LAPPEENRANTA UNIVERSITY OF TECHNOLOGY

School of Engineering Science

Double Degree Master’s Program in Computational Engineering and Mathematics

Ekaterina Sokolova

SOLVING LARGE SPARSE LINEAR SYSTEMS OVER THE FIELD GF(2)

Examiners: Ass. Prof. Tuomo Kauranne

Post. Doc. Matylda Jablonska-Sabuka

ABSTRACT

Lappeenranta University of Technology

School of Engineering Science
Faculty of Science and Technology
Double Degree Master’s Program in Computational Engineering and Mathematics

Ekaterina Sokolova

Solving large sparse linear systems over the field GF(2)

Master’s thesis

2016

65 pages, 7 figures, 14 tables

Examiners: Ass. Prof. Tuomo Kauranne

Post. Doc. Matylda Jablonska-Sabuka

Keywords: large sparse systems of linear equations, Montgomery method, Wiedemann-Coppersmith method

Nowadays problem of solving sparse linear systems over the field GF(2) remain as a challenge. The popular approach is to improve existing methods such as the block Lanczos method (the Montgomery method) and the Wiedemann-Coppersmith method.

Both these methods are considered in the thesis in details: there are their modifications and computational estimation for each process. It demonstrates the most complicated parts of these methods and gives the idea how to improve computations in software point of view.

The research provides the implementation of accelerated binary matrix operations computer library which helps to make the progress steps in the Montgomery and in the Wiedemann-Coppersmith methods faster.

Acknowledgements

I would first like to thank my thesis supervisor Ass. Prof. Tuomo Kauranne. Without his advice and help, the study could not have been successfully conducted. I would also like to thank the Department of Technomathematics for the warmth and friendly atmosphere.

I would like to acknowledge the Lappeenranta University of Technology and the Moscow Power Engineering Institute for the opportunity to take a part in a double degree program. It gave me the new experience, skills, friends, and ideas.

Finally, I must express my very profound gratitude to my mother and to my boyfriend Boris for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis.

Lappeenranta, April 14, 2016

Ekaterina Sokolova

Contents

1 INTRODUCTION

The problem of solving large sparse linear systems over the field GF(2) has widespread application but it has high computational complexity and there are few programs capable of solving the systems of linear equations fast. Among fields of use, there are problems such as breaking RSA encryption by sieve methods, applied to the problem of factoring a large natural number into two primes; and information data transfer by a noisy channel. Presently there are two algorithms widely using for large linear system solving over GF(2): Montgomery algorithm [1] and Wiedemann-Coppersmith algorithm [2, 3]. Over the last twenty years there have been attempts to accelerate individual steps of these algorithms. For example, in the last achievement of breaking the 768-bit RSA-key, it was implemented by a modified block Coppersmith algorithm [4]. Nevertheless, the two algorithms have their advantages, disadvantages and limitations. A question of special interest are implementation details: taking into account computing systems characteristics to choose data format and development priority. Such information is uncommon in professional papers, although it is not less important than advancement in general.

The goals of this paper are investigation of Montgomery and Wiedemann-Coppersmith algorithms, making analytical models for them, giving each a computational complexity estimate and practical results. Besides that, the goal is suboptimization as a result of theoretical and practical evidence of this work. To achieve the goals, the paper consider extended algorithms' models according to computing systems down to details of the most computationally heavy operations. The work provides special data format and an accelerated binary matrix operations computer library.

The first chapter presents a theoretical overview of Montgomery and Wiedemann-Coppersmith methods and their modifications used in this work. The second chapter considers an analytical model of computation for both algorithms which is used as a base for detailed theoretical estimates of the computational complexity of each step of both algorithms. The third chapter describes the software implementation of both algorithms and practical estimates of programming work. The fourth chapter provides a theoretical basis and practical results of accelerated operations of addition and multiplication of binary matrices which play an important role in the implementation of both algorithms.

1.1 The current state of the problem

Both methods, which will be discussed in this paper are iterative methods that use Krylov space. In the case of the real field there are successful applications of some similar methods: first of all, the Lanczos algorithm [5], which is the basis for Montgomery's work, the methods of Conjugate Gradients[6], the generalized minimum residual algorithm (GMRES) [7] and others. All listed algorithms in standard form are unsuitable for the finite fields case. For the GF(2) case, there are several implementations of the so-called block Lanczos algorithm ([1], [8], [9]). This paper considers only Montgomery's work because there is no essential difference between other version block methods. The second algorithm presented in this work is the Wiedemann-Coppersmith algorithm that was first proposed in 1986 by Wiedemann ([2]). Later in 1995 Coppersmith published a block version of it ([3]). The Wiedemann-Coppersmith method uses the characteristic polynomial of a matrix. A common way to construct this polynomial is to apply one of the so-called fast methods for it, for example, Berlekamp–Massey algorithm [10], the superfast algorithm from [11] and Thomé's modification [12].

1.2 Montgomery method

Take a system

$$Ax = b,$$

where *A* is a large sparse symmetric *n*-by-*n* matrix, *b* and *x* are a vector of length *n*. The Lanczos idea is to construct an approximation to its solution by employing a Krylov space at every step that can be determined as

$$K_m(b, A) = Span\{b, Ab, A^2b, \dots, A^{m-1}b\}.$$

In the Lanczos algorithm vectors w_i should be

- non-zero: $w_i \neq 0$ if $0 \leq i < m$, $w_m = 0$;
- orthogonal with respect to the system matrix $w_j^T A w_i = 0$ if $i \neq j$;
- $w_i^T A w_i \neq 0$;
- The linear span of the vectors w_i defines the Krylov space $K_m = span\{w_1, \cdots, w_m\}$.

If these conditions are true, vectors w_i will be basis vectors for a sequence of Krylov spaces.

As an initial approximation, Lanczos proposes the vector *b* from the right-hand side of the system. The further process resembles Gram-Schmidt orthogonalization. Following vectors will be calculated as

$$w_i = Aw_{i-1} - \sum_{j=0}^{i-1} w_j c_{ij}, \quad (1)$$

where $c_{ij} = \frac{w_j^T A^2 w_{i-1}}{w_j^T A w_j}$, while $w_i \neq 0$.

The approximation formula on the m-iteration is

$$x_m = \sum_{j=0}^{m} \frac{w_j^T b}{w_j^T A w_j} w_j = x_{m-1} + \frac{w_m^T b}{w_m^T A w_m}.$$

At first glance, it may seem that the computations are complex because all w_i are needed in the composition. In fact, only two of them are non-zero when A is symmetric. The conclusion of the mentioned conditions for vectors w_i is a statement

$$w_j^T A^2 w_{i-1} = \left(Aw_j\right)^T Aw_{i-1} = \left(w_{j+1} + \sum_{k=0}^{j} c_{j+1,k} w_k \right)^T Aw_{i-1} = 0 ,$$

if $j < i-2$. Simplification of expression (1) gives

$$w_i = Aw_{i-1} - c_{i,i-1} w_{i-1} - c_{i,i-2} w_{i-2} ,$$

if $i \geq 2$. The bottleneck of this approach is in statement $w_i^T A w_i \neq 0$. In the case when it is not true, the method could create some approximation x_{m-1} instead of a true solution. It is a critical property for finite fields because of high probability for such condition. Lanczos algorithm in standard form is not suitable for GF(2). Foremost about half of all vectors in a vector space over the Galois field of two elements are self-orthogonal vectors, so, the solution will not be found with probability close to one. Secondly, even the smallest difference is crucial for the solution vector in GF(2) and cannot be tolerated [13].

With respect to GF(2) Montgomery reformulated conditions for Lanczos vectors w_i. Primarily he replaces the condition $w_i^T A w_i \neq 0$ by A-invertibility. By definition, a vector space is called A-invertible if it has such a basis that $W^T AW$ is invertible. By Montgomery, the algorithm is based on subspaces W_i instead that comprises vectors such that

- $W_i^T AW_i$ is invertible
- $W_j^T AW_i = 0$ is satisfied if $j \neq i$, thus W_i should be pairwise orthogonal
- $AW \in W$ that means W is an invariant space of matrix A

Let us assume that an n-by-N matrix is called a block vector if N is small compared to n. We will discuss the value of N in the chapter about algorithm implementation. Then the solution of the system $Ax = b$ is given by

$$x = \sum_{j=0}^{m-1} W_j \left(W_j^T AW_j\right)^{-1} W_j^T \, b.$$

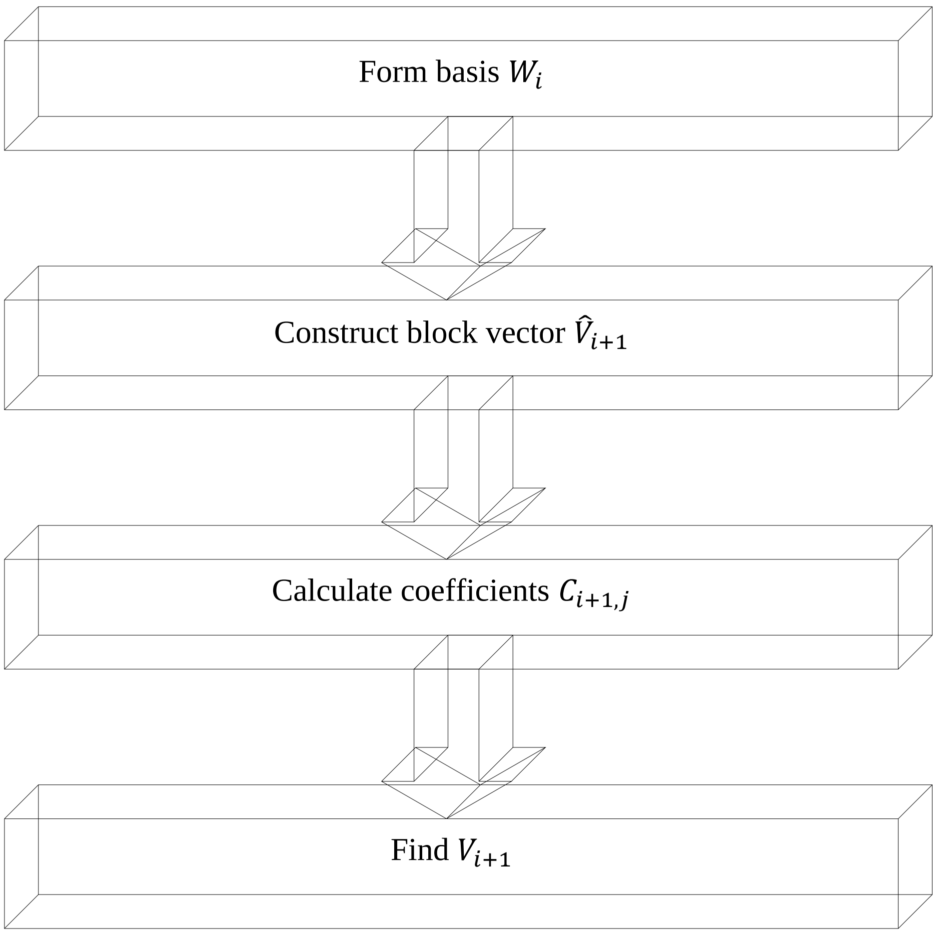
To construct the necessary sequence of spaces, a procedure is needed. The first step is to choose randomly a block vector V_0. The next steps assumes using maximum linear independent vector-columns chosen from V_i to satisfy the listed characteristics for W_i. Gauss elimination for $V_i^T AV_i$ is suitable in this case. Obviously, the vectors sought are a linear basis of subspace W_i. Other columns fill the array $\widehat{W}_j$, which will be needed on the next step. If all leftover columns do not satisfy the conditions stated for W_{i+1}, the next iteration will be impossible and the solution will not be found.

The following V_i are formed as

$$V_{i+1} = \hat{V}_{i+1} + W_i C_{i+1,i} + W_{i-1} C_{i+1,i-1} + W_{i-2} C_{i+1,i-2},$$

here $\hat{V}_{i+1}$ is the column-wise concatenation $[AW_i, \widehat{W}_i]$; $C_{i+1,j} = \left(W_j^T A W_j\right)^{-1} W_j^T A \hat{V}_{i+1}$. Iterations continue until V_m will be zero. It will happen if the solution has been found. Comparing to the Lanczos algorithm, in the Montgomery modification one additional summand is needed as an adaptation to GF(2) characteristics.

Montgomery algorithm steps:


Form basis W_i
Construct block vector $\hat{V}_{i+1}$
Calculate coefficients $C_{i+1,j}$
Find V_{i+1}

1.3 Wiedemann-Coppersmith method

In the Wiedemann-Coppersmith algorithm the system solution bases on the characteristic polynomial of the invertible square matrix *A*:

$$f(\lambda) = 1 + \sum_{i=1}^{n} f_i \lambda^i ,$$

here 1 is a value of the summand when $i = 0$.

By the Cayley-Hamilton theorem $f(A) = 0$. Thus, if $f(A)b = 0$, then

$$b = \sum_{i=1}^{n} f_i A^i b = A\left(\sum_{i=0}^{n-1} f_{i+1} A^i b\right)$$

is concluded. The solution to the system could be formulated as a linear combination of Krylov vectors

$$x = \sum_{i=0}^{n-1} f_{i+1} A^i b .$$

Coefficients of the characteristic polynomial are computed step-by-step. Initially, the Wiedemann method suggests a random vector $y \in F_2^n$. The statement

$$y^T f(A) A^j b = y^T \left(\sum_{i=0}^{n} f_i A^i\right) A^j b = \sum_{i=0}^{n} \left(y^T A^{i+j} b\right) f_i = 0,$$

is true, if $j = 0, \ldots, n$.

Let $\alpha_i = y^T A^i b \in F_2$ for all $i = 0, \ldots, n$, then the statement can be reformulated as $\sum_{i=0}^{n} \alpha_{i+j} f_i = 0,\ j = 0, \ldots, n$. Here α_{i+j} is a Hankel matrix, i.e. by definition it has equal elements on each ascending skew-diagonal. Krylov spaces also are used to calculate this matrix as in Montgomery algorithm. A transposed Hankel matrix is a Toeplitz matrix. It is important because there are fast computational algorithms for these matrices.

Coppersmith proposed ([3]) a modification to the Wiedemann algorithm. He manipulated with blocks of vectors. Computationally it seems more efficient because computers operate with machine words, i.e. block of binary vectors.

According to Coppersmith, let us take two random block vectors $z \in F_2^{n \times M}$ and $x \in F_2^{n \times N}$, where *n* is the size of the system matrix; *N*, *M* are random and small in comparison to *n*.

The value of *N* and *M* will be important in term of algorithm implementation. A further step is calculating $y = z \cdot A$. The formula

$$L = \frac{n}{N} + \frac{n}{M} + O(1).$$

displays how many coefficients $a_i = X^T A^i Y$ are in a sequence of Krylov spaces. Then the polynomial of the matrix will be given as the sum $a(\lambda) = \sum_{i=0}^{L} a_i \lambda^i$. It will be useful as initial data in a modification of the Berlekamp–Massey algorithm that obtains characteristic polynomial $f(x)$ such that

$$a(\lambda) f(\lambda) = g(\lambda) \bmod \lambda^L, \quad \deg g(\lambda) < \deg f(\lambda)$$

In the mentioned work [10] the modification uses a matrix as the set of coefficients of the characteristic polynomial. The task is to find a linear combination of vectors $\{A^i y_\nu, \quad 0 \le i \le n/N, \quad 1 \le \nu \le N\}$ that are orthogonal to each $\{x_\mu^T A^i, \quad 0 \le i \le n/M, \quad 1 \le \mu \le N\}$.

The *i*-th iteration of the Berlekamp–Massey algorithm gives the $(M+N)\times N$ matrix-coefficient for $f^{(i)}(\lambda)$ where the power is in GF(2). There should be a condition check for the polynomial power upper bound $\deg_{nom} f^{(i)} \le t$. Index ‘nom’ means that the polynomial power is usually higher than the actual value. Each coefficient by power λ_j in $f^{(j)}(\lambda) a(\lambda)$ should satisfy two conditions:

- all coefficients by powers less then t must be zero (C1)
- rank of a matrix-coefficient by power t must be equal to M(C2)

Let us set $t_0 = ceil(M/N)$ as the initial power and put polynomials with power less then t_0 into the first M columns. If C2 is not true, there are two ways to proceed: take another block of *x* vectors or increase power t_0. The last n rows $f^{(t_0)}$ could be taken from the matrix product: the first operand is the coefficient for power t_0, the second one is identity *N*-by-*N* matrix.

$$f = \left.\underbrace{\begin{pmatrix} P^{(t\le t_0)} & \dots & P^{(t\le t_0)} & \lambda^{t_0} & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \underbrace{P^{(t\le t_0)} \quad \dots \quad P^{(t\le t_0)}}_{M} & & & \underbrace{0 \quad 0 \quad \lambda^{t_0}}_{N} & & \end{pmatrix}}_{M+N}\right\} N ,$$

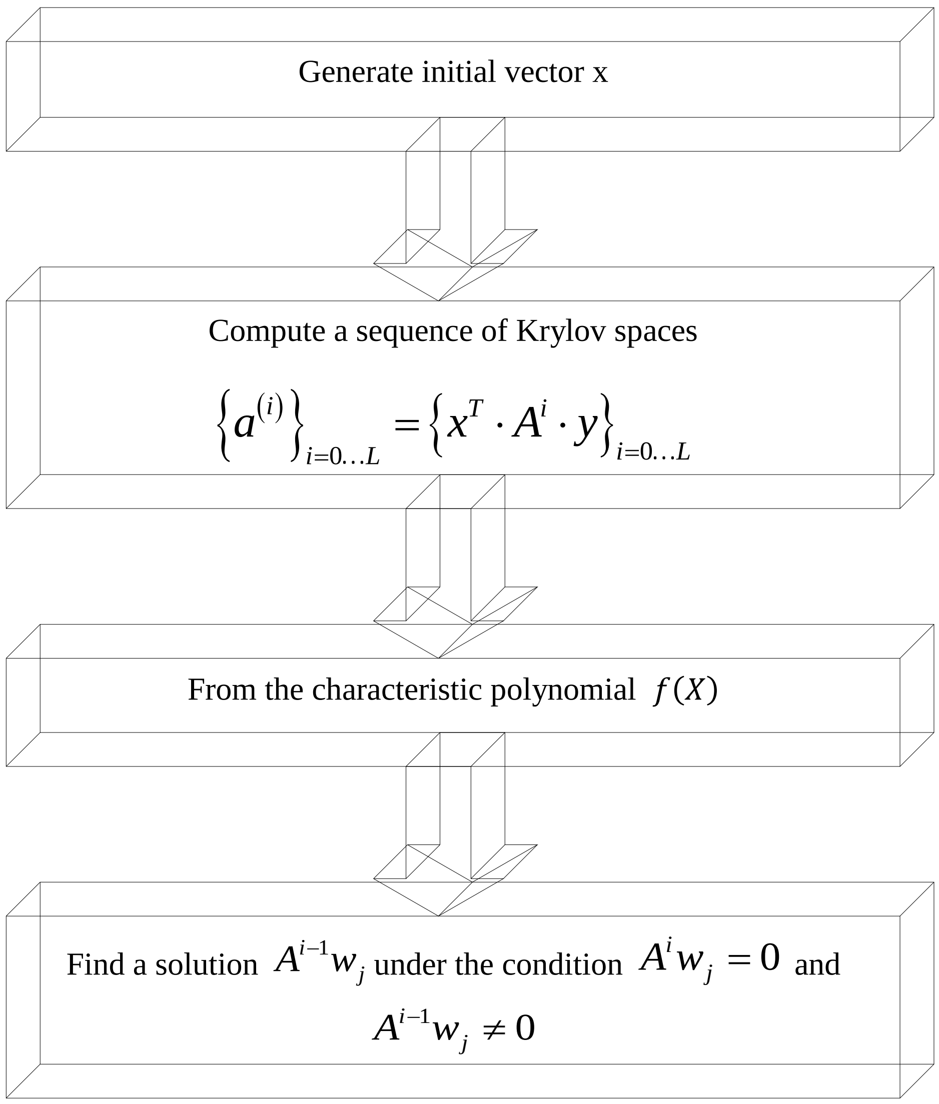
here $P^{(t\le t_0)}$ is a random polynomial with power less then t_0. Next iterations will compute $f^{(t+1)}$ from $f^{(t)}$. To organize its columns of matrix-coefficients by the current power t should be ordered and diagonalized. Columns of the identity matrix should be put in the same order. Zeroing all columns except *m* linearly independent ones (as C2 say) make C1 true for all other columns. To satisfy C1 for power $t+1$, these columns need to be multiplied by *x*. Iteration matrix could help to do all necessary computations and give a characteristic polynomial for the next step $f^{(t+1)}(x) = f^{(t)}(x)P^{(t)}(x)$. The aim of iterations is to increase the difference between power values until $t-\bar{\delta} > n/M$, $\bar{\delta}$ is a mean power relative to the numerical order of non-zero columns in matrix-coefficients of the characteristic polynomial f

$$\bar{\delta} = \frac{1}{M+N} \cdot \sum_{i=1}^{M+N} \delta_i$$

therefore, *t* increments yield an $\frac{M}{M+N}$ increase of $\bar{\delta}$. If the computational process of forming characteristic polynomial breaks under the condition C1 and C2, there are *l* columns in f (under the condition $t - \deg_{nom} f_l^{(t)} > n/M$) such that the vector $\sum_{\nu,k} f_{l,\nu}^{(t,k)} A^{d-k} y_\nu$ is orthogonal to vectors $x_\mu^T A^{j-d}$ while $1 \le \mu \le M, \quad d \le j < t$. At this point, the solution is found with high probability. Proof and details are considered in the work [3].

This chapter uses the Berlekamp–Massey algorithm [10] because it is easy-to-use. There are optimized version for it in references [11] and [12].

To sum up the Wiedemann-Coppersmith method:


Generate initial vector x
Compute a sequence of Krylov spaces
$\left\{a^{(i)}\right\}_{i=0\ldots L}=\left\{x^T \cdot A^i \cdot y\right\}_{i=0\ldots L}$
From the characteristic polynomial $f(X)$
Find a solution $A^{i-1}w_j$ under the condition $A^i w_j=0$ and $A^{i-1}w_j \neq 0$

2 COMPUTATIONAL MODELS

2.1 Computational model and computational cost for the Montgomery method

In theoretical overview, we say that the column size for a block vector is computationally important. To take into account the length of machine words, we could optimize algorithm implementation. Binary operations allow us to compute matrix-by-vector multiplication simultaneously if the number of vectors is less than the length of the machine word. Modern machines operate with 64-bit words (32-bit words in other cases). Therefore, the size of block vectors should be divisible by the length of the machine word. The Montgomery method manipulates columns of W_i simultaneously (it equals the rank of V_i). Rank value has a probabilistic impact onto the computational cost. According to Montgomery's evaluation in the case of GF(2) ([1]) rank expectation is

$$Er_N \sim N - 0.764499780,$$

if values of *N* and *n* are large.

Then, the number of iterations could be estimated as

$$\frac{n}{N - 0.764499780}.$$

Let us consider arithmetic operations associated with elements of matrices step-by-step.

1. Gauss elimination determines basis vectors in V_i. The number of operations for this step is

$$\sum_{k=1}^{N} \sum_{i=1}^{N} \Pr_1 \sum_{j=k}^{N} 1 = \Pr_1 \frac{N^2 (N-1)}{2},$$

here $N \times N$ is the $V_i^T A V_i$ matrix size, Pr_1 is the probability for matrix element $a_{ij} = 1$ if $a_{ij} \in \{0,1\}$.

2. To get the Krylov space and its vector $\hat{V}_{i+1}$, we need to multiply our matrix by the block vector AW_i. Sizes of matrices are A $[n \times n]$ and W_i $[n \times M]$. Hence, this process has $\prod(n, M, n)$ operations disregarding vector concatenation $[AW_i, \widehat{W}_i]$. The $\prod(n, M, n)$ means the function which determines how many operations would be needed to multiply matrices of size $[n \times m] \times [m \times k]$. This generalization simplifies the estimation of multiplication count per all operations depending on the method.

3. Each iteration involves three coefficients $C_{i+1,i}, C_{i+1,i-1}, C_{i+1,i-2}$. The last two of them are almost computed on the previous steps except the summand $\hat{V}_{i+1}$. The computational cost for the last two coefficients is two multiplications of $[M \times n] \times [n \times N]$ matrices or $2\prod(M, N, n)$ operations. The third coefficient $C_{i+1,i}$ includes

   a. W_j^T-by- AW_j multiplication with $\prod(M, M, n)$ operations

   b. Inverse of $(W_j^T A W_j)^{-1}$ matrix with $Pr_1 M^2 (M-1)$ as forward and backward paths of Gauss elimination

   c. $(W_j^T A W_j)^{-1}$ -by- $W_j^T A = AW_j$ multiplication with $\prod(M, n, M)$ operations

   d. $(W_j^T A W_j)^{-1} W_j^T A$ -by- $\hat{V}_{i+1}$ multiplication with $\prod(N, n, M)$ operations.

4. V_{i+1} computation is the last step. Two summands $W_{i-1}C_{i+1,i-1}$ and $W_{i-2}C_{i+1,i-2}$ are calculated on the previous iterations. To achieve the solution we compute

a. $W_iC_{i+1,i}$ product with $\prod(n, M, N)$ operations

b. Addition $\hat{V}_{i+1} + W_iC_{i+1,i} + (W_{i-1}C_{i+1,i-1} + W_{i-2}C_{i+1,i-2})$ with $2\sum(nN)$ operations

To summarize, the number of operations for one iteration lets us combine the estimates:

$$\begin{aligned}
iteration = &\Pr_1 n\frac{N(N-1)}{2} + \prod(n, M, n) + 2\prod(M, N, n) + \prod(N, n, \mathrm{M}) + \\
&+ \Pr_1 M^2(M-1) + \prod(n, M, M) + \prod(n, n, M) + \prod(N, n, M) + \\
&+ \prod(n < M, N) + 2\sum(nN)
\end{aligned}$$

Round off $M \approx N$ and $\prod(M, N, n) \approx \frac{N}{n}\prod(n, M, n)$ give a shorter formula

$$\begin{aligned}
iteration \approx &\Pr_1 \frac{N^2(N-1)}{2} + \left(1 + \frac{6N}{n}\right)\prod(n, M, n) + \Pr_1 M^2(M-1) + \\
&+ 2\sum(nN).
\end{aligned}$$

The number of iterations is also known

$$complexity = \frac{n \cdot iteration}{M}.$$

It is clear, that the most complex part is the multiplication by a large matrix. Montgomery iteration has two of them: when Krylov space is computed at the second step and in calculations of the coefficient at the third step. The statement of the problem says that $n \gg N, M$, that allows us to simplify the estimate to

$$complexity = \left(\frac{n}{M} + 6\right)\prod(n, M, n).$$

The column number M in matrix W_i is a probabilistic value and it makes the above-listed estimates probabilistic too.

2.2 A computational model and the computational cost for the Wiedemann-Coppersmith model

There are several steps in the Wiedemann-Coppersmith algorithm:

1. Generation of random vectors is an initial data process. It is not taken into account.
2. Each element in a sequence of Krylov spaces demands two multiplications, one of them is a *y*-by-*A* multiplication. The further step is multiplication of the result by *x*. The value of $A^{i-1} \cdot g$ was obtained previously. The size of *A* is $[n \times n] \times [n \times N]$ and the size of x is $[M \times n] \times [n \times N]$. Hence, there are *L* iterations with complexes $\prod(n, n, M) + \prod(M, n, N)$.
3. The major part of operations consists of obtaining coefficients for the characteristic polynomial, Gauss elimination, and polynomial multiplication:
    a. Coefficients could be found as a sequence of $N \times (M + N)$ matrix products and sums with $M \times N$ matrices a_i from a sequence of Krylov

spaces. Computations for power t take $(t+1)$ multiplications $\prod(M,N,M+N)$ and t sums $\sum(N,N)$.

b. Gauss elimination for matrices $e_{[M\times(M+N)]}$ and $P_{[(M+N)\times(M+N)]}$ feature $\Pr_{1,e} M\frac{(M+N)(M+N-1)}{2}$ and $\Pr_{1,P}\frac{(M+N)^2(M+N-1)}{2}$ operations, respectively.

c. The power of $P_{[(M+N)\times(M+N)]}$ polynomial equals to one, therefore, there are only two coefficients, so $3(t+1)$ multiplications containing $\prod(N,M+N,M+N)$ operations and $3t$ additions containing $\sum(N,M+N)$ operations are needed.

4. The final step requires the calculations to compute the w vector. It includes an $A^i_{[n\times n]}$-by-$z_{[n\times N]}$ multiplication and multiplying the result by $f_{(L-i)}$, of size $[N\times(M+N)]$. Consider that the multiplier $A^{i-1}\cdot z$ uses values saved at the previous step. There are L+1 iterations before we check the result columns for the condition $A^i w_j = 0$ and $A^{i-1}w_j \neq 0$. According to the estimate from [13] $i \approx 2\left(\frac{n}{M}+1\right)$.

The total computational cost for the Wiedemann-Coppersmith algorithm is provided by

$$complexity = L\left(\prod(n,n,M)+\prod(M,n,N)\right)+(t+1)\prod(M,N,M+N)+t\sum(N,N)+$$
$$\Pr_{1,e} M\frac{(M+N)(M+N-1)}{2}+\Pr_{1,\mathrm{P}}\frac{(M+N)^2(M+N-1)}{2}+3(t+1)\prod(N,M+N,M+N)+$$
$$+3t\sum(N,N+M)+\prod(n,N,M+N)+2\left(\frac{n}{M}+1\right)\prod(n,n,1).$$

In the Wiedemann-Coppersmith algorithm, the computationally hardest part features a large matrix A as a multiplier. It occurs three times: in a sequence of Krylov spaces and twice at the last step, when a vector w_i is calculated and checked. The number of iterations in this method is equal to the length of a sequence of Krylov spaces $L = \frac{n}{N} + \frac{n}{M} + O(1)$. We retain the largest summand and get

$$complexity = L\prod(n,n,N) + (L+1)\prod(n,n,N) + 2\left(\frac{n}{M}+1\right)\prod(n,n,1) \approx$$
$$\approx \frac{2\cdot n\cdot\prod(n,n,N)}{M} + 2\cdot\left(\frac{n}{M}+1\right)\prod(n,n,1).$$

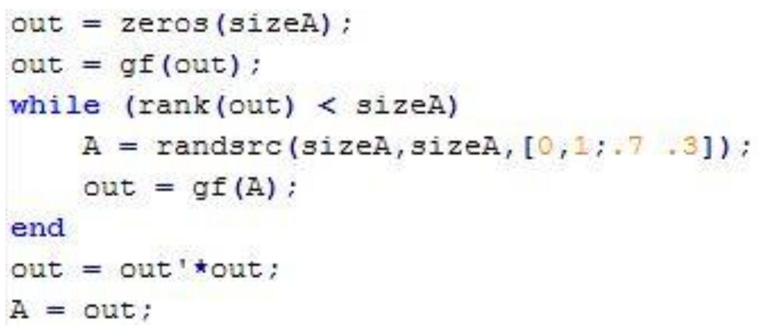
It is greater than the previous algorithm by $2\left(\frac{n}{M}+1\right)\prod(n,n,1)$.

3 SOFTWARE IMPLEMENTATIONS FOR METHODS

3.1 Software implementation for the Montgomery algorithm

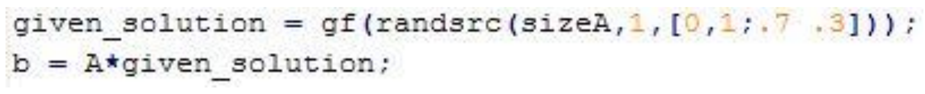
Code implementation for the Montgomery method was produced in Matlab framework.

The system matrix *A* and the right-hand vector *b* are input values for this problem. It is important to create them explicitly in GF(2) for further operations. The problem statement says that there should be a random, symmetric and positive definite matrix:



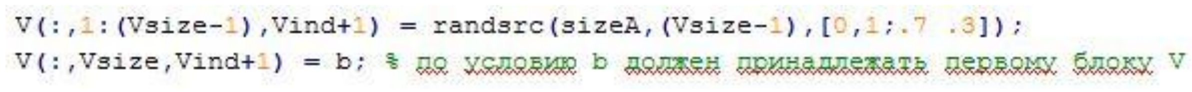
```
out = zeros(sizeA);
out = gf(out);
while (rank(out) < sizeA)
    A = randsrc(sizeA,sizeA,[0,1;.7 .3]);
    out = gf(A);
end
out = out'*out;
A = out;
```

To check the result we should firstly set the solution vector *x* (given_solution in program) by random generation and then produce the right-hand vector *b* as a product $A \cdot x$:



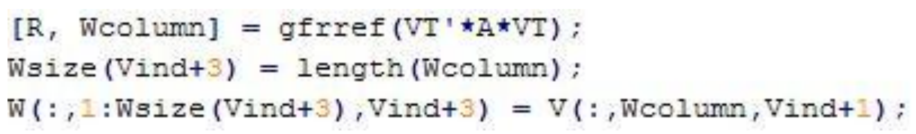
```
given_solution = gf(randsrc(sizeA,1,[0,1;.7 .3]));
b = A*given_solution;
```

Randomizer creates block vector V_0 under condition to include vector *b*



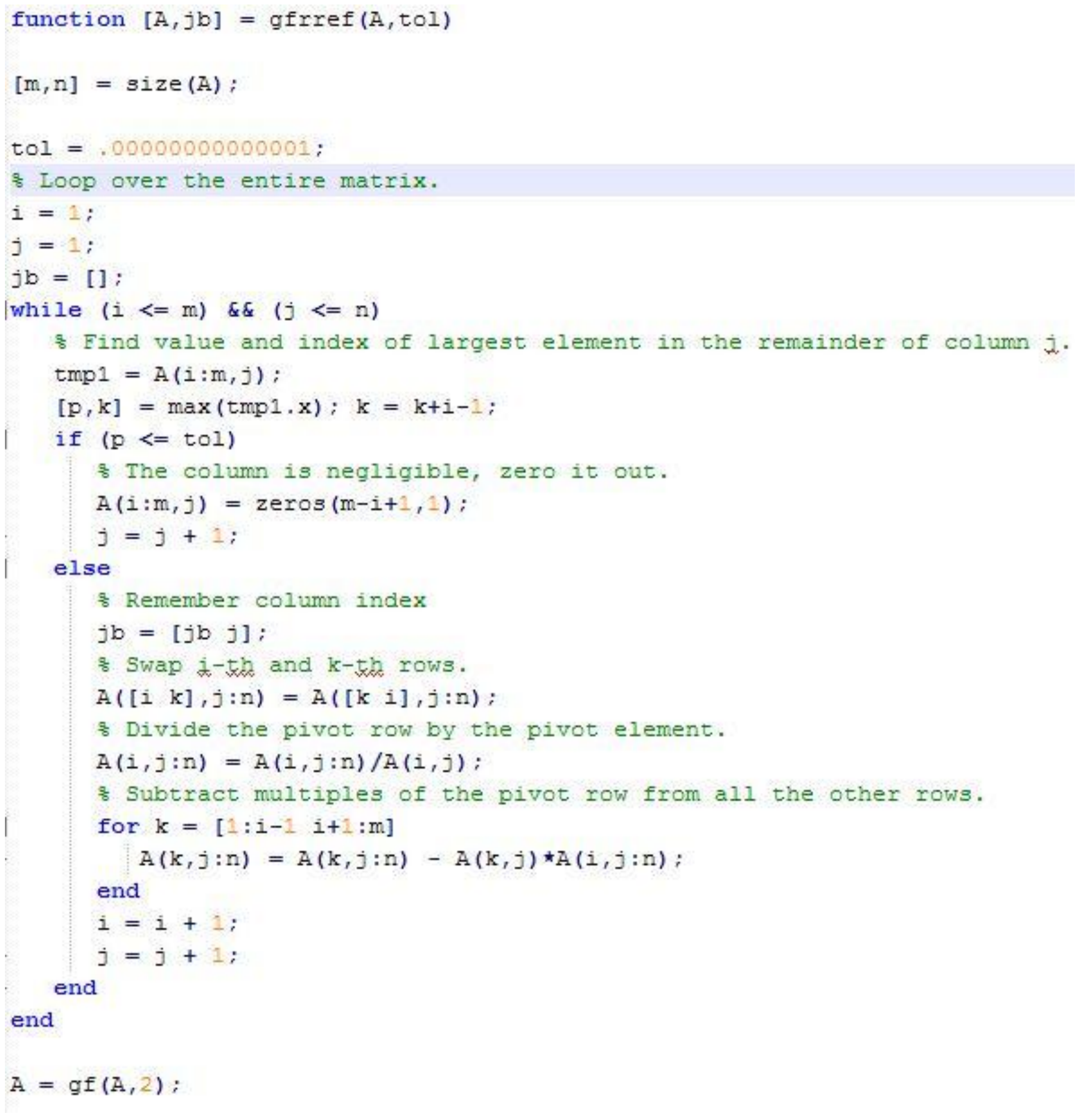
```
V(:,1:(Vsize-1),Vind+1) = randsrc(sizeA,(Vsize-1),[0,1;.7 .3]);
V(:,Vsize,Vind+1) = b; % по условию b должен принадлежать первому блоку V
```

There is a special function for Gauss elimination in the case of GF(2) called gfrref. It is similar to the function rref for real numbers



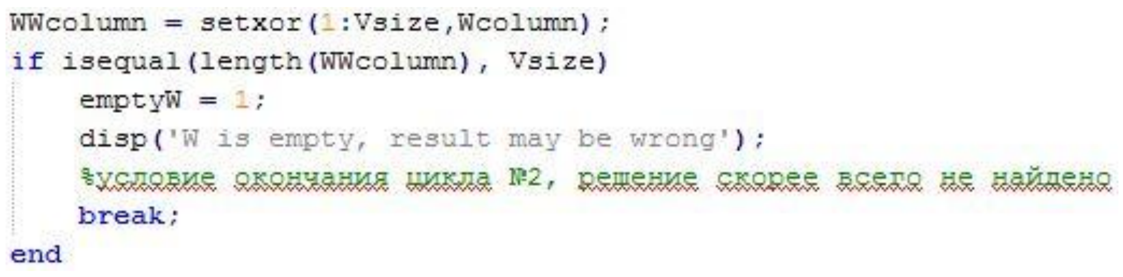
```
[R, Wcolumn] = gfrref(VT'*A*VT);
Wsize(Vind+3) = length(Wcolumn);
W(:,1:Wsize(Vind+3),Vind+3) = V(:,Wcolumn,Vind+1);
```

This function returns a transformed matrix and a number of linearly independent column, which is the required result



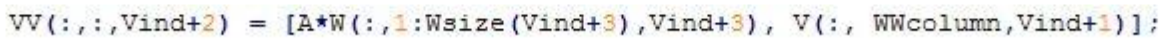
```
function [A,jb] = gfrref(A,tol)

[m,n] = size(A);

tol = .00000000000001;
% Loop over the entire matrix.
i = 1;
j = 1;
jb = [];
while (i <= m) && (j <= n)
   % Find value and index of largest element in the remainder of column j.
   tmp1 = A(i:m,j);
   [p,k] = max(tmp1.x); k = k+i-1;
   if (p <= tol)
      % The column is negligible, zero it out.
      A(i:m,j) = zeros(m-i+1,1);
      j = j + 1;
   else
      % Remember column index
      jb = [jb j];
      % Swap i-th and k-th rows.
      A([i k],j:n) = A([k i],j:n);
      % Divide the pivot row by the pivot element.
      A(i,j:n) = A(i,j:n)/A(i,j);
      % Subtract multiples of the pivot row from all the other rows.
      for k = [1:i-1 i+1:m]
         A(k,j:n) = A(k,j:n) - A(k,j)*A(i,j:n);
      end
      i = i + 1;
      j = j + 1;
   end
end

A = gf(A,2);
```

Remaining columns should be put in the block vector W . Check by their numbers if there are all columns V , W will be void and the method will break down. This happens mostly only with a set of matrices of certain special types that have been studied in detail in [14].



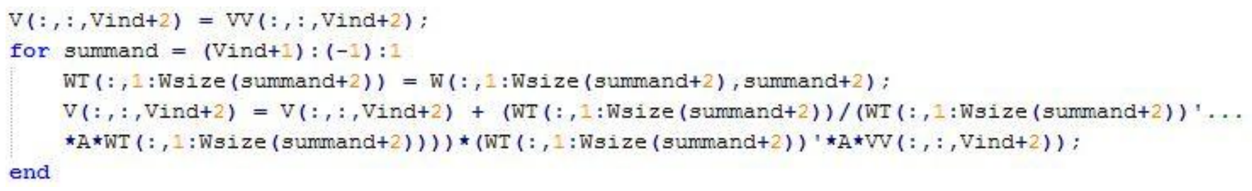
```
WWcolumn = setxor(1:Vsize,Wcolumn);
if isequal(length(WWcolumn), Vsize)
    emptyW = 1;
    disp('W is empty, result may be wrong');
    %условие окончания цикла №2, решение скорее всего не найдено
    break;
end
```

If the method proceeds on without a break, a block vector V could be found as a concatenation of AW and W



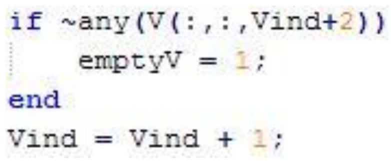
```
VV(:,:,Vind+2) = [A*W(:,1:Wsize(Vind+3),Vind+3), V(:, WWcolumn,Vind+1)];
```

Now all data for the next iteration block vector V_{i+1} are known



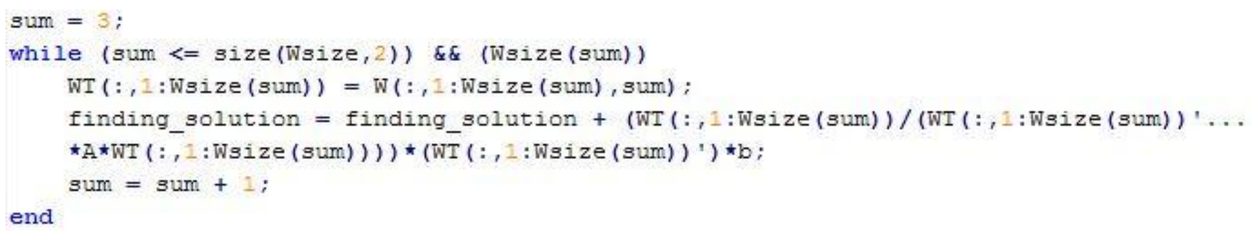
```
V(:,:,Vind+2) = VV(:,:,Vind+2);
for summand = (Vind+1):(-1):1
    WT(:,1:Wsize(summand+2)) = W(:,1:Wsize(summand+2),summand+2);
    V(:,:,Vind+2) = V(:,:,Vind+2) + (WT(:,1:Wsize(summand+2))/(WT(:,1:Wsize(summand+2))'...
    *A*WT(:,1:Wsize(summand+2))))*(WT(:,1:Wsize(summand+2))'*A*VV(:,:,Vind+2));
end
```

If the function returns a zero matrix, the method has converged. Otherwise make the next iteration



```
if ~any(V(:,:,Vind+2))
    emptyV = 1;
end
Vind = Vind + 1;
```

In case of success, there is a solution



```
sum = 3;
while (sum <= size(Wsize,2)) && (Wsize(sum))
    WT(:,1:Wsize(sum)) = W(:,1:Wsize(sum),sum);
    finding_solution = finding_solution + (WT(:,1:Wsize(sum))/(WT(:,1:Wsize(sum))'...
    *A*WT(:,1:Wsize(sum))))*(WT(:,1:Wsize(sum))')*b;
    sum = sum + 1;
end
```

Prove the method by comparing it first with a generated solution. An unsuccessful case could be resolved by generating another initial block vector V_0.

Running time of the program for test cases are in table 1. *P* means set probability for non-zero elements.

Table 1. Execution time for implementation of the Montgomery method.

№	*n*	Columns in *V*	*P*	Time, sec
1	48	4	0.1	2.6746
2	64	8	0.1	2.4129
3	96	12	0.1	4.7809
4	128	16	0.1	8.1159
5	192	24	0.1	17.4575
6	256	32	0.1	30.5969

The curve (figure 1) shows the growth of the function to be $O\left(n^2\right)$. This is attained without optimization, because this implementation uses only Matlab’s embedded optimization.

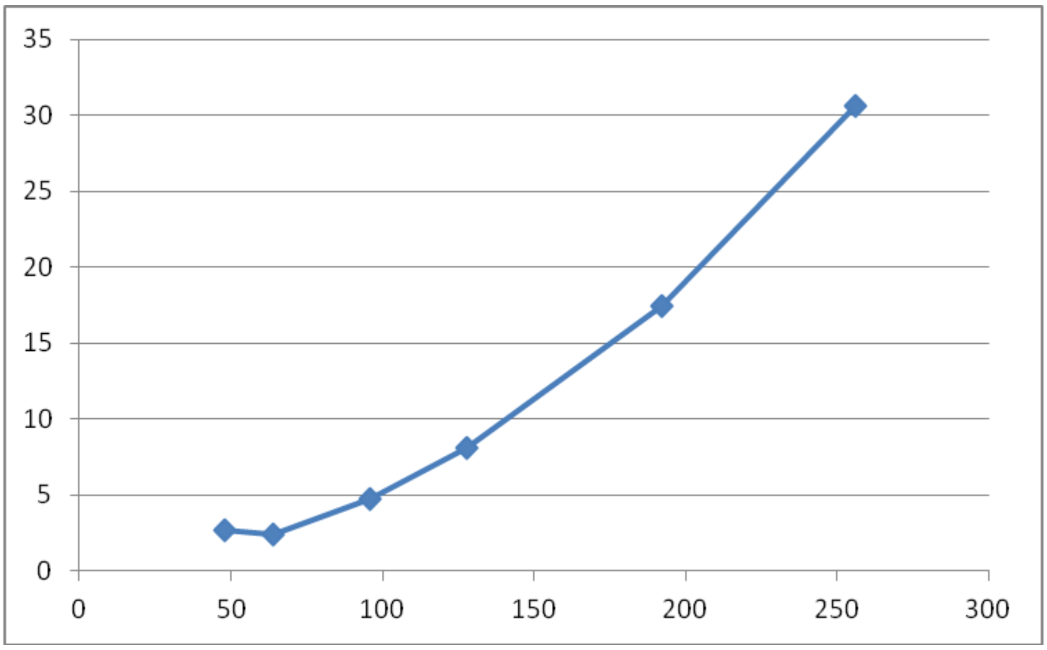


Figure 1. Execution time dependence of system matrix size, sec.

3.2 Software implementation for the Wiedemann-Coppersmith algorithm

Code implementation for the Wiedemann-Coppersmith method uses C++.

There are functions for matrix operations in the library abo (accelerated binary operations). The library has detailed description in chapter 4. In this chapter, we present the implementation of the method omitting library functions.

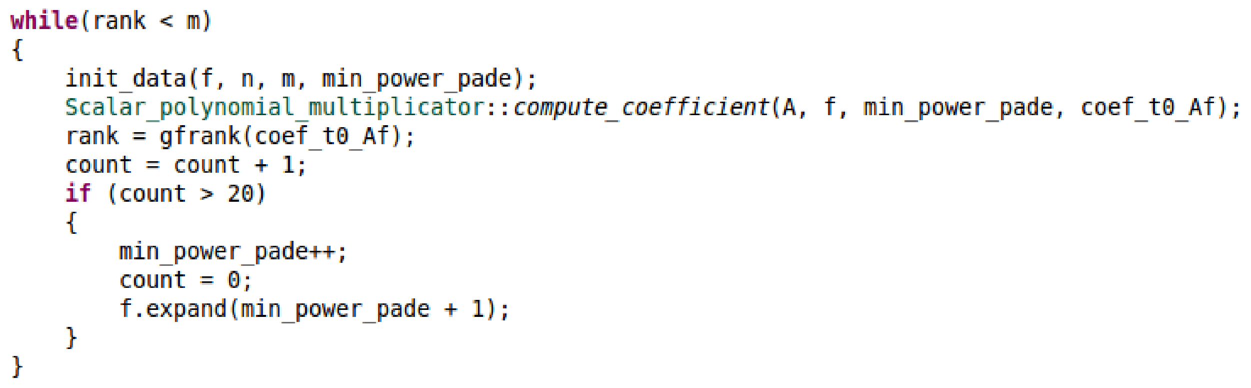
The main function carries out a few operations. It generates input data, calls functions to generate a sequence of Krylov spaces to create the characteristic polynomial and to find and check the solution

```
//Krylov_sequence_size
unsigned short L = ceil((N/n) + (N/m) + (2*n/m) + 1);

Polynomial<bool> A(L + 1, m, n, Storing::by_rows);
Polynomial<bool> f;

GenSeqKr<bool, scalar::Block_multiplicator>(B, x, y, L, A);

vector<unsigned short> vDelta;

min_poly<bool>(n, m, A, L, vDelta, f);

Matrix<bool> solution(z.row_count(), f.coefficient_column_count(), Storing::by_columns);
Matrix<bool> w(z.row_count(), f.coefficient_column_count(), Storing::by_columns);
find_res<bool, scalar::Block_multiplicator>(solution, L, w, z, f, B);

if (check_res<bool, scalar::Block_multiplicator>(solution, B))
    cout << "Check finished, solution is true" << endl;
else
    cout << "Check finished, solution is false" << endl;
```

The sequence of Krylov spaces requires the matrix product $x^T \cdot B^i \cdot y$

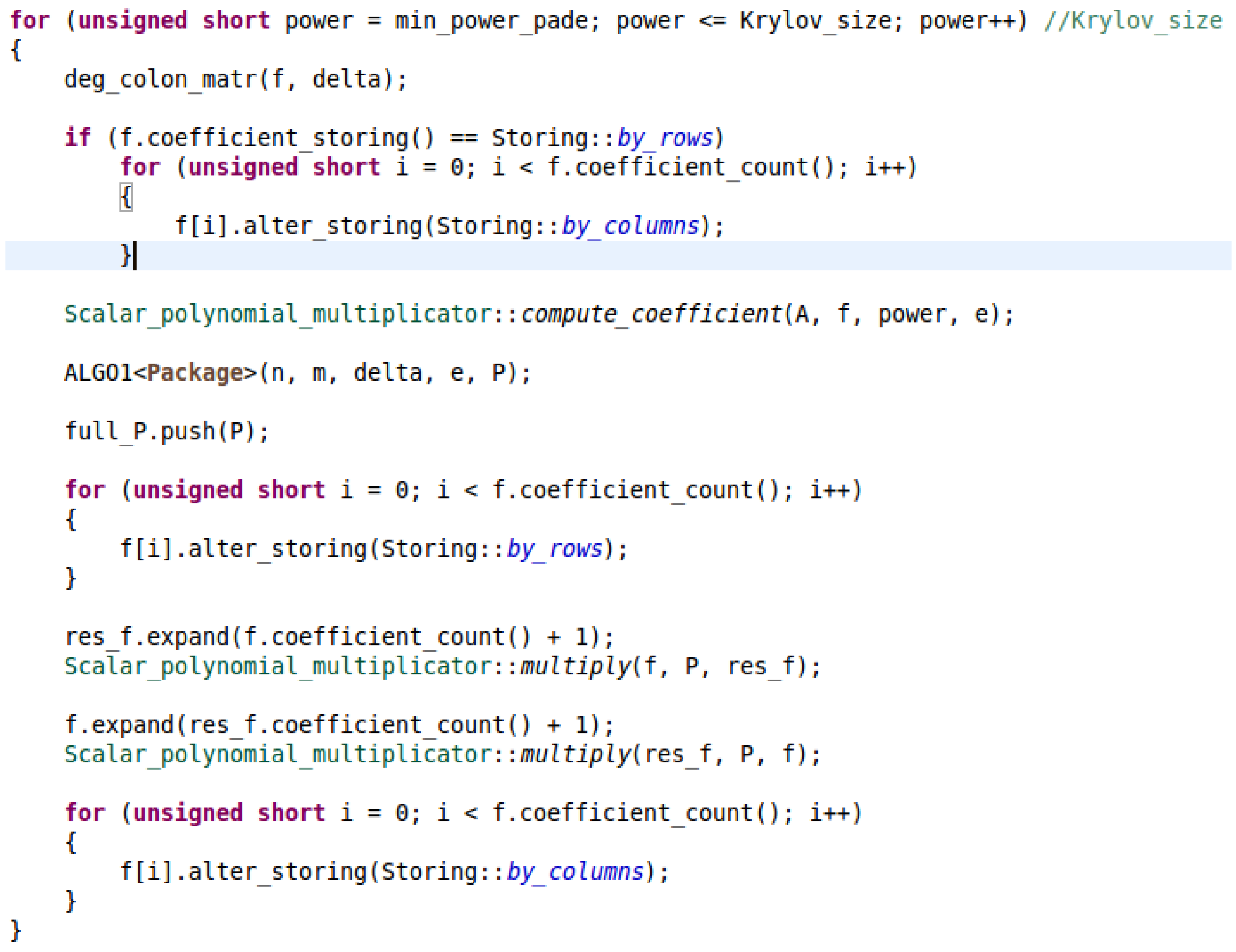
```
template <typename Package, typename Operation>
void GenSeqKr(Matrix<Package>& B, Matrix<Package>& x, Matrix<Package>& y,
 unsigned short Krylov_size, Polynomial<Package>& A)
{
    Polynomial<Package> Temp(Krylov_size + 1, y.row_count(), y.column_count(),
    Storing::by_columns);
    Matrix<Package> transp_x(x.column_count(), x.row_count(), Storing::by_rows);
    Temp[0] = y;
    x.transpose(transp_x);
    Operation::multiply(transp_x, y, A[0]);

    for (unsigned short i = 1; i <= Krylov_size; i++)
    {
        Operation::multiply(B, Temp[i - 1], Temp[i]);
        Operation::multiply(transp_x, Temp[i], A[i]);
    }
}
```

The next step is computationally hard. The characteristic polynomial is formed through several sub-steps. Each of them has its particular function. Firstly we form the initial polynomial



```
while(rank < m)
{
    init_data(f, n, m, min_power_pade);
    Scalar_polynomial_multiplicator::compute_coefficient(A, f, min_power_pade, coef_t0_Af);
    rank = gfrank(coef_t0_Af);
    count = count + 1;
    if (count > 20)
    {
        min_power_pade++;
        count = 0;
        f.expand(min_power_pade + 1);
    }
}
```

Further on, we cycle through a number of iterations which equals the length of the sequence of Krylov spaces



```
for (unsigned short power = min_power_pade; power <= Krylov_size; power++) //Krylov_size
{
    deg_colon_matr(f, delta);

    if (f.coefficient_storing() == Storing::by_rows)
        for (unsigned short i = 0; i < f.coefficient_count(); i++)
        {
            f[i].alter_storing(Storing::by_columns);
        }

    Scalar_polynomial_multiplicator::compute_coefficient(A, f, power, e);

    ALGO1<Package>(n, m, delta, e, P);

    full_P.push(P);

    for (unsigned short i = 0; i < f.coefficient_count(); i++)
    {
        f[i].alter_storing(Storing::by_rows);
    }

    res_f.expand(f.coefficient_count() + 1);
    Scalar_polynomial_multiplicator::multiply(f, P, res_f);

    f.expand(res_f.coefficient_count() + 1);
    Scalar_polynomial_multiplicator::multiply(res_f, P, f);

    for (unsigned short i = 0; i < f.coefficient_count(); i++)
    {
        f[i].alter_storing(Storing::by_columns);
    }
}
```

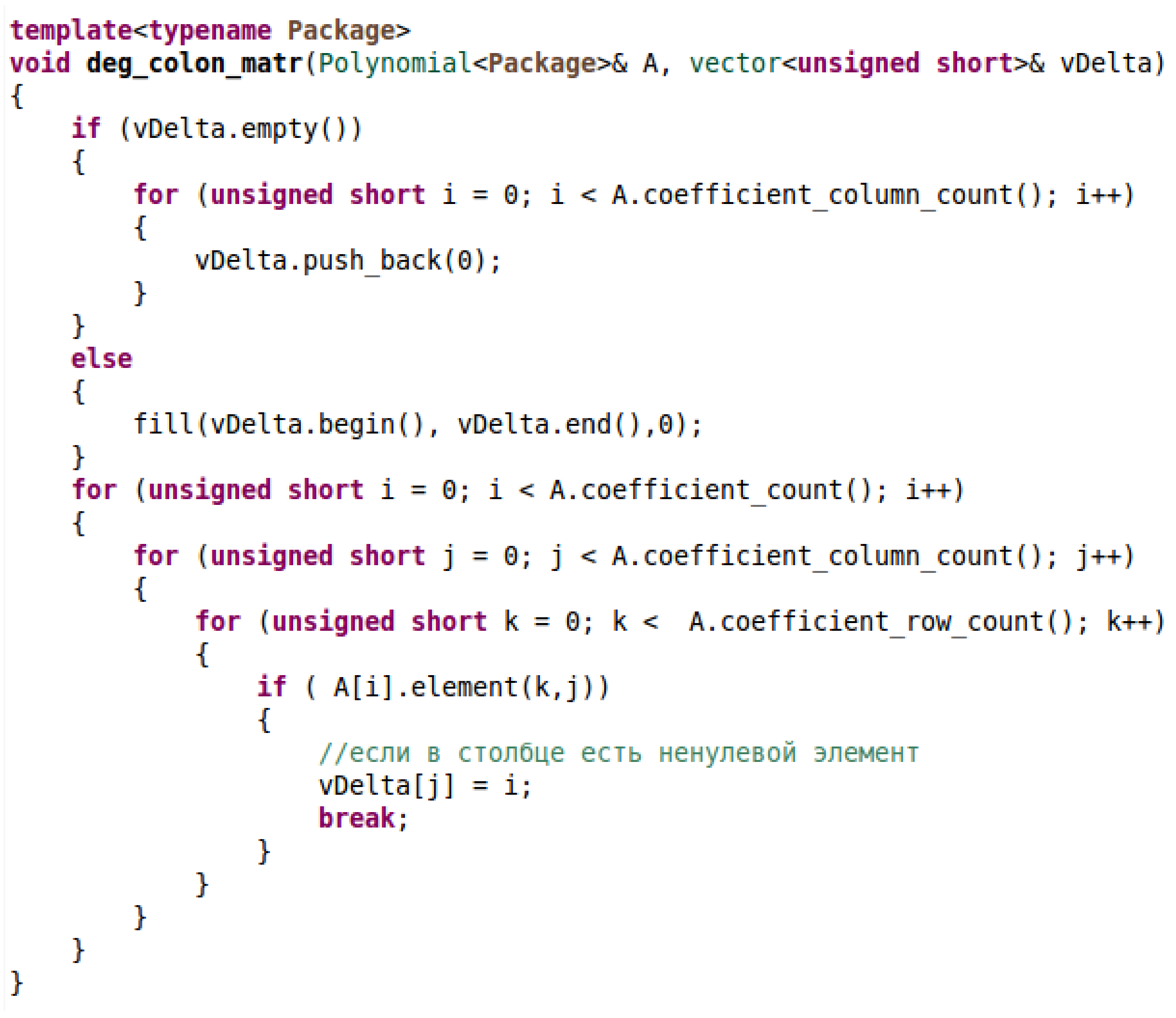
An initial characteristic polynomial is created by matrix concatenation. It includes a matrix of random polynomials as a left operand and identity matrix as a right one. Powers of polynomials are under condition $t < t_0$

```
template<typename Package>
void init_data(Polynomial<Package>& polinom, size_t n, size_t m, unsigned short t0)
{
    Matrix<Package> rand_matr(n, m, Storing::by_columns);
    double const probability_of_zero = 0.5;

    for (unsigned short i = 0; i <= t0; i++)
    {
        if (i < t0)
        {
            rand_matr.generate(probability_of_zero);
            for (unsigned short j = 0; j < m; j++)
            {
                for (unsigned short k = 0; k < n; k++)
                {
                    polinom[i].element(k, j) = rand_matr.element(k, j);
                }
            }
        }
        else
        {
            rand_matr.generate(probability_of_zero);
            for (unsigned short j = 0; j < m; j++)
            {
                for (unsigned short k = 0; k < n; k++)
                {
                    polinom[i].element(k, j) = rand_matr.element(k, j);
                    polinom[i].element(k, k + m) = 1;
                }
            }
        }
    }
}
```

In case of GF(2) a particular function returns the rank of a binary matrix:

```
template<typename Package>
unsigned short gfrank(const Matrix<Package>& Matr)
{
    vector<bool> busy;
    size_t j0;
    static unsigned short count;
    count = 0;
    static Matrix<Package> temp;
    temp = Matr;
    for (j0 = 0; j0 < temp.column_count(); j0++)
    {
        busy.push_back(0);
    }
    for (size_t i = 0; i < temp.row_count(); i++)
    {
        for (j0 = 0; j0 < temp.column_count(); j0++)
        {
            if ((temp.element(i, j0)) && (!(busy[j0])))
            {
                count++;
                busy[j0] = 1;
                break;
            }
        }
        for (size_t j = j0 + 1; j < temp.column_count(); j++)
        {
            auto lambda = static_cast<bool>(temp.element(i, j));
            for (size_t k = 0; k < temp.row_count(); k++)
            {
                //если лямбда 1, столбец остается таким же, если 0 - зануляется
                temp.element(k, j) = static_cast<bool>(temp.element(k, j))
                        xor lambda*(static_cast<bool>(temp.element(k, j0)));
            }
        }
    }
    return count;
}
```

While approximation of the characteristic polynomial is going on, the power of columns in matrix-coefficients should be checked on each step. Call the function:



```
template<typename Package>
void deg_colon_matr(Polynomial<Package>& A, vector<unsigned short>& vDelta)
{
    if (vDelta.empty())
    {
        for (unsigned short i = 0; i < A.coefficient_column_count(); i++)
        {
            vDelta.push_back(0);
        }
    }
    else
    {
        fill(vDelta.begin(), vDelta.end(),0);
    }
    for (unsigned short i = 0; i < A.coefficient_count(); i++)
    {
        for (unsigned short j = 0; j < A.coefficient_column_count(); j++)
        {
            for (unsigned short k = 0; k <  A.coefficient_row_count(); k++)
            {
                if ( A[i].element(k,j))
                {
                    //если в столбце есть ненулевой элемент
                    vDelta[j] = i;
                    break;
                }
            }
        }
    }
}
```

ALGO1 function creates an iteration matrix according to Thome’s work [12]:

```
template<typename Package>
void ALGO1(size_t n, size_t m, vector<unsigned short>& delta,
        Matrix<Package>& mE, Polynomial<Package>& P)
{
    vector<pair<unsigned short, unsigned short>> sorted_Delta;
    unsigned short size;
    static Matrix<Package> mE_sorted(m, m + n, Storing::by_rows);
    mE_sorted = mE;
    static Matrix<Package> P_sorted(m + n, m + n, Storing::by_columns);
    //создание единичной матрицы
    P[0].unity();

    //сортировка степеней
    sorting_delta(delta, sorted_Delta);
    sorting_matrix<Package>(sorted_Delta, mE_sorted, mE);
    sorting_matrix<Package>(sorted_Delta, P[0], P_sorted);

    P[1].zero();

    //метод Гаусса
    Gauss_elimination<Package>(mE, P_sorted, P[0], P[1]);

}
```

Sorting_delta is a sorting function that uses a standard procedure stable_sort. It saves the initial order of so called nominal degree. According to Coppersmith, nominal degree of column is an upper bound on the degree of the coefficients in this column of the polynomial [12]:

```
void sorting_delta(vector<unsigned short>& vect,
        vector<pair<unsigned short, unsigned short>>& sorted_vector)
{
    pair<unsigned short, unsigned short> pp;
    //сортировка степеней
    for (unsigned short i = 0; i < vect.size(); i++)
    {
        pp = make_pair(vect[i],i);
        sorted_vector.push_back(pp);
    }
    stable_sort(sorted_vector.begin(),sorted_vector.end(),pairCompare);
}
```

The sorted vector holds the order to change matrix columns by its powers:

```
template<typename Package>
void sorting_matrix(vector<pair<unsigned short, unsigned short>>& sorted_vector,
        Matrix<Package>& Matr, Matrix<Package>& sorted_matr)
{
    size_t column_number;
    //сортировка матрицы по степеням
    for (unsigned short j = 0; j < sorted_vector.size(); j++)
    {
        column_number = (size_t)sorted_vector[j].second;
        for (size_t i = 0; i < Matr.row_count(); i++)
        {
            sorted_matr.element(i, j) = Matr.element(i, column_number);
        }
    }
}
```

When matrices are transformed, the program calls particular Gauss elimination for GF(2). This function is similar to the listed gfrank function:

```
template<typename Package>
void Gauss_elimination(Matrix<Package>& E_sorted, Matrix<Package>& P_sorted,
        Matrix<Package>& P_result, Matrix<Package>& XP_result)
{
    vector<bool> busy;
    size_t j0;
    for (j0 = 0; j0 < E_sorted.column_count(); j0++)
        busy.push_back(0);
    for (size_t i = 0; i < E_sorted.row_count(); i++)
    {
        for (j0 = 0; j0 < E_sorted.column_count(); j0++)
        {
            if ((E_sorted.element(i, j0)) && (!(busy[j0])))
            {
                busy[j0] = 1;
                break;
            }
        }
        for (size_t j = j0 + 1; j < E_sorted.column_count(); j++)
        {
            auto lambda = static_cast<bool>(E_sorted.element(i, j));
            for (size_t k = 0; k < E_sorted.row_count(); k++)
            {
                E_sorted.element(k, j) = static_cast<bool>(E_sorted.element(k, j))
                        xor lambda*(static_cast<bool>(E_sorted.element(k, j0)));
            }
            for (size_t k = 0; k < P_sorted.row_count(); k++)
                P_sorted.element(k, j) = static_cast<bool>(P_sorted.element(k, j))
                xor lambda*(static_cast<bool>(P_sorted.element(k, j0)));
        }
    }
    for (size_t j =0; j < P_sorted.column_count(); j++)
    {
        if (busy[j])
        {
            for (size_t i = 0; i < P_sorted.row_count(); i++)
            {
                XP_result.element(i, j) = P_sorted.element(i, j);
                P_result.element(i, j) = 0;
            }
        }
        else
        {
            for (size_t i = 0; i < P_sorted.row_count(); i++)
                P_result.element(i, j) = P_sorted.element(i, j);
        }
    }
}
```

The solution contains the vector w which can be found as $w = z \cdot f_{(L)} + B \cdot z \cdot f_{(L-1)} + \dots + B^L \cdot z \cdot f_{(0)}$

```
template<typename Package, typename Operation>
void find_res(Matrix<Package>& solution, unsigned short Krylov_size, Matrix<Package>& w,
        Matrix<Package>& z, Polynomial<Package>& f, Matrix<Package>& B)
{
    Matrix<Package> temp_w(z.row_count(), 1, Storing::by_columns);
    Matrix<Package> temp_w2(z.row_count(), 1, Storing::by_columns);
    Matrix<Package> temp_kappa(B.row_count(), z.column_count(), Storing::by_columns);
    Matrix<Package> kappa(B.row_count(), z.column_count(), Storing::by_rows);
    vector<unsigned short> count_deg;
    unsigned short jj, ii;
    unsigned short L = 0.85*Krylov_size;

    solution.zero();
    Operation::multiply(B, z, kappa);
    z.alter_storing(Storing::by_rows);
    Operation::multiply(z, f[L], w);
    for (unsigned short k = 0; k <= L; k++)
    {
        kappa.alter_storing(Storing::by_rows);
        for (unsigned short i = 0; i < f.coefficient_row_count(); i++)
        {
            for (jj = 0; jj < f.coefficient_column_count(); jj++)
                if (f[L - k].element(i, jj))
                {
                    Operation::multiply_add(kappa, f[L - k], w);
                    break;
                }
            //если выход из цикла был по брейку
            if (jj < f.coefficient_column_count())
                break;
        }
        kappa.alter_storing(Storing::by_columns);
        temp_kappa = kappa;
        Operation::multiply(B, temp_kappa, kappa);
    }
```

The series of products $B^i w_j$ finally returns a solution if there is a zero value. It means that the result sought for is the previous product $B^{i-1} w_j$:

```
    for (unsigned short j = 0; j < f.coefficient_column_count(); j++)
    {
        count_deg.push_back(0);
        temp_w.column(0) = w.column(j);
        for (unsigned short i = 0; i < temp_w.row_count(); i++)
            if (temp_w.element(i, 0)) //если столбец ненулевой
            {
                while (count_deg[j] < L*0.75)
                {
                    Operation::multiply(B, temp_w, temp_w2);
                    //проверяем, если матрица нулевая - решение найдено
                    for (ii = 0; ii < temp_w2.row_count(); ii++)
                        if (temp_w2.element(ii, 0))
                        {
                            temp_w = temp_w2;
                            count_deg[j]++;
                            break;
                        }
                    //если цикл дошел до конца, а не прервался по брейку - матрица нулевая
                    if (ii == temp_w2.row_count())
                    {
                        solution.column(j) = temp_w.column(0);
                        break;
                    }
                }
                break;
            }
    }
}
```

Function check_res takes given solution and tests if it suits the system:

```
template<typename Package, typename Operation>
bool check_res(Matrix<Package>& w, Matrix<Package>& B)
{
    Matrix<Package> check(B.row_count(), w.column_count(), Storing::by_rows);
    Matrix<Package> temp_zero(B.row_count(), w.column_count(), Storing::by_rows);
    temp_zero.zero();

    for (unsigned short i = 0; i < w.row_count(); i++)
        for (unsigned short j = 0; j < w.column_count(); j++)
            if (w.element(i,j))
            {
                cout << "Solution is nonzero!" << endl;
                Operation::multiply(B, w, check);
                if (check == temp_zero)
                    return true;
                else
                    return false;
            }
    return false;
}
```

With the aim of measuring the computational cost, a simulation exercise was conducted. Presentation of the results is in table 2. *N* denotes the column of block vectors *x* and *z*, *P* denotes the probability for the case when the matrix element is zero. Figure 2 presents graphically the data from Table 2.

Table 2. Execution time for implementation of Wiedemann-Coppersmith method.

№	*n*	*N*	*P*	Total time, sec	I step, sec	II step, sec	III step, sec
1	96	12	0,995	0,209725	0,018433	0,158559	0,031938
2	128	16	0,996	0,454333	0,042353	0,350721	0,073813
3	192	24	0,9975	1,542	0,138905	1,11645	0,280819
4	256	32	0,998	2,909263	0,294589	2,071512	0,530434
5	512	64	0,999	21,929575	2,332576	15,460627	4,038455
6	1024	128	0,9995	182,53589	18,54862	130,29009	32,92925

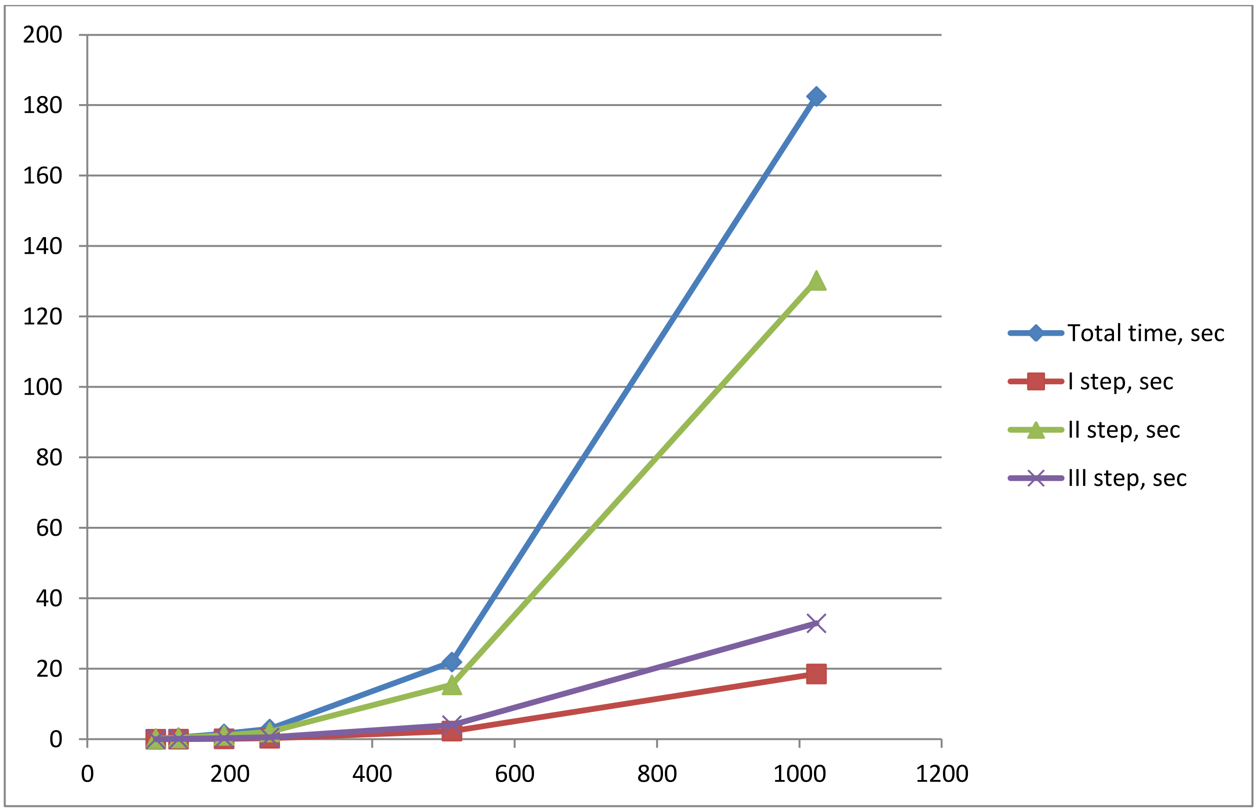


Figure 2. Execution time dependence of system matrix size.

4 BINARY MATRIX OPERATIONS LIBRARY: THEORY AND IMPLEMENTATION

4.1 Matrix multiplication method

Matrix multiplication methods fall into two classes: element-wise processing and block operations. Element-by-element processes manipulate matrix elements while block operations use whole columns or rows depending on the format of data storage. The second group of methods seems to be computationally efficient because block operations reduce memory access. Element-by-element processes refer to the “naive” method, method known as the Strassen algorithm, and the method of Four Russians [15]. There is one interesting method that suits well for row-stored matrices described by Gustavson in [16, 17] for the case of sparse matrices. This method shows good result in practice [18].

Current work considers two ways: a “naive” method involving the method of Four Russians and a modification of the Gustavson method in case of dense matrices.

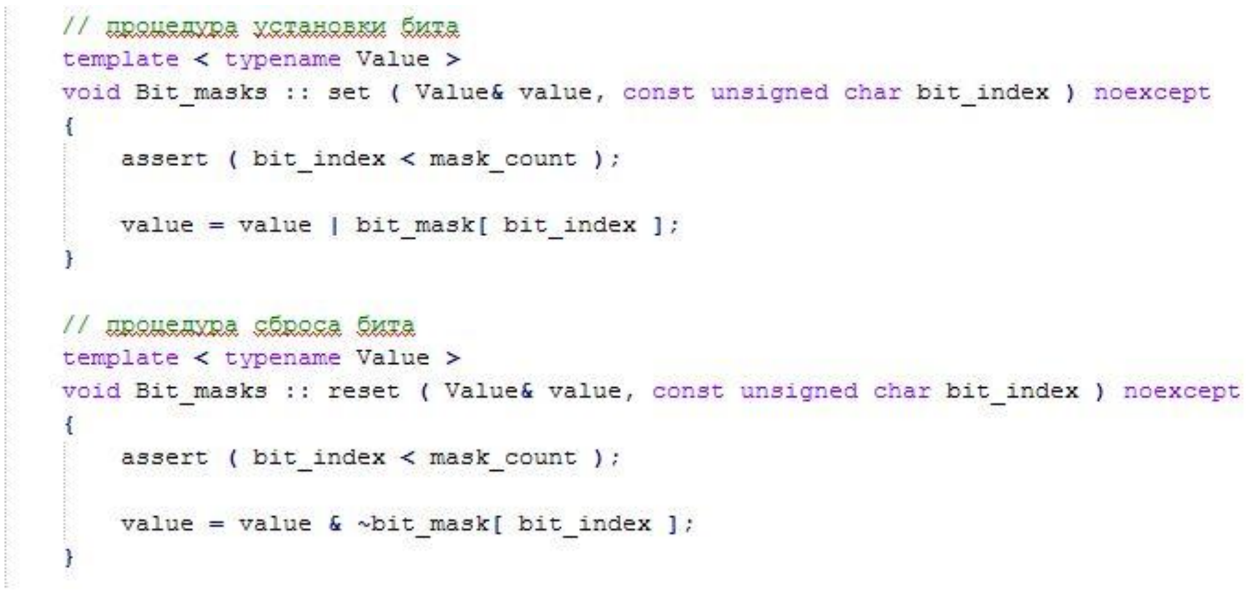
4.1.1 “Naive” method

This method has an intuitive implementation. Scalar multiplication applies to row and column vectors according to the definition of matrix multiplication.

For reasons of optimization, left matrices are stored in memory by rows, right matrices by columns. From computer representation point of view there are several available implementations:

- for different data types (1-bit, 2-bit, 4-bit and 8-bit machine words)
- for 64-bit, 128-bit and 256-bit blocks of data (using MMX, SSE and AVX intrinsic-functions by Intel to parallel operations [18])

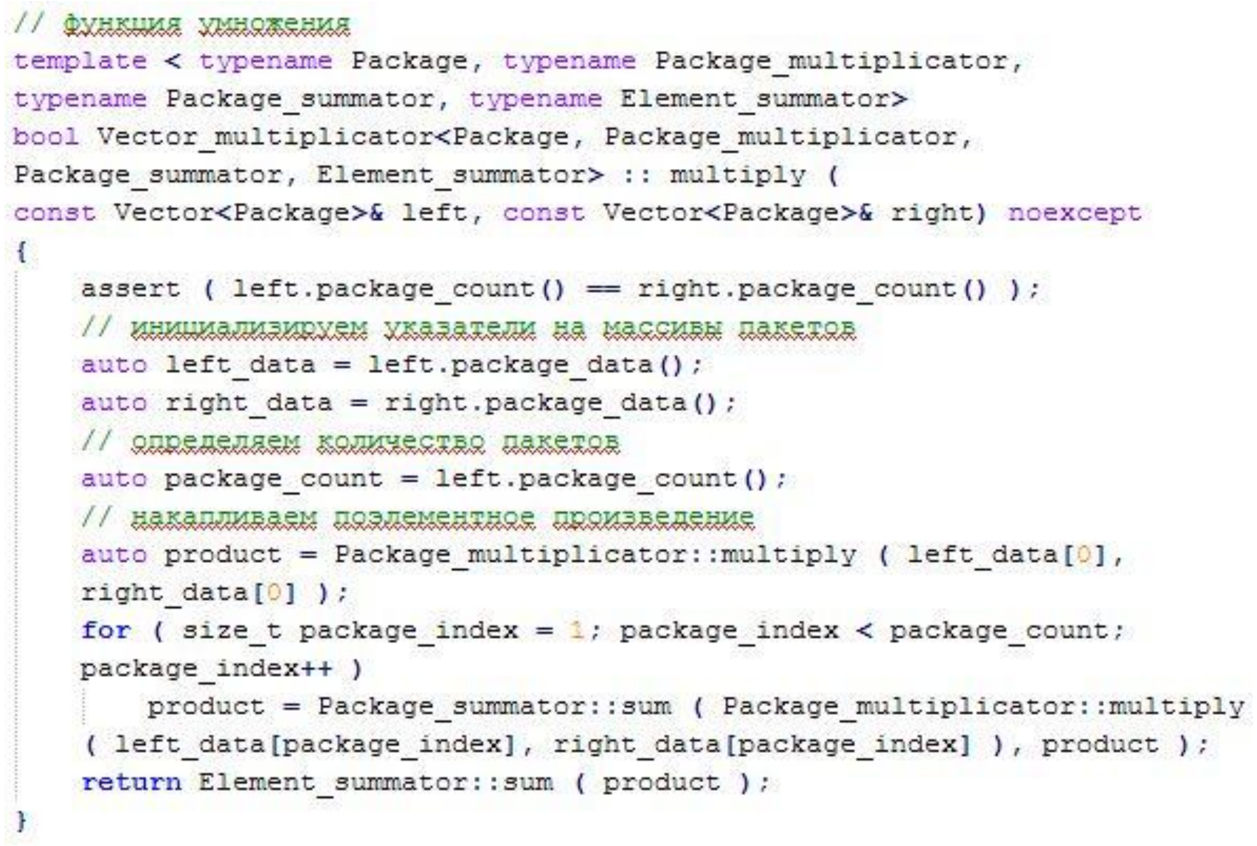
Bitwise operations for addition and multiplication suit for bitwise matrix storage. The only challenge is to fill such matrices properly



```
// процедура установки бита
template < typename Value >
void Bit_masks :: set ( Value& value, const unsigned char bit_index ) noexcept
{
    assert ( bit_index < mask_count );

    value = value | bit_mask[ bit_index ];
}

// процедура сброса бита
template < typename Value >
void Bit_masks :: reset ( Value& value, const unsigned char bit_index ) noexcept
{
    assert ( bit_index < mask_count );

    value = value & ~bit_mask[ bit_index ];
}
```

A scalar method manipulates boolean type and logic operations.

Data storage for intrinsic functions is also critical. Abo library provides a record to vectors which form 64-bit, 128-bit or 256-bit blocks in their turn. There are special procedures for arithmetic operations developed for intrinsic-functions. Its technical specification and details are described in [19]. In this special case, matrix multiplication is implemented as



```
// функция умножения
template < typename Package, typename Package_multiplicator,
typename Package_summator, typename Element_summator>
bool Vector_multiplicator<Package, Package_multiplicator,
Package_summator, Element_summator> :: multiply (
const Vector<Package>& left, const Vector<Package>& right) noexcept
{
    assert ( left.package_count() == right.package_count() );
    // инициализируем указатели на массивы пакетов
    auto left_data = left.package_data();
    auto right_data = right.package_data();
    // определяем количество пакетов
    auto package_count = left.package_count();
    // накапливаем поэлементное произведение
    auto product = Package_multiplicator::multiply ( left_data[0],
    right_data[0] );
    for ( size_t package_index = 1; package_index < package_count;
    package_index++ )
        product = Package_summator::sum ( Package_multiplicator::multiply
    ( left_data[package_index], right_data[package_index] ), product );
    return Element_summator::sum ( product );
}
```

4.1.2 Method of Four Russians

In 1970 Arlazarov, Dinic, Kronrod and Faradzev public a paper. That included the algorithm named as the method of Four Russians. The main idea was to use a preset table for vector multiplication. The size of vectors is bounded.

For example, let us calculate and record scalar multiplication by mod 2 for all possible pairs of binary vectors of size *k*. The results are placed in the table on the cross of left and right operands. (figure 3). Take the first matrix and decompose each row into vectors of size *k*. Each of them matches up with one binary vector. The second matrix is decomposed in the same way, by columns instead of rows. This trick cuts computation by a factor *k*.

There are three modifications for the method of Four Russians in the abo library:

- *Multiplication table for 4-size vectors with results put in bytes*

  The table has 16×16 elements of logic type. Before multiplication function splits each operand into 4-element vectors and finds a solution of conjunction from the table.

- *Multiplication table for 8-size vectors with results put in bytes*

  The table has 256×256 elements of logic type. Before multiplication function splits each operand into 8-element vectors and finds a solution of conjunction from the table.

- *Multiplication table for 8-size vectors with results put in rows of bits*

  The table has 32×256 bit elements. Before multiplication function splits each operand into 8-element vectors and finds a solution of bitwise conjunction from the table.

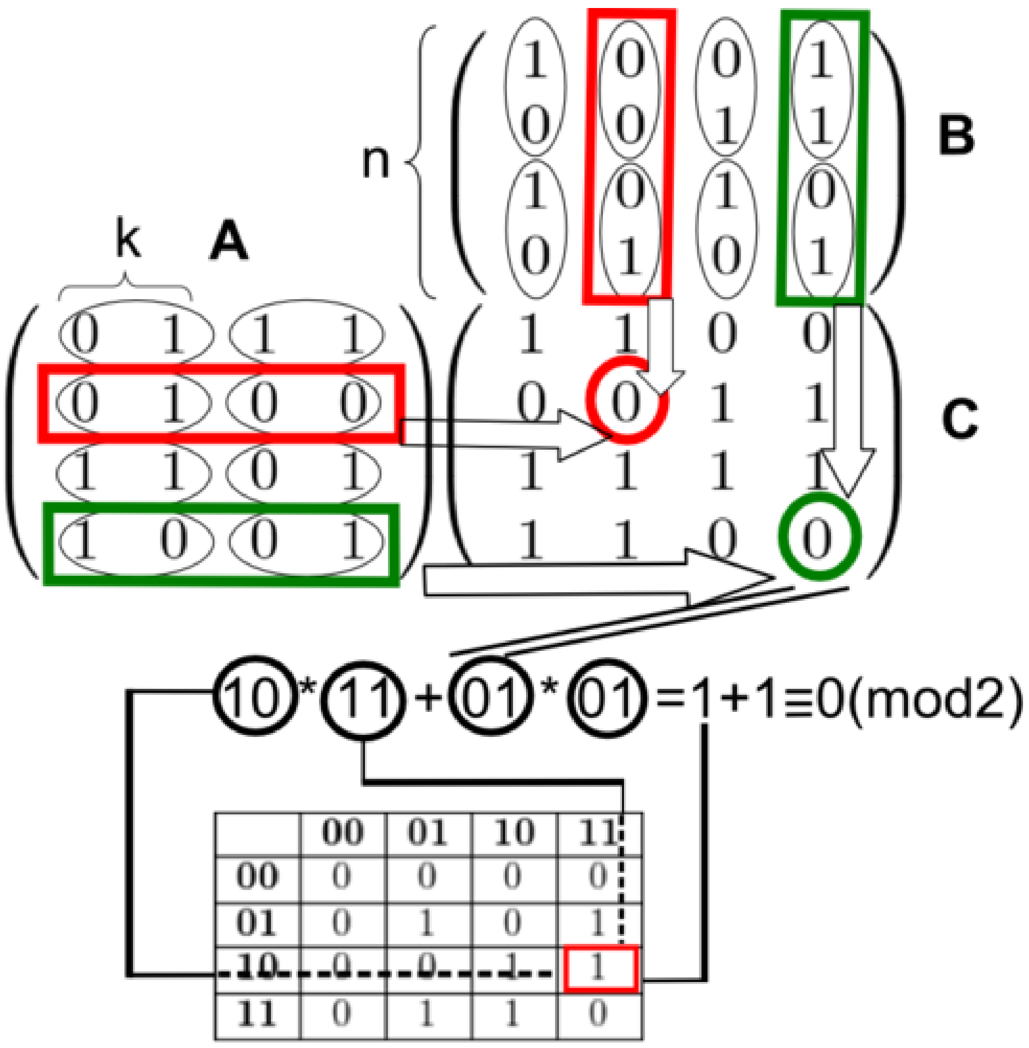
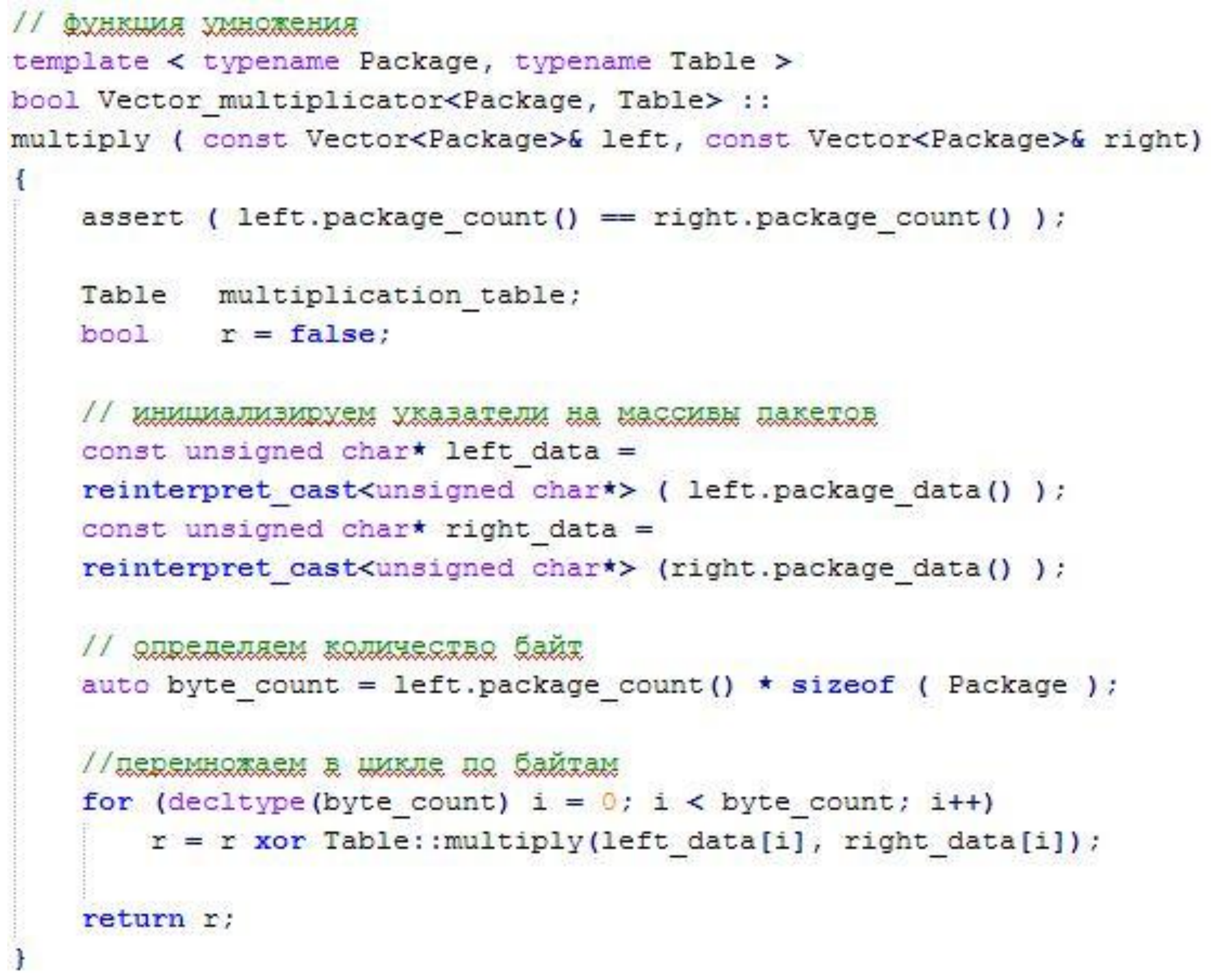


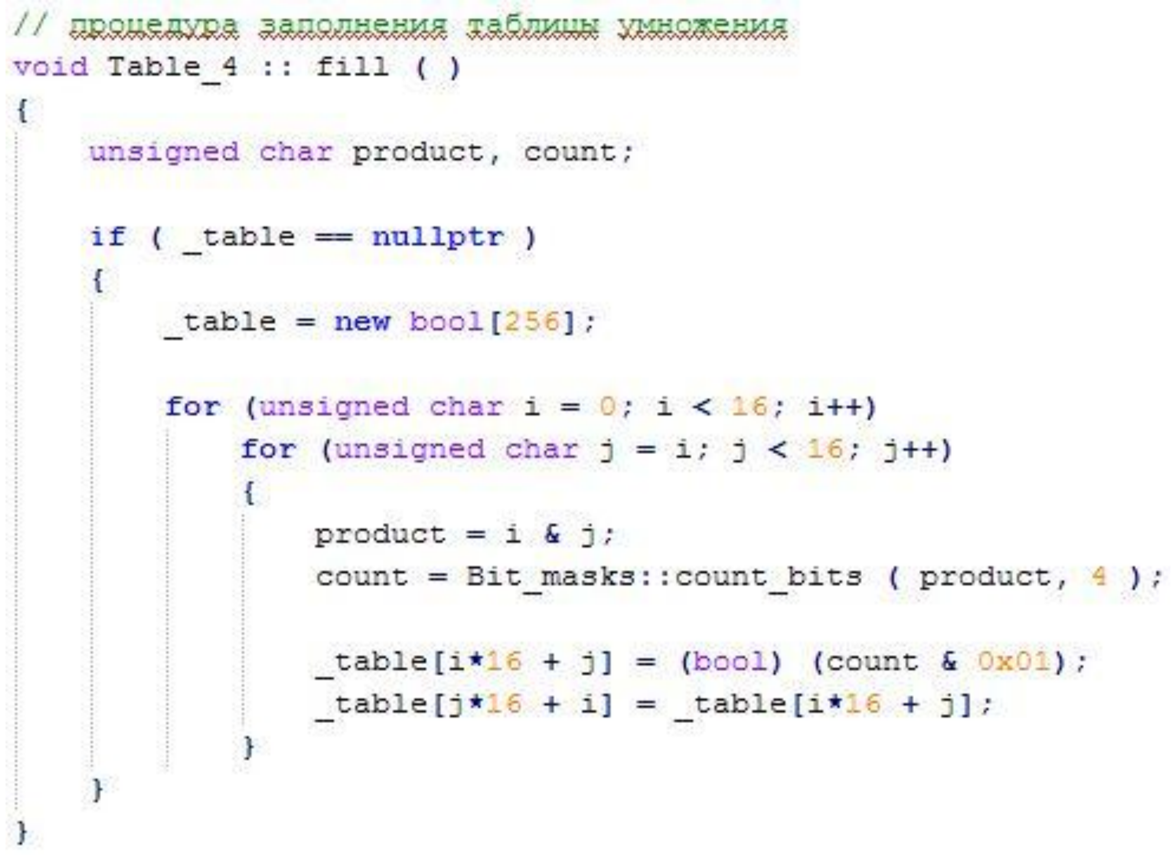
Figure 3. The method of Four Russians [20].

Implementation for method of Four Russians calls preset multiplication table:



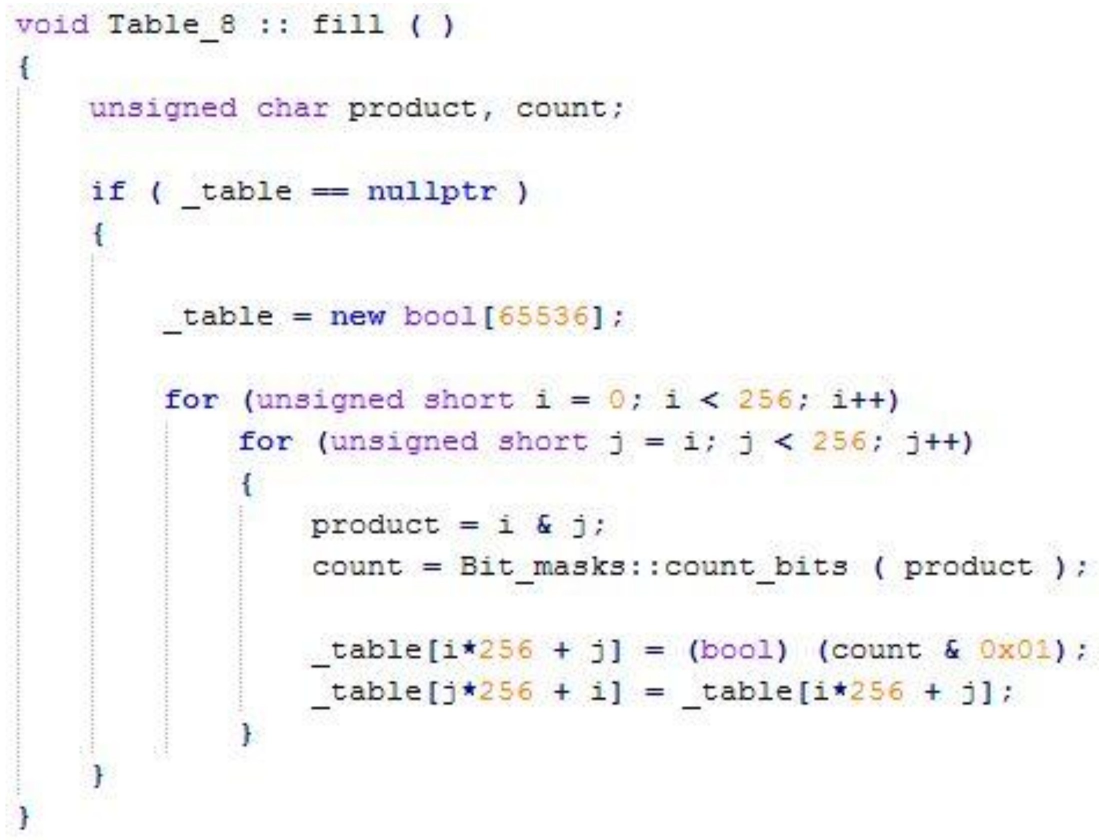
```
// функция умножения
template < typename Package, typename Table >
bool Vector_multiplicator<Package, Table> ::
multiply ( const Vector<Package>& left, const Vector<Package>& right)
{
    assert ( left.package_count() == right.package_count() );

    Table   multiplication_table;
    bool    r = false;

    // инициализируем указатели на массивы пакетов
    const unsigned char* left_data =
    reinterpret_cast<unsigned char*> ( left.package_data() );
    const unsigned char* right_data =
    reinterpret_cast<unsigned char*> (right.package_data() );

    // определяем количество байт
    auto byte_count = left.package_count() * sizeof ( Package );

    //перемножаем в цикле по байтам
    for (decltype(byte_count) i = 0; i < byte_count; i++)
        r = r xor Table::multiply(left_data[i], right_data[i]);

    return r;
}
```

One procedure manipulates bytes and 4-element operands:



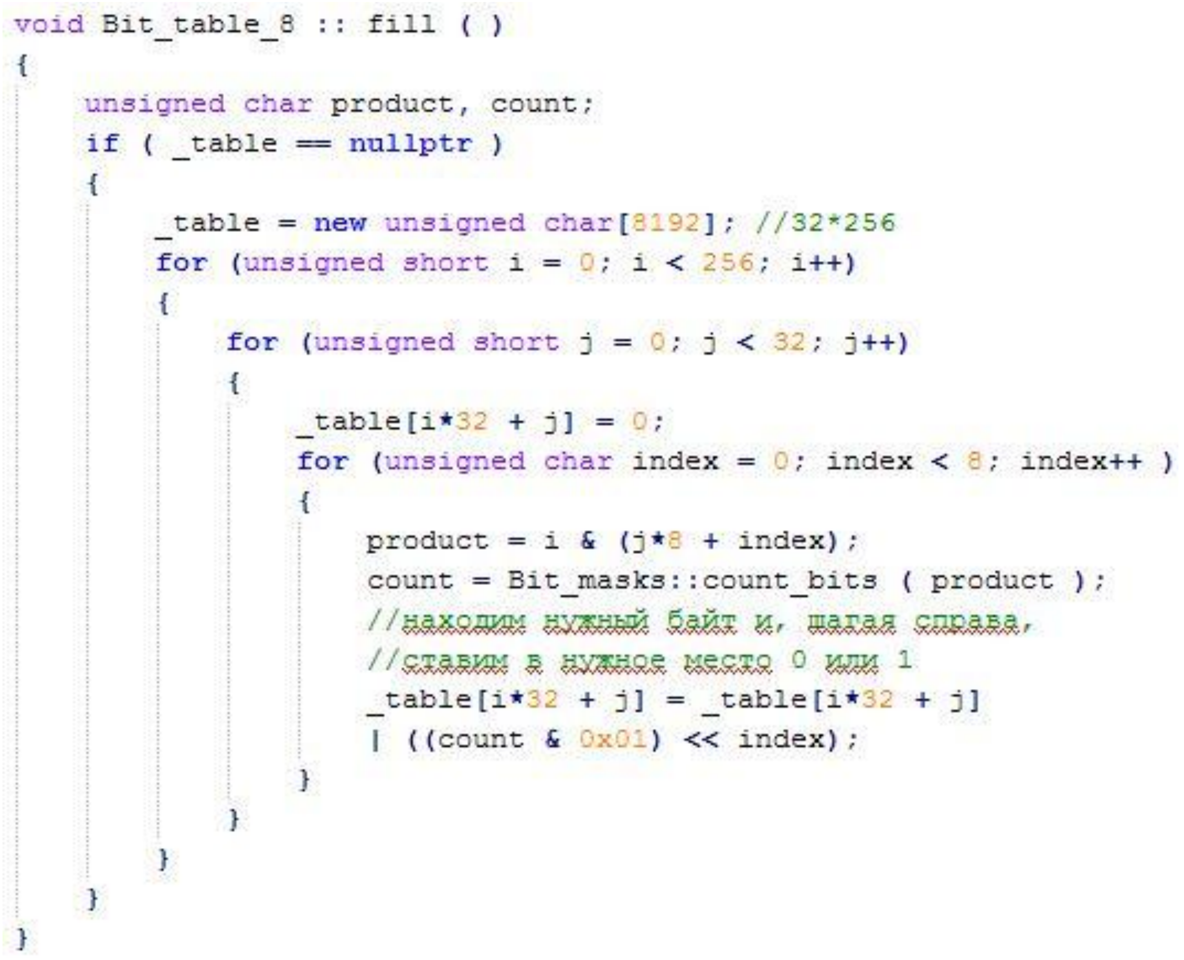
```
// процедура заполнения таблицы умножения
void Table_4 :: fill ( )
{
    unsigned char product, count;

    if ( _table == nullptr )
    {
        _table = new bool[256];

        for (unsigned char i = 0; i < 16; i++)
            for (unsigned char j = i; j < 16; j++)
            {
                product = i & j;
                count = Bit_masks::count_bits ( product, 4 );

                _table[i*16 + j] = (bool) (count & 0x01);
                _table[j*16 + i] = _table[i*16 + j];
            }
    }
}
```

The other procedure also uses bytes and operate an 8-element operands:



```
void Table_8 :: fill ( )
{
    unsigned char product, count;

    if ( _table == nullptr )
    {

        _table = new bool[65536];

        for (unsigned short i = 0; i < 256; i++)
            for (unsigned short j = i; j < 256; j++)
            {
                product = i & j;
                count = Bit_masks::count_bits ( product );

                _table[i*256 + j] = (bool) (count & 0x01);
                _table[j*256 + i] = _table[i*256 + j];
            }
    }
}
```

The third procedure sets densely packed operands in bits by size 8:



```
void Bit_table_8 :: fill ( )
{
    unsigned char product, count;
    if ( _table == nullptr )
    {
        _table = new unsigned char[8192]; //32*256
        for (unsigned short i = 0; i < 256; i++)
        {
            for (unsigned short j = 0; j < 32; j++)
            {
                _table[i*32 + j] = 0;
                for (unsigned char index = 0; index < 8; index++ )
                {
                    product = i & (j*8 + index);
                    count = Bit_masks::count_bits ( product );
                    //находим нужный байт и, шагая справа,
                    //ставим в нужное место 0 или 1
                    _table[i*32 + j] = _table[i*32 + j]
                    | ((count & 0x01) << index);
                }
            }
        }
    }
}
```

4.1.3 Gustavson method

The Gustavson method suits row-packed matrices. There is an interesting modification for a dense matrix in [16].

Express an element in a product matrix as

$$c(i; j) = (a(i; 0) \& b(0; j)) \text{ xor } \dots \text{ xor } (a(i; m) \& b(m; j)).$$

The method by definition is really inefficient when two operands are row-packed matrices. In this case, the standard way is matrix transposition, but it has also high computational cost. It is obvious that zeros could be canceled for disjunction; therefore, the number of operations for matrix multiplication could be reduced. It is the main idea according to Gustavson. The task is to find all non-zero elements in each row by bit masks. If a non-zero element $a(1; j)$ is found in the first row, relative matrix row $B(j)$ will be put into temp store for the first row of matrix product $C(1)$. Continue searching all non-zero elements in the first row of matrix *A*. In case of each suitable element *a*(1; *k*) add row *B*(*k*) to temp row *C*(1). When the row *A*(1) will be finished, there will be a matrix product sought to in the row *C*(1). Make the same for all other rows.

The disadvantage of this method is enumeration in matrix rows to find non-zero elements. To optimize it there is a simple iterator which allows to operate 4 elements of a row simultaneously. Switch case provides 16 possible combinations of 0 and 1.

Software implementations:

Bitwise operations are performed

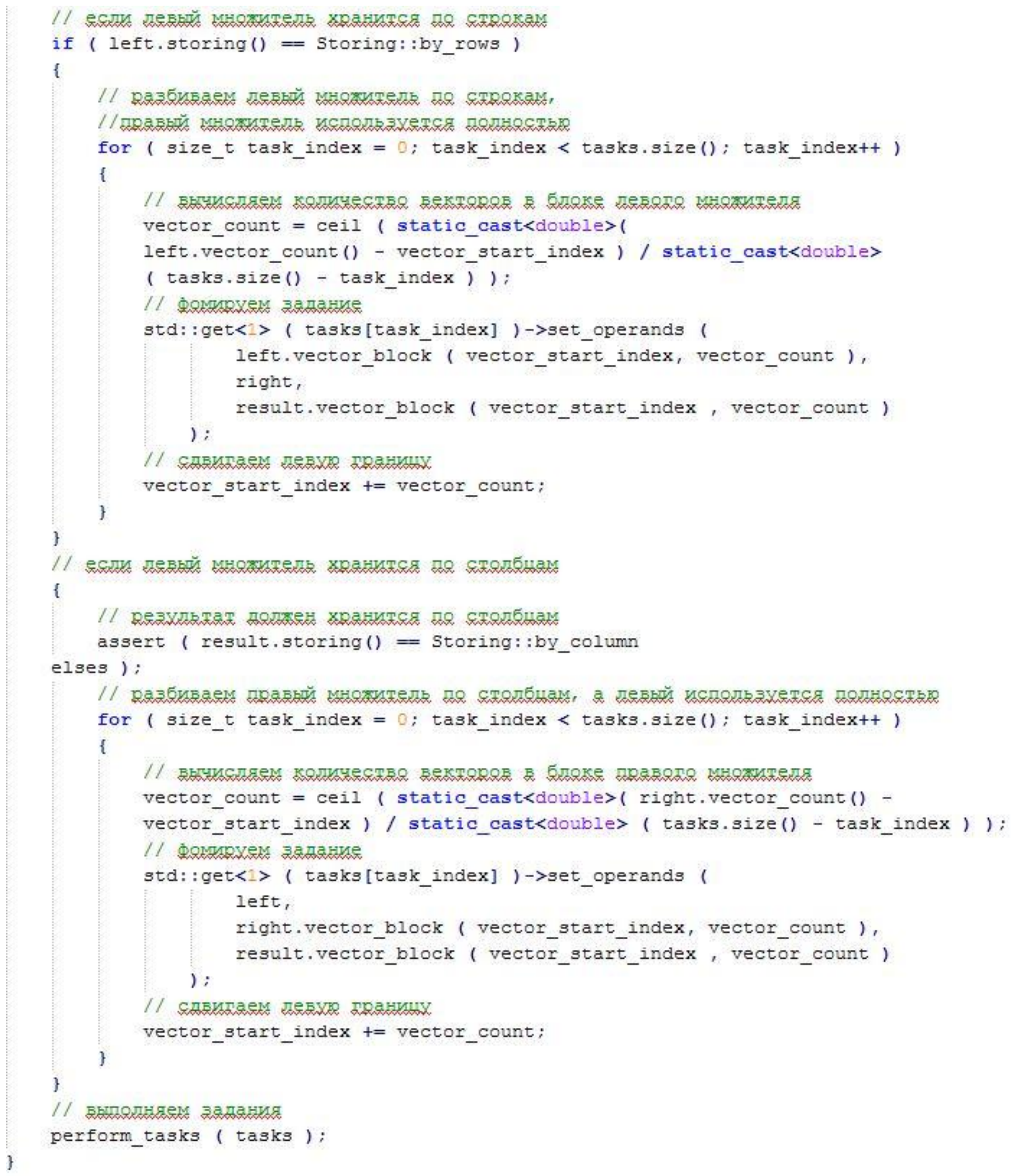
- for different data types (1-bit, 2-bit. 4-bit and 8-bit)
- for data blocks (MMX, SSE, and AVX for 64-bit, 128-bit and 256-bit relatively).

Execution time and performance are also estimated for optimized search with 4-element vectors.

Gustavson method is embedded in abo library:

```
template < typename Package, typename Vector_summator >
void Block_xor_multiplicator<Package, Vector_summator > :: multiply_add
( const Block<Package>& left, const Block<Package>& right, const Block<Package>& result )
{
    // количество элементов в строке левого операнда должно равняться количеству строк в
    //правом операнде
    assert ( left.column_count() == right.row_count());
    // количество строк левого операнда должно равняться количеству строк результата
    assert ( left.row_count() == result.row_count() );
    // количество столбцов правого операнда должно равняться количеству столбцов результата
    assert ( right.column_count() == result.column_count() );
    // режимы хранения операндов и результата должны быть одинаковыми
    assert ( left.storing() == result.storing() );
    assert ( right.storing() == result.storing() );
    // указатель на операнд, по которому производится итерации по векторам и элементам
    const Block<Package> * iterative_operand { nullptr };
    // указатель на операнд, векторы которого прибавляются к результату
    const Block<Package> * summand_operand { nullptr };
    // количество векторов
    typename Block<Package>::size_type vector_count { 0 };
    // количество элементов
    typename Block<Package>::size_type element_count { 0 };
    // если режим хранения по строкам
    if ( result.storing() == Storing::by_rows )
    {
        iterative_operand = &left;
        summand_operand = &right;
        vector_count = left.row_count ();
        element_count = left.column_count ();
    }
    else
    {
        iterative_operand = &right;
        summand_operand = &left;
        vector_count = right.column_count ();
        element_count = right.row_count ();
    }
    for (decltype(vector_count) vector_index = 0; vector_index < vector_count;
    vector_index++)
    {
        // инициализируем вектор результата
        auto result_vector = result.vector ( vector_index );
        // инициализируем итератор элементов текущего вектора итерируемого операнда
        auto element_iterator = iterative_operand->vector ( vector_index ).begin();
        // по всем элементам текущего вектора
        for ( decltype(element_count) element_index = 0; element_index < element_count;
        element_index++, element_iterator++ )
            // если элемент текущего вектора равен единице
            if ( static_cast<bool> (*element_iterator) )
                // прибавляем вектор суммируемого операнда к результату
                Vector_summator :: add (summand_operand->vector ( element_index ),
                result_vector);
    }
}
```

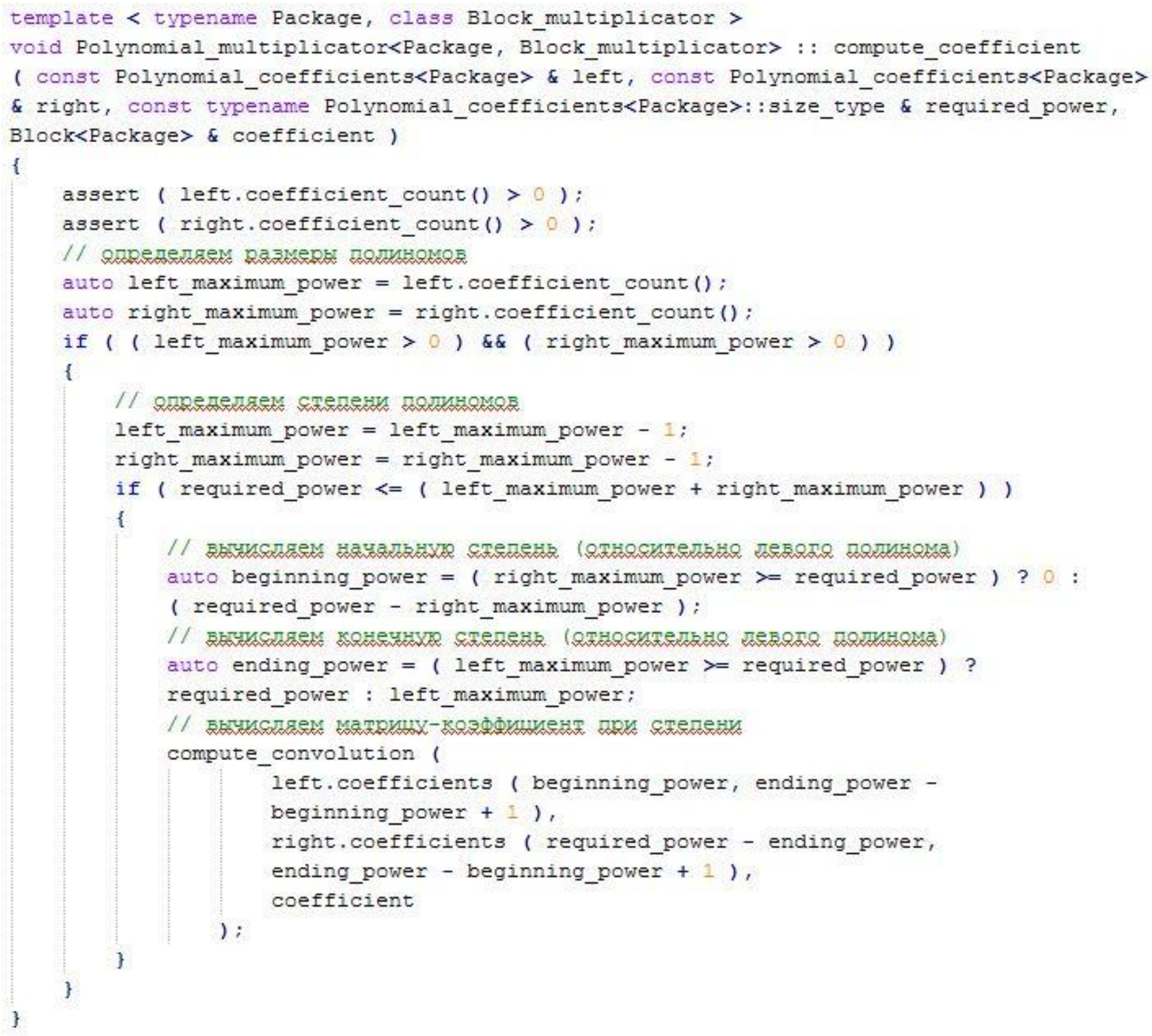
Matrix storage provides two storage options: by rows and by columns. It is important to check what type is used before running the function:



```cpp
    // если левый множитель хранится по строкам
    if ( left.storing() == Storing::by_rows )
    {
        // разбиваем левый множитель по строкам,
        //правый множитель используется полностью
        for ( size_t task_index = 0; task_index < tasks.size(); task_index++ )
        {
            // вычисляем количество векторов в блоке левого множителя
            vector_count = ceil ( static_cast<double>(
            left.vector_count() - vector_start_index ) / static_cast<double>
            ( tasks.size() - task_index ) );
            // фомируем задание
            std::get<1> ( tasks[task_index] )->set_operands (
                    left.vector_block ( vector_start_index, vector_count ),
                    right,
                    result.vector_block ( vector_start_index , vector_count )
                );
            // сдвигаем левую границу
            vector_start_index += vector_count;
        }
    }
    // если левый множитель хранится по столбцам
    {
        // результат должен хранится по столбцам
        assert ( result.storing() == Storing::by_column
    elses );
        // разбиваем правый множитель по столбцам, а левый используется полностью
        for ( size_t task_index = 0; task_index < tasks.size(); task_index++ )
        {
            // вычисляем количество векторов в блоке правого множителя
            vector_count = ceil ( static_cast<double>( right.vector_count() -
            vector_start_index ) / static_cast<double> ( tasks.size() - task_index ) );
            // фомируем задание
            std::get<1> ( tasks[task_index] )->set_operands (
                    left,
                    right.vector_block ( vector_start_index, vector_count ),
                    result.vector_block ( vector_start_index , vector_count )
                );
            // сдвигаем левую границу
            vector_start_index += vector_count;
        }
    }
    // выполняем задания
    perform_tasks ( tasks );
}
```

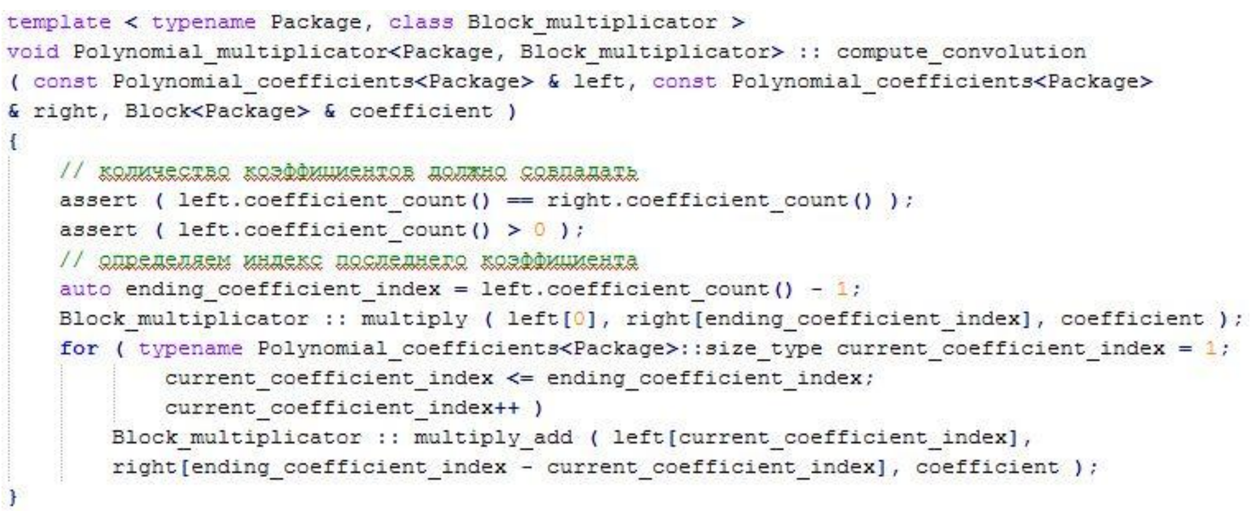
4.1.4 Options for polynomial matrices

Functions for matrix addition and multiplication allow us to extend them for the case of matrix polynomials. The current work has an interest in matrix polynomials because it is used in the Wiedemann-Coppersmith algorithm. One of the most frequently used operations for the polynomials in this method is searching the matrix-coefficient for a given degree of the polynomial:



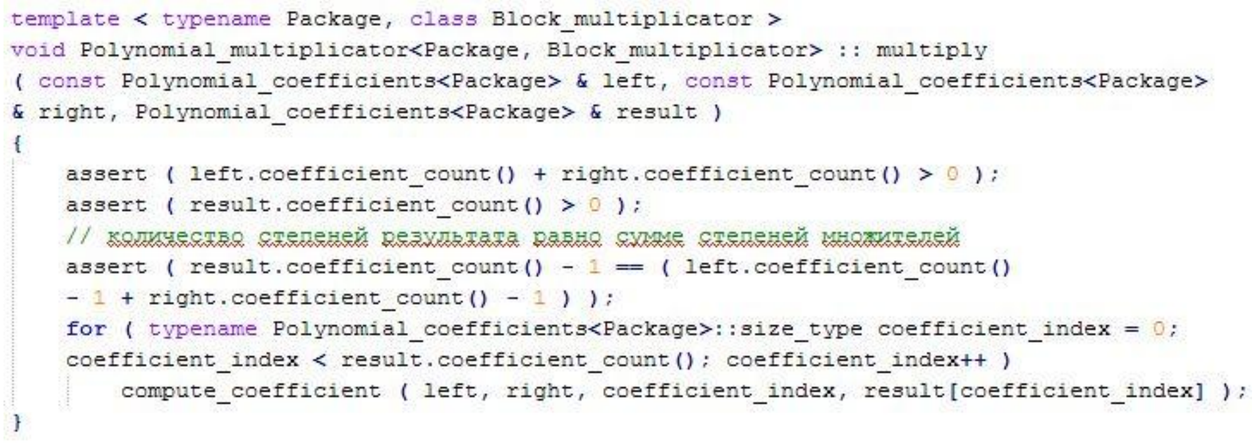
```
template < typename Package, class Block_multiplicator >
void Polynomial_multiplicator<Package, Block_multiplicator> :: compute_coefficient
( const Polynomial_coefficients<Package> & left, const Polynomial_coefficients<Package>
& right, const typename Polynomial_coefficients<Package>::size_type & required_power,
Block<Package> & coefficient )
{
    assert ( left.coefficient_count() > 0 );
    assert ( right.coefficient_count() > 0 );
    // определяем размеры полиномов
    auto left_maximum_power = left.coefficient_count();
    auto right_maximum_power = right.coefficient_count();
    if ( ( left_maximum_power > 0 ) && ( right_maximum_power > 0 ) )
    {
        // определяем степени полиномов
        left_maximum_power = left_maximum_power - 1;
        right_maximum_power = right_maximum_power - 1;
        if ( required_power <= ( left_maximum_power + right_maximum_power ) )
        {
            // вычисляем начальную степень (относительно левого полинома)
            auto beginning_power = ( right_maximum_power >= required_power ) ? 0 :
            ( required_power - right_maximum_power );
            // вычисляем конечную степень (относительно левого полинома)
            auto ending_power = ( left_maximum_power >= required_power ) ?
            required_power : left_maximum_power;
            // вычисляем матрицу-коэффициент при степени
            compute_convolution (
                    left.coefficients ( beginning_power, ending_power -
                    beginning_power + 1 ),
                    right.coefficients ( required_power - ending_power,
                    ending_power - beginning_power + 1 ),
                    coefficient
                );
        }
    }
}
```

It also applies to the function for matrix convolution:



```
template < typename Package, class Block_multiplicator >
void Polynomial_multiplicator<Package, Block_multiplicator> :: compute_convolution
( const Polynomial_coefficients<Package> & left, const Polynomial_coefficients<Package>
& right, Block<Package> & coefficient )
{
    // количество коэффициентов должно совпадать
    assert ( left.coefficient_count() == right.coefficient_count() );
    assert ( left.coefficient_count() > 0 );
    // определяем индекс последнего коэффициента
    auto ending_coefficient_index = left.coefficient_count() - 1;
    Block_multiplicator :: multiply ( left[0], right[ending_coefficient_index], coefficient );
    for ( typename Polynomial_coefficients<Package>::size_type current_coefficient_index = 1;
            current_coefficient_index <= ending_coefficient_index;
            current_coefficient_index++ )
        Block_multiplicator :: multiply_add ( left[current_coefficient_index],
        right[ending_coefficient_index - current_coefficient_index], coefficient );
}
```

The next necessary step is to implement a polynomial multiplication:



```
template < typename Package, class Block_multiplicator >
void Polynomial_multiplicator<Package, Block_multiplicator> :: multiply
( const Polynomial_coefficients<Package> & left, const Polynomial_coefficients<Package>
& right, Polynomial_coefficients<Package> & result )
{
    assert ( left.coefficient_count() + right.coefficient_count() > 0 );
    assert ( result.coefficient_count() > 0 );
    // количество степеней результата равно сумме степеней множителей
    assert ( result.coefficient_count() - 1 == ( left.coefficient_count()
    - 1 + right.coefficient_count() - 1 ) );
    for ( typename Polynomial_coefficients<Package>::size_type coefficient_index = 0;
    coefficient_index < result.coefficient_count(); coefficient_index++ )
        compute_coefficient ( left, right, coefficient_index, result[coefficient_index] );
}
```

4.2 Performance results for abo library

4.2.1 Testing technique

There are new matrix-operands for each method. Measuring time is called directly before and after the multiplication function. The timer is monotonic and independent of current system time. The resulting duration of the multiplication function is added for a given number of iterations to get a mean value.

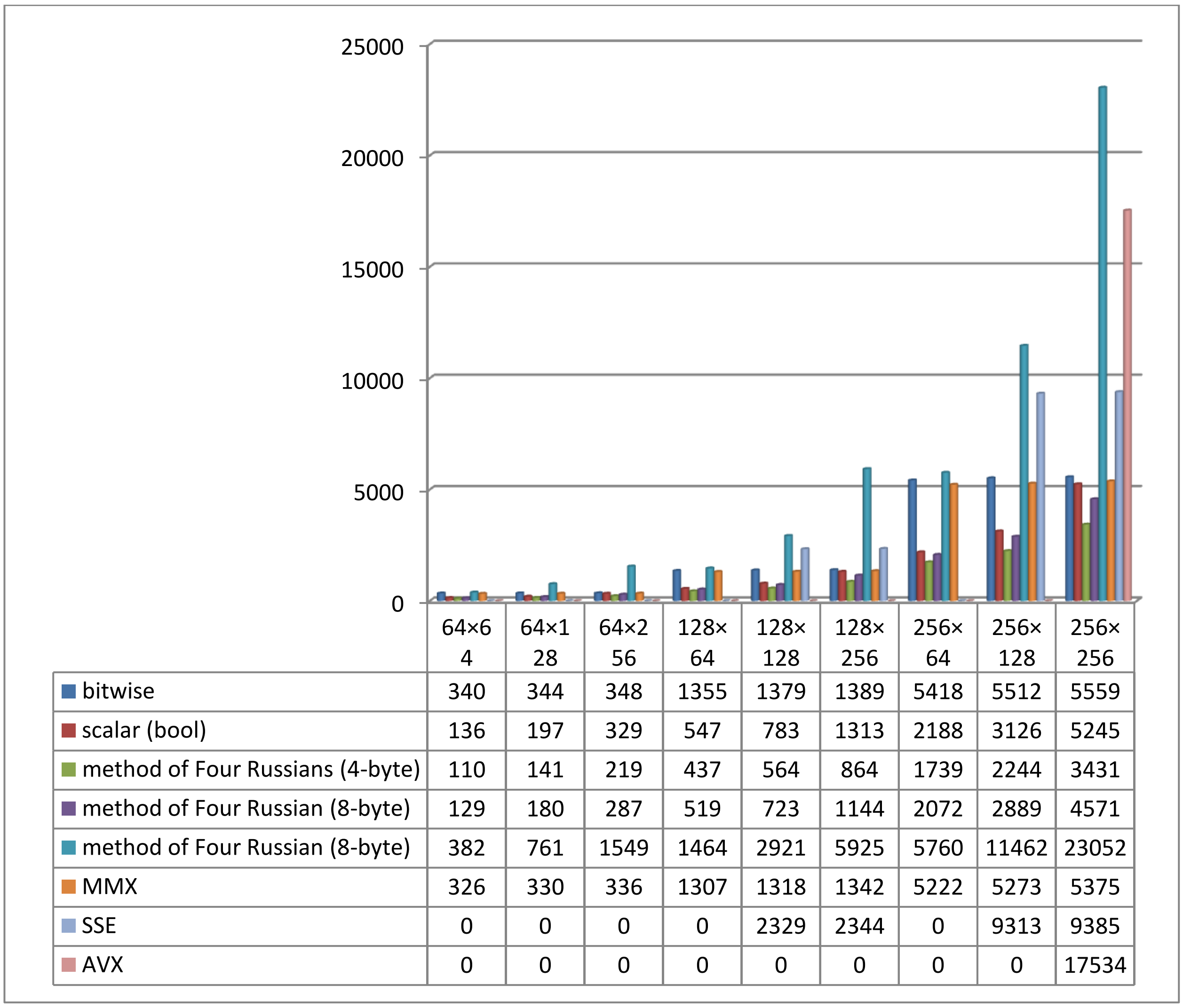
System characteristics:

- Platform: Ubuntu 14.04 (core 3.13.0-24-generic)
- Processor: Intel Xeon CPU E5-2690 v2 @ 3.00GHz, 10 cores, 20 threads
- Compiler: g++ 4.8.2
- Optimization flags: –Ofast –march=core-avx-i –mavx
- Number of iterations: 100
- Dataset: matrix sized from 64 to 2048 elements
- Timer: steady_clock

4.2.2 Results

The presented results of the bitwise method are performed for an 8-byte data type. This method manipulates densely packed operands. The scalar method uses logic data type and loosely packed vectors element-by-element. The method of Four Russians has three scores in the result table with respect to three different modifications. There is time complexity for element-wise process in Table 3.

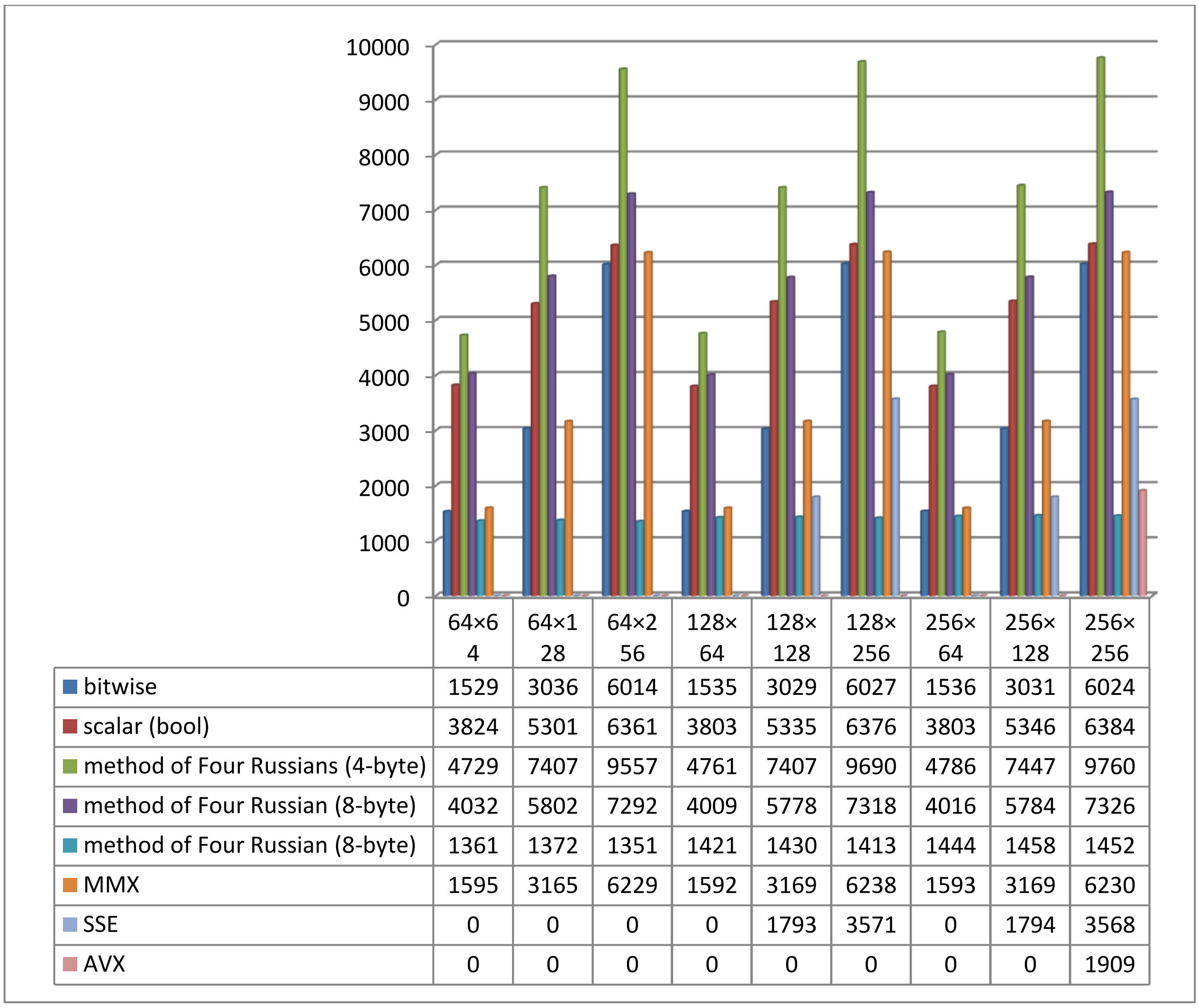
Table 3. Execution time item process methods for matrices sized from 64 to 256 elements by a row/column, sec $\times 10^{-6}$.



	64×64	64×128	64×256	128×64	128×128	128×256	256×64	256×128	256×256
🟦 bitwise	340	344	348	1355	1379	1389	5418	5512	5559
🟥 scalar (bool)	136	197	329	547	783	1313	2188	3126	5245
🟩 method of Four Russians (4-byte)	110	141	219	437	564	864	1739	2244	3431
🟪 method of Four Russian (8-byte)	129	180	287	519	723	1144	2072	2889	4571
🟦 method of Four Russian (8-byte)	382	761	1549	1464	2921	5925	5760	11462	23052
🟧 MMX	326	330	336	1307	1318	1342	5222	5273	5375
🟦 SSE	0	0	0	0	2329	2344	0	9313	9385
🟥 AVX	0	0	0	0	0	0	0	0	17534

Dependence of performance has a visual demonstration in Table 4. Performance is calculated as a ratio of execution time to the number of arithmetic operations (addition and multiplication) applied to matrix elements.

Table 4. Performance for element-wise process methods for matrices sized from 64 to 256 elements by a row/column, operations $\times 10^6$/sec.

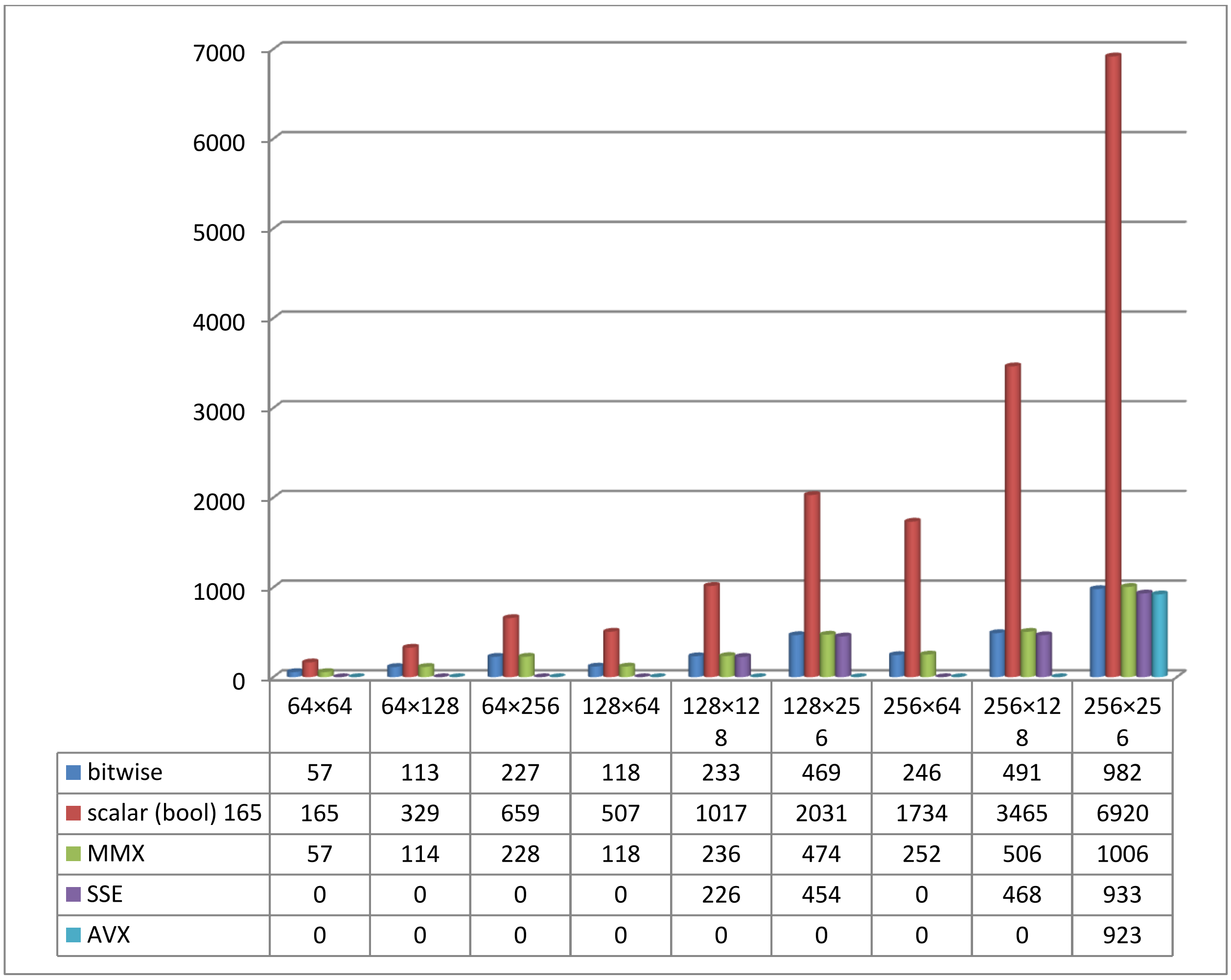


	64×64	64×128	64×256	128×64	128×128	128×256	256×64	256×128	256×256
🟦 bitwise	1529	3036	6014	1535	3029	6027	1536	3031	6024
🟥 scalar (bool)	3824	5301	6361	3803	5335	6376	3803	5346	6384
🟩 method of Four Russians (4-byte)	4729	7407	9557	4761	7407	9690	4786	7447	9760
🟪 method of Four Russian (8-byte)	4032	5802	7292	4009	5778	7318	4016	5784	7326
🟦 method of Four Russian (8-byte)	1361	1372	1351	1421	1430	1413	1444	1458	1452
🟧 MMX	1595	3165	6229	1592	3169	6238	1593	3169	6230
🟦 SSE	0	0	0	0	1793	3571	0	1794	3568
🟥 AVX	0	0	0	0	0	0	0	0	1909

The method of Four Russians with table for 8-element vectors has lower performance because the table is larger hence the search is harder. If this table has been densely packed, the modification would have had even lower performance. In this case, bitwise descriptor slows down processing rate.

Type of dependence for time complexity differs for block multiplication methods. It is shown in table 5.

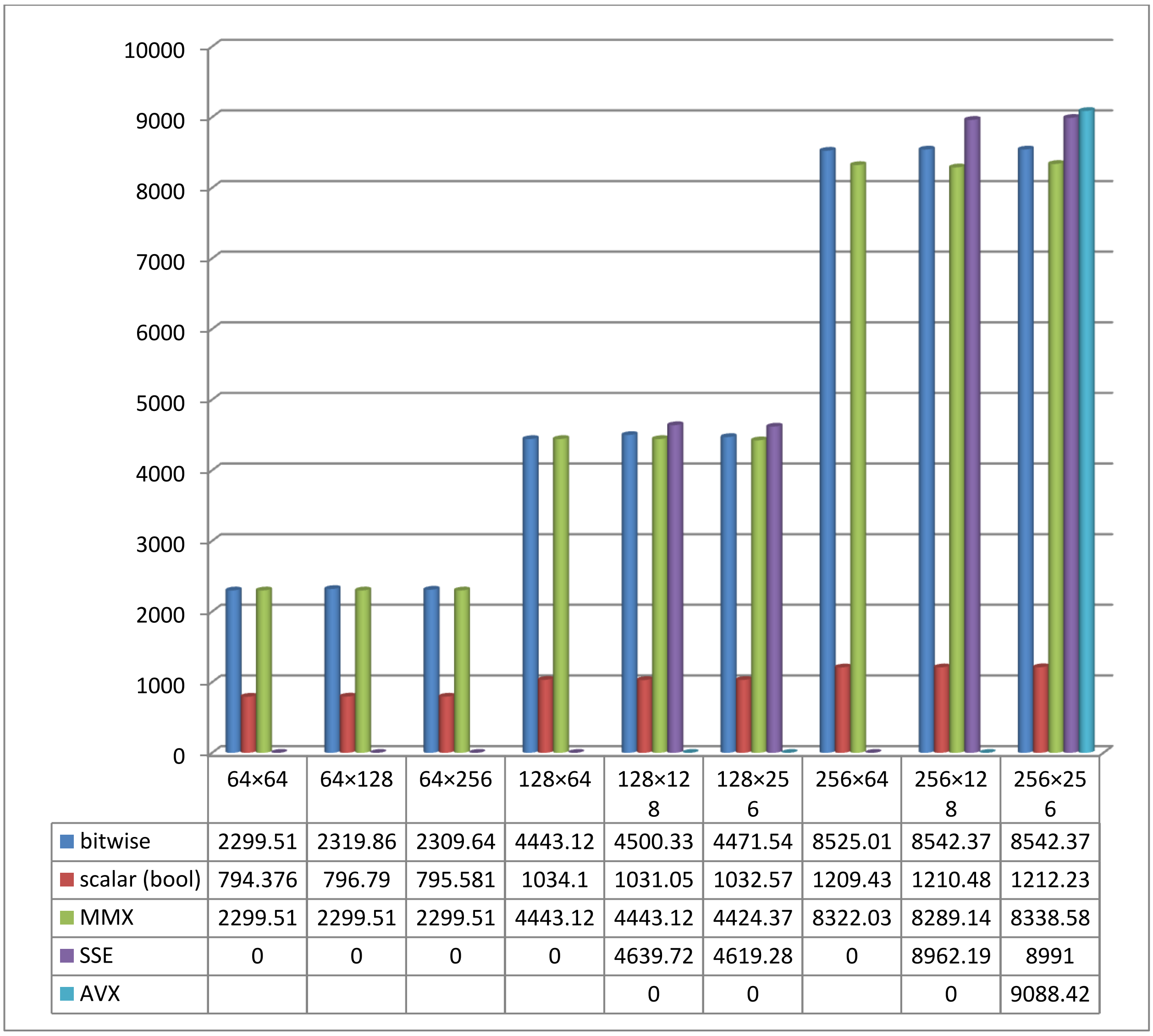
Table 5. Execution time for block operations methods for matrices sized from 64 to 256 elements by a row/column, sec $\times 10^{-6}$.



	64×64	64×128	64×256	128×64	128×128	128×256	256×64	256×128	256×256
🟦 bitwise	57	113	227	118	233	469	246	491	982
🟥 scalar (bool) 165	165	329	659	507	1017	2031	1734	3465	6920
🟩 MMX	57	114	228	118	236	474	252	506	1006
🟪 SSE	0	0	0	0	226	454	0	468	933
🟦 AVX	0	0	0	0	0	0	0	0	923

Conversion to performance exhibits a slight gap between intrinsic-functions and others modifications (table 6).

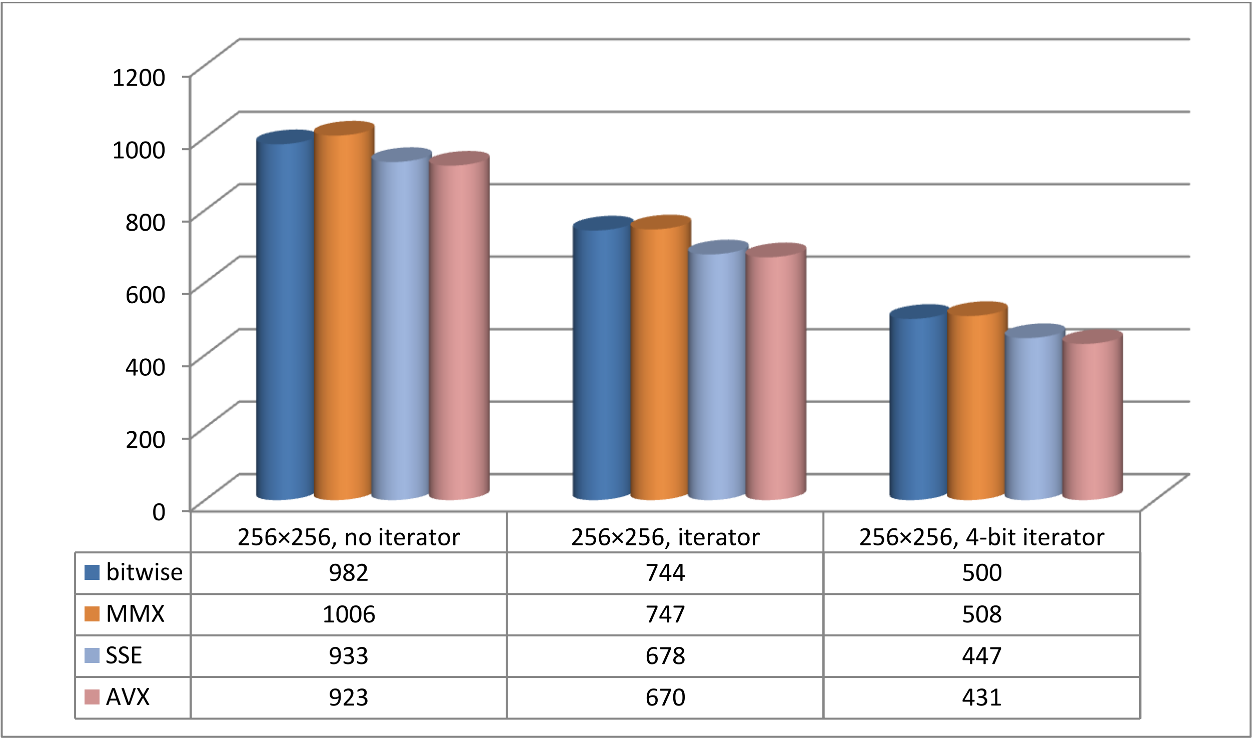
Table 6. Performance for the Gustavson method for matrices sized from 64 to 256 elements by a row/column, operations $\times 10^6$/sec.



	64×64	64×128	64×256	128×64	128×128	128×256	256×64	256×128	256×256
🟦 bitwise	2299.51	2319.86	2309.64	4443.12	4500.33	4471.54	8525.01	8542.37	8542.37
🟥 scalar (bool)	794.376	796.79	795.581	1034.1	1031.05	1032.57	1209.43	1210.48	1212.23
🟩 MMX	2299.51	2299.51	2299.51	4443.12	4443.12	4424.37	8322.03	8289.14	8338.58
🟪 SSE	0	0	0	0	4639.72	4619.28	0	8962.19	8991
🟦 AVX					0	0		0	9088.42

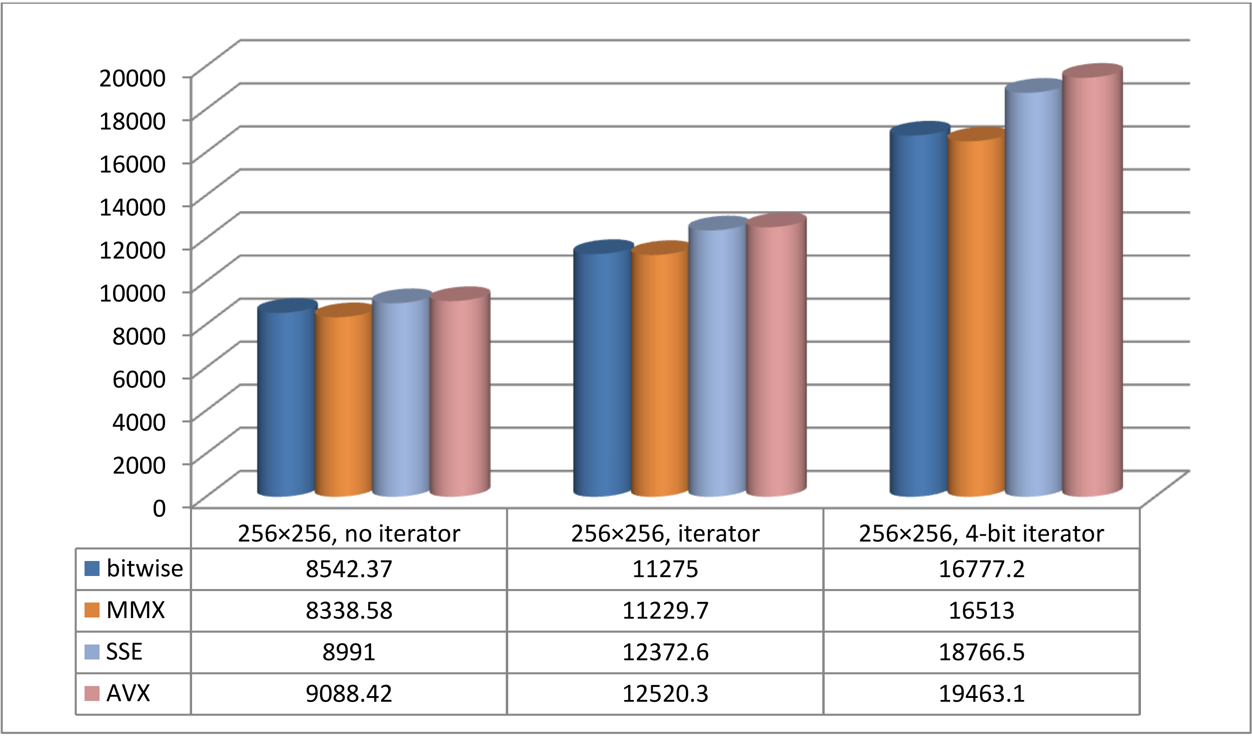
For these methods using iterators to 1-bit or 4-bit elements [21] there are also bars for time complexity function (table 7) and performance (table 8).

Table 7. Dependence of time complexity for Gustavson method with iterators, sec $\times 10^{-6}$.



	256×256, no iterator	256×256, iterator	256×256, 4-bit iterator
🟦 bitwise	982	744	500
🟧 MMX	1006	747	508
🟦 SSE	933	678	447
🟥 AVX	923	670	431

Table 8. Performance for Gustavson method with iterations, operations $\times 10^{6}$/sec.



	256×256, no iterator	256×256, iterator	256×256, 4-bit iterator
🟦 bitwise	8542.37	11275	16777.2
🟧 MMX	8338.58	11229.7	16513
🟦 SSE	8991	12372.6	18766.5
🟥 AVX	9088.42	12520.3	19463.1

Let us consider the dependence between performance and number of zero elements in matrices sized 256×256 with p as a probability for generated zero elements:

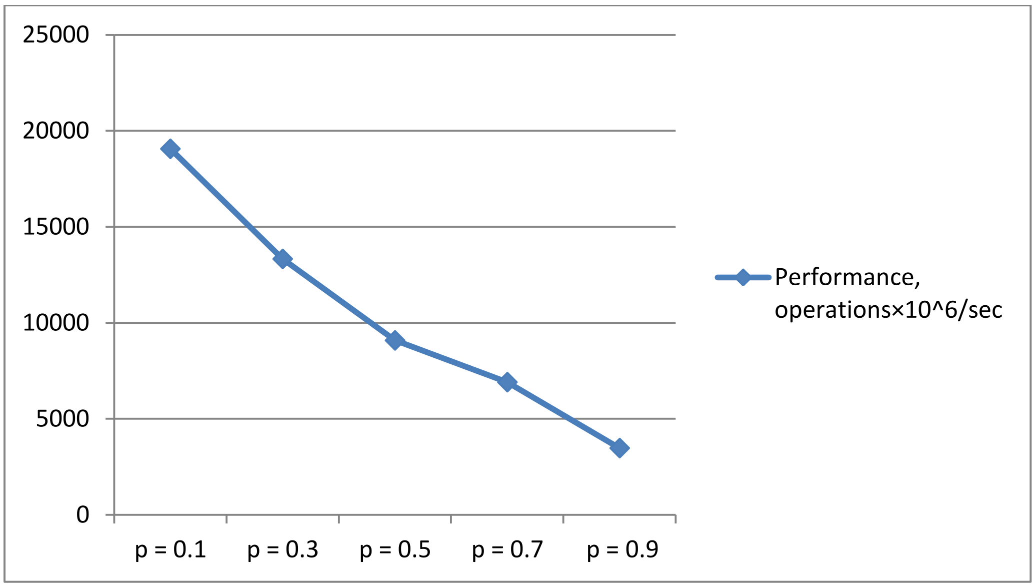


Figure 4. The performance curve dependence on the number of zero elements.

The character of the performance function is interesting from the cache memory point of view. There are two levels of cache. When one level is filled, performance has a dramatic drop (figures 5 – 7). Let us choose storage by rows in such a way that increasing number of columns.

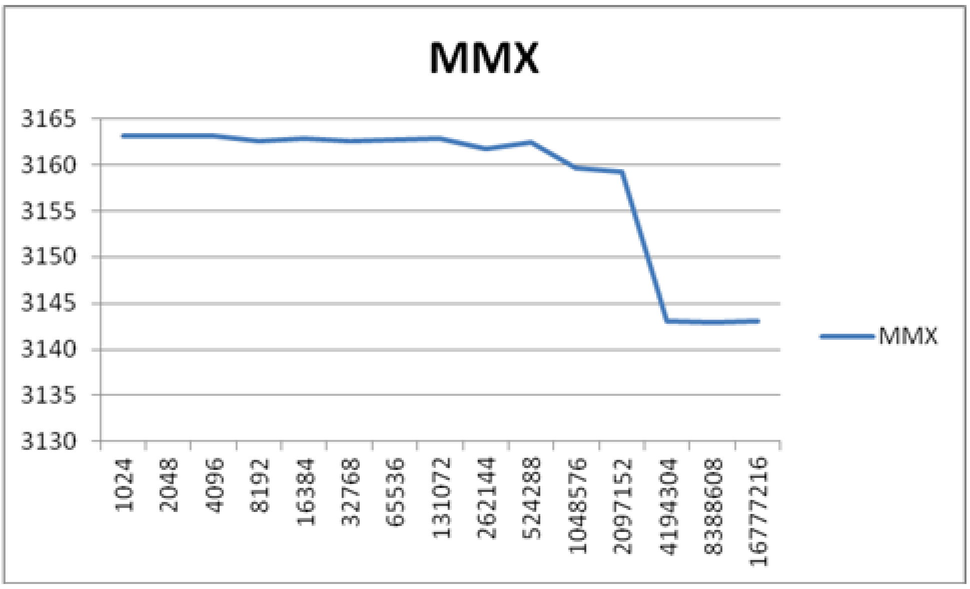


Figure 5. Performance for a matrix with row size = 64 and column size from 210 = 1024 to 224 = 16777216, operations $\times 10^6$/sec.

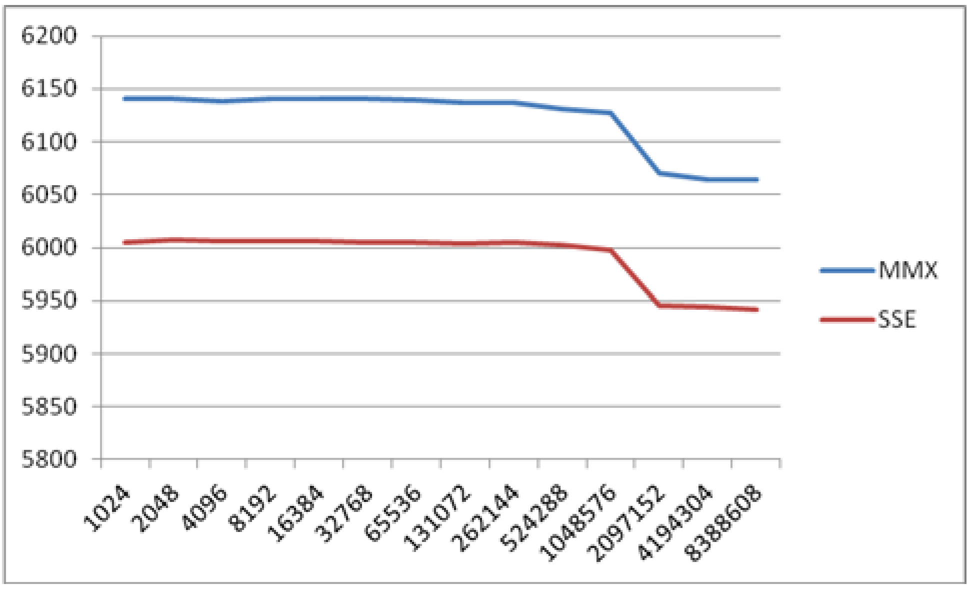


Figure 6. Performance for a matrix with row size = 128 and column size from 210 = 1024 to 223 = 8388608, operations $\times 10^6$/sec.

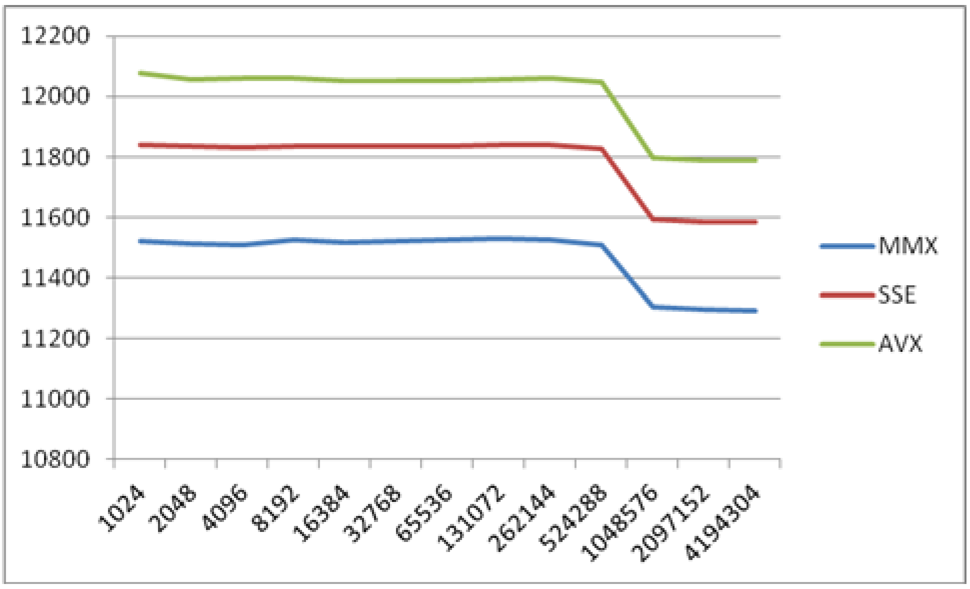


Figure 7. Performance for a matrix with row size = 256 and column size from 210 = 1024 to 222 = 4194304, operations $\times 10^6$/sec.

4.3 Performance data for external libraries provided boolean matrix multiplication

There are test results for some platforms (tables 9 – 13) using such libraries as MAGMA, GAP and NTL [22]. Input data is two random square Boolean matrices. GAP operates the method of Four Russians, Magma and M4RI use the method of Four Russians applied to the Strassen-Winograd algorithm. All time results are in seconds.

Table 9. Test results for 64-bit Debian/GNU Linux, 2.33Ghz Intel Core2Duo (MacbookPro2,2).

Matrix Dimension	**Magma 2.14-17 (64-bit)**	**GAP 4.4.10 (64-bit)**	**M4RI-20080821 (64-bit)**
10,000 x 10,000	1.892	6.130	1.504
16,384 x 16,384	7 .720	25.048	6.074
20,000 x 20,000	13.209	-	10.721
32,000 x 32,000	53.668	-	43.197

Table 10. Test results for 64-bit Debian/GNU Linux, 2.6Ghz Intel i7 (MacbookPro6,2).

Matrix Dimension	**Magma 2.15-10 (64-bit)**	**GAP 4.4.12 (64-bit)**	**M4RI-20121224 (64-bit)**
10,000 x 10,000	1.200	6.524	0.942
16,384 x 16,384	4.735	20.777	3.672
20,000 x 20,000	8.085	-	6.660
32,000 x 32,000	33.395	-	28.006

Table 11. Test results for 64-bit Debian/GNU Linux, 2.6Ghz AMD Opteron 858 (VMWare Virtualised).

Matrix Dimension	**Magma 2.14-13 (64-bit)**	**GAP 4.4.12 (64-bit)**	**M4RI-200100817 (64-bit)**
10,000 x 10,000	2.855	10.256	2.656
16,384 x 16,384	10.865	-	11.470
20,000 x 20,000	19.505	-	18.929
32,000 x 32,000	70.110	-	66.208

Table 12. Test results for 64-bit Ubuntu Linux, 2.6Ghz AMD Athlon X2.

Matrix Dimension	**Magma 2.14-13 (64-bit)**	**GAP 4.4.10 (64-bit)**	**M4RI-200100817 (64-bit)**
10,000 x 10,000	2.005	5.920	2.700
16,384 x 16,384	7.625	24.180	10.100
20,000 x 20,000	13.870	-	19.120
32,000 x 32,000	51.155	-	73.725

Table 13. Test results for 64-bit RHEL 5 Linux, 1.6Ghz Intel Itanium.

Matrix Dimension	**Magma 2.14-16 (64-bit)**	**GAP 4.4.10 (64-bit)**	**M4RI-20080909 (64-bit)**
10,000 x 10,000	7.941	-	4.200
16,384 x 16,384	31.046	-	16.430
20,000 x 20,000	55.654	-	28.830
32,000 x 32,000	209.483	-	109.414

4.4 Summary for the developed abo library

For small matrices sized 64-256 element Gustavson method is faster and the fastest one is the modification for a 256×256 matrix with AVX and iterator implementations. Its performance equals 19463.1 operations in a microsecond.

For large matrices, the optimum number of columns for a given number of rows when there is no noticeable performance reduction for different row-by-row methods is given in table 14.

Table 14. Cache-optimum number of columns for a given number of rows.

Number of rows	MMX	SSE	AVX
64	$< 2^{21}$	-	-
128	$< 2^{20}$	$< 2^{20}$	-
256	$< 2^{19}$	$< 2^{19}$	$< 2^{19}$

This tendency is due to the size of the second-level cache in the operating system.

Time result data are given for functions without iterators but even then there is a much greater processing speed compared with data reported for external libraries. For example, take a 256×2^19 matrix (what is greater than 10^8) in Gustavson method with $n \cdot m \cdot n(p-1)$ complexity, where $n \cdot m$ is a matrix size and *p* is a probability for generated zero elements. Here in the test results, it takes 5.5924 milliseconds. It is the best time result compared to any of the three external libraries in case of GF(2).

5 CONCLUSION

Algorithms in the current work were chosen due to their high practical importance. They are the most efficient ones and therefore the most popular algorithms in different fields, mainly in the problem of data decryption. But there are a lot of studies directed at extending or modifying them for attaining even higher efficiency. Since their publication, many efficient modifications have appeared. In our current work, we have tried to start from a theoretical model and came to implementations for the Montgomery and the Wiedemann-Coppersmith methods. It was done with the aim of efficient development based on a detailed computational cost analysis for each step of the methods. Original algorithms, as well as their modern modifications (such as Berlekamp–Massey and Thomé versions for the Wiedemann-Coppersmith algorithm), were considered. The choice of method was based on a review in related fields. Computational efficiency and high probability for result achievement were key parameters for algorithms.

Building a theoretical model included a detailed assessment of the computational complexity of all the relevant operations. This approach allows educated organization for method development and test estimation for further optimization.

The most significant contribution to improving the efficiency of software implementations of the algorithms was the decision through special library operations for the case of binary vectors and matrices. Theoretical estimation shows that multiplication of large matrices occupies quite a large share of total execution time. Acceleration of it reduces the time complexity of the entire algorithm. It should be clarified that for this operation there is a special data record type of binary matrices and vectors. Also, a computational tool of elementary operations for it was prepared. Test results for the abo library demonstrate its advantages compared to existing well-known libraries such as Magma, GAP, M4RI (in the case of GF(2)).

The current state of research for Montgomery and Wiedemann-Coppersmith algorithms was discussed, theoretical analysis and estimation for computational cost were provided. An optimized implementation for two algorithms and a library of computationally intensive binary operations was developed. Further use will include commercial use and enhancing derived software solution.

This work may be developed further in the field of parallel architecture for the Montgomery and Wiedemann-Coppersmith algorithms.

References

[1] P. Montgomery. A block Lanczos algorithm for finding dependencies over GF(2). // Lecture Notes in Computer Science. 921. Springer-Verlag, 1995. P. 106-120.

[2] D.H. Wiedemann. Solving sparse linear equations over finite fields. // IEEE Trans. Inform. Theory, 1986, 32, 1. P. 54–62.

[3] Don Coppersmith. Solving Homogeneous Linear Equations over GF(2) via Block Wiedemann Algorithm. // Mathematics of Computation. 1995. 62, 205. P. 333–350.

[4] T. Kleinjung, K. Aoki, J. Franke, et al. Factorization of a 768-bit RSA modulus //CRYPTO-2010. LNCS. 2010. V. 6223. P. 333-350.

[5] Y. Saad. On the Lanczos Method for Solving Symmetric Linear Systems with Several right-hand Sides. // Mathematics of Computation. 1987. 48, 178.

[6] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems // Journal of Research of the National Bureau of Standards. 1952. 49, 6.

[7] Y. Saad and M.H.Schultz. GMRES: a generalized minimum residual algorithm for solving nonsymmetric linear systems. // SIAM J. Sci. Stat. Comput. 1986. 7. 856–857.

[8] Don Coppersmith. Solving Linear Equations over GF(2): Block Lanczos Algorithm. // Linear Algebra Appl. 1993. v. 193 pp. 33–60.

[9] M. Peterson and C. Monico. F_2 Lanzcos revisited. // Linear Algebra Appl. 2008. 428. 1135–1150.

[10] J.L. Massey. Shift-Register Synthesis and BCH Decoding. // IEEE Trans. Inform. Theory, 1969, 15,1.

[11] F.G. Gustavson and D.Y. Yun. Fast Algorithms for Rational Hermite Approximation and Solution of Toeplitz Systems. // IEEE Trans. Circ. Syst., 1979, 26, 9.

[12] E. Thomé. Fast computation of linear generators for matrix sequences and application to the block Wiedemann algorithm // Proc. ISSAC. 2010. P. 323-331.

[13] Н.Л. Замарашкин. Алгоритмы для разреженных систем линейных уравнений в GF(2). – М.: Издательство Московского университета. 2013.

[14] B. Hovinen. Blocked Lancsoz-style Algorithms over Small Finite Fields. University of Waterloo, 2004.

[15] А. Ахо, Дж. Хопкрофт, Дж. Ульман. Построение и анализ вычислительных алгоритмов. – М.: Мир, 1979.

[16] F. G. Gustavson. Two fast algorithms for sparse matrices: multiplication and permuted transposition // ACM Trans. Math. Software (TOMS). 1978. Vol. 4. № 3. P. 250 – 269.

[17] С. Писсанецки. Технология разреженных матриц. – М.: Мир, 1977.

[18] И.Б. Мееров, А.В. Сысоев. Разреженное матричное умножение. – Н.Новгород: ННГУ, 2011.

[19] Intel Intrinsic functions. Available online at https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[20] The method of Four Russians. Available online at http://neerc.ifmo.ru/wiki/index.php?title=Метод_четырех_русских_для_умножения_матриц

[21] B. Stroustrup. The C++ Programming Language. Addison-Wesley, 2013.

[22] Benchmarketing results for M4RI and other systems. Available online at http://malb.bitbucket.org/m4ri-e-website-2008-2015/performance.html

List of Tables

List of Figures