Mat-1.125, Matematiikan erikoistyö: Numeerisen laskennan menetelmät tietotekniikassa

Mikko Nummelin, 44065E

arkistoversio, to 12.10.2006

Sisältö

Luku 1

Johdanto

Nykyaikainen tietotekniikan kehitys asettaa merkittäviä vaatimuksia monimutkaisten matemaattisten laskutoimitusten nopeuksille. Esimerkkeinä voi mainita varsinaisten laskentaohjelmien (Matlab [1], Maple [2], Yacas [3] jne.) lisäksi myös korkealaatuisella ja nopealla grafiikalla varustetut tietokonepelit, samoin kuin sääennusteet, isot lujuuslaskelmat tai tarkat simulaatiot. Tämä vaatii käytännössä sitä, että laskennan eri osa-alueet on toteutettu mahdollisimman tehokkaasti:

- Itse ohjelma ja ohjelman syöte on hyvin suunniteltu ja mahdollisimman virheetön.
- Käyttöjärjestelmässä olevat matemaattiset kirjastot on toteutettu hyvin.
- Muistia ja mikroprosessorin tehoa on riittävästi.

Tämä erikoistyö käsittelee kahta jälkimmäistä osa-aluetta ja ottaa tutkittavaksi erikseen alkeislaskutoimitusten ja alkeisfunktioiden tehokkaat toteutukset. Monissa tapauksissa on syytä käyttää eri algoritmeja pienen ja suuren tarkkuuden laskutoimituksille. Kaikkein alkeellisimmat laskutoimitukset on toteutettu käytännössä prosessorin elektroniikan tai mikrokoodin tasolla kun taas suuren tarkkuuden laskutoimitukset on useimmiten ohjelmoitu korkeamman tason ohjelmointikielillä kuten C/C++:lla [5], [6] tai Fortranilla [7].

Luku 2

Käsitteitä ja perusteita

Tässä luvussa käsitellään lyhyesti loogiset operaatiot yhteen- ja vähennyslaskujen suorittamisen apuna, suuren tarkkuuden kokonaislukujen yhteen- ja vähennyslaskun "apuvälineet" `AMD64`-prosessoriperheessä [8], liukulukujen ANSI/IEEE-standardi [9] sekä polynomien evaluointi Hornerin säännöllä.

2.1 Yhteen- ja vähennyslaskut

Lause 1 (Binaarilukujen yhteenlasku) *Olkoot x ja y binaarilukuja seuraavasti:*

$$x = \sum_{k=0}^{n-1} x_k 2^k, \quad y = \sum_{k=0}^{n-1} y_k 2^k \tag{2.1}$$

missä $\forall k,\ x_k, y_k \in \{0,1\}$. Tällöin voidaan näiden summa $z = x+y = \sum_{k=0}^{n} z_k 2^k$ lausua seuraavasti:

$$\begin{aligned} m_0 &= 0 \\ m_{k+1} &= (x_k \wedge y_k) \vee (x_k \wedge m_k) \vee (y_k \wedge m_k) \\ z_k &= x_k \not\leftrightarrow y_k \not\leftrightarrow m_k \end{aligned} \tag{2.2}$$

Kaavassa (2.2) esiintyvät loogiset operaatiot on kuvailtu taulukossa 2.1. On myös huomattava ns. muistinumeron m_k *käyttö, koska kaikki bitit $z_k \in \{0,1\}$.*

Lause 2 (Binaarilukujen vähennyslasku) *Olkoon x ja y kuten lauseessa 1 Tällöin voidaan näiden erotus z (kuten edellä) lausua seuraavasti:*

$$\begin{aligned} m_0 &= 0 \\ m_{k+1} &= (\neg x_k \wedge y_k) \vee (\neg x_k \wedge m_k) \vee (y_k \wedge m_k) \\ z_k &= x_k \not\leftrightarrow y_k \not\leftrightarrow m_k \end{aligned} \tag{2.3}$$

Tässä muistinumeroa vastaa lainaus m_k.

Nämä lauseet voidaan yleistää koskemaan suuremmissa kannoissa esitettyjä lukuja. Esimerkiksi, jos halutaan laskea kahden q-kantaisen luvun yhteenlasku, missä siis

$$x = \sum_{k=0}^{n-1} x_k q^k, \quad y = \sum_{k=0}^{n-1} y_k q^k \tag{2.4}$$

Taulukko 2.1: Loogisten operaatioiden merkitys

x	y	$\neg x$	$x \wedge y$	$x \vee y$	$x \not\leftrightarrow y$
0	0	1	0	0	0
1	0	0	0	1	1
0	1	1	0	1	1
1	1	0	1	1	0

ja $z = x + y$, menetellään seuraavasti:

Asetetaan $m_0 = 0$ ja lasketaan $k = 0$ alkaen $z_k^* = x_k + y_k + m_k$. Jos nyt jollakin askeleella osoittautuu, että $z_k^* \geq q$, on asetettava $z_k = z_k^* - q$ ja $m_{k+1} = 1$, muussa tapauksessa $z_k = z_k^*$ ja $m_{k+1} = 0$. Ideana on siis se, että kun kerrointa z_k lasketaan, huomioidaan rajoite $\forall k,\ z_k \in [0, q-1]$ ja siis *siirretään* ylijäämäbitti seuraavaan yhteenlaskuun.

Mikroprosessoreissa tätä on helpotettu käyttämällä hyväksi *muistinumerolippua* (engl. carry flag) ja erityistä summauskäskyä, joka huomioi automaattisesti muistinumerolipun arvon $\in \{0, 1\}$ lisäämisen tulokseen. `AMD64`-arkkitehtuurissa [8] tätä varten on käsky `adc` ja vähennyslaskun lainausta varten (tarkempi käsittely sivuutetaan), käsky `sbb`.

Esimerkki 1 (`AMD64`-arkkitehtuurin muistinumerolippu) *Seuraava konekieliohjelma suorittaa kahden mielivaltaisen tarkkuuden luvun summauksen käyttäen hyväksi muistinumerolippua:*

```
/* Kahden mielivaltaisen pituisen kokonaisluvun summa
 * muistinumerolippua hyväksikäyttäen. Kutsutapa on
 *
 * sum_xl(n,r,a1,a2)
 * u64 n,*r,*a1,*a2;
 *
 * r osoittaa lopputulokseen, a1 ja a2 summattaviin,
 * n on e.m:ssa olevien 64-bittisten sanojen lukumäärä.
 */

.globl sum_xl
        .type   sum_xl, @function
sum_xl:
        testq   %rdi, %rdi      /* Onko n==0? */
        je      end             /* Jos on, niin ulos */
        xorq    %r8, %r8        /* i=0; */
        clc                     /* muistinumerolippu alas */
loop:
        movq    (%rcx,%r8,8), %rax
        adcq    (%rdx,%r8,8), %rax
        movq    %rax, (%rsi,%r8,8)      /* r[i]=a1[i]+'a2[i] */
        incq    %r8                     /* i++; */
        decq    %rdi                    /* n--; */
        jne     loop                    /* if(n==0) goto loop */
end:
```

```
        ret                     /* ulos */
```

Kannattaa huomata erityisesti sudenkuoppa käskyn `decq %rdi` *kohdalla, jossa on helppo tehdä ohjelmointivirhe. Jos kirjoittaisi hätäisesti sen tilalle esim:* `cmpq %r8, %rdi`*, idealla, että verrataan C-tyylisesti*

```
if(i!=n) goto loop;
```

mennään metsään, koska `cmpq` *tärvelee muistinumerolipun arvokkaan sisällön. Siksi on tyydyttävä käyttämään parametria* `n` *"toisena laskurina", jonka pudotessa nollaan nollalippu nousee, mutta muistinumerolippu pysyy koko ajan siinä asennossa, mihin se oli käskyn* `adcq (%rdx,%r8,8), %rax` *jälkeen asettunut. Seuraavalla kierroksella sama* `adcq`*-käsky lisää sitten ylimääräisen* 1*:en, jos muistinumerolippu oli ylhäällä.*

Tätä konekieliohjelmaa voi testata vaikkapa C-ohjelmalla

```
#include <stdio.h>

#define NUM_SIZE        0x40

typedef unsigned long long      u64;

void sum_xl(u64,u64 *,u64 *,u64 *);

main() {
        int i;
        u64 r[NUM_SIZE],a1[NUM_SIZE],a2[NUM_SIZE];
        for(i=1;i<NUM_SIZE-1;i++) {
                a1[i]=0xffffffffffffffff;
                a2[i]=0x0000000000000000;
        }
        a1[0]=0xfffffffffffffff8;
        a2[0]=0x000000000000000a;
        a1[NUM_SIZE-1]=0x0000000000000000;
        a2[NUM_SIZE-1]=0x0000000000000000;
        printf("a1:\n");
        for(i=0;i<NUM_SIZE;i++) {
                printf("%3d: %16llx\n",i,a1[i]);
        }
        printf("a2:\n");
        for(i=0;i<NUM_SIZE;i++) {
                printf("%3d: %16llx\n",i,a2[i]);
        }
        sum_xl(NUM_SIZE,r,a1,a2);
        printf("r:\n");
        for(i=0;i<NUM_SIZE;i++) {
                printf("%3d: %16llx\n",i,r[i]);
        }
}
```

2.2 Liukuluvut

Liukuluvut ovat lukuja tyyppiä $\pm x\ q^m$, missä x on mantissa, q kantaluku ja m eksponentti. Erityisiä merkintätapoja näille on xEm, mikäli $q = 10$ ja $x@m$, mikäli $q = 2$. Useat saman kannan liukuluvut ovat keskenään samansuuruisia, esim: $12E3 = 1.2E4 = 12000$. Tästä syystä on käytössä erilaisia määritelmiä sille, milloin liukuluku on *normalisoidussa* muodossa, esimerkiksi liukulukujen ANSI/IEEE-standardin [9] mukaan binaarisen liukuluvun mantissa voi olla väliltä $[1, 2)$. Sama standardi määrittelee tietokoneiden prosessoreissa käytettävät liukuluvut joko *yksinkertaisen tarkkuuden* liukuluvuiksi, jolloin normalisoitua mantissaa varten varataan 23 bittiä, suunnattua eksponenttia varten 8 bittiä ja merkkiä varten 1 bitti tai *kaksinkertaisen tarkkuuden* liukuluvuiksi, jolloin mantissaa varten varataan 52 bittiä, suunnattua eksponenttia varten 11 bittiä ja merkkiä varten taaskin 1 bitti. Eräät yhdistelmät on varattu tarkoittamaan ± 0:aa, $\pm\infty$:tä ja "epälukuja" eli `NaN`:ia. Jälkimmäinen tapaus tulee kyseeseen esim. "laittoman" jakolaskun $\frac{0}{0}$ tuloksena. Tarkempia tietoja varten ks. [8] ja/tai [9].

2.2.1 Vähennyslaskuongelma

Liukuluvuilla operoitaessa usein unohtuu ns. *vähennyslaskuongelma* (engl. catastrophic cancellation). Kysymyksessä on suhteellisen tarkkuuden menetys vähennettäessä hyvin lähellä toisiaan olevia liukulukuja toisistaan. Esimerkiksi

$$1.10011@0 - 1.10001@0 = 0.00010@0 = 1.0\mathit{0000}@(-4) \tag{2.5}$$

jos mantissanpituus olisi ollut 5 bittiä. Tulos on kyllä periaatteessa oikein, mutta jos vähennettävät olisivat olleet likiarvoja jostakin, niin suhteellista tarkkuutta on menetetty valtavasti, tuloksessa olevat kursivoidut nollat kuvaavat lopputulokseen "tyhjästä liu'utettuja" nollia. Jos siis on mahdollista organisoida laskut niin, että välituloksissa ei esiinny tällaista laskutoimitusta, niin kannattaa tehdä.

Esimerkki 2 *Neliöjuurilausekkeen*

$$\sqrt{1+z} - 1 \tag{2.6}$$

tilalla kannattaa käyttää mieluummin lavennettua lauseketta

$$\frac{(\sqrt{1+z}+1)(\sqrt{1+z}-1)}{\sqrt{1+z}+1} = \frac{z}{\sqrt{1+z}+1}. \tag{2.7}$$

2.3 Hornerin sääntö

Hornerin sääntö on tärkeä evaluoitaessa polynomeja ja katkaistuja sarjoja. Se on "perusmuodossaan"

$$\sum_{k=0}^{n-1} a_k z^k = a_0 + z\sum_{k=0}^{n-2} a_{k+1} z^k \tag{2.8}$$

Tietokonelaskentaa varten tämä kaava on kuitenkin sellaisenaan huono, johtuen sen eturekursiivisuudesta. Ongelma syntyy siitä, että tuloa $z\sum_{k=0}^{n-2} a_{k+1}z^k$ ei

```
double horner(int n, double *a, double z) {
        int i;
        double t;
        t=a[n-1];
        for(i=n-2;i>=0;i--) {
                t=a[i]+t*z;
        }
        return t;
}
```

Kuva 2.1: Hornerin sääntö iteraatiolla

voida evaluoida ennen kuin koko jälkimmäisen summan arvo on tiedossa ja tämä kuormittaa tietokoneessa pinoa ja lisää muistiliikennettä, jos polynomi on hyvin moniterminen. Evaluointijärjestystä on kuitenkin mahdollista muuttaa niin, että kertoimet a_k käsiteltäisiinkin järjestyksessä suurimmasta pienimpään! Havainnollistetaan tätä tapauksella $n = 5$:

$$a_0 + a_1 z + a_2 z^2 + a_3 z^3 + a_4 z^4 = a_0 + z\left(a_1 + z\left(a_2 + z\left(a_3 + a_4 z\right)\right)\right) \tag{2.9}$$

Tämä on helppo purkaa seuraavanlaisiksi sijoituksiksi laskemalla sulkulausekkeet sisimmästä uloimpaan:

$$\begin{aligned} t_0 &:= a_4 \\ t_1 &:= a_3 + t_0 z \\ t_2 &:= a_2 + t_1 z \\ t_3 &:= a_1 + t_2 z \\ t_4 &:= \underline{a_0 + t_3 z} \end{aligned} \tag{2.10}$$

ja algoritmin rakenne on selvä, alustetaan t polynomin suurimmalla kertoimella ja sen jälkeen suoritetaan peräjälkeen kertominen z:lla ja seuraavaksi suurimman kertoimen lisääminen, kunnes viimeisenä lisätään 0:nnen termin kerroin, jolloin laskutoimitus on valmis. C-kielellä [5],[6] tämä on esitetty kuvassa 2.1. Hornerin sääntö iteraatiolla on käytössä Yacas:issa [3] suurten polynomien evaluoinnissa (esim. eräät Taylorin sarjat), muttei `mpfr`:ssä [11]. Omia sovelluksia varten se olisi tietysti helposti sovellettavissa, sen sijaan useimmissa Taylorin sarjoissa on mahdollisuus pitää muistissa apumuuttujia ja päivittää niitä, ks. myöhempää lukua kosinista 4.3.2.

Hornerin sääntöä voidaan soveltaa myös rinnakkaislaskennassa hajauttamalla polynomi ensin kahtia

$$\sum_{k=0}^{n-1} a_k z^k = \sum_{k=0}^{n/2-1} a_k z^k + z^{n/2} \sum_{k=0}^{n/2-1} a_{k+n/2} z^k \tag{2.11}$$

ja laskemalla summat erikseen eri suoritussäikeissä (jolloin ne voivat moniprosessorisissa järjestelmissä päätyä eri prosessoreille). UNIX-järjestelmissä tällainen tapahtuu muodostamalla ensin prosessien välille putki `pipe(2)`-systeemikutsulla, sitten jakamalla prosessi `fork(2)`-systeemikutsulla ”vanhempi”- ja ”lapsi”-prosesseiksi

sekä lukemalla ja kommunikoimalla näiden prosessien välillä `write(2)` ja `read(2)`-systeemikutsuilla. Kuvassa 2.2 esitelty esimerkki on Linux Programmer's Manual:ista [4] `pipe(2)`-systeemikutsua käsittelevän luvun kohdalta.

2.4 Suuren tarkkuuden matemaattiset kirjastot

Koska tietokoneiden rekistereihin mahtuu vain rajallinen määrä tietoa (esim. aiemmin esillä olleessa `AMD64`-arkkitehtuurissa 64 bittiä per "tavallinen" rekisteri ja 128 bittiä per multimediarekisteri), asettaa tämä rajoituksia sille laskutarkkuudelle, joka voidaan saada käyttöön suoraan konekäskyillä. Joskus kuitenkin tarvitaan korkeampaa tarkkuutta, ja tätä varten on kehitetty erityisiä matemaattisia kirjastoja, jotka implementoivat suuren tarkkuuden luvut taulukkoina (ks. kuvaa 2.3) ja näiden operaatiot näihin taulukoihin kohdistuvina ohjelmina. Tässä erikoistyössä käytetään tarpeen mukaan esimerkkinä `GNU GMP`:tä [10] ja sen oheen suunniteltua funktioapukirjastoa `mpfr`:ää [11].

```
#include <sys/wait.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int
main(int argc, char *argv[])
{
    int pfd[2];
    pid_t cpid;
    char buf;

    assert(argc == 2);

    if (pipe(pfd) == -1) { perror("pipe"); exit(EXIT_FAILURE); }

    cpid = fork();
    if (cpid == -1) { perror("fork"); exit(EXIT_FAILURE); }

    if (cpid == 0) {    /* Child reads from pipe */
        close(pfd[1]);          /* Close unused write end */

        while (read(pfd[0], &buf, 1) > 0)
            write(STDOUT_FILENO, &buf, 1);

        write(STDOUT_FILENO, "\n", 1);
        close(pfd[0]);
        _exit(EXIT_SUCCESS);

    } else {            /* Parent writes argv[1] to pipe */
        close(pfd[0]);          /* Close unused read end */
        write(pfd[1], argv[1], strlen(argv[1]));
        close(pfd[1]);          /* Reader will see EOF */
        wait(NULL);             /* Wait for child */
        exit(EXIT_SUCCESS);
    }
}
```

Kuva 2.2: Prosessien jakaminen UNIX-järjestelmissä

```
typedef struct
{
  int _mp_alloc;              /* Number of *limbs* allocated and pointed
                                 to by the _mp_d field.  */
  int _mp_size;               /* abs(_mp_size) is the number of limbs the
                                 last field points to.  If _mp_size is
                                 negative this is a negative number.  */
  mp_limb_t *_mp_d;           /* Pointer to the limbs.  */
} __mpz_struct;
```

Kuva 2.3: GNU GMP:n suuren tarkkuuden lukujen perusrakenne (tiedostosta `/usr/include/gmp.h`). `__mpz_struct` muodostaa muistiosoitteesta `_mp_d` alkavan `_mp_alloc`-kokoisen taulukon, jota käytetään suurten kokonaislukujen esittämiseen. Murtoluvut ja liukuluvut on toteutettu tällaisia rakenteita yhdistämällä.

Luku 3

Kerto- ja jakolaskut

Kerto- ja jakolaskut ovat tietokoneissa olennaisia peruslaskutoimituksia. Monissa arkkitehtuureissa ne on toteutettu myös mikroprosessorikäskyinä, jolloin sellaisen käskyn suorittaminen tyypillisesti tarkoittaa tehokkaan prosessorin sisäisen mikrokoodiohjelman suorittamista. Tässä luvussa käsitellään erikseen, millaiset kertolaskumenetelmät sopivat pienen tarkkuuden ja millaiset suuren tarkkuuden luvuille. Lisäksi käsitellään nopean kertolaskun käyttö jakolaskun apuna. Aloitetaan tärkeästä apulauseesta

Lause 3 (Diskreetti konvoluutio) *Kahden itseisesti suppenevan sarjan Cauchy-tulo on*

$$\left(\sum_{k=0}^{\infty} a_k z^k\right)\left(\sum_{k=0}^{\infty} b_k z^k\right) = \sum_{k=0}^{\infty} c_k z^k \tag{3.1}$$

missä

$$c_k = \sum_{i=0}^{k} a_i b_{k-i} \tag{3.2}$$

eli

$$(c_k) = (a_k) * (b_k). \tag{3.3}$$

Jono (c_k) *on jonojen* (a_k) *ja* (b_k) diskreetti konvoluutio.

3.1 Pitkä kertolaskumenetelmä

Tämä on varsin triviaali menetelmä, jossa kaavan (3.1) vasen puoli vain kirjoitetaan uusiksi seuraavasti (lisäoletuksena, että polynomit nyt ovat n-asteisia):

$$\begin{aligned}\left(\sum_{k=0}^{n} a_k z^k\right)\left(\sum_{k=0}^{n} b_k z^k\right) &= \\ \left(\sum_{k=0}^{n} a_k z^k\right)\left(\sum_{i=0}^{n} b_i z^i\right) &= \\ \left(\sum_{k=n}^{0} a_k z^k \left(\sum_{i=n}^{0} b_i z^i\right)\right)& \end{aligned} \tag{3.4}$$

```
  10011
x 10101
=======
  10101
     10101
      10101
===========
   ***
  110001111
```

Kuva 3.1: Binaarilukujen kertolasku pitkällä kertolaskumenetelmällä. Tähdet ovat muistinumeron käytön merkkinä. Muutamia alimpia bittejä olisi voinut katkaista laskutoimituksesta pois, jos olisi haluttu tyytyä esim. vain 2^{-4}:n suhteelliseen tarkkuuteen.

Huomaa, että kaavassa (3.4) on summaussuunta käännetty. Tämä johtuu siitä, että joskus tarvitaan tulosta vain likiarvo ja tällöin on tietysti syytä evaluoida ensisijaisesti eniten merkitseviä bittejä kuin vähemmän merkitseviä. Kysymykseen tulee myös se, että jonosta (b_k) olisi samankaltaisesta pyöristyssyystä katkaistu pois matalaindeksisiä termejä. Binaarilukuesimerkki, jossa laskutoimitus on järjestetty allekkain kuvassa 3.1.

`GNU GMP` [10] ja `mpfr` [11] käyttävät pitkää kertolaskumenetelmää (joskin korkeammalla kantaluvulla kuin 2:lla) sellaisiin kertolaskuihin, joissa kerrottavat eivät ole kovin suuria. Käytännössä tätä säädetään makroilla `SQR_BASECASE_THRESHOLD`, `SQR_KARATSUBA_THRESHOLD` ja `SQR_TOOM3_THRESHOLD`, jotka määrittävät, tuleeko käyttöön pitkän kertolaskumenetelmän sijasta jompikumpi kahdesta seuraavana esiteltävistä menetelmistä. Fourier-menetelmä *ei ole* näissä matemaattisissa kirjastoissa "automaattisesti" käytössä, koska käytännössä on havaittu Toom-3-menetelmän olevan numeerisesti lähes yhtä tehokas (ks. edempää).

3.2 Toom-Cook-menetelmät

Näissä menetelmissä hyödynnetään interpolaatiopolynomeja. Tarkastellaan esim. seuraavaa polynomien kertolaskua:

$$\underbrace{\left(\sum_{k=0}^{m} a_k z^k\right)}_{=P(z)}\underbrace{\left(\sum_{k=0}^{n} b_k z^k\right)}_{=Q(z)} = \sum_{k=0}^{m+n} c_k z^k \tag{3.5}$$

Huomio kiinnittyy siihen, että m ja n-asteisten polynomien tulo on $m+n$-asteen polynomi. Toisaalta tällaisen polynomin kiinnittää $m+n+1$ datapistettä. Ideana on siis se, että valitaan $m+n+1$ referenssipistettä, evaluoidaan molemmat polynomit, P ja Q näissä ja lasketaan pisteittäin tulot. Sen jälkeen sovitetaan saadun datapisteistön läpi interpolaatiopolynomi, joka onkin suoraan polynomi PQ. Tutkitaan erikseen kolmea erityyppistä Toom-Cook-menetelmää

3.2.1 Toom-2 ja Karatsuban algoritmi

Olkoon $P(z) = a_0 + a_1 z$ ja $Q(z) = b_0 + b_1 z$. Valitaan referenssipisteistöksi $z \in \{0, 1, -1\}$ ja saadaan

$$\left\{ \begin{array}{lcll} c_0 & = & P(0)Q(0) & = & a_0 b_0 \\ c_0 + c_1 + c_2 & = & P(1)Q(1) & = & (a_0 + a_1)(b_0 + b_1) \\ c_0 - c_1 + c_2 & = & P(-1)Q(-1) & = & (a_0 - a_1)(b_0 - b_1) \end{array} \right. \tag{3.6}$$

On selvää, ettei koko ryhmän (3.6) evaluoiminen kovin paljoa yksinkertaista itse kertolaskua. Huomio kiinnittyy kuitenkin seikkaan, että jos c_0 eliminoidaan, saadaan ryhmä

$$\left\{ \begin{array}{lcl} c_1 + c_2 & = & (a_0 + a_1)(b_0 + b_1) - a_0 b_0 \\ -c_1 + c_2 & = & (a_0 - a_1)(b_0 - b_1) - a_0 b_0 \end{array} \right. \tag{3.7}$$

ja siksi toiseksi, ilmeisesti on $c_2 = a_1 b_1$ kaavan (3.2) mukaan. Eliminoimalla tällä säännöllä c_2 yhtälöparin (3.7) ylemmästä yhtälöstä, saadaan

$$c_1 = (a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1 \tag{3.8}$$

Riittää siis suorittaa pitkän kertolaskualgoritmin vaatiman neljän kertolaskun sijasta vain kolme, eli

$$\begin{array}{lcl} c_0 & = & a_0 b_0 \\ c_2 & = & a_1 b_1 \\ c_1 & = & (a_0 + a_1)(b_0 + b_1) - c_0 - c_2. \end{array} \tag{3.9}$$

Kaavaa (3.9) sanotaan *Karatsuban algoritmiksi*. Se on alunperin keksitty ilman viittausta Toom-Cook-menetelmiin, mutta on mielenkiintoista huomata, että se on erikoistapaus yleisemmästä kertolaskumenetelmästä.

3.2.2 Toom-3

Tämä on erittäin tavallinen Toom-Cook-menetelmä. Siinä kerrottavat polynomit P ja Q ovat toisen asteen polynomeja ja referenssipisteistöksi valitaan $z \in \{0, 1, -1, 2, -2\}$. Se johtaa yhtälöryhmään

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ 2 & 4 & 8 & 16 \\ -2 & 4 & -8 & 16 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix} = \begin{pmatrix} P(1)Q(1) - P(0)Q(0) \\ P(-1)Q(-1) - P(0)Q(0) \\ P(2)Q(2) - P(0)Q(0) \\ P(-2)Q(-2) - P(0)Q(0) \end{pmatrix} \tag{3.10}$$

toisin sanoen

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ 2 & 4 & 8 & 16 \\ -2 & 4 & -8 & 16 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix} = \begin{pmatrix} (a_0 + a_1 + a_2)(b_0 + b_1 + b_2) - a_0 b_0 \\ (a_0 - a_1 + a_2)(b_0 - b_1 + b_2) - a_0 b_0 \\ (a_0 + 2a_1 + 4a_2)(b_0 + 2b_1 + 4b_2) - a_0 b_0 \\ (a_0 - 2a_1 + 4a_2)(b_0 - 2b_1 + 4b_2) - a_0 b_0 \end{pmatrix} \tag{3.11}$$

jonka ratkaisu on

$$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix} = \frac{1}{24} \begin{pmatrix} 16 & -16 & -2 & 2 \\ 16 & 16 & -1 & -1 \\ -4 & 4 & 2 & -2 \\ -4 & -4 & 1 & 1 \end{pmatrix} \begin{pmatrix} (a_0+a_1+a_2)(b_0+b_1+b_2)-a_0b_0 \\ (a_0-a_1+a_2)(b_0-b_1+b_2)-a_0b_0 \\ (a_0+2a_1+4a_2)(b_0+2b_1+4b_2)-a_0b_0 \\ (a_0-2a_1+4a_2)(b_0-2b_1+4b_2)-a_0b_0 \end{pmatrix}. \tag{3.12}$$

Järkeä voi jälleen käyttää huomioimalla että $c_0 = a_0b_0$, $c_4 = a_2b_2$ ja että suuri osa kertolaskuista on 2:n potensseilla kertomisia, mitkä tietokoneissa merkitsevät pelkästään bittien liu'uttamista vasemmalle tai liukuluvun eksponentin nostoa, eivätkä siis ole kompleksisuudeltaan varsinaisia "kertomisia". Korostettakoon, että tämä menetelmä on järkevä vain *suurten* lukujen kertolaskujen hajottamiseksi. Kun luvut ovat tarpeeksi pieniä, tulee siirtyä Karatsuban algoritmiin ja sitten pitkään kertolaskumenetelmään.

3.2.3 Kompleksinen referenssipisteistö ja Fourier-muunnos

Merkitään

$$W_n = \exp\left(\frac{2\hat{\imath}\pi}{n}\right) \tag{3.13}$$

jolloin pisteet $\{W_n^k : \ k \in 0..n-1\}$ muodostavat yksikköympyrälle tasasivuisen n-kulmion kärjet. Valitaan nyt referenssipisteistöksi origo ja mainitut pisteet, eli $z \in \{0, 1, W_n, W_n^2, ...W_n^{n-1}\}$. Yhtälöryhmä koostuu nyt triviaalista yhtälöstä $c_0 = a_0b_0$ sekä useammista tyyppiä

$$\sum_{\ell=0}^{n-1} W_n^{\ell k} c_\ell = \left(\sum_{\ell=0}^{n-1} W_n^{\ell k} a_\ell\right)\left(\sum_{\ell=0}^{n-1} W_n^{\ell k} b_\ell\right) \tag{3.14}$$

olevista yhtälöistä. Jos merkitään kompleksisina vektoreina

$$\mathbf{w_k} = (1, W_n^k, W_n^{2k}, .., W_n^{(n-1)k})^T \tag{3.15}$$

niin $\mathbf{w_0}..\mathbf{w_{n-1}}$ muodostavat keskenään ortonormaalin kannan sisätulon

$$< \mathbf{u}, \mathbf{v} >= \frac{1}{n}\mathbf{v}^*\mathbf{u} \tag{3.16}$$

nojalla. Tässä siis $*$ on kompleksikonjugoinnin ja transponoinnin yhdistelmä. Tämä tarkoittaa sitä, että yhtälöryhmän (3.14) vasemmalle puolelle tuleva matriisi on mahdollista kääntää vain kompleksikonjugoimalla, transponoimalla ja jakamalla n:llä.

Polynomien P ja Q tulo voidaan siis laskea esimerkiksi ottamalla molemmista Fourier-muunnos kaavan

$$\mathcal{F}_k(\mathbf{a}) = \sum_{\ell=0}^{n-1} W_n^{\ell k} a_\ell \tag{3.17}$$

mukaan (Q:n kertoimille sama juttu) ja laskemalla sitten pisteittäin tulot

$$\gamma_k = \mathcal{F}_k(\mathbf{a})\mathcal{F}_k(\mathbf{b}) \tag{3.18}$$

Tämän jälkeen lopputulos käänteismuunnetaan kaavalla

$$c_k = \frac{1}{n} \sum_{\ell=0}^{n-1} W_n^{-\ell k} \gamma_\ell. \tag{3.19}$$

Tähän tulokseen oltaisiin voitu päätyä myös tarkastelemalla sitä konvoluution ominaisuutta, että sitä vastaa Fourier-muunnospuolella pisteittäinen tulo, mutta tämä käsittelytapa on valittu sen seikan tähdentämiseksi, että Fourier-muunnoksiin perustuva kertolaskumenetelmä on myös Toom-Cook-menetelmän erikoistapaus.

3.2.4 Nopea Fourier-muunnos

Fourier-muunnoksessa on etuna se, että se on mahdollista ottaa kumpaankin suuntaan nopealla kahtiajakomenetelmällä, mikäli muunnettavat vektorit ovat 2^N-jäsenisiä. [1] Kahtiajakomenetelmän idea on

$$\mathcal{F}_k(\mathbf{a}) = \sum_{\ell=0}^{n-1} W_n^{\ell k} a_\ell \tag{3.20}$$

$$= \sum_{\ell=0}^{n/2-1} W_n^{2\ell k} a_{2\ell} + W_n^k \sum_{\ell=0}^{n/2-1} W_n^{2\ell k} a_{2\ell+1} \tag{3.21}$$

eli sanallisesti

$$\begin{aligned} \mathcal{F}(\text{kaikki indeksit}) = {} & \mathcal{F}(\text{parilliset indeksit}) + \\ & \text{siirtokerroin} \cdot \mathcal{F}(\text{parittomat indeksit}) \end{aligned} \tag{3.22}$$

Siirtokerroin on kaavassa (3.21) muutettava muotoon W_n^{-k} muunnettaessa käänteiseen suuntaan sen lisäksi, että tulos on jaettava n:llä.

Esimerkki 3 *Halutaan laskea binaariluvuilla* $5 \cdot 3$. *Tätä varten asetetaan*

$$\mathbf{a} = (1\ 0\ 1\ 0)^T \tag{3.23}$$

ja

$$\mathbf{b} = (1\ 1\ 0\ 0)^T \tag{3.24}$$

Nyt on

$$\begin{aligned} \mathcal{F}((1\ 0\ 1\ 0)^T) &= \mathcal{F}((1\ 1)^T) \\ &= 1 + W_2^k \end{aligned} \tag{3.25}$$

ja

$$\begin{aligned} \mathcal{F}((1\ 1\ 0\ 0)^T) &= \mathcal{F}((1\ 0)^T) + W_4^k \mathcal{F}((1\ 0)^T) \\ &= 1 + W_4^k. \end{aligned} \tag{3.26}$$

γ *voidaan nyt generoida lausekkeesta* $(1 + W_2^k)(1 + W_4^k)$, *jolloin saadaan*

$$\gamma = (4\ 0\ 0\ 0)^T \tag{3.27}$$

[1] Myös muunlaisiakin vektoreita varten on nopeita Fourier-muunnoskaavoja, mutta tässä ne sivuutetaan, koska kertolaskua varten riittää se, että vektorit tarvittaessa täydennetään suurempikokoisiksi.

ja lopulta

$$\mathcal{F}^{-1}(\gamma) = \frac{1}{4} W_0^k = (1\ 1\ 1\ 1)^T \tag{3.28}$$

Vastaus on oikein, koska todellakin $1111_2 = 15$*. On huomattava, että jos olisi laskettu vähänkin isommilla luvuilla, esim.* $3 \cdot 6$ *olisi pitänyt ennemminkin alustaa* $\mathbf{a} = (0\ 1\ 1\ 0\ 0\ 0\ 0\ 0)^T$ *ja* $\mathbf{b} = (1\ 1\ 0\ 0\ 0\ 0\ 0\ 0)^T$*, koska muuten olisi ylimpiä bittejä "leikkautunut" pois.*

Nopeiden Fourier-muunnosten menetelmä on varsinaisesti käytännöllinen vain hyvin suurille luvuille, eikä se ole lainkaan käytössä automaattisesti `GMP`:ssä [10] eikä `mprf`:ssä [11], kuten aiemmin jo todettiin. Eräs merkittävä matemaattinen kirjasto, joka on laadittu nopeiden Fourier-muunnosten ottamiseksi, on FFTW [12].

3.3 Jakolasku

3.3.1 Jakokulma

Tavallisin jakolaskumenetelmä pienille luvuille on ns. pitkä jakolaskumenetelmä, joka on sama kuin "kynällä ja paperilla" toimitettava jakokulma. Laskettaessa y/x:ää q-kantaisilla luvuilla pyritään ensin liu'utuksien avulla saamaan aikaan tilanne, jossa x on korkeintaan y, mutta y alle qx, ts.

$$\frac{y}{q} < x \leq y. \tag{3.29}$$

Sen jälkeen sovelletaan peräjälkeen kaavaa

$$\frac{y}{x} = d + \frac{y - d \cdot x}{x} \tag{3.30}$$

missä d on uusi numero q-kantaisessa kehitelmässä (y/x:n kokonaisosa). Sen jälkeen sijoitetaan y:n paikalle $y - d \cdot x$ ja jatketaan kertomalla x:ää taas sopivan monta kertaa q:lla, jotta ehto (3.29) täyttyisi. Jos joudutaan kertomaan q:lla useamman kuin kerran, lisätään kehitelmään näillä kerroilla 0. Jakokulma on `GMP`:ssä [10] ja `mpfr` [11] tavanomainen menetelmä lyhyillä luvuilla ja pienellä tarkkuudella laskettaessa. Sovelluksissa, joissa ei ole nopeaa kertolaskumenetelmää ja tarkkuusvaatimus pieni, käytetään jakokulmaa "bitti kerrallaan"-periaatteella tyyliin

$$\frac{y}{x} = \begin{cases} 2\left(\frac{y}{2x}\right) & , \quad y \geq 2x \\ 1 + \frac{y-x}{x} & , \quad x < y < 2x \\ 1 & , \quad y = x \\ 2^{-1}\left(\frac{2y}{x}\right) & , \quad y < x \end{cases} \tag{3.31}$$

3.3.2 Geometrinen sarja päättymättömänä tulona

Oletetaan, että käytössä on nopea kertolaskumenetelmä ja että jakaja ja jaettava ovat huomattavan suuria. Tällöin tulee kyseeseen kaavan $y/x = x^{-1}y$ hyväksikäyttö. Tässä tapauksessa on myös syytä operoida normalisoiduilla binaarisilla liukuluvuilla. Liu'utusten avulla on aina mahdollista palauttaa mielivaltaisen luvun invertoiminen jonkun luvun muotoa $1 - x, x \in [0, 1/2)$ invertoimiseksi

```
#include <stdio.h>

int main(int argc, char *argv[]) {
        int i;
        double x,r;
        printf("Syötä invertoitava luku 0.5<x<=1: ");
        scanf("%lf",&x);
        x=1-x;
        r=1.0;
        for(i=0;i<10;i++) {
                r=r+r*x;
                x=x*x;
                printf("%17.15lf\n",r);
        }
        return 0;
}
```

Kuva 3.2: Jakolasku päättymättömän tulon avulla (C-ohjelma)

ja tätä voidaan käyttää tehokkaasti hyväksi. Tällä lukualueella pätee Taylorin sarjakehitelmä

$$(1-x)^{-1} = \sum_{k=0}^{\infty} x^k = 1 + x + x^2 + x^3 + ... \tag{3.32}$$

Mutta toisaalta tämä voidaan vaihtoehtoisesti lausua päättymättömänä tulona

$$(1+x)(1+x^2)(1+x^4)(1+x^8)... = \prod_{k=0}^{\infty} \left(1 + x^{2^k}\right) \tag{3.33}$$

eli algoritmina

$$\begin{aligned} t_0 &:= x \cdot x \\ r_0 &:= 1 + x \\ t_{k+1} &:= t_k \cdot t_k \\ r_{k+1} &:= r_k \cdot (1 + t_k) \end{aligned} \tag{3.34}$$

Kaksi tuloa joka kierroksella kaksinkertaistavat bittien/desimaalien määrän. Tätä on havainnollistettu esimerkkiohjelmalla (kuva 3.2) ja sen tulosteella 3.3.

```
Syötä invertoitava luku 0.5<x<=1: .7
1.300000000000000
1.417000000000000
1.428477700000000
1.428571422421897
1.428571428571429
1.428571428571429
1.428571428571429
1.428571428571429
1.428571428571429
1.428571428571429
```

Kuva 3.3: Esimerkkituloste ohjelmasta 3.2

Luku 4

Funktioiden evaluointi

4.1 Approksimointi

Kun yhteen-, vähennys-, kerto- ja jakolaskulle on kehitetty tehokkaat ja tarkat algoritmit, tulee kyseeseen näiden hyödyntäminen monimutkaisempien funktioiden approksimoinnissa. Approksimoinnin yleisenä ideana on, että approksimoiva funktio g täyttäisi täysin tai likimääräisesti joitakin approksimoitavaa funktiota f koskevia ehtoja. Näitä voivat olla mm. g:n yhtyminen f:ään tietyissä pisteissä, g:n derivaattojen yhtyminen f:n derivaattoihin tietyissä pisteissä tai integraalin $\int_\Omega |f-g|^p \, d\mu$ minimoiminen (ks. esim. [13]). Tässä käsitellään vain kahta ensinmainittua approksimointitapaa.

4.1.1 Polynomiapproksimointi

Tyypillisin ja monesti hyödyllisin polynomiapproksimaatio on Taylorin polynomit- ja sarjat. Tällöin funktion f, $n-1$-asteinen kehitelmä pisteessä z_0 on

$$f(z) = \sum_{k=0}^{n-1} \frac{f^{(k)}(z_0)}{k!}(z-z_0)^k \; + \; \underbrace{\frac{f^{(n)}(\xi)}{n!}(z-z_0)^n}_{=R_n} \tag{4.1}$$

missä $\xi \in [z_0, z]$. Taylorin polynomit ovat hyödyllisiä sellaisten funktioiden kohdalla, joilla ei ole napoja, oleellisia erikoispisteitä tai heikkoja singulariteetteja. Joissakin tapauksissa voi olla paikallaan hyödyntää funktion ominaisuuksia useammassa kuin yhdessä pisteessä, jos esimerkiksi tunnetaan $f(0)$, $f'(0)$, $f(1)$ ja $f'(1)$, voidaan hakea 3. asteen approksimaatio $f(z) \approx a_0 + a_1 z + a_2 z^2 + a_3 z^3$ ratkaisemalla yhtälöryhmä

$$\begin{cases} a_0 & = & f(0) \\ a_1 & = & f'(0) \\ a_0 + a_1 + a_2 + a_3 & = & f(1) \\ a_1 + 2a_2 + 3a_3 & = & f'(1) \end{cases} \tag{4.2}$$

4.1.2 Rationaaliapproksimointi

Rationaaliapproksimoinnin ideana on, että funktio $R(z) = P(z)/Q(z)$ täyttäisi e.m. ”hyviä” approksimoinnin ominaisuuksia. Merkittävimmässä asemas-

sa tässä suhteessa ovat Padé-approksimaatiot, jotka ovat tyyppiä $R_{m,n}(z) = P_m(z)/Q_n(z)$, missä P on m. asteen polynomi, Q n. asteen polynomi ja approksimaatio yhtyy halutussa pisteessä f:n $m+n$. derivaattaan asti. Padé-approksimaatioita voidaan muodostaa useilla tavoilla, mutta suoraviivaisin on laskea ensin $f(z)$:lle $m+n$. asteen Taylorin polynomi ja sitten ratkaista kertoimia Cauchy-tulon $Q_n(z)R_{m,n}(z) = P_m(z)$ virittämästä yhtälöryhmästä. Ks. kaavaa (3.2).

Esimerkki 4 *Olkoon approksimoitava funktio* $f(z) = 1 + r_1 z + r_2 z^2 + r_3 z^3 + r_4 z^4 + ... \approx R_{2,2}(z)$. *Tällöin saadaan* $P(z) = 1 + p_1 z + p_2 z^2$*:n ja* $Q(z) = 1 + q_1 z + q_2 z^2$*:n kertoimille yhtälöryhmä*

$$\begin{aligned} p_1 &= q_1 + r_1 \\ p_2 &= q_2 + r_1 q_1 + r_2 \\ 0 &= r_1 q_2 + r_2 q_1 + r_3 \\ 0 &= r_2 q_2 + r_3 q_1 + r_4 \end{aligned} \tag{4.3}$$

mikä on matriisimuodossa

$$\begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & -r_1 & -1 \\ 0 & 0 & -r_2 & -r_1 \\ 0 & 0 & -r_3 & -r_2 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ q_1 \\ q_2 \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix}. \tag{4.4}$$

Kaavan (4.4) perusteella voidaan yleisemminkin päätellä, että Padé-approksimaatioiden muodostamismatriisit ovat kohtalaisen harvaa ja helposti ratkeavaa muotoa

$$M = \left(\begin{array}{c|c} I & M_1 \\ \hline 0 & M_2 \end{array}\right) \tag{4.5}$$

missä I on identiteettimatriisi, 0 nollamatriisi, M_1 alakolmionauhamatriisi, "nauhoina" -1:t ja $-r_k$:t ja M_2 nauhamatriisi "nauhoina" $-r_k$:t. Käytännössä ainoastaan matriisiyhtälön

$$M_2\,\mathbf{q} = \mathbf{r}^{\uparrow} \tag{4.6}$$

[1] ratkaisemisella on numeerisesti enemmän merkitystä, koska muut kertoimet saadaan sen jälkeen sijoittamalla. Pienikokoisia Padé-approksimaatioita varten tähän riittää hyvin M_2:ien käänteismatriisien tallettaminen, mutta vähääkään isommissa tulee mieluummin kyseeseen yhtälöryhmän ratkaiseminen suoraan Gaussin algoritmilla, erityisiä Toeplitz-nauhastruktuureja [14] koskevilla algoritmeilla tai LU-hajotelmalla [15].

4.2 Neliöjuuri ja aritmeettis-harmoninen keskiarvoistaminen

Suosittu menetelmä $\sqrt{z}$:n laskemiseksi on lähteä liikkeelle yhtälöstä

$$z = x \cdot y \tag{4.7}$$

[1] Tässä $\mathbf{q} = (q_1\ q_2\ ...)^T$ ja $\mathbf{r}^{\uparrow}$ on vektorin $\mathbf{r} = (r_1\ r_2\ ...)^T$ tarpeeksi korkeat kertoimet, ylläolevassa $R_{2,2}$-tapauksessa $(r_3\ r_4)^T$.

ja pyrkiä siihen, että oikealla puolella olevan tulon tekijät olisivat mahdollisimman lähekkäin toisiaan. Tällöin voidaan tilannetta parantaa edellisestä lausumalla

$$z = \underbrace{\frac{x+y}{2}}_{=AM(x,y)} \cdot \underbrace{\frac{2xy}{x+y}}_{=HM(x,y)} . \tag{4.8}$$

Tässä esiteltyä lauseketta $AM(x,y)$ sanotaan *aritmeettiseksi keskiarvoksi* ja lauseketta $HM(x,y)$ *harmoniseksi* eli *hyperboliseksi keskiarvoksi.*

Lause 4 (Geometrinen keskiarvo) *Olkoon* $a_0 = 1$, $b_0 = z$, $a_{k+1} = \frac{a_k+b_k}{2}$ *ja* $b_{k+1} = \frac{a_k b_k}{a_{k+1}}$. *Tällöin*

$$\lim_{k\to\infty} a_k = \lim_{k\to\infty} b_k = \sqrt{z} = GM(a_k, b_k) \tag{4.9}$$

ja iteraatio suppenee kvadraattisesti. Jälkimmäistä lauseketta kutsutaan myös a_k*:n ja* b_k*:n geometriseksi keskiarvoksi* $GM(a_k, b_k) = \sqrt{a_k b_k}$.

Todistus 1 *Todistetaan vain kvadraattinen suppeneminen:* $AM(1+\epsilon, 1-\epsilon) = 1$, $HM(1+\epsilon, 1-\epsilon) = 1-\epsilon^2$ *eli* $GM(1+\epsilon, 1-\epsilon) = GM(1, 1-\epsilon^2)$. □

Aritmeettis-harmonisessa keskiarvoistamisessa aritmeettisten keskiarvojen jonon seuraaminen vastaa tavallista Newtonin menetelmää, mutta lisäetuna on, että harmoninen keskiarvo rajaa neliöjuuren sisältävää väliä toiselta puolen, jolloin on helpompi lopettaa halutun tarkkuuden saavutettua.

Todistus 2 *Ratkaistaessa yhtälöä* $f(z) = a$, *iteroidaan kaavan* $z_{k+1} = z_k - f(z_k)/f'(z_k)$ *mukaan. Yhtälölle* $w = \sqrt{z}$ *eli t.s.* $w^2 - z = 0$ *tämä tarkoittaa iteraatiokaavaa*

$$w_{k+1} = w_k - \frac{w_k^2 - z}{2w_k} = \frac{w_k^2 + z}{2w_k} = \frac{w_k + \left(\frac{z}{w_k}\right)}{2}. \ \square \tag{4.10}$$

4.3 Eksponenttifunktio ja trigonometriset funktiot

Eksponenttifunktion, sinin ja kosinin osalta Taylorin sarjakehitelmät ovat suositeltavampia kuin Padé-approksimaatiot, sillä näillä funktioilla ei ole napoja, oleellisia erikoispisteitä tai heikkoja singulariteetteja (paitsi ∞:ssä). Taylorin sarjat ovat niinikään pienillä arvoilla erityisen nopeasti suppenevia. Vaikka Padé-approksimaatio joissakin tapauksissa antaisi näille hivenen tarkemman tuloksen, on kuitenkin kyseisen approksimaation Taylorin sarjaan verrattuna monimutkaisempi muodostaminen sen verran suuri haitta, ettei haluttua hyötyä välttämättä saavutettaisi.

Kokeilun arvoinen voisi tästä huolimatta olla $R_{n,1}$-tyyppinen Padé-approksimaatio, koska siinä nimittäjä saadaan (ks. kaavaa (4.6)) yksinkertaisesti

$$Q(z) = 1 - \frac{r_{n+1}}{r_n} z \tag{4.11}$$

ja osoittaja P(z) edelleen (Cauchy-)tulolla $R(z)Q(z)$, missä $R(z)$:sta ei tarvitse huomioida ylintä, $n+1$-asteen kerrointa. Toisaalta kaikkialla kompleksitasossa analyyttisillä funktioilla n:n kasvaessa tämä kehitelmä degeneroituu tavanomaiseksi Taylorin sarjaksi (todistettavissa sarjaopin suhdetestin avulla), jolloin esim. $R(z)Q(z)$:n laskeminen olisi hukka-aikaa. Olisi mielenkiintoista tutkia, olisiko tämä järkevää mihinkään tiettyyn n:ään asti.

4.3.1 Eksponenttifunktio

Eksponenttifunktiolle on olemassa Taylorin kehitelmä origossa:

$$\exp(z) = \sum_{k=0}^{n-1} \frac{z^k}{k!} \; + \; \frac{\exp(\xi) z^n}{n!}, \; \xi \in [0, z] \tag{4.12}$$

Vähennyslaskuongelman 2.2.1 ja virhetermin muodon huomioiden, tätä sarjaa on syytä käyttää ainoastaan pienillä positiivisilla z:n arvoilla. Suurille negatiivisille z:n arvoille sarjan suorasta soveltamisesta olisi haittana alternoivuus ja väistämättä toisiaan lähellä olevien liukulukujen vähentäminen. Sen sijaan apuna käytetään yleisesti seuraavia kaavoja:

$$e^{-z} = \frac{1}{e^z} \tag{4.13}$$

ja

$$e^z = e^{\frac{z}{2}} e^{\frac{z}{2}} \tag{4.14}$$

Kaavalla (4.13) voidaan aina saada evaluoitavat termit pelkästään positiivisiksi, kaavalla (4.14) taas voidaan varmistaa nopea suppenevuus ja virhetermin pienuus. Potenssiin kaksi korottaminen kaksinkertaistaa suhteellisen virheen, joten se hävittää yhden merkitsevän bitin lopputuloksesta. Tämä tarkoittaa sitä, että numeeriset kirjastot tässä kohden usein *balansoivat* sarjan (4.12) termien lukumäärän ja kaavan (4.14) soveltamisten kanssa riippuen etukäteen päätetystä tarkkuusvaatimuksesta. `mpfr` [11] käyttää tätä menetelmää eksponenttifunktion evaluoimiseksi.

4.3.2 Kosini

Tarkastellaan tässä ainoastaan kosinia. Muut trigonometriset funktiot on nimittäin mahdollisuus laskea helposti kosinin avulla

$$\sin z \;=\; \sqrt{1 - \cos^2 z} = \cos\left(\frac{\pi}{2} - z\right) \tag{4.15}$$

$$\tan z \;=\; \frac{\sin z}{\cos z}. \tag{4.16}$$

Kosinifunktion $2n-2$-asteinen Taylorin kehitelmä origossa on

$$\cos z = \sum_{k=0}^{n-1} \frac{(-1)^k z^{2k}}{(2k)!} \; + \; O(z^{2n}) \tag{4.17}$$

[2] Taaskin on huomattava, ettei sarjaa 4.17 kannata soveltaa kuin hyvin pienillä kulman arvoilla eikä koskaan yli $\frac{\pi}{4}$:n (muunnoskaavat), joten kulmia kannattaa

[2] Tässä voidaan virhetermiä arvioida ottamalla yksinkertaisesti sarjasta seuraava termi.

puolittaa kosinin summakaavaa hyväksi käyttäen:

$$\cos z = 2\cos\left(\frac{z}{2}\right)\cos\left(\frac{z}{2}\right) - 1 \tag{4.18}$$

Kosinin evaluointi Taylorin sarjan ja kulmanpuolituksen yhdistelmällä on jälleenkin käytössä `mpfr`:ssä [11] kuten eksponenttifunktionkin tapauksessa. Taylorin sarjan evaluoinnissa ei ole käytössä Hornerin sääntö suoraan, mutta sen sijaan sarjan termejä voidaan kätevästi laskea kertomalla edellistä sopivalla kertoimella, s.o.

$$\begin{aligned} t_0 = s_0 &:= 1 \\ t_{k+1} &:= -\frac{z^2 t_k}{(2k+1)(2k+2)} \\ s_{k+1} &:= s_k + t_{k+1}. \end{aligned} \tag{4.19}$$

jolloin $s_k \to \cos(z)$ nopeasti kun z on pieni ja muuten asiaa autetaan jaksollisuusominaisuudet huomioiden sekä kaavalla (4.18).

Sama huomautus kuin eksponenttifunktion kohdalla koskien sarjan (4.17) termien lukumäärää ja kaavan (4.18) sovelluskertoja. Hyperbolisten funktioiden osalta voidaan käydä tälle analoginen tarkastelu, joka sivuutetaan.

4.4 Logaritmi ja π

4.4.1 Yleistä

Tässä käsitellään näiden osalta kaksi eri lähestymistapaa, pienen tarkkuuden luvuille ja suuren tarkkuuden luvuille, koska nämä lähestymistavat poikkeavat toisistaan huomattavasti.

Pienen tarkkuuden luvuille tulee kyseeseen edellisten tapaan verraten yleiset rationaaliapproksimaatioiden ja kulmanpuolitusten yhdistelmät. Tällä kertaa Padé-approksimaatiolla on myöskin enemmän merkitystä johtuen käsittelyssä olevien funktioiden oleellisista erikoispisteistä.

Suuren tarkkuuden luvuille taas käytetään aritmeettis-geometrista keskiarvoistamista ja nopeasti suppenevien sarjojen summia. Eräät sarjat soveltuvat mm. hyvin π:n tai logaritmien arvojen laskemiseen siten, että halutaan vain joitakin tiettyjä pitkällä kehitelmässä esiintyviä desimaaleja tai bittejä laskematta kaikkia aikaisempia.

4.4.2 Arkustangentti ja kulmanpuolitus

Logaritmin sijasta kannattaa usein tarkastella ennemminkin funktioita

$$\arctan(z) = z - \frac{z^3}{3} + \frac{z^5}{5} - \frac{z^7}{7} + \ldots\,, |z| < 1 \tag{4.20}$$

ja

$$\operatorname{arctanh}(z) = z + \frac{z^3}{3} + \frac{z^5}{5} + \frac{z^7}{7} + \ldots\ |z| < 1 \tag{4.21}$$

sillä nämä sarjat suppenevat nopeammin, niistä on helpompi muodostaa tarkkoja Padé-approksimaatioita ja niistä on helppoja muunnoskaavoja toisiinsa ja

logaritmifunktioon. Asetetaan erityisesti

$$\psi(z) = 1 + \frac{1}{3}z + \frac{1}{5}z^2 + \frac{1}{7}z^3 + \frac{1}{9}z^4 \approx \frac{\operatorname{arctanh}\sqrt{z}}{\sqrt{z}} \tag{4.22}$$

jolloin $\operatorname{arctanh}(z) \approx z\psi(z^2)$. Funktiolle ψ löytyy helposti Padé-approksimaatio kaavan (4.4) avulla, jolloin saadaan siis sievennysten ja manipulaatioiden jälkeen

$$\operatorname{arctanh}(z) \approx z\left(\frac{945 - 735z^2 + 64z^4}{945 - 1050z^2 + 225z^4}\right) \tag{4.23}$$

ja samaten kaavaa

$$\operatorname{arctanh}(z) = -\hat{\imath}\arctan(\hat{\imath}z) \tag{4.24}$$

sopivasti soveltaen:

$$\arctan(z) \approx z\left(\frac{945 + 735z^2 + 64z^4}{945 + 1050z^2 + 225z^4}\right). \tag{4.25}$$

Pätee myös

$$\ln(z) = 2\operatorname{arctanh}\left(\frac{z-1}{z+1}\right) \tag{4.26}$$

joten logaritmillekin saadaan hyvä approksimaatio. Kokeellisesti (Maple[2]) voidaan mm. havaita kaavojen (4.23) ja (4.26) olevan 8 desimaalia tarkka kun $z \in (0.6, 1.8)$. Kun vielä vastaavasti tehokas neliöjuurenottomenetelmä 4 on tunnettu, niin logaritmit voidaan tarvittaessa ottaa ainoastaan hyvin lähellä 1:tä sijaitsevalta alueelta (ja arkusfunktiot lähellä nollaa sijaitsevalta). Käsitellään tämä arkustangentin suhteen:

Kaksinkertaisen kulman tangentin kaava on

$$\tan(2\theta) = \frac{2\tan\theta}{1 - \tan^2\theta} \tag{4.27}$$

mistä voidaan johtaa 2. asteen yhtälön ratkaisulla puolikkaan kulman tangentin kaava

$$\tan\left(\frac{\theta}{2}\right) = \frac{\tan\theta}{\sqrt{1 + \tan^2\theta} + 1} \tag{4.28}$$

joka pätee (ainakin) kulmavälillä $\theta \in [0, \pi/4]$. Kirjoittamalla $\theta = \arctan z$, saadaan edelleen

$$\tan\left(\frac{\arctan z}{2}\right) = \frac{z}{\sqrt{1 + z^2} + 1} \tag{4.29}$$

ja edelleen

$$\arctan(z) = 2\arctan\left(\frac{z}{\sqrt{1 + z^2} + 1}\right). \tag{4.30}$$

Kaavoihin (4.25) ja (4.30) voidaan perustaa mm. kuvan 4.1 mukainen tehokas algoritmi π:n laskemiseksi, joskaan tämä ei enää nykyään ole yleisessä käytössä johtuen jäljempänä käsiteltävän aritmeettis-geometrisen keskiarvoistamisen paremmuudesta.

Vastaavasti pätee kaavan (4.24) nojalla

$$\operatorname{arctanh}(z) = 2\operatorname{arctanh}\left(\frac{z}{\sqrt{1 - z^2} + 1}\right) \tag{4.31}$$

```
#include <stdio.h>
#include <math.h>

double pade_arctan(double z) {
        double t1,t2,t3;
        t1=z*z;
        t2=945.0+t1*(735.0+64.0*t1);
        t3=945.0+t1*(1050.0+225.0*t1);
        return z*t2/t3;
}

double half_angle(double z) {
        return z/(sqrt(1.0+z*z)+1.0);
}

int main(int argc, char *argv[]) {
        int i,n;
        double tangent=1.0,value;
        printf("Montako kulmanpuolitusta? ");
        scanf("%d",&n);
        for(i=0;i<n;i++) {
                tangent=half_angle(tangent);
        }
        value=pade_arctan(tangent);
        for(i=0;i<(n+2);i++) {
                value+=value;
        }
        printf("Pii on noin %17.15lf\n",value);
        return 0;
}
```

Kuva 4.1: Tämä ohjelma laskee π:n arvoa sopivalla kulmanpuolituksen ja $R_{2,2}$-Padé-approksimaation yhdistelmällä. Jo 4 kierroksen valitseminen tuottaa 14 oikeaa desimaalia.

mikä on logaritmin laskemisen kannalta hyödyllisempi muoto.

Kulmanpuolituksella on tällaisessa mielessä merkitystä korkeintaan $\ln(2)$:n laskemisen kannalta, koska tätä ilmeisesti tarvitaan erikseen kaavassa (4.47). `mpfr` [11] käyttää tästäkin huolimatta ennemminkin sarjaa

$$\ln(2) = \frac{3}{4} \sum_{n=0}^{\infty} \frac{(-1)^n \, (n!)^2}{2^n \, (2n-1)!} \tag{4.32}$$

mutta on kyseenalaista, onko näin yksinkertainen menetelmä tässä paras mahdollinen.

Funktiot arcsin ja arccos saadaan vastaavasti lausuttua funktion arctan avulla seuraavasti:

$$\arcsin(z) \quad = \quad \arctan\left(\frac{z}{\sqrt{1-z^2}}\right) \tag{4.33}$$

$$\arccos(z) \quad = \quad \arctan\left(\frac{\sqrt{1-z^2}}{z}\right). \tag{4.34}$$

4.4.3 Machin-tyyppiset kaavat

Arkustangentille pätee vähennyslaskukaava

$$\arctan(z_1) - \arctan(z_2) = \arctan\left(\frac{z_1 - z_2}{1 + z_1 z_2}\right) \tag{4.35}$$

ks. esimerkiksi [19], s. 87. Tämä tarkoittaa sitä, että arkustangentti voidaan helposti hajauttaa useamman pienempiargumenttisen arkustangentin summaksi. Esimerkiksi (huomaa, että $\arctan(1) = \frac{\pi}{4}$)

$$\begin{aligned} \pi &= 4\arctan\left(\frac{1}{2}\right) + 4\arctan\left(\frac{1}{3}\right) \\ &= 8\arctan\left(\frac{1}{3}\right) + 4\arctan\left(\frac{1}{7}\right) \\ &= 12\arctan\left(\frac{1}{7}\right) + 8\arctan\left(\frac{2}{11}\right). \end{aligned} \tag{4.36}$$

Kaavassa (4.36) alimpana olevat sarjat ovat jo kohtuullisen nopeasti suppenevia.

Logaritmia varten on tietysti taas $\operatorname{arctanh}(z)$ olennaisen tärkeä (ks. kaavaa (4.26)). Tälle löytyy vieläkin paremmin kuin arkustangentille, hyviä kaavoja ja nopeasti suppenevien sarjakehitelmien muodostamiseen. Esimerkkeinä näistä säännöistä voisi mainita rationaaliargumenttien summan

$$\operatorname{arctanh}\left(\frac{p_1}{q_1}\right) + \operatorname{arctanh}\left(\frac{p_2}{q_2}\right) = \operatorname{arctanh}\left(\frac{p_1 q_2 + p_2 q_1}{p_1 p_2 + q_1 q_2}\right) \tag{4.37}$$

ja tähän sijoittamalla saatavan/todistettavan erään monista *dekompositiosäännöistä*

$$\operatorname{arctanh}\left(\frac{1}{k}\right) = \operatorname{arctanh}\left(\frac{1}{2k-1}\right) + \operatorname{arctanh}\left(\frac{1}{2k+1}\right). \tag{4.38}$$

Kaavan (4.38) eräänä sovelluksena voidaan saada vaikkapa seuraava kehitelmä $\ln(2)$:lle:

$$\begin{aligned} \ln(2) &= 2\operatorname{arctanh}\left(\frac{1}{3}\right) \\ \operatorname{arctanh}\left(\frac{1}{3}\right) &= \operatorname{arctanh}\left(\frac{1}{5}\right) + \operatorname{arctanh}\left(\frac{1}{7}\right) \\ \operatorname{arctanh}\left(\frac{1}{5}\right) - \operatorname{arctanh}\left(\frac{1}{7}\right) &= \operatorname{arctanh}\left(\frac{1}{17}\right) \\ \operatorname{arctanh}\left(\frac{1}{7}\right) &= \operatorname{arctanh}\left(\frac{1}{13}\right) + \operatorname{arctanh}\left(\frac{1}{15}\right) \\ \ln(2) &= 4\operatorname{arctanh}\left(\frac{1}{13}\right) + 4\operatorname{arctanh}\left(\frac{1}{15}\right) + \\ &\quad 2\operatorname{arctanh}\left(\frac{1}{17}\right). \end{aligned} \tag{4.39}$$

Bailey, Borwein ja Plouffe ovat artikkelissaan [20] vieneet näitä kaavoja askeleen pidemmälle ja löytäneet ns. *polylogaritmisten sarjojen teoriaa* tutkimalla sellaisia kehitelmiä π:lle, joista voidaan päätellä esim. tietyn kaukanakin kehitelmässä sijaitsevan bitin tai desimaalin arvo. Eräs heidän tärkeimmistä havainnoistaan oli, että

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right). \tag{4.40}$$

Kaava (4.40) soveltuu hyvin π:n heksadesimaalisen kehitelmän numeroiden laskemiseen (ja samalla tietysti myös binaarisen kehitelmän bitteihin).

Seuraava Dik T. Winter:in kirjoittama hieno ohjelma [16] laskee π:n 800-desimaalisen kehitelmän. Ohjelma alkuperäisessä muodossaan oli vain 160 merkin pituinen:

```
int a=10000,b,c=2800,d,e,f[2801],g;main(){for(;b-c;)f[b++]=a/5;
for(;d=0,g=c*2;c-=14,printf("%.4d",e+d/a),e=d%a)for(b=c;d+=f[b]*a,
f[b]=d%--g,d/=g--,--b;d*=b);}
```

mutta valitettavasti sikäli virheellinen, että se ajoittain tuotti vääriä tuloksia. Huomasin syyksi

a) `printf`-funktiota ei oltu esitelty

b) ainakin muuttujat `b` ja `e` olivat jääneet alustamatta (nolliksi). Ohjelma siis toimi satunnaisesti jos niiden varaamissa muistipaikoissa oli nollia, muuten ei.

Ohjelman korjattu versio, jossa on lisäksi tehokkuuden vuoksi siirretty globaalit muuttujat `main()`-funktion sisään (sijoitettavaksi rekistereihin), kuuluu seuraavasti:

```
#include <stdio.h>

main() {

int a=10000,b=0,c=2800,d=0,e=0,f[2801],g=0;
        for(;b-c;)
                f[b++]=a/5;
                for(;d=0,g=c*2;c-=14,printf("%.4d",e+d/a),e=d%a)
                        for(b=c;d+=f[b]*a,f[b]=d%--g,d/=g--,--b;d*=b);
}
```

4.4.4 Aritmeettis-geometrinen keskiarvo, π ja logaritmi

Lukujen x ja y välinen *aritmeettis-geometrinen keskiarvo* $\mathrm{AGM}(x, y)$ määritellään seuraavasti: sijoitetaan $a_0 = x$, $b_0 = y$ ja iteroidaan kaavan

$$a_{k+1} = \frac{a_k + b_k}{2}, \; b_{k+1} = \sqrt{a_k b_k} \tag{4.41}$$

mukaan. Tällöin on

$$\lim_{k\to\infty} a_k = \lim_{k\to\infty} b_k = \mathrm{AGM}(x, y) = \mathrm{AGM}(a_k, b_k). \tag{4.42}$$

Aritmeettis-geometrisella keskiarvolla on mm. seuraavanlainen yhteys elliptisten integraalien teoriaan:

$$AGM(x,y) = \frac{\pi}{2} \frac{1}{\int_{\theta=0}^{\pi/2} \left(x^2 \cos^2\theta + y^2 \sin^2\theta\right)^{-1/2} \, d\theta} \tag{4.43}$$

ks. esim. [17], §46 (the arithmetic-geometrico mean). Todistus, joka sivuutetaan tässä, perustuu siihen, että kaavassa (4.43) esiintyvä integraali on mahdollista muuntaa tavoilla, jossa x ja y korvautuvat sopivilla ag-keskiarvoistamista vastaavilla lausekkeilla (Landenin muunnos).

Käyttäen hyväksi mm. tätä ja elliptiset integraalit toisiinsa ja π:hin kytkevää Legendren lausetta [3] [17], §44, [19], luku 17, saadaan π:lle seuraavanlainen `mpfr`:ssä [11] käytössä oleva kaava:

$$\pi = \frac{AGM\left(1, \frac{1}{\sqrt{2}}\right)^2}{\frac{1}{4} - \sum_{k=1}^{\infty} 2^{k-1} \left(a_k^2 - b_k^2\right)} \tag{4.46}$$

missä a_k:t ja b_k:t ovat peräisin kaavasta (4.41). Logaritmille on vastaavasti olemassa jälleen `mpfr`:ssä [11] käytössä oleva kaava

$$\ln(x) \approx \frac{\pi}{2} \frac{1}{AGM\left(1, \frac{4}{s}\right)} - m \ln(2) \tag{4.47}$$

missä $s = x\, 2^m > 2^{p/2}$ ja p on haluttu bittitarkkuus. [4] Kaava (4.47) perustuu siihen, että elliptiset funktiot lähenevät asymptoottisesti trigonometrisia ja hyperbolisia funktioita tietyillä arvoilla ja vastaavasti elliptiset integraalit arkus- ja areafunktioita. Ks. esim. [19], luku 17, erityisesti § 17.3.26. [5]

---

[3]
$$E'K + EK' - KK' = \pi/2 \tag{4.44}$$
missä
$$\begin{aligned} K(k) &= \int_{\theta=0}^{\pi/2} \left(1 - k^2 \sin^2\theta\right)^{-1/2} \, d\theta \\ E(k) &= \int_{\theta=0}^{\pi/2} \left(1 - k^2 \sin^2\theta\right)^{1/2} \, d\theta \\ K' = K(k') &= K(1-k^2) \\ E' = E(k') &= E(1-k^2). \end{aligned} \tag{4.45}$$
Todistus perustuu tämän Legendren relaation (4.44) vasemman puolen derivoimiselle ja sen toteamisella, että kyseinen derivaatta on $= 0$.

[4] $\ln(2)$ on tietysti laskettava jotenkin. Aiemmin esiteltiin `mpfr`:n [11] sitä varten käyttämä sarjamenetelmä (4.32) ja kulmanpuolitus (4.31).

[5] Muunneltu versio tästä (modulin tilalla parametri k^2 ja ilmaistu A-S:n [19] mukaan logaritmimuodossa) kuuluu
$$\lim_{k \to 1} \left(K(k) - \frac{1}{2} \ln\left(\frac{16}{1-k^2}\right)\right) = 0. \tag{4.48}$$
Todistus perustuu $K(k)$:n ja $\ln\left(\frac{1}{1-k^2}\right)$:n Taylorin sarjakehitelmiin (sivuutetaan).

Luku 5

Lopuksi

5.1 Yhteenveto

Tämä erikoistyö käsittelee sinänsä kapeaa numeriikan alaa keskittyen vain alkeisfunktioiden evaluoimiseen (vähäisenä poikkeuksena muutamat viittaukset elliptisiin funktioihin ja integraaleihin). Tämä on kuitenkin siinä mielessä tärkeä osa-alue, että heikkoudet näiden funktioiden evaluointimenetelmissä tietokoneen prosessoritasolla tai standardi-C-kirjastoissa johtavat monien muidenkin sovellusten hitauteen tai epätarkkuuteen.

Asiaa johdatellaan ensin täysin perusteista - yhteen ja vähennyslaskun ilmaisemisesta loogisin operaatioin. Tämä on pohjana sille, miten nämä laskutoimitukset ovat helposti ja alkeellisesti evaluoitavissa mikroprosessoreissa. Tämän jälkeen esitellään lyhyesti liukuluvut tärkeänä reaalilukujen likimääräisen esittämisen tekniikkana. Polynomien evaluoimisesta Hornerin säännöllä on tehty kaksi tärkeää huomautusta, rekursion purkaminen vaihtamalla termien evaluointijärjestys, samoin kuin hajauttaminen rinnakkaislaskentaa varten.

Kertolaskun osalta on ensin esitelty alkeellinen pitkän kertolaskualgoritmin tekniikka, joka on kuitenkin erittäin suositeltava pienen tarkkuuden liukuluvuilla ja kokonaisluvuilla. Tämän jälkeen on esitelty Toom-Cook-algoritmeja ja todettu hyvin tunnettujen Karatsuban ja Fourierin algoritmien olevan tosiasiassa Toom-Cook-algoritmien erikoistapauksia. Tämä teoria on tärkeässä asemassa suuren tarkkuuden (esim. tuhansia bittejä) käsittävien lukujen kertolaskuissa. Eräänä keskeisenä sovellusalueena suuren tarkkuuden tuloille on mm. kryptografia. Jakolaskun osalta on olennaisessa asemassa nopean kertolaskun ja inverssiä koskevan geometrisen sarjan hyväksikäyttö.

Tämän jälkeen siirrytään rationaaliapproksimointiin ja esitellään tärkeä Padé-approksimaatiotekniikka erikoispisteitä käsittävien funktioiden evaluoimiseksi. Tällöin usein päästään kiertämään monia Taylorin sarjojen huonon suppenemisen tai hajaantumisen ongelmia.

Neliöjuurenotossa on ”standardi” Newtonin menetelmä käytännössä sivuutettu aritmeettis-harmonisen keskiarvoistamisen virhearvioita koskevien etujen vuoksi. Eksponenttifunktion osalta on lähdetty pääasiassa liikkeelle Taylorin sarjasta, mutta esitetty kaksi yleisessä käytössä olevaa parannusta koskien suppenemisen nopeuttamista ja liukulukujen vähennyslaskuongelman välttämistä. Samoilla linjoilla on esitelty trigonometrisistä funktioista esimerkkinä kosini.

Logaritmien ja arkusfunktioiden, samoin kuin π:n osalta on aluksi esitelty yksinkertaisuuden vuoksi kulmanpuolitukseen ja Padé-approksimaatioihin perustuva menetelmä. Tämä siksi, että näillä tekniikoilla päästään tavallisilla tarkkuusvaatimuksilla jo aivan tarpeeksi pitkälle. Machin-tyyppiset kaavat on esitelty lyhyesti ja tehty huomautus, että eräät niiden polylogaritmiset yleistykset tarjoavat mahdollisuuden logaritmien ja π:n haluttujen kehitelmänosien evaluoinnille. Lopuksi on esitelty luettelonomaisesti aritmeettis-geometrisen keskiarvoistamisen kaavat, koska nämä ovat nykyään suuren tarkkuuden liukulukusovelluksissa "standardimenetelmiä".

5.2 Tästä eteenpäin

Logaritmien ja π:n osalta oleva teoria on tilanpuutteen vuoksi jäänyt osittain pinnalliselle tarkastelulle. Tämä aihealue on sitäpaitsi nykyään runsaan tutkimuksen alla ja monet tämän aihepiirin edistysaskelista on otettu vasta 1900-luvun jälkipuoliskolla tai jopa 2000-luvun puolella. Erityisen merkillepantavaa on se, että monet mm. elliptisten funktioiden teoriaa koskevat kaavat ovat olleet tunnettuja 1700-luvulta asti, mutta niiden merkitys laskennan kannalta on ymmärretty paremmin vasta tietokoneiden kehityksen myötä. Tästä aihealueesta on myös paljon avoimena olevia kysymyksiä, mm. se, onko π:n ja logaritmien laskemiseksi yhtä tehokasta menetelmää kuin aritmeettis-geometrinen keskiarvoistaminen on, vähemmillä viittauksilla elliptisten funktioiden teoriaan. Samoin on olennainen kysymys, millä eri tavoin voidaan etukäteistietoa π:n tai tiettyjen logaritmien suuren tarkkuuden arvoista muiden logaritmien ja arkusfunktioiden arvojen laskemiseksi. Aritmeettis-geometrisen keskiarvoistamisen kaavastakin käy ilmi, kuinka on helpompi laskea π:n ja logaritmien välisiä suhteita kuin logaritmeja itsessään.

Kirjallisuutta

[1] The MathWorks: *MATLAB and SimuLink for Technical Computing*, `http://www.mathworks.com/`

[2] Maplesoft:
`http://www.maplesoft.com/`

[3] Yet another computer algebra system:
`http://yacas.sourceforge.net/`

[4] Linus Torvalds, Free Software Foundation etc.: *Linux Programmer's Manual* (Teos on luettavissa Linux:in manuaalisivujen muodossa, lisäksi siitä on olemassa eri levitysversioiden toimittajien julkaisemia toisistaan sisällöltään poikkeavia painoksia.)

[5] Kernighan, Ritchie: *The C programming language*
Prentice Hall, 1988, 1978

[6] GNU Compiler Collection:
`http://gcc.gnu.org/`

[7] Fortran: `http://www.fortran.com/`

[8] Advanced Micro Devices (AMD): *AMD64 Architecture Programmers Manual Volume 1: Application Programming*
`http://www.amd.com/us-en/Processors/TechnicalResources/`

[9] The ANSI/IEEE standard for floating point arithmetic (IEEE-754-1985),
`http://www.psc.edu/general/software/packages/ieee/ieee.html`

[10] GNU GMP
`http://www.gnu.org/software/gmp/`

[11] The MPFR library
`http://www.mpfr.org/`
`http://www.mpfr.org/algo.html`

[12] FFTW (Fastest Fourier Transformation in the West)
`http://www.fftw.org/`

[13] Timo Eirola, Harri Hakula: *Approksimaatioteorian kurssi ja monisteet*
http://math.tkk.fi/opetus/approx/

[14] Weisstein, Eric W. "Toeplitz Matrix."From MathWorld–A Wolfram Web Resource.
http://mathworld.wolfram.com/ToeplitzMatrix.html

[15] Weisstein, Eric W. "LU Decomposition."From MathWorld–A Wolfram Web Resource.
http://mathworld.wolfram.com/LUDecomposition.html

[16] How to compute digits of pi?
`http://www.cs.uwaterloo.ca/~alopez-o/math-faq/mathtext/node12.html`

[17] N. I. Akhietzer: *Elements of the Theory of Elliptic Functions*
American Mathematical Society

[18] Wikipedia: *Natural logarithm*
`http://en.wikipedia.org/wiki/Natural_logarithm`

[19] Abramowitz, Stegun: *Handbook of Mathematical Functions*
`http://www.math.sfu.ca/~cbm/aands/frameindex.htm`

[20] Bailey, Borwein, Plouffe: *On the rapid computation of various polylogarithmic constants*
`http://www.lacim.uqam.ca/~plouffe/articles/BaileyBorweinPlouffe.pdf`