ELEKTRYKA 2016
Zeszyt 1 (237) Rok LXII

Marcin SOWA
Silesian University of Technology, Gliwice

SIMPLE C# CLASSES FOR FAST MULTIVARIATE POLYNOMIAL SYMBOLIC COMPUTATION – PART III: POLYNOMIAL MULTIPLICATION ALGORITHM AND ITS EFFICIENCY

Summary. Part I of the paper represented the implementation details of a lightweight C# implementation of symbolic computation of multivariate polynomials. Part II represented the results of an analysis of the numerical efficiency. The current part discusses the multiplication and exponentiation operations. An original algorithm for sparse multivariate polynomial multiplication is proposed. An analysis on polynomial exponentiation is also performed.

Keywords: symbolic computation, computation time, C# implementation, polynomial multiplication

PROSTE KLASY W JĘZYKU C# DO OBLICZEŃ SYMBOLICZNYCH WIELOMIANÓW WIELU ZMIENNYCH – CZĘŚĆ III: ALGORYTM MNOŻENIA WIELOMIANÓW I JEGO EFEKTYWNOŚĆ

Streszczenie. Część I niniejszego artykułu zawierała szczegóły prostej implementacji obliczeń symbolicznych wielomianów wielu zmiennych. W części II przedstawiono wyniki analizy efektywności numerycznej. Niniejsza część omawia operacje mnożenia i potęgowania. Zaproponowano oryginalny algorytm do mnożenia rzadkich wielomianów wielu zmiennych. Przeprowadzona jest również analiza na temat potęgowania wielomianów.

Słowa kluczowe: obliczenia symboliczne, czas obliczeń, implementacja C#, mnożenie wielomianów

1. INTRODUCTION

The efficient multiplication of polynomials is an important issue on its own, which is why this separate part is dedicated to the discussion of how it can be carried out and, most of all, how it is done in the proposed C# implementation.

1.1. Sparse and dense polynomial analysis

Much research in the past has been dedicated to the topic of polynomial multiplication. The issue is often divided into the cases of dense and sparse polynomials [1]. There is no strict distinction as to this classification – there are cases in practice where one cannot tell whether a polynomial is dense or sparse [2]. However, the methodologies in which they are generally handled are much different. It is analogous to the way matrices are represented and dealt with, where dense polynomials can be (without much loss of memory) placed in n-dimensional arrays allowing for a fast access to its monomial terms.

With a purely sparse representation with linked lists (as the one applied in the proposed implementation) one allows for versatility without the requirement to initially know the number of terms or the exponentiations in each monomial term. The most known drawback is that in a linked list – one should avoid long searches for a specific term (the memory access speed critically affects the computation time for polynomial multiplication as was concluded in [3]). However, one can overcome this with the proper algorithms. For many reasons it is favorable to implement polynomials as being sparse. First off, a general support is provided for analyses where polynomials of the form:

$$\zeta(\boldsymbol{s}) = \sum_{i=1}^{N} a_i \prod_{j=1}^{n} {s_j}^{k_{i,j}}, \quad a_i \in \mathbb{R},\ k_{i,j} \in \mathbb{N}_0. \tag{1}$$

are concerned (where $\boldsymbol{s}$ represents a vector containing all symbolic variables s_1, s_2, s_3, …, s_n). Secondly, the amount of required memory is reduced since zero-coefficient terms do not need to be stored. Finally, the zero terms are not interpreted in each operation, which reduces the computation time.

1.2. Multivariate and univariate polynomials

Much effort is dedicated to improve the efficiency of symbolic polynomial multiplications. Most of these are focused on an assumption that the polynomials are in dense form and their degree is known (like in [4], [5]). Moreover, the development of methods (and the analyses of their complexities) are often only performed for univariate polynomials. The consideration of the univariate case is strongly justified in theory, as generally one can define a Kronecker map [6] for multivariate polynomials to simplify the analysis (e.g. as presented in [7]). This allows the well-known fast methods for univariate polynomials to be applied to multivariate ones as well.

1.3. Commonly discussed algorithms

One of the most known methods for fast polynomial multiplications is the application of the FFT based formulae presented i.a. in [8] (often attributed to Gentleman and Sande [9]). It is most known for its use in univariate polynomial multiplication, though attempts have been made at generalizing it to the multivariate case [10]. The assumption is, however, that common multipliers between the polynomials are found first. If none actually exist then the algorithm has no advantages. The FFT has also been concluded to be not applicable for sparse polynomials in [11].

In the past years there have been some papers definitely worth mentioning where original methods for fast polynomial multiplications are presented. In the author's opinion – those that deserve the most attention are the ones that deal with the general, most difficult case of sparse polynomials of arbitrary numbers of variables as in [12, 13].

Most interestingly the method still recently being referred to as the fastest available, in terms of sequential computation, is the one proposed by Johnson in 1974 [14]. It comprises of steps, where first the monomial multiplications are performed, and then the results are ordered. Rightly it has been noticed that the number of term comparisons (hence also iterations) affects the computation time, thus their reduction is suggested.

2. PROPOSED ALGORITHM FOR POLYNOMIAL MULTIPLICATION

With the term comparison minimization in mind – the author has proposed his own, simple algorithm, which mostly is directed at minimizing the number of possible term comparisons. The considered operation is:

$$C = A \cdot B = (a_0 + a_1 + ... + a_{N_A}) \cdot (b_0 + b_1 + ... + b_{N_B}), \tag{2}$$

where it is assumed that $N_A \geq N_B$. As mentioned by Johnson in his paper, when the polynomials A and B are ordered then in the result (also ordered) the monomial product $a_i b_j$ will appear after $a_{i-k}b_j$ and $a_i b_{j-k}$, and before $a_{i+k}b_j$ and $a_i b_{j+k}$ (with $k > 0$). There are various ways in how this can be used, however one must also keep in mind not to add more numerical weight to the algorithm. In the author's algorithm N_B auxiliary references are stored (which is why, internally, B refers to the polynomial with less terms) along with arrays for quick access to the terms of both polynomials. The method follows the steps depicted in Figure 1.

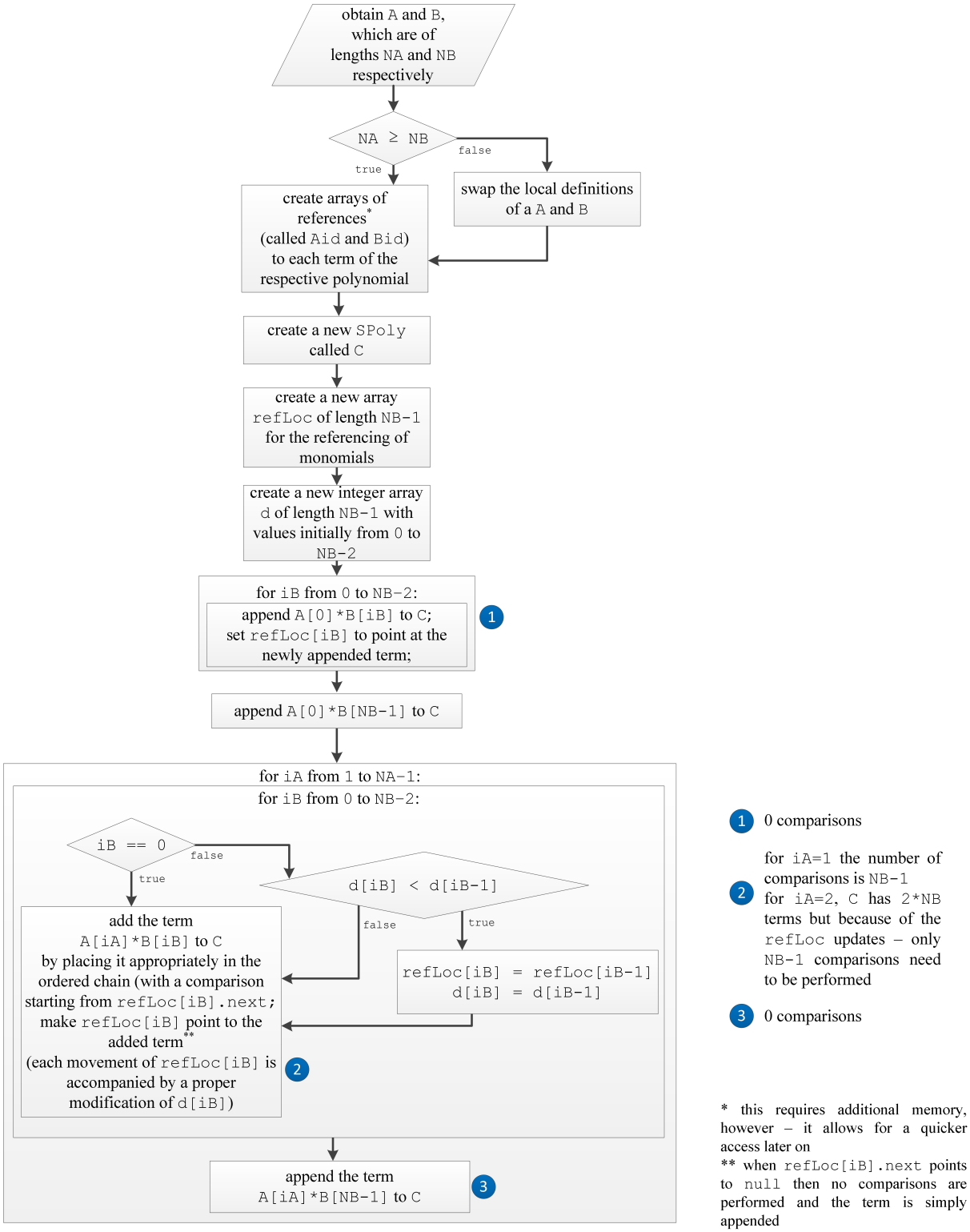

Fig.1. The proposed symbolic polynomial multiplication algorithm
Rys.1. Zaproponowany algorytm symbolicznego mnożenia wielomianów

From the figure it can be noticed that:

I) `refLoc[iB]` being updated (if `refLoc[iB-1]` points further, what is denoted by the `d` integers) allows to use the property that $a_i b_j$ appears after $a_i b_{j-k}$,

II) each `refLoc[iB]` being saved allows it to be used for the next `iA` iteration, hence the property that $a_i b_j$ appears after $a_{i-k} b_j$ is used,

III) the core of the algorithm uses exactly $(N_A - 1)\cdot(N_B - 1)$ comparisons.

3. ANALYSIS OF NUMERICAL EFFICIENCY

In order to ascertain the efficiency of the multiplications performed in the C# implementation with the use of the proposed algorithm – several test trials have been performed where a polynomial A of N_A terms is multiplied by a polynomial B of N_B terms, where for the i-th test:

$$N_A = i \cdot 2000 + 1, \tag{3}$$

and:

$$N_B = (i+1) \cdot 2000 + 1. \tag{4}$$

For the computation time results to be more credible, 30 internal trials have been completed for each i and the average values have been taken as the actual final results. The results for the subsequent tests are presented in Figure 2. The required memory has not been studied as it has already been discussed in part II of the paper.

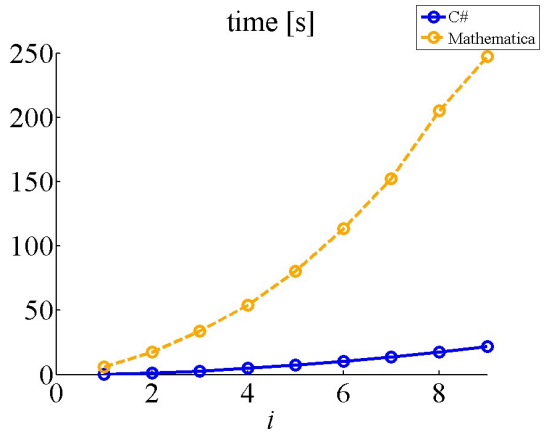

Fig.1. Comparison of the computation times for polynomial multiplications using the proposed C# implementation and Mathematica

Rys.1. Porównanie czasów obliczeń dla mnożenia wielomianów w zaproponowanej implementacji C# i w oprogramowaniu Mathematica

As was concluded in part II it is obvious that the C# implementation performs faster for such operations as it handles simpler base operations. However, to judge the implementation in relation to N_A and N_B – an additional dependency has been derived. Since all multiplication algorithms actually require $N_A \cdot N_B$ monomial multiplications – the computation times are compared with this number in the next figure (along with linear regressions of the plots).

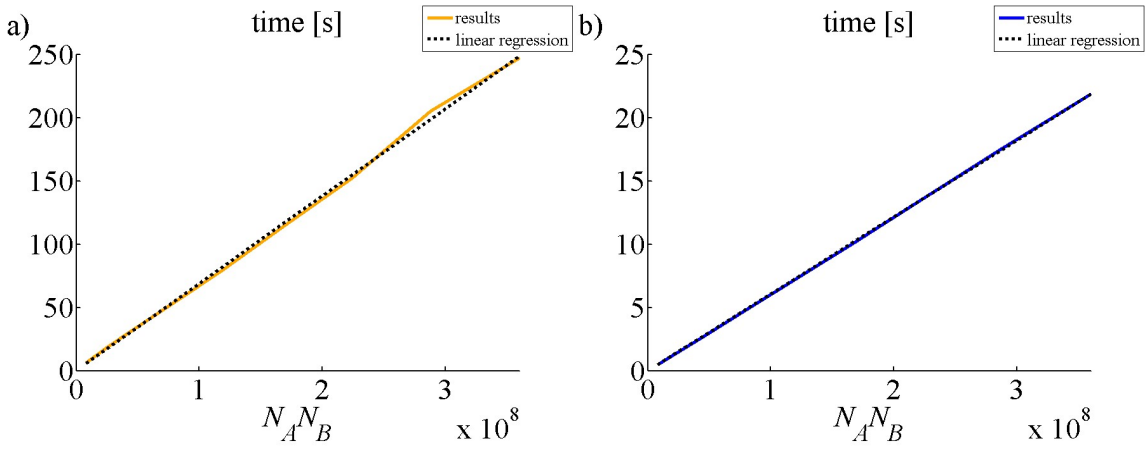

Fig.2. Comparison of the computation times for polynomial multiplications using: a) Mathematica, b) the proposed C# implementation

Rys.2. Porównanie czasów obliczeń dla mnożenia wielomianów w: a) oprogramowaniu Mathematica, b) zaproponowanej implementacji w języku C#

In both cases the computation time is nearly proportional to $N_A \cdot N_B$, hence one can deduce that in the performed trials the C# implementation is faster because of the lesser complexity of its basic operations. The observed proportionality could be due to a so-called asymptotic behavior as not only the number of multiplications influences the computation time but also the number of comparisons used for the polynomial ordering. A study of the algorithm complexity is not included in this paper, though it could give answers as to whether the algorithm proposed in the previous section could actually be advantageous to the one proposed in [14].

4. POLYNOMIAL EXPONENTIATION

4.1. Initial trials

In the C# implementation the polynomial exponentiation has been performed using sequential multiplications. This has been established as the most optimal algorithm when dealing with polynomials with coefficients defined by `double` variables. A part of the analysis leading to this conclusion is presented in the current section.

A test has been proposed dealing with the expansion of the expression:

$$g = f^k = (xy^3z^2 + x^2y^2z + xy^3z + xy^2z^2 + y^3z^2 + y^3z + \\ + 2y^2z^2 + 2xyz + y^2z + yz^2 + y^2 + 2yz + z)^k, \tag{5}$$

which has served as a benchmark in [15]. In Mathematica the above expression is by default left in a factored form and can be expanded by applying the `Expand` function. The computation time comparisons for expanding the expression in the C# implementation and in Mathematica are given in Figure 3.

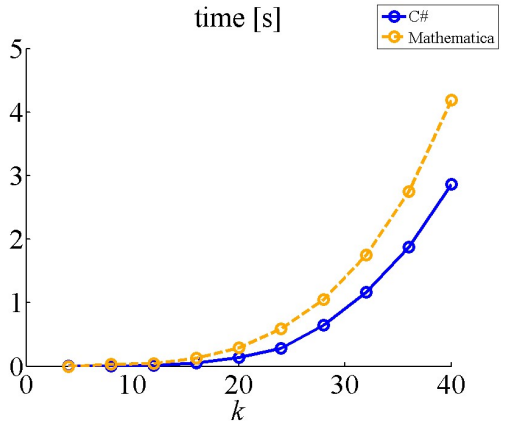

Fig.3. Comparison of the computation times for polynomial exponentiation using the proposed C# implementation and Mathematica
Rys.3. Porównanie czasów obliczeń dla potęgowania wielomianu w zaproponowanej implementacji C# i w oprogramowaniu Mathematica

The results are interesting as in all the previously compared operations (addition, subtraction, differentiation and definite integration in part II of the paper and multiplication in the previous section of the current part) the C# implementation was much more significantly faster. In the exponentiation test the C# implementation seems to only have a slight advantage. It is apparent that Mathematica has functions implemented that deal with exponentiations differently than just by applying sequential multiplications of the polynomial. The questions are therefore: first off – what could be the cause for the sudden improvement and whether it can also be applied in the C# implementation. After some thoughts and trials – the answers are given in subsection 4.3.

4.2. Discussion on algorithms

For exponentiations – Maple's researchers [15] use an algorithm (attributed to Euler), where the exponentiation of the polynomial:

$$f = \sum_{i=0}^{d} f_i x^i, \tag{6}$$

(displayed as a univariate polynomial that could be the result when applying a Kronecker map to a multivariate polynomial) resulting in:

$$g = f^k = \sum_{i=0}^{kd} g_i x^i, \tag{7}$$

is performed by applying the formula:

$$g_i = \frac{1}{if_0} \sum_{j=1}^{\min(d,i)} ((k+1)j - i) f_j g_{i-j}, \quad i = 1,...kd. \tag{8}$$

It is by far the fastest when concerning the quoted example. The paper [15] contains some useful formulae to be applied for multivariate polynomial multiplication, however – what is interesting – the most efficient, given by equations (6) to (8), requires exact rational number arithmetics. The author has attempted to implement this algorithm for floating-point numbers – some of the most important observations are:

a) for higher k values – the real-valued multipliers can contain some errors – this is also a problem for the sequential multiplications and possibly any algorithm if applying floating-point arithmetics,
b) for higher k, because of errors arising in the iterative procedure (comprising of the equations (6) to (8)), additional terms could emerge that would normally be cancelled out (this drawback has not been noticed when applying sequential multiplications),
c) attempts at introducing a tolerance, below which the resulting values are omitted (even when introducing a relative tolerance) does not solve the problem in general for higher k values (e.g. in this case when $k \geq 40$), as more complex errors emerge,
d) an introduction of more exact variable types in this procedure – e.g. by applying the `decimal` structure – allows to reduce the magnitude of the problem of additional terms, however – in the case of larger exponentiations ($k > 20$ for the considered example) an arithmetic overflow occurs (a constant solution would only be an application of a structure that dynamically changes its maximum/minimumaccepted value size),
e) for $k \leq 20$ the implemented algorithm has not been observed as advantageous in comparison to sequential multiplications (this, however, could be caused by the fact that the algorithm has only been implemented with certain initial optimizations).

The Euler algorithm implementation has also been attempted in Mathematica. However, it was very slow due to its dependence on auxiliary arrays and expressions, which have been updated in every step. All object types in Mathematica known to the author are immutable once having assigned them a value, hence every iteration in the algorithm involves the creation of new objects (which, in most cases, increase their size in each step).

4.3. Coefficient assumptions in Mathematica

The current subsection presents some observations that have been made by the author when attempting some experiments on expanding the expression in equation (5). There are some interesting results when at least one of the coefficients in the expression is changed into a number that is not an integer value. In this case it seems to operate in a completely different manner drastically increasing the computation time (Figure 4). In this test the monomial $2xyz$ has simply been changed to $2.1xyz$. Figure 5, on the other hand, presents results for when this monomial has been modified to $\frac{21}{10}xyz$.

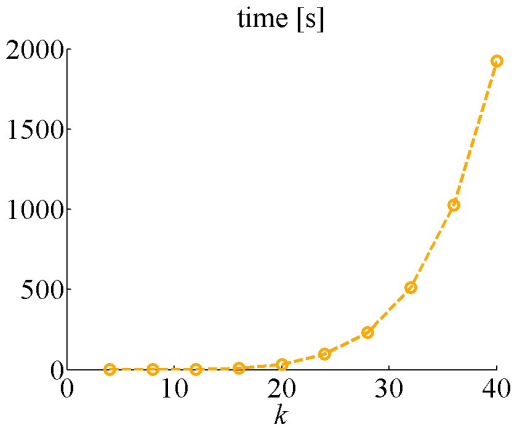

Fig.4. Comparison of the computation times for polynomial exponentiation given by equation (1) in Mathematica when a floating-point valued coefficient was present in the polynomial *f*

Rys.4. Porównanie czasów obliczeń dla potęgowania podanego w równaniu (1) w oprogramowaniu Mathematica, gdy wielomian *f* zawierał współczynnik zdefiniowany poprzez liczbę zmiennoprzecinkową

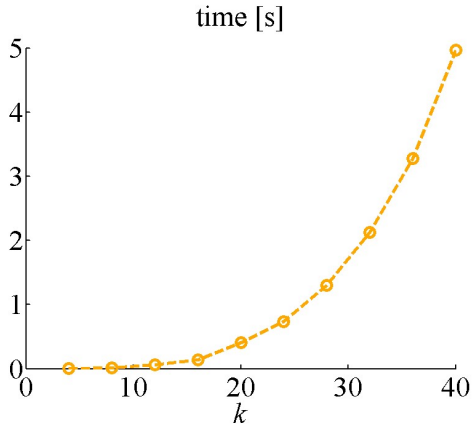

Fig.5. Comparison of the computation times for polynomial exponentiation given by equation (1) in Mathematica when a rational valued coefficient was present in the polynomial *f*

Rys.5. Porównanie czasów obliczeń dla potęgowania podanego w równaniu (1) w oprogramowaniu Mathematica, gdy wielomian *f* zawierał współczynnik zdefiniowany poprzez liczbę wymierną

The results depict that the exponentiation algorithm in Mathematica is very efficient for coefficients that are generally rational numbers. The reduction of efficiency in the floating-point case could be due to Mathematica dynamically adapting the accuracy of the coefficients in floating-point form to avoid loss of information.

5. CONCLUSIONS

An original, easy to implement algorithm for sparse polynomial multiplication has been proposed. The paper provides an analysis, which shows that the algorithm uses only exactly $(N_A - 1)\cdot(N_B - 1)$ comparisons between the symbolic multipliers of monomial terms. The computation times for a given example have been studied and it was shown that they were nearly proportional to $N_A N_B$. A complexity analysis was, however, not provided in the paper so it was not generally proven that the algorithm is of $O(N_A N_B)$ complexity – as then not only comparisons need to be taken into account but also multiplications and e.g. comparisons between auxiliary variables if they are to also have a greater effect on the computation time.

A discussion was presented concerning the polynomial exponentiation. It was concluded that the best option, when concerning polynomials with constant floating-point arithmetics (e.g. using the `double` variable for this purpose) was to use sequential multiplications. An algorithm has been presented, which was shown to be one of the fastest available for polynomial exponentiations – however it was inapplicable for the C# implementation so far operating on the assumptions given in part I of this paper.

A further improvement of the C# implementation in the case of exponentiations could be made if the coefficients could be defined as rational values. This is however not in the author's near-future plans as in his applications the symbolic computations are combined with approximation procedures that yield floating-point values [16].

In Mathematica – polynomial exponentiation is the fastest when dealing with coefficients defined as rational numbers. When the coefficients were proposed as floating-point values – the operation was approximately around 400 times slower.

BIBLIOGRAPHY

1. Ponder C.G.: Parallel multiplication and powering of polynomials. "Journal of Symbolic Computation" 1991, 11, s.307-320.
2. van der Hoeven J., Lecerf G.: On the bit-complexity of sparse polynomial and series multiplication."Journal of Symbolic Computation" 2013, 50, s.227-254.

3. Fateman R.: Comparing the speed of programs for sparse polynomial multiplication. “ACM SIGSAM Bulletin” 2003, 37 (1), s.4-15.
4. Harvey D.: A cache-friendly truncated FFT. “Theoretical Computer Science” 2009, 410 (27), s.2649-2658.
5. Cenk M., Özbudak F.: Multiplication of polynomials modulo x^n. “Theoretical Computer Science” 2011, 412 (29), s.3451-3462.
6. Kronecker L.: Grundzüge einer arithmetischen Theorie der algebraischen Grössen. 1882, 92, s.1-122.
7. Pan V.Y.: Simple multivariate polynomial multiplication. “Journal of Symbolic Computation” 1994, 18 (3), s.183-186.
8. Jia Y.-B.: Polynomial Multiplication and Fast Fourier Transform. “Com S 477/577 Notes” 2014.
9. Gentleman W.M., Sande G.: Fast fourier transforms – for fun and profit. “Proc. AFIPS 1966 FJCC 29”, 1966, s.563-578.
10. van der Hoeven J.: Notes on the truncated fourier transform. “Tech.Rep. 2005-5”, 2005.
11. Monagan M., Pearce R.: Parallel sparse polynomial multiplication using heaps. “Proceedings of the 2009 international symposium on symbolic and algebraic computation. ACM”, 2009, s.263-270.
12. Gastineau M., Laskar J.: Development of TRIP: Fast sparse multivariate polynomial multiplication using burst tries. “Computational Science-ICCS” 2006, s.446-453.
13. Roche D.S.: Chunky and equal-spaced polynomial multiplication. “Journal of Symbolic Computation” 2011, 46 (7), 791-806.
14. Johnson S.C.: Sparse polynomial arithmetic. “ACM SIGSAM Bulletin” 1974, 8(3), s.63-71.
15. Monagan M., Pearce R.: Sparse polynomial powering using heaps. “Computer Algebra in Scientific Computing”, Springer, 2012, s.236-247.
16. Sowa M., Typańska D.: Pre-assembly for FEM 2D non-curvilinear quadrilateral Lagrangian elements. “Computer Applications in Electrical Engineering” 2014, Vol.12, s.106-119.

Dr inż. Marcin Sowa
Silesian University of Technology
Faculty of Electrical Engineering
Institute of Electrical Engineering and Computer Science
ul. Akademicka 10
44-100, Gliwice
e-mail: Marcin.Sowa@polsl.pl