) × *sign*(*denom*0);

    for *j* := 1 **step** 1 **until** *terms* **do**

    begin

        k := *numerator*(*j*);   *l* := *denominator*(*j*);   *num*[*j*] := *abs*(*k*);

        denom[*j*] := *abs*(*l*);   *coef*[*j*] := *coef*[*j*−1] × *sign*(*k*) × *sign*(*l*)

    end;

    comment Calculate digits one at a step by extracting the integer part of base × (1) and restoring the fractional part in form (1);

    for *i* := 1 **step** 1 **until** *places* **do**

    begin

        l := 0;

        for *j* := *terms* **step** −1 **until** 0 **do**

        begin

            k := *num*[*j*] × (*coef*[*j*]×*base*+*l*);   *l* := *k* ÷ *denom*[*j*];

            coef[*j*] := *k* − *l* × *denom*[*j*];   *num*[*j*] := *num*[*j*−1]

        end *j*;

        digit[*i*] := *l*

    end *i*;

    comment Some digits may be negative or larger than *base* in absolute value. Process the array *digit* to obtain true base representation;

    l := 0;

    for *i* := *places* **step** −1 **until** 1 **do**

    begin

        k := *digit*[*i*] + *l*;   *l* := *k* ÷ *base*;   *digit*[*i*] := *k* − *base* × *l*;

        if *digit*[*i*] < 0 **then**

        begin *digit*[*i*] := *digit*[*i*] + *base*;   *l* := *l* − 1 **end**

    end;

    digit[0] := *l*;   *sgn* := *sign*(*l*);

    if *l* < 0 **then**

    begin

        digit[0] := −*l* − 1;   *digit*[*places*] := *digit*[*places*] − 1;

        for *i* := 1 **step** 1 **until** *places* **do** *digit*[*i*] := *base* − 1 − *digit*[*i*]

    end

end *series*