

TP-Book – Addenda

Arnold Schönhage

December 2008

Preface. This text shall provide additional information for readers of our TP-book *Fast Algorithms – A Multitape Turing Machine Implementation* (Mannheim 1994) concerning a few extensions of *TPAL*, many additional routines, and other news related to this project. Following the presentation style of Chapters 6 and 8 on the basic modules INTARI and RECARI, we will below specify 16 new INTARI routines and 14 new RECARI routines, plus several enhancements for ‘old’ routines.

The material of Chapter 9 on ‘Complex and Real Polynomials’ has meanwhile been increased to such an extent that its module CREPARI had to be split into two separate modules CREPAR plus CREPEX, which we shall briefly outline below. Moreover there do now exist the main routines *CP1FSP*, *CPLFSP* for approximate factorization of complex polynomials by our splitting circle method that are used by routines *CP1RTS*, *CPRTS* for approximate computation of polynomial roots. These are combined with about 30 other underlying routines forming a family ‘*PORTS*’ of modules PORTA, PORTB, ..., so the prospective Chapter 10 (see page 284 of the TP-book) could now be written—but that is far beyond the scope of this text.

Instead, we are now maintaining so-called *docfiles*, like ‘portsdoc’, ‘crepsdoc’ specifying the interface conventions for all (public) routines of these new modules, and there are similar docfiles for the old modules of Chapters 6–8 (see Appendix).

1 Extensions of TPAL

1.1 Local jumps to next label (Skips)

Typically, *TPAL* code is containing many conditional jumps for skipping just a few instructions. Examples like $m := \max(a, b)$ for words a, b coded by the sequence

```
LAM a, LB b, JNLS.NEXT, LM.B, NEXT: MT m,
```

with label `NEXT:` not referred to explicitly from anywhere else have been the motivation for this extension of *TPAL* by so-called “skips” (in use since 1995), admitting to write this example in the simpler form

```
LAM a, LB b, JNLS/ LM.B,: MT m,
```

The following additions to the TP-book give the formal rules for such skips.

Primary Syntax (page 42): In paragraph 2.1.14, insert after “or with EOL”
, except for “skips” ending with ‘/’

Jump Instructions (page 54): In the middle of the page, preceding the announcement “The following examples ...”, insert this paragraph:

Furthermore, local jumps (without explicit destination, cf. 2.1.77) ending with ‘/’ instead of a delimiting comma or EOL (cf. 2.1.14) called “skips” are directed *forwardly* to the label coming next, possibly (and frequently) being an empty label inserted just for this purpose. Usually these are conditional jumps, but use of a `JS` may be of interest as well – compare `JS/:` used as a prefix to any subroutine with the constructs of the subsequent *Example 5*.

1.2 Table look-up

It has turned out that adding features for table look-up can greatly enhance the efficiency of Turing programs. In principle, as long as the size of such tables does not exceed the size of the Turing machine alphabet, there is no objection against such an additional construct. Here we give the details for a corresponding extension of our assembly language *TPAL* by listing the TP-book changes and supplements required as appropriate reference for this new feature. — While so far such tables had only been used in an adhoc manner with a few special applications, our new *TPS02* (see Appendix) now includes a full implementation of this part as well.

Tables. At the end of TP-book Section 2.1.1, the following text is inserted, with a change in item 2.1.18, where the second line describing the syntax of capnames *xx* should now end with “(used for labels and tables)” .

In addition to this naming of single constants, *TPAL* admits to specify one-dimensional arrays of word constants numbered by 0,1,2,... that are accessible by table look-up. Such tables can be introduced by means of *TAB-directives* telling the name of the table (always a capname) and listing its contents (or part of it) according to the syntax

2.1.20. *TAB-dir* := **TAB** [*P.*]*xx* = {*c*;\vb\;}(*c*\vb\)

with *vb* for string constants in *verbatim* mode to be explained below. Without the optional prefix ‘*P.*’, this directive declares table *xx* to be *local*, thus accessible only from within this module. When furnished with ‘*P.*’, however, *xx* is introduced as a *public name* with the effect that then this table *xx* becomes accessible also from within all other modules of the Turing program that do not contain their own table *xx*. Table names form a separate name space, so the same capname *xx* may be used both as a table name and for a label (cf. Section 2.1.7), but no capname should occur in more than one *TAB* directive of the module, and public table names should be unique as well (cf. end of this section). If all of the optional naming part [*P.*]*xx* is missing, such *TAB* directive is understood as prolongation of the last preceding *TAB* directive. If used in the very first *TAB* directive of the module, this form is an error. We shall restrict the following explanations to the simplified case without such prolongation directives assuming that all entries of a table are given in its initial *TAB* directive, although this may conflict with (implementation dependent) bounds on the length of lines.

Simple examples of short tables defined in this way by listing their entries as sequences of constants *c* according to the syntax rules in 2.1.17, separated by delimiting semicolons and ending with a comma or EOL, are

```
TAB MONTHL = 0; 31;28;31;30;31;30; 31;31;30;31;30;31, no leap year!  
with a redundant zero entry to have January numbered by 1,
```

```
W32- TAB P.DAYS ='S'u'n; 'M'o'n; 'T'u'e; 'W'e'd; 'T'h'u;'F'r'i;'S'a't
```

where the second table is limited to TP32 (see item 2.1.81), and each of the words is automatically filled up with a high end zero byte.

Specifying longer pieces of running text, however, would become rather cumbersome in this way. Therefore rule 2.1.20 admits the additional verbatim mode *\vb* initiated and ending with ‘\’ (code 92, or hex C5), where *vb* stands for any sequence of ASCII codes 32–126 (blanks and *visible* characters). In this low level mode, ‘\’ serves as an *escape* symbol in several ways. If \ is followed by a hex digit *h* (cf. 2.1.17), then also the next byte must contain a hex digit, where this (*\h*)*h* is taken as hex value of a single byte (read from low to high), thus any byte

value 00–FF can be used, in particular \C5 for \ itself, and \A0 for line-feed, but for these two also \\ and \n may be written. In all other cases, \ is read as the ending of the verbatim mode, then necessarily followed by a semicolon, a comma, or EOL (except for intermediate blanks or tabs, see 2.1.11). Note that an *initial* line-feed requires TAB $xx = \\n\dots$, while $= \backslash n\dots;$; $\\n\dots\backslash$ specifies strings beginning with ‘n’, or with ‘\’.

After such (\h)h codes (or \\, \n) have been resolved, the bytes of *vb* are packed into successive words of the table, perhaps ending with an incomplete final word. If there follows another *vb* element (possibly resulting from a prolongating directive TAB= \. .\), then packing continues with that incomplete word, else (if still incomplete) the final word is filled up with zero bytes. Accordingly the above example could also be written as

```
TAB P.DAYS = \Sun\00\; \Mon\00\; \Tue\00Wed\00Thu\; \\00Fri\00\;\Sat\,
```

now also valid for TP16, whereas the same text without these insertions of \00 would yield a shorter table with dense packing of those triples.

In preliminary stages of program development, it may sometimes be convenient to define a dummy table of length one. This is easily achievable by an empty string, as in TAB ADHOC = \\, ready for later extension.

Furthermore, tables (with public names) can also be added to a Turing program by linking the modules with additional “tb-files” $xx.tb$ (see Section 3.6.1), where the public name of the table is given as the capname xx in the file name (preceding the extension $.tb$). This form of definition is usually preferable for all kinds of large tables automatically generated in a preprocessing stage. The length of table xx denoted by $\#xx$ must not be greater than $\mathbf{2}$, which may sometimes be a problem for TP16, but is practically no restriction with TP32.

This is the end of the new text at the end of Section 2.1.1. Moreover, in the third line of rule 2.1.11 we replace “... ignored, except for the citation ...” with the new version “... ignored, except for the verbatim mode of 2.1.20 or citation ...”.

Table look-up. Item 2.1.21 specifying the syntax of ‘source’ is extended by two other possibilities, the two preceding lines are replaced by the following new text:

tape address t as explained below. In addition the source operand w can be specified by table look-up in the form $r + xx$ with one of the register codes r from 2.1.18, and by naming the table xx , predominant in its local meaning. The contents $\langle r \rangle$ of that register is used as index to read w as the $\langle r \rangle$ -th word of table xx , provided $\langle r \rangle < \#xx$, else w is some garbage word. Moreover $\#xx$ itself can be specified as source operand (yielding $w = 0$ if $\#xx = \mathbf{2}$). The tape addressing modes are also used as *destination* code for the storing instructions. The syntactical rules are:

2.1.21. source $s := .r \mid =c \mid =yy \mid =r + xx \mid \#xx \mid t$

2.1.22. tape address $t := i \mid i + \mid i - \mid -n \mid -yy \mid -K \mid yy$

Timing. By taking an average over several ways of implementation, and modes of table access, we have found a fair estimate of 5–6 time units per table look-up. Accordingly, Table 2.1 in Section 2.3.2 shall contain the following additional rule:

load instructions, Boolean instructions,
and additions, subtractions with source $= r + xx$ 6 (units)

This is meant for tables of moderate size, nowadays not bigger than a few megabytes. Otherwise much higher timings may occur — TP32 is no RAM, after all!

1.3 Masking the contents of register K

For programming the decomposition of bit length values ll into their *word count* $lw := \lceil ll/32 \rceil$ and their *bit part* $lb := ll \bmod 32$, the latter typically being wanted in register K , we have frequently used coding like LA 11, SMAL 5, SML 27, LK.M, apparently a bit awkward and too costly by that double word shift via register M . Therefore we have now introduced a new TP instruction MSK for masking the contents of register K that admits to replace the former construct by LAK 11, SAL5, MSK 31, with the following syntax for MSK to be added at the end of TP-book Section 2.1.4:

The following item specifies a special AND instruction for register K :

2.1.146. *mask instruction* := MSK n | MSK yy (timing = 1 tu)

with the effect to mask the contents of K with the 8-bit pattern of n , or of the value of lowname yy , respectively, which must then be less than 256.

Clearly this can then be used for other purposes as well, e.g. for branching by bit testing, like LK b, MSK 8, JK... , and similar constructs.

2 New Routines in INTARI

2.1 Supplements to TP-book Section 6.1.1

After item 6.1.14, insert this specification of a new special ci-code routine.

6.1.14c. *Copying a Gaussian integer.* CICOPY

Inputs: x in ci-code on T_1 , with components of i-length n_0 and n_1 ,
 p_1 on top word, taboo, p_2 stacky;

Returns: x in ci-code on T_2 , p_2 on top word, p_1 as before,
 <A> = $n_0 + n_1 + 1$, e.g. useful for later discarding by DP1-A,
 = n_1 . – registers E, K and T_0 are not used –

At the end of Section 6.1.1 add the following new item.

6.1.17. *Generating a pseudo random integer.* IRANDOM

Inputs: <C> = n , $0 \leq n < \mathfrak{Z}/2$, <K> = $k < 32$ (<<16>>) (else $k = 31$ (<<15>>) is used),
 <MA> = x as initial seed (cf. routine RANDOM), p_1 stacky;

Returns: z in ci-code on T_1 , p_1 on top word, with $z = 0$ iff $n = 0$,
 else $-\frac{1}{2}\mathfrak{Z}^n \leq z \cdot 2^k < -\frac{1}{4}\mathfrak{Z}^n$ or $\frac{1}{4}\mathfrak{Z}^n \leq z \cdot 2^k < \frac{1}{2}\mathfrak{Z}^n$,
 <MA> = value of x after n steps $x := 5 * x + 1 \bmod \mathfrak{Z}^2$, <K> as before.

2.2 New addition and subtraction routines

At the end of Section 6.1.2 append the following two items about new routines for ci-code operands. — These routines do also use register K and tape T_0 .

6.1.23c. *Sum and difference of Gaussian integers.* CISUM, CIDIF

Inputs: x in ci-code on T_1 , p_1 on top word, taboo,
 y in ci-code on T_2 , p_2 on top word, taboo, p_3 stacky;

Returns: $z = x \pm y$ in ci-code on T_3 , p_3 on top word of z , p_1, p_2 as before;
 – also register K and T_0 are used –

Timing: $tt(n | CISUM) \simeq 22 \cdot n + 99$ for ci-code operands with
 $tt(n | CIDIF) \simeq 22 \cdot n + 101$ components of i-length n .

6.1.25c. *Addition and subtraction of Gaussian integers.* *CIADD, CISUB*

Inputs: x in ci-code on T_1 , p_1 on top word, stacky above,
 y in ci-code on T_2 , p_2 on top word, taboo;

Returns: $z = x \pm y$ in ci-code replaces x on T_1 , p_1, p_2 on top words of z, y ;
 – also register K and T_0 are used –

Timing: $tt(n|CIADD) \simeq 36 \cdot n + 104$ for ci-code operands with
 $tt(n|CISUB) \simeq 36 \cdot n + 106$ components of i-length n .

2.3 News related to multiplication routines

Insert at the beginning of Section 6.1.3 (after item 6.1.29) the following interfaces for two new routines, *IDBL* for doubling an integer, and *IMLW* (similar to *WML*) for multiplying an i-code with a word operand.

6.1.29d. *Doubling an integer.* *IDBL*

Inputs: x in i-code on T_1 , p_1 on top word, stacky above;

Returns: $z = 2 \cdot x$ in i-code replaces x on T_1 , p_1 on top word;
 – registers B, K and T_0 are not used –

Timing: $tt(n|IDBL) \simeq 9 \cdot n + 13$ for results z of i-length n .

6.1.29i. *i-code multiplication by a word operand.* *IMLW*

Inputs: x in i-code on T_1 , p_1 on top word, taboo,
 $\langle \mathbf{B} \rangle = w > 0$, p_2 stacky;

Returns: $z = x \cdot w$ in i-code on T_2 , p_2 on top word, p_1 as before,
 $\langle \mathbf{B} \rangle = b$ unchanged; — register K and T_0 are not used.

Timing: $tt(n|IMLW) \simeq 40 \cdot n + 26 \langle \langle 24 \cdot n + 26 \rangle \rangle$ for inputs of i-length n .

A new version of routine *CIML* now asymptotically performed by means of *two* multiplications mod $(\mathfrak{Z}^{2^k} + 1)$ and similar innovations related to routine *LCPML* have led to the following further updates of Section 6.1.3.— These new routines are mainly needed as auxiliaries for certain enhancements of modules *CREPAR* and *CREPEX*. Since these modules are no longer TP16 maintained, some of the new routines are now merely supporting *single* word sml-length inputs, although the TP16 version of routine *SML* (cf. 6.1.37) does accept longer operands.

6.1.37n. *Suitable n for SML and SSQU application.* *SMLNS, SMLNS2*

Inputs: $\langle \mathbf{A} \rangle = n_o \leq 2^{31} - 8192 \langle \langle \mathbf{MA} \rangle = n_o \leq 2^{31} - 8192 \rangle \rangle$;

Returns: $\langle \mathbf{A} \rangle = n < \mathfrak{Z}/2 \langle \langle \text{or also } \mathfrak{Z}/2 \leq n = \langle \mathbf{MA} \rangle - \mathfrak{Z}^2/2 \text{ with } \langle \mathbf{HA} \rangle = 1 \rangle \rangle$
 for suitable $n \geq n_o$ so that condition (1.14) on page 32 is satisfied;
 for SMLNS: with $n = n_o$ for $n_o < 16$, else n divisible by 2^κ ,
 where $\kappa = \lfloor (\log n_o - 4)/2 \rfloor$, thus n is even for $n_o \geq 64$;
 for SMLNS2: always with even n , by $\kappa = \max(1, \lfloor (\log n_o - 4)/2 \rfloor)$.

6.1.37i. *Encoding ci-code by two sml-codes mod $(\mathfrak{Z}^{2k} + 1)$.* *CI2SML, CI1SML*

Inputs: $\langle p_0 - 2 \rangle = 2k + 1$, $\langle p_0 - 1 \rangle = k > 1$ on T_0 , with suitable $2k$ (see 6.1.37n),
 $x = x_0 + i \cdot x_1$ with integers $|x_0|, |x_1| \leq \frac{1}{2} \cdot K^2 - 2K$, where $K = \mathfrak{Z}^k$,
for CI2SML: with x in ci-code on T_1 , p_1 on top word, stacky above, p_2, p_3 stacky;
for CI1SML: x_0 in i-code on T_3 , p_3 on top word, stacky above,
 x_1 in i-code on T_1 , p_1 on top word, stacky above, p_2 stacky;
Returns: $q_x \equiv K \cdot x_1 - x_0 \pmod{(K^2 + 1)}$ in $2k + 1$ words above x_1 on T_1 , p_1 above,
 $p_x \equiv K \cdot x_1 + x_0 \pmod{(K^2 + 1)}$ in $2k + 1$ words on T_2 , p_2 above,
for CI2SML: p_3 as before, for CI1SML: x_0 discarded from T_3 .

The following routines are useful for the converse task of *decoding* from two residues $x \equiv a + K \cdot b$ and $y \equiv a - K \cdot b \pmod{(\mathfrak{Z}^{2k} + 1)}$, with $K = 2^k$ as imaginary unit $\pmod{(\mathfrak{Z}^{2k} + 1)}$. First one can call addition routine *SADSB* (cf. 6.1.38) to compute $x + y \equiv 2a$, or a new (simpler) version *SSUM* to put this sum upon a third tape, then one calls the new routine *SIMDC* to obtain $(x - y)/K \equiv 2b$. — Moreover, there are two new routines for *encoding* added as items 6.1.39i and 6.1.39j.

6.1.38i. *Imaginary decoding of two sml-codes mod $(\mathfrak{Z}^{2k} + 1)$.* *SIMDC*

Inputs: x in sml-code $\pmod{(\mathfrak{Z}^{2k} + 1)}$ on T_1 , p_1 on top word x_{2k} , stacky above,
 y in sml-code $\pmod{(\mathfrak{Z}^{2k} + 1)}$ on T_2 , p_2 on top word y_{2k} , stacky above,
 $\langle C \rangle = k$, $2 \leq k < \mathfrak{Z}/4$, with $K = 2^k$ as imaginary unit,
 p_3 stacky;
Returns: z in sml-code $\pmod{(K^2 + 1)}$ on T_3 , p_3 on top word z_{2k} ,
where $z \equiv (x - y)/K \pmod{(K^2 + 1)}$, $z < \mathfrak{Z}^{2k} + 3 \cdot \mathfrak{Z}^{2k-1}$,
 p_1, p_2 on bottom words of x, y , and x, y discarded. — T_0 is not used —
Timing: $tt(k | SIMDC) \simeq 26 \cdot k + 54$.

6.1.38s. *Simple sum of two sml-codes mod $(\mathfrak{Z}^m + 1)$.* *SSUM*

Inputs: x in sml-code $\pmod{(\mathfrak{Z}^m + 1)}$ on T_1 , p_1 on top word x_m , taboo,
 y in sml-code $\pmod{(\mathfrak{Z}^m + 1)}$ on T_2 , p_2 on top word y_m , taboo,
 $\langle C \rangle = m$, p_3 stacky;
Returns: $z \equiv x + y \pmod{(\mathfrak{Z}^m + 1)}$ in sml-code on T_3 , p_3 on top word z_m ,
also $\langle K \rangle = z_m$, p_1, p_2 as before. — register B and T_0 are not used —
Timing: $tt(m | SSUM) \simeq 11 \cdot m + 24$.

6.1.39i. *Mapping i-code to sml-code.* *ISRED*

Inputs: x in i-code on T_2 , p_2 on top word n , taboo, p_1 stacky,
 $\langle A \rangle = t < \mathfrak{Z}/2$, $\langle B \rangle = m$, with $2 \leq m < \mathfrak{Z}/2$;
Returns: $z \equiv x \cdot \mathfrak{Z}^t \pmod{(\mathfrak{Z}^m + 1)}$ in sml-code put on T_1 , p_1 on top word z_m ,
always with $z < \mathfrak{Z}^m + 4 \cdot \mathfrak{Z}^{m-1}$, p_2 as before (T_0 is used).
Timing: $tt(m | ISRED) \simeq 16.5 \cdot m + 125$ for typical cases with $n \leq m$, $t < 2m$.

6.1.39j. *Multiply sml-code with imaginary unit.* *SMLI2*

Inputs: x in sml-code $\pmod{(\mathfrak{Z}^{2k} + 1)}$ on T_1 , taboo, p_1 on top word x_{2k} ,
 $\langle B \rangle = 2k$, $2 \leq k < \mathfrak{Z}/4$, with $K = 2^k$ as imaginary unit,
 p_2, p_3 stacky;
Returns: $z \equiv x \cdot K \pmod{(\mathfrak{Z}^{2k} + 1)}$ in sml-code on T_2 and T_3 , p_2, p_3 on top word z_{2k} ,
also $\langle K \rangle = z_{2k}$, p_1 on the “middle word” x_k ,
 $\langle B \rangle = 2k$, $\langle M \rangle = k$. — T_0 is not used —
Timing: $tt(k | SMLI2) \simeq 22 \cdot k + 27$.

2.4 A new division routine

Insert in Section 6.1.5 (after item 6.1.51) the following interface for a new routine *IDIVW* similar to *WDIV*, now for the numerator in i-code.

6.1.51 i. *i-code division by a word operand.* *IDIVW*

Inputs: x in i-code on T_1 , p_1 on top word, taboo,
 $\langle \mathbf{B} \rangle = w > 0$, p_2 stacky;

Returns: q in i-code on T_2 , p_2 on top word,
 where $x = q \cdot w + r$ with $0 \leq r < w$ (or $q = x$, $r = 0$, if $w = 0$),
 $\langle \mathbf{A} \rangle = r$, $\langle \mathbf{B} \rangle = w$, $\langle \mathbf{C} \rangle = \text{i-length}(q)$,
 p_1 on bottom word of x , $\langle \mathbf{M} \rangle = n$ (for later **P1+M**, if needed).

Timing: $tt(n|IDIVW) \simeq 77 \cdot n + 100 \ll \langle \langle 45 \cdot n + 68 \rangle \rangle$ for inputs of i-length n ,
 $\simeq 77 \cdot n + 35 \ll \langle \langle 45 \cdot n + 35 \rangle \rangle$ for $|x| < w \cdot \mathfrak{Z}^{n-1}$.

3 New Routines in RECARI

3.1 Special multiplication and division routines

Sections 8.1.3 and 8.1.4 get new routines generalizing *RML2K*, *CML2K*, see 8.1.35, and for multiplying or dividing r-code or c-code operands by a positive word operand.

8.1.35a. *Multiplication by arbitrary powers of two.* *RML2L*, *CML2L*

Inputs: x in r-code [or c-code] on T_1 , p_1 on top word, stacky above,
 $\langle \mathbf{A} \rangle - \langle \mathbf{HA} \rangle \cdot \mathfrak{Z} = l$, thus $-\mathfrak{Z}/2 \leq l < \mathfrak{Z}/2$;

Returns: $z = x \cdot 2^l$ in r-code [or c-code] replaces x on T_1 , p_1 on top word;
 – scaling overflow causes Alarm, also with size underflow –

Timing: $tt(n|RML2L) \simeq 11 \cdot n + 51$, $tt(n|CML2L) \simeq 23 \cdot n + 104$.

8.1.36. *Multiplication by word $w > 0$.* *RMLW*, *CMLW*

Inputs: x in r-code [or c-code] on T_1 , p_1 on top word, stacky above,
 $\langle \mathbf{B} \rangle = w$, $0 < w < \mathfrak{Z}/2$;

Returns: $z = x \cdot w$ in r-code [or c-code] replaces x on T_1 , p_1 on top word,
 $\langle \mathbf{B} \rangle = w$ unchanged.

Timing: $tt(n|RMLW) \simeq 41 \cdot n + 43$, $tt(n|CMLW) \simeq 91 \cdot n + 77 \ll \langle \langle 59 \cdot n + 77 \rangle \rangle$.

8.1.40. *Division by word $w > 0$.* *RDIVW*, *CDIVW*

Inputs: x in r-code [or c-code] on T_1 , taboo, p_1 on top word,
 $\langle \mathbf{A} \rangle = s < \mathfrak{Z}/2$, $\langle \mathbf{B} \rangle = w > 0$, p_3 stacky, T_2 taboo;

Returns: Some z in r-code [or c-code] on T_3 , p_3 on top word,
 with $|z - x/w| < 1/\mathfrak{Z}^s$, p_1 as before,
 for $w \geq 2$: $\langle \mathbf{A} \rangle = s$, then equal to scaling of z ,
 for $w = 1$ faster by *RNDW* [or *CRNDW*], for $w = 0$ no action on tapes;

Timing: $tt(n|RDIVW) \simeq 76 \cdot n + 70 \ll \langle \langle 44 \cdot n + 70 \rangle \rangle$ for $|x| < \mathfrak{Z}/2$ and
 $tt(n|CDIVW) \simeq 154 \cdot n + 120 \ll \langle \langle 90 \cdot n + 120 \rangle \rangle$ $s = n = \text{i-length of } x$.

3.2 New routines for square roots and cube roots

According to the following additional routines, TP-book Section 8.1.5 should now have the new heading “Square Root and Cube Root Routines”.

8.1.51w. *One word precision square root.* SQRTW

Inputs: $\langle \mathbf{A} \rangle = w$ specifying $x = w/\mathbf{X}$;

Returns: $\langle \mathbf{A} \rangle = v$ for $y = v/\mathbf{X}$; if $\sqrt{w} \in \mathbb{Z}$ then exact result $y = \sqrt{x}$
 else $-1/\mathbf{X} < y - \sqrt{x} < (1/2\mathbf{X})/\sqrt{x}$ ($2/\mathbf{X} \leq x$);

Timing: $tt(\text{SQRTW}) \simeq 258 \langle \langle 138 \rangle \rangle$. – register K and T_0 are not used –

8.1.52w. *One word integer square root, floored.* ISQRTW

Inputs: $\langle \mathbf{A} \rangle = x$;

Returns: $\langle \mathbf{B} \rangle = y = \lfloor \sqrt{x} \rfloor$, $\langle \mathbf{A} \rangle = x - y^2$;

Timing: $tt(\text{ISQRTW}) \simeq 178 \langle \langle 90 \rangle \rangle$. – register K and T_0 are not used –

8.1.53. Due to an improved complex square root routine (cf. *T. Ahrendt*, Proc. ISSAC '96, Zürich 1996, 142–149), here the last line on timing is now

$tt(n|\text{CSQRT}) \simeq 9\frac{1}{3} \cdot n \cdot L(n)$ asymptotically, cf. 6.1.37.

8.1.55. *Real cube root.* CBRT (CBRTP)

Inputs: x in r-code on T_1 , p_1 on top word, stacky above, p_2, p_3, p_4 stacky;

$\langle \mathbf{M} \rangle = e > 0$, $\langle \mathbf{A} \rangle = s < \mathbf{X}/2$ for $\varepsilon = e/\mathbf{X}^s$ (or $\varepsilon = 2^{-p}$ by $\langle \mathbf{A} \rangle = p$);

Returns: Some y in r-code on T_2 , p_2 on top word, so that $|x - y^3| < \varepsilon$,
 p_1, p_3, p_4 as before.

Timing: $tt(n|\text{CBRT}) \simeq 33 \cdot n^2 + 420 \cdot n + 1550 \cdot \log_3 n \langle \langle 21 \cdot n^2 + \dots \rangle \rangle$
 for moderate $s = n$, and $|x| < \mathbf{X}/2$,
 $\simeq 9\frac{2}{3} \cdot n \cdot L(n)$ asymptotically, cf. 6.1.37.

8.1.55c. *Complex cube root.* CCBRT (CCBRTP)

Inputs: $x = a + i \cdot b$ in c-code on T_1 , p_1 on top word, stacky above,

$\langle \mathbf{M} \rangle = e > 0$, $\langle \mathbf{A} \rangle = s < \mathbf{X}/2$ for $\varepsilon = e/\mathbf{X}^s$ (or $\varepsilon = 2^{-p}$ by $\langle \mathbf{A} \rangle = p$),

$\langle \mathbf{C} \rangle = h \leq 2$ as root selection index (any $h > 2$ is read as $h = 2$),

p_2, p_3, p_4 stacky;

Returns: Some z in c-code on T_2 , p_2 on top word, so that $|x - z^3| < \varepsilon$,
 and either $z = 0$, or with $\arg(z)$ close to $\arg(x)/3 + h \cdot 2\pi/3$,
 p_1, p_3, p_4 as before; (where $\arg(x) \in (-\pi, \pi]$)

Timing: $tt(n|\text{CCBRT}) \simeq 132 \cdot n^2 + 2000 \cdot n + 6000 \cdot \log_3 n \langle \langle 85 \cdot n^2 + \dots \rangle \rangle$
 for moderate $s = n$, and $|x| < \mathbf{X}/2$,
 $\simeq 17\frac{1}{3} \cdot n \cdot L(n)$ asymptotically, cf. 6.1.37.

3.3 Further supplements

Here we mention some enhancements to TP-book Section 8.1.6 on size routines. First we specify useful bounds for the scaling s of the bounds $r = u/\mathfrak{Z}^s$ returned by any of the routines in 8.1.62, 8.1.63. If $r > 0$, then (depending on the input p)

$$s \leq \lceil p/\lambda \rceil \text{ for } RSIZE, \quad s \leq \lceil (p+1)/\lambda \rceil \text{ for } C1SIZE, \quad s \leq \lceil (p+2)/\lambda \rceil \text{ for } CSIZE.$$

8.1.65. *Lower bound product of epsilon sequence.*

EPSPROD

Inputs: p_1 stacky; on T_2 a bottom word $i_0 = \mathfrak{Z}-1$ or $\mathfrak{Z}-2$, followed by a (possibly empty) sequence of word triples $(w, t, i)_j$ encoding values ε_j according to the following rules, always with $w > 0$:

$$\begin{aligned} \text{if } i = 0 & \quad \text{then } \varepsilon = w/\mathfrak{Z}^{t'} & \quad \text{with } t' = t - \mathfrak{Z} \cdot \text{top}(t), \\ \text{if } i = \mathfrak{Z}/2 & \quad \text{then } \varepsilon = 1/(w \cdot \mathfrak{Z}^{t'}) & \quad \text{with } t' = t - \mathfrak{Z} \cdot \text{top}(t), \\ \text{for } 0 < i < \mathfrak{Z}/2: & \quad \varepsilon = (w/\mathfrak{Z} \cdot 2^{-t})^i, & \quad 0 \leq t < \mathfrak{Z}; \end{aligned}$$

p_2 on top word of the sequence, stacky above;

Returns: p_1 as before, sequence discarded from T_2 , p_2 on bottom word i_0 ;

lower bound $b \leq$ product of all ε_j in one of the following forms:

$$i_0 = \mathfrak{Z}-1: \quad \langle \mathbf{M} \rangle = e > 0, \quad \langle \mathbf{A} \rangle = s < \mathfrak{Z}/2 \text{ for } b = e/\mathfrak{Z}^s, \text{ usually } \langle \mathbf{C} \rangle = 0;$$

if b is too small ($s \geq \mathfrak{Z}/2$) then Alarm exit,

if $b \geq \mathfrak{Z}$ would be possible ($s=0$), then warning by

$\langle \mathbf{C} \rangle = \text{i-length}(t) > 0$, with same meaning as below;

$$i_0 = \mathfrak{Z}-2: \quad \langle \mathbf{B} \rangle = q \geq \mathfrak{Z}/2, \quad \langle \mathbf{MA} \rangle - \langle \mathbf{HA} \rangle \cdot \mathfrak{Z}^2 = t, \quad \langle \mathbf{C} \rangle = 0 \text{ for } b = q/\mathfrak{Z} \cdot 2^{-t},$$

if t is representable in this form, otherwise $\langle \mathbf{C} \rangle = n \geq 2$

for t in i-code of length n on T_1 , p_1 on its bottom word.

8.1.71. *Generating complex pseudo random numbers.*

CRANDOM

Inputs: $\langle \mathbf{B} \rangle = m > 0$, $\langle \mathbf{C} \rangle = s < \mathfrak{Z}/2 - 2$, usually $s > 0$ required,

$\langle \mathbf{MA} \rangle = x$ as random seed (cf. 6.1.16, routine *RANDOM*),

$\mathfrak{Z}r$ in i-code on T_2 , p_2 on top word, stacky above, p_1, p_3, p_4 stacky,

$\mathfrak{Z}r=0$ stands for $r=1$, then also $s=0$ admitted;

Returns: m pseudo random numbers z_1, \dots, z_m in c-code on T_1 , p_1 above,

all with scaling $s+1$, satisfying $|z_j| < r$,

$\mathfrak{Z}r$ in i-code on T_2 , with p_2, p_3, p_4 as before,

$\langle \mathbf{MA} \rangle = x'$ as new seed, promoted by routine *RANDOM*.

8.1.72. *Complex high word normalizing.*

CHWNM

Inputs: z in c-code $(x + i \cdot y)/\mathfrak{Z}^s$ on T_1 , taboo, p_1 on top word,

with $nx = \text{i-length}(x)$, $ny = \text{i-length}(y)$;

Returns: $\langle \mathbf{C} \rangle = n = \max(nx, ny)$; if $n = 0$ then also $\langle \mathbf{A} \rangle = \langle \mathbf{B} \rangle = 0$, else

$\langle \mathbf{K} \rangle = k$, maximal k with $\text{i-length}(x \cdot 2^k) \leq n$ and $\text{i-length}(y \cdot 2^k) \leq n$,

$\langle \mathbf{A} \rangle = a \equiv \lfloor x \cdot 2^k / \mathfrak{Z}^{n-1} \rfloor \pmod{\mathfrak{Z}}$, $\langle \mathbf{B} \rangle = b \equiv \lfloor y \cdot 2^k / \mathfrak{Z}^{n-1} \rfloor \pmod{\mathfrak{Z}}$,

p_1 always as before.

8.1.73. *Real or complex scaling extension.*

RSCLEXT, CSCLEXT

Inputs: x in r-code [or c-code] with scaling s on T_3 , p_3 on top word, stacky above,

$\langle \mathbf{A} \rangle = t < \mathfrak{Z}/2$ as prescribed scaling (typically with $s \leq t$);

Returns: If $t \leq s$ then no action on tapes,

else x in r-code [or c-code] with scaling t replaces x on T_3 , p_3 on top word,

always $\langle \mathbf{A} \rangle = t$ unchanged.

– register K not used –

4 New Routines in CREPARI

4.1 New modules, new specifications

Since its first TP-book presentation, module CREPARI has meanwhile undergone a considerable increase in size, so it has been split into two separate modules CREPAR plus CREPEX, where the suffix of the latter stands for **extras**, for **extensions**. These two modules require linking with modules INTARI, RECARI and with each other.

Our docfile ‘crepsdoc’ (see Appendix) contains a full description of the interface conventions of all routines of CREPAR and CREPEX in their present version. In writing that text, we have felt that it might be clarifying to sharpen the notions of *rp-code*, *cp-code* etc. (cf. TP-book pp. 16–17) by explicit reference to the dimensions of the underlying affine spaces. Accordingly, we have to replace the last line

- of item 1.2.11 by “is an $rp(n)$ -code for $f(x) = \dots$ ”
- of item 1.2.12 by “is a $cp(n)$ -code for $g(x) = \dots$ ”
- of item 1.2.13 by “is an $ip(n)$ -code for $f(x) = \dots$ ”,

and in 1.2.14 Any $cp(n)$ -code for $g(x)$ is a $cp1(n)$ -code for $g^+(x) = g(x) + x^n$.

Analogous rules apply for $rp1(n)$ -code, $ip1(n)$ -code, etc., and in the very same manner the codes for decimal input of polynomials (cf. TP-book page 272) are now to be called “ $dcp(n)$ -code” and “ $drp(n)$ -code”.

4.2 New routines for multiplying polynomials

In addition to the standard routines *CPML*, *RPML* there are now also new routines *LCPML*, *LRPML* for *low* multiplication of polynomials f, g , which means to compute some h satisfying $|h - (f \cdot g \bmod x^l)| < \varepsilon$. These routines achieve the obvious savings for small degrees as well as substantial asymptotical savings by ideas from our paper *Bivariate polynomial multiplication patterns* (Proc. AAEECC-11, Paris, July 1995, LNCS 948, 70–81).

CREPAR now also offers special routines for *squaring* of complex and real polynomials, faster than *CPML*, *RPML* by about 30 to 40 percent, meanwhile also asymptotically fast for large degrees. Their elementary parts are also used for the low degree parts of our root squaring routines *CRTSQ*, *CRT1SQ*, *RRTSQ*, *RRT1SQ* furnishing an efficient implementation of Graeffe’s method for complex and real polynomials, while the asymptotical parts of these are by an economic application of *SML* already briefly indicated on TP-book page 284.

Furthermore we mention a bunch of new *scaling routines* now available for (approximate) transition from $f(x)$ to $g(x) = f(r \cdot x)$ both for arbitrary real $r > 0$ and for binary scaling with $r = 2^l$ as announced on TP-book page 277.

4.3 New routines in module CREPEX

The routines for integer polynomials and the DFT-routines (cf. TP-book Sections 9.1.6 and 9.2.1) are now contained in the new module CREPEX. Substantial new parts are routines for Horner’s rule (evaluation of a few Taylor coefficients of a complex or real polynomial at some point) and for complete Taylor shifts, both asymptotically fast, serving as important tools for our root finding algorithms.

Another such tool is a collection of efficient size routines (based on those root squaring routines *CRTSQ*, *CRT1SQ*) for computing approximations for the outer or inner root radius of a complex polynomial, for instance routine *CRTMAX*: For

any f given in $\text{cp}(n+1)$ -code with $n \geq 1$, top coefficient $a_n \neq 0$ and root radius $R = \max\{|z| : f(z) = 0\}$, and for given precision word $e > 0$ specifying $\varepsilon = e/\mathfrak{X}$, routine *CRTMAX* returns the number tz of *trailing* zero coefficients of f and (in case of $tz < n$) some u in i-code and a signed word s ($-\mathfrak{X}/2 \leq s < \mathfrak{X}/2$) such that $r = u/\mathfrak{X}^s$ satisfies

$$r/(1 + \varepsilon) < R < r \cdot (1 + \varepsilon).$$

5 PORTS – Modules for Polynomial Roots

These modules contain (or shall contain) the routines for approximate factorization of complex polynomials and for approximate computation of their roots, prospectively also more special versions for polynomials with real or integral coefficients. The first part *PORTA* briefly outlined below hosts the main routines for complex polynomials, while many of the pertaining auxiliary routines have been moved to module *PORTB*. These include, in particular, routines *CRTRK*, *CRT1RK* (given r , find k) for counting roots of modulus $\leq r$, and converse routines *CRTKS*, *CRT1KS* (given k , find r 's) for computing approximate root radii all based on the *CREPAR* routines *CRTSQ*, *CRT1SQ* for Graeffe's root squaring method. Relying on these tools, routine *GAPFD* serves to find wide splitting gaps.

Further parts of *PORTB* are fast conversion from power sums to elementary symmetric functions and a central routine *UCSP* for so-called unit circle splitting. A third module *PORTC* is planned to cover other more special topics.

All these modules are restricted to TP32. For more details on their present state, we must refer to docfile 'portsdoc' (see Appendix).

5.1 Classical root finding

A very basic initial part of module *PORTA* contains the classical procedures, first of all our implementation of Newton iteration for simple roots of complex polynomials, expecting some suitable starting point as input and checking its a priori sufficiency for convergence by a robust version of the Kantorovich criterion.

Moreover we have implemented the solution of quadratic and cubic equations by means of square roots and cube roots with great care to provide fast leaf routines enhancing the efficiency of our general splitting routines. In June 2001, a similar treatment of *quartic* equations has been completed as well.

5.2 Splitting routines and root finding

General linear factor splitting is by routine *CPLFSP* subject to these conventions: For f given in $\text{cp}(n+1)$ -code and precision bound $\varepsilon = e/\mathfrak{X}^s$, this routine returns words k , $m = n - k$, and $n+1$ complex numbers q , u_1, \dots, u_k , and v_1, \dots, v_m in c-code so that $g(x) = q \cdot \prod_{j=1}^k (x - u_j) \cdot \prod_{j=1}^m (1 - v_j \cdot x)$ satisfies $|f - g| < n \cdot \varepsilon$.

There is a similar routine *CP1FSP* for *monic* linear factor splitting (applied to f in $\text{cp1}(n)$ -code) returning u_1, \dots, u_n (with $q = 1$) for such a monic $g(x)$, plus several more technical subroutines for performing such splittings by applying those *PORTB* routines *GAPFD* and unit circle splitting *UCSP* recursively.

For our *root finding* routines we have combined these splitting routines with (rather sharp) a posteriori perturbation bounds in an adaptive manner. For higher degree polynomials, root precision of a few words will usually require a splitting precision of many words, depending on the clustering of the (yet unknown) roots, which one should keep in mind when using these routines. Again we must refer to docfile 'portsdoc', but let us here present one such interface for illustration.

Inputs: f in $\text{cp1}(n)$ -code on T_1 , p_1 above f , for $0 < n < 2^{22}$,
 $\langle \mathbf{B} \rangle = e_0$, $\langle \mathbf{C} \rangle = s_0 < \mathbf{X}/2$ specifying “pre-error” bound
 $|f_0 - f| \leq \varepsilon_0 = e_0/\mathbf{X}^{s_0}$ for original monic polynomial f_0 ,
with $e_0 = 0$ for $\varepsilon_0 = 0$, $f_0 = f$,
 $\langle \mathbf{A} \rangle = p < \mathbf{X}/2$ for root precision $\delta = 1/2^p$,
 $\langle \mathbf{M} \rangle = t < \mathbf{X}/2$ recommending initial splitting precision ε ,
with $t > 0$ suggesting choice of $\varepsilon = 1/2^t$,
or $t = 0$ for “autopilot” (which chooses initial $t = p + 2n$);
 p_1, p_2, p_3, p_4 stacky;

Returns: $\langle \mathbf{C} \rangle = n$, approximate roots u_1, \dots, u_n in c -code on T_2 , p_2 above u_n ,
ordered by approximately increasing real parts,
 $2n$ words w_j, t_j ($1 \leq j \leq n$) on T_3 , p_3 above t_n ,
specifying a posteriori bounds δ_j in the following way:
either $w_j > \mathbf{X}/2$ with $s_j = t_j - \mathbf{X} \cdot \text{top}(t_j)$ for $\delta_j = (w_j/\mathbf{X})/2^{s_j}$,
or $w_j = 0$ indicating $\delta_j = \infty$ (cases without valid bound),
and there is a numbering of the true roots z_j of the
original polynomial f_0 with $|z_j - u_j| < \delta_j$ for all $j \leq n$;
 $\langle \mathbf{M} \rangle = w$, $\langle \mathbf{A} \rangle - \langle \mathbf{HA} \rangle \cdot \mathbf{X} = h$ for $(w/\mathbf{X})/2^h = \Delta := \max_j \delta_j$,
 $\langle \mathbf{K} \rangle = 0$ iff $\Delta \leq \delta$ else $\langle \mathbf{K} \rangle = 1$ ($w=0$ for $\Delta = \infty$);
 p_1, p_4 as before, above p_4 multiplicity bounds m_j for the δ_j ,
 $\langle p_4 + 1 \rangle = m_1, \dots, \langle p_4 + n \rangle = m_n$, $\langle p_4 \rangle = \max_j m_j$;
 $\langle p_0 \rangle = t$, specifying final splitting bound $|f - g| < 2n/2^t$,
where $g(x) = q \cdot \prod_{j=1}^n (x - u_j)$ with $q=1$ (or q close to 1).

We add some comments about the role of that parameter t . For $f_0 = f$ ($e_0 = 0$), routine *CP1RTS* increases the splitting precision ε until it can guarantee the desired $\Delta \leq \delta$ by corresponding a posteriori bounds, whereas a positive pre-error bound $\varepsilon_0 = e_0/\mathbf{X}^{s_0} > 0$ implies a natural limitation to the root precision (safely) achievable by factorization of f . In this case the splitting precision will thus be restricted by something like $1/2^t \geq \varepsilon_0/8n$, which may end up in some weaker a posteriori worst case bound $\Delta > \delta$. Even then, however, some of the u_j may still have sufficient precision $\delta_j \leq \delta$, so this ε_0 -mechanism can also be utilized to limit the amount of work in cases, where root finding can be restricted to certain subsets of the roots.— Dummy bounds $\delta_j = \infty$ can occur only with multiplicity $m_j = n$, and only with poor pre-error bound $\varepsilon_0 > 1/2^n$ (so that $\delta_j \geq 1 - o(1)$ even for ‘sharp’ bounds).

6 Extensions of INTAPE and OUTAPE

With our new TP32 implementation *TPS02* and its more recent version *TPS04* especially written for Pentium 4 processors (see Appendix), it is now possible to run some Turing program as a child process initiated from some other *main* process, with convenient data exchange via connecting pipes. Under these implementations, the eigenroutines *INTAPE* and *OUTAPE* (see TP-book items 2.3.1 and 2.3.2) have been extended accordingly: There is a pipe “intp” serving as an *INTAPE* channel (written by the parent process and read by *INTAPE*), and a second pipe “outp” serving as *OUTAPE* channel (written by *OUTAPE* and read by the parent process).

With the particular input label $\langle \mathbf{B} \rangle = 2$, routine *INTAPE* first reads a single word k from that pipe *intp*. In case of $k \geq \mathbf{X}/2$ (or $k=0$), *INTAPE* does not fetch any further data, immediately returning $\langle \mathbf{M} \rangle = 0$ and $\langle \mathbf{A} \rangle = k$ as a “command” word, whereas for $k < \mathbf{X}/2$ it then transfers k words from the pipe upon tape T_1 in the usual way, setting $\langle \mathbf{MA} \rangle = k$.

Similarly, routine *OUTAPE* with input label $\langle B \rangle = 2$ and $\langle M \rangle = k$ writes the single word k to its pipe *outp*, for $k \geq \mathfrak{Z}/2$ (or $k=0$) returning $\langle MA \rangle = 0$, else it then transfers k words from tape T_1 to the pipe in the usual way, setting $\langle MA \rangle = k$.

Appendix

Actual versions. The table below shows the current version numbers and dates (of last updates) of the standard modules. Their associated docfiles are pure *ascii*-files to provide robust portability. They can also be used to set up some online help for TP-programming (we have done so by using *bookmark jumps* under *EMACS*).

Module	version	date	docfile
intari	220	24.11.2007	intardoc
qari	19	27. 6.2002	qardoc
recari	133	22. 2.2005	recardoc
crepar	60	16.12.2008	crepsdoc
crepex	56	16.12.2008	crepsdoc
porta	23	24. 2.2005	portsdoc
portb	20	16.12.2008	portsdoc

Availability. With regard to TP-book Section 3.1.1 (How to get this software), the December 2000 issue of these Addenda pages contained a ‘negative addendum’: because of several difficulties, we had simply to apologize for ‘nonavailability’ of this software. One of the problems was, that some time ago the disk of that ftp server with our TP-data had crashed, backup was insufficient, and there was no manpower to reinstall an operable version of E. Vetter’s ‘tpc’.

Meanwhile, however, we have been able to complete a new implementation of TP32 called *TPS02*, now being in use since 2002. This new version is a machine dependent TP-system, written for Intel’s IA-32 Architecture (Pentium II or later, for use of *MMX* instructions) under GNU-Linux, employing the ‘gcc’ with its assembler and the GNU Libc. — More on this is explained in a separate document *tps02.pdf*.

In Fall 2005, we have also finished a new version *TPS04* with special coding enhancements for Pentium 4 processors—more on that is explained in *tps04.pdf*.

Since August 2004, that system *TPS02* (and now also *TPS04*) together with the sources of our standard TP-modules (in their extra dry versions) are available for others from the author’s homepage

<http://www.informatik.uni-bonn.de/~schoe/>