KYBERNETIKA — VOLUME 19 (1983), NUMBER 4

A FAST FLOATING-POINT SQUARE-ROOTING ROUTINE FOR THE 8080/8085 MICROPROCESSORS

VALENTIN CHAMRÁD

A speed-oriented implementation of the Newton-Raphson algorithm is described, reducing the worst-case execution time to the level of standard floating-point multiplication and thus supporting a wider use of square-root filters in microprocessor-based self-tuning controllers.

1. INTRODUCTION

Square-root is a function for which numerous numerical methods have been developed. In most math packages for microprocessors, simple iterative methods have been used, as no special demands for speed — nor even for accuracy in some cases — are expected: e.g. in [1] and [2] the execution time of square-rooting is approx. 2·5 times longer than that of multiplication, and in [3] a quintuple error limit compared with other operations is accepted. Some floating-point packages do not support this function at all leaving its evaluation to user defined programs (e.g. [4]).

In the floating-point subroutine package for the Intel 8080/8085 microprocessors [6] developed in the Institute of Information Theory and Automation in Prague, the speed of operation has been strongly emphasized; and as a frequent use in software for self-tuning controllers with square-root filters has been expected, there was a special demand for a fast square-rooting subroutine with the same accuracy (± 1 LSB[1]) as with all the other operations. It took a considerable effort to match this condition, and every promising method of accelerating the calculation was tested — even empirical or intuitive; special testing programs were developed for this purpose, checking the real deviation of the square-root returned by the subroutine under test for all the 32 k significantly different input values and printing those values yielding results with an error exceeding a preset limit of 0·5, 0·75 or 1 LSB only.

[1] The abbreviations MS and LS will be used for *most significant* and *least significant* respectively in this paper; in connection with them, B will be reserved for *bit*, while *byte* will not be abbreviated.

2. NUMBER FORMAT

The first important step to improve the performance of a floating-point subroutine is to realize some special properties of the number format used; in our case, this is defined for every representable number as

$$x = a \,.\, 2^b\,, \quad 0{\cdot}5 \leqq |a| < 1 \tag{1}$$

where a (mantissa) is a 16 bit FRACTION number in two's complement form, and b (exponent) is a 7 bit INTEGER in the "excess-64" code, i.e. with an added offset of 64 to avoid negative values, so that the real value stored in the exponent byte is

$$b' = b + 64\,, \quad 0 \leqq b' \leqq 127 \tag{1a}$$

which enables us to use the MSB for overflow/underflow detection. As explained in [6], the precision of 0·003% and the range of representable numbers approx. $\pm 3 \,.\, 10^{-19}$ to $\pm 10^{+19}$ proved to be quite sufficient for most engineering applications; on the other hand, the achievable execution speed of arithmetic subroutines is much higher compared to longer formats due to the possibility of using register instructions only for most operations.

With respect to square-rooting, the first important thing to realize is that we deal with a *product* of two numbers, the second of them being a power of two; the operation thus can be simplified by a conversion – may be fictive only – to an unnormalized format with the next higher even exponent, which can be square-rooted by an integer division by 2. If we denote the input operand x and the result y, then

$$b_y = \mathrm{INT}\left[\tfrac{1}{2}(b_x + 1)\right] \tag{2}$$

where INT denotes integer part of the expression in square brackets. In the format used, the result exponent will be computed as

$$b'_y = \mathrm{INT}\left[\tfrac{1}{2}(b'_x + 65)\right] \tag{2a}$$

Division by the shift right (RAR) instruction yields the *Carry* bit (LSB of the sum in parentheses) representing the directive for denormalizing the mantissa; we shall see later that a different treatment of mantissa instead of real denormalization will be more useful. A simple analysis of the limit values of b'_y shows that neither overflow nor underflow can occur; no final testing of exponent will therefore be needed.

3. THE ALGORITHM AND ITS CODING

The square-rooting algorithm proper will then operate on numbers in the range

$$0{\cdot}25 \leqq a_x < 1 \tag{3}$$

only; the results shall lie within the range of

$$0{\cdot}5 \leqq a_y < 1 \tag{4}$$

and that means that they will be automatically normalized; consequently, no final normalization will be needed.

Halving of the result range, together with the same resolution of 15 bits for both the input and result values and with the nonlinearity of the function, causes that we shall get the same results for two or even three adjacent input values, which should not be considered erroneous.

For square-rooting of mantissa, we have adopted the Newton-Raphson iteration formula, used in [2] and [3] as well, which in the ith iteration computes the new approximation y_{i+1} as

$$y_{i+1} = \tfrac{1}{2}\left(y_i + \frac{x}{y_i}\right) \tag{5}$$

i.e. as the mean value of the old approximation y_i and the quotient of the input value and the old approximation. For the known deviation of the ith approximation

$$\Delta y_i = y - y_i \tag{6}$$

where y is the correct value of the square root, the deviation of the next step can be estimated as

$$\Delta y_{i+1} = \frac{(\Delta y_i)^2}{2(y - \Delta y_i)} \tag{7}$$

As a rule, in conventional computers the iteration cycle starts for simplicity with $y_0 = x$, and the iteration process is stopped when the difference between two successive values of y_i is lower than the accuracy required. A similar method – used in our testing programs – determines the accuracy of the estimate using the difference between y_i and the quotient computed when evaluating formula (5); using (6), this difference d_i equals

$$d_i = y_i - \frac{x}{y_i} = y + \Delta y_i - \left(\frac{x}{y + \Delta y_i} + \Delta q_t\right) \tag{8}$$

where Δq_t is the truncation error of the quotient. For $\Delta y_i \ll y_i$ we can approximate

$$d_i = \frac{\Delta y_i(2y + \Delta y_i) - \Delta q_t(y + \Delta y_i)}{y + \Delta y_i} \doteq 2\,\Delta y_i - \Delta q_t \tag{8a}$$

and if we assume Δq_t to be small enough (which is satisfied by extended precision in our test programs), we can take

$$\Delta y_i = \tfrac{1}{2} d_i \tag{8b}$$

Note that the last but one approximation is tested here, so that an accuracy exceeding the precision of the format used can theoretically be achieved with the final result.

The possibilities of reducing the overall execution time of this iterative process comprise both reducing the execution time of a single iteration cycle and reducing the total number of iterations. For the latter way, the only means of reduction is a better original estimate, which could be constructed simply enough. Practically the choice is limited to a linear function, as any more complicated function (e.g. polynomial) would consume more time than another iteration cycle. Let us remind that this estimate should be constructed using a still normalized mantissa and the *Carry* bit representing an even or odd exponent; in other words, the *Carry* bit tells us whether the mantissa belongs to the "lower octave" of operands described by

$$Carry = 0\,, \quad 0{\cdot}25 \leqq x_L < 0{\cdot}5, \quad x_L = 0{\cdot}5a_L \tag{9}$$

or to the "higher octave" with

$$Carry = 1\,, \quad 0{\cdot}5 \leqq x_H < 1\,, \quad x_H = a_H \tag{10}$$

We found advantageous to choose different estimates for each octave: in fact, we used the results of the first iteration, taking the known correct values in the end points of interval (3) as primitive estimates, but we used a direct method to construct them.

For the upper octave, we used $y_{0H} = x_H$ (correct for $x = 1$), and from (5) and (10) we obtained

$$y_{1H} = \tfrac{1}{2}\left(x_H + \frac{x_H}{x_H}\right) = 0{\cdot}5a_H + 0{\cdot}5 \tag{11}$$

Similarly, we used $y_{0L} = 2x_L$ (correct for $x = 0{\cdot}25$) for the lower octave and obtained from (5) and (9)

$$y_{1L} = \tfrac{1}{2}\left(2x_L + \frac{x_L}{2x_L}\right) = 0{\cdot}5a_L + 0{\cdot}25 \tag{12}$$

Equations (11) and (12) can be interpreted geometrically as equations of tangents touching the square root curve in the end points of the interval (3). Thus we get an approximation by a broken line (Fig. 1) with a maximum deviation in the breaking point between both octaves

$$\Delta_{1\max} = 0{\cdot}75 - \sqrt{(0{\cdot}5)} = 0{\cdot}0428$$

i.e. approx. 6·1% of the correct value.

The construction of the estimate is extremely simple, as the additive constant only depends on the value of the *Carry* bit after the calculation of result exponent; a simple logic function has been adopted for the realisation (see the listing at the end, lines 27 through 35).

An advantageous side effect of this estimate is that it always holds

$$y_{1L} \leqq a_L \quad \text{and} \quad y_{1H} > a_H \tag{13}$$

so that the information on the real exponent (or "octave affiliation") of the input operand need not be stored for the calculation of the iteration formula (see later).

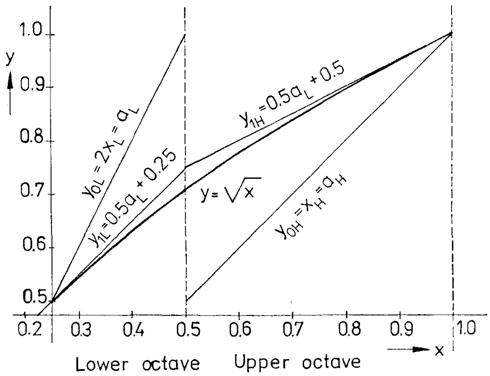

Fig. 1. Square root and its approximations.

For this approximation, we can estimate the maximum error after first iteration cycle using (7) as

$$\Delta_{2\max} \doteq 0{\cdot}0014 \approx 0{\cdot}2\%$$

and after the second cycle

$$\Delta_{3\max} \doteq 0{\cdot}0000013 \approx 0{\cdot}0002\%$$

which exceeds already the precision of our floating-point format; a constant number of two iteration cycles is fully acceptable and helps to reduce the execution time of each cycle by omitting the test for the accuracy of the result. From this point of view, the choice of a better first estimate can be considered useless, as even the best linear approximation – by an intersecting line with symmetrical deviations – might reduce the max. error Δ_1 to appr. 1·5% only, which would yield an accuracy of 0·012% after the first iteration cycle, and consequently would not enable any further reduction of the number of iterations; the construction of any nonlinear approximation would evidently consume more time than one iteration cycle saved.

By this important modification we entered the second group of methods, i.e. reducing the execution time of a single iteration cycle, with our next attention concentrated on the division in (5) as the most time-consuming operation. However queer it may sound, the most important decision was *not* to handle the iterations as a loop and *not* to use standard division subroutines, and to reduce instead the precision of computing according to the expected accuracy in the respective iteration cycle. In the first cycle, 10 bits are sufficient for the accuracy calculated above, and – on conditions given later – even 8 bits (with the precision of 0·4%) will maintain the accuracy of 0·0016% in the next iteration, which still exceeds almost twice the accuracy needed for the final result. A special division routine was therefore adopted for the first division: the MS bytes only enter the operation, the LS byte of dividend being replaced by its MSB followed by the mean of all possible values of the remainder (i.e. 07FH); in the program, this is realized by shifting in trailing ones into the remainder instead of zeros except of the first cycle. Full 8 bits are calculated and shifted right before addition of the first estimate.

For both divisions, due to (13) the end-of-loop is tested for the normalized format of result only, i.e. one left shift of dividend is added if the starting value of divisor is greater; as explained before, this can – and always will – occur with the upper octave operands only. This simplification requires an added precaution for the first iteration: as the difference may not appear in the MS bytes, a test for zero result of the first subtraction is unavoidable, causing a skip of the whole first iteration if true. In this case, the argument lies very close to 0·25 or 1 (within 2^{-6}), and the corresponding error of the first estimate is less than 0·05%, so that one iteration is fully sufficient.

For the second iteration, the kernel of the standard division subroutine only was adopted, thus enabling to omit all the unnecessary parts (such as testing of signs and zero values of operands, exponent operations etc.); two calls to the internal division loop FTDSR of the FTAR.LIB package (appended to the program listing for reference) reduce the extra memory requirement to an acceptable extent. This enabled us a different testing of operands to be incorporated; with the second division, the equivalence of operands means that the estimate is correct, and even the second iteration is skipped.

The final result of the second iteration is rounded using the shifted-out bit of the final division by two. This is important not only to maintain the accuracy (the limits of ± 1 LSB would be met even with mere truncation), but to ensure the stability of a test loop invoking square root and square in turn: due to the not unique assignment of values mentioned above, the loop will reach a stable pair of values not later than in the second repetition, while with the final truncation it would produce a sequence of continuously decreasing values. An analysis of this type of numerical instability exceeds the scope of this paper.

The flow diagram of the subroutine described here is given in Fig. 2 and the complete listing of the program in Fig. 3.

4. COMPARISON WITH OTHER METHODS

The effect of matching the algorithm, its coding and the instruction set of the given microprocessor can be demonstrated by comparison of the worst-case execution times and memory requirements of four types of programs:

a) A user program created by mechanical coding of the Newton-Raphson formula, using standard floating-point arithmetic subroutines, the primitive initial estimate $y_0 = x$ and the condition $y_{i+1} = y_i$ for end of iteration, would need 42 bytes of memory and execute in 1·8 ms per iteration[2], i.e. in 6 to 650 ms approx. depending on the size of input numbers.

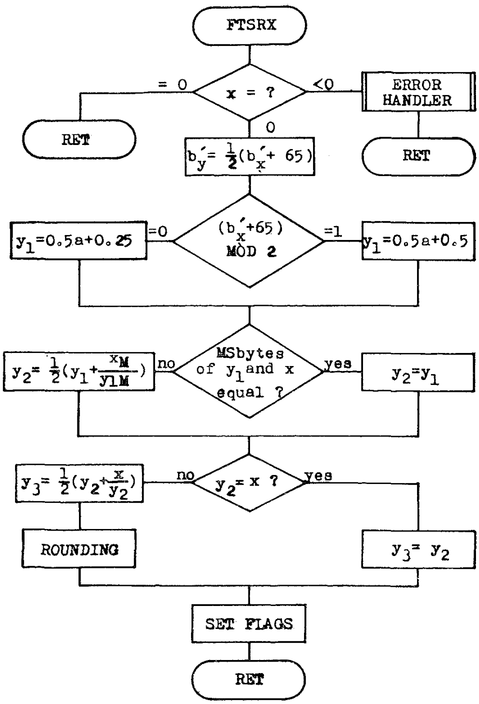

Fig. 2. Flow diagram for square root.

[2] 2 MHz clock frequency assumed in all compared cases.

```
  LOC  OBJ         LINE        SOURCE STATEMENT

                         ;=============================================================
                         ; THIS SUBROUTINE ACCEPTS THREE-BYTE F-P INPUT OPERANDS
                         ; IN D-E-B (ENTRY FTSRX) OR H-L-A (ENTRY FTSRY) REGISTERS,
                         ; RETURNS SQUARE ROOT IN D-E-B REGISTERS; MAX.ERROR < 1 LSB,
                         ; MAXIMUM (WORST CASE) EXECUTING TIME 1479 CLOCK PERIODS.

                         ; VALENTIN CHAMRAD, MICROELECTRONIC SYSTEMS DEPT.,
                         ; INSTITUTE OF INFORMATION AND AUTOMATION THEORY,
                         ; CZECHOSLOVAK ACADEMY OF SCIENCE, PRAGUE, CZECHOSLOVAKIA
                         ;=============================================================

                                 NAME    FTSRT
                                 CSEG
                                 PUBLIC  FTSRX,FTSRY
                                 EXTRN   FTECH

0000 47                  FTSRY:  MOV     B,A     ; ENTRY FOR INPUT OP IN H-L-A REGS
0001 EB                          XCHG
0002 AF                  FTSRX:  XRA     A       ; ENTRY FOR INPUT OP IN D-E-B REGS
0003 B2                          ORA     D       ; SET FLAGS ACCORDING TO INPUT OP
0004 FA7200     C                JM      FTSNH   ; BRANCH FOR NEGATIVE OPERANDS
0007 C8                          RZ              ; EXIT FOR ZERO OPERAND
0008 78                          MOV     A,B
0009 C641                        ADI     41H     ; ADD BIAS TO EXPONENT
000B 1F                          RAR             ; DIVIDE BY 2, LSB TO CARRY
000C F5                          PUSH    PSW     ; STORE RESULT EXPONENT ON STACK
000D 9F                          SBB     A       ; REPEAT CARRY IN ALL ACCU BITS
000E EE40                        XRI     40H     ; INVERT 6-TH BIT
0010 E6C0                        ANI     0C0H    ; CLEAR ALL LOWER BITS
0012 82                          ADD     D       ; ADD HI BYTE OF INPUT OPERAND
0013 1F                          RAR             ; DIVIDE BY 2 AND
0014 47                          MOV     B,A     ; MOVE FIRST ESTIMATE TO B-C REGS
0015 7B                          MOV     A,E
0016 1F                          RAR
0017 4F                          MOV     C,A
0018 7A                          MOV     A,D     ; SUBTRACT HI BYTES OF
0019 90                          SUB     B       ; INPUT OPERAND AND ESTIMATE
001A FA2300     C                JM      *+9     ; SKIP 2 LINES IF RESULT NEGATIVE
001D CA3E00     C                JZ      FTSR2   ; SKIP FIRST ITERATION IF ZERO
0020 C32800     C                JMP     *+8     ; GO TO FIRST ITERATION IF POSITIVE
0023 7B                          MOV     A,E     ; SHIFT INPUT OPERAND LEFT
0024 17                          RAL
0025 7A                          MOV     A,D
0026 17                          RAL
0027 90                          SUB     B       ; AND REPEAT SUBTRACTION OF HI BYTES
0028 2102F8                      LXI     H,0F802H; INITIALIZE RESULT REGISTERS
002B 17                  FTSR1:  RAL             ; FIRST DIVISION LOOP:SHIFT REMAINDER
002C B8                          CMP     B       ; TRY IF SUBTRACTION POSSIBLE
002D FA3200     C                JM      *+5     ; SKIP NEXT 2 INSTRUCTIONS IF NOT
0030 23                          INX     H       ; SET RESULT BIT
0031 90                          SUB     B       ; SUBTRACT ESTIMATE
0032 29                          DAD     H       ; SHIFT RESULT, TEST BIT TO CARRY
0033 DA2B00     C                JC      FTSR1   ; TEST FOR END OF LOOP
0036 7D                          MOV     A,L     ; RESULT TO A
0037 0F                          RRC             ; SHIFT BACK
0038 80                          ADD     B       ; ADD FIRST ESTIMATE
0039 1F                          RAR             ; SHIFT TO DIVIDE BY 2 AND
003A 47                          MOV     B,A     ; REPLACE FIRST ESTIMATE BY THE
003B 79                          MOV     A,C     ; RESULT OF FIRST ITERATION
003C 1F                          RAR
```

Fig. 3*a*. Program listing for square root.

```
 LOC  OBJ         LINE        SOURCE STATEMENT

 003D 4F            61             MOV     C,A
 003E EB            62 FTSR2:      XCHG             ; MOVE INPUT OP TO H-L
 003F AF            63             XRA     A
 0040 91            64             SUB     C
 0041 5F            65             MOV     E,A      ; PREPARE [DE] := -[BC]
 0042 9F            66             SBB     A
 0043 90            67             SUB     B
 0044 57            68             MOV     D,A
 0045 19            69             DAD     D        ; SUBTRACT 2-ND ESTIMATE FROM IN.OP
 0046 D25300   C    70             JNC     FTSR3    ; BRANCH IF RESULT NEGATIVE
 0049 7C            71             MOV     A,H
 004A B5            72             ORA     L
 004B C25600   C    73             JNZ     FTSR4    ; GO TO SECOND DIVISION IF POSITIVE
 004E 09            74             DAD     B        ; RESTORE ESTIMATE IF CORRECT,
 004F EB            75             XCHG             ; MOVE TO D-E
 0050 C36C00   C    76             JMP     FTSR5    ; AND GO TO FINAL FLAG SETTING
 0053 09            77 FTSR3:      DAD     B        ; RESTORE INPUT OPERAND
 0054 29            78             DAD     H        ; SHIFT LEFT
 0055 19            79             DAD     D        ; REPEAT SUBTRACTION OF ESTIMATE
 0056 3E05          80 FTSR4:      MVI     A,05H    ; INITIALIZE RESULT REGISTER
 0058 CD7E00   C    81             CALL    FTDSR    ; FIRST PART OF SECOND DIVISION
 005B F5            82             PUSH    PSW      ; PUSH FIRST BYTE OF RESULT ON STACK
 005C 3E01          83             MVI     A,01H    ; INITIALIZE RESULT REGISTER
 005E CD7E00   C    84             CALL    FTDSR    ; SECOND PART OF SECOND DIVISION
 0061 E1            85             POP     H        ; POP FIRST BYTE OF RESULT
 0062 6F            86             MOV     L,A      ; APPEND SECOND BYTE FROM ACCUMULATOR
 0063 09            87             DAD     B        ; ADD SECOND ESTIMATE
 0064 7C            88             MOV     A,H
 0065 1F            89             RAR              ; SHIFT RIGHT TO DIVIDE BY 2 AND
 0066 57            90             MOV     D,A      ; MOVE THIRD ESTIMATE TO D-E
 0067 7D            91             MOV     A,L
 0068 1F            92             RAR
 0069 CE00          93             ACI     0        ; ADD SHIFTED OUT BIT TO ROUND
 006B 5F            94             MOV     E,A
 006C 3E00          95 FTSR5:      MVI     A,0
 006E 8A            96             ADC     D        ; ADD CARRY FROM LO BYTE AND SET
 006F 57            97             MOV     D,A      ; FLAGS ACCORDING TO FINAL RESULT
 0070 C1            98             POP     B        ; POP RESULT EXPONENT FROM STACK
 0071 C9            99             RET
                   100
 0072 3E42         101 FTSNH:      MVI     A,42H    ; ERROR CODE FOR NEG.INPUT OPERANDS
 0074 CD0000   E   102             CALL    FTECH    ; INVOKE USER DEFINED ERROR HANDLER
 0077 AF           103 FTSUR:      XRA     A        ; SET DEFAULT RESULT ZERO AND
 0078 47           104             MOV     B,A      ; CORRESPONDING FLAGS
 0079 57           105             MOV     D,A
 007A 5F           106             MOV     E,A
 007B C9           107             RET
                   108
                   109 ;INTERNAL DIVISION LOOP OF FTDIV SUBROUTINE INVOKED ABOVE :
                   110
 007C 17           111 FTD1:       RAL              ; SHIFT IN RESULT BIT FROM CARRY
 007D D8           112             RC               ; TEST FOR END OF DIVISION LOOP
 007E 29           113 FTDSR:      DAD     H        ; SHIFT DIVIDEND (REMAINDER) LEFT
 007F 19           114             DAD     D        ; SUBTRACT DIVISOR
 0080 DA7C00   C   115             JC      FTD1     ; BRANCH IF RESULT POSITIVE OR ZERO
 0083 09           116             DAD     B        ; RESTORE DIVIDEND IF RESULT NEGATIVE
 0084 87           117             ADD     A        ; SHIFT RESULT LEFT ADDING ZERO LSB
 0085 D27E00   C   118             JNC     FTDSR    ; TEST FOR END OF DIVISION LOOP
 0088 C9           119             RET
```

Fig. 3*b*. Program listing for square root (cont'd) .

b) The same program, but with a better initial estimate (halving the exponent) and testing the difference (8) for end of loop would need appr. 60 bytes and execute in appr. 5 ms.
c) A BASIC-oriented subroutine published in [2], using separate treatment of exponent and mantissa, with the same initial estimate and fixed number of iterations as described above, but standard arithmetic subroutines for mantissa operations, needs 68 bytes and executes in 4099 clock periods, i.e. slightly more than 2 ms.
d) The subroutine described here needs 117 bytes (FTDSR not included) and executes in 1479 clock periods, i.e. 0·74 ms, which is 8·5% only more than needed for the speed-oriented multiplication subroutine described in [6], and even 14% less than for a standard multiplication (such as that described in [2] with a maximum of 0·861 ms).

5. CONCLUSION

This subroutine seems to be attractive for use in self-tuning controllers with square-root filters, because its execution time lies near the geometric average and thus fills the gap between that of the standard software solution and of a peripheral hardware unit (such as iSBC 310 with 0·205 ms), the price of which is much higher than that of the necessary extension of memory and seems to be unaffordable for usual controller applications.

(Received September 27, 1982.)

REFERENCES

[1] S. N. Cope: Floating-point arithmetic routines and macros for an Intel 8080 microprocessor. Oxford University Engineering Laboratory Report No. 1123/75.

[2] D. W. Clarke, S. N. Cope and P. J. Gawthrop: Feasibility study of the application of microprocessors to self-tuning controllers. O.U.E.L. Report No. 1137/75.

[3] KIMath Subroutines Programming Manual (KIM Mathematics Subroutines). MCDS Microcomputer Datensysteme GmbH, Darmstadt 1977.

[4] C. B. Falconer: Falconer floating point arithmetic. Dr. Dobb's Journal of Computer Calisthenics & Orthodontia, Vol. 4, No. 33, 4–14 and No. 34, 16–25.

[5] D. E. Knuth: The Art of Computer Programming II – Seminumerical Algorithms. Addison-Wesley, Reading, Mass. 1971.

[6] V. Chamrád: A speed-oriented floating-point subroutine package for the Intel 8080 microprocessors. In: Preprints SOCOCO '79, The 2nd IFAC/IFIP Symposium on Software for Computer Control, Vol. I., Prague 1979.

Ing. Valentin Chamrád, CSc., Ústav teorie informace a automatizace ČSAV (Institute of Information Theory and Automation — Czechoslovak Academy of Sciences), Pod vodárenskou věží 4, 182 08 Praha 8. Czechoslovakia.