How fast can we compute products?

V. Kislenkov *V. Mitrofanov*
Dept. of Comp. Math. &
Cybernetics, MSU,
Moscow 119899, Russia.
{kvv,mitrofan}@cs.msu.su

E. Zima
Symbolic Computation Group,
University of Waterloo,
Waterloo, Canada
ezima@daisy.uwaterloo.ca

Abstract

In this paper we consider the problem of fast computation of n-ary products, for large n, over arbitrary precision integer or rational number domains. The combination of loop unrolling, chains of recurrences techniques and analogs of binary powering allows us to obtain order-of-magnitude speed improvements for such computations. Three different implementations of the technique (in Maple, C++ and Java) are described. Many examples together with timings are given.

1 Introduction

An arbitrary precision arithmetic is undoubtedly the "work horse" of general purpose computer algebra systems and numerous specialized packages. Advanced algorithms to perform basic operations on arbitrary precision integers are very well known. Many books [1, 4, 7] give overviews of those algorithms together with detailed implementation remarks. Most computer algebra systems (such as Maple [3]) and specialized number theory packages (such as NTL [8]) contain implementations of these algorithms. For example for multiplication they typically use the Karatsuba [7] algorithm. Even some general-purpose programming languages have arbitrary precision arithmetic. As examples, Java has BigInteger as a core API class in the language, and C may be extended with the GNU Project's gmp library.

Section 4.3.3 in [7] is entitled "How fast can we multiply?" In this paper we would like to address a related problem and try to answer the question "how fast we can compute n-ary products for large n?" By "compute" we mean obtaining the exact integer or rational result. By "product" we mean:

$$F(n) = a_1 \cdot a_2 \cdot \ldots \cdot a_n = \prod_{i=1}^{n} a_i, \qquad (1)$$

where $a_i \in Z$ or Q. We are interested in cases when a_i and n are large numbers. We also assume that $a_i = f(i)$, i.e. can be written as closed form expression in i.

Computational problems like (1) are not rare in practice. Such problems arise for example in the evaluation of series of rational numbers [2, 5]. A typical approach to the computation of (1) is binary splitting, which does not reduce operational complexity. The binary splitting algorithm is asymptotically faster then repeated multiplication only if multiplication of n-digits integers is asymptotically faster than $O(n^2)$. In this paper we will show how it is possible to reduce both operational and bit-complexity of (1). Our main effort will be to reduce the number of multiplications and to replace as much of the remaining multiplications as possible by multiplications of smaller in size numbers.

Many familiar computational problems can be written in form (1). For example,
a) $a_i = c$ (constant): $F(n) = c^n$,
b) $a_i = i$: $F(n) = n!$,
c) $a_i = (c + i - 1)$: $F(n) = (c)^{\overline{n}}$ (rising factorial power),
d) $a_i = \frac{N-i+1}{i}$: $F(n) = \binom{N}{n}$,
and so on.

Case a) has received a lot of attention and one has very efficient solutions - variations of binary powering algorithms [4, 7]. In current software, cases b) through d) are implemented by iterated multiplication. Such naive implementations are slow, relative to what has been achieved for case a).

Consider three very simple computational tasks
T1. Compute 7789^{20000}.
T2. Compute 20000!.
T3. Compute $\binom{250000}{125000}$.
For these tasks the number of decimal digits in the answer is between 75000 and 78000. Table 1 contains timings [1] for these 3 tasks obtained on P-200 with 64 Mb of RAM, in Maple, C++(NTL) and Java.

	T1	**T2**	**T3**
Maple	1.40 (2.27)	70.99 (140.07)	6153.48
NTL	0.74	24.39	488.52
Java	7.71	109.41	3461.73

Table 1: Time in seconds for 3 computational tasks in 3 systems.

Times for **T2** and **T3** are much worse than times for **T1**. It is not surprising since binary powering is used for **T1**. We

[1] Factorial and powering are implemented in the kernel of Maple (compiled), and binomial is implemented in the library (interpreted). The parenthesized times are the times for an interpreted factorial and powering.

will describe some approaches to the improvement of times for **T2**, **T3** and more general tasks of the form (1) in this paper.

There are several different techniques which help to accelerate computations of products. Some of them are system dependent, some of them are general. We are particularly interested in general techniques. That is why we choose three very different arbitrary precision integer arithmetics for practical illustrations. Here are interesting features of these arithmetics.

Maple (Maple V Release 5): the base for integer arithmetic is some power of 10 (depending on platform); our implementation is library level Maple code. We have essentially no control over the internal representation of integers and no control at all over memory allocations and reallocations.

NTL (NTL 3.02): the base for integer arithmetic is some power of 2 (depending on platform); our implementation is C++ code; we have control over the internal representation, memory allocations and also over algorithms (for example we can force Karatsuba algorithm for multiplication to be enabled/disabled).

Java (JDK 1.1.7): the internal representation and implementation of the basic arithmetic over BigIntegers are inaccessible; we have no control over the internal representation and very limited information about algorithms used.

We will show in this paper that some improvements can be obtained for all of these systems due to our general approach to the acceleration of computation of products. However, better results in absolute timings (see Tables 3-5) are obtained with NTL where we can combine the general approach with several technical improvements, taking advantage of knowledge gained from NTL sources and using special features of concrete problems.

For our purpose we will combine several different techniques, well known in the theory and practice of optimized compiling and fast computations: binary powering, loop unrolling, chains of recurrences, etc. The chains of recurrences technique is proven to give good results in accelerating iterative computations over regular intervals. One of the objectives of this paper is to show how this technique can be used to accelerate one-point evaluation of a function, especially in cases when this evaluation is computationally intensive.

The rest of the paper is organized as follows. In Section 2 we recall some necessary optimizing techniques and give a brief discussion of bit-complexity. In Section 3 we present combinations of loop unrolling and chains of recurrences techniques to accelerate problem (1). In Section 4 we give an analog of binary powering for computation of factorials. Section 5 describes implementations of our algorithms. Section 6 contains concluding discussions and plans for future works.

2 Preliminaries

2.1 Loop unrolling

Loop unrolling is an optimization transformation widely used in optimizing compilers. It replicates the body of a loop some number of times called the unrolling factor (v) and iterates by step v instead of step 1. Unrolling can improve the performance because

- it reduces loop overhead v times;
- other optimizations become possible (e.g. it may be possible to find common subexpressions in replicated instances of the loop body).

Since all computational tasks considered in this paper can be rewritten as loops, we will use a technique similar to unrolling as one of our tools.

Remark 1. We can say in advance that in the case of arbitrary precision integer computations, loop unrolling alone can not help materially. Loop unrolling must be combined with other techniques in order to obtain significant speed improvements.

Consider an integer $v > 1$ and rewrite n in (1) as $n = k \cdot v + l$ $(0 \leq l < v)$. In terms of (1) loop unrolling can be expressed as

$$F(n) = \prod_{i=1}^{k} \left(\prod_{j=1}^{v} a_{(i-1)v+j} \right) \cdot \prod_{j=1}^{l} a_{kv+j} \qquad (2)$$

If $a_i = c, i = 1, \ldots, n$ and $l = 0$ (for simplicity) then the computation of $F(n) = c^n$ can be written as
$b := \prod_{j=1}^{v} c; \quad F(n) := \prod_{i=1}^{k} b.$

Remark 2. We write the scheme in terms of products as above, giving rise to the following program:

```
b:=1; for j:=1 to v do b:= b * c od;
f:=1; for i:=1 to k do f:= f * b od;
```

This program was obtained by the application of loop unrolling (factor v) to the loop computing $c^{(kv)}$ by repeated multiplications, and by application of substitutions and code motion to the result of unrolling.

Of course, the program above can not compete with binary powering, but we will use it as a base-line model in our discussions. Observe that repeated multiplication takes kv multiplications while the unrolled program takes $k + v$ multiplications. In the case of a fixed point computation if k and v are chosen so that $k \approx v \approx \sqrt{n}$, the unrolling gives $\frac{\sqrt{n}}{2}$ speed improvement.

Remark 3. Repeated multiplication is rarely used in practice, but we time it to provide a baseline for comparisons. In computation of products (1) we will compare repeated multiplication, unrolling and binary-like methods.

2.2 Chains of recurrences

Chains of recurrences (CR) is a technique to accelerate iterative computations. This technique has proven to be of practical use in cases when evaluation of a closed form expression has to be performed over regular intervals [6, 10, 11]. Here we will show that this technique can be useful when we need to compute a single value. In this paper we will use only simple chains of recurrences and simple CR-expressions. We recall briefly necessary definitions and facts and describe useful transformations of CRs.

2.2.1 Definitions.

Given constants $\varphi_0, \ldots, \varphi_{k-1}$, a function f_k defined over $\mathrm{N} \cup \{0\}$, and operators $\odot_1, \ldots, \odot_k$ equal to either $+$ or $*$, we define a *Chain of Recurrences (CR)* as the set of functions $f_0, f_1, \ldots, f_{k-1}, f_k$ such that for $0 \leq j < k$

$$f_j(i) = \begin{cases} \varphi_j, & \text{if } i = 0, \\ f_j(i-1) \odot_{j+1} f_{j+1}(i-1), & \text{if } i > 0. \end{cases} \qquad (3)$$

Further, to denote CRs (3), we will use the shorthand

$$f_0(i) = \Phi(i) = \{\varphi_0, \odot_1, \varphi_1, \odot_2, \varphi_2, \ldots \odot_k, f_k\}(i). \qquad (4)$$

For example,
$f(i) = i = \{0, +, 1\}(i)$,
$f(i) = i^3 + 2i = \{0, +, 2, +, 6, +, 6\}(i)$,
$f(i) = i! = \{1, *, 1, +, 1\}(i)$, etc.
Recall that in (4) k is called the length of CR Φ; Φ is simple if $f_k(i) = \varphi_k$ is constant expression; Φ is a *pure-sum* CR, if $\odot_1 = \odot_2 = \ldots = \odot_k = +$;
$\Phi_r = \{\varphi_r, \odot_{r+1}, \varphi_{r+1}, \ldots \odot_k, f_k\}$, $(0 \leq r \leq k)$ is called *r-order subchain* of the CR Φ. Expressions with CRs as operands are called CR-expressions.

Here we will deal primarily with simple CRs and CRs whose first order subchain is an expression involving simple CRs, for example
$\binom{n}{i} = \{1, *, \frac{\{n,+,-1\}}{\{1,+,1\}}\}$ which gives the well known recurrence $\binom{n}{i} = \binom{n}{i-1}\frac{n-i+1}{i}$ in CR notation.

All CRs above assume that i starts at 0 and steps by 1. However we will use CRs with variable starting and step values. Accordingly we extend the CR notation as follows

$$\Phi(i = i_0, h) = \{\varphi_0, \odot_1, \varphi_1, \odot_2, \varphi_2, \ldots \odot_k, f_k\}(i = i_0, h).$$

For example, $i! = \{1, *, 1, +, 1\}(i = 0, 1)$, at the same time $i! = \{1, *, 2, +, 10, +, 8\}(i = 0, 2)$, or

$$\binom{n}{i} = \{1, *, \frac{\{n, +, -1\}}{\{1, +, 1\}}\}(i = 0, 1),$$

at the same time

$$\binom{n}{i} = \{1, *, \frac{\{n^2 - n, +, -4n + 6, +, 8\}}{\{2, +, 10, +, 8\}}\}(i = 0, 2).$$

2.2.2 Value and shift of a CR.

Interpretation of CRs is based on function `Value` and shift operator E which are defined as follows. Given CR-expression Φ,
$\mathtt{Value}(\Phi) =$
$$\begin{cases} c, & \text{if } \Phi \text{ is a constant expression } c \\ \varphi_0, & \text{if } \Phi = \{\varphi_0, \odot_1, \varphi_1, \ldots, \odot_k, f_k\} \\ F(\mathtt{Value}(\Phi_1), \ldots, \mathtt{Value}(\Phi_m)), & \text{if } \Phi = F(\Phi_1, \ldots, \Phi_m) \end{cases},$$
$E(\Phi) =$
$$\begin{cases} c, & \text{if } \Phi \text{ is a constant expression } c \\ \{\varphi_0 \odot_1 \varphi_1, \odot_1, \varphi_1 \odot_2 \varphi_2, \ldots, \varphi_{k-1} \odot_k \mathtt{Value}(f_k), \odot_k, E(f_k)\}, & \\ \qquad \text{if } \Phi = \{\varphi_0, \odot_1, \varphi_1, \ldots, \odot_k, f_k\} & \\ F(E(\Phi_1), \ldots, E(\Phi_m)), & \text{if } \Phi = F(\Phi_1, \ldots, \Phi_m) \end{cases}$$
Function $\mathtt{Value}(\Phi)$ returns the current value associated with a CR-expression, $E(\Phi)$ returns a CR-expression updated for the next value of i.

It is useful to have specialized definitions of `Value` and E for simple CRs and CRs with simple CR-expressions as subchains:

$$\begin{aligned} &\mathtt{Value}(\{\varphi_0, \odot_1, \Phi_1\}) = \varphi_0, \\ &E(\{\varphi_0, \odot_1, \Phi_1\}) = \{\varphi_0 \odot_1 \mathtt{Value}(\Phi_1), \odot_1, E(\Phi_1)\}, \\ &E(\{\varphi_0, \odot_1, \varphi_1, \odot_2, \varphi_2, \ldots \odot_k, \varphi_k\}) = \\ &= \{\varphi_0 \odot_1 \varphi_1, \odot_1, \varphi_1 \odot_2 \varphi_2, \odot_2, \ldots, \varphi_{k-1} \odot_k \varphi_k, \odot_k, \varphi_k\}, \\ &E(\{\varphi_0, \odot_1, \tfrac{\Delta}{\Gamma}\}) = \{\varphi_0 \odot_1 \frac{\mathtt{Value}(\Delta)}{\mathtt{Value}(\Gamma)}, \odot_1, \tfrac{E(\Delta)}{E(\Gamma)}\}. \end{aligned}$$

2.2.3 Fast method to change the step of a CR.

Assume we are given a simple pure-sum CR

$$\Phi(i = 0, 1) = \{\varphi_0, +, \varphi_1, +, \varphi_2, \ldots, +, \varphi_k\}(i = 0, 1)$$

which corresponds to a polynomial $P(i)$ of degree k in i with the table of finite differences $\varphi_0, \varphi_1, \varphi_2, \ldots, \varphi_k$. How can we find the CR

$$\Psi(i = 0, v) = \{\psi_0, +, \psi_1, +, \psi_2, \ldots, +, \psi_k\}(i = 0, v)$$

without reconstructing it from scratch?

We call the corresponding transformation a *factor v CR unrolling* and denote it $U_v : \Phi(i = 0, 1) \to \Psi(i = 0, v)$. Let $I : \Phi \to \Phi$ be identity transformation. Write Φ as $\Phi = \{\varphi_0, +, \Phi_1\}(i = 0, 1)$ where Φ_1 is the first order subchain of Φ. After v steps of computations with this CR we have
$\Phi(i = v, 1) = \{\varphi_0 + \mathtt{Value}(\Phi_1) + \mathtt{Value}(E(\Phi_1)) + \ldots$
$\ldots + \mathtt{Value}(E^{v-1}(\Phi_1)), +, E^v(\Phi_1)\}$.
From here
$\psi_1 = \mathtt{Value}(\Phi(i = v, 1)) - \mathtt{Value}(\Phi(i = 0, 1)) =$
$= \mathtt{Value}(\Phi_1) + \mathtt{Value}(E(\Phi_1)) + \ldots + \mathtt{Value}(E^{v-1}(\Phi_1)) =$
$= \mathtt{Value}((I + E + E^2 + \ldots + E^{v-1})(\Phi_1))$.
Now consider $\hat{\Phi}_1 = \{\hat{\varphi}_1, +, \hat{\Phi}_2\}$ where $\hat{\Phi}_2$ is first order subchain of $(I + E + E^2 + \ldots + E^{v-1})(\Phi_1)$ and perform the same transformation:
$\mathtt{Value}((I + E + E^2 + \ldots + E^{v-1})(\hat{\Phi}_2))$ gives ψ_2. Continue the same procedure for 3d, 4th, ... order subchains we get the following general formula:

$$\psi_j = \mathtt{Value}((I + E + E^2 + \ldots + E^{v-1})^j(\Phi_j)),\ j = 0, 1, \ldots, k,$$

where Φ_j is j-order subchain of Φ. Note, that these transformations work exactly the same when starting value for i differs from 0.

We are particularly interested in transformation U_2 which doubles the step of a pure-sum CR:
$U_2 : \Phi(i = 0, 1) \to \Psi(i = 0, 2)$, where $\Psi(i = 0, 2) =$
$= \{\varphi_0, +, \mathtt{Value}((I + E)\Phi_1), +, \mathtt{Value}((I + E)^2\Phi_2), +,$
$\ldots, +, \mathtt{Value}((I + E)^k\Phi_k)\}$.

Computationally all this is much simpler than it looks at first glance, because application of $(I + E)$ to Φ_1 applies the same operator to all subchains $\Phi_2, \Phi_3, \ldots, \Phi_k$. That means that an analog to the Horner scheme is possible. Let us write the algorithm to perform transformation U_2:
`Input:` $\varphi_0, \varphi_1, \varphi_2, \ldots, \varphi_k$.
`Output:` $\psi_0, \psi_1, \psi_2, \ldots, \psi_k$.

```
for i:=0 to k do  $\psi_i$  :=  $\varphi_i$  od;
for i:=1 to k do
    { apply (I+E) to subchain  $\Psi_i$  }
    for j:=i to k-1 do
       $\psi_j$  :=  $2\psi_j + \psi_{j+1}$
    od;
    $\psi_k$  :=  $2\psi_k$
od;
```

This multiplication-free [2] algorithm gives us the possibility of doubling the step of a pure-sum CR very fast. It will be used as one of the main tools for fast computation of products. Consider an example: given $i^3 - 2i + 1 = \Phi(i = 0, 1) = \{1, +, -1, +, 6, +, 6\}(i = 0, 1)$, the following intermediate CRs appear during computation of $U_2(\Phi)$:
$\{1, +, 4, +, 18, +, 12\} \to \{1, +, 4, +, 48, +, 24\} \to \{1, +, 4, +, 48, +, 48\} = U_2(\Phi)$.

A transformation U_v does not change the length of a pure-sum CR, and can be performed using additions only.

[2] Multiplication by 2 can be performed by hardware shift in NTL, or can be written as the addition in Maple.

It is somewhat more expensive in the case of CRs of the form $\Phi = \{\varphi_0, *, \Phi_1\}$ and $\Phi = \{\varphi_0, *, \frac{\Delta}{\Gamma}\}$, where Φ_1, Δ and Γ are pure-sum CRs. In the first case we have

$$\begin{aligned}&\Psi(i=0,v) = U_v(\{\varphi_0, *, \Phi_1\}(i=0,1)) = \\ &= \{\varphi_0, *, \Psi_1\}(i=0,v) = \\ &= \{\varphi_0, *, U_v(\Phi_1) \cdot U_v(E(\Phi_1)) \cdot \ldots \cdot U_v(E^{v-1}(\Phi_1))\}(i=0,v).\end{aligned}$$

Here Ψ_1 is a pure-sum CR obtained as the result of multiplication of v CRs $U_v(E^j(\Phi_1))$, $j = 0, \ldots, v-1$ and is v times longer that CR for Φ_1. In the second case we have

$$\begin{aligned}&\Psi(i=0,v) = U_v(\{\varphi_0, *, \tfrac{\Delta}{\Gamma}\}(i=0,1)) = \\ &= \{\varphi_0, *, \tfrac{\hat{\Delta}}{\hat{\Gamma}}\}(i=0,v) = \\ &= \{\varphi_0, *, \tfrac{U_v(\Delta) \cdot U_v(E(\Delta)) \cdot \ldots \cdot U_v(E^{v-1}(\Delta))}{U_v(\Gamma) \cdot U_v(E(\Gamma)) \cdot \ldots \cdot U_v(E^{v-1}(\Gamma))}\}(i=0,v).\end{aligned}$$

Here again the numerator and the denominator of Ψ_1 are pure-sum chains v times longer than Δ and Γ respectively.

Example. Let $\Phi(i=0,1) = \{1, *, 1, +, 1\}$ which defines $i!$ for $i = 0, 1, 2, \ldots$.
$U_2(\Phi) = \{1, *, U_2(\{1, +, 1\}) \cdot U_2(\{2, +, 1\})\} = \{1, *, \{1, +, 2\} \cdot \{2, +, 2\}\} = \{1, *, \{2, +, 10, +, 8\}\} = \{1, *, 2, +, 10, +, 8\}(i = 0, 2)$, which defines $i!$ for $i = 0, 2, 4, \ldots$ (or, what is the same, $(2i)!$ for $i = 0, 1, 2, \ldots$).

2.3 Bit-complexity vs operational complexity.

In this subsection we show that theoretical estimates of complexity and practical timings may differ. In other words bit-complexity is not the only issue to worry about when we try to accelerate computations of products.

Let us start with simple observations. Consider the computation of c^n where $n = 2^k$, and let $L(a)$ denote the bit length of an arbitrary precision integer a. Assume that it costs $L(a)L(b)$ bit operations to compute the product ab (no advanced multiplication algorithm is in use), and that $L(ab) = L(a) + L(b)$ (which is often the case and simplifies further estimates). Compare the complexity $C_r(n)$ of repeated multiplication ($2^k - 1$ multiplications) and the complexity $C_b(n)$ of binary powering (k squaring):

$$\begin{aligned}&C_r(n) = L(c)^2 + 2L(c)^2 + \ldots + (2^k - 1)L(c)^2 = \\ &\quad L(c)^2 \textstyle\sum_{i=1}^{2^k-1} i = L(c)^2 \left(\frac{1}{2}(2^k)^2 - \frac{1}{2}2^k\right); \\ &C_b(n) = L(c)^2 + 4L(c)^2 + \ldots + (2^{2k-2})L(c)^2 = \\ &\quad L(c)^2 \textstyle\sum_{i=0}^{k-1} 2^{2i} = L(c)^2 \left(\frac{1}{3}4^k - \frac{1}{3}\right); \\ &\tfrac{C_r(n)}{C_b(n)} = \tfrac{3}{2} \tfrac{4^k - 2 \cdot 2^{k-1}}{4^k - 1} \approx \tfrac{3}{2}.\end{aligned}$$

This simple estimate coincides with a pessimistic statement about binary powering found in [7]: "If we wish to calculate the exact multiple-precision value of x^n, when x is an integer greater than the computer word size, binary methods are not much help unless n is so huge that the high-speed multiplication routines of Section 4.3.3 are involved; and such applications are rare." Our experiments (see Table 2) show that even when multiplication is naive and x is an integer greater than the computer word size, binary powering is more than 2 times faster than repeated multiplication. This has a simple explanation: the operational complexity of binary powering is much smaller than that of repeated multiplication. There is a consequent reduction of related complexity factors, such as loop organization or memory management overheads. When x is an integer smaller than the computer word size, binary powering is much faster. When high-speed multiplication is available, binary powering is vastly superior.

For an unrolling powering, the situation remains the same. Return to the program in section 2.1 and estimate the complexity C_r of repeated multiplication versus the complexity C_u of unrolling powering with the same assumptions about L as above and $n = kv$:

$$\begin{aligned}&C_r(n) = L(c)^2 + 2L(c)^2 + \ldots + (kv - 1)L(c)^2 = \\ &\quad L(c)^2 \textstyle\sum_{i=1}^{kv-1} i = L(c)^2 \left(\frac{1}{2}k^2v^2 - \frac{1}{2}kv\right); \\ &C_u(n) = L(c)^2 + 2L(c)^2 + \ldots + (v-1)L(c)^2 + \\ &\quad +L(b)^2 + 2L(b)^2 + \ldots + (k-1)L(b)^2 = \\ &L(c)^2 \textstyle\sum_{i=1}^{v-1} i + L(b)^2 \sum_{i=1}^{k-1} i = \\ &\text{(taking into account that } L(b) = L(c^v) = vL(c)) \\ &= L(c)^2 \tfrac{1}{2}\left(k^2v^2 - v^2k + c^2 - v\right). \\ &\tfrac{C_r(n)}{C_u(n)} = 1 + \tfrac{kv - k - v + 1}{vk^2 - kv + v - 1}.\end{aligned}$$

This analysis suggests that speedup because of unrolling is less than $1 + \frac{1}{\sqrt{n}}$ in the best case, when k and v are taken approximately equal to each other. Experiments in Maple, NTL and Java (Table 2) show that this is not the case.

	Maple 5.3 (no K)	Maple 5.5	NTL (no K)	NTL	Java
c_1^{4500}					
RM (time)	10.0	9.26	1.76	1.64	6.70
BP	10.3	35.2	6.3	14.9	9.2
UF 30	5.9	6.8	3.0	3.0	5.8
UF 60	6.5	8.9	3.1	4.0	6.4
UF 90	6.7	11.0	3.0	4.5	6.8
c_1^{9000}					
RM (time)	40.0	37.65	7.09	7.08	24.44
BP	9.4	52.8	6.8	32.1	8.9
UF 30	5.9	6.7	2.6	2.6	5.2
UF 60	6.5	9.1	2.5	3.4	5.7
UF 90	6.9	11.2	2.7	4.0	5.9
UF 120	7.1	11.9	2.6	4.6	6.0
c_2^{4500}					
RM (time)	193.0	164.78	53.66	53.06	108.69
BP	2.4	27.1	3.2	28.37	2.5
UF 30	1.7	4.3	1.1	2.5	1.7
UF 60	1.8	5.8	1.1	3.3	1.7
UF 90	1.8	7.2	1.1	3.9	1.8
c_2^{9000}					
RM (time)	809.0	681.34	220.03	220.10	477.68
BP	2.7	34.8	3.3	38.9	2.6
UF 30	1.5	4.5	1.1	2.5	1.7
UF 60	1.5	6.0	1.1	3.4	1.8
UF 90	1.6	7.4	1.1	4.0	1.8
UF 120	1.6	7.9	1.1	4.5	1.8

Table 2: Time for repeated multiplication powering and speedups given by binary powering and unrolled powering in 3 systems (time in sec.); $c_1 = 2^{15} - 1$; $c_2 = 2^{120} - 1$. UF stands for unrolling factor, RM stands for repeated multiplication, no K means the Karatsuba algorithm is disabled (Maple 5.3 is the last release of Maple which does not use the Karatsuba algorithm for integer multiplication).

The speed improvement for unrolling is not as impressive as that for the binary powering, but it is much better than theoretical estimates indicate. The improvement depends on the size of the base and the system. It is also much better when the Karatsuba algorithm is enabled, because unrolling gives more opportunities for this algorithm to be applied. We can conclude from this discussion that for our purposes the main emphasis must be (a) trading multiplications for additions and (b) reducing the size of multiplicands. The cost of the addition of arbitrary precision integers is almost negligible in comparison with the cost of multiplication when the length of the numbers grows.

3 Unrolling and CRs

In this section we describe a combination of unrolling and CR-techniques to accelerate computations of products. The general approach is simple. Given (1) and n let $a_i = f(i)$ be polynomial or rational function in i. We can reformulate (1) in terms of CR

$$\Phi(i = 1, 1) = \{1, *, \Phi_1\}(i = 1, 1),$$

where Φ_1 is a pure-sum CR of the length l for $f(i)$ if $f(i)$ is a polynomial of degree l, or a CR-expression of the form Δ/Γ if $f(i)$ is a rational function (here Δ of the length l_1 is a pure-sum CR for the numerator of $f(i)$ and Γ of the length l_2 is a pure-sum CR for the denominator of $f(i)$).

Now let $n = kv$. Apply U_v to Φ and obtain the CR

$$\Psi(i = 1, v) = \{1, *, \Psi_1\}(i = 1, v). \tag{5}$$

Computation of $\texttt{Value}(E^k(\Psi))$ gives the value $F(n)$. Each application of E to Ψ costs 1 multiplication and lv additions if $f(i)$ is a polynomial of degree l, or 1 multiplication, 1 division and $v(l_1 + l_2)$ additions if $f(i)$ is a rational function with a numerator of degree l_1 and a denominator of degree l_2.

This approach relies on the proper choice of v, which in the general case is not obvious. Consideration of concrete problem opens more opportunities for improvements.

Further in this section we assume that v will be equal to some power of 2. The difference with an arbitrary v is in the complexity of factor v unrolling. If v is some power of 2 the unrolling can be done by consequent application of U_2 in $O(v^2 \log v)$ additions, while in the case of arbitrary v it takes $O(v^3)$ additions.

3.1 Factorial

Given a large integer n, compute $F(n) = n!$. Rewrite the product definition of $n!$ using the unrolling equation (2):

$$F(n) = \prod_{i=1}^{n} i = \underbrace{\prod_{i=1}^{k} \left(\prod_{j=1}^{v} ((i-1)v + j) \right)}_{\text{denote this term } A(k,v)} \cdot \prod_{j=1}^{l} (kv + j). \tag{6}$$

We are concerned about the first term $A(k, v)$ in this product since $l < v$. The innermost product in j defines v-degree polynomial in i

$$p_v(i) = ((i-1)v + 1)((i-1)v + 2) \ldots ((i-1)v + v)$$

which has to be computed for $i = 1, 2, \ldots, k$. Now it is not surprising that the CR-technique can help here, providing each value of this polynomial for the price of v additions. The question is – how to construct a CR for $p_v(i)$ reasonably fast? The general CR-construction procedure requires $O(v^2)$ multiplications and $O(v^2)$ additions. We will show that we can accomplish it in at most v multiplications and $O(v^2 \log_2 v)$ additions if v is a power of 2 (or v multiplications and $O(v^3)$ additions for and arbitrary v; in either of this cases v multiplications is the price of computing $v!$).

First we observe that the polynomial
$\tilde{p}_v(i) = i(i+1) \ldots (i + v - 1)$ gives the same values for $i = 1, 1 + v, 1 + 2v, \ldots$ as $p_v(i)$ for $i = 1, 2, 3 \ldots$.
Secondly, observe that polynomial
$\hat{p}_v(i) = i(i-1) \ldots (i - v + 1)$ gives the same values for $i = v, 2v, 3v, \ldots$ as $p_v(i)$ for $i = 1, 2, 3 \ldots$. Now use the fact [12] that $\hat{p}_v(i)$ has a very special and easy to construct CR of the length v for i starting from 0 with step 1:

$$\hat{p}_v(i)(i = 0, 1) = \{0, +, 0, +, ..., 0, +, v!\}(i = 0, 1). \tag{7}$$

We can apply multiplication-free transformation U_2 to this CR $\log_2 v$ times and get

$$\begin{aligned} &\hat{p}_v(i)(i = 0, v) = \Phi(i = 0, v) = \\ &= \{\varphi_0, +, \varphi_1, +, \ldots, +, \varphi_v\}(i = 0, v). \end{aligned} \tag{8}$$

One application of shift finalizes the construction: the CR $E(\Phi)$ being interpreted will provide values of $p_v(i), i = 1, 2, 3, \ldots$. This means that we get the following CR to compute the first term in (6):

$$\Psi(i = 0, 1) = \{1, *, E(\Phi)\}. \tag{9}$$

Applying E to Ψ will consequently compute values of $v!, (2v)!, \ldots, (kv)!$ performing v additions and 1 multiplication at each step. After k shifts $\texttt{Value}(E^k(\Psi))$ gives the value of $A(k, v)$ in (6).

We can make scheme (9) even more effective involving binary powering into the computational process. It follows from definitions of $\texttt{Value}$, E and from basic properties of pure-sum CRs [11] that

$$\begin{aligned} &E(c\Phi) = cE(\Phi), \\ &\texttt{Value}(c\Phi) = c\texttt{Value}(\Phi), \\ &U_v(c\Phi) = cU_v(\Phi) \end{aligned} \tag{10}$$

for any constant c and a pure-sum CR Φ. Also for CR in (7) the following equality holds

$$\{0, +, ..., 0, +, v!\}(i = 0, 1) = v!\{0, +, ..., 0, +, 1\}(i = 0, 1).$$

Starting with the CR $\{0, +, 0, +, ..., 0, +, 1\}(i = 0, 1)$ instead of $\{0, +, 0, +, ..., 0, +, v!\}(i = 0, 1)$ we get after $\log v$ applications of U_2 a CR

$$\hat{\Phi}(i = 0, v) = \{\hat{\varphi_0}, +, \hat{\varphi_1}, +, \ldots, +, \hat{\varphi_v}\}(i = 0, v).$$

instead of (8) (note, that $\hat{\varphi_j} = \varphi_j / v!, j = 0, \ldots, v$) and write Ψ in (9) as

$$\Psi(i = 0, 1) = \{1, *, v!E(\hat{\Phi})\}. \tag{11}$$

It is now straightforward to check that

$$\begin{aligned} &A(k, v) = \texttt{Value}(E^k(\Psi)) = \\ &= \texttt{Value}(E^k(\{1, *, v!E(\hat{\Phi})\})) = \\ &= (v!)^k \texttt{Value}(E^k(\{1, *, E(\hat{\Phi})\})). \end{aligned} \tag{12}$$

With this transformation we replace k multiplications in (9) by k multiplications of $v!$-times smaller numbers plus one binary powering and one extra multiplication. For large enough n with proper choice of v and k this last scheme is faster, because it reduces the time of construction of CR $\hat{\Phi}$ and the time of interpretation of the scheme (brief analysis will be given later). Table 3 contains timing for scheme (12).

Remark 4. Obviously we could start consideration by taking out $v!$ factor from innermost product in (6), consider polynomial $p_v(i)/v!$ instead of $p_v(i)$ and arrive to the same scheme (11). For us it was important to show how this transformation works in CR notation, because formulae (10) can be used in the same manner in the general case. For example, if $f(i)$ in (1) is a polynomial, after obtaining (5) it can be useful to compute gcd of components of CR Ψ_1 — g, and take out the term g^k in the same way as we have done it with $(v!)^k$. It will decrease the size of components and the cost of multiplication and additions during shifting CR Ψ in (5). We did not need to compute this gcd in the case $f(i) = i$ because we knew its value $(v!)$ in advance. In the case of rational function $f(i)$ more complex analysis (e.g. computing of the dispersion of the numerator and denominator) can help to cancel larger common factor which will appear after unrolling. Sometimes this common factor can be derived without computation of gcd, as in the case of binomials.

	Maple		Sp.	NTL		Sp.	Java	Sp.
	time	$C+I+F$		time	$C+I+F$		time	
6400!								
RM	11.06			2.53			8.90	
UF 16	2.06	$0.03+1.88+0.15$	5.37	0.66	$0.0+0.55+0.11$	3.83	1.61	5.53
UF 32	1.81	$0.19+1.46+0.16$	6.11	0.55	$0.0+0.44+0.11$	4.6	1.56	5.71
UF 64	2.58	$1.32+1.07+0.19$	4.29	0.44	$0.0+0.33+0.11$	5.75	2.18	4.08
UF 128	14.69	$12.62+1.88+0.19$	0.75	0.55	$0.27+0.17+0.11$	4.6	6.95	1.28
12800!								
RM	54.97			9.61			40.34	
UF 32	6.55	$0.17+5.78+0.6$	8.39	2.58	$0.0+2.20+0.38$	3.72	6.24	6.46
UF 64	5.58	$1.2+3.73+0.65$	9.85	1.98	$0.0+1.65+0.33$	4.85	6.64	6.08
UF 128	17.81	$11.96+5.14+0.71$	3.09	1.65	$0.28+0.99+0.38$	5.82	11.05	3.65
UF 256	110.31	$106.73+2.83+0.75$	0.49	2.75	$1.7+0.61+0.44$	3.49	43.62	0.92
32000!								
RM	415.35			65.42			301.41	
UF 32	39.04	$0.2+35.94+2.9$	10.64	19.93	$0.0+18.29+1.64$	3.28	47.04	6.41
UF 64	28.01	$1.1+23.84+3.07$	14.83	14.83	$0.06+13.02+1.75$	4.41	43.87	6.87
UF 128	37.2	$11.95+21.88+3.39$	11.17	10.49	$0.22+8.41+1.86$	6.23	48.00	6.28
UF 256	126.13	$109.65+12.81+3.67$	3.29	8.90	$1.65+5.27+1.98$	7.35	78.74	3.83

Table 3: Speedup given by unrolling and CR techniques for computation of factorials in 3 systems (time in sec.): (UF – unrolling factor, RM – repeated multiplication, Sp – speed up; $C+I+F$ — times for construction of a CR, interpretation of the CR and binary powering together with final multiplication in (12).)

3.2 Binomials

Given large N and n compute

$$F(n) = \binom{N}{n} = \prod_{i=1}^{n} \frac{N-i+1}{i} \tag{13}$$

(we assume that $n \leq N/2$). For this computation there are several different first-order recurrences to start with, e.g.

$$\binom{N}{n} = \binom{N}{n-1}\frac{N-n+1}{n}, \binom{N}{n} = \binom{N-1}{n-1}\frac{N}{n},$$
$$\binom{N}{n} = \binom{N-1}{n}\frac{N}{N-n}.$$

We withdraw the third one because it requires $N-n>n$ steps to compute the result. First and second recurrences correspond to different orderings of terms in the numerator and the denominator of (13). We will start with the first one and apply the procedure described in the beginning of this section [3]. This procedure gives us the scheme (5) for the first term in product (2), where $\Psi_1 = \Delta/\Gamma$: Δ and Γ are both pure-sum CRs of the length v. It replaces v multiplications and v divisions by $2v$ additions, 1 multiplication and 1 division of larger numbers. However, as in the case of factorial, it is known in advance that all components of Δ and Γ are divisible by $v!$ and this factor can be cancelled from both numerator and denominator. Note, that it is not necessary to perform actual cancellation. For denominator we can compute the result of cancellation which is equal to the CR $\hat{\Phi}$ from previous subsection (it can be done again with $\log v$ applications of U_2 to the CR $\{0,+,0,+,\ldots,0,+,1\}$ of the length v). Construction of CR Δ is done in $v-1$ steps of CR multiplication. After the step number j all components of the intermediate result can be divided by $j+1$, which by the end will produce CR Δ with the factor $v!$ cancelled. Timing for different computations with different unrolling factor are given in Table 4.

3.3 Choosing the unrolling factor

In order to choose the unrolling factor for concrete problem we must carry out some complexity analysis. We can not rely on bit-complexity analysis only, because, as shown earlier, this will not give us true prediction of acceleration. That is why our choice of v is heuristic, partially based on the following observations about complexity.

Consider (6) and (12) and assume $l=0$ for simplicity. Informally we have the following trade-off:
1) we do not want v to be very large, because v is the length of CR $\hat{\Phi}$ in (12) [4].
2) we do not want k to be very large, because we have to perform k multiplications computing E^k in (12).
More formally:
1) the number of additions in the scheme (12) is always the same – $n=kv$; the size of components of CR $\hat{\Phi}$ is bounded by $2vL(v)$ (where $L(u)$ denotes the number of digits of integer u in corresponding number system); the cost of all additions is $2nvL(v)$.
2) the number of multiplications to compute E^k in (12) is k; j-th multiplier is equal to $((j-1)v)^{\overline{v}}/v!$; the cost of all multiplications is $O(n^2k\log((n+v-1)/(n-1))$ if the Karatsuba algorithm is disabled.
3) the time to construct the CR $\hat{\Phi}$ is $O(v^3\log^2 v)$ if v is some power of 2, or $O(v^4 \log v)$ otherwise.
4) there is yet 1 more binary powering to compute $(v!)^k$ and 1 final multiplication.
All this gives a non-linear equation to be solved for estimation the value of v. Enabling the Karatsuba algorithm makes the analysis more complicated. Also, memory management has to be taken into account. Empirical results suggest that good choice for v be some power of 2 close to $\sqrt{n/2}$.

[3] We do not discuss multiplication-free computation of $\binom{N}{n}$ using the Pascal triangle here, because of the huge storage requirements.

[4] It seems very attractive to set $k=2, v=n/2$ in (12) and compute $v!$ recursively with the similar scheme, but in this case `Value`$(E^k(\{1,*,E(\hat{\Phi})\})) = \binom{2v}{v}$ and CR $\hat{\Phi}$ keeps essentially the same information as the corresponding to $\binom{2v}{v}$ line of Pascal triangle.

	Maple	speed-	NTL	speed-	Java	
	time	up		up	time	speedup
$\binom{20000}{10000}$						
ST	15.12		3.00		10.45	
UF 25	3.57 (0.11)	4.23	1.16 (0)	2.59	2.17 (0.09)	4.82
UF 50	3.41 (0.72)	4.44	0.99 (0.02)	3.03	2.27 (0.66)	4.6
UF 100	4.62 (2.12)	3.27	0.94 (0.18)	3.19	3.96 (3.75)	2.64
UF 200	21.6 (19.94)	0.7	2.2 (1.56)	1.36	24.15 (23.95)	0.43
$\binom{50000}{10000}$						
ST	35.26		4.64		25.71	
UF 25	8.52 (0.12)	4.14	1.91 (0.01)	2.43	4.88 (0.10)	5.27
UF 50	7.55 (0.74)	4.67	1.57 (0.02)	2.96	4.83 (0.71)	5.32
UF 100	8.56 (2.25)	4.12	1.37 (0.18)	3.39	7.75 (3.80)	3.32
UF 200	26.35(20.72)	1.34	2.55 (1.58)	1.82	28.62 (24.28)	0.90
$\binom{50000}{25000}$						
ST	200.04		19.11		112.16	
UF 25	39.46(0.12)	5.07	8.15 (0)	2.34	18.73 (0.11)	5.99
UF 50	34.49(0.75)	5.80	6.67 (0.02)	2.87	16.09 (0.71)	6.97
UF 100	32.63(2.54)	6.13	5.42 (0.19)	3.53	17.90 (3.84)	6.27
UF 200	53.32(23.14)	3.75	5.8 (1.63)	3.29	38.77 (24.33)	2.89
$\binom{200000}{100000}$						
ST	4167.87		311.05		2089.42	
UF 25	654.3 (0.14)	6.37	149.28 (0)	2.08	288.47 (0.11)	7.24
UF 50	570.16 (0.77)	7.31	120.4(0.02)	2.58	234.31 (0.77)	8.92
UF 100	494.41 (2.56)	8.43	95.42(0.19)	3.26	205.70 (4.01)	10.16
UF 200	502.96(24.07)	8.29	78.96(1.63)	3.94	212.07 (25.26)	9.85

Table 4: Speedup given by unrolling and CR techniques for computation of binomials in 3 systems (time in sec.): (UF – unrolling factor, ST – straightforward use of the first-order recurrence, parenthesized times are the times for construction of CRs Δ and Γ.)

4 An analog to binary powering

Earlier we saw that binary powering was appearing in the factor v CR unrolling scheme to compute factorial. In this section we find a more direct analog to binary powering.

We write $n = k \cdot 2^w + l$, $1 \leq l < 2^w$. Assuming that $k!$ is computed we can use the following recurrence for $(k2^w)!$

$$(k \cdot 2^i)! = ((k \cdot 2^{i-1})!)^2 \cdot \binom{k2^i}{k2^{i-1}}, \quad i = 1, 2, \ldots, w. (14)$$

This is an analog to binary powering. At each step squaring and some extra work is being performed. This extra work here is of course more expensive than that in the case of powering, but the squaring operation dominates. For the second term in (14) we construct a CR of the form $\Phi(j = 0, 1) = \{\binom{2k}{k}, *, \frac{\Delta}{\Gamma}\}$. Straightforward use of this CR will double the number of steps in j between $\binom{k2^i}{k2^{i-1}}$ and $\binom{k2^{i+1}}{k2^i}$. But we can apply the transformation U_2 after every k steps in j. There are also several technical observations here influencing the effectiveness of the actual computations. The simplification of the basic recurrence for $\binom{2j}{j}$

$$\binom{2j}{j} = \binom{2j-2}{j-1} \frac{2j(2j-1)}{j^2} = \binom{2j-2}{j-1} \frac{2(2j-1)}{j}$$

is useful. If we additionally use factor v unrolling of this recurrence we reduce the size of the numerator and denominator (due to cancellation by $v!$) and as in the case of binomials we save $v-1$ multiplications and $v-1$ divisions at each step. Also there is factor $2^{\lfloor k/2 \rfloor}$ which can be taken out from numerator of the CR (in systems with base equal to power of 2 it will be cheaper to perform one hardware shift at the end). Table 5 gives timing for different values of n in Maple, C++(NTL) and Java.

	Maple	NTL	Java
12800!			
RM	53.38	9.66	40.76
BAF	4.38(3.19)	0.88 (0.44)	5.21 (1.21)
speedup	12.19	10.98	7.82
25600!			
RM	248.96	40.87	189.02
BAF	13.08(9.00)	3.13 (1.43)	22.74 (3.62)
speedup	19.04	13.06	8.31
51200!			
RM	1109.22	176.20	819.02
BAF	45.57(33.93)	10.76 (5.04)	102.05 (1.03)
speedup	24.34	16.28	8.03

Table 5: Speedup given by an analog to binary powering for computation of factorials (BAF) in 3 systems (time in sec.): (RM – repeated multiplication, parenthesized times are the times for computation of binomial coefficients in (14).)

5 Implementation

All algorithms described in this paper were implemented in Maple, C++ and Java. The Maple implementation extends the Maple CR package [6]. Parts of the routines were optimized for arbitrary precision arithmetic, which reduced construction time significantly.

Our C++ implementation was built on top of NTL library because of NTL's power and flexibility. The NTL library allows control over memory allocation and supports optional use of the Karatsuba algorithm. The implementation encapsulates a CR engine, which was specifically designed for this purpose, and provides the user with a set of classes and routines to compute products, including factorials, binomial coefficients, etc. The implementation is a C++ library, which can be linked to any application on the variety of platforms supported by NTL.

Java BigInteger numbers were chosen for our experiments mostly because of Java's popularity. Moreover, the performance constraints of Java Virtual Machines make it important to accelerate time-consuming operations. The Java implementation is similar to the C++ implementation. It is a package, containing classes for basic CR tools and wrapping classes for products-computing routines. This package can be easily included into any Java project.

6 Concluding remarks

We presented in this paper an approach to accelerating the computation of n-ary products for large n over arbitrary precision integers. The main tool was a combination of CR techniques, loop unrolling and binary powering. Return to Table 1 and compare the timings for new implementations (Table 6) of tasks **T2**, **T3** with the previous.

	T2	**T3**
Maple	7.75 (140.07)	823.17 (6153.48)
NTL	1.98 (24.39)	124.62 (488.52)
Java	14.11 (109.41)	299.72 (3461.73)

Table 6: Final time in seconds for **T2** and **T3** in 3 systems; time before improvement is given in brackets; all Maple timings are based on interpreted (library) code.

The technique proposed here is easy to combine with other fast methods to compute products. For example, one can find in [5] a fast formula to compute factorial, which is based on binary splitting algorithm combined with a reduction of the even factors into odd factors and multiplication by a power of 2. Application of the CR-based unrolling technique to the innermost product in this formula is easy to implement. Our preliminary experiments show a 2 to 3 times acceleration after such a combination, although the resulting scheme is still slower than (14).

A factor v unrolling of a CR for $f(i), i = 0, 1, \ldots$ corresponds to the substitution $i = vi$ into the closed form expression for f. There are two possibilities:
a) (numeric) to construct the CR for $f(i), i = 0, 1, \ldots$ after substitution; b) (symbolic) to construct a CR for $f(i), i = 0, h, 2h, \ldots$ symbolically and when the value of v is decided - to substitute $h = v$ in this CR. This will require numerical computations after the substitution, whose complexity is not better than the complexity of case a). In this paper we described an alternative numeric approach which is quite suitable for computation of factorials or binomials. In the general case (1) (when $f(i)$ is not known in advance) symbolic-numeric mixture of a) and b) can help, allowing symbolic simplifications.

It would be also interesting to carry out the series of experiments on a parallel architecture (especially SIMD type). The reason for this is that basic operations on CRs ($E(\Phi)$, $U_v(\Phi)$ etc.) are highly parallelizable and methods from [9] reduce the shift of a CR to two parallel operations only, e.g. one parallel shift and one parallel addition. This will change the heuristics for v and w. The choice of v and w will also be different if we apply this technique to the computations modulo a large natural number (not necessarily prime). It will move us closer to fixed point arithmetic. We also think that improvements to our technique may be possible if a modular implementation of arbitrary precision integer arithmetic is used. Also p-adic representation of numbers can bring new possibilities for further progress.

Acknowledgments

The authors would like to thank Chris Howlett (Web Pearls Inc.) for his help during preparation of this paper. The third author is grateful to Bruno Salvy (INRIA, France) for pointing out the reference [5] and for useful remarks to the first draft, and also to Ha Quang Le (University of Waterloo) for many discussions during the work on the paper.

References

[1] Bini D., Pan V. *Polynomial and Matrix Computations. Fundamental Algorithms, vol.1.* Birkhauser, 1994.

[2] Borwein J., Borwein P. *Pi and the AGM.* Wiley, 1987.

[3] Char B.W., Geddes K.O., Gonnet G.H., Leong B.L., Monagan M.B., Watt S.M. *Maple-V. Language Reference Manual.* Springer-Verlag, 1991.

[4] Cohen H. *A Course in Computational Algebraic Number Theory.* Springer-Verlag, 1995.

[5] Haible B., Papanikolaou T. Fast multiprecision evaluation of series of rational numbers. Technical report TI-97/7. Technical report, University of Darmstadt, 1997.

[6] Kislenkov V., Mitrofanov V., Zima E. Multidimensional Chains of Recurrences. In *Proc. of ISSAC'98, Rostock, Germany, ACM Press*, pages 199–206, 1998.

[7] Knuth D.E. *The art of computer programming. V.2. Seminumerical Algorithms. Third edition.* Addison-Wesley, 1997.

[8] Shoup V. `http://www.cs.wisc.edu/~shoup/ntl/`.

[9] Zima E. Recurrent relations technique to vectorize function evaluation in loops. In *PARCELLA'94, Potsdam, Germany*, pages 161–168, 1994.

[10] Zima E.V. Automatic construction of systems of recurrence relations. *USSR Comput. Maths. Math. Phys., N 6*, 24:193–197, 1984.

[11] Zima E.V. Simplification and Optimization Transformations of Chains of Recurrences. In *Proc. of ISSAC'95, Montreal, Canada, ACM Press*, pages 42–50, 1995.

[12] Zima E.V. Safe numerical computations with chains of recurrences. *Programmirovanie N 3*, pages 36–42, 1997.