ELEKTRYKA 2016
Zeszyt 1 (237) Rok LXII

Marcin SOWA
Silesian University of Technology, Gliwice

SIMPLE C# CLASSES FOR FAST MULTIVARIATE POLYNOMIAL SYMBOLIC COMPUTATION – PART I: IMPLEMENTATION DETAILS

Summary. The paper deals with a lightweight C# implementation that allows to perform symbolic computation on multivariate polynomials in expanded forms. The classes called `SPoly` and `SMono` (which represent the expressions) are explained along with their limitations. Furthermore a few remarks are made on the usefulness of methods working as += and −= operators. A further analysis of the implementation is given in parts II and III of the paper.

Keywords: symbolic computation, sparse multivariate polynomials, computation time, C# implementation

PROSTE KLASY W JĘZYKU C# DO OBLICZEŃ SYMBOLICZNYCH NA WIELOMIANACH WIELU ZMIENNYCH – CZĘŚĆ I: SZCZEGÓŁY IMPLEMENTACJI

Streszczenie. W artykule opisano prostą implementację (w języku C#) możliwości wykonywania obliczeń symbolicznych na wielomianach wielu zmiennych. Klasy opisujące wyrażenia symboliczne (zwane `SPoly` i `SMono`) zostały objaśnione. W publikacji również wymieniono ich ograniczenia. Ponadto wypunktowano kilka korzyści jakie przynosi zastosowanie metod działających jako operatory += i −=. Dalszą analizę implementacji zaprezentowano w częściach II i III niniejszego artykułu.

Słowa kluczowe: obliczenia symboliczne, wielomiany rzadkie wielu zmiennych, czas obliczeń, implementacja C#

1. INTRODUCTION

In the practice of computational problems, arising from modeling tasks of engineering branches, it is an often case that it is convenient (and also often more efficient on the long run) to obtain dependencies in symbolic forms. The ability of a computer program to perform mathematical operations on formulae in a symbolic representation is widely known as

symbolic computation. There are many programs available that allow these types of computations – they are referred to as CAS (Computer Algebra Systems). The most known are Maple and Mathematica. These programs are able to perform symbolic computations on a wide variety of expression types and provide various options concerning their presentation (e.g. in expanded or factored forms).

Specialists most commonly reach out for the well-known CAS treating them as standalone programs to solve their problems. As for programmers' needs (when symbolic computation is required) CAS are also used – this usually involves a link being established between the main program and the Computer Algebra System. It is an often case however – that when dealing with engineering problems (like in the author's experience in computational electromagnetics [1]) symbolic computation is only required for some specific expression classes and operations. In these cases a simple implementation of the ability to perform symbolic computations is sufficient and could provide a few benefits. The presented research concerns such an implementation in C# for multivariate polynomials in expanded forms.

The author regularly uses the library in computational electromagnetics and simpler tasks such as interpolation or studies in theoretical electrical engineering. It will be made available for free to anyone who needs the proposed limited (yet useful) functionality for symbolic computations on multivariate polynomials. More comprehensive CAS may also be built with the use of the proposed classes in the future. Because the implementation is introduced into a programming language – many other benefits are gained, some of which are:

- no need to install and execute additional linking processes between a computer algebra system and a lower level programming language (i.a. as discussed in [2]), especially if only multivariate polynomials with positive integer exponents are concerned (in this case only a limited CAS functionality is required),
- a direct execution of symbolic computations and convenient access to the results,
- no need for additional objects that would store the results and allow them to be transferred through auxiliary methods,
- a pre-compiled code generally outperforms one that is interpreted at run-time (which is the case for all CAS) in terms of computation speed (however, there are efforts to at least make basic, built in operations in CAS much faster, as is the case with Maple's implementations in [3]).

The proposed basic implementation could potentially also provide valuable insight as to what could be improved in terms of the numerical efficiency for the multivariate polynomial computations of computer algebra systems. The paper also provides general educational value to anyone who wants to write their own symbolic computation libraries. In this paper the numerical efficiency is understood by the author as an ability to perform symbolic computations in a reasonable period of time (judging by the problem complexity) where it is

also the case that the obtained expressions do not require relatively excessive amounts of memory for their storage.

The sections of the paper are divided as follows:

- section 2 gives a brief explanation of the classes' functionality,
- in section 3 the essential classes, representing multivariate polynomials, are presented,
- section 4 contains details on the implemented operators of addition and subtraction,
- section 5 deals with the ability of analytical differentiation and definite integration of the expanded symbolic expressions,
- section 6 provides a summary of the work completed so far.

In part II of the paper – an analysis of numerical efficiency is presented (for all the implemented operations of this part).

Multivariate polynomial multiplication and exponentiation is discussed separately – in part III of the paper (along with an analysis of computation times).

2. SUPPORTED SYMBOLIC EXPRESSIONS AND FUNCTIONALITY

It is assumed that the symbolic expressions must satisfy the expanded formula for multivariate polynomials:

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

where $\boldsymbol{s}$ represents a vector containing all symbolic variables s_1, s_2, s_3, …, s_n while $T_i(\boldsymbol{s})$ represents the subsequent terms of the symbolic expression. The product $\prod_{j=1}^{n} s_j^{\,k_{i,j}}$ is later on referred to as the symbolic multiplier, while a_i is called the real-valued multiplier.

At the current stage – efforts have been made so that the functionality of the implementation covers:

a) a convenient generation of symbolic expressions by giving the multipliers a_i and exponentiations $k_{i,j}$; alternatively a string, which is interpreted and converted into a set of objects,
b) addition, subtraction and multiplication operations concerning expressions of the form (1),
c) exponentiation of a polynomial by a positive integer,
d) the possibility of obtaining the derivative $\partial\zeta/\partial s_l$ in terms of a selected symbolic variable s_l,
e) the ability to analytically evaluate the definite integral:

$$I_{\text{def}} = \int_{c(\boldsymbol{s})}^{d(\boldsymbol{s})} \zeta(\boldsymbol{s})\,\mathrm{d}s_l\,, \tag{2}$$

for a selected interval from $s_l = c(\boldsymbol{s})$ to $s_l = d(\boldsymbol{s})$ (hence also – symbolic variable substitution must be considered).

3. MAIN CLASSES

It was in the author's interest to make the implementation simple, while also keeping it potentially efficient. Therefore it requires the standard C# libraries alone, without additional composite packages and classes.

Polynomials of the form (1) are expressed by objects that are instances of the `SPoly` class. This class provides a root that contains methods allowing for the operations pointed out in the previous section. It contains references to only some of the monomials. The details on subsequent monomials $T_i(\boldsymbol{s})$ will, however, be contained inside a unidirectional linked list of objects that are instances of a separate class called `SMono`. The dependencies between an `SPoly` object and the subordinate `SMono` object list are displayed in Figure 1.

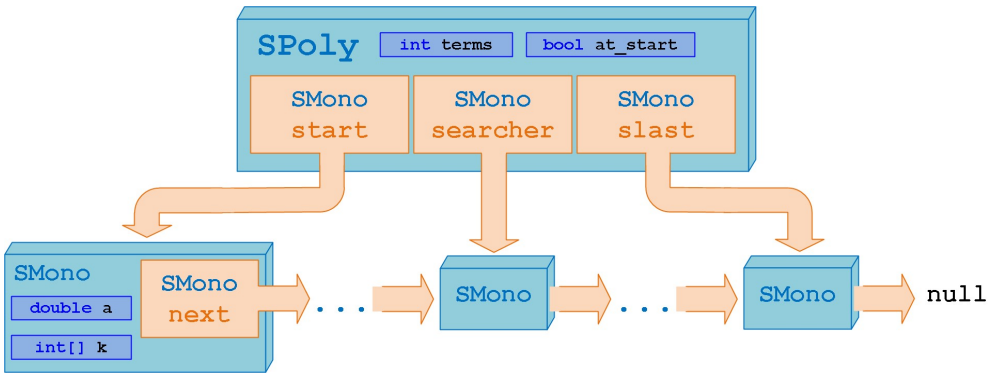

Fig.1. Dependencies between the proposed symbolic math objects
Rys.1. Zależności pomiędzy zaproponowanymi obiektami do obliczeń symbolicznych

The `start` and `slast` references of an `SPoly` object represent respectively the first and last object of the linked list. start and `at_start` allow to conveniently begin a search for common symbolic multipliers, while `slast` helps in monomial appending. The `searcher` reference is used in order to mark the monomial that was compared as the last one – this is useful for polynomial addition and subtraction.

Each `SMono` object contains information about a separate monomial, where the double variable `a` represents the real multiplier a, while `k` is an array containing the exponentiations of each symbolic variable (i.e. the integers $k_{i,1}, k_{i,2}, \ldots, k_{i,n}$ of a monomial with the index i).

In order to reduce the amount of monomial comparisons when addition or subtraction is performed – polynomial ordering needs to be applied. Out of the many possible methods used for multivariate polynomials [4] – a type of lexicographical ordering is used in the implementation (where the exponent of s_n is the most significant and of s_1 is the least significant). Therefore, when polynomials are added, the `searcher` reference allows to skip terms that are surely placed earlier than the one added (as they precede a previously added monomial).

The simple monomial search method, which also informs what should be done when the search is complete, is presented in Figure 2.

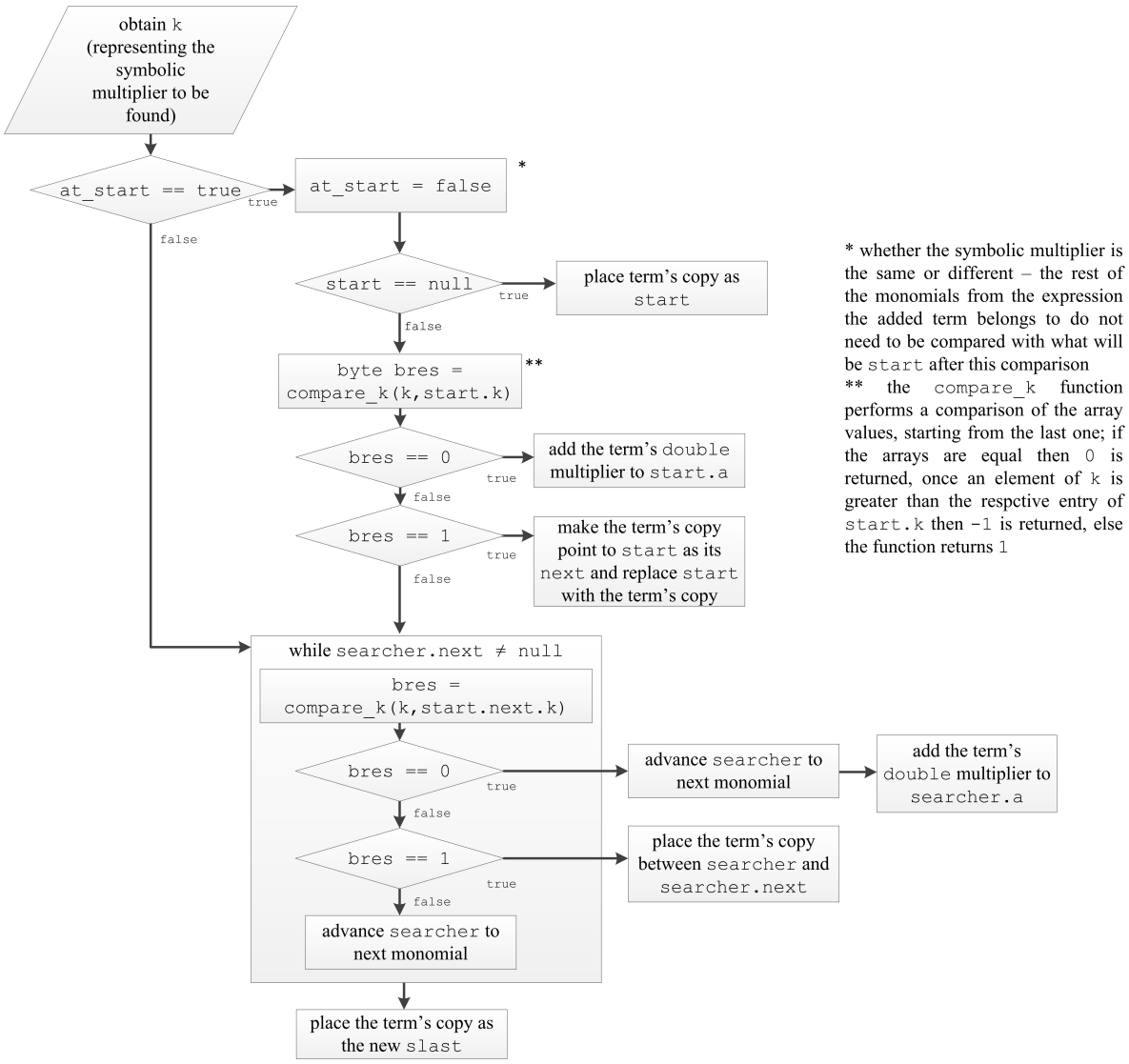

Fig.2. The algorithm for the search of a common symbolic multiplier
Rys.2. Algorytm poszukiwania wspólnego mnożnika symbolicznego

4. ADDITION AND SUBTRACTION OF SYMBOLIC EXPRESSIONS IN C#

4.1. Implemented addition/subtraction operators

Subject to the assumptions given in section 2 – the addition, subtraction and multiplication operators have been implemented. This section concerns the first two. For the implementation to be efficient – methods have been built so that one can perform += and −= operations more efficiently (allowing to overwrite the polynomial, which another one is being added to); these are referred to, in the paper, as `(op)=` operations. The C# language does, however, not allow the overload of the regular `(op)=` operators, so instead functions called `pluseq` and `mineq` have been written.

An idea is also introduced so that one can use alternative (even more efficient) methods if it is admissible to overwrite both objects, which take part in the operation. These are later on referred to as the overtaking operators.

The operators and methods that have been implemented are presented in Table 1.

Table 1.

Addition and subtraction operators (and methods) between `SPoly` objects

	Regular	**Fast `(op)=`**	**Overtaking**
Addition	`c=a+b` `a+=b`	`a.pluseq(b)`	`plus_combine(a,b)`
Subtraction	`c=a-b` `a-=b`	`a.mineq(b)`	`minus_combine(a,b)`
Comments on addition and subtraction operators and methods	`(op)=` operators build a new object for `a`	the operations overwrite `a`, results are the same as in regular `(op)=` operators	use `a` and `b` to build `a+b` (or `a-b`) without acquiring new memory; `a` is overwritten, `b` is useless after these operations (as its `SMono` objects are used in the result); the methods return the result but it is also stored in `a`

4.2. Comparison of addition/subtraction operation performance

A comparison is performed in terms of the computation speed for the symbolic math classes when using:

- the regular operators += and −=,
- the fast `(op)=` methods `pluseq` and `mineq`,
- the overtaking operations `plus_combine` and `minus_combine`,

The first procedure being tested is one where a continuous addition/subtraction of terms (with repeating symbolic multipliers) is performed. This procedure follows the pseudo-code:

```
S=0; i=0; m=round(n/5);
For j=1 to n:
        w = (j+1) * a^i * b^(m-i) * c^i;
        S+=w; OR S-=w;
        i++;
        If i>m:
                i=0;
        End
End
```

The results (in terms of computation time) for selected addition and subtraction operators/methods are presented in Figure 3. The results for regular `(op)=` computations are not displayed since their duration was hundred to thousand times longer (this is because such continuous addition/subtraction of terms requires the object to be rebuilt each time).

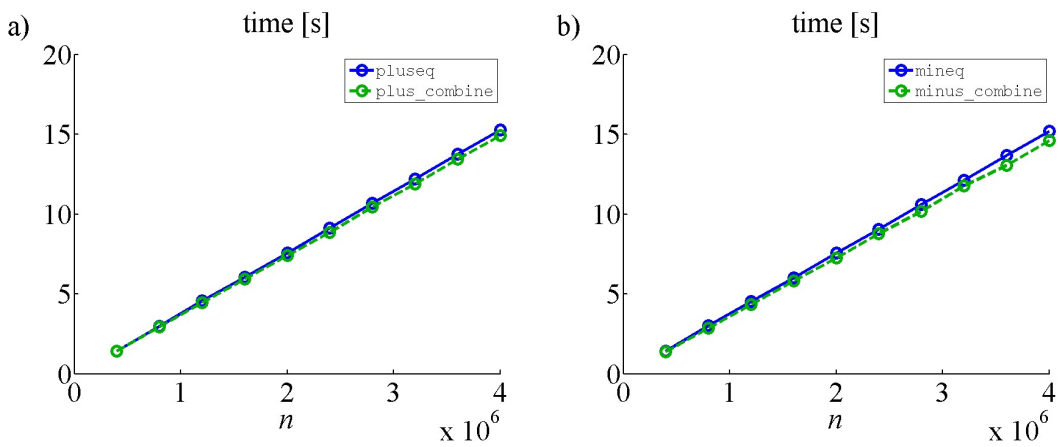

Fig.3. Comparison of the computation times for various: a) addition methods, b) subtraction methods
Rys.3. Porównanie czasów obliczeń dla różnych: a) metod dodawania, b) metod odejmowania

A different task for the operators is the addition/subtraction of polynomials consisting of more than one term. The results from the trials of the previous test are placed as subsequent entries of an array `S` and then new trials are performed, where the neighboring entries are added together e.g. for the second trial the operation `S[1](op)=S[2]` is performed. The results are presented in Figure 4.

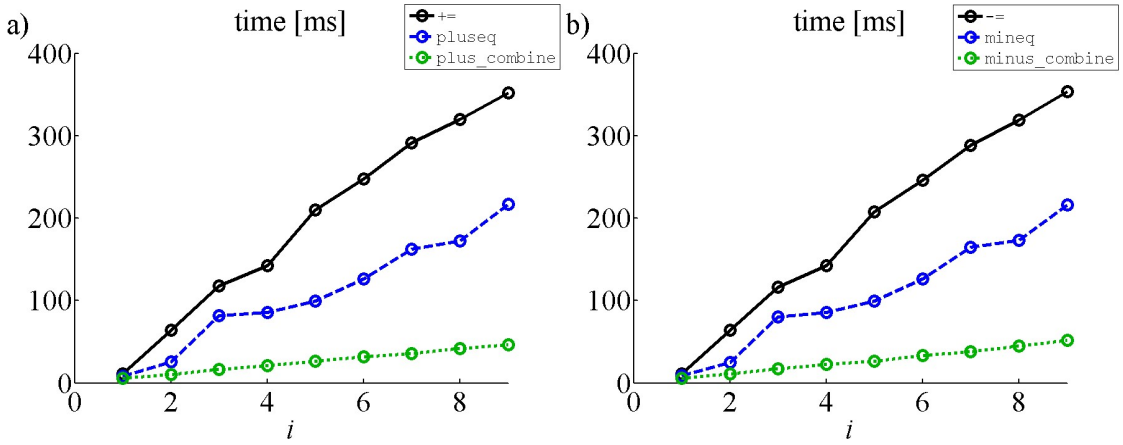

Fig.4. Comparison of the computation times for various: a) addition methods, b) subtraction methods
Rys.4. Porównanie czasów obliczeń dla różnych: a) metod dodawania, b) metod odejmowania

Because the results for both the addition and subtraction operations are very close – the analysis observations are the same:

I) the `pluseq` and `mineq` methods have been thousands of times faster for frequent (loop) additions and about twice as fast as the regular `(op)=` operators for regular (single expression) additions and subtractions,

II) the expression emerging as a result of the operation is the same in every case,

III) this resulting expression requires the same amount of memory in every operation, however – for the overtaking operators the terms of the partaking objects are used, hence only terms with repeating symbolic multipliers are not taken over by the result; the overtaking operations can be useful when dealing with expressions that require large amounts of memory,

IV) for the looped-operation test, the overtaking methods did not display significant advantages (faster by about 4 to 5 percent), while for the second test the computation time was 4 to 7 times shorter than for the += and −= operators.

In conclusion – the fast operation methods `pluseq` and `mineq` should be used instead of += and −=, especially for frequent additions/subtractions. If two long expressions are added/subtracted (and not used later on), in order to save memory, one can apply the `plus_combine` and `minus_combine` methods.

With all of its advantages – unfortunately the presented concept of overwriting operations has a drawback, since it requires objects to be available for modification. This opposes the idea of object immutability. Lately immutability is strongly motivated by trends in computer architecture i.e. allowing for a several times increased computation speed through parallelization [5]. This issue has not yet been resolved in the discussed research.

5. DIFFERENTIATION AND DEFINITE INTEGRATION

A method that returns the derivative, with respect to a selected variable s_l, can apply the well-known formula for multivariate polynomials:

$$\frac{\partial \zeta(\boldsymbol{s})}{\partial s_l} = \sum_{i=1}^{N} \frac{\partial T_i(\boldsymbol{s})}{\partial s_l} = \sum_{i=1, k_{i,l} \neq 0}^{N} k_{i,l} a_i s_l^{k_{i,l}-1} \prod_{j=1, j \neq i}^{n} s_j^{k_{i,j}}. \tag{3}$$

According to the above – a derivative along s_l is simply computed by adding the sum's consecutive terms in which $k_{i,l} \neq 0$. One can notice that the ordering is retained as each monomial's exponentiations change with the same degree. Hence, actually, the terms can be appended using the `slast` reference.

The computation of the definite integral is slightly more difficult (however – initially it follows an equally obvious formula, what is displayed later on). Its steps could be divided into, firstly, a symbolic evaluation of the indefinite integral and, secondly, substitutions taking into account some boundaries $s_l = c(\boldsymbol{s})$ and $s_l = d(\boldsymbol{s})$.

The auxiliary polynomial I is computed according to the formula:

$$I = \int \zeta(\boldsymbol{s}) \mathrm{d}s_l - C = \sum_{i=1}^{N} \frac{a_i}{k_{i,l}+1} s_l^{k_{i,l}+1} \prod_{j=1, j \neq i}^{n} s_j^{k_{i,j}}, \tag{4}$$

where C is the integration constant, which does not need to be considered since it is subtracted later on after substitutions. Up to this moment the ordering is retained as in the case of differentiation. However, it is no longer the case when substitutions are made. One can distinguish two types of $s_l = \vartheta$ substitutions. The first is where ϑ is a constant, while the more difficult case is where it is generally another polynomial. The definite integral:

$$I_{\mathrm{def}} = \int_{c(\boldsymbol{s})}^{d(\boldsymbol{s})} \zeta(\boldsymbol{s}) \mathrm{d}s_l = I\big|_{s_l = d(\boldsymbol{s})} - I\big|_{s_l = c(\boldsymbol{s})}, \tag{5}$$

is obtained through an application of the following procedures:

1) judging on whether c and d are numerical values or polynomials, the appropriate method overload is executed:

a) if both values are numerical then the following auxiliary formula is used:

$$\int_{c(\boldsymbol{s})}^{d(\boldsymbol{s})} \zeta(\boldsymbol{s}) \mathrm{d}s_l = \sum_{i=1}^{N} \frac{a_i}{k_{i,l}+1} (d^{k_{i,l}+1} - c^{k_{i,l}+1}) \prod_{j=1, j \neq i}^{n} s_j^{k_{i,j}}. \tag{6}$$

b) if at least one is a symbolic expression then first the polynomial I is obtained by means of the formula (4),

c) if only one of the values is numerical then for that case the substitution is made for each monomial, which is then added to a resulting polynomial, while maintaining a proper ordering (this is actually done with methods that the `plus_combine` operation also executes),

d) for each polynomial substitution $s_l = \vartheta$:
   - an auxiliary array of `SPoly` objects (called `K` further on) is obtained and filled by results of exponentiations that appear subject to the substitution,
   - for each monomial the substitution is made by applying a proper entry of `K` (note that repeated exponentiations do not need to be made),
   - the obtained polynomials are added with `plus_combine`;

2) the subtraction of equation (5) can now be made.

6. SUMMARY

A simple, lightweight C# implementation has been presented, which allows to perform symbolic computations on multivariate polynomials. It mainly bases on the `SPoly` and `SMono` classes, whose instances represent general information on the polynomial and details on each monomial term respectively. An algorithm has been presented for searches of monomials that can be added up forming one (this was used in every type of addition and subtraction method later on).

Two types of additional methods have been introduced allowing for faster += and −= operations. The first type (represented by the `pluseq` and `mineq` methods) allowed for object overwriting, which resulted in a reduced computation time. The second type (referred to as the overtaking operators) leads to a further reduction of computation time and allows to save memory if dealing with large expressions (the partaking objects are, however, useless after this operation, hence these methods are only suggested in critical cases).

Simple procedures have also been implemented to deal with differentiation and definite integration.

In part II of this paper – the operations presented in the current part are subjected to tests and the results are discussed in terms of the numerical efficiency.

The author has paid particular attention to the implementation of polynomial multiplication, hence it is discussed separately in part III of this paper.

BIBLIOGRAPHY

1. Sowa M., Typańska D.: Pre-assembly of FEM n-th order triangular elements. “Poznań University of Technology Academic Journals, Electrical Engineering” 2014, 77, s.133-139.

2. Bauer C., Frink A., Kreckel R.: Introduction to the GiNaC framework for symbolic computation within the C++ language. “Journal of Symbolic Computation” 2002, 60, s.120-136.
3. Monagan M., Pearce R: POLY: A new polynomial data structure for Maple 17. “ACM Communications in Computer Algebra” 2013, 46 (3/4), s.164-167.
4. Geddes K.O., Labahn G.: Algorithms for Computer Algebra. Springer, 1992.
5. Gordon C.S., Parkinson M.J., Parsons J., Bromfield A., Duffy J.: Uniqueness and reference immutability for safe parallelism. “SIGPLAN Notices” 2012, 47 (10), s.21-40.

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