CUSTOM CODE OPTIMISATION OF SELECTED NUMERIC CALCULATIONS IN VIRTUAL MACHINES FOR JIT COMPILERS

Summary. The optimisation of calculations in selected virtual machines is described and non-conventional performance optimisation techniques for selected numerical expressions are proposed. The first presents mantissa expression transformation, while the second presents degrees to radians conversion. Considerations are illustrated by code and tests carried out in Java, MS.NET and Mono environments.

Keywords: component platforms, just-in-time compilation, compilers, optimisation

NIESTANDARDOWE TECHNIKI OPTYMALIZACJI WYBRANYCH ZAGADNIĘĆ NUMERYCZNYCH DLA KOMPONENTOWYCH MASZYN WIRTUALNYCH


Słowa kluczowe: platforma komponentowa, JIT, kompilatory, optymalizacja

---

1This work was realised as a contribution to statutory research of the Department of Geoinformatics and Applied Computer Science AGH.
1. Introduction

Component software engineering is a modern software production technique. Available technologies that allow the creation of components are, for example, Enterprise Java Beans [19], Spring [20], niche PicoContainer [21], and Google Guice [22]. Current component software development trends are based on virtual machine technology, whose main advantage is hardware independence. This implies portability as numerous numerical algorithms can be computed on various platforms. In addition, programmers can deploy algorithms that they have not created and the reusability of code makes it possible for coders to cooperate. There are many numerical solutions that use component software, some of which are time-consuming, such as seismic wave field modelling [8] and heat transfer modelling [9].

2. Component virtual machines

There are a few solutions for this domain, the most advanced and popular of which are Java Virtual Machine (JVM) [17] developed by Oracle [11] and Microsoft .NET Framework [10]. This Microsoft product is a commercial platform, but there is also an alternative open source solution, currently developed by Xamarin, called Mono [16] that can run .NET applications on operating systems like Linux or Mac OS X.

The main advantage of component environments is a feature called bytecode or managed code [18]. This is special precompiled code executed under the management of a virtual machine: bytecode in a Java Virtual Machine environment and managed code under the control of .NET Framework or Mono. Therefore, bytecode and managed code are portable and hardware independent. It might seem that the process of translating on-the-fly intermediate code to native processor code (Just-in-Time compiling) is time-consuming, but modern compilers and virtual machines can effectively optimise execution [6, 7]. Some optimisation techniques contribute to increased calculation efficiency.

3. Optimisation Techniques

Optimisation is the transformation of original code to make the resulting code faster, more compact, or less memory consuming [1]. The code optimisation methods presented in this article apply to execution speed, as this is the main numerical calculation problem. Virtual machines optimise code during its execution (Just-in-Time) [3]. One of the most time-consuming aspects of algorithms is the processing of loops. There are few well-known opti-
misation techniques in this scope, such as code motion, but virtual machines do not use it satisfactorily [13].

To achieve the best effect, source code should be optimised by programmers. The Mono platform implements common algorithms like peephole post pass, branch optimisations, inline method calls, constant folding, constant propagation, copy propagation, dead code elimination, linear scan global registry allocation, conditional moves, emit per-domain code, instruction scheduling, intrinsic method implementations, tail recursion and tail calls, loop related and leaf procedure optimisations [16], but they do not provide all capabilities to optimise code, although in some cases there is the ability to suggest solutions.

Custom techniques for code optimisation are presented in [14]. A simple DSP function that calculates the average and instantaneous power of the values in a complex value array was examined. In this work, the author’s solutions for improving the efficiency of numerical operations are described. The focus was on the transformations and rearrangements of the code, especially in the loops.

The problem of efficient numeric operations involving sequences of operations is described in [4]. Expression graphs were used as an analysis method. Time-consuming algorithms use a lot of data in matrices. The problem of optimal access to this data, including bound checking optimisation, is shown in [2].

The issue of a low overhead path-profiling scheme for dynamically profiling JIT produced native code is discussed in [15]. In this paper, profile information was used in recompilation during a subsequent invocation of the hot method for a Just-in-Time Java compiler.

Loops can also be optimised as some contain expressions that only need evaluating once. Code analysis allows custom optimisation.

All optimisations performed by compilers have a common quality in that they do not influence the result generated by a program, as code after optimisation works in exactly the same way as before. However, if this restriction can sometimes be overcome to a limited acceptable degree, in some cases the advantages of optimisation can be much greater than using standard methods. Such non-standard optimisation methods are the focus of this paper. The next sections present two cases in which non-conventional optimisation can be applied.

4. Proposed optimisation algorithms

Some numerical problems use common mathematical expressions like mantissa or metric angle unit conversion from degrees to radians. Below, two ways of accelerating these operations are presented.
4.1. Mantissa

The mantissa operation is used to calculate a fractional part of a real number. It is given by the formula 1.

\[ y = x - \lfloor x \rfloor, \]

To calculate it using Java, the following expression is executed:

\[ y = x - \text{Math.floor}(x) \]

The code above calculates the `floor` function using the standard mathematical library. All presented virtual machines have this function in the core programming environment. JVM uses the `java.lang.Math` mathematical library, while .NET Framework and Mono use the `System.Math` class. The aforementioned mathematical libraries provide high-performance implementation [10, 11]; however, with certain restrictions the calculation of mantissa can be performed more efficiently. The proposed non-conventional optimisation in Java is presented as full and compact versions in listings 1.1 and 1.2.

```java
if (x < 0 && x % 1 != 0)
    y = x - (int) x + 1;
else
    y = x - (int) x;
```

Listing 1.1: Mantissa optimisation (Java)

```java
y = (x < 0 && x % 1 != 0) ? (x - (int) x + 1) : (x - (int) x);
```

Listing 1.2: Mantissa optimisation – compact version (Java)

In both code versions, a `%` sign was used as division operator. The `modf` function would probably be more efficient, but none of x87 FPUs is able to handle this command. As shown in the performed tests, the presented mantissa calculation techniques (listings 1.1 and 1.2) are more efficient because they replace the floor function with type casting. The proposed optimisation for C# is almost identical, the difference being in the first letter of the floor function name.

The proposed optimisation has two disadvantages:

1. The input variable is a double, but the correct result is generated only if the input value is an integer. This restriction is associated with casting types from double to integer.
2. The code is hard to read and understand. Clean and self-documenting code is very important for modern software development techniques based on agile programming methodology. The proposed optimisation completely disrupts this property.

The optimisation can be used in the proposed form if awareness of the disadvantages is maintained. However, the aforementioned problems can be solved using a small calling method overhead. The solution could be:
1. use a standard mantissa calculating method for numbers outside the range of integer numbers;
2. encapsulate calculation methodology in a separate class

An example of a Java class which could be used to calculate mantissa is presented in listing 1.3.

```java
public class MathUtil {
    public static double mantissa(double param) {
        if (param > Integer.MAX_VALUE || param < Integer.MIN_VALUE)
            return mantissaOpt(param);
        else
            return mantissaOrg(param);
    }

    private static double mantissaOrg(double param) {
        return param - Math.floor(param);
    }

    private static double mantissaOpt(double param) {
        if (param < 0 && param % 1 != 0)
            return param - (int) param + 1;
        else
            return param - (int) param;
    }
}
```

Listing 1.3: Example of a class for high performance calculations of mantissa (Java implementation)

The usage of the presented mantissa calculation implementation is very simple:

```java
y = MathUtil.mantissa(x);
```

The calculation always gives correct results. This Mantissa calculation technique is better than using the standard method for integers, but the overhead caused by the calling method is significant. A simple test was carried out in order to measure the performance gain. The test was performed in an E2 environment whose specification is described in section 5. Using the class presented in listing 1.3, the mantissa calculation is more than 37% faster for integers. As shown in table 1, mantissa calculation for non-integers took about 19% longer than the standard method. Results of the tests carried out in section 5 show that the performance gain of the presented optimisation could be much better without encapsulation.

4.2. Metric angle unit conversion from degrees to radians

Degrees to radians conversion is often used to calculate positions of points after rotation (in 2D or 3D) when angles are given in degrees. It is defined by formula 2:

\[
\text{radians} = \frac{(2\pi \times \text{degrees})}{360}
\]

The following expression is executed to calculate it using Java:
\[
\text{rad} = ((\text{deg} * 2.0 * \text{Math.PI}) / 360.0);
\]

This operation can be done in another way with a changed order of operations:

\[
\text{rad} = (\text{deg} * ((2.0 * \text{Math.PI}) / 360.0));
\]

There seem to be no differences between the presented expressions, but the change is significant from the standpoint of the compiler. Compilers perform an optimisation technique called constants folding in which operations on constant values are calculated once while compiling the source code to an intermediate representation. Using this fact, the first expression after compiling looks like:

\[
\text{rad} = ((\text{deg} * 2.0 * 3.141592653589793d) / 360.0);
\]

The second operation is faster, because the output code contains only a single multiplication:

\[
\text{rad} = \text{deg} * 1.7453292519943295e-002;
\]

Listings 1.4 and 1.6 show original algorithms in bytecode and CIL representation while listings 1.5 and 1.7 show optimised algorithms. Without focusing on the details of the intermediate code, it is easy to see that the code after optimisation is shorter, simpler, and contains the calculated result of the constants multiplication; therefore, it is more efficient.

```
8: ldc2_w #18; //double 2.0d
11: dmul
12: ldc2_w #20; //double 3.141592653589793d
15: dmul
16: ldc2_w #22; //double 360.0d
19: ddiv
```

Listing 1.4: Original algorithm of metric angle unit conversion – Bytecode for Java

```
IL_0016: ldc.r8 2.
IL_001f: mul IL_0020: ldc.r8 3.1415926535897931
IL_0029: mul IL_002a: ldc.r8 360.
IL_0033: div
```

Listing 1.5: Proposed algorithm of metric angle unit conversion – Bytecode for Java

```
IL_0016: ldc.r8 1.7453292519943295e-002
IL_001f: mul
```

Listing 1.6: Original algorithm of metric angle unit conversion – CIL code for .NET and Mono

```
IL_0016: ldc.r8 1.7453292519943295e-002
```

Listing 1.7: Proposed algorithm of metric angle unit conversion – CIL code for .NET and Mono

```
During compilation, the optimisers in most compilers perform operations on numbers and insert the results in the output code. This does not apply to intermediate representation of code (for Java, C#). It is worth mentioning that division is the most time consuming arithme-
tic operation, as it can be over ten times slower than multiplication and twenty times slower than addition, taking into account modern x86 architectures. The disadvantage of the presented method is the possible loss of precision for small real numbers that is caused by division before multiplication of all factors. The proposed method offers faster code execution, but the programmer should check for precision errors and estimate the error value as required.

5. Tests

Tests were carried out to ascertain the efficiency of the proposed algorithms to check precision errors.

5.1. Hardware specification

The experimental tests of the efficiency of the proposed algorithms were performed in two different environments.

The first environment (E1) was a notebook with an Intel Pentium M 1.5 GHz CPU and 1024 MB DDR PC2100 (133 MHz) RAM. The second environment (E2) was a PC computer with a Pentium(R) Dual-Core CPU E6500 @ 2.93 GHz and 2 GB DDR2-667 (333 MHz) RAM.

5.2. Operating systems and software specification

The performance of the proposed algorithms was measured in the following operating systems: Linux Ubuntu 10.04 and MS Windows XP (SP3).

The following component environments were installed: .NET Framework 2.0 and 4.0, Oracle Java Runtime Environment 1.6.0 20 (server version), Mono 2.6.4 (Windows XP), gmcs compiler and Mono 2.4.4 (Linux, Ubuntu distribution).

5.3. Tested codes

To test both algorithms, an application containing two pairs of loops was written: the original code and the proposed. The application was written in two languages: Java for JVM and C# for MS .NET and Mono environments. The execution time of each loop was measured and written to a results file. It is important to show the main loops.

The code for mantissa calculation performance testing is presented in listing 1.8.
double i, sum = 0.0, sum_opt = 0.0;
timer_org.Start();
for(i = -1000; i < 1000; i += 0.0001) {
    sum += (i - Math.floor(i));
}
timer_org.Stop();

timer_opt.Start();
for(i = -1000; i < 1000; i += 0.0001) {
    sum_opt += (i < 0 && i % 1 != 0) ? (i - (int) i + 1) : (i - (int) i);
}
timer_opt.Stop();

Listing 1.8: Mantissa – performance test code (Java)

Next, loops checking the correctness of the sums are needed:

if(sum != sum_opt)
    System.err.println("Optimisation generates incorrect result!");

To avoid dead code elimination during code optimisation, results were summed for each loop (sum and sum_opt variables). The C# code is very similar (except for the size of the first letter of the math floor function), so it is omitted. The code for degrees to radians conversion performance testing is presented in listing 1.9:

double i, sum = 0.0, s_opt = 0.0;
timer_org.Start();
for(i = -1000; i < 1000; i += 0.0001) {
    sum += ((i * 2.0 * Math.PI) / 360.0);
}
timer_org.Stop();

timer_opt.Start();
for(i = -1000; i < 1000; i += 0.0001) {
    s_opt += (i * ((2.0 * Math.PI) / 360.0));
}
timer_opt.Stop();

Listing 1.9: Degrees to radians conversion – performance test code (Java)

The code presented in listing 1.10 was run to ascertain how important the loss of precision is.

for(i = -2000; i < 2000; i += 0.0001) {
    diff = (i * ((2.0 * Math.PI) / 360.0)) - ((i * 2.0 * Math.PI) / 360.0);
    if(i != 0.0)
        rate = Math.abs(diff / (((i * 2.0 * Math.PI) / 360.0)));
    maxrate = Math.max(rate, maxrate);
}

Listing 1.10: Degrees to radians conversion – measurement of precision loss (Java)

Maximum ratio precision difference to computed value equals 2.22044574054295E-16. The achieved precision corresponds to the range of the double precision variable type. For most applications, this precision is fully satisfactory and acceptable.
6. Tests results

Two tests were conducted to measure the performance optimisation in a real environment. The first measures speedup percentages for Mantissa calculations. Listing 1.8 shows code without optimisation (the section with the floor function) and after optimisation (the part with integer casting). The second test was performed for degrees to radians conversion. Listing 1.9 contains code without optimisation (code between the `org.Start()` and `org.Stop()` timer commands with time measured in milliseconds) and optimised code, between the `opt.Start()` and `opt.Stop()` timer functions. Each test was repeated five times. The minimum loop execution times were considered. The results of tests are given in tables 1 and 2, where Org and Opt time show optimised values. The speedup (Org to Opt) and reduction (Opt to Org) ratios were calculated. Loop execution times for the fractional part of a real number are presented in a plot (figure 1a). In the next plot (figure 1b), loop execution times for degrees to radians conversion are shown.

The shortest calculation time without optimisations was achieved in the .NET platform for both hardware environments and both problems. The fractional part of the real number calculation optimisation took the shortest time in the Java platform. It is also worth mentioning that optimised code for Mantissa calculations can significantly increase execution time for negative values because of the use of the % operation.

<table>
<thead>
<tr>
<th>Env</th>
<th>OS</th>
<th>Platform</th>
<th>Org time [s]</th>
<th>Opt time [s]</th>
<th>Speed up [%]</th>
<th>Reduct [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>Win XP</td>
<td>.NET 2.0</td>
<td>0.8388</td>
<td>0.4820</td>
<td>174.0</td>
<td>57.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.NET 4.0</td>
<td>0.8376</td>
<td>0.4949</td>
<td>169.2</td>
<td>59.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mono</td>
<td>2.7958</td>
<td>0.4453</td>
<td>627.8</td>
<td>15.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Java</td>
<td>1.2548</td>
<td>0.3914</td>
<td>320.6</td>
<td>31.2</td>
</tr>
<tr>
<td></td>
<td>Ubuntu</td>
<td>Mono</td>
<td>1.5660</td>
<td>0.4486</td>
<td>349.1</td>
<td>28.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Java</td>
<td>1.4419</td>
<td>0.3930</td>
<td>366.9</td>
<td>27.3</td>
</tr>
<tr>
<td>E2</td>
<td>Win XP</td>
<td>.NET 2.0</td>
<td>0.3802</td>
<td>0.1962</td>
<td>193.8</td>
<td>51.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.NET 4.0</td>
<td>0.3865</td>
<td>0.1811</td>
<td>213.4</td>
<td>46.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mono</td>
<td>1.1952</td>
<td>0.2098</td>
<td>569.7</td>
<td>17.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Java</td>
<td>0.6881</td>
<td>0.1343</td>
<td>512.4</td>
<td>19.5</td>
</tr>
<tr>
<td></td>
<td>Ubuntu</td>
<td>Mono</td>
<td>0.6007</td>
<td>0.2100</td>
<td>286.0</td>
<td>35.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Java</td>
<td>0.6249</td>
<td>0.1272</td>
<td>491.3</td>
<td>20.4</td>
</tr>
</tbody>
</table>

In the case of the second optimisation (degrees to radians conversion), the shortest times with and without optimisation were achieved with the .NET platform in hardware environment E1. The loop execution times for the second environment E2 without optimisation are
very similar for all tested platforms. The shortest execution time after optimisation was achieved with the Java environment.

Speed-up ratios for the first optimisation are shown in figure 2a. The next plot (figure 2b) contains speed up ratios for the second optimisation algorithm. The fastest first algorithm execution was achieved in the Mono platform for both hardware environments. The second fastest was Java under the E2 environment. The speedup of the second algorithm is highest for the .NET platform in environment E1, which is higher than Java in the same environment. In environment E2 (Windows XP), the speedup obtained for the Java platform is definitely the best.

<table>
<thead>
<tr>
<th>Env</th>
<th>OS</th>
<th>Platform</th>
<th>Org time [s]</th>
<th>Opt time [s]</th>
<th>Speed up [%]</th>
<th>Reduct [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>Win XP</td>
<td>.NET 2.0</td>
<td>0.4237</td>
<td>0.1072</td>
<td>395.2</td>
<td>25.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.NET 4.0</td>
<td>0.4181</td>
<td>0.1741</td>
<td>240.1</td>
<td>41.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mono</td>
<td>0.4452</td>
<td>0.1243</td>
<td>358.2</td>
<td>27.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Java</td>
<td>0.4665</td>
<td>0.1205</td>
<td>387.1</td>
<td>25.8</td>
</tr>
<tr>
<td></td>
<td>Ubuntu</td>
<td>Mono</td>
<td>0.4428</td>
<td>0.1250</td>
<td>354.2</td>
<td>28.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Java</td>
<td>0.4698</td>
<td>0.1209</td>
<td>388.6</td>
<td>25.7</td>
</tr>
<tr>
<td>E2</td>
<td>Win XP</td>
<td>.NET 2.0</td>
<td>0.1364</td>
<td>0.0614</td>
<td>222.1</td>
<td>45.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.NET 4.0</td>
<td>0.1364</td>
<td>0.0614</td>
<td>222.1</td>
<td>45.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mono</td>
<td>0.1364</td>
<td>0.0659</td>
<td>207.0</td>
<td>48.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Java</td>
<td>0.1364</td>
<td>0.0409</td>
<td>333.5</td>
<td>30.0</td>
</tr>
<tr>
<td></td>
<td>Ubuntu</td>
<td>Mono</td>
<td>0.1365</td>
<td>0.0686</td>
<td>199.0</td>
<td>50.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Java</td>
<td>0.1365</td>
<td>0.0720</td>
<td>189.6</td>
<td>52.7</td>
</tr>
</tbody>
</table>

Table 2

Results of degrees to radians conversion tests

![Fig. 1. Loop execution times for: a) mantissa calculation and b) conversion from degrees to radians](image_url)

Rys. 1. Czasy wykonania pętli dla: a) wyznaczania mantysy oraz b) przeliczania ze stopni na radiany
7. Proposed Optimisations and Standard Optimisation Techniques

Code is optimised on the fly by virtual machines using the Just-In-Time technique. It is difficult to determine which of the well-known set of optimisation algorithms are used: standard compilers (for example C compiler) or Ahead-Of-Time compilation. The main environments (JVM, .NET Framework) do not provide any information about current execution, and except for one switch (java -server) there is no option to select the optimisation technique that should be used. Only the Mono environment provides this option with the use of switches. It is advisable to check how efficient these techniques are for the proposed optimisations. The tests were done for the following techniques: peephole post pass (peephole), branch optimisations (branch), inline method calls (inline), constant folding (cfold), constant propagation (consprop), copy propagation (copyprop), dead code elimination (deadce), linear scan global registry allocation (linears), conditional moves (cmov), emit per-domain code (shared), instruction scheduling (sched), intrinsic method implementations (intrins), tail recursion and tail calls (tailc), loop related optimisations (loop), leaf procedures optimisations (leaf), SSA based Partial Redundancy Elimination (ssapre), and array bound checks removal (abcrem).

To complete the research, the processor dependent techniques were surveyed: SSE2 instructions on x86 (sse2) and Fast x86 FP were compared (fcmov). Mono environment specific options were also tested: usage of Ahead Of Time compiled code (act), all optimisations (all), without optimisations (-all), default optimisations, selected by Mono (default).

Although the SSE2 technique is particularly suited to arithmetic that is more complicated such as butterfly calculations or matrix operations, it was used in this research in the interests of completeness. The test results in environment E1 are presented in figure 3a for the fractional part of real number calculation and in figure 3b for degrees to radians conversion. The loop execution time without optimisations (-all) presented in figure 3a is interesting. Al-
though some optimisation took place, this speed reduction was not expected and requires further research.

In the case of the original computational problems, the results are comparable. Code execution of the first optimisation algorithm can be accelerated by the cmov, loop, leaf, and aot techniques and sse2 hardware specific instructions. The results for the second algorithm are comparable (except for the three techniques described below). Worse results were achieved with abcrem, ssapre and all optimisations for both proposed algorithms, which shows the weakness of the optimisation methods implemented for the Mono platform, which uses a simplified Single Static Assignment (SSA) solution.

Fig. 3. Loop execution times for: a) mantissa calculation and b) degrees to radians conversion
Rys. 3. Czasy wykonania pętli dla: a) wyznaczania mantysy oraz b) przeliczania ze stopni na radiany
8. Summary

The proposed algorithms achieve great computation time reductions in the three leading component platforms. The level of efficiency depends on software and hardware specifications [5]. For example, operation execution times for both optimisations are reduced in the .NET 2.0 framework from 109 ms to 93 ms (Intel Core Duo 2.2 GHz) and from 125 ms to 109 ms in .NET 2.0 and from 234 ms to 156 ms in Mono (AMD K2 5000+). This confirms the objective of implementing optimisations in the source code of component environments. Further research could focus on the possibilities of applying alternative optimisation methods for code written in C and consider other numerical operations that could be optimised in software for Pareto joint inversion of 2D geophysical data [23].

BIBLIOGRAPHY


Omówienie

Artykuł porusza temat wydajnościowej optymalizacji kodu przeznaczonego do uruchamiania w środowisku maszyn wirtualnych. Rozważania te mają szczególne znaczenie dla obliczeń numerycznych, które bywają dość czasochłonne. Opisane zostały dwie niestandardowe metody przyspieszające operacje matematyczne. Pierwsza technika dotyczy przekształcenia wyrażenia obliczania mantysy (listing 1.1 – 1.3), dając w rezultacie ponadściorokrotne przyspieszenie wykonywanej operacji, z założeniem ograniczeń zakresu wartości wejśció-
wych. Druga optymalizacja dotyczy zagadnienia konwersji miar kątów ze stopni na radiany (2), dająca nawet czterokrotne przyśpieszenie, jednakże kosztem (aczkolwiek wciąż w zakresie dopuszczalnym) zmniejszenia precyzji rozwiązania. Rozważania zilustrowano listingami kodów oraz wynikami testów (tabela 1, tabela 2), które zostały przeprowadzone w różnych środowiskach (Java, MS .NET i Mono). Uzyskane rezultaty pokazują znaczną przewagę wydajnościową opisywanych metod nad standardową optymalizacją kompilatora, co jednak powiązane jest z pewnymi ograniczeniami zakresu precyzji otrzymywanych wyników.

Addresses

Marek ŻUPNIK: marek.zupnik@gmail.com.
Katarzyna MIERNIK: AGH University of Science and Technology, Department of Geoinformatics and Applied Computer Science, al. A. Mickiewicza 30, 30-059 Cracow, Poland, kmiernik@agh.edu.pl.
Zdzisław ONDERKA: AGH University of Science and Technology, Department of Geoinformatics and Applied Computer Science, al. A. Mickiewicza 30, 30-059 Cracow, Poland, zonderka@agh.edu.pl.