Algorithms On Continued And Multicontinued Fractions

Franck Nielsen[1]
E.N.S. Lyon
France
fnielsen@ens.ens-lyon.fr

under the direction of
Peter Kornerup[2]
University of Odense
Denmark

August 12, 1993

[1]Visiting student from the E.N.S. Lyon.
[2]Professor of Computer Science, Odense University, Odense.

Acknowledgments:

I would like to thank **Søren Peter Johansen** and **Peter Kornerup** who have helped me during this work and have improved considerably the quality of this rapport.

That the topless towers be burnt
And men recall that face,
Move gently if move you must
In this lonely place.
She thinks, part woman, three parts a child,
That nobody looks her feet
Practice a tinker shuffle
Picked up on street.
Like a long-legged fly upon the stream
Her mind moves upon silence.

W.B. Yeats, "Longed-legged Fly"

Year - MCMXCIII

Keywords: Continued Fraction, Redundant representation, Lexicographic Continued Fraction, Arithmetic Unit, Fine grained Parallelism, On-line, Hypercube, **Gray** Code, Multicontinued Fraction.

Abstract

We introduce a continued fraction binary representation of the rationals, and several associated algorithms for computing certain functions on rationals exactly. This work follows the path taken by the previous works done in that area and extends the notion of continued fraction to multicontinued fraction. For that purpose, we introduce the notion of generalized matrix. This work is composed of six parts. We begin by a survey on the different kinds of representation of numbers. We then describe and algorithm which compute $\frac{ax+b}{cx+d}$ for any x when both inputs and outputs are provided piecewise. We generalize this algorithm to compute complex functions $f(x_1,\ldots,x_n) = \frac{P_1(x_1,\ldots,x_n)}{P_2(x_1,\ldots,x_n)}$ where $P_1(\cdot)$ and $P_2(\cdot)$ are polynomial functions of n variables. In the remaining, we show how it is possible to mix various formats of numbers (both for the input and output of numbers). Approximation of a real to a rational vector is investigated (**Szekeres**' algorithm)). A matrix representation of multicontinued fractions is introduced and we develop an algorithm based on that representation to compute quotient of polynomial functions.

Contents

1 Several Codings of Numbers and their Matrix Representations:

In this section, we present several ways of coding numbers. Some of them are *redundant* codings (i.e. a number can be coded in several admissible codes). Redundancy is sometimes used to bound time-computation when arithmetic operations are performed on these numbers. We can distinguish two kinds of reals: the rationals which can be coded by a finite coding and the irrationals that must be processed as symbolic information. The purpose of this article is to describe how an effective cell unit based on continued fraction format can be built to compute complex functions. So that, if an operand is irrational, it must be first approximated to a rational[1]. As computer hardwares deal with finite length string coding, both rational and irrational approximation are in reality done. For each code, we first give its definition, followed by some of its main properties we use.

1.1 Continued Fraction *CF*:

Continued Fraction

Let $[a_0/\dots/a_n]$ be a coding of $\frac{p}{q} \geq 0$. Then we have:

$$\frac{p}{q} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots + \cfrac{1}{a_n}}}}}$$

with the imposed conditions on the partial quotients a_i:

$$\begin{cases} a_0 \geq 0 \\ a_i \geq 1 \ \forall \ i \geq 1 \end{cases}$$

For instance, if we want to represent $\frac{5}{2}$, then its continued fractions are $[2/2]$ and $[2/1/1]$. Each rational is represented exactly without approximation . This is the main advantage of continued fraction compared with radix representation of numbers. Indeed, $\frac{1}{3}$ is simply coded by $[0/3]$ but in the radix representation, we can only have an approximation of $\frac{1}{3}$. The conditions on the partial quotient a_i eliminate some redundancy. Each rational $\frac{p}{q}$ has only two possible codings:$[a_0/\dots/a_n]$ and $[a_0/\dots, a_n - 1/1]$ (for further references, see [4][6][7][8][9][10][11][12]). Note that the partial quotient are obtained when performing **Euclid**'s algorithm. The complexity of this algorithm has been studied in *The Art Of Computing* **Knuth**, Vol. 2.

Given a continued fraction, we can compute its rational representation by using the following property:

Property 1 *Let $a = [a_0/\dots/a_n]$ be a continued fraction. We term tail of a ($T(a)$), the continued fraction $[a_1, \dots, a_n]$. Then we have:*

$$a = a_0 + \frac{1}{T(a)} = a_0 + \frac{1}{[a_1/\dots/a_n]}$$

Only a_0 may be equal to 0 since $[a_0/\dots/a_i/0/a_{i+2}/\dots/a_n] = [a_0/\dots/a_i + a_{i+2}/\dots/a_n]$.

The complexity of computing $\frac{p}{q}$ is proportional to the length of its coding (*forward recursion*).

Let us run the algorithm on $\frac{23}{12} = [1/1/11]$:

i	a_i	x
0	1	$\frac{12}{11}$
1	1	$\frac{11}{1}$
2	11	STOP

[1] We describe in the part dealing with the **Szekeres**' algorithm how this approximation can be computed.

ALGORITHM -I – CONTINUED FRACTION REPRESENTATION

INPUT:
$x = \frac{p}{q}$ (it is not required that $\gcd(p, q) = 1$)

OUTPUT:
$[a_0, \ldots, a_n]$ such that $\frac{p}{q} = [a_0, \ldots, a_n]$.
ALGORITHM:

$i = 0$;
repeat
  $a_i = \lfloor x \rfloor$;
  if $(x \neq a_i)$ **then** $x := \frac{1}{x - a_i}$;
  $i := i + 1$;
until $(x = a_{i-1})$;

Given a continued fraction coding $[a_0/\ldots/a_n]$, we want to compute its rational value $\frac{p}{q}$, $\gcd(p, q) = 1$. For that purpose, we use the property 1. This algorithm uses three registers but can only compute the $\frac{p}{q}$ (using property 5, we can compute all the convergents with four registers). Since $[a_i/\ldots/a_n] = a_i + \frac{1}{[a_{i+1}/\ldots/a_n]}$, if the rational denoted by $[a_{i+1}/\ldots/a_n] = \frac{p'}{q'}$ has been computed, we can compute $[a_i/\ldots/a_n] = \frac{p' * a_i + q'}{p'} = \frac{p''}{q''}$.

The steps generated by the algorithm when computing the value of $[1/1/11]$ are:

i	a_i	p	q
2	11	11	1
1	1	12	11
0	0	23	12

If $[a_0/\ldots/a_n]$ is a continued fraction representing $\frac{p}{q}$, we term $i - th$ convergent $(0 \leq i \leq n)$ the continued fraction $[a_0/\ldots/a_i] = \frac{p_i}{q_i}$.

Property 2 *The sequence* $(\frac{p_{2i}}{q_{2i}})_i$ *of even convergents is increasing and satisfies* $\frac{p_{2i}}{q_{2i}} \leq \frac{p}{q}$.

Property 3 *The sequence* $(\frac{p_{2i+1}}{q_{2i+1}})_i$ *of odd convergents is decreasing and satifies* $\frac{p_{2i+1}}{q_{2i+1}} \geq \frac{p}{q}$.

Property 4 *If* $[a_0/\ldots/a_n]$ *is a continued fraction representing* $\frac{p}{q}$ *with* $a_0 \neq 0$ *then* $[0/a_0/\ldots/a_n]$ *is a continued fraction denoted* $\frac{q}{p}$. *It follows, that if* $a_0 = 0$ *then* $\frac{q}{p} = [a_1/\ldots/a_n]$.

n-th convergent	Continued Fraction Expansion	Rational	Decimal Rep.
0	[1]	$\frac{1}{1}$	1
1	[1/2]	$\frac{3}{2}$	1.5
2	[1/2/3]	$\frac{10}{7}$	1.428571429...
3	[1/2/3/4]	$\frac{43}{30}$	1.433333333...
4	[1/2/3/4/5]	$\frac{225}{157}$	1.433121019...

Convergents of $\frac{225}{157} = [1/2/3/4/5]$.

We define the relational symbol $\leqslant$ on the rational as follows: $\frac{p}{q} \leqslant \frac{r}{s}$ iff $p < r$ and $r < s$.

The convergents are often used when analysing the performance of an algorithm on continued fractions. We cite below the main properties:

ALGORITHM -II – RATIONAL NUMBER REPRESENTATION

INPUT:
$[a_0/\ldots/a_n]$ a coding of a rational number.

OUTPUT:
$\frac{p}{q} = [a_0, \ldots, a_n]$ with $\gcd(p, q) = 1$

ALGORITHM:

$p := a_n$;
$q := 1$;
$i := n - 1$;
while$(i \geq 0)$ **do**
    begin
    $tmp := q + a_i * p$;
    $q := p$;
    $p := tmp$;
    $i := i - 1$;
    end

Property 5 *The convergents $\frac{p_i}{q_i} = [a_0/\ldots/a_i]$ of any continued fraction $\frac{p}{q} = [a_0/\ldots/a_n]$ satisfy the following properties:*

- *Recursive ancestry: With $p_{-2} = 0, p_{-1} = 1, q_{-2} = 1, q_{-1} = 0$, we have $p_i = a_i p_{i-1} + p_{i-2}$ and $q_i = a_i q_{i-1} + q_{i-2}$.*
- *Irreductibility: $\gcd(p_i, q_i) = 1$.*
- *Adjacency: $q_i p_{i-1} - p_i q_{i-1} = (-1)^i$.*
- *Simplicity: $\frac{p_i}{q_i} \leqslant \frac{p_{i+1}}{q_{i+1}}$ for $i \leq n-1$.*
- *Alternating convergence:*
$$\frac{p_0}{q_0} < \frac{p_2}{q_2} < \ldots < \frac{p_{2i}}{q_{2i}} < \ldots \leq \frac{p}{q} \leq \ldots < \frac{p_{2i-1}}{q_{2i-1}} < \ldots < \frac{p_1}{q_1}$$
- *Best rational approximation:*
$$\frac{r}{s} \leqslant \frac{p_i}{q_i} \Rightarrow \left|\frac{r}{s} - \frac{p}{q}\right| > \left|\frac{p_i}{q_i} - \frac{p}{q}\right|$$
- *Quadratic Convergence:*
$$\frac{1}{q_i(q_{i+1} + q_i)} < \left|\frac{p_i}{q_i} - \frac{p}{q}\right| \leq \frac{1}{q_i q_{i+1}} \text{ for } i \leq n-1$$
- *Real approximation: $|x - \frac{p}{q}| < \frac{1}{2q^2}$ for irreductible $\frac{p}{q} \Rightarrow \frac{p}{q}$ is a convergent of a continued fraction expansion of x.*

If $[a_0/\ldots/a_n]$ is the coding of $|\frac{p}{q}|$, then if $\frac{p}{q} \leq 0$, its continued fraction is denoted by $-[a_0/\ldots/a_n] = [-a_0/\ldots/-a_n]$.

1.2 Redundant Continued Fraction:

Redundant Continued Fraction

Redundancy is often used in arithmetic algorithms([10][16]) where it allows to produce result more quickly (when the result is coded by a continued fraction, redundancy allows to deliver a partial quotient earlier).

$$\frac{p}{q} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots + \cfrac{1}{a_n}}}}}$$

with the following conditions

$$\begin{cases} a_0 \geq 0 \\ |a_i| = 1 \text{ , } 1 \leq i \leq n-1 \text{ implies that } a_i \text{ and } a_{i+1} \text{ have the same sign} \\ |a_n| \geq 2 \text{ whenever } k \geq 2 \end{cases}$$

The set of all redundant continued fraction expansions of $\frac{11}{4}$ is then

$$\frac{11}{4} = \begin{cases} [2/1/3] \\ [2/2/\overline{1}/\overline{2}] \\ [2/2/\overline{2}/2] \\ [3/\overline{4}] \end{cases}$$

The algorithm to determine all the redundant continued fraction representation of $\frac{p}{q}$ is more complex since it uses a recursive process which must take care of the initial 2 steps (a_0 and a_1). Let us consider the rational $\frac{p}{q}$. If we choose the partial quotient a, then the remaining rational is $\frac{q}{p-aq}$. Using the fact that $|\frac{q}{p-aq}| \geq 1$ or $|\frac{p-aq}{q}| \leq 1$, it follows:

$$a \text{ must satisfy the range constraint } \begin{cases} a \leq \frac{p+q}{q} = \frac{p}{q} + 1 \\ a \geq \frac{p-q}{q} = \frac{p}{q} - 1 \\ a \in \mathbb{N} \end{cases} \tag{1}$$

Then, we iterate the process until the current partial quotient equals $\frac{p}{q}$ (in that case $\frac{p}{q} \in \mathbb{N}$).

To have all the codes possible given a rational, the conditions on redundancy must be taken into account. If 1 or -1 is chosen, the next partial quotient must have the same sign. If $n \geq 2$, we must ensure that the last partial quotient a_n satisfies $a_n \geq 2$.

In order to simplify the algorithm, we use a queue $\mathcal{Q}$ in which a value can be added (appended) in its queue by the operator $\circ$.

In the algorithm, we use a boolean function to assert that the partial quotient is in the proper range: for example if $COND$ is the boolean function, $COND = COND(.) = COND(a) = (|a| \geq 2)$ specifies that the partial quotient must have its absolute value greater or equal than 2.

Running the algorithm on the rational $\frac{11}{4}$, we obtain the following step:

<table>
<tr><td>$a_0[2;3]$</td><td>$\frac{p'}{q'}$</td><td colspan="5">Recursive step</td></tr>
<tr><td rowspan="5">2</td><td rowspan="5">$\frac{4}{3}$</td><td>$a_1[1;2]$</td><td>$\frac{p''}{q''}$</td><td colspan="3">Recursive step</td></tr>
<tr><td>1</td><td>$\frac{3}{1} \rightarrow a_2 = 3 \rightarrow$ STOP</td><td colspan="3"></td></tr>
<tr><td rowspan="3">2</td><td rowspan="3">$-\frac{3}{2}$</td><td>$a_2[-2;-1]$</td><td>$\frac{p'''}{q'''}$</td><td>Recursive step</td></tr>
<tr><td>-1</td><td>$-\frac{2}{1}$</td><td>$\rightarrow a_3 = -2 \rightarrow$ STOP</td></tr>
<tr><td>-2</td><td>$\frac{2}{1}$</td><td>$\rightarrow a_3 = 2 \rightarrow$ STOP</td></tr>
<tr><td>3</td><td>$-\frac{4}{1}$</td><td colspan="5">$\rightarrow a_1 = -4 \rightarrow STOP$</td></tr>
</table>

ALGORITHM -III – Redundant Continued Fraction

INPUT:
A rational $\frac{p}{q}$, $\gcd(p,q)=1$.
Initial Call: GiveRedundantCode(p,q,TRUE,$\mathcal{Q} \leftarrow$ null).
OUTPUT:
All the possible redundant codings of $\frac{p}{q}$.
ALGORITHM:

Procedure GiveRedundantCode(p,q,COND,n,$\mathcal{Q}$);
p,q,n: **integer**;
COND: Boolean Function on the next partial quotient a that must be computed ;
$\mathcal{Q}$: represents the Queue where the partial quotients are stocked;

begin
$min := \lceil \frac{p-q}{q} \rceil$;
$max := \lfloor \frac{q+p}{q} \rfloor$;
for $i := min$ **to** max **do**
  begin
  if (COND(i)) **then do**
    begin
    if $(n \geq 1)$ **then** NEWCOND:=$(a \geq 2)$ **else**
    NEWCOND:=$a \neq 0$;
    if $(|i| = 1)$ **then** NEWCOND:=NEWCOND$\bigwedge(\frac{|a|}{a} = i)$;
    if $(\frac{p}{q} = i)$ **then** $\mathcal{Q} \leftarrow \mathcal{Q} \circ i$;
    else GiveRedundantCode(q,p-iq,NEWCOND,n+1,$\mathcal{Q} \circ i$);
    end
  end
end

1.3 Continued Logarithmic Fraction:

Continued Logarithmic Fraction

This way of coding rationals allows easy-coding-decoding binary digits. We now consider the following unique coding of $[a_0, \dots, a_n]_{\mathrm{CL}}$:

$$\frac{p}{q} = 2^{a_0} + \cfrac{2^{a_0}}{2^{a_1} + \cfrac{2^{a_1}}{2^{a_2} + \cfrac{2^{a_2}}{2^{a_3} + \cfrac{2^{a_3}}{\ddots + \cfrac{2^{a_{n-1}}}{2^{a_n}}}}}}$$

with the condition on the logarithmic partial quotients

$$\begin{cases} a_0 \geq 0 \\ a_i \geq 1 \ \forall\ i \geq 1 \end{cases}$$

By factorizing at each level the partial quotient (if we suppose $\frac{p}{q} \geq 1$, otherwise $\frac{p}{q} = [0] \circ [\text{coding of } \frac{q}{p}]$), we find

$$2^{a_0} + \cfrac{1}{2^{a_1-a_0} + \cfrac{1}{2^{a_2-a_1+a_0} + \cfrac{1}{2^{a_3-a_2+a_1-a_0} + \cfrac{1}{\ddots + \cfrac{1}{2^{a_n-a_{n-1}+a_{n+2}-\dots+(-1)^n a_0}}}}}} =$$

$$[2^{a_0}, 2^{a_1-a_0}, \dots, 2^{a_n-a_{n-1}+a_{n+2}-\dots+(-1)^n a_0}]$$

which is a continued fraction. We term it continued logarithmic fraction because of the fact that it can be ciphered just by taking the logarithm in base 2 at each step (the remaining number is $\frac{q}{p-2^{a_i}q}$). This factorization was introduced in an unpublished paper of **Gosper** in 1977.

For instance, if ALG.4 is applied to $\frac{28}{11}$, we have the following steps:

step (i)	$\frac{p}{q}$	a'_i	a_i	sum
0	$\frac{28}{11}$	$1 = a_0$	1	1
1	$\frac{11}{6}$	$0 = a_1 - a_0$	1	0
2	$\frac{6}{5}$	$0 = a_2 - a_1 + a_0$	0	0
3	$\frac{5}{1}$	$2 = a_3 - a_2 + a_1 - a_0$	2	2
4	$\frac{1}{1}$	$0 = a_4 - a_3 + a_2 - a_1 + a_0$	0	0

So finally, it comes that $\frac{28}{11} = [2/1/1/4/1] = [2/1/1/5] = [1/1/0/2/2]_{\mathrm{CL}}$.

Since the partial quotients of that coding are generally small[2], we can code p, a partial quotient, as $l(p) = 1^p0$ (a (p)-string of 1 followed by a 0).

Hence, we have $l(0) = 0$, $l(1) = 10$, $l(2) = 110$, , $l(3) = 1110$, ... With that binary sized coding, $CL(\frac{28}{11}) = 10 \circ 10 \circ 0 \circ 110 \circ 110$.

This simple algorithm can be implemented in hardware with a simple look-up table (this look-up table is described by a small number of states. For a complete description of that table, see [16]).

[2] From classical material on continued fractions it is known that the partial quotients in the continued fraction expansion of a randomly chosen $\frac{p}{q} \in [0, 1]$ ([13]) will have the value i with the probability essentially given by $p_i = \log_2 (1 + \frac{1}{i(i+2)})$.

ALGORITHM -IV – CONTINUED LOGARITHMIC FRACTION

INPUT: $\frac{p}{q}$, $\gcd(p, q) = 1$
OUTPUT: $[a_0/\ldots/a_n]_{\text{CL}} = [a'_0/\ldots/a'_n]$
ALGORITHM:

```
i := 0;
sum := 0;
if (p < q) then
            begin
            a_0 := 0;
            i := 1;
            tmp := p;
            p := q;
            q := tmp;
            end;
while (q ≠ 0) do
            begin
            a'_i := ⌊log_2 p/q⌋;
            a_i := a'_i + sum;
            tmp := q;
            q := p − 2^{a'_i} q;
            p := tmp;
            sum := a'_i;
            i := i + 1;
            end
```

1.4 LCF - Lexicographic Continued Fraction:

Lexicographic
Continued
Fraction

This way of representing rational numbers has been introduced by **Matula** and **Kornerup** in 1985([3]). We say a code follows the lexicographic order iff given two strings $s(a)$, $s(b)$ of binary digits denoting the rationals a and b:

$$\begin{cases} a < b \Leftrightarrow s(a) < s(b) \\ a = b \Leftrightarrow s(a) = s(b) \\ a > b \Leftrightarrow s(a) > s(b) \end{cases}$$

Note that the relational symbols $<, =, >$ used when comparing a with b, and the relational symbols $<, =, >$ used with $s(.)$ are not the same. If $p = 2^n + \sum_{i=0}^{n_1} b_i 2^i$ is an integral number, then we define $l(p) = 1^n 0 b_{n-1} \dots b_0$ as its lexicographic coding.

For instance, we have $l(1) = 0,\ l(2) = 100,\ l(3) = 101,\ l(4) = 11000, \dots$.

In general case, when we want to code $\frac{p}{q}$, we first code the sign, then the partial quotients $[a_0/\dots/a_n]$. Taking into account the fact that the even convergents are smaller than the odd ones, and the sign of the rational (Signed LCF), it comes:

$$SLCF([a_0/a_1/\dots/a_n]) = \begin{cases} 1 \circ \overline{LCF([a_0/\dots/a_n])} \text{ for } [a_0/\dots/a_n] \geq 0. \\ 0 \circ \overline{LCF(|[a_0/\dots/a_n]|)} \text{ otherwise} \end{cases}$$

where,

$$LCF([a_0/a_1/\dots/a_{2n}]) = \begin{cases} 1 \circ l(a_0) \circ \overline{l(a_1)} \circ \dots \circ l(a_{2n}) \text{ for } [a_0/\dots/a_n] \geq 1. \\ 0 \circ \overline{l(a_0)} \circ l(a_1) \circ \dots \circ \overline{l(a_{2n})} \text{ otherwise} \end{cases}$$

$$LCF([a_0/a_1/\dots/a_{2n}/a_{2n+1}]) = \begin{cases} LCF([a_0/a_1/\dots/a_{2n}]) \circ \overline{l(a_{2n+1})} \text{ if } [a_0/a_1/\dots/a_{2n}/a_{2n+1}] \geq 1 \\ LCF([a_0/a_1/\dots/a_{2n}]) \circ l(a_{2n+1}) \text{ otherwise} \end{cases}$$

Worst case representation-induced precision loss for any real number by a fixed length representable number of the system has been shown to be at most 19% of bit word length, with no precision loss whatsoever induced in the representation of any reasonably sized rational number (a complete description of the proof can be found in [3]).

Using the probability of $a_i = j$ $(p_j = \log_2(1 + \frac{1}{j(j+2)}))$, it follows that:

i	p_i
1	0.415
2	0.170
3	0.093
4	0.059

An average partial quotient will be coded, therefore, by n bits where

$$n = \sum_i (2\lfloor \log_2 i \rfloor + 1) \log_2 (1 + \frac{1}{i(i+2)}) \simeq 3.51...$$

This yields straigthforwardly the observation([3]):

Observation 1 *From the known distribution of partial quotient size, it follows that the canonical continued fraction expansion of LCF expansion of a rational* $\frac{p}{q} = [a_0/\dots/a_n] = b_0 b_1 b_2 \dots b_{k-1} 1$ *yields an expected convergent to convergent ratio of* $e^{\frac{k}{n}} = 3.51...$

We conclude the description of LCF by a property on the gap sizes between two adjacents LCF representation coded by a k-string:

Property 6 *Given* $\epsilon > 0$, *then for sufficiently large* k, *the maximum gap size will be at least* $2^{-(a+\epsilon)k}$ *for* $a = \frac{1}{4}\log_2(5 + 2\sqrt{6}) = 0.82682...$ *and the minimum gap size will be no bigger than* $2^{(-b-\epsilon)k}$ *for* $b = \log_2 \frac{3+\sqrt{5}}{2} = 1.38848....$

A description of the proof can be found in [3].

1.5 Matrix Representation of Continued Fraction Expansion:

We have seen that the general principle of continued fraction was a suite of partial quotients $a_i, 0 \leq i \leq n$. If we take the convention of writing $\frac{p}{q} \equiv (p\ q)$, then we can write $\frac{p}{q} = [a_0/\dots/a_n]$ as a product of easy-and-inversible matrices as follows:

If n is odd:

$$\frac{p}{q} \equiv (p\ q) = (x_i\ 1) = (1\ 0) \times \begin{pmatrix} 1 & a_n \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ a_{n-1} & 1 \end{pmatrix}\dots\begin{pmatrix} 1 & 0 \\ a_{2j} & 0 \end{pmatrix}\begin{pmatrix} 1 & a_{2j-1} \\ 0 & 1 \end{pmatrix}\dots\begin{pmatrix} 1 & 0 \\ a_0 & 1 \end{pmatrix}$$

If n is even:

$$\frac{p}{q} \equiv (p\ q) = (x_i\ 1) = (0\ 1) \times \begin{pmatrix} 1 & 0 \\ a_n & 1 \end{pmatrix}\begin{pmatrix} 1 & a_{n-1} \\ 0 & 1 \end{pmatrix}\dots\begin{pmatrix} 1 & a_{2j+1} \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ a_{2j} & 0 \end{pmatrix}\dots\begin{pmatrix} 1 & 0 \\ a_0 & 1 \end{pmatrix}$$

Let p be a partial quotient, $p = 2^n + \sum_{i=0}^{n-1} 2^i b_i$, then we have:

$$\begin{pmatrix} 1 & p \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & \frac{b_0}{2} \\ 0 & \frac{1}{2} \end{pmatrix}\dots\begin{pmatrix} 1 & \frac{b_{n-1}}{2} \\ 0 & \frac{1}{2} \end{pmatrix} \times \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}^n$$

$$\begin{pmatrix} 1 & 0 \\ p & 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & 0 \\ \frac{b_0}{2} & 1 \end{pmatrix}\dots\begin{pmatrix} \frac{1}{2} & 0 \\ \frac{b_{n-1}}{2} & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}^n$$

Since the length of the coding is not fixed, we must specify the length by a leading symbol: "u". Hence p is denoted by $\underbrace{u\dots u}_{n \text{ times}} b_{n-1}b_{n-2}\dots b_0 = u^n b_{n-1}b_{n-2}\dots b_0$. While p is entered piecewise thanks to its radix representation of its partial quotient, simple product of matrices are performed.

Since the coefficient of the factorization belongs to $\{0, \frac{1}{2}, 1, 2\}$, performing a product of matrices with these generating matrices correspond to left/right-shift and sum. The input process must keep track of the last bit entered, so it can perform the *switch* matrix when the last u is read.

Regarding the next sections in the remaining of this article, this factorization is useful if we consider a cell that can perform arithmetic on binary continued fraction of variable length and it is required that its operands must be entered piecewise.

For example, if $\begin{pmatrix} 1 & 0 \\ p & 1 \end{pmatrix}$ with $p = 2^n + \sum_{i=0}^{n-1} a_i b_i$: the cell will receive the input $\underbrace{u\dots u}_{n \text{ times}} b_{n-1}\dots b_0$

$$\underbrace{\left\{\begin{pmatrix} \frac{1}{2} & 0 \\ b_0 & 1 \end{pmatrix} \times \dots \times \begin{pmatrix} \frac{1}{2} & 0 \\ b_{n-1} & 1 \end{pmatrix}\right\}}_{b_0 b_1 \dots b_{n-1}} \overbrace{\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}}^{\text{switch}} \times \underbrace{\left\{\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} \times \dots \times \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}\right\}}_{n\ u \text{ entered}} \times \boxed{\text{CELL UNIT}}$$

input	even matrix	odd matrix
u	$\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$
switch	$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$
$b_i = 0$	$\begin{pmatrix} \frac{1}{2} & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & \frac{1}{2} \end{pmatrix}$
$b_i = 1$	$\begin{pmatrix} \frac{1}{2} & 0 \\ \frac{1}{2} & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & \frac{1}{2} \\ 0 & \frac{1}{2} \end{pmatrix}$

A rational number can therefore afeter be denoted by a 2-vector ($(p\ q) \equiv \frac{p}{q}$) be written as a product of simple inversible matrices.

Note: We can avoid the even-odd matrix problem by using matrices of the following form:

$$\begin{pmatrix} 0 & 1 \\ 1 & a_i \end{pmatrix}$$

since

$$\begin{pmatrix} 1 & 0 \\ a_{2i+1} & 1 \end{pmatrix} \times \begin{pmatrix} 1 & a_{2i} \\ 0 & 1 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 \\ a_{2i+1} & 1 \end{pmatrix} \times \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}}_{\begin{pmatrix} 0 & 1 \\ 1 & a_{2i+1} \end{pmatrix}} \times \underbrace{\begin{pmatrix} 1 & 0 \\ a_{2i} & 1 \end{pmatrix} \times \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}}_{\begin{pmatrix} 0 & 1 \\ 1 & a_{2i} \end{pmatrix}}$$

In the last section, we define a multicontinued fraction by means of matrices. The switch matrix is no more used and an homogene writing[3] is used.

1.6 Radix Coding:

Radix

We consider, now as it is often used in pratice, a rational $\frac{p}{q}$ in a radix representation ($\frac{3}{4} = 0.75 = (0.11)_2$). A number x has its integral part $\lfloor x \rfloor$ and its fractional part $x - \lfloor x \rfloor$. Both parts can be coded in radix representation. A number $p = \sum_{i=-k}^{n} 2^i b_i = 2^n b_n + 2^{n_1} b_{n-1} + \ldots + 2^{-k+1} b_{-k+1} + 2^{-k} b_{-k}$ is coded as $u^{n-1} b_n b_{b_1} \ldots b_1 b_0 b_{-1} \ldots b_{-k}$ with the alphabet $\mathcal{B} = \{u, 0, 1\}$ where u is a leader code allowing to count the length of the integral part.

For instance, $7.375 = (111.011)_2 \equiv uu111011$.

We can also use, more elaborate form of coding based on radix representation ([11][13][16]), like the two's complement fixed-point numbers (C-2) ,...

1.7 Redundant Binary Representation (RPQ):

Redundant
Radix

In the algorithm developed later, redundancy will allow us to bound delays between inputs and outputs. Outputs will be produced faster, and the velocity of streams of data in the pipeline computational tree will be higher. LCF provides the bit-grained the structure without redundancy. If p is an integral number, then $[p]_2 = b_n \ldots b_0$ where $b_i \in \{\bar{1}, 0, 1\}$, is one of its coding. We constrain redundancy as follows:

$$R(p) = u^{n-1} b_n b_{n-1} \ldots b_0 \text{ with } |b_n| = 1$$

satisfying the range constraint:

$$2^{n-1} + 1 \leq p \leq 2^{n+1} - 1 \text{ for } n \geq 2$$

and for $p = 0$, we have $R(p) = b_0 = 0$.

Definition 1 *For $n \geq 2$, a self-delimiting signed bit string $u^{n-1} b_n b_{n-1} \ldots b_1 b_0$ is admissible iff when $b_n b_{n-1} = 1\bar{1}$ or $\bar{1}1$, the sign of $b_{n-2} b_{n-3} \ldots b_1 b_0$ agrees with that of b_n.*

For example, the string $(1\bar{1}\bar{1})_2 = 1$ is inadmissible but $(1\bar{1}1)_2 = 3$ is admissible.

The test must be easy to perform, since in practice, a stream of bits will enter piecewise the arithmetic cell.

For any redundant continued fraction, we obtain the following coding of $\frac{p}{q} = [a_0 / \ldots / a_n]$:

$$R(\frac{p}{q}) = R(a_0) \circ R(a_1) \circ \ldots \circ R(a_n)$$

In general, all the algorithms which deals with the enumeration of redundant representations of a number are recursive. In the redundant radix representation, the range constraint is defined by $2^{n-1}+1 \leq$

[3]The "switch matrix" of a j-multicontinued fraction is unique and defined as e_{j+1} where e_{j+1} is the $(j+1)$-th unit vector in base $\mathbb{Q}^{j+1}$

ALGORITHM -V – RADIX REPRESENTATION

INPUT:
A real number x.
OUTPUT:
Its radix representation in the list $\mathcal{Q}$.

Note that the operator $\circ$ is a concatenator. Hence $a \circ b \neq b \circ a$, $a \neq b$.

Procedure IntegerPart(x, $\mathcal{Q}$);
x:integer;
$\mathcal{Q}$: queue;

begin
length: **integer**;

length := 0;
while ($x \neq 0$) **do**
    begin
    $\mathcal{Q} \leftarrow (x \bmod 2) \circ \mathcal{Q}$;
    $x := x$ div 2;
    $length := length + 1$;
    end
$\mathcal{Q} \leftarrow u^{length} \circ \mathcal{Q}$
end

Procedure FractionalPart(x, $\mathcal{Q}$);
x:integer;
$\mathcal{Q}$: queue;

begin
if (x=0) **then** EXIT;
if ($x \geq \frac{1}{2}$) **then** FractionalPart(2x-1, $\mathcal{Q} \circ 1$);
    else FractionalPart(2x, $\mathcal{Q} \circ 0$);
end

begin
Empty($\mathcal{Q}$);
IntegerPart($\lfloor x \rfloor$, $\mathcal{Q}$);
$\mathcal{Q} \leftarrow \mathcal{Q} \circ$ ".";
FractionalPart($x - \lfloor x \rfloor$, $\mathcal{Q}$);
end

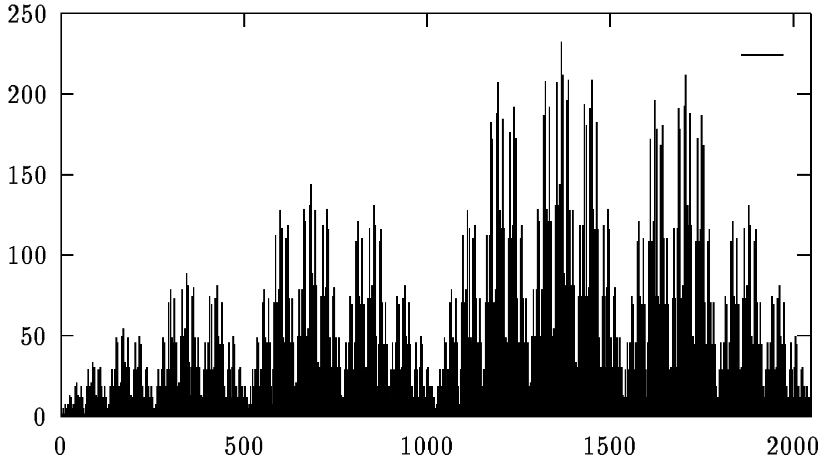


Figure 1: Number of redundant codings for integers $x \in [0, 2048]$.

$|p| \leq 2^{n+1} - 1$. Hence, we have $\lfloor \log_2(|p|+1) \rfloor - 1 \leq n \leq \lfloor \log_2(|p|-1) \rfloor + 1$, which depending on the value of $|p|$ gives 2 or 3 different choices for n.

Let's take an example: we'd like to have the representations of $10 = (1010)_2$. Computing the n-range gives: $2 \leq n \leq 4$. So, the different kind of format are:

$$\begin{cases} b_3 b_2 b_1 b_0 \\ b_4 b_3 b_2 b_1 b_0 \\ b_5 b_4 b_3 b_2 b_1 b_0 \end{cases}$$

Let's take, for instance, the format $b_4 b_3 b_2 b_1 b_0$. Then, since 10 is positive, we have $b_4 = 1$ but $2^4 = 16$, so we have to code -6 in redundant radix representation with at most format of length $3 : b_3 b_2 b_1 b_0$ since $b_4 = 1$. The last point shows the recursive process.

Proceeding in that way, we find the 8 different codings for $x = 10$:

The output displayed below is porduced by the **MulCF-function** program described in annex. The alphabet denoting the results is $\{[1], 0, 1\}$ where $[1]$ denote $\overline{1}$.

```
>1[1][1]1[1]0
>1[1][1]010
>1[1]0[1][1]0
>01[1]1[1]0
>01[1]010
>010[1][1]0
>0011[1]0
>001010
```

Note: Redundancy can be present both in the partial quotients (see for example RPQ) and in the binary representation of these partial quotients.

2 Computation of Functions of One Variable:

2.1 Notations and Introduction to the Problem:

To understand the underlying principles of the general algorithm, we begin by a presentation of the general concept. We want to compute the following function $f(x) = \frac{ax+c}{bx+d}$ where x is a variable (x_1). The

ALGORITHM -VI – REDUNDANT BINARY REPRESENTATION

INPUT:
An integer x: call: RedundantNumber(x,"",MAX_NUMBER) where MAX_NUMBER represents the maximal number of bits that can be used to code it.

OUTPUT:
All the redundant binary radix codings for x as defined previously.

```
Procedure RedundantNumber(x,s,m)
x:integer;
m:integer;
s: string;

begin
mincode:integer;
maxcode:integer;
i:integer;

if (x = 0) then Display(s0^m); else
begin
mincode := log_2 ⌊|x| + 1⌋ + 1;
maxcode := log_2 ⌊|x| − 1⌋ + 3;
maxcode := MAX(maxcode, m);
for i := maxcode downto mincode do
                begin
                if (x > 0) then RedundantNumber(x − 2^(i−1),s0^(maxcode−i)1,i-1);
                else RedundantNumber(x + 2^(i−1),s0^(maxcode−i)1̄,i-1);
                end
end
end
```

1.000000000	$\frac{1}{1}$	$1/\infty$
1.500000000	$\frac{3}{2}$	$1/2/\infty$
1.570000000	$\frac{157}{100}$	$1/1/1/3/14/\infty$
1.570000000	$\frac{157}{100}$	$1/1/1/3/14/\infty$
1.570700000	$\frac{15707}{10000}$	$1/1/1/3/27/1/2/1/1/1/4/\infty$
1.570790000	$\frac{157079}{100000}$	$1/1/1/3/31/1/2/16/4/2/\infty$
1.570796000	$\frac{392699}{250000}$	$1/1/1/3/31/1/41/1/1/2/1/3/\infty$
1.570796300	$\frac{15707963}{10000000}$	$1/1/1/3/31/1/121/3/1/4/3/1/1/2/\infty$
1.570796320	$\frac{9817477}{6250000}$	$1/1/1/3/31/1/138/1/2/2/1/4/4/\infty$
1.570796326	$\frac{785398163}{500000000}$	$1/1/1/3/31/1/144/1/18/8/2/1/7/4/\infty$

Figure 2: Approximations of $\frac{\pi}{2}$.

computation can also be seen in term of matrix using the convention $x \equiv (p\ q)$ if $x = \frac{p}{q}, \gcd(p,q) = 1$ (in practice, its always the case since irrationals must be approximated to a closer rational according to the selected precision):

$$f(x) \equiv (x\ 1) \times \begin{pmatrix} a & b \\ c & d \end{pmatrix} = (p\ q) \times \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

For example, if we take $f(x) = \frac{3x+1}{2x+3}$, we have the general matrix notation $f(x) \equiv (x\ 1) \times \begin{pmatrix} 3 & 2 \\ 1 & 3 \end{pmatrix}$. Our algorithm takes its input piecewise and must also deliver output piecewise. Thus, it is possible to develop a pipeline structure corresponding to the evaluation of a more complex function. If x is coded into its continued fraction $[a_0/a_1/\ldots/a_n]$, the partial quotients $a_0, a_1, \ldots, a_n$ will be successively entered (the partial quotient defines the size[4] of the input). At step i, the first i partial quotients will be consumed and the result of $f(x_i)$ where x_i is the i-th convergent of x is known. It is therefore possible to consider a process that approximate a real to a rational written in continued fraction and deliver its successive partial quotients to the process that compute the function.

$$\boxed{\text{EVALUATION OF } \frac{\pi}{2}} \longrightarrow \boxed{\text{COMPUTATION OF } f(\tfrac{\pi}{2})}$$

Observation 2 *It is worth noting that the $(i+1)$-th partial quotient $a_i(\sigma)$ determined at step[5] σ when performing approximation of $x = [a_0/a_1/\ldots/a_i/\ldots]$ is the same as a_i iff there exists two consecutive steps σ_1 and σ_2 where $a_i = a_i(\sigma_1) = a_i(\sigma_2)$.*

2.2 Consuming Input Piecewise:

If we note $X = (x\ 1) = (p\ q) \equiv x$, we can rewrite the function into

$$Y = X \times \left\{ \underbrace{I^{-1} \times I}_{\text{Identity matrix}} \right\} \times F \times \left\{ \underbrace{O^{-1} \times O}_{\text{Identity matrix}} \right\}$$

We can group the terms in a different way

$$Y = \left\{ \underbrace{X \times I^{-1}}_{\text{input process}} \right\} \times \left\{ \underbrace{I \times F \times O^{-1}}_{\text{update process}} \right\} \times \left\{ \underbrace{O}_{\text{output process}} \right\} \tag{2}$$

[4] The size is seen as a parameter which defines the granularity of the atom.
[5] *The notion of step, here, denotes the current rational approximation.*

ALGORITHM -VII – CONSUMING INPUT

Scan the input until a partial quotient a_{i+1} is available or the termination signal is perceived. Upon the termation signal, depending on the parity of i, perform the product of matrices $S_e \times F_i$ (i is even) or $S_o \times F_i$ (i is odd) else input a_{i+1} is F_i by multiplying M_{i+1} with F_i: $Y = X \times X_{i+1}^{(-1)} \times (M_{i+1} \times F_i)$.

---

where I, O are well chosen inversible matrices (i.e. $\det O = \det I = 1$). Given F and the shape of X, it can be deduced that both I and O are (2×2) matrices.

We can write the latest equation by expanding $X \equiv x = [a_0/\ldots/a_n] = M_n \times \ldots \times M_0$ where $M_n, \ldots, M_0$ are inversible matrices representing the partial quotients (we associate the matrix M_i to the partial quotient a_i).

Property 7 *Let $x = [a_0, \ldots, a_{2i}]$ be an even continued fraction, then we can rewrite x in term of products of simple matrices:*

$$X = (x\ 1) = \underbrace{\overbrace{(0\ 1)}^{\textit{choose row}}}_{S_e} \times \begin{pmatrix} 1 & 0 \\ a_{2i} & 1 \end{pmatrix} \times \begin{pmatrix} 1 & a_{2i-1} \\ 0 & 1 \end{pmatrix} \times \ldots \times \begin{pmatrix} 1 & 0 \\ a_0 & 1 \end{pmatrix}$$

if $x = [a_0, \ldots, a_{2i+1}]$, then the choose-row matrix is $(1\ 0)$; Hence

$$X = (x\ 1) = \underbrace{\overbrace{(1\ 0)}^{\textit{choose row}}}_{S_o} \times \begin{pmatrix} 1 & a_{2i+1} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ a_{2i} & 1 \end{pmatrix} \times \ldots \times \begin{pmatrix} 1 & 0 \\ a_0 & 1 \end{pmatrix}$$

If X is the matrix coding of $x = [a_0/\ldots/a_{2i}]$, then we can input a suite of Id transformations (theses transformations can be expressed as $M \times M^{-1}$), namely

$$Y = X \times \left\{ \begin{pmatrix} 1 & 0 \\ -a_0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & a_{-2i-1} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -a_{2i} & 1 \end{pmatrix} \right\} \times \left\{ \begin{pmatrix} 1 & 0 \\ a_{2i} & 1 \end{pmatrix} \begin{pmatrix} 1 & a_{2i-1} \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ a_0 & 1 \end{pmatrix} \right\} \times F$$

Expanding $X = M_{2i} \times \ldots \times M_0$ in the equation and using the fact that $M \times M^{-1} = Id$, we obtain

$$Y = (0\ 1) \times \underbrace{\left\{ \begin{pmatrix} 1 & 0 \\ a_{2i} & 1 \end{pmatrix} \begin{pmatrix} 1 & a_{2i-1} \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ a_0 & 1 \end{pmatrix} \right\}}_{\text{piecewise input}} \times F$$

Let us denote X_{2i} (X_{2i+1}) the matrix denoting the $2i$-th (respectively $(2i+1)$-th) convergent of x. Then we can write

$$X \times X_{2i}^{-1} \times (X_{2i} \times F)$$

We denote by F_i the result of the product of matrices $X_i \times F$ where X_i is the matrix coding the i-th convergent of x. Since as the end of the algorithm, we will have consumed a_n (and the current convergent is X_n), we will have

$$X \times X_n^{-1} \times (X_n \times F) = X \times X_N^{-1} \times F_n$$

But as depending on the parity, $X = S_e \times X_n$ (even parity) or $X = S_n \times X_n$ (odd parity), it follows:

- n is even: $Y = S_e \times F_n$
- n is odd: $Y = S_o \times F_n$

The algorithm to consume input is therefore easy.

2.3 Producing Output Piecewise:

We now focus on the problem of delivering output while inputing. Our result $r = f(x)$ can also be written in the continued fraction format. Hence, if $r = [b_0/\ldots/b_{2j}]$, its matrix representation is $(0\ 1) \times M_{2j} \times \ldots \times M_0$.

$$r = \frac{p'}{q'} \equiv (p'\ q') = \begin{pmatrix} 1 & 0 \\ b_{2j} & 1 \end{pmatrix} \times \begin{pmatrix} 1 & b_{2j-1} \\ 0 & 1 \end{pmatrix} \times \ldots \times \begin{pmatrix} 1 & 0 \\ b_0 & 1 \end{pmatrix}$$

We will also denote by F_i the resulting F when $(i+1)$ inputs have been processed:

$$F_i = \begin{pmatrix} 1 & 0 \\ a_{2i} & 1 \end{pmatrix} \begin{pmatrix} 1 & a_{2i-1} \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ a_0 & 1 \end{pmatrix} \times F$$

And for convenience, we will term F the current F_i. So we have always $Y = C_i \times F$ at step $(i+1)$ where C_i is the selector matrix defined by

$$C_i = \begin{cases} (0\ 1) \iff i \text{ is even} \\ (1\ 0) \iff i \text{ is odd} \end{cases}$$

When the i-th input has been consumed, $Y \equiv f(x_i)$ where x_i is the i-th convergent of x.

Y is a *two dimensional vector* $Y = (p'\ q')$. This vector represents the rational $\frac{p'}{q'}$ which can be written in $[b_0/\ldots/b_{2j}]$. We must therefore assert that Y denotes always a positive rational number. This defines the output condition.

Depending on the parity of the output index, we must check if producing output is possible by asserting the output condition $\mathcal{P}(a)$:

$$\mathcal{P}(a) = \begin{cases} F_{(1)} - aF_{(2)} \geq 0 & \iff \text{output index is even} \\ -aF_{(1)} + F_{(2)} \geq 0 & \iff \text{output index is odd} \end{cases}$$

where $F_{(i)}$ term the i^{th} column of F

It is worth noting, that if some accumulation of inputs is done, we can just multiply the input matrices and then compute the product of the resulting matrix with the current F. This statement corresponds to the equation

$$Y = X \times X_k^{-1} \times \underbrace{(M_k \times \ldots \times M_{i+1})}_{\text{accumulation}} \times F_i$$

$$Y = X \times X_k^{-1} \times M' \times F_i$$

where M' represents the product of matrices $M_k \times \ldots \times M_{i+1}$:$M' = X_k \times X_i^{-1}$.

We choose, for the velocity of the algorithm, the larger a that satisfies $\mathcal{P}(a)$ (in practice, as it is presented with the generalized matrices, we use the *decision hypercube* which determines in a constant time which a, if there exists such a a, must be delivered).

When no input can be performed (for instance, ∞ has been entered[6]) then we choose a row of F depending on the input parity. This step corresponds to compute the product of matrices $Y = S_e \times F_n$ (second row) or $Y = S_o \times F_n$ (first row). This row represents the rational number $\frac{p'}{q'} = r$ which can be represented by a continued fraction.

We then merge both results (the continued fraction representing $\frac{p'}{q'}$ and the previous results delivered before) taking care of the output parity when the algorithm has read ∞ as input. Indeed, if the output parity is odd, then the next output that could have been produced will be even (and the factorization of $\frac{p'}{q'}$ begins by a'_0) so the normalization is already done. Otherwise, we add the first coefficient of $\frac{p'}{q'}$ and the last output.

In term of matrix representation, if $[r_0/\ldots/r_i]$ has been produced like output when the algorithm catches ∞ and if $[f_1/\ldots/f_j]$ denotes the remaining rational $\frac{p'}{q'}$, then it follows:

$$Y = X \times X_n^{-1} \times F_j \times M_{f_j} \times \ldots \times M_{f_1} \times M_{r_i} \times \ldots \times M_{r_0}$$

where $F_j = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}$ if $(j+i-1)$ is odd or $\begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$ otherwise.

[6] This can be denoted by a signal in an hardware realization.

ALGORITHM -VIII – FUNCTION OF ONE VARIABLE

Process 1:**Consuming input**
Scan the source until a partial quotient a_i (corresponding to the i-th input) is available . Update F by performing the multiplication of matrices $M_i \times F$.

Process 2:**Producing output**
Find the larger a such that $\mathcal{P}(a)$ is satisfied by intersecting the two interval conditions. Output M_j and update F by performing the right-side product of matrix $F \times M_j^{-1}$.

Process 3:**Calling termination**
Depending on the output parity, choose the first or the second row of the current F.

As the final result expressed as product of matrices must denote a continued fraction, we must have an alternance of the kind of partial quotient matrices $M_{f_j}, \ldots, M_{f_0}, M_{r_i}, \ldots, M_{r_0}$. But as M_{f_0} is of kind "even", we must have M_{r_i} that must be "odd". So that if M_{r_i} is not "odd", we insert between M_{r_i} and M_{f_0} an Id matrix which corresponds to a null partial quotient.

The last case, implies that we must not deliver output directly, but wait that there is another output (partial quotient) to deliver the previous one.

In practice, once the termination signal is catched, we input accordingly to the input parity M_∞ and produce output until $S_{input} \times F$ represents a switch matrix (S_{input} is the input switch matrix.

A *graphic simulator* **MCF-graphic**[7] that traces each step of the algorithm has been written. In annex, a LaTeXoutput given by *MCF-SIM.* is enclosed.

Finally, one of the more interesting things, is that we can combine in a same cell both the way we input and output data. Indeed, in the previous example we have chosen I, O such as they satisfy the inversibility and that our initial vector can also be factorized in the same way as I. Then, without a lot of modification we can use several codes:

1. Continued Fraction
2. Continued Logarithmic Fraction
3. Lexicographic Continued Fraction
4. Redundant Partial Quotient (RPQ)
5. Redundant Bit-grained Continued Fraction
6. Radix
7. Redundant radix

The reader is invited to read the first section dealing with the codings of numbers ,to see how to proceed. For each format of output, we must change accordingly the *output condition.* For instance, in the radix representation, we must perform the sum of the elements belonging to each column of F to determine how many inputs must be processed before output of the bit of order k can be produced (2^k). Indeed, assume we have the bit of 2^i as input, then if r denotes the sum of the chosen column, then the relation $r2^j < 2^k$ must be satisfied. To bound output delay, we see in that example that it is preferable to have a redundant coding of number which tightens the output delay.

[7]This simulator allows both function of continued fractions and multicontinued fractions to be computed

3 Introduction to Generalized Matrices:

In this section, we study the computation of a function of n rational variables $(X_1 = (x_1\ 1), \ldots, X_n = (x_n\ 1))$ into the space $\mathbb{Q}$ (we denote F this mapping function). F represents a mapping function of $\mathbb{Q}^n$ to $\mathbb{Q}$ expressed as follows:

$$F : \mathbb{Q}^n \to \mathbb{Q}$$

$$X_1, \ldots, X_n \qquad \longmapsto \qquad f(x_1, \ldots, x_n)$$

We want to provide input *piecewise* and supply output *piecewise*, thus allowing a pipelined structure for evaluating complex rational functions.

For our purpose, we introduce the notion of generalized matrix in which components are denoted by product of matrices which cannot be evaluated while the algorithm runs. Let us consider the following example:
Consider the matrix M defined by

$$M = \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix} = \begin{pmatrix} a_1x + c_1 & b_1x + d_1 \\ a_2x + c_2 & b_2x + d_2 \end{pmatrix}$$

Then M can be rewritten as follows:

$$\begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix} = \begin{pmatrix} (x\ 1) \times \begin{pmatrix} a_1 & b_1 \\ c_1 & d_1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 0 \end{pmatrix} & (x\ 1) \times \begin{pmatrix} a_1 & b_1 \\ c_1 & d_1 \end{pmatrix} \times \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ (x\ 1) \times \begin{pmatrix} a_2 & b_2 \\ c_2 & d_2 \end{pmatrix} \times \begin{pmatrix} 1 \\ 0 \end{pmatrix} & (x\ 1) \times \begin{pmatrix} a_2 & b_2 \\ c_2 & d_2 \end{pmatrix} \times \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix}$$

We can rewrite the previous equation into the following form:

$$\begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix} = \begin{pmatrix} (x\ 1) \times \begin{pmatrix} a_1 & b_1 \\ c_1 & d_1 \end{pmatrix} \\ (x\ 1) \times \begin{pmatrix} a_2 & b_2 \\ c_2 & d_2 \end{pmatrix} \end{pmatrix}$$

By denoting $M_1 = \begin{pmatrix} a_1 & b_1 \\ c_1 & d_1 \end{pmatrix}$ and $M_2 = \begin{pmatrix} a_2 & b_2 \\ c_2 & d_2 \end{pmatrix}$, we finally obtain a compact writing which will be used in the remaining article:

$$\begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix} = \begin{pmatrix} (x\ 1) \times M_1 \\ (x\ 1) \times M_2 \end{pmatrix}$$

The mapping function can, then , be expressed as term of generalized matrix: The computational function we allow, can be denoted by:

In dimension 2, we allow computation of functions that can be represented by

$$(x_1\ 1) \times \begin{pmatrix} (x_2\ 1) \times \begin{pmatrix} a_1 & b_1 \\ c_1 & d_1 \end{pmatrix} \\ (x_2\ 1) \times \begin{pmatrix} a_2 & b_2 \\ c_2 & d_2 \end{pmatrix} \end{pmatrix}$$

or simpler in

$$(x_1\ 1) \times \begin{pmatrix} (x_2\ 1) \times M_1^{(2)} \\ (x_2\ 1) \times M_2^{(2)} \end{pmatrix}$$

And to extend that notion to the general case, we introduce the recursive definition of $M(\cdot)$:

$$M(x_1) = (x_1\ 1) \times \begin{pmatrix} a & b \\ c & d \end{pmatrix} \qquad \text{and} \qquad M(x_1, \ldots, x_n) = (x_1\ 1) \times \begin{pmatrix} M(x_2, \ldots, x_n) \\ M(x_2, \ldots, x_n) \end{pmatrix}$$

Each input X_i is characterized by a (1×2) matrix (2-vector) representing the variable $x_i = \frac{p_i}{q_i} \equiv (p_i\ q_i) = (\frac{p_i}{q_i}\ 1) = (x_i\ 1)$. We allow transformations on each variable of the following type:$\mathcal{T}(x) = \mathcal{T}(\frac{p}{q}) = \frac{ap+bq}{cp+dq} = \frac{ax+b}{cx+d}$. Since a, b and c, d can also be result of the same kinds of transformations, we can generate with that recursive process polynomial rational functions of n variables.

It is worth noting, that the matrix $(x_i\ 1)$ is multiplied by 2^{i-1} generalized matrices and that only the matrix variable $(x_n\ 1)$ are multiplied by integral coefficient 2×2 matrix (we term *concrete matrix* this kind of matrices, on the contrary we say M is *virtual* if it iiis a generalized matrix). The computation of $F(x_1, \ldots, x_n)$ required then 2^{n-1} (2×2)–matrix (2^{n+1} numbers of $\mathbb{Z}$). The family of functions described by the recursive process can also be expressed recursively: $\frac{x_i N(x_{i+1},\ldots,x_n)+N(x_{i+1},\ldots,x_n)}{x_i N(x_{i+1},\ldots,x_n)+N(x_{i+1},\ldots,x_n)}$ where N is the generic recursive functions ; By recurrence hypothesis, it can be shown that the family of functions generated by the process is:

$$\mathcal{F} = \left\{ \frac{\displaystyle\sum_{v=(v_1,\ldots,v_n)\in V} B(v) \times x_1^{v_1} \ldots x_i^{v_i} \ldots x_N^{v_n}}{\displaystyle\sum_{u=(u_1,\ldots,u_n)\in V} B(v) \times x_1^{u_1} \ldots x_i^{u_i} \ldots x_N^{u_n}} \right\}$$

where

V is a vector in $\{0,1\}^n$ and $B(o)$ is the binary number indexed variable representing by o.

$$B((o_1, \ldots, o_n)) = B_{\sum_{i=1}^{n} 2^{i-1} o_i}$$

For instance, taking $n = 2$, and expanding the sums in the numerator and denominator ,we obtain:

$$\mathcal{F}_2 = \left\{ \frac{a \times xy + b \times x + c \times y + d}{a' \times xy + b' \times x + c' \times y + d'} \right\}$$

Several steps (different kinds of events) are to be observed before describing entirely the algorithm:

- The way input data are processed piecewise by allowing interleaved entrance.
- The output condition to supply result partial quotient r_i
- Reducing the depth of the computational tree when a variable has been entirely entered.

We also investigate the possibility to change the format of numbers and, hence, combine either representation of numbers in radix, redundant radix, continued fraction, logarithmic continued fraction, redundant continued fraction, ...

In dimension n, the function $(f(x_1, \ldots, x_n))$ that can be computed is a fraction of two polynomial functions of n variables.

$$f(x_1, \ldots, x_n) = \frac{P_{num}(x_1, \ldots, x_n)}{P_{den}(x_1, \ldots, x_n)}$$

Definition 2 *We can write this function in extenso as follows:*

$$f(x_1, \ldots, x_n) = \frac{\displaystyle\sum_{j=0}^{2^n-1} q_{2j} \prod_{i=0}^{n-1} x_i^{\frac{j \odot 2^i}{2^i}}}{\displaystyle\sum_{j=0}^{2^n-1} q_{2j+1} \prod_{i=0}^{n-1} x_i^{\frac{j \odot 2^i}{2^i}}}$$

where $q_j \in \mathbb{N}, \forall\, j \in [0, 2^{n+1} - 1]$ *and* $\odot$ *represents the binary operator* $AND(\cdot,\cdot)$.

Developing the kind of function for $n = 3$ (three variables x_1, x_2, x_3) , we obtain:

$$f(x_1, x_2, x_3) = \frac{\displaystyle\sum_{j=0}^{7} q_{2j} \prod_{i=0}^{2} x_i^{\frac{j \odot 2^i}{2^i}}}{\displaystyle\sum_{j=0}^{7} q_{2j+1} \prod_{i=0}^{2} x_i^{\frac{j \odot 2^i}{2^i}}}$$

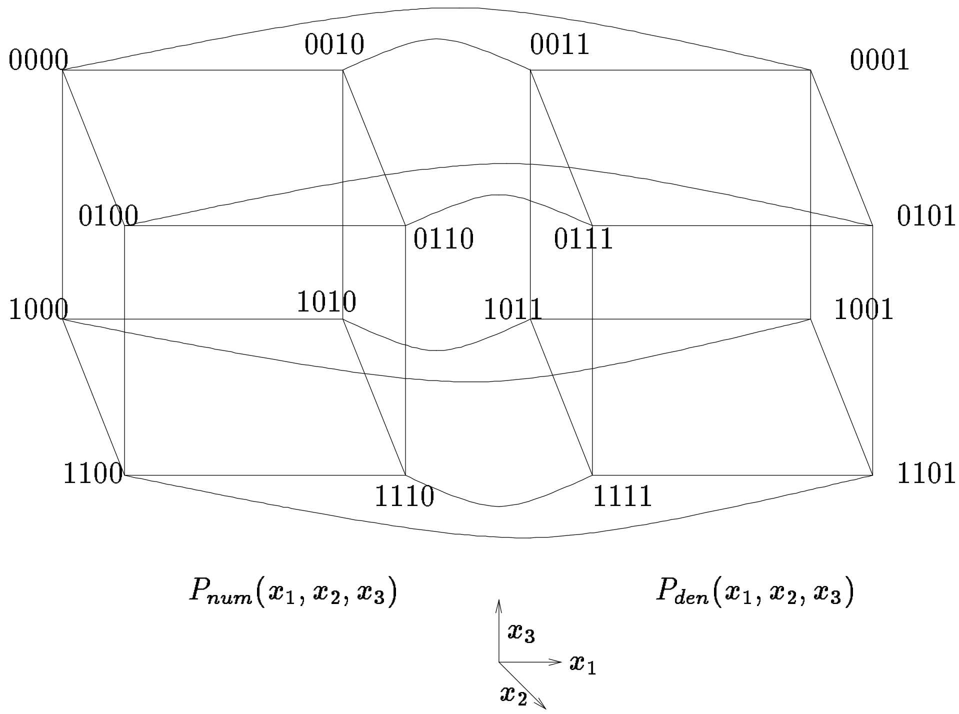


Figure 3: The **Gray** code on $\mathcal{H}$.

We denote by $(\cdot)_2$, the binary representation of a number (for example $(1010)_2 = 10$).

$$f(x_1, x_2, x_3) =$$
$$\frac{q_{(0000)_2}+q_{(0010)_2}x_1+q_{(0100)_2}x_2+q_{(0110)_2}x_2x_1+q_{(1000)_2}x_3+q_{(1010)_2}x_3x_1+q_{(1100)_2}x_3x_2+q_{(1110)_2}x_1x_2x_3}{q_{(0001)_2}+q_{(0011)_2}x_1+q_{(0101)_2}x_2+q_{(0111)_2}x_2x_1+q_{(1001)_2}x_3+q_{(1011)_2}x_3x_1+q_{(1101)_2}x_3x_2+q_{(1111)_2}x_1x_2x_3}$$

$$f(x_1, x_2, x_3) = \frac{q_0 + q_2x_1 + q_4x_2 + q_6x_2x_1 + q_8x_3 + q_{10}x_3x_1 + q_{12}x_3x_2 + q_{14}x_1x_2x_3}{q_1 + q_3x_1 + q_5x_2 + q_7x_2x_1 + q_9x_3 + q_{11}x_3x_1 + q_{13}x_3x_2 + q_{15}x_1x_2x_3}$$

The i-th coefficient q_i can be placed on a 4-hypercube $\mathcal{H}$ according to its index i. Numbering the hypercube $\mathcal{H}$ with a **Gray** code, the integral coefficient q_i is positioned on the node that has its **Gray** code equals to i. The figure 3 shows how the 4-dimensional hypercube $\mathcal{H}$ is numbered with a **Gray** code. The repartition of the coefficients $q_i,\ i \in [0, 15]$ is shown in figure 4.

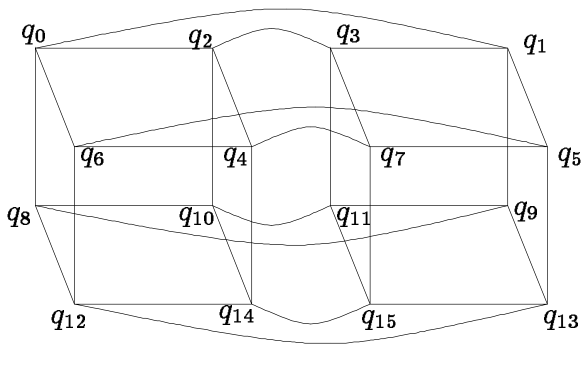


Figure 4: Positionning the coefficients q_i on $\mathcal{H}$.

3.1 Consuming Input Piecewise:

In our algorithm, we can interleave data from any variable once information is available. Hence, a process scans permanently if a partial quotient from any variable is available. Variable $(x_i\ 1)$ denote the rational $\frac{p_i}{q_i}$, $\gcd(p_i, q_i) = 1$. In the following we develop the update process when variables are entered in term of partial quotients (further types of codings are analyzed at the end of that paper). Let $(p\ q)$ denote a variable. We have $(p\ q) = (\frac{p}{q}\ 1) = [a_0/a_1/\ldots/a_n]$. Using **non-redundant partial quotients**, each variable $(x_i\ 1)$ can be rewritten in term of products of simple inversible matrix:

If $n^{(i)}$ is odd:

$$(x_i\ 1) = (1\ 0) \times \begin{pmatrix} 1 & a_n^{(i)} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ a_{n-1}^{(i)} & 1 \end{pmatrix} \cdots \begin{pmatrix} 1 & 0 \\ a_{2j}^{(i)} & 0 \end{pmatrix} \begin{pmatrix} 1 & a_{2j-1}^{(i)} \\ 0 & 1 \end{pmatrix} \cdots \begin{pmatrix} 1 & 0 \\ a_0^{(i)} & 1 \end{pmatrix}$$

If $n^{(i)}$ is even:

$$(x_i\ 1) = (0\ 1) \times \begin{pmatrix} 1 & 0 \\ a_n^{(i)} & 1 \end{pmatrix} \begin{pmatrix} 1 & a_{n-1}^{(i)} \\ 0 & 1 \end{pmatrix} \cdots \begin{pmatrix} 1 & a_{2j+1}^{(i)} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ a_{2j}^{(i)} & 0 \end{pmatrix} \cdots \begin{pmatrix} 1 & 0 \\ a_0^{(i)} & 1 \end{pmatrix}$$

These matrix factorizations can be expressed simply by two kinds of easy-inversible matrix:

- Even Matrix: $E(a_{2j}^{(i)}) = \begin{pmatrix} 1 & 0 \\ a_{2j}^{(i)} & 1 \end{pmatrix}$.
- Odd Matrix: $O(a_{2j-1}^{(i)}) = \begin{pmatrix} 1 & a_{2j-1}^{(i)} \\ 0 & 1 \end{pmatrix}$.

The inverse matrix can also be expressed in term of these generating matrix:

- Even Inverse Matrix: $E^{-1}(a_{2j}^{(i)}) = E(-a_{2j}^{(i)}) = \begin{pmatrix} 1 & 0 \\ -a_{2j}^{(i)} & 1 \end{pmatrix}$.
- Odd Inverse Matrix: $O^{-1}(a_{2j-1}^{(i)}) = O(-a_{2j-1}^{(i)}) = \begin{pmatrix} 1 & -a_{2j-1}^{(i)} \\ 0 & 1 \end{pmatrix}$.

Notice, that the *leading* matrix $(0\ 1)$ or $(1\ 0)$ allows to choose among the 2 rows of the (2×2)-matrices : $\begin{pmatrix} p_i & q_i \\ p_{i+1} & q_{i+1} \end{pmatrix}$ or $\begin{pmatrix} p_{i+1} & q_{i+1} \\ p_i & q_i \end{pmatrix}$ (this way of proceeding takes into account the reciprocity **Euclid**'s algorithm).

We can rewrite the equation characterizing the computational functional matrix by replacing each variable denoted by $(x_i\ 1)$ by its appropriate partial quotient factorization.

Then, when an input of any variable is available, say from variable x_i, we perform the 2^{i-1} *generalized* product of matrix.

Indeed, the computational function can be seen as a tree where levels represent the index of variable and left edge, right edge, respectively the $M_1(.)$ and $M2(.)$ generalized matrix. Each leaf is a real (2×2) matrix and each internal node is a virtual matrix. We consider each edge as a *link* to a (1×2) matrix (when all the products after this node as been produced).

Suppose that $a_j^{(i)}$ is available, updatings on each subtree rooted at a node in depth $i-1$ must be performed in parallel. Depending on the parity on j, we must perform the matrix product $O(i) \times M$ or $E(i) \times M$ (odd respectively even parity). If M is a real (2×2) matrix (i.e. $i = n$) then that product can be done directly, otherwise we must update each node in the subrooted tree according the arithmetic operation.

Let us examine, the way we update generalized matrices when we perform a left-side product of a matrix by a generalized matrix:

Consider the product $\begin{pmatrix} a & b \\ c & d \end{pmatrix} \times \begin{pmatrix} (x_{i+1}\ 1) \times M_1 \\ (x_{i+1}\ 1) \times M_2 \end{pmatrix}$.

As we know that M_1, M_2 are (2×2) matrix, we can rewrite the equation such that each of the coefficient of the generalized matrix is expressed.

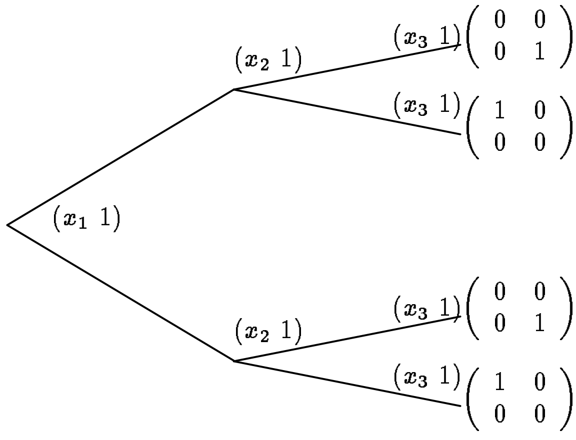


Figure 5: Representation of $M(x_1, x_2, x_3)$.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \times \begin{pmatrix} (x_{i+1}\ 1)M_1\begin{pmatrix} 1 \\ 0 \end{pmatrix} & (x_{i+1}\ 1)M_1\begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ (x_{i+1}\ 1)M_2\begin{pmatrix} 1 \\ 0 \end{pmatrix} & (x_{i+1}\ 1)M_2\begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix}$$

which straightforwardly gives:

$$\begin{pmatrix} (x_{i+1}\ 1)(aM_1 + bM_2)\begin{pmatrix} 1 \\ 0 \end{pmatrix} & (x_{i+1}\ 1)(aM_1 + bM_2)\begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ (x_{i+1}\ 1)(cM_1 + dM_2)\begin{pmatrix} 1 \\ 0 \end{pmatrix} & (x_{i+1}\ 1)(cM_1 + dM_2)\begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix}$$

We can therefore use the same notation and contract the last equation (and then reform a generalized matrix) in:

$$\begin{pmatrix} (x_{i+1}\ 1)M_1' \\ (x_{i+1}\ 1)M_2' \end{pmatrix} = \begin{pmatrix} (x_{i+1}\ 1)(aM_1 + bM_2) \\ (x_{i+1}\ 1)(cM_1 + dM_2) \end{pmatrix}$$

The $\alpha M_1 + \beta M_2$ operation can be dealt with recursivity as follows:

Since M_1, M_2 are generalized matrix, we can add each component by a recursive process which ends when reaching a leaf, i.e. a concrete (2×2) matrix.

$$\alpha \times \begin{pmatrix} \boxed{M_1'} \\ \boxed{M_1''} \end{pmatrix} + \beta \times \begin{pmatrix} \boxed{M_2'} \\ \boxed{M_2''} \end{pmatrix}$$

This is trivially equal to:

$$\begin{pmatrix} \boxed{\alpha M_1' + \beta M_2'} \\ \boxed{\alpha M_1'' + \beta M_2''} \end{pmatrix}$$

The recursivity stops when M_1' and M_2' are leaves (in that case, the result is computed immediately).

This operation can be realized fastly in an hardware implementation (when the number of variables is fixed, we update in parallel each leaf by a butterfly[8] chip.). One input can be processed at most in the same time (regarding the modifications of leaves).

[8] This principle is also used in the Fast Fourier Transform. It is worth noting that in an hardware realization, the complexity of consuming input is constant

3.2 Another Way to Process Input:

A partial quotient a must be proceeded according to the i-th variable x_i. We can first reorganize the computational generalized matrix, such as x_n is permuted with x_i. Then we perform the input by just performing the 2^{n-1} multiplications in parallel with $\begin{pmatrix} 1 & a \\ 0 & 1 \end{pmatrix}$ (odd) or $\begin{pmatrix} 1 & 0 \\ a & 1 \end{pmatrix}$ (even). The problem is therefore to reorganize the generalized matrix T denoting the function $f(x_1, \ldots, x_i, \ldots, x_n)$ into a generalized matrix T' which will denote $f'(x_1, \ldots, x_n, \ldots, x_i)$ such that $f(x_1, \ldots, x_i, \ldots, x_n) = f'(x_1, \ldots, x_n, \ldots, x_i)$. This permutation of the variable can be denoted by its unique signature, so that the problem is to find the transformation which can permute two successive variables x_i and x_{i+1}.

3.2.1 Analyze in Term of Matrices:

Let us consider the factorization

$$T = (x_i\ 1) \times \begin{pmatrix} (x_{i+1}\ 1)M_1 \\ (x_{i+1}\ 1)M_2 \end{pmatrix}$$

We want to find a transformation $\mathcal{R}$, such that if $\mathcal{R}$ is applied on T, the new generalized matrix computing the same function can be written as:

$$\mathcal{R}(T) = (x_{i+1}\ 1) \times \begin{pmatrix} (x_i\ 1)M'_1 \\ (x_i\ 1)M'_2 \end{pmatrix}$$

Let us develop the function denoted by T according to respectively x_{i+1} and x_i and factorize it again following x_{i+1} and x_i:

$$(x_i\ 1) \times \begin{pmatrix} (x_{i+1}\ 1)\begin{pmatrix} N_1 & D_1 \\ N'_1 & D'_1 \end{pmatrix} \\ (x_{i+1}\ 1)\begin{pmatrix} N_2 & D_2 \\ N'_2 & D'_2 \end{pmatrix} \end{pmatrix}$$

$$(x_i\ 1) \times \begin{pmatrix} x_{i+1}N_1 + N'_1 & x_{i+1}D_1 + D'_1 \\ x_{i+1}N_2 + N'_2 & x_{i+1}D_2 + D'_2 \end{pmatrix}$$

$$(x_i x_{i+1} N_1 + x_i N'_1 + x_{i+1} N_2 + N'_2 \qquad x_i x_{i+1} D_1 + D'_1 + x_{i+1} D_2 + D'_2)$$

Factorizing:

$$(x_{i+1}\ 1) \times \begin{pmatrix} x_i N_1 + N_2 & x_i D_1 + D_2 \\ x_i N'_1 + N'_2 & x_i D'_1 + D'_2 \end{pmatrix}$$

$$(x_{i+1}\ 1) \times \begin{pmatrix} (x_i\ 1)\begin{pmatrix} N_1 & D_1 \\ N_2 & D_2 \end{pmatrix} \\ (x_i\ 1)\begin{pmatrix} N'_1 & D'_1 \\ N'_2 & D'_2 \end{pmatrix} \end{pmatrix}$$

The new matrices, corresponding to that computation, are $M'_1 = \begin{pmatrix} N_1 & D_1 \\ N_2 & D_2 \end{pmatrix}$ and $M'_2 = \begin{pmatrix} N'_1 & D'_1 \\ N'_2 & D'_2 \end{pmatrix}$.

3.2.2 Using the Hypercube Structure:

Processing the input of a partial quotient p of variable x_i can be seen as rotating the hypercube such that the variable x_i takes the orientation of x_n and then processing the input of partial quotient p from variable $x'_n = x_i$. The rotation of the hypercube is always constrained by the orientation of the output axis[9]. Once this rotation is done, processing input can be done by multiplying the (2×2) input matrix $M(p)$ with the 2^{n-1} concrete matrices.

[9] The output axis can also be interpreted as variable x_0.

Observation 3 *Rotating the hypercube $\mathcal{H}$, so that the i-axis and the $n-$axis are swapped, corresponds for each node to swap the bit of weight n with the bit of weight i of its* ***Gray*** *code.*

$$(x_n \dots x_i \dots x_0)_2 \Rightarrow (x_i \dots x_n \dots x_0)$$

An input of matrice $M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ in dimension i is noted $\begin{pmatrix} a & b \\ c & d \end{pmatrix}_i$. So that if $M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ is an input corresponding to the i-th variable (x_i), we must perform the updating

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}_i \times \boxed{\text{Generalized Matrix Denoting } f}$$

3.3 Shrinking the Computational Tree when an Input is Exhausted:

When no more partial quotient can be entered in variable i, i is said to be exhausted (in an hardware, it can be a specific signal or a symbol representing ∞ which, in that case, denotes the continued fraction infinite expansion. The computational tree can be collapsed into a smaller one where the level corresponding to the variable i has been deleted and the remaining nodes below level i have been updated according to the parity of the last input of that variable. The update process is simple since only two cases can occur:

- x_i ended with odd parity: then the switch matrix is $(1\ 0)$. And performing the multiplication yields to choose the *top* son.
- x_i ended with even parity: then the switch matrix is $(0\ 1)$. And performing the multiplication yields to choose the *bottom* son.

In term of matrix writing, this correspond to the product:
Even Parity:

$$(0\ 1) \times \begin{pmatrix} (x_{i+1}\ 1)M_1 \\ (x_{i+1}\ 1)M_2 \end{pmatrix} = (x_{i+1}\ 1)M_2$$

Odd Parity:

$$(1\ 0) \times \begin{pmatrix} (x_{i+1}\ 1)M_1 \\ (x_{i+1}\ 1)M_2 \end{pmatrix} = (x_{i+1}\ 1)M_1$$

It must be noted,that when an exhaustion operation is performed, the function becomes a function of $n-1$ variables and can then be set to a hypercube of dimension n (instead of $n+1$). This corresponds to select the nodes n which gray code satisfies:

- x_i ends with even parity: $n \odot 2^i = 2^i$
- x_i ends with odd parity: $n \odot 2^i = 0$

3.4 The Output Condition:

To allow pipelined evaluation of complex functions, we must also deliver the result piecewise. For that purpose, we must assume that the result is in some interval :$[r_0, r_1]$

Remind that in the case where the output is expressed as a non redundant continued fraction, we have

$$\begin{array}{l} \{\frac{p_{r-1}}{q_{r-1}}, \frac{p_r}{q_r}\} \text{ if } r \text{ is odd} \\ \{\frac{p_r}{q_r}, \frac{p_{r-1}}{q_{r-1}}\} \text{ if } r \text{ is even} \end{array}$$

which represent a simplex in dimension 1 (for further information about the simplex, refer to the linear and geometrical book [2]). We must assure that the 4 coefficients of the matrix $\begin{pmatrix} (x_2\ 1)M_1 \\ (x_2\ 1)M_2 \end{pmatrix} \times$

$\begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix}$ are positive, hence conserving the convex property of the output simplex (the output matrix is $\begin{pmatrix} 1 & a_{2r+1} \\ 0 & 1 \end{pmatrix}$ or $\begin{pmatrix} 1 & 0 \\ a_{2r} & 1 \end{pmatrix}$). Performing the product, we obtain the following generalized matrix:

$$\begin{pmatrix} (x_2\ 1)(a'M_1\begin{pmatrix}1\\0\end{pmatrix}+c'M_1\begin{pmatrix}0\\1\end{pmatrix}) & (x_2\ 1)(b'M_1\begin{pmatrix}1\\0\end{pmatrix}+d'M_1\begin{pmatrix}0\\1\end{pmatrix}) \\ (x_2\ 1)(a'M_2\begin{pmatrix}1\\0\end{pmatrix}+c'M_2\begin{pmatrix}0\\1\end{pmatrix}) & (x_2\ 1)(b'M_2\begin{pmatrix}1\\0\end{pmatrix}+d'M_2\begin{pmatrix}0\\1\end{pmatrix}) \end{pmatrix}$$

$$\begin{pmatrix} (x_2\ 1)M_1\begin{pmatrix}a'\\c'\end{pmatrix} & (x_2\ 1)M_1\begin{pmatrix}b'\\d'\end{pmatrix} \\ (x_2\ 1)M_2\begin{pmatrix}a'\\c'\end{pmatrix} & (x_2\ 1)M_2\begin{pmatrix}b'\\d'\end{pmatrix} \end{pmatrix}$$

The operations to determine if an output can be produced are done inside the datastructure since the right-side product of matrix doesn't conserve the datastructure.

The frames inside the matrix show the rigidity of the datastructure that must be conserved:

$$\begin{pmatrix} \boxed{M_1'\ M_1''} \\ \boxed{M_2'\ M_2''} \end{pmatrix} \times \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

We have exibited the two components on our single opaque structure($M_1 = \boxed{M_1'\ M_1''}$) in order to ease the visualization. The operations to perform are

- $M_1' \leftarrow aM_1' + bM_1''$
- $M_1'' \leftarrow cM_1' + dM_1''$
- $M_2' \leftarrow aM_2' + bM_2''$
- $M_2'' \leftarrow cM_2' + dM_2''$

But since, in the general case, M_1', M_1'', M_2', M_2'' are also generalized matrices, we must execute a recursive process on both the updating and conditions procedures (indeed, we do not know the parity of variables before they end and we must therefore assume the condition on both cases). So, that each row of real matrix: e_1e_2 must satisfy:

- $ae_1 - ce_2 \geq 0$
- $ce_1 - ce_2 \geq 0$

We see that this output condition represents the bottleneck of the algorithm. This output condition can nonetheless be computed quickly as described later with the decision hypercube. A more exhausted study and generalization of the output condition is given with the decision hypercube.

3.5 Changing the Format of Numbers:

In order to have a bitwise grained algorithms, we must refine the input and output in the previous algorithm. This can be achieved by noting the two matrix factorizations of a simple partial quotient.

$$\begin{pmatrix} 1 & p \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 2^n & 2^n \\ 0 & 1 \end{pmatrix} \times \left\{ \begin{pmatrix} \frac{1}{2} & b_{n-1} \\ 0 & 1 \end{pmatrix} \cdots \begin{pmatrix} \frac{1}{2} & b_0 \\ 0 & 1 \end{pmatrix} \right\}$$

$$\begin{pmatrix} 1 & 0 \\ p & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 2^n & 2^n \end{pmatrix} \times \left\{ \begin{pmatrix} 1 & 0 \\ \frac{b_{n-1}}{2} & \frac{1}{2} \end{pmatrix} \cdots \begin{pmatrix} 1 & 0 \\ \frac{b_0}{2} & \frac{1}{2} \end{pmatrix} \right\}$$

where

$$p = 2^n + \sum_{i=0}^{n-1} b_i 2^i$$

We want to have a one-to-one relation between the bit string in input: $u^{n-1}b_{n-1}b_{n-2}\dots b_1 b_0$ and the matrix. So that, when one bit is available, the corresponding matrix is applied in our output process.

We decompose $\begin{pmatrix} 1 & 0 \\ 2^n & 2^n \end{pmatrix}$ into a product of simple matrices:

$$\begin{pmatrix} 1 & 0 \\ 2^n & 2^n \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}^n \times \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

and $\begin{pmatrix} 2^n & 2^n \\ 0 & 1 \end{pmatrix}$ in

$$\begin{pmatrix} 2^n & 2^n \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}^n \times \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

Since our coding is $u^{n-1}b_{n-1}b_{n-2}\dots b_1 b_0$, we must perform a switch matrix when input switch between the last u and b_{n-1}. This matrix takes into account the u which has been simplified and the matrix $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$ or $\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$. So that, depending on the parity of the partial quotient, we have:

$$S_e = \begin{pmatrix} 1 & 0 \\ 2 & 2 \end{pmatrix}$$

and the odd switch matrix

$$S_o = \begin{pmatrix} 2 & 2 \\ 0 & 1 \end{pmatrix}$$

These matrices are simple and performing multiplication with another (2×2) matrix just corresponds to shift-or-add operations. In an hardware implementation, each partial quotient must be entered as follows: a leading sequence of u^{n-1} which indicate the length n, then successively $b_{n-1}, \dots, b_0$. For instance 7 is represented by the binary string 111011. There are many codings that are available (some of them are redundant and allow output to be produced fastly). It is worth noting that we can mix **input/output** coding format since each format has its own factorization matrix (see before for a complete definition of codings).

3.6 Computing the Tree-like Matrix Given a Function:

We describe in this part an algorithm which takes a function $f = \frac{N(x_1,\dots,x_n)}{D(x_1,\dots,x_n)}$ and give its generalized matrix factorization following $x_1, \dots, x_n$. We factorize both numerator $N(\cdot)$ and denominator $D(\cdot)$ following x_1. It comes

$$f = \frac{x_1 N_1(x_2,\dots,x_n) + N_2(x_2,\dots,x_n)}{x_1 D_1(x_2,\dots,x_n) + D_2(x_2,\dots,x_n)}$$

That function can be written in the matrix form:

$$f = (x_1\ 1) \times \begin{pmatrix} N_1(x_2,\dots,x_n) & D_1(x_2,\dots,x_n) \\ N_2(x_2,\dots,x_n) & D_2(x_2,\dots,x_n) \end{pmatrix}$$

where $N_1(\cdot), N_2(\cdot), D_1(\cdot), D_2(\cdot)$ are obtained by the same process.

With the trivial case (which ends the recursivity) $f = \frac{x_i a + b}{x_i c + d} \equiv (x_i\ 1) \times \begin{pmatrix} a & c \\ b & d \end{pmatrix}$.

Let us develop the algorithm on $f = \frac{3xyz + 2xy + 3z + y}{4xyz + 2z + 3}$. The factorization is done successively on x, y, z.

Factorization according to x:

$$f = \frac{x(3yz + 2y) + (3z + y)}{x(4yz) + (2z + 3)}$$

Current generalized matrix:

$$(x\ 1) \times \begin{pmatrix} 3yz + 2y & 4yz \\ 3z + y & 2z + 3 \end{pmatrix}$$

Factorization according to y:

- $\frac{y(3z+2)}{y(4z)}$
- $\frac{y(1)+(3z)}{y(0)+(2z+3)}$

Current generalized matrix:

$$(x\ 1) \times \begin{pmatrix} (y\ 1)\begin{pmatrix} 3z+2 & 4z \\ 0 & 0 \end{pmatrix} \\ (y\ 1)\begin{pmatrix} 1 & 0 \\ 3z & 2z+3 \end{pmatrix} \end{pmatrix}$$

Factorization according to z:

- $\frac{z(3)+(2)}{z(4)+(0)}$
- $\frac{z(0)+(0)}{z(0)+(0)}$[10]
- $\frac{z(0)+(1)}{z(0)+(0)}$
- $\frac{z(3)+(0)}{z(2)+(3)}$

At that step, the factorization is finished and the generalized matrix obtained is

$$(x\ 1) \times \begin{pmatrix} (y\ 1)\begin{pmatrix} (z\ 1)\begin{pmatrix} 3 & 4 \\ 2 & 0 \end{pmatrix} \\ (z\ 1)\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \end{pmatrix} \\ (y\ 1)\begin{pmatrix} (z\ 1)\begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \\ (z\ 1)\begin{pmatrix} 3 & 2 \\ 0 & 3 \end{pmatrix} \end{pmatrix} \end{pmatrix}$$

The coefficients can also be placed on the hypercube $\mathcal{H}$ and the hypercube is unfolding to give the generalized matrix (remind there is a one-to-one relation concerning the nodes between of complete tree of depth k and an hypercube in dimension k).

3.7 Analogy with the Hypercube Structure:

Each coefficient og the generalized computational matrix M (denoting the function) can be positioned on an hypercube in dimension $n+1$(let us recall that each coefficient q_j denoting a monome is set on the node that has the same gray code as j). Let us consider a simple arithmetic operator $C(A, B)$ which takes to hypercubes $\mathcal{H}_1, \mathcal{H}_2$ ($\dim \mathcal{H}_1 = \dim \mathcal{H}_2$) in input and deliver the final hypercube $\mathcal{H} = A * \mathcal{H}_1 + B * \mathcal{H}_2$. This operation can be performed recursively. Indeed, remember that a hypercube in dimension n is constituted by 2 hypercubes of dimension $(n-1)$ in which corresponding vertices are linked (form the edges in the $(n+1)$ dimension). Let us denote $\mathcal{H}_1 = [\mathcal{H}_1^{(1)}\ \mathcal{H}_1^{(2)}]$ and $\mathcal{H}_2 = [\mathcal{H}_2^{(1)}\ \mathcal{H}_2^{(2)}]$. If $\dim \mathcal{H}_1 = 2$ then $\mathcal{H}_1 = [a_1\ b_1]$ ($\mathcal{H}_2 = [a_2\ b_2]$) and the cell $C(A, B)$ deliver $[A * a_1 + B * a_2\ A * b_1 + B * b_2]$. Otherwise, we perform the recursive process:

$$[A * \mathcal{H}_1^{(1)} + B * \mathcal{H}_2^{(1)}\ A * \mathcal{H}_1^{(2)} + B * \mathcal{H}_2^{(2)}]$$

[10] We allow to have a denominator which is null 0 at step i, $i \geq 2$.

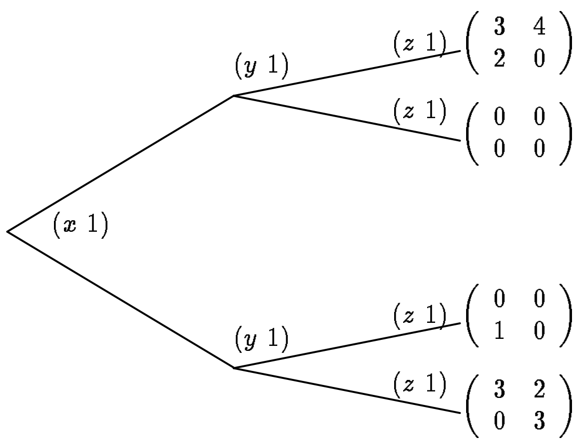


Figure 6: Representation of $f = \frac{3xyz+2xy+3z+y}{4xyz+2z+3}$.

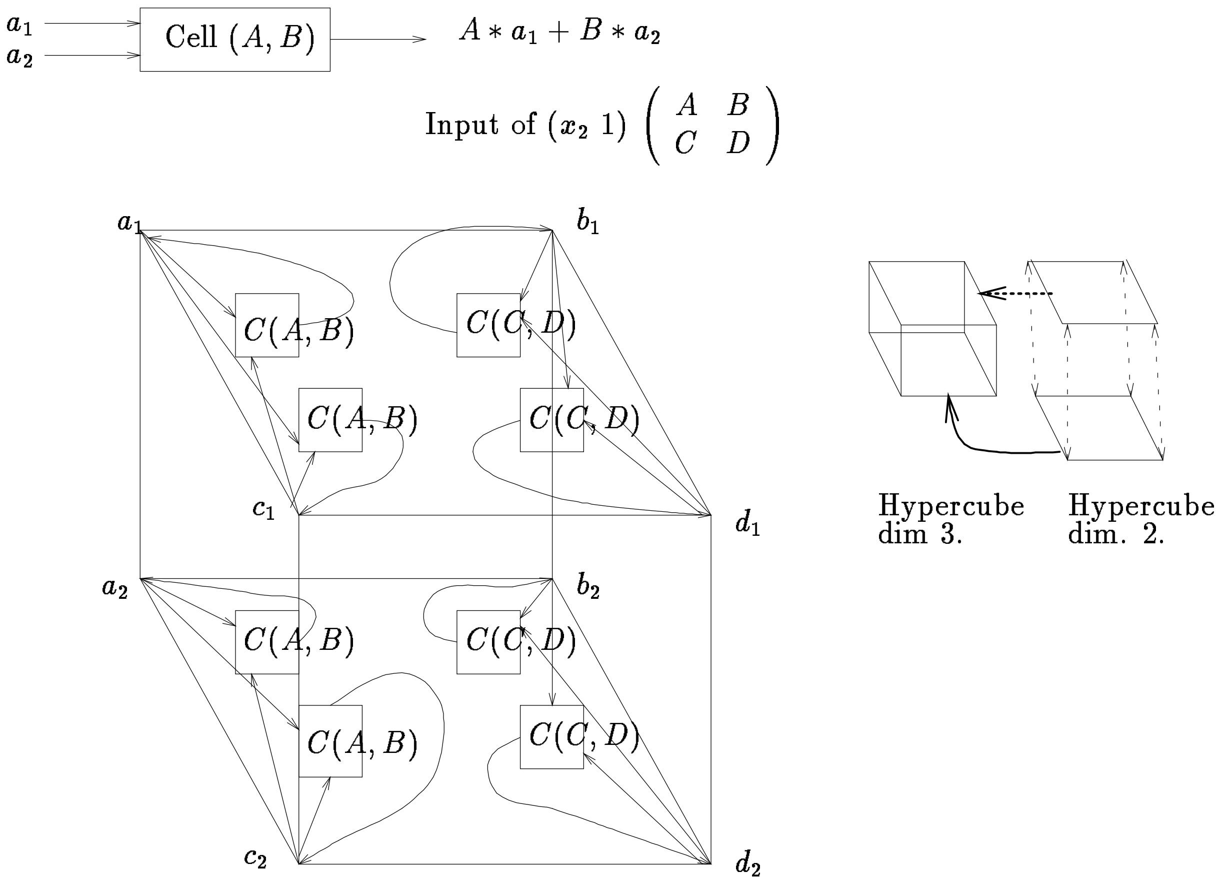


Figure 7: Analogy with the hypercube structure.

ALGORITHM -IX – BUILDING THE GRAY'S CODE

INPUT:
The dimension k of the hypercube to build and the 2^k data defining the function of $k-1$ variables.
Call: BuildHyperCube(k,0)

OUTPUT:
The hypercube

ALGORITHM:

Function BuildHyperCube(k,x) : HyperCube ;
k:integer;
x:integer;

begin
$\mathcal{H}$: HyperCube;

if $(k=0)$ **then** $\mathcal{H}$ = LoadData(x);
else $\mathcal{H}$ = BuildHyperCube($k-1, x$) $\otimes$ BuildHyperCube($k-1, x+2^{k-1}$);
return $\mathcal{H}$;
end;

3.7.1 Construction of Gray Code:

To develop an efficient algorithm based on the hypercube structure, we need to enumerate nodes correctly such that when an input on x_i is done, only particular **edges** will work. The most famous enumeration of the nodes of an hypercube is **Gray** code.

Property 8 *In dimension k, two nodes of the hypercubes are connected by an edge only if their binary representations differ with one bit. So that, if $n_1 = a_{k-1}\dots a_0$ and $n_2 = b_{k-1}\dots b_0$ then n_1 is connected to n_2 iff $XOR(n_1, n_2) = 2^j$ for some j $0 \le j \le k-1$. Furthermore, the edge $e = (n_1, n_2)$ is termed a j-edge connecting the vertices n_1 and n_2 in dimension j.*

We denote by $\oplus$ the XOR function. Hence we have $\text{XOR}(n_1, n_2) = n_1 \oplus n_2 = n_2 \oplus n_1$. The procedure to enumerate the vertices on th k-hypercube (and load data accordingly), is therefore easy:

Property 9 *If $n = b_{k-1}\dots b_0$ is a vertex belonging to a k-hypercube, we say n is even (odd) according to the $i-th$ dimension iff $b_i = 1$ (respectively $b_i = 0$). So that each j-edge connects an even vertex n_1 with an odd vertex n_2 (furthermore, n_1 differs from n_2 in the $(j+1)$ bits).*

3.7.2 Processing Input and Output in the Hypercube $\mathcal{H}$:

Inputs are now proceeded efficiently, considering the parity of its current partial quotient. Let us consider an input of variable x_i : Only the $(i+1)$-edges will be active. Taking into account the parity of the partial quotient, the following computations must be performed:

- **Even Parity:** It corresponds to the input matrix $\begin{pmatrix} 1 & 0 \\ a & 1 \end{pmatrix}$ in dimension i. The value of the even vertex will be changed into $v_e \leftarrow v_e + av_o$.
- **Odd Parity:** It corresponds to the input matrix $\begin{pmatrix} 1 & a \\ 0 & 1 \end{pmatrix}$ in dimension i. The value of the odd vertex will be changed into $v_o \leftarrow v_o + av_e$.

The output condition is also realized efficiently according to the first dimension (it can be seen as a variable x_0). If a can be delivered as output, the remaining coefficient of each node of the hypercube

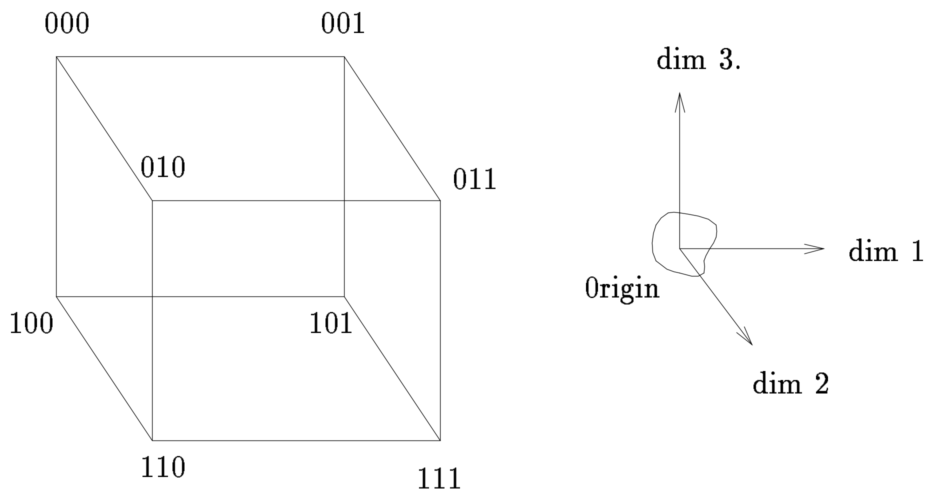


Figure 8: Construction of the **Gray** Code.

must be positive (hence, when the algorithm will be finished, we select the 1-edge respectively to the parity of variables $x_1, \ldots, x_n$.).

Illustration of the active edges are shown for a 3-hypercube (function of two variables x_1 and x_2).

3.7.3 The Butterfly ($\bowtie$) Operation:

We describe in this present part, how to perform a multiplication between a (2×2)-matrix M (which represents an input) and an hypercube $\mathcal{H}$ in dimension n according to the i-th dimension (variable x_i). M can be written as $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$. Then the operation to perform is:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}_i \bowtie \mathcal{H}$$

where $\bowtie$ denotes the multiplication.

If $\dim \mathcal{H} = 2$ (one variable x_1, $\mathcal{H} \equiv \begin{pmatrix} q_0 & q_2 \\ q_1 & q_3 \end{pmatrix}$) then we define the $\bowtie$ operation as

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}_i \bowtie \mathcal{H} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \times \begin{pmatrix} q_0 & q_2 \\ q_1 & q_3 \end{pmatrix}$$

this yields straightforwardly to the matrix

$$\begin{pmatrix} aq_0 + bq_1 & aq_2 + bq_3 \\ cq_0 + dq_1 & cq_2 + dq_3 \end{pmatrix}$$

Figure 11 explains why the $\bowtie$-operation is called "butterfly".

If $\dim \mathcal{H} = n$ is greater than 2 then $\mathcal{H}$ can be decomposed on to two hypercubes $\mathcal{H}_1$ and $\mathcal{H}_2$ according to the i-th dimension:

We eliminate all the i-edges (i.e. the edges that connect two vertices n_1 and n_2 such that $n_1 \oplus n_2 = 2^i$). We obtain two hypercubes of dimension $n-1$. We denote by $\mathcal{H}_1$ the hypercube that has all its edges even (i-even vertices i.e. if n is a vertex of $\mathcal{H}_1$ then $n \oplus 2^i = 0$) and by $\mathcal{H}_2$ the one that has all its edges odd (i-odd vertices i.e. if n is a vertex of $\mathcal{H}_1$ then $n \oplus 2^i = 2^i$).

The butterfly operation $\bowtie$ is defined by two steps: the initial step and the recursive step.

Homogeneous Step:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}_i \bowtie \mathcal{H} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \bowtie [\mathcal{H}_1 \overset{i}{-} \mathcal{H}_2] = [\begin{pmatrix} a & b \\ c & d \end{pmatrix}_0 \bowtie \mathcal{H}_1 \overset{i}{-} \begin{pmatrix} a & b \\ c & d \end{pmatrix}_0 \bowtie \mathcal{H}_2]$$

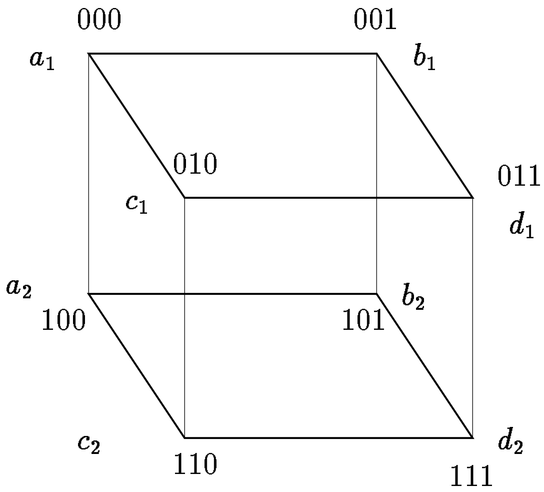


Figure 9: The 3-hypercube once the coefficients are *loaded*.

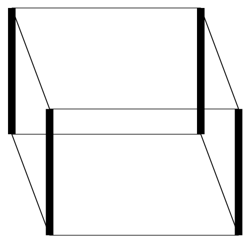
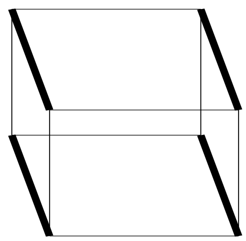
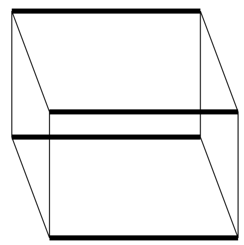
Proceed on Variable	Active edges are shown in bold
x_1	
x_2	
Output (x_0)	

Figure 10: Active edges according to the i-th axe.

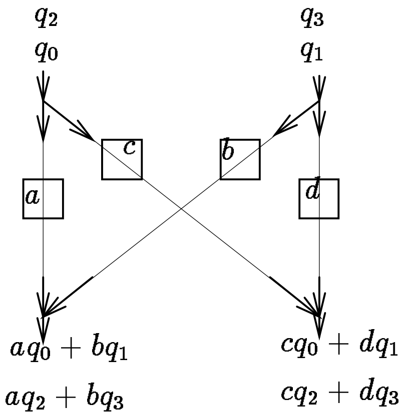


Figure 11: The butterfly operation ($\bowtie$).

Note that the input of the matrix M in dimension i is now 2 inputs of the same matrix M but in dimension 0 (this corresponds to the homogene operation step - it corresponds to a rotation of the hypercube).

We must know define how the butterfly operation is performed when the matrix must be entered in dimension 0. This is defined by the unique equation

Recursive Step:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}_j \bowtie \mathcal{H} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \bowtie [\mathcal{H}_1 \stackrel{i}{—} \mathcal{H}_2] = [\begin{pmatrix} a & b \\ c & d \end{pmatrix}_{s(j)} \bowtie \mathcal{H}_1 \stackrel{i}{—} \begin{pmatrix} a & b \\ c & d \end{pmatrix}_{s(j)} \bowtie \mathcal{H}_2] \tag{3}$$

where $s(j)$ is defined according to the first step (multiplication in the i-th dimension as follows:

$$s(j) = \begin{cases} j = i - 1 \rightarrow j + 2 \text{ jump the initial step} \\ j + 1 \text{ otherwise} \end{cases} \tag{4}$$

At the last recursive step (this step is defined by $\dim \mathcal{H} = 2$) $j = n$ ($i \neq n$) or $j = n - 1$ ($i = n$) and we perform the $\bowtie$ operation on 2^{n-1} hypercubes of dimension 2.

This observation has leaded to the general property:

Property 10 *Given a $(n+1)$ dimensional hypercube $\mathcal{H}$, performing $\begin{pmatrix} a & b \\ c & d \end{pmatrix}_i \bowtie \mathcal{H}$ corresponds to consider each i-edge $e = (n_1, n_2)$ of the hypercube $\mathcal{H}$ where n_1 (n_2) is the value of the even (respectively odd) vertex and perform in parallel with the $\bowtie$ operator.*

- $n_1 \leftarrow an_1 + bn_2$
- $n_2 \leftarrow cn_1 + dn_2$

Theorem 1 *If $\mathcal{H}$ is the hypercube coded the generalized functional matrix then entering an input matrix M from variable i corresponds to the butterfly operation $\mathcal{H} \leftarrow M_i \bowtie \mathcal{H}$.*

3.7.4 An example: $f(x_1, x_2, x_3) = \frac{x_1+x_2+x_3}{x_1x_2x_3}$.

To illustrate the previous concept, we develop in this part the steps generated when computing $f(x_1, x_2, x_3) = \frac{x_1+x_2+x_3}{x_1x_2x_3}$ with

- $x_1 = [0/2] = \frac{1}{2}$
- $x_2 = [0/3] = \frac{1}{3}$
- $x_3 = [4] = \frac{4}{1}$

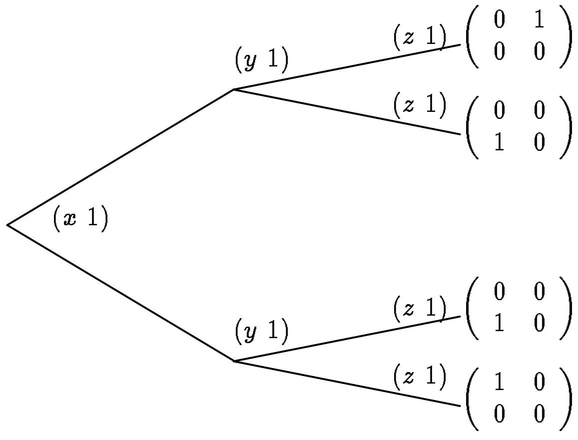


Figure 12: Generalized matrix of $f(x_1, x_2, x_3) = \frac{x_1+x_2+x_3}{x_1 x_2 x_3}$.

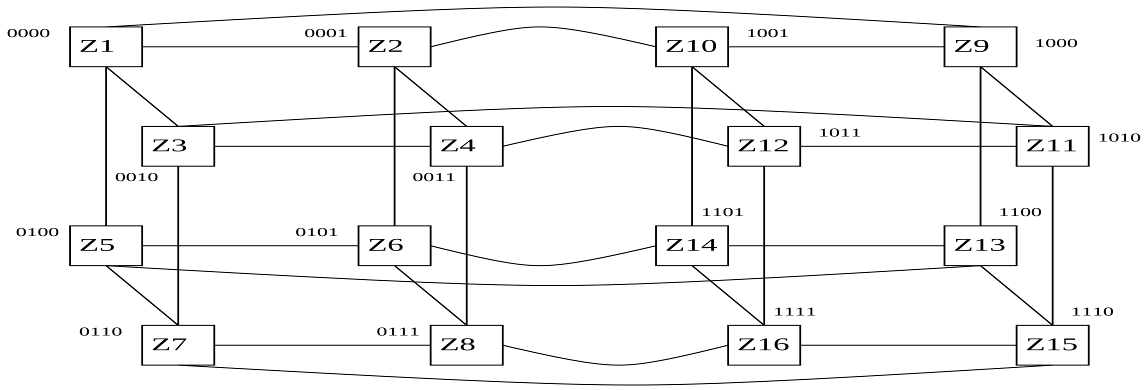


Figure 13: The 4-hypercube used to compute $f(.,.,.)$

The result is $f(\frac{1}{2}, \frac{1}{3}, 4) = \frac{29}{4} \equiv [7/4]$.

The first step is to compute the generalized matrix denoting the function (see figure 12).

As the function $f(.,.,.)$ has 3 variables, we must use a 4-hypercube. We represent it as usual where each vertex has an integer register which will be updated according the inputs (figure 13). Each step of the algorithm is detailed in the figure 14. According with the parity of each input variable (x_1:odd, x_2:odd, x_3:even), we choose the 0-edge[11]: $100 \equiv (1000, 1001)$. The remaining value is $\frac{5}{4}$ which can be written as its continued fraction format $[1/4]$. Hence, our final result is $[6/0/1/4] = [7/4]$ (the 0 partial quotient comes from the fact that our output process ends with the even parity). In reality, the algorithm reads the termination signal (∞) for x_1, x_2 and x_3. It, then output partial quotients until it can output ∞. Proceeding in that way, it delivers successively $0/1/4/\infty$.

3.7.5 Algorithm on the Hypercube:

From the materail developed before, the algorithm on the hypercube structure appears to be quite simple. It first loads the generalized matrix on the hypercube and then scans input until no more partial quotients are to be processed. Each time a partial quotient p is available form x_i, it performs the $a*p+b$-operation on all i-edges. If output can be produced, it delivers the corresponding partial quotient.

[11] We set the bit in the i-th dimension according to the parity of the i-th variable. If x_i ends with an odd parity, we set 0, otherwise 1.

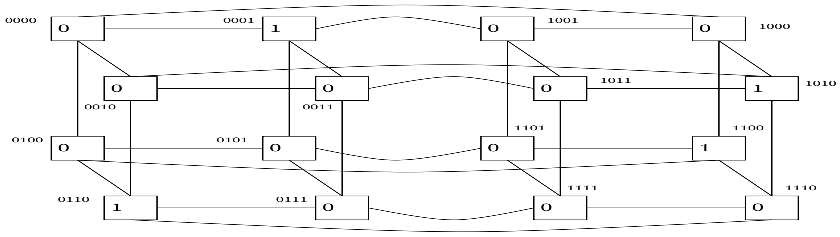
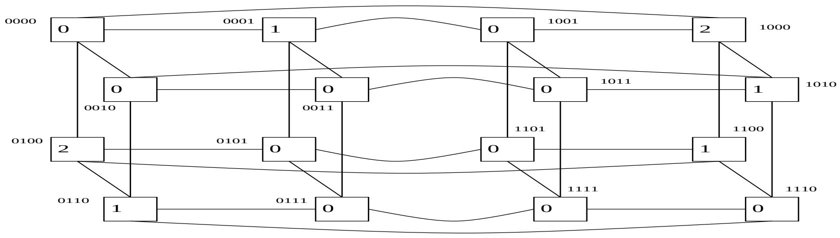
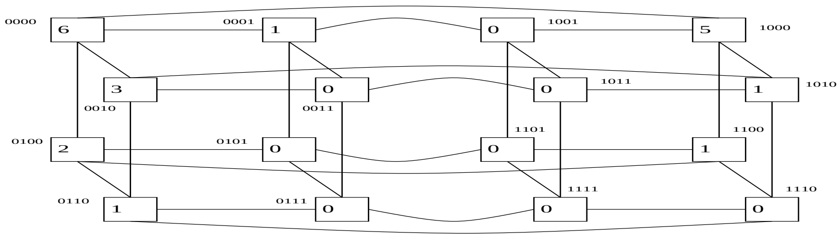
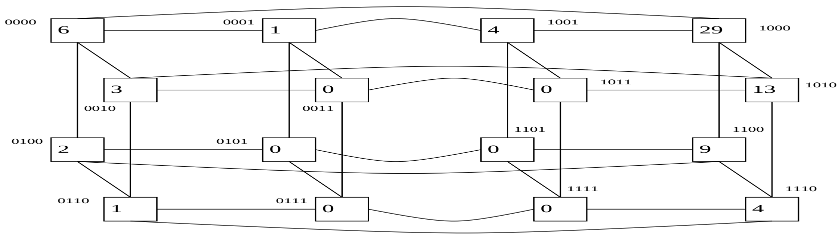
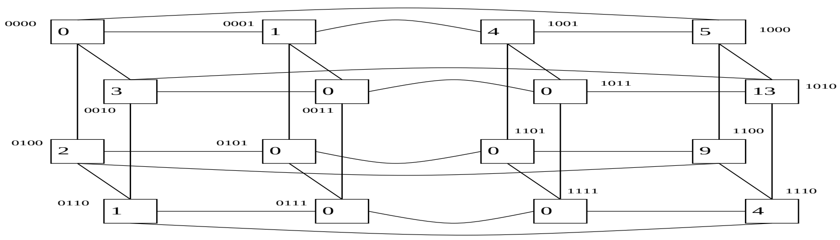
<table>
<tr><th>Events</th><th>4-hypercube coefficients</th></tr>
<tr><td>Load coefficients</td><td>
</td></tr>
<tr><td>Input 0 and 2 from x_1</td><td>
</td></tr>
<tr><td>Input 0 and 3 from x_2</td><td>
</td></tr>
<tr><td>Input 4 from x_3</td><td>
</td></tr>
<tr><td>Output 6</td><td>
</td></tr>
</table>

Figure 14: Steps generated when computing $f(\frac{1}{2}, \frac{1}{3}, 4)$

ALGORITHM -X – ALGORITHM ON THE HYPERCUBE

INPUT:
A generalized matrix representing the function of $x_1, \ldots, x_n$.

OUTPUT:
For a set of value $x_1 = value_1, \ldots, x_n = value_n$, the result $f(value_1, \ldots, value_n)$.

ALGORITHM:

BuildHypercube$(n+1)$;
while$(TRUE)$ **do**
begin
Process 1: Wait for partial quotient p from x_i;
Proceed p on each i-th edge
Process 2: If producing output is possible[12], choose the next partial quotient according to the number representation
end;

4 An Algorithm Based on the Hypercube to Compute $f(x_1, \ldots, x_n) = \frac{P_{num}(x_1,\ldots,x_n)}{P_{den}(x_1,\ldots,x_n)}$:

At the end of the previous section, we have introduced the analogy between the generalized matrices and the hypercubes. Processing input and output on the coefficient hypercubes have been detailed previously (see the $\bowtie$ operation) but the efficient computation of the output condition has been hidden. In this section, we first develop the decision hypercube which allows to compute efficiently the extremum values of the function f and we end with the bit level algorithm which is closed to a concrete implementation[13].

4.1 The Decision Hypercube

In the algorithm described previously, the bottleneck of the output stream comes from the output condition which required too much time when computed. To eliminate this latency, we introduce the notion of the *decision*[14] *hypercube*, which allows to determinate whether output can be produced or not by comparing the extremum points of the function. Let us consider the evaluation of a function of n variables as described before. The main idea is to determine $Frange(x_1, \ldots, x_n)$ the set of points generated when $x_1 \in]\delta_{1,0}; \delta_{2,0}[, \ldots, x_n \in]\delta_{n,0}; \delta_{n,1}[$. When a partial quotient p of x_i has been processed in the coefficient hypercube, the new variable $x'_i = \frac{1}{x_i - p}$ has its range in $]\frac{1}{\delta_{i,0}-p}; \frac{1}{\delta_{i,0}-p}[$. In particular, whenever the first partial quotient of variable x_i has been entered, we know that the tail of x_i which corresponds to the new variable lies between $\Delta =]1; \infty[$ and this interval Δ does not change after, so that $\delta_{i,0} = 1$ and $\delta_{i,1} = \infty$. The general notation of intervals allows as it will be demonstrated alter to mix various formats of numbers without changing the theory but only the bounds of variables.

Let us consider the non redundant continued fraction when $x_i \in]1; \infty[\ \forall\ i \in [1, n]$, i.e $\delta_{i,0} = 1$ and $\delta_{i,1} = \infty\ \ \forall i \in [1, n]$.

The $Frange$ point set is defined as the result of the function F on the generalized interval $I = \prod_{i=1}^{n}]\delta_{i,0}; \delta_{i,1}[$

$$Frange = \{f(x_1, \ldots, x_n)\ ,\ \forall\ x_1 \in]\delta_{1,0}; \delta_{1,1}[, \ldots, x_n \in]\delta_{n,0}; \delta_{n,1}[\}$$

or in the general interval notation

[13] A complete study of the implementation in hardware of the algorithm for two variables x and y has been analyzed in [6]

[14] *The decision hypercube contains information about the range where the next partial quotient belonging to the result lies. This notion was first introduce in a paper of* ***Kornerup*** *and* ***Matula*** *in [7].*

$$Frange = f(\prod_{i=1}^{n}]\delta_{i,0}; \delta_{i,1}[)$$

Hence, if at each step with a little amount of work, $Frange$ is known, we have the information that the result is between the bounding limits m and M of $Frange$ where m and M are the extreme values of $Frange$ ($m = \min Frange$ and $M = \max Frange$.

Observation 4 *If $[m, M] \cap \mathbb{N} = \{r\}$ then we can produce with certitude the output partial quotient r.*

4.1.1 Definition of the Decision Hypercube:

The function $f(x_1, \ldots, x_n)$ can be written in $f(x_1, \ldots, x_n) = \frac{P_{num}(x_1,\ldots,x_n)}{P_{den}(x_1,\ldots,x_n)}$, where P_{num} and P_den are both polynomial functions of n variables. We build the hypercube as $\mathcal{H}_1 \otimes \mathcal{H}_2$ where $\mathcal{H}_1$ is an hypercube in dimension n in which each vertex defined by its binary **Gray**'s code $b_n \ldots b_1$ contains the value of the evaluation of the function $P_{num}(T(b_1), \ldots, T(b_n))$ where $T(\cdot)$ is a simple function defined as follows:

$$T(bit_i) = \begin{cases} 1 \Rightarrow \delta_{i,1} \\ 0 \Rightarrow \delta_{i,0} \end{cases}$$

For example, the vertex v belonging to the hypercube $\mathcal{H}_1$ denoted by its **Gray**'s code $(01011)_2$ contains the value of $f(T(1), T(1), T(0), T(1), T(0)) = f(\delta_{1,1}, \delta_{2,1}, \delta_{2,0}, \delta_{3,0}, \delta_{4,1}, \delta_{5,0})$. We define in the same way the n dimensional hypercube $\mathcal{H}_2$ denoting the value of P_{den}.

These two hypercubes are linked to form the $\mathcal{H}$-hypercube in dimension $n+1$ with a new low bit b_0. It is worth noting that since the coefficients of both polynomes are integers and the bounds of each variable are also integers, the nodes of $\mathcal{H}$ are integers.

We term d_n the value stocked in the node having gray code n. It follows that

$$f(T(b_1), \ldots, T(b_n)) = \frac{P_{num}(T(b_1), \ldots, T(b_n))}{P_{den}(T(b_1), \ldots, T(b_n))} = \frac{d_{b_n \ldots b_1 1}}{d_{b_n \ldots b_1 0}}$$

At the beginning of the algorithm , when the coefficients are loaded on the coefficient hypercube, we compute (or load) the value $P_{num}(T(b_1), \ldots, T(b_n))$ and $P_{den}(T(b_1), \ldots, T(b_n))$ in the decision hypercube for each $b_1 \in \{0,1\}, \ldots, b_n \in \{0,1\}$ (each vertex of the hypercube $\mathcal{H}$). The extremum values, m and M, are computed[15] (or loaded). We denote by $index_m = (m_n \ldots m_1)_2$ ($index_M = (M_n \ldots M_1)_2$) the node wich gray code is $index_m$ (respectively $index_M$ and value $\frac{d_{2*index_m+1}}{d_{2*index_m}} = m$ (respectively $\frac{d_{2*index_M+1}}{d_{2*index_M}} = M$).

4.1.2 Updating the Decision Hypercube when an Input is Performed:

When processing input of a partial quotient from x_i, let say p, we operate the transformation on x_i:

$$x_i \rightarrow x_i = p + \frac{1}{x'_i}$$

where x'_i denote the new variable ($x'_i \in]1; +\infty[$). Hence, if our function was $f(x_1, \ldots, x_i, \ldots, x_n) = \frac{x_i N + N'}{x_i D + D'}$ where $N(\cdot), N'(\cdot), D(\cdot), D'(\cdot)$ are polynomial functions of $n-1$ variables $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$, we have the new function

$$f(x_1, \ldots, x'_i, \ldots, x_n) = \frac{x_i(N' + pN) + N'}{x_i(D' + pD) + D}$$

So that the new coefficients in the coefficient hypercube are (updated by the butterfly operator) :

- $N \leftarrow N' + pN$
- $N' \leftarrow N$
- $D \leftarrow D' + pD$

[15] In reality, these values are only computed when the function is well-defined. A function is term well-defined if it does not admit a point where the result of the function is undefined. This case can occur only when both numerator and denominator are nulls $\frac{0}{0}$ but $\frac{\infty}{0}$ is ∞

- $D' \leftarrow D$

and the updated value in the decision hypercube are

- for the minimum bounding $f_n(T(b_1),\ldots,T(b_{i-1}),T(0),\ldots,T(b_n)) = \frac{(N'+pN)\delta_{i,0}+N}{(D'+pD)\delta_{i,0}+D}$
- for the maximum bounding $f_n(T(b_1),\ldots,T(b_{i-1}),T(1),\ldots,T(b_n)) = \frac{(N'+pN)\delta_{i,1}+N}{(D'+pD)\delta_{i,1}+D}$

If we are working with $\delta_{i,0} = 1, \delta_{i,1} = \infty \ \forall\ i \in [\![1,n]\!]$ we have

- for the minimum bounding $f_n(T(b_1),\ldots,T(b_{i-1}),T(0),\ldots,T(b_n)) = \frac{(N'+pN)\delta_{i,0}+N}{(D'+pD)+D}$
- for the maximum bounding $f_n(T(b_1),\ldots,T(b_{i-1}),T(1),\ldots,T(b_n)) = \frac{(N'+pN)}{(D'+pD)}$

Re-writing, the last equation in matrix form, we obtain

$$\begin{array}{l} \text{m bounding}\rightarrow \\ \text{M bounding}\rightarrow \end{array} \begin{pmatrix} (N'+pN)\delta_{i,0}+N & (D'+pD)\delta_{i,0}+D \\ (N'+pN)\delta_{i,1}+N & (D'+pD)\delta_{i,1}+D \end{pmatrix} = \begin{pmatrix} p\delta_{i,0} & \delta_{i,0}+1 \\ p\delta_{i,1} & \delta_{i,1}+1 \end{pmatrix} \times \begin{pmatrix} N & D \\ N' & D' \end{pmatrix}$$

and with $\delta_{i,0} = 1, \delta_{i,1} = \infty \ \forall\ i \in [\![1,n]\!]$

$$\begin{pmatrix} N'+pN+N & D'+pD+D \\ N'+pN & D'+pD \end{pmatrix} = \begin{pmatrix} p+1 & 1 \\ p & 1 \end{pmatrix} \times \begin{pmatrix} N & D \\ N' & D' \end{pmatrix}$$

This generalized operation (dealing with a (2×2)-matrix and an hypercube) has been described in the butterfly ($\bowtie$) process. Each time, a new partial quotient is processed, the decision hypercube is also updated accordingly. So that, if the index of m and M can also be updated, the output condition is known in constant time ($O(1)$).
But let us examine the output process before!

4.1.3 Updating the Decision Hypercube when an Output is Performed:

If r is the next partial quotient to produce in output, our function can be written as follows:

$$f(x_1,\ldots,x_n) = r + \frac{1}{f_n(x_1,\ldots,x_n)}$$

When producing output we obtain the new function coded in the coefficient hypercube $f_n(x_1,\ldots,x_n) = \frac{1}{f(x_1,\ldots,x_n)-r}$. If $f(x_1,\ldots,x_n) = \frac{x_1N+N'}{x_1D+D'}$ then the new function is

$$f_n(x_1,\ldots,x_n) = \frac{x_1D+D'}{x_1(N-Dr)+N'-D'r}$$

The analogy with the coefficient cube can still be done with the right-side product of matrix:

$$\begin{pmatrix} D & N-Dr \\ D' & N'-D'r \end{pmatrix} = \begin{pmatrix} N & D \\ N' & D' \end{pmatrix} \times \begin{pmatrix} 1 & 1 \\ 0 & -r \end{pmatrix}$$

This operation (right-side multiplication) has also been explained in the previous sections (see the butterfly operation $\bowtie$).

4.1.4 Updating the Index of m and M:

The purpose of this part is to explain how it is possible to update the index of both extremum values with simple process when an input or output is processed. We begin by a nice property linking both indices ($index_m$ and $index_M$).

Property 11 *If $index_m = (m_1 \dots m_n)_2$ is the index of the minimum(whenever the function is well defined) then we have $index_M$ that is defined as the complementary binary number:*

$$index_M = \overline{index_m} = \overline{(m_1 \dots m_n)_2} = (\overline{m_1 \dots m_n})_2$$

It follows from property 11 that only one index has to been known. This property can be prooved using the well-definedness of f and the monotony following variables $x_1, \dots, x_n$.

Two cases can occur when the index $index_m$ (and by property 11 also $index_M$) must be updated:

- Input of partial quotient p from variable x_i: The function f can be factorized following x_i, which leads straightforwardly to

  $$f(x_1, \dots, x_n) = \frac{x_i N + N'}{x_i D + D'}$$

  When we input a partial quotient p from x_i we operate the transformation

  $$x_i \to a + \frac{1}{x_i'}$$

  where x_i' is the new variable lying in the new interval $\Delta_i =]\delta_{i,0}; \delta_{i,1}[$ where both $\delta_{i,0}$ and $\delta_{i,1}$ have been updated following the same transformation.

  This transformation can also be expressed when considering the new variable x_i:

  $$x_i' \to \frac{1}{x_i - p}$$

  which can be decomposed in two steps following the scheme;

  $$x_i \overset{\text{translation}}{\longrightarrow} x_i - p \overset{\text{inversion}}{\longrightarrow} \frac{1}{x_i - p}$$

  Regarding the trace of the function in $I = \prod_{i=1}^{n}]\delta_{i,0}; \delta_{i,1}[$, the transformation acts as follows

  - Step 1 : Substract p from x_i. This corresponds to translation of the tracing to the "left" (see figure 15). Since the function is monotonic regarding x_i (increasing or decreasing) the index of both the minimum value and the maximum value have not changed.
  - Step 2 : Once step 1 has been processed, we operate the inversion $x_i \to \frac{1}{x_i}$. So that if the function was increasing regarding x_i it becomes decreasing and vice-versa (see bottom of the figure 15). We have changed both index by flipping the bit m_i into $\overline{m_i}$ and M_i into $\overline{M_i}$.

- Output partial quotient r (from x_0): When we output a partial quotient r from the coefficient hypercube, we perform also accordingly the following diagram:

  $$f(x_1, \dots, x_n) \overset{\text{translation}}{\longrightarrow} f(x_1, \dots, x_n) - r \overset{\text{inversion}}{\longrightarrow} \frac{1}{f(x_1, \dots, x_n) - r}$$

  But if we look at the trace of f each value has been first decreased by r so that both the position where *min* and *max* are does not change and after we perform the inversion so that the position where min stands, contains now the maximum value and vice-versa(see figure 16).

4.2 The Bit Level Algorithm:

We develop here how partial quotient can be entered and output piecewise at the bit level. For that purpose, we describe how the algorithm works when partial quotient are entered bitwise. We then introduce the general tables for the LCF and RPQ formats.

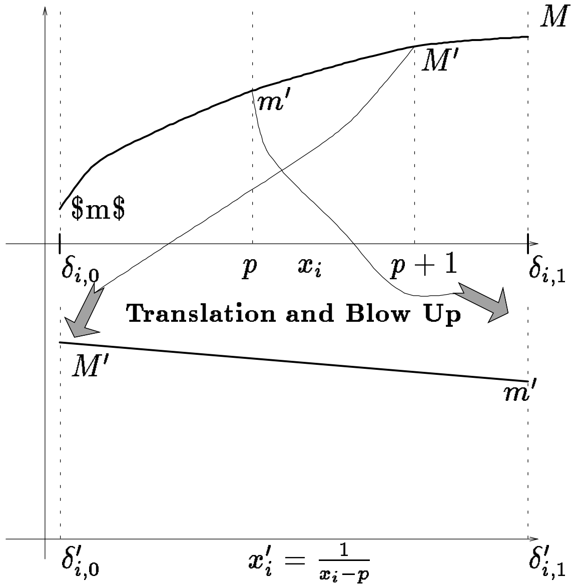


Figure 15: Transformations of *Frange* when an input on x_i is processed.

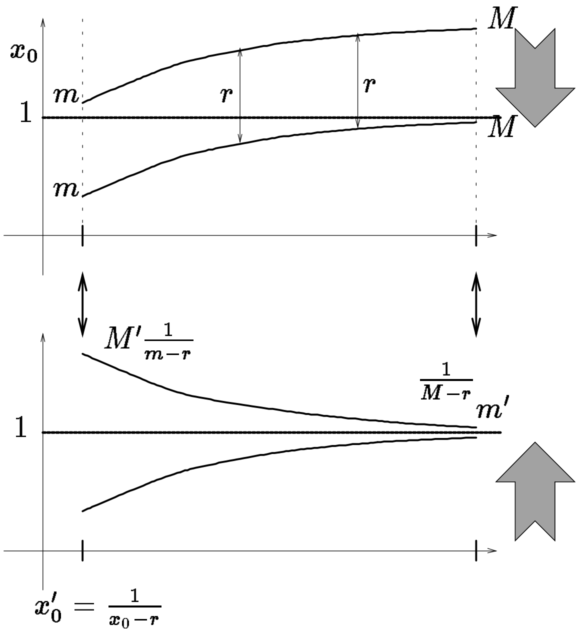


Figure 16: Transformations on *Frange* when an output is processed.

4.2.1 General Principle:

Since we can can factorize with products of matrices the partial quotient according to the bit level format, the input process does not change. It is worth noting that the coefficients of the matrices are simple and belong to the set $\mathcal{S} = \{-\frac{1}{2}, 0, \frac{1}{2}, 1, 2\}$. So that when an input matrix is processed in the coefficient hypercube, only shift/add/multiplication operations are used. As inputs are processed, *Frange* the image set of f is updated. We want also to deliver output piecewise according to a selected format with the eternal condition that the result lies in $]1;\infty[$ (partial quotient interval). Backtracking this condition for each factorizing matrix M, we obtain the corresponding intervals Δ_M so that if $Frange \in \Delta_M$ then we can output M (indeed we know that the result lies in $]1;\infty[$).

$$Frange \xrightarrow{M} Frange'$$

$$\Delta_i \longrightarrow \Delta_f$$

where Δ_i is the initial interval defined by *Frange* and Δ_f is the new interval defined when we output the matrix M.

Denoting by the $\otimes$-sign the output operation, we can write

$$\Delta_f = \Delta_i \otimes M$$

but Δ_f is known, so that for each different M, we compute the associated interval

$$\Delta_b = \Delta_f \otimes M^{-1}$$

We describe now the results of the computation for the LCF and RPQ formats. The tables can also be found in[16].

4.2.2 Using the LCF Format:

The lexicographic continued fraction format has been described in the first section (pages 1.4–1.4). We now give the automate that deliver the sequence of matrices denoting the partial quotient and the different output intervals.

Given a LCF representation of $x = \frac{p}{q}, x \in \mathbb{Q}$, the automate (see table 1) delivers the matrix representation of the continued fraction denoted x.

In the tables denoting the automate, n stands for the partial quotient a_n so that in the beginning n is null. The column where stands the matrices describe which transformation must be performed on the coefficient hypercube. For example, $\frac{3}{5} = [0/1/1/2]$ has for LCF code the sequence of bits $(1010011)_2$ where the first bit denote its sign (1:positive) and the second its reciprocity $(0 : \frac{p}{q} < 1)$.

When using the LCF format as output, at the beginning we set the state of the output to A. If *Frange* is in an interval (as defined in table 1) then we output the corresponding matrix (or bit) and go to the new state.

Observation 5 *If the format we use allow redundancy then there exists at least one state S where some of its intervals overlap*

4.2.3 Using the RPQ Format:

We give here in table 2 the automate characterizing the RPQ format. Please note that m denotes the $\overline{1}$ bit.

For example, we can see that the RPQ format is redundant if we take the inserection of intervals:

- State A: $]-2;2[\cap]-4;0[\neq\emptyset$
- State B: $]-1;\infty[\cap]-2;2[\neq\emptyset$

State	Interval	Bit Digit	Matrix	New n	New State
start	–	–	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	0	A
A	$[0;\infty[$	1	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	0	B
(*sign*)	$]-\infty;0[$	0	$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$	0	B
B	$[1;\infty[$	1	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	0	C
(*reciproc.*)	$[0;1[$	0	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	0	E
C	$[2;\infty[$	1	$\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$	$n+1$	C
(Unary mode normal)	$[1;2[$	0	$\begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}$	n	$D_{n\neq 0}, E_{n=0}$
D	$[2;\infty[$	0	$\begin{pmatrix} \frac{1}{2} & 0 \\ 0 & 1 \end{pmatrix}$	$n-1$	$D_{n\neq 0}, E_{n=0}$
(Binary mode normal)	$[1;2[$	1	$\begin{pmatrix} \frac{1}{2} & 0 \\ -\frac{1}{2} & 1 \end{pmatrix}$	$n-1$	$D_{n\neq 0}, E_{n=0}$
E	$[2;\infty[$	0	$\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$	$n+1$	E
(Unary mode reverse)	$[1;2[$	1	$\begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}$	n	$F_{n\neq 0}, C_{n=0}$
F	$[2;\infty[$	1	$\begin{pmatrix} \frac{1}{2} & 0 \\ 0 & 1 \end{pmatrix}$	$n-1$	$F_{n\neq 0}, C_{n=0}$
(Binary mode reverse)	$[1;2[$	0	$\begin{pmatrix} \frac{1}{2} & 0 \\ -\frac{1}{2} & 1 \end{pmatrix}$	$n-1$	$F_{n\neq 0}, C_{n=0}$

Table 1: The automate characterizing the LCF code.

State	Interval	Bit Digit	Matrix	New n	New State
start	–	–	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	0	A
A	$]-1;1[$	0	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	0	A
(Unary)	$]0;4[$	1	$\begin{pmatrix} 0 & 2 \\ 1 & -2 \end{pmatrix}$	$n+1$	B
	$]-4;0[$	m	$\begin{pmatrix} 0 & 2 \\ 1 & 2 \end{pmatrix}$	$n+1$	B
	$]-2;2[$	u	$\begin{pmatrix} 1 & 0 \\ 0 & 3 \end{pmatrix}$	$n+1$	A
B	$]1;\infty[$	1	$\begin{pmatrix} \frac{1}{2} & 0 \\ -\frac{1}{2} & 1 \end{pmatrix}$	$n-1$	$B_{n\neq 0}, A_{n=0}$
(Binary)	$]-\infty;-1[$	m	$\begin{pmatrix} \frac{1}{2} & 0 \\ \frac{1}{2} & 1 \end{pmatrix}$	$n-1$	$B_{n\neq 0}, A_{n=0}$
	$]-2;2[$	0	$\begin{pmatrix} \frac{1}{2} & 0 \\ 0 & 1 \end{pmatrix}$	$n-1$	$B_{n\neq 0}, A_{n=0}$

Table 2: The automate characterizing the RPQ code.

5 The Szekeres Multidimensional Continued Fraction:

In this present section, we formalize the algorithm described by *Szekeres* in term of products of simple inversible matrix. We then apply these transformations in the computation of complex functions. **G. Szekeres** introduced new algorithms for generating higher dimensional analogue of the ordinary continued fraction expansion of a single real number. The algorithm described in the remaining, takes a real k-vector $(\alpha_1, \alpha_2, \dots, \alpha_k)$ and deliver a $(k+1)$-vector in $\mathbb{N}^{k+1}$: $(a_1, a_2, \dots, a_k, b)$ where $\frac{a_1}{b} \to \alpha_1$, $\frac{a_2}{b} \to \alpha_2$, $\frac{a_n}{b} \to \alpha_n$ (if$\forall\ i \in [\![1, k]\!], \alpha_i \in \mathbb{Q}$ then the algorithm ends with $\alpha_i = fraca_ib$). **Szekeres** conjectured, on the basis of extensive computation, that the intermediate vectors produced by the algorithm contains the best simultaneous rational approximations to $(\alpha_1, \dots, \alpha_k)$.

5.1 Best and Good Rational Approximation to a real:

As the continued fraction contained in $]0,1]$ are the reciprocal of those contained in $[1, \infty[$, we suppose that $0 < \alpha_i < 1\ \forall\ i \in [\![1, k]\!]$. It is known from the *Theory of Numbers* that each number can be written in a continued fraction (which is infinite when it is an irrational).

$$\alpha = [a_1/a_2/a_3/...]\ , a_i > 0 \tag{5}$$

We say $\frac{P}{Q}$ is a best rational approximation to α when $|Q\alpha - P| < |q\alpha - p|$ for all rationals $\frac{p}{q}$ with $0 < q < Q$. The sequence of convergents $\frac{p_n}{q_n} = [a_1/\dots, a_n]$ is then precisely the sequence of best approximations to α.

We term good rational approximation to α when $|Q\alpha - P| < \frac{1}{Q}|$. In that case, the sequence $\frac{rp_{n+1}+p_n}{rq_{n+1}+q_n}$ with $1 \leq r < a_{n+2}; n \in \{0, 1, 2, ...\}$ of intermediate convergents contains the sequence of good approximations to α.

5.2 Good and Best Rational Approximations of a real k-vector:

We can extend the previous notion to higher dimension. Given a α k-vector: $\alpha = (\alpha_1, \dots, \alpha_k)$, we want simultaneous rational approximations $\frac{P_1}{Q}, \dots, \frac{P_k}{Q}$ to $\alpha_1, ..., \alpha_k$ with the same common denominator Q. We say $(P_1, P_2, \dots, P_k, Q)$ is a best approximation if

$$\max_{1 \leq i \leq k} \{|Q\alpha_i - P_i|\} < \max_{1 \leq i \leq k} \{|q\alpha_i - p_i|\} \tag{6}$$

for all the rational k-tuples $\frac{p_1}{q}, \dots, \frac{p_k}{q}$ with $0 < q < Q$.

We say that $(P_1, P_2, \dots, P_k, Q)$ is a good approximation if

$$\max_{1 \leq i \leq k} \{Q^{\frac{1}{k}} |Q\alpha - P_i|\} < 1 \tag{7}$$

In the following section, we describe an algorithm that takes a real vector and gives its best rational approximation where the denominator of each rational is the same.

5.3 Algorithm to Compute Approximations to a Real k-vector:

We first suppose without loss of generality that the real vector $\alpha = (\alpha_1, \dots, \alpha_k)$ to approximate is constrained by

$$1 > \alpha_1 > \alpha_2 > \dots > \alpha_k > 0 \tag{8}$$

Indeed, we can memorize the sign of α_i first, so that we input only non negative real numbers. If one component is null, then the problem is to approximate a $(k-1)$ real vector. We can also before processing normalize the real vector by $n = \frac{1}{\max_i\{a_i\}+1}$ so that the resulting vector $\alpha' = (\frac{\alpha_1}{n}, \dots, \frac{\alpha_k}{n})$ satisfy the input condition.

We must also assume that $1, \alpha_1, \alpha_2, \dots, \alpha_k$ are linearly independent over the rationals, i.e.:

$$a_0 + \sum_{i=1}^{k} a_i \alpha_i = 0 \Rightarrow \sum_{i=0}^{n} |a_i| = 0 \tag{9}$$

where $a_i \in \mathbb{Q}$.

The continued k-fraction, which is denoted by $[b_1/b_2/\ldots]$ is a sequence (finite, if the real vector contains no irrational and infinite otherwise) of positive integers b_i with which we associate the k-tuple $(\alpha_1,\ldots,\alpha_k)$ of real numbers via the algorithm described below:

Let us define s_m the partial sum of the first m results:

$$s_m = \sum_{i=1}^{m} b_i \tag{10}$$

We define a boolean function $\epsilon(n)$, $n \geq 1$ as follow:

$$\epsilon(n) = \begin{cases} & \text{if } n = s_m \text{ for some } m \\ & \text{if } n \neq s_m \text{ for any } m \end{cases} \tag{11}$$

It is worth noting, as each term b_i is strictly positive, $(s_m)_{n\in\mathbb{N}}$ form an increasing suite. And then the determination of $\epsilon(.)$ insure the determination of $b_1, b_2, \ldots$. In particular, if $b_1, b_2, \ldots, b_r$ is a finite sequence, we then have $\epsilon(n) = 0$ for all $n > s_r$.

$$000\ldots0 \underbrace{1^{n_1}}_{n_1=b_1} 000\ldots000 \underbrace{1^{n_2}}_{n_2=\underbrace{b_1}_{n_1}+b_2} \ldots\ldots\ldots00 \underbrace{1^{n_r}}_{n_r=s_r} 000000000000 \rightarrow \ldots\ldots$$

Proofs and hints of the algorithm are not given. The reader can obtain these results in the original paper of **G. Szekeres** or a paper written by **Cusic**.

First, we define a set of integral vectors A:

$$A(n,j) = (A^{(1)}(n,j), \ldots, A^{(k)}(n,j)) \ , \forall\ j \in [\![0,k]\!] \tag{12}$$

where $A(n,j)$ represents a $(k+1)$ vector at step n in the algorithm.

We then define a sequence of positive integer:

$$B = (B(n,0), \ldots, B(n,i), \ldots, B(n,k)) \ , n \in \{0,1,2,\ldots\} \tag{13}$$

Our algorithm consists in updating both A and B vectors according to the real α vector.

Initial assignment:

we assign initially:

$$A^{(i)}(0,j) = \begin{cases} 0, & 0 \leq j < i \leq k \\ 1, & 0 \leq i \leq j \leq k \end{cases} \tag{14}$$

and

$$B(0,j) = A^{(0)}(0,j) = 1, \ \forall\ j \in [0,k] \tag{15}$$

Matrix Form

In matrix notation, we define $A(n)$ as $(k+1)\times(k+1)$ matrix and $B(n)$ as a $(k+1)$-vector at step n.

$$A(n) = \begin{pmatrix} A^{(0)}(n,0) & \ldots & A^{(0)}(n,k) \\ \vdots & \ddots & \vdots \\ A^{(k)}(n,0) & \ldots & A^{(k)}(n,k) \end{pmatrix} \tag{16}$$

and

$$B(n) = (B(n,0), \ldots, B(n,k)) \tag{17}$$

with the initial conditions

$$A(0) = \begin{pmatrix} 1 & \dots & \dots & 1 \\ 0 & \ddots & & 1 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & 1 \end{pmatrix} \tag{18}$$

$A(0)^{-1}$ is therefore:

$$A(0)^{-1} = \begin{pmatrix} 1 & -1 & 0 & \dots & 0 \\ 0 & 1 & -1 & \dots & 0 \\ \vdots & & \ddots & -1 & \vdots \\ 0 & \dots & 0 & 1 & -1 \\ 0 & 0 & \dots & \dots & 1 \end{pmatrix} \tag{19}$$

$$B(0) = (1\ 1\ \dots\ 1) \tag{20}$$

The new state described by $B(n+1)$ and $A(n+1)$ is achieved by performing a transformation denoted by $T(\epsilon(n+1), \mu(n))$ where $\mu(n)$ is an integer bounded by $0 \leq \mu(n) \leq k$ determined in step n by a procedure which is detailed later. Note that this way of proceeding is inductive.

Transformation on A:

$$A^{(i)}(n+1, 0) = (1 - \epsilon(n+1))A^{(i)}(n, 0) + \epsilon(n+1)A^{(i)}(n, \mu(n)) \ , i \in [\![0, k]\!] \tag{21}$$

$$A^{(i)}(n+1, \mu(n)) = A^{(i)}(n, 0) + A^{(i)}(n, \mu(n)) \ , i \in [\![0, k]\!] \tag{22}$$

$$A^{(i)}(n+1, j) = A^{(i)}(n, j) \ , 1 \leq j \leq k, j \neq \mu(n) \tag{23}$$

Matrix Form

The transformation associated is denoted by $T(\epsilon(n+1), \mu(n))$ where $\epsilon(n+1)$ is a boolean value and $\mu(n)$ locates the transformation.

By denoting I the identity matrix and e_i, the $(k+1)$ vector, where only i-th component as value 1 and the others 0, it follows:

$$T(\epsilon(n+1), \mu(n)) = T(\epsilon, \mu) = I + \mu(e_\mu - e_0)e_0^T + e_0 e_\mu^T \tag{24}$$

$$T(\epsilon, \mu) = \begin{pmatrix} 1-\epsilon & 0 & \dots & 1 & \dots & 0 \\ 0 & 1 & \dots & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & & \vdots \\ \epsilon & 0 & \dots & 1 & \dots & 0 \\ \vdots & \vdots & & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & \dots & 1 \end{pmatrix}$$

Computations generated by these transformations are very simple since ϵ is a boolean value.

- $\epsilon = 0$

$$T(0, \mu) = \begin{pmatrix} 1 & 0 & \dots & 1 & \dots & 0 \\ 0 & 1 & \dots & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & & \vdots \\ 0 & 0 & \dots & 1 & \dots & 0 \\ \vdots & \vdots & & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & \dots & 1 \end{pmatrix}$$

- $\epsilon = 1$

$$T(1,\mu) = \begin{pmatrix} 0 & 0 & \dots & 1 & \dots & 0 \\ 0 & 1 & \dots & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & & \vdots \\ 1 & 0 & \dots & 1 & \dots & 0 \\ \vdots & \vdots & & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & \dots & 1 \end{pmatrix}$$

The rank of these transformations are $\text{rank}(T(\epsilon,\mu)) = k+1$.

Transformation on B:

$$B(0,j) = A^{(0)}(n+1,j) \tag{25}$$

$B(n)$ is always the first row of $A(n)$
We call the $a(n)$ k-vector the n-th approximation fraction to α defined by

$$a(n) = \left(\frac{A^{(1)}(n,\mu(n-1))}{B(n,\mu(n-1))}, \dots, \frac{A^{(k)}(n,\mu(n-1))}{B(n,\mu(n-1))}\right) \tag{26}$$

which is written by factorizing out the common denominator,

$$a(n) = \frac{1}{B(n,\mu(n-1))}(A^{(1)}(n,\mu(n-1)), \dots, A^{(k)}(n,\mu(n-1))) \tag{27}$$

To complete our algorithm, we must specify the computation of $\mu(n)$ and $\epsilon(n+1)$. For that purpose, we define the set of $(k+1)$ positive real numbers $\Gamma(n) = \{\gamma_{nj} : 0 \le j \le k\}$, $n \in \{0,1,2,\dots\}$ where

$$\Gamma(0) = \begin{cases} \gamma_{00} = 1 - \alpha_1 \\ \gamma_{0j} = \alpha_j - \alpha_{j+1} \text{ , } j \in [\![1, k-1]\!] \\ \gamma_{0k} = \alpha_k \end{cases} \tag{28}$$

Note that since we have ensured $1 > \alpha_1 > \dots > \alpha_j > \alpha_{j+1} > \dots > \alpha_0 > 0$, $\gamma_{0j} > 0$, $\forall\, j \in [\![0,k]\!]$.
Then, $\epsilon(n+1)$ is determined as follow:

$$\epsilon(n+1) = \begin{cases} 0 \text{ if } \gamma_{n0} > \gamma_{n\mu(n)} \\ 1 \text{ if } \gamma_{n0} < \gamma_{n\mu(n)} \end{cases} \tag{29}$$

We express now the inductive process which define the transition from $\Gamma(n)$ to $\Gamma(n+1)$:

$$\gamma_{(n+1)0} = (1 - 2\epsilon(n+1))(\gamma_{n0} - \gamma_{n\mu(n)}) \tag{30}$$
$$\gamma_{(n+1)\mu(n)} = \epsilon(n+1)\gamma_{n0} + (1 - \epsilon(n+1))\gamma_{n\mu(n)}) \tag{31}$$
$$\gamma_{(n+1)j} = \gamma_{nj} \text{ , for } 1 \le j \le k,\ j \ne \mu(n) \tag{32}$$

Matrix Form

Let us consider $\Gamma(n)$ as a $(k+1)$-vector, namely:

$$\Gamma(n) = \begin{pmatrix} \gamma_{n0} \\ \vdots \\ \gamma_{nk} \end{pmatrix}$$

The following transformation depending on ϵ must be performed in order to obtain $\Gamma(n+1)$:

$$T_\Gamma(\epsilon) = \begin{array}{cc} & \begin{array}{cccccc} & & & \mu & & \\ & & & \downarrow & & \end{array} \\ \begin{array}{c} \\ \\ \\ \mu \rightarrow \\ \\ \\ \end{array} & \begin{pmatrix} 1-2\epsilon & 0 & \dots & 2\epsilon - 1 & \dots & 0 \\ 0 & 1 & & 0 & \dots & 0 \\ \vdots & & \ddots & \vdots & \vdots & \vdots \\ \epsilon & \dots & 0 & 1-\epsilon & \dots & 0 \\ \vdots & & & 0 & \ddots & \vdots \\ 0 & \dots & \dots & \dots & 0 & 1 \end{pmatrix} \end{array} \tag{33}$$

$\Gamma(1)$ can be rewritten as $I_\Gamma \times \alpha^T$ where I_Γ is a $(k+1) \times (k+1)$ matrix defined by

$$I_\Gamma = \begin{pmatrix} 1 & -1 & 0 & \dots & \dots & 0 \\ 0 & 1 & -1 & & (0) & \vdots \\ \vdots & & \ddots & \ddots & & \vdots \\ \vdots & (0) & & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 0 & 1 & -1 \\ 0 & \dots & \dots & \dots & 0 & 1 \end{pmatrix} \tag{34}$$

In order to complete the description of the algorithm, we must now describe the computation of $\mu(n)$. But before, we recall the reader on the $\prec$ relation between two vectors. Given a, b two vectors (which we order increasingly by a permutation of their components, in a', b') , we say $a \prec b$ if there exists an index j such as $a'_j < b'_j$ and $\forall i \in [1, j-1]$, $a'_i = b'_j$.

We define the $(k+1)$-vector

$$V(n,j) = \frac{A(n,j)}{B(n,j)} - \frac{A(n,0)}{B(n,0)} \tag{35}$$

Then $\mu(n)$ (recall that $1 \leq \mu(n) \leq k$) is defined to be the largest integer t such that for every i , $1 \leq i \leq t$, we have

$$V(n,i) \prec V(n,t) \;\text{ or } V(n,j) = V(n,h) \tag{36}$$

It follows straightforwardly from the definition of $\mu(.)$ that $\mu(0) = k$.

It must be noted that if α is a rational vector (embedded in $\mathbb{R}$ space), then the algorithm will stop at some step n. Indeed, we have therefore

$$\Gamma(n) = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} = (0) \tag{37}$$

$$\forall\, t > n,\ \epsilon(t) = 0 \tag{38}$$

In the general case, where at least one of the components α_i is irrational, *Szekeres* has proved that

$$\lim_{n \to \infty} \gamma_{nj} = 0 \;, \forall\, j \in [\![0, k]\!] \tag{39}$$

5.4 Application of the Szekeres' Algorithm in the Computation of $Y = F \times X$:

We want to compute $Y = X \times F$ where X is a $(k+1)$-vector defined by $(d\ a_1\ \dots\ a_k)$ representing the point $(\frac{a_1}{d}\ \dots\ \frac{a_k}{d})$ of $\mathbb{Q}^k$ which is computed by the **Szekere's** algorithm with rationals $\alpha_1, \dots, \alpha_n$ that are known piecewise and F is a $(k+1 \times 2)$ matrix (the result is an ordinary continued fraction). We want to process inputs and deliver output piecewise. Inputs are entered in continued fraction format and can

ALGORITHM -XI – SZEKERES' ALGORITHM

Input:
A vector $\alpha = (\alpha_1, \ldots, \alpha_k)$ which satisfies $1 > alpha_1 > \ldots > \alpha_i > \alpha_{i+1} > \ldots > alpha_k > 0$ (see 8).

Output:
A k-multicontinued fraction, best approximation of α. If α has one of its component irrational, then the user must choose a step n where he wants to have his approximation (he breaks the loop), else the algorithm stops at a step denoted also by n. The approximation to α can be obtained from the remaining A (see equation 8).

Algorithm:

$\mu = k$;
$\Gamma = I_\Gamma \times \alpha^T$;
$\epsilon = ComputeEpsilon(\mu)$;
$A = A(0)$;
while($NotFinished(\Gamma)$) **do**
  begin
  $A = A \times T(\epsilon, \mu)$;
  $\mu = ComputeMu(A)$;
  $\epsilon = ComputeEpsilon(\mu, \Gamma)$;
  $\Gamma = T_\Gamma(\epsilon) \times \Gamma$;
  end;

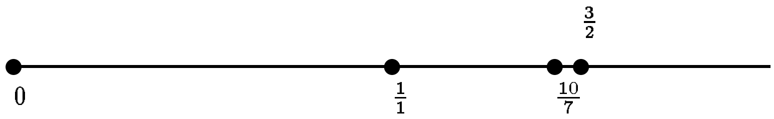


Figure 17: Convergence of a $a_i(r)$ to a_i for $[1/2/3] = \frac{10}{7}$.

be interleaved (for instance, we process the partial quotient $p_j^{(i)}$ of variable a_j, then partial quotient $p_k^{(l)}$ of variable a_l). If one of the input variables is an infinite continued fraction, then, our algorithm has an infinite number of steps, but the result is refined at each step such that we can have an approximation to the exact result at any step. Referring to the *Szekeres*' algorithm, we can factorize the n-th rational approximation to X^t in

$$X^T(n) = \underbrace{A(0) \times \{T(\epsilon(1), \mu(0)) \ldots T(\epsilon(i), \mu(i-1)) \ldots T(\epsilon(n), \mu(n-1))\}}_{A(n)} \times e^T_{\mu(n-1)} \tag{40}$$

where $e^T_{\mu(n-1)}$ is the transposed $(\mu(n-1)+1)$-th unit vector which choose the right column in $A(n)$.

$$e^T_{\mu(n-1)} = \begin{array}{r} \\ \\ \mu(n-1) \text{ line} \rightarrow \\ \\ \\ \\ \end{array} \begin{pmatrix} 0 \\ \vdots \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{41}$$

We can output piecewise by the same process described in the computation of $Y = F \times X$. The main problem is to provide input piecewise, i.e. to deliver transformation $T(\epsilon(i), \mu(i))$. This problem is the same as to compute $\epsilon(i)$ when the current inputs ,that have already been processed , approximate X. We must assume for correctness of calculus, that the transformation $T(\epsilon(i), \mu(i))$ done when a certain amount of input data have been entered, is the same as if we have waited for all inputs to be entered and then produce the transformations.

As input data are entered in continued fraction format, we can know if the current variable $a_i(r)(a_i(r) = [a_0^{(i)}/\ldots/a_r^{(i)}]$ is the r-th convergent approximated to a_i) is lower or greater than a_i. Indeed, it depends only on the parity of r. If r is even then $a_i(r) \leq a_i$ else $a_i(r) \geq a_i$.

Considering the first step of the algorithm, i.e. case $n = 0$, we must determine when $\epsilon(1)$ can be known **surely**. For that purpose we must determine if

$$\epsilon(1) \begin{cases} 0 \text{ if } \gamma_{00} > \gamma_{0k}. \\ 1 \text{ if } \gamma_{00} < \gamma_{0k}. \end{cases} \tag{42}$$

But $\gamma_{00} = 1 - \alpha_1$ and $\gamma_{0k} = \alpha_k$, so we must determine which case is true:

$$\epsilon(1) \begin{cases} 0 \text{ if } 1 - \alpha_1 > \alpha_k. \\ 1 \text{ if } 1 - \alpha_1 < \alpha_k. \end{cases} \tag{43}$$

Using the property of increasingness of the even convergents $a_0, a_2, \ldots, a_{2i}, \ldots$ and decreasingness of the odd convergents $a_1, a_3, \ldots, a_{2i+1}, \ldots$ (see figure 17), it comes that $1 - (\alpha_1 + \alpha_k) > 0$ is true when it can be determined safety that $(\alpha_1 + \alpha_k)$ is a upper bound (parity of both α_1 and α_k is odd) . Proceeding in the same way, we can assert that $1 - (\alpha_1 + \alpha_k) < 0$ is true when α_1 and α_k are even and their sum is greater than 1. To resume, computation of $\epsilon(1)$ can be achieved when α_1 and α_k have the same parity and their sum assert the condition (in their current approximation to the initial value).

$\frac{\alpha_1}{\alpha_k}$	odd	even
odd	Determined if $1 > \alpha_1 + \alpha_2$	Wait
even	Wait	Determined if $\alpha_1 + \alpha_2 > 1$

The general problem is to give the boolean value of $\gamma_0 > \gamma_\mu$ at a step of the algorithm where only some, but not all, the inputs have been processed.

Let us call σ the state of the algorithm described by the inputs already entered:

$$\sigma = (\alpha'_1, \ldots, \alpha'_k) \tag{44}$$

where α'_i is the last approximation to α_i done by considering the input $a_0^{(i)}, \ldots, a_{r(i)}^{(i)}$ ($r(i) + 1$ partial quotient of α_i have been processed so far).

$$\alpha'_i = [a_0^{(i)}, \ldots, a_{r(i)}^{(i)}]$$

We will design by $\gamma_{ni}(\sigma)$ the approximation of γ_{ni} at step n when σ is the current state. The problem is to deliver the state of "$\gamma_{n0}(\sigma) > \gamma_{n\mu}(\sigma)$" which can take three different values:

- **TRUE** in that case, we have $\gamma_{ni} > \gamma_{n0}$ when α is known.
- **FALSE**: in that case, we have $\gamma_{ni} < \gamma_{n0}$ when α is known (remind that $\gamma_{ni} \neq \gamma_{n0}$).
- **UNDETERMINED:** When either **TRUE** or **FALSE** can be decided.

Our algorithm gives bound to the value γ_{ni}:

$$\gamma_{ni}(\sigma) - \Delta_{ni}(\sigma) \leq \gamma_{ni} \leq \gamma_{ni}(\sigma) + \Delta_{ni}(\sigma) \tag{45}$$

The value of "$\gamma_{n0}(\sigma) > \gamma_{n\mu}(\sigma)$" can be determined only if:

- $\gamma_{n0}(\sigma) - \Delta_{n0}(\sigma) > \gamma_{n\mu}(\sigma) + \Delta_{n\mu}(\sigma)$, and in that case, the result is **TRUE**.
- $\gamma_{n0}(\sigma) + \Delta_{n0}(\sigma) < \gamma_{n\mu}(\sigma) - \Delta_{n\mu}(\sigma)$, and the result is **FALSE**.

Otherwise, the result is **UNDETERMINED** and we must wait for another input $a_j(i)$ which will change σ in σ'.

We ensure that each trust interval associated with γ_{ni} is non-increasing when σ changes to σ'.

Once $\epsilon(n)$ can be computed, we must update Γ and Δ:

- $\epsilon(n) = 1$

$$\gamma_{n0} = \gamma_{(n-1)\mu(n-1)} - \gamma_{(n-1)0} \tag{46}$$
$$\gamma_{n\mu} = \gamma_{(n-1)0} \tag{47}$$

- $\epsilon(n) = 1$

$$\gamma_{n0} = \gamma_{(n-1)0} - \gamma_{(n-1)\mu(n-1)} \tag{48}$$
$$\gamma_{n\mu} = \gamma_{(n-1)\mu(n-1)} \tag{49}$$

The decide to stock γ_{ni} as a function linear function of α_i with integral coefficient, hence performing "symbolic" calculus upon these functions.

$$\gamma_{ni}(\sigma) = \sum_{j=0}^{k} a_{ij}(n)\alpha_j \tag{50}$$

where, for convenience of writing, we have $\alpha_0 = 1$.

When a transformation cannot be decided because of its **UNDETERMINED** state, more inputs must be performed (leading to a σ' state), shrinking the value of α_i , $i \in [1, k]$, and then shrinking the interval $\Delta(\sigma)$ into $\Delta(\sigma')$.

At each time a computation of $\epsilon(n)$ is possible or an input is proceeded, we first compute the new trust interval and then check is we can perform a new transformation.

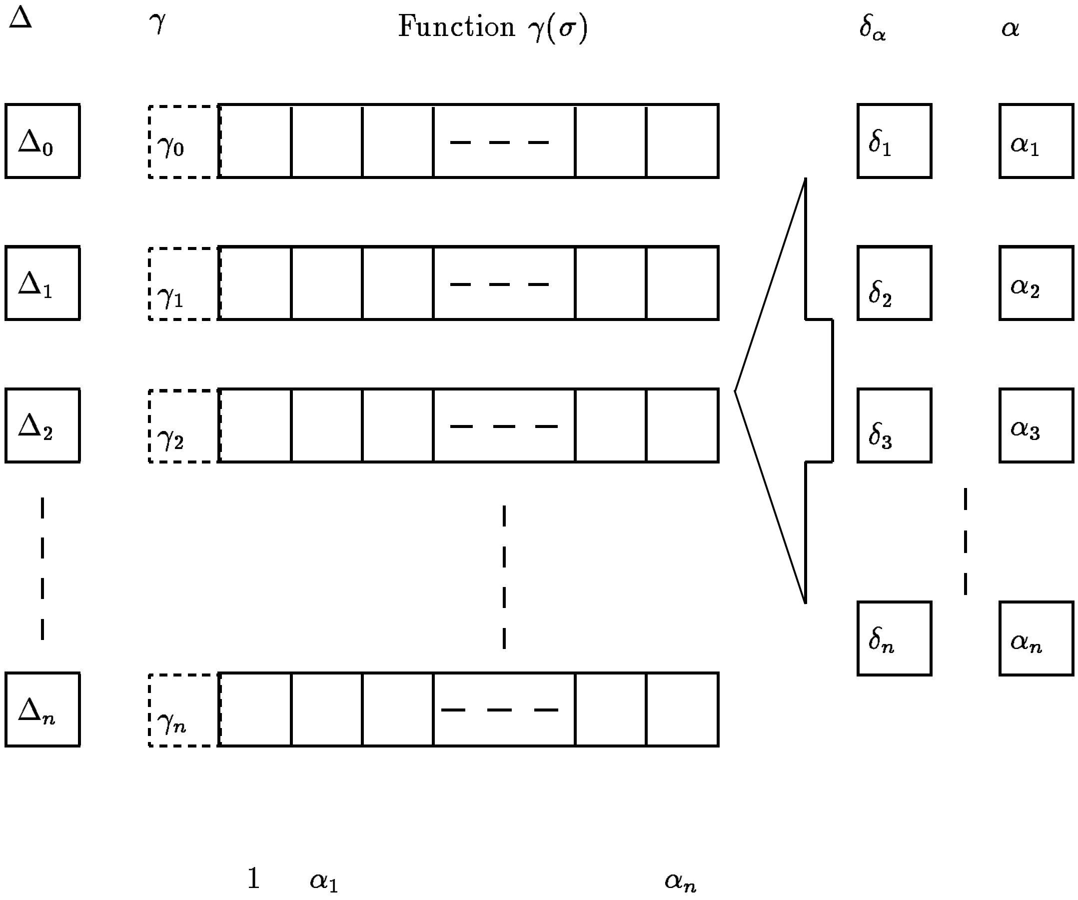


Figure 18: Computing Transformations Piecewise

The trust intervals are computed as follows:

$$\Delta_{ni}(\sigma) = \sum_{j=0}^{k} a_{ij}(n) * \Delta_{\alpha_k}(\sigma) \tag{51}$$

where $\Delta_{\alpha_0} = 0$ and $\Delta_{\alpha_i} \overset{\sigma \rightarrow \sigma'}{=} |\alpha_i(\sigma) - \alpha_i(\sigma')|$.

Then, the boolean value of $\gamma_{n0} > \gamma_{n\mu}$ can already be known, if in state σ, we have:

$$\gamma_0(\sigma) - \Delta_0(\sigma) > \gamma_\mu(\sigma) + \Delta_\mu(\sigma)$$

or

$$\gamma_0(\sigma) + \Delta_0(\sigma) < \gamma_\mu(\sigma) - \Delta_\mu(\sigma)$$

Otherwise, more inputs have to be processed, and updatings on Δ and Γ are done, until one of the two conditions is satisfied.

6 An Algorithm for Computing Functions On Variables Entered in Multicontinued Fractions Format:

In this section, we show briefly how an algorithm based on multicontinued fractions can be developed using the same principles described in the previous sections. We first define a k-multicontinued fraction (by means of an algorithm) and associates its matrix representation. Then we show how it is possible

ALGORITHM -XII – COMPUTING A k-MULTICONTINUED FRACTION

INPUT: A $(k+1)$-vector of $\mathbb{N}^{k+1}$: $x = (p_1 \ \dots \ p_k \ q)$ with $q > 0$.
OUTPUT: The k-multicontinued fraction $[f_0, s_0, \dots, f_n, s_n]$ denoting x.
ALGORITHM:

$i := 0$;
while $(x \neq e_{k+1})$ **do**
begin
For each $i \in [\![1,k]\!]$ **do** $p'_i := \lfloor \frac{p_i}{q} \rfloor$;
$f_i = (p'_1 \ \dots \ p'_k)$
$x := (p_1 - p'_1 q \ \dots\dots \ p_k - p'_k q \ \ q)$;
Choose j such as $p_j = \max_{i \in [\![1,k]\!]}\{p_1, \dots, p_k\}$;
$q' := \lfloor \frac{q}{p_j} \rfloor$;
$s_i := q' * e_j$;
$i := i + 1$;
end

to follow the path of the previous algorithm and extend the computational function to several variables coded by multicontinued fractions.

6.1 Definition of a k-multicontinued Fraction:

We represents by $x = (p_1 \ \dots \ p_k \ q)$ the point of $\mathbb{Q}^k$ defined by the k-uplet $(\frac{p_1}{q} \ \dots \ \frac{p_k}{q})$. Given a point P of $\mathbb{R}^k$, we can first approximate by the **Szekeres**' algorithm P to P' such as P' is now a point of $\mathbb{Q}^k$. P' can be represented by a $(k+1)$-vector $x = (p_1 \ \dots \ p_k \ q)$. We associate to x a multicontinued fraction defined by

$$x = [f_0, s_0 / \dots / f_n, s_n]$$

where $f_i, s_i \ \forall \ i \in [\![0,n]\!]$ are k-vectors of $\mathbb{N}^k$ defined successively by the following algorithm:

Given a $(k+1)$-vector $x = (p_1 \ \dots \ p_k \ q)$ with $q > 0$ of $\mathbb{Q}^{k+1}$ denoting a point of $\mathbb{Q}^k$, we compute its multicontinued fraction by applying to x a succession of two steps.

$$x = (p_1 \ \dots \ p_k \ q) = \frac{(p_1 \ \dots \ p_k)}{q}$$

Computing f_i, s_i:

- Step 1: Processing the numerator $p_i \ \forall \ i \in [\![1,k]\!]$ Compute for each numerator p_i the integer number a_i such as $p_i - a_i q \geq 0$. For each i do $x := x - a_i q e_i$.
- Step 2: Processing the denominator q Find p_j such as p_j is the maximum value and compute b defined by $q - bp_j \geq 1$. Compute $x := x - bp_j e_{k+1}$.

We obtain a new $(k+1)$-vector x'. If $x' = e_{k+1}$ then we stop the algorithm (this corresponds to the ∞ signal) else we inject x' and obtain $f_{i+1}, s_{i+1}, \dots$
For example, if we run the algorithm on the 3-vector of $\mathbb{N}^3$ $(5 \ 3 \ 7)$ denoting a point of $\mathbb{Q}^2$ then we find:

1. - $f_0 = (0 \ 0)$
   - $s_0 = (1 \ 0) \rightarrow x' = (5 \ 3 \ 2)$
2. - $f_1 = (2 \ 1)$
   - $s_1 = (1 \ 0) \rightarrow x' = (1 \ 1 \ 1)$
3. - $f_2 = (1 \ 1)$

- $s_2 = (\infty\ \infty) \rightarrow x' = (0\ 0\ 1)$

Hence, $x = (5\ 3\ 7)$ is represented by the 2-multicontinued fraction:

$$x = [(0\ 0), (1\ 0)/(2\ 1), (1\ 0)/(1\ 1), (\infty\ \infty)]$$

Please note that if x is a 2 dimensional vector then the 1-multicontinued fraction is a continued fraction obtained by the algorithm is $[a_0, a_1/\ldots/a_{2n}, a_{2n+1}]$ such that $a_{2n} = 1$.

6.2 Matrix Representation of a Multicontinued Fraction:

We represents the k-multicontinued fraction of x by means of matrices in order to formalize the computational function in term of product of matrices.

Let X be a $(k+1)$-vector of $\mathbb{N}^{k+1}$ denoted by its k-multicontinued fraction $[f_0, s_0/f_1, s_1/\ldots/f_n, s_n]$ then X can be written in term of product of matrices as follows:

$$X = e_{k+1} \times S_n \times F_n \times \ldots \times S_0 \times F_0$$

where S_i, F_i are defined by

$$S_i = Id + \sum_{j=1}^{k} M_{jk} \times s_i.e_j$$

and

$$F_i = Id + \sum_{j=1}^{k} M_{kj} \times f_i.e_j$$

where . is the scalar product and $M_{kj} = (m)_{a \in [\![1,k+1]\!] b \in [\![1,k+1]\!]}$ is the matrix defined by $m_{ab} = 1$ if $a = k$ and $b = j$, $m_{ab} = 0$ otherwise.

$$M_{ij} = e_i \times e_j^t$$

It's worth noting that F_i and S_i are inversible matrices.

$$S_i^{-1} = Id - \sum_{j=1}^{k} M_{jk} \times s_i.e_j$$

and

$$F_i^{-1} = Id - \sum_{j=1}^{k} M_{kj} \times f_i.e_j$$

6.3 Computing Functions of One Variable Entered in the MultiContinued Fraction Format:

Once the matrix representation of the multicontinued fraction has been defined, we can formalize the computation in term of matrix which are entered piecewise[16].

$$\begin{array}{r} \mathrm{F} : Q^k \longrightarrow Q^n \\ X = (p_1\ \ldots\ p_k\ q) \rightarrow \mathrm{F}(X) = (f_1(x_1\ \ldots\ x_k), \ldots, f_n(x_1\ \ldots\ x_k)) \end{array}$$

In term of matrices, the computation can be seen as performing the product

$$Y = X \times F$$

where F is a $(k+1) \times (n+1)$ matrix defining the function F.

[16] Piecewise here means at the <u>"s"</u> or <u>"f"</u> level.

$$(p_1 \ \dots \ p_k \ q) \times \left(\underbrace{\begin{matrix} a_{1,1} & \dots & a_{1,n+1} \\ \vdots & & \vdots \\ a_{k,1} & \dots & a_{k,n+1} \\ d_1 & \dots & d_{n+1} \end{matrix}}_{(n+1) \text{ columns}} \right)$$

The function $f_i(\cdot)$ is defined by

$$f_i(\frac{p_1}{q}, \dots, \frac{p_k}{q}) = \frac{\sum_{j=1}^{k} a_{j,i}p_j + d_i q}{\sum_{j=1}^{k} a_{j,n+1}p_j + d_{n+1}q}$$

But $\frac{p_i}{q} = x_i$, so it yields to

$$f_i(x_1, \dots, x_k) = \frac{\sum_{j=1}^{k} a_{j,i}x_j + d_i}{\sum_{j=1}^{k} a_{j,n+1}x_j + d_{n+1}}$$

The scheme is always the same:

$$\boxed{Y = X \times I^{-1} \times I \times F \times O^{-1} \times O}$$

But X is a $(k+1)$-vector that can be rewritten according its multicontinued fraction $[f_0, s_0/f_1, s_1/\dots/f_n, s_n]$ to:

$$X = e_{k+1} \times S_n \times F_n \times \dots \times S_1 \times F_1 \times S_0 \times F_0$$

We denote by Sim_i the $(k=1)\times(k+1)$ matrix defined by $S_i \times F_i \times \dots \times S_0 \times F_0$. Sim_i can be interpreted as a *k-simplex* (see the appendix corresponding to the simplex A). Each time, input is received (f_{i+1} and s_{i+1}) the new simplex $Sim_{i+1} = S_{i+1} \times F_{i+1} \times Sim_i$ shrinks so that

$$Vol(Sim_{i+1}) < Vol(Sim_i)$$

When no more inputs can be consumed (inputs are exhausted), we shrink the simplex to a single point defined by the last line of Sim_n :

$$X = e_{k+1} \times Sim_n$$

The output can also be interpreted as a n-simplex that shrinks each time an output is produced.
Since we do not know when the input will be exhausted, we must always assure that $e_j \times Sim_i \times F$ is a n-multicontinued fraction (each number is positive). This condition denotes the output process.

Definition 3 *We denote by $M_{.l}$ ($M_{l.}$) the l-th column (respectively the l-th line) of the matrix M.*

Output Process:

- The F-type : $F_{.i} - p_i F_{.(k+1)} \geq 0, \ \forall \ i \in [\![1, k]\!]$
- The S-type : $F_{.(k+1)} - p_i F_{.i} \geq 0, \ \forall \ i \in [\![1, k]\!]$

6.4 Computation of Functions of Several Variables Entered in the Multicontinued Fraction Format:

We develop in this part how to used the generalized matrix to compute functions of n variables entered in multicontinued fraction format $X_1, \ldots, X_n$ where $X_i = (p_1^{(i)} \ \ldots \ p_l(i)^{(i)})$ ($l(\cdot)$ denotes the application that code the length of vector $X_1, \ldots, X_n$). We want to compute:

$$\mathrm{F} : \mathbb{Q}^{l(1)} \times \ldots \times \mathbb{Q}^{l(n)} \longrightarrow \mathbb{Q}^r$$

$$\left.\begin{array}{l} X_1 = (p_1^{(1)} \ \ldots \ p_{l(1)}^{(1)} \ q_1) \\ X_2 = (p_1^{(2)} \ \ldots \ p_{l(2)}^{(2)} \ q_2) \\ \quad\quad\quad \vdots \\ \quad\quad\quad \vdots \\ X_n = (p_1^{(n)} \ \ldots \ p_{l(n)}^{(n)} \ q_n) \end{array}\right\} \longrightarrow (f_1(X_1, \ldots, X_n), \ldots, f_r(X_1, \ldots, X_n), f_{r+1}(X_1, \ldots, X_n))$$

The generalized matrix $M(X_1, \ldots, X_n)$ denoting the computation is defined recursively by:

$$M(X_1, \ldots, X_n) = (p_1^{(1)} \ \ldots \ p_{l(1)}^{(1)} \ q_1) \times \begin{array}{r} \scriptstyle\text{1-st line} \\ \\ \scriptstyle l(1)\text{-th line} \\ \scriptstyle\text{denominator} \end{array} \begin{pmatrix} M(X_2, \ldots, X_n) \\ \vdots \\ M(X_2, \ldots, X_n) \\ M(X_2, \ldots, X_n) \end{pmatrix}$$

and

$$M(X_n) = (p_1^{(n)} \ \ldots \ p_{l(n)}^{(n)} \ q_n) \begin{pmatrix} a_{1,1} & \ldots & a_{1,n} \\ \vdots & & \vdots \\ a_{l(n),1} & \ldots & a_{l(n),n} \\ a_{l(n)+1,1} & \ldots & a_{l(n)+1,n} \end{pmatrix}$$

For example, all the computable functions of $X_1 = (p_1^{(1)} \ p_2^{(1)} \ q_1)$, $X_2 = (p_1^{(2)} \ p_2^{(2)} \ p_3^{(2)} \ q_2)$ and $X_3 = (p_1^{(3)} \ q_3)$ in $\mathbb{Q}^n$ can be denoted by the generalized matrix M (table 3).
We can do the analogy between the generalized matrix and a n-block structure defined by the scalar product ($\boxtimes$) of graph

$$Mesh_{l(1)\times l(2)} \boxtimes \ldots \boxtimes Mesh_{l(i)\times l(i+1)} \boxtimes \ldots \boxtimes Mesh_{l(n)\times (r+1)}$$

The mesh graph $Mesh_{l_1 \times l_2}$ is also obtained by a scalar product of two "lines":

$$Mesh_{l_1 \times l_2} = \underbrace{Mesh_{l_1 \times 1}}_{\text{horizontal line of length } l_1} \boxtimes \underbrace{Mesh_{1 \times l_2}}_{\text{vertical line of length } l_2}$$

$$
M = (p_1^{(1)}\ p_2^{(1)}\ q_1) \times \begin{pmatrix}
(p_1^{(2)}\ p_2^{(2)}\ p_3^{(2)}\ q_2) \begin{pmatrix}
(p_1^{(3)}\ q_3) \times \begin{pmatrix} a_{1,1}^{(0)} & \cdots & a_{1,n+1}^{(0)} \\ a_{2,1}^{(0)} & \cdots & a_{2,n+1}^{(0)} \end{pmatrix} \\
(p_1^{(3)}\ q_3) \times \begin{pmatrix} a_{1,1}^{(1)} & \cdots & a_{1,n+1}^{(1)} \\ a_{2,1}^{(1)} & \cdots & a_{2,n+1}^{(1)} \end{pmatrix} \\
(p_1^{(3)}\ q_3) \times \begin{pmatrix} a_{1,1}^{(2)} & \cdots & a_{1,n+1}^{(2)} \\ a_{2,1}^{(2)} & \cdots & a_{2,n+1}^{(2)} \end{pmatrix} \\
(p_1^{(3)}\ q_3) \times \begin{pmatrix} a_{1,1}^{(3)} & \cdots & a_{1,n+1}^{(3)} \\ a_{2,1}^{(3)} & \cdots & a_{2,n+1}^{(3)} \end{pmatrix}
\end{pmatrix} \\
(p_1^{(2)}\ p_2^{(2)}\ p_3^{(2)}\ q_2) \begin{pmatrix}
(p_1^{(3)}\ q_3) \times \begin{pmatrix} a_{1,1}^{(4)} & \cdots & a_{1,n+1}^{(4)} \\ a_{2,1}^{(4)} & \cdots & a_{2,n+1}^{(4)} \end{pmatrix} \\
(p_1^{(3)}\ q_3) \times \begin{pmatrix} a_{1,1}^{(5)} & \cdots & a_{1,n+1}^{(5)} \\ a_{2,1}^{(5)} & \cdots & a_{2,n+1}^{(5)} \end{pmatrix} \\
(p_1^{(3)}\ q_3) \times \begin{pmatrix} a_{1,1}^{(6)} & \cdots & a_{1,n+1}^{(6)} \\ a_{2,1}^{(6)} & \cdots & a_{2,n+1}^{(6)} \end{pmatrix} \\
(p_1^{(3)}\ q_3) \times \begin{pmatrix} a_{1,1}^{(7)} & \cdots & a_{1,n+1}^{(7)} \\ a_{2,1}^{(7)} & \cdots & a_{2,n+1}^{(7)} \end{pmatrix}
\end{pmatrix} \\
(p_1^{(2)}\ p_2^{(2)}\ p_3^{(2)}\ q_2) \begin{pmatrix}
(p_1^{(3)}\ q_3) \times \begin{pmatrix} a_{1,1}^{(8)} & \cdots & a_{1,n+1}^{(8)} \\ a_{2,1}^{(8)} & \cdots & a_{2,n+1}^{(8)} \end{pmatrix} \\
(p_1^{(3)}\ q_3) \times \begin{pmatrix} a_{1,1}^{(9)} & \cdots & a_{1,n+1}^{(9)} \\ a_{2,1}^{(9)} & \cdots & a_{2,n+1}^{(9)} \end{pmatrix} \\
(p_1^{(3)}\ q_3) \times \begin{pmatrix} a_{1,1}^{(10)} & \cdots & a_{1,n+1}^{(10)} \\ a_{2,1}^{(10)} & \cdots & a_{2,n+1}^{(10)} \end{pmatrix} \\
(p_1^{(3)}\ q_3) \times \begin{pmatrix} a_{1,1}^{(11)} & \cdots & a_{1,n+1}^{(11)} \\ a_{2,1}^{(11)} & \cdots & a_{2,n+1}^{(11)} \end{pmatrix}
\end{pmatrix}
\end{pmatrix}
$$

Table 3: The generalized computational matrix

A A Few Words About the Simplex

Definition and Volume of a Simplex

We just give here the definition and the volume of a simplex. If further information is required, the reader can find information in [2]

Definition 4 *An n-simplex is a set point composed of $n+1$ vertices defined by*

$$S : u + \sum_{i=0}^{n} \mu_i c_i, \begin{cases} \mu_i > 0 \\ \sum_{i=1}^{n} \mu_i = 1 \\ u, c_i \in V \end{cases}$$

where V is the space where the simplex is embedded.

Property 12 *This yields straightforwardly that S is convex.*

Note that u is the reference vector (translation of the simplex).

Given a simplex, we want to compute its volume. For that purpose, we define the unit volume as the size of the polytope defined by the unit vectors of V. Then, supposed we are given $\mathcal{A} = (a_1, a_2, \ldots, a_n)$ a basis of R^n space, σ an endomorphism which verifies that $\forall\ i \in [1, n]$, $\sigma a_i = b_i$ where b_i are the vectors defining the simplex. then it follows that the volume of the simplex is

$$Vol(S) = |\det \sigma| = \begin{vmatrix} \sigma_{11} & \ldots & \sigma_{1n} \\ \vdots & \ddots & \vdots \\ \sigma_{n1} & \ldots & \sigma_{nn} \end{vmatrix}$$

Appendices – Tools

These appendices describe how to use the different programs that have been implemented during the training period. When it is required, a brief description of the input and output files are given.

The programs can be found in `fnielsen@imada.ou.dk --- Directory entrance: EXEC/`

If further information is required, please do not hesitate and mail to `fnielsen@ens.ens-lyon.fr`.

B The MCF-Simulator program:

The MCF-Simulator was one of my first program implemented. It deals with multicontinued fractions as introduced before.

B.1 The type of functions allowed:

Let X denote a n-multicontinued function: $X = (k_1\ k_2\ \ldots\ k_n\ d)$. X represents the n variables: $x_1 = \frac{k_1}{d}, x_2 = \frac{k_2}{d}, \ldots, \frac{k_n}{d}$. The program uses the simplex theory where both input and output is interpreted as a simplex that shrinks to its final value. Regarding with the problem of termination, the result is approximated to its correct value for $k \geq 2$. We allow the computation of $f(X)$ as follows:

$$f(X) = (k_1\ k_2\ \ldots\ k_n\ d) \times \begin{pmatrix} r_{11} & r_{12} & \ldots & r_{1p} \\ r_{21} & r_{22} & \ldots & r_{2p} \\ \vdots & \vdots & \vdots & \vdots \\ r_{n1} & r_{n2} & \vdots & r_{np} \end{pmatrix}$$

The matrix $R = (r_{ij})_{i \in [1,n], j \in [1,p]}$ represents a mapping function from $\mathbb{Q}^n$ to $\mathbb{Q}^p$. The programs described each step. Inputs are proceeded randomly.

B.2 A session with MCF-Simulator:

At the shell prompt, type `MCF-Simulator`. You obtain the following welcome message:

```
TTY MCF-SIM. Multicontinued Fraction:
1993 -- v1.0    Computation of Y= X F


Do you want to retrieve information of a file (0:NO/1:YES):
```

You can proceed entries automatically, if a file containing the information existed. The format of the file is described later. Once answering `0` to the first question, you have the possibility to create a LaTeX file describing the steps genered by the program. If you activate that option, a filename will be requested. Successively, n and p must be entered. For example, we have chosen the following parameters:

```
Latex Option (1:YES/0:NO) :0
Size X:2
Size F:2
Matrix [1,2] must be entered row by row (each return carriage validating the current row
[ 0]>
```

You enter therefore the $(r-1)$-multicontinued fraction and the computational matrix.

```
Matrix [1,2] must be entered row by row (each carriage return validating the current row
[ 0]>1 5
Matrix [2,2] must be entered row by row (each carriage return validating the current row
[ 0]>1 2
[ 1]>5 2
START: --Computation of X * Y--
.
.
.
```

The programs simulates the behavior of the algorithm on the multicontinued fraction. At each step, the volume of both simplexs (input and output) are computed. If that volume is infinite, the convention is to display -1.

B.3 Description of the input file:

Data can be retrieved automatically using a source file that contains the following information (the file displayed below is `ex`):
1 ← LaTeX option (1 if selected, 0 otherwise)
ex.tex ← only if the LaTeX option is selected

2 ← Dimension of n (vector in $\mathbb{Q}^{n-1}$).
2 ← Dimension of p (vector in $\mathbb{Q}^{p-1}$).
7 5 ← The multicontinued fraction (here it represents the point of $\mathbb{Q} : \frac{7}{5}$).

$$\left|\begin{array}{cc} 1 & 0 \\ 2 & 3 \end{array}\right\}\text{The computational matrix}$$

B.4 How to use the graphic interface:

A graphic interface has been developed. The program allows a step-by-step execution when the button –STEP–[17] is invoked. At the shell prompt, you just type `MCF-graphic`. The main window appeared. You the select, –READ DATA–. The file is the same as the one used in **MCF-Simulator**, except that it does not contain the LaTeX option.
Description of the buttons:

- –READ DATA– Display a subwindow which allows to enter the filename.
- –MANIPULATION– This subwindow allows to convert numbers in different codings.
- –SHOW INPUT– displays a window where the input simplex is drawn in a plane (if the dimension is higher than 3, then it chooses the first three components).
- –SHOW OUTPUT– the same for the output simplex.
- –STEP– execute one step in the algorithm.
- –QUIT– exit.

Comments can be added in the text window and saved (like a texteditor).

B.5 Output delivered by MCF-Simulator:

[17] when a button BUTTON must be pressed, we note this event –BUTTON–

B.5.1 LaTeX output: an example

We display below the output delivered by the program once compiled:

LaTeX output of *MCF-SIM*–
Computation of

$$Y = X \times F$$

$$Y = \begin{pmatrix} 7 & 5 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ 2 & 3 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Input piece of the 0-th component : 1 with entrance parity 0 The state produced by input is then:

$$Y = \begin{pmatrix} 2 & 5 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ 3 & 3 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

With the input simplex:

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

The state produced by outputing $0 : 1$ with parity 1 is :

$$Y = \begin{pmatrix} 2 & 5 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ 0 & 3 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

Input piece of the 0-th component : 2 with entrance parity 1 The state produced by input is then:

$$Y = \begin{pmatrix} 2 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 6 \\ 0 & 3 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

With the input simplex:

$$\begin{pmatrix} 3 & 2 \\ 1 & 1 \end{pmatrix}$$

The state produced by outputing $0 : 6$ with parity 0 is :

$$Y = \begin{pmatrix} 2 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ 0 & 3 \end{pmatrix} \times \begin{pmatrix} 7 & 6 \\ 1 & 1 \end{pmatrix}$$

Input piece of the 0-th component : 2 with entrance parity 0 The state produced by input is then:

$$Y = \begin{pmatrix} 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ 2 & 3 \end{pmatrix} \times \begin{pmatrix} 7 & 6 \\ 1 & 1 \end{pmatrix}$$

With the input simplex:

$$\begin{pmatrix} 3 & 2 \\ 7 & 5 \end{pmatrix}$$

The state produced by outputing $0 : 0$ with parity 1 is :

$$Y = \begin{pmatrix} 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ 2 & 3 \end{pmatrix} \times \begin{pmatrix} 7 & 6 \\ 1 & 1 \end{pmatrix}$$

Here, all the variables habe bee processed previously. We perform the product of matrix.The resulting vector is:

$$\begin{pmatrix} 2 & 3 \end{pmatrix}$$

The output produced by the algorithm is: Component $0 : 1/6/0/1/2/\infty$ The result of the computation is $\frac{17}{15}$

B.5.2 Hardcopy of MCF-Simulator:

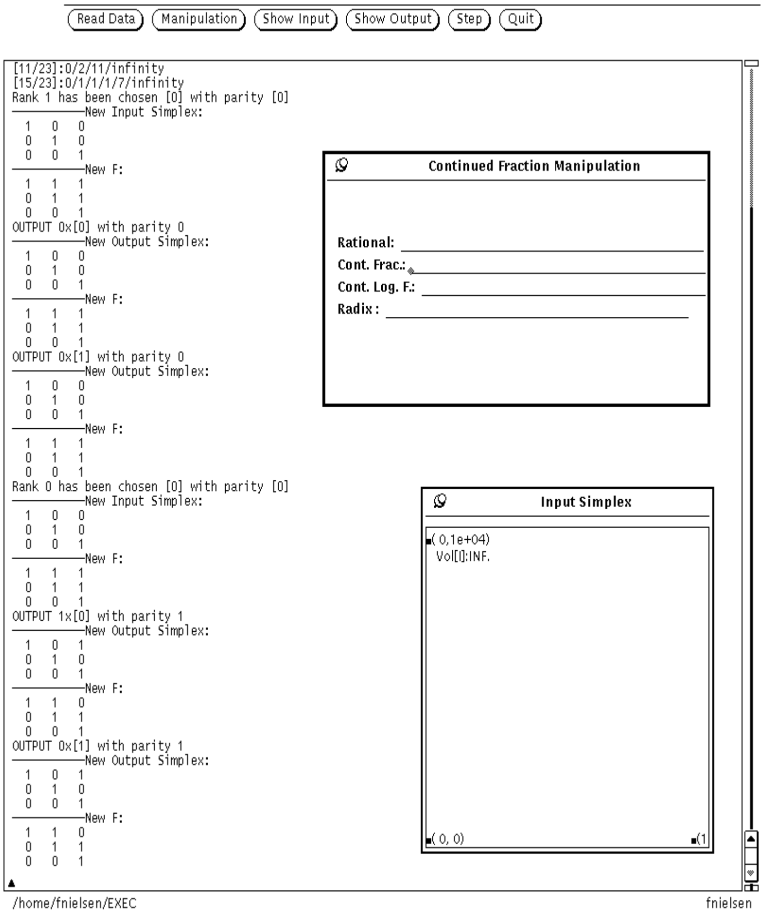


Figure 19: The graphical interface.

C The MulCF-Simulator program:

The program uses the principle of generalized matrix to compute function of n variables that are entered in the continued fraction format. Inputs are proceeded randomly. The type of function that can be evaluated has been studied in the second section. Computing the corresponding generalized matrix to a function of k variable can sometimes be heavy. The user can use **MulCF-function** which given a function of k variables, computes the associated k-generalized matrix (2^{n-1} (2×2)-matrices). The program is simply called with `MulCF-Simulator filename` where filename is the source file. Both operands and generalized matrix are written in filename, following the format :

- n: Number of variables (defined the dimension)
- n rational numbers $p_i q_i$ designing $\frac{p_i}{q_i}$. The variables must be entered following the order $x_n, x_{n-1}, \ldots, x_0$.
- the n-generalized matrix.

`testMulCF` is displayed below:

3

1 1 (variable $x_3 = \frac{1}{1}$)
2 1 (variable $x_2 = \frac{2}{1}$)
3 1 (variable $x_1 = \frac{3}{1}$)

0 0
0 1

1 0
0 0

0 0
0 1

1 0
0 0

D The HyperCF-Simulator program:

This program implements the algorithm according the hypercube structure. The inputs are proceeded randomly. The source file is decomposed in three part:

- The number n of variables (build the hypercube in dimension $n+1$).
- The n-generalized matrix
- The n numbers, written in the continued fraction format: $[a_0/\ldots/a_n]$.

We display below, for example, a model (filename: `testHyperCF`):

```
2

1 2
3 4

1 2
0 4



[1/2/3/4/5]
[9/8/7]
```

To run the program on that example, type:
`beethoven /EXEC 61> HyperCF testHyperCF | more`

Then, the program begins by creating the hypercube and loading the data (generalized matrix and variables).

```
Simulator on the hypercube structure:

Create hypercube in dimension 3
Loading data in a Gray's coding
Hypercube is ready to computation
Loading 2 numbers in CF format
[ 1]    < 5>       225/  157
[ 2]    < 3>       520/   57
Numbers loaded
```

The generalized matrix can be also computed using **MulCF-function** which is described in the next annex.

E The MulCF-function program:

This program has been written to ease the creation of source files to the simulators. Indeed, computing handly the generalized matrix given a function of n variables can require time... This program has been written using *flex* and *bison*.

A session with **MulCF-function** is shown below:

```
beethoven ~/EXEC 79>MulCF-function


2>x[2]*x[1]*10*5+x[2]*4+2*x[1]+3/1+4*x[2]*7*x[1]
;
Execute current statement...
Numerator[4]:
50*x[1]*x[2]+4*x[1]+2*x[2]+3
Denominator[2]:
1+28*x[1]*x[2]
[50     28]
[2      0]

[4      0]
[3      1]
```

First, the user must enter the dimension of the generalized matrix (the variables are numbered $x[1], x[2], \ldots, x[n]$). Then, a delimiter ">" is used, and the function is entered: $\frac{\text{numerator}}{\text{denominator}}$

The statement is validated and the generalized matrix computed when a ";" (delimiter) is encountered. The grammar of **MulCF-function** is:

PROG	:	STATEMENT | PROG STATEMENT
STATEMENT	:	INTEGER SUPERIOR POL DIVIDE POL DELIM
POL	:	POL + TERM | TERM
TERM	:	TERM * TYPE | TYPE
TYPE	:	x[INTEGER] | INTEGER

F The SzekeresMCF program:

The **Szekeres**' algorithm allows to have the best rational approximation of a set of n real numbers. The program shows that the number of steps required to have a correct approximation is high. In practice, this algorithm can not be used as an input process to a program that computes functions on multicontinued fractions as it has been studied.

The input file contains the n numbers that must be approximated by a n-multicontinued fraction. These numbers must be entered in decreasing order and must not be superior to 1[18]. For example, if we want to approximate to a 3-multicontinued fraction 0.66667, 0.12345, 0.0343412, the corresponding input file is:

```
3


0.66667
0.12345
0.0343412
```

To run the program on that file, the user must chose also a file that will contain a precise description of each step. For example, if `toapproxime` is the file containing the previous information, the user type at the prompt:

```
schumann ~/EXEC 19> SzekeresMCF toapproxime /dev/tty2
```

If the user do not want to have all the information of each variables, the display file can be set to `/dev/null`.

```
schumann ~/EXEC 20> SzekeresMCF toapproxime /dev/null
```

[18] The user is invited at read the part concerning the description of the algorithm

Conclusion and Perspectives:

I WOULD LIKE to end this rapport with a summary of what has been done and what must still be studied. I have developed, by generalizing the concept described in [4][5][7][8], an algorithm which computes a function of n variables that can be entered piecewise and deliver output also piecewise. The decision cube has also been generalized leading to the notion of decision hypercube. A definition (different from the **Szekeres**' one) of multicontinued fraction has been chosen and an algorithm based on that representation has been outlined. Further studies on the decision "Block" is required but I am quite convinced that it follows the principle evoked in the algorithm which computes function on continued fraction numbers. In summary there are several interesting open questions related to the efficiency of this unit:

- Regarding pipelining, how can the depth of parsing trees be reduced employing the generality of the operations allowed by $f(x_1, \ldots, x_n)$?
- Regarding time complexity, what are the respective performance of the algorithm applied to different matrix factorization like LCF, RPQ, ...
- Regarding space complexity, in terms of space/time/accurancy (coefficient **AT**[19]) tradeoffs, what is the appropriate register size in the unit to most effectively support our proposed bit-serial on-line pipelined computational environment.

Nielsen Franck†,
University of Odense,
Denmark,
August 12, 1993.

† Email: fnielsen@ens.ens-lyon.fr

[19]**The AT coefficient denotes the factor *Area* $\times$ *Times*. A unit that has a better AT is considered better.**

List of Figures

List of Algorithms

List of Tables

References

[1] William B. Jones, W.J. Thron *Continued Fractions : Analytic Theory and Applications*, *Encyclopedia of Mathematics And Its Applications*, *1980* , Addison-Wesley.

[2] Nicolaas H. Kuiper *Linear Algebra and Geometry*, *1965*, North-Holland Publishing Compagny Amsterdam.

[3] Peter Kornerup, David W. Matula *LCF: A lexicographic Binary Representation Of The Rationnals*, April *1989*, Odense Universitet.

[4] Peter Kornerup, David W. Matula *Exploiting Redundancy in Bit-Pipelined Rational Arithmetic*, July *1989*, Odense Universitet.

[5] Peter Kornerup, David W. Matula *Finite Precision Lexicographic Continued Fraction Number Systems*, June 4-5 *1985*, IEEE COMPUTER SOCIETY.

[6] Keld Antonsen, Kim Ramassen *Analysis and Design of an On-line Arithmetic Unit*, *1991*, Odense Universitet.

[7] Peter Kornerup, *An On-line Arithmetic Unit for Bit-Pipelined Rational Arithmetic*, *1987*, Journal of Parallel and Distributed Computing.

[8] Peter Kornerup, David W. Matula *An Algorithm for Redundant Binary Bit-Pipelined Rational Arithmetic*, August *1990*, IEEE TRANSACTIONS ON COMPUTERS.

[9] N.M. Blackman, *The Continued Fraction as an Information Source*, IEEE Trans.Inf.Th Vol. IT-30, *1984*,671-674

[10] R.W. Gosper, *item 101 in HAKMEM*, MIT-AIM, Feb. *1972*,37-44

[11] G.H. Hardy and E.M. Wright, *An Introduction to the Theory of Numbers* 5th ed., Oxford University Press, London, *1979*

[12] A.Y. Khintchin, *Continued Fractions 1935*

[13] D.E. Knuth, *Supernatural Numbers*, in Mathematical Gardner, D.A. Klarner, ed., Van Nostrand, New-York, *1982* ,310-325

[14] D.E. Knuth, *The Art of Computer Programming*, Vol. 2, *Seminumerical Algorithms*, 2nd Ed., Addison-Wesley, reading, 1981.

[15] T.W. Cusik, *The Skezeres Multidimensional Continued Fraction*, Mathematics of Computation, Volume 31, Number 137, January 1977, Pages 280-317.

[16] Søren Peter Johansen, *Specialet Ciffer-Seriel On-line Aritmetik*, Institut for Matematik og Datalogi, Odense Universitet, August 1990, *master project*.