Computer Science • 17 (4) 2016 http://dx.doi.org/10.7494/csci.2016.17.4.461

Roman Dębski

SIMULATION-BASED SAILBOAT TRAJECTORY OPTIMIZATION USING ON-BOARD HETEROGENEOUS COMPUTERS

Abstract *A dynamic programming-based algorithm adapted to on-board heterogeneous computers for simulation-based trajectory optimization was studied in the context of high-performance sailing. The algorithm can efficiently utilize all OpenCL-capable devices, starting the computation (if necessary, in single-precision) on a GPU and finalizing it (if necessary, in double-precision) with the use of a CPU. The serial and parallel versions of the algorithm are presented in detail. Possible extensions of the basic algorithm are also described. The experimental results show that contemporary heterogeneous on-board/mobile computers can be treated as micro HPC platforms. They offer high performance (the OpenCL-capable GPU was found to accelerate the optimization routine 41 fold) while remaining energy and cost efficient. The simulation-based approach has the potential to give very accurate results, as the mathematical model upon which the simulator is based may be as complex as required. The black-box represented performance measure and the use of OpenCL make the presented approach applicable to many trajectory optimization problems.*

Keywords black-box optimization, trajectory optimization, dynamic programming, heterogeneous computing, micro HPC platform

Citation Computer Science 17 (4) 2016: 461–481

1. Introduction

Trajectory optimization is a classic problem in many engineering disciplines, such as robotics, aerospace engineering, or optimal control. In most cases, however, it cannot be solved analytically. A typical situation is when the explicit formula of the performance measure is unknown (i.e., it is *black-boxed*) to the optimization routine and its values are obtained from computer simulation. Since the cost (time complexity) of a single simulation is often significant, simulation-based optimization tasks are usually solved using HPC-clusters (or cloud computing platforms). However, in many on-board (or embedded) systems like the trajectory planners of sailboats (or autonomous robots), this approach cannot be applied. What is often used instead are (over-)simplified mathematical models that can only give very rough approximations of optimal trajectories. This can be unacceptable in many situations; for instance, in high-performance sailing (like the America's Cup) or when sailing in bad and/or fast-changing weather conditions.

The main aim of this paper is to present an approach that allows us to obtain accurate results of trajectory planning using only on-board/mobile computers. The main contributions of this paper are:

- an effective algorithm (adapted to on-board heterogeneous computer systems) for simulation-based sailboat trajectory optimization (simulation-based optimization can be very accurate, because the mathematical model that the simulator is based on may be as complex as required),
- experimental results that prove that contemporary heterogeneous on-board (or mobile) computers can be treated as micro HPC platforms, as they offer high performance while remaining energy and cost efficient (which is often crucial in many on-board and/or embedded systems).

The target platform is assumed to be heterogeneous; i.e., composed of multiple processor types[1], and that the algorithm is implemented in OpenCL (Open Computing Language), which is defined by an open standard and is cross-platform (unlike CUDA [Compute Unified Device Architecture]). OpenCL implementations are now available for widely used CPUs and GPUs. The OpenCL standard also defines *the OpenCL Embedded Profile* – a special version of the platform for embedded systems and mobile devices. The proposed approach (i.e., the use of OpenCL and black-box representation of performance measure) is quite general, both from the deployment point of view (mobile/on-board devices, modern embedded systems, but also HPC-clusters), and because of the scope of the optimization problems it covers.

The remainder of this paper is organized as follows. The next section contains a review of related work. Following that, the optimization problem is defined (including a description of the simulation model). Next, the proposed algorithm is described, and some remarks about augmenting the algorithm are also given. After

[1] CPU and GPU, but also possibly of FPGA and/or DSP.

that, the experimental results are presented and discussed. The last section contains the conclusion of the study.

2. Related work

The brachistochrone problem[2], formulated by Johan Bernoulli in 1696, is often considered the first scientific formulation of the problem of trajectory optimization. One of its first solutions (published by Johan's brother Jakob) significantly contributed to the development of the calculus of variations (see, for instance, [32]) – the field of mathematics which has played a crucial role in trajectory optimization over the last three hundred years.

The real breakthrough in this field came in the 1950s with the development of the digital computer and introduction of dynamic programming ([2]), effective shortest path algorithms ([3, 13]), and the Pontryagin Maximum Principle ([26]). All of these, together with non-linear programming (NLP), have become the bases for many effective trajectory optimization methods that are commonly classified as either *direct* or *indirect* (see, for instance, [4, 19, 33, 37]). The indirect methods, based on the calculus of variations, aim for solutions that satisfy the necessary conditions of optimality. By contrast, the direct ones search for solutions having the best value of performance measure. Trajectory optimization methods based on the shortest path algorithms (see, for instance, [7, 14, 29]) can be considered as a special case of this second approach.

A special group of trajectory optimization problems comprises those having *black-box represented* performance measures. A typical example of this situation is when the performance measure values are received from computer simulation. In this case, most classic optimization methods cannot be used (at least not directly), and the optimization process is often based on *soft-computing* methods (see, for instance, [6, 27, 28, 36, 38]).

Many existing trajectory optimization case studies are related to aerospace engineering (see, for instance, [6, 29]), but there are also others (see, for instance, [10, 15]), and a number of these are related to (autonomous) sailboat trajectory planing (see, for instance, [8, 23–25, 31]).

Another important research area in the context of this paper is related to the parallelization of trajectory optimization algorithms (see, for instance, [7, 18]) including the possibility of their GPU-acceleration (see, for instance, [1, 17, 22, 30, 39]).

3. Problem formulation

The trajectory optimization task is *to find among all admissible trajectories the one with the best value of the performance measure*. The performance measure can be

[2]Discussed in a broad sense by [34, 35].

formulated this way:

$$J = h\big(\boldsymbol{x}(t_f), t_f\big) + \int_{t_0}^{t_f} g\big(\boldsymbol{x}(t),\ \boldsymbol{u}(t),\ t\big)dt \tag{1}$$

where: t_0 and t_f are the initial and final times, h and g are scalar functions, and $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ is the input vector representing a trajectory (encoded in some way). Sometimes, for instance when the values of the performance measure are taken from a computer simulation, the explicit formula of the performance measure[3] is unknown; i.e., it is "opaque" (or black-boxed) to the optimization routine, and therefore indirect methods cannot be applied (at least directly). In such cases, special direct-method based algorithms are usually used.

The sailboat trajectory optimization problem analyzed in this paper (see Fig. 1) is an example of a two-dimensional, continuous trajectory optimization task having a black-box represented performance measure. Simulation-based optimization means that the sailboat model (simulator) can be as complex as required.

Figure 1. A sailboat trajectory optimization: an admissible trajectory of a sailboat sailing upwind from point A to point B via points $P1$ and $P2$ (it is impossible to sail directly upwind).

The details of the sailboat trajectory evaluator (simulator) used in this paper are described in Appendix A.

4. Proposed algorithm

The approach presented in this section is based on the following main two steps:

1. transformation (using a grid-based discretization scheme) of the continuous optimization problem into a search problem over a specially constructed finite graph[4],

[3]Id est, the objective function (functional).

[4]Note that the problem (and, as a result, the corresponding grid) can be also formulated in a polar coordinate system.

2. application of dynamic programming to find the approximation of the minimum-time sailing line[5].

The discretization process can be repeated several times[6]. The next stage mesh (grid) can be generated through mesh refinement, making use of the best trajectory found thus far.

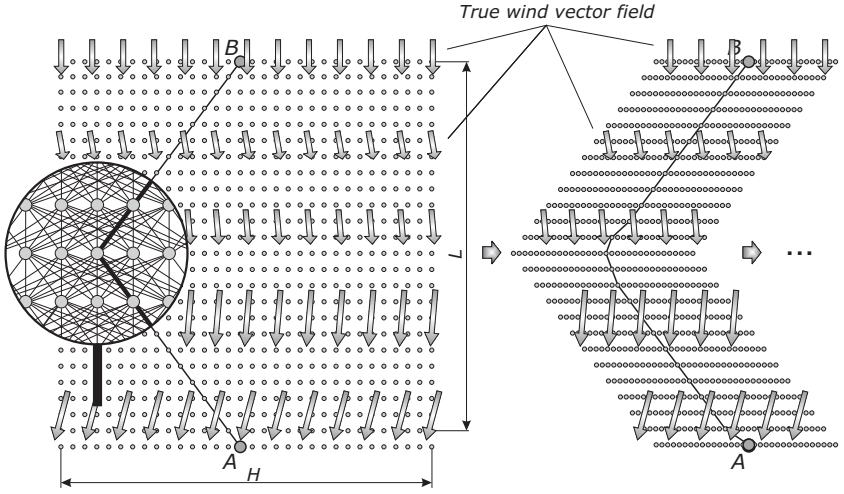

Figure 2. Grid-based discretization scheme transforming the continuous optimization problem into a search problem. The grid nodes are grouped in rows – the grid shown in the figure has thirty two nodes in a row. The discretization process can be repeated several times (a form of iterative improvement algorithm through successive mesh refinements). *Note*: the circle on the left is the magnification of one part of the grid.

The grid is based on equidistant nodes (see Fig. 2), and the graph it represents is directed, acyclic, and "topologically sorted" (the nodes in row r are followed by the nodes in row $r+1$). The above features allow the search process to be more effective than in the standard Dijkstra Shortest Path algorithm. It is worth noting here that, at the beginning of this simulation-based optimization process, there is no cost matrix – the cost of each edge (i.e., linear segment of the trajectory) can be calculated (through simulation) only when the corresponding initial velocity is known; this value depends on the results received from the simulation for preceding segments. This dependence is a sequential component of the optimization algorithm.

We assume that nodes from any two subsequent rows (layers) are fully connected (i.e., each node in row r is connected to all nodes from rows $r-1$ and $r+1$). Because of the way the boundary conditions are formulated (see Eqn.13 and Fig. 10), C simulations have to be performed to evaluate a single linear segment. This number

[5]Represented as a piecewise-linear function (see Fig. 2).
[6]This can be considered as a form of the *iterative improvement algorithm*.

can usually be reduced if we make use of some features of the model (as discussed in the final paragraph of this section).

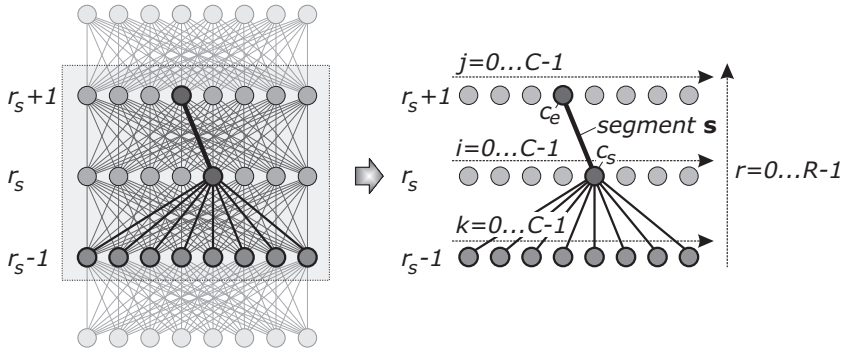

Figure 3. Dependence of the segments in the mesh (grid): segment $s(r_s, c_s, c_e)$ depends on all segments $s^{(k)}(r_s-1, k, c_s)$. Arrows with ranges $r = 0, \ldots R-1$; $i = 0, \ldots C-1$; $j = 0, \ldots C-1$; $k = 0, \ldots C-1$ describe the loops of the algorithm pseudo-codes presented in this section (see also [10]).

The pseudo-code of simulation for the segments in the first row (note that they do not have any predecessors) is shown as Algorithm 1 [10].

Algorithm 1. sim_0 – perform simulation for a segment in mesh row 0.

Require: c_s, c_e

1: {@param c_s - the segment start node column index}
2: {@param c_e - the segment end node column index}
3: $(l^{(s)}, \beta_t^{(s)}) := get\_segm\_stat\_data(0, c_s, c_e)$
4: $(t_1^{(s)}, v_1^{(s)}) := simulation\_for(l^{(s)}, \beta_t^{(s)}, 0, 0)$
5: {set $t_1^{(s)}$ and $v_1^{(s)}$ as s's optimal values}
6: $update\_segment(0, c_s, c_e, t_1^{(s)}, v_1^{(s)}, NULL)$

In the first step, we calculate segment length $l^{(s)}$ and corresponding (local) true wind angle $\beta_t^{(s)}$. Next, the values for time $t_1^{(s)}$ and speed $v_1^{(s)}$ are received through simulation. And finally, these values are stored, as they will be needed in simulations for segments from the next layer (see Eqn.13 and Fig. 10).

Algorithm 2 shows the main steps of the simulation for the segments in rows $1..R-2$.

To find the shortest time of travel to the end point of a segment (for instance, point $(r_s + 1, c_e)$ in Fig. 3), it is necessary to perform C simulations because of the way the boundary conditions for each segment are formulated (see Eqn.13 and Fig. 10), updating (if necessary) the best solution found thus far (compare [13]). Variable $pidx_{opt}$ stores the index of the segment from row $r_s - 1$, which corresponds

to the locally optimal section of the trajectory[7]. *Note:* sim_0 and sim_{1R} presented as Algorithms 1 and 2 are implemented as *OpenCL kernels*.

Algorithm 2. sim_{1R} – perform simulation for a segment in mesh rows $[1..R-1)$.

Require: r_s, c_s, c_e
1: {@param r_s - the segment start node row index}
2: {@param c_s - the segment start node column index}
3: {@param c_e - the segment end node column index}
4: $t_{opt} := T_{MAX}$
5: $(l^{(s)}, \beta_t^{(s)}) := get\_segm\_stat\_data(r_s, c_s, c_e)$
6: {For all mesh nodes in the $(r_s - 1)$-th row}
7: **for all** k in [0..C) **do**
8: {Initialize $t_0^{(s)}, v_0^{(s)}$ with the corresponding final values for segment $(r_s - 1, k, c_s)$}
9: $(t_0^{(s)}, v_{0i}^{(s)}) := get\_segm\_dyn\_data(r_s - 1, k, c_s)$
10: {If necessary, correct v_0, see Eqn.13}
11: $v_0^{(s)} = correct\_v_0(r_s - 1, k, c_s, c_e, v_{0i}^{(s)})$
12: $(t_1^{(s)}, v_1^{(s)}) := simulation\_for(l^{(s)}, \beta_t^{(s)}, t_0^{(s)}, v_0^{(s)})$
13: {If necessary, update the best values so far}
14: **if** $(t_0^{(s)} + t_1^{(s)} < t_{opt})$ **then**
15: $t_{opt} := t_0^{(s)} + t_1^{(s)}$
16: $v_{opt} := v_1^{(s)}$
17: $pidx_{opt} := k$
18: **end if**
19: **end for**
20: {Update the segment data}
21: $update\_segment(r_s, c_s, c_e, t_{opt}, v_{opt}, pidx_{opt})$

Algorithm 3 is the serial version of the trajectory optimization procedure (see loop control variables with Fig. 3). The algorithm time complexity is equal to $\Theta\left(RC^3 t_{sim}\right)$, where R and C are the numbers of rows and columns (in the mesh/grid), respectively, and t_{sim} is the time of a single simulation.

The current implementation of the algorithm needs $\Theta\left(RC^2\right)$ memory but can be reduced[8] to $\Theta\left(\max(RC, C^2)\right)$.

We can parallelize Algorithm 3, making use of the fact that the graph representing the grid is "topologically sorted", and so the simulations for all segments in the same row are independent of each other. The simulations can be performed in parallel on any (or all) of the OpenCL-capable devices with which the on-board computer system is equipped. This idea is shown in Figure 4 and Algorithm 4.

sim_0_kernel and sim_{1R}_kernel are intended to be OpenCL kernels; i.e., pieces of code prepared for parallel execution on OpenCL-capable devices.

[7]Id est, the one with the shortest time of travel to the segment end point; indexes $pidx_{opt}$ are used in the final part of the algorithm to find the optimal solution.

[8]The complexity can be additionally reduced, for instance, by reduction of the graph node order.

Algorithm 3. Serial trajectory optimization.

1: {For all mesh nodes in row 0; note: A_{idx} is the index of point A}
2: **for all** j in [0..C) **do**
3: $sim_0(A_{idx}, j)$
4: **end for**
5: {For rows 1..R-3 in the mesh}
6: **for all** r in [1..R-2) **do**
7: {For all mesh nodes in row r }
8: **for all** i in [0..C) **do**
9: {For all mesh nodes in the $(r+1)$-th row}
10: **for all** j in [0..C) **do**
11: $sim_{1R}(r, i, j)$
12: **end for**
13: **end for**
14: **end for**
15: {For all mesh nodes in row $R-2$; note: B_{idx} is the index of point B}
16: **for all** i in [0..C) **do**
17: $sim_{1R}(R-2, i, B_{idx})$
18: **end for**
19: {$opt\_trajectory$ is calculated using back-pointers ($pidx_{opt}$)}
20: **return** $opt\_trajectory, fin\_time, fin\_velocity$

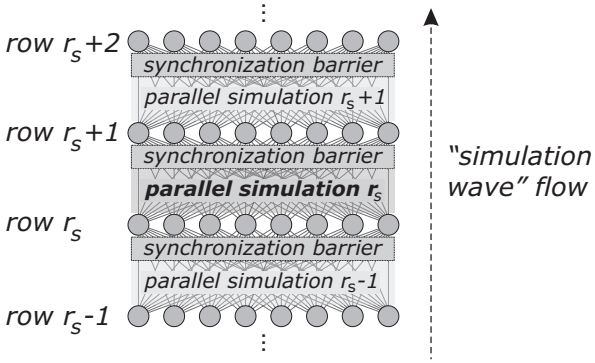

Figure 4. Parallel simulation for all segments in the same mesh row. The synchronization barrier is needed because of the dependence between nodes from the subsequent rows (see Eq.13 and also [10]).

The parallel algorithm time complexity is equal to $\Theta\left(RCt_{sim} \max\left(C^2/p_e, 1\right)\right)$, where p_e is the number of processing elements (CUDA cores). Annotation $@PARALLEL(f(.))$ expresses (in pseudo-code, only as there is no such annotation in OpenCL) that function f CAN BE executed in parallel for different pieces of data referenced by pointer – $data_p$.

Algorithm 4. Parallel trajectory optimization.

1: {For all segments starting at point A and ending at row 1}
2: @PARALLEL ($sim_0\_kernel(data_p)$)
3: **for all** r in [1..R-2) **do**
4: @PARALLEL ($sim_{1R}\_kernel(r, data_p)$)
5: **end for**
6: {For all segments starting at row R-2 and ending at point B}
7: @PARALLEL ($sim_{1R}\_kernel(R-2, data_p)$)
8: **return** $opt\_trajectory, fin\_time, fin\_velocity$ {as before, using back-pointers ($pidx_{opt}$)}

In some on-board (embedded) computer systems, the described version of the optimization procedure has to be changed to meet certain constraints (for instance, regarding time and/or memory complexity). One way of addressing this issue (apart from successive mesh refinements, already proposed in the basic algorithm) is reducing the number of connections between grid nodes (see Fig. 5). This decreases both time and memory complexity (fewer segments means fewer simulations to perform and less memory to store segment data). However, the effectivenesses of this approach is very problem-dependent – in many cases, we cannot reduce the search space in this way without risking the loss of good solution candidates.

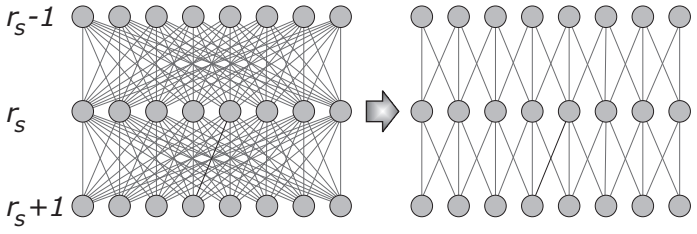

Figure 5. Reduction of the number of connections between nodes (i.e., the degree of the graph nodes).

It is worth noting here that the presented algorithm can be augmented by a local search/optimization (the basic algorithm is a global search in a discrete space). In some cases, this can significantly improve the final result. This is because the search space of the local algorithm can be continuous, which means that there is no accuracy limit related to mesh granularity. One of the key aspects of the successful application of local optimization is choosing a basis appropriate for the problem in the search space (see [9]).

5. Experimental results

To demonstrate the effectiveness of the algorithms presented in Section 4, a series of experiments was carried out to find the optimal sailing line for two different locations of target point B relative to the true wind direction (which, without loss of generality, was assumed constant – see angle $\beta_0 = \pi/36$ in Fig. 8). The true wind speed $|\bar{v}_t|$ was the same in all experiments and set to 12 m/s. The constants of the model (see Eq.11) were as follows: $c_1 = 0.03, c_2 = 0.005, k = 0.7$. The size of the "simulation rectangle" $H \times L$ was 800 $m \times 500$ m (see Fig. 2). In all experiments, a MacBook Pro was used[9] with OS X 10.9.3 and OpenCL 1.2, having two OpenCL-capable devices:

- processor *Intel Core i7-3740QM* @ 2.7 GHz,
- graphic card *nVidia GeForce GT 650m*[10].

5.1. Performance analysis: the impact of compiler optimizations

Modern compilers have optimization abilities that can significantly improve program performance. Some of them, including Clang/LLVM 5.1[11], try to provide optimizations for programs at every possible optimization stage (i.e., compile time, link time, and runtime). Available optimization options usually let the programmer choose whether they prefer a smaller target file size, faster code, or faster build times. In the case of on-board computer systems (or embedded systems), the combination of the first two (i.e., fast and small) is usually the preferred option.

This paragraph contains a brief analysis of the impact of the compiler optimizations on sequential program execution time. The results are summarized in Table 1.

Table 1

Average execution times m (in seconds) and standard deviations s, from 11 runs, for different optimization levels set in Apple LLVM 5.1 compiler (sequential algorithm run on the CPU; the model with $\beta_0 = \pi/36$, $C = 128$, where C – the number of nodes in one row of the grid).

	$-O0$	$-O1$	$-O2$	$-O3$	$-Ofast$	$-Os$
m	1284.23	873.22	857.88	857.35	815.34	874.43
s	1.665	1.140	1.265	0.780	0.812	3.358

The second column in the table ($-O0$ option) presents the mean value (m) and standard deviation (s) of the execution times of the code compiled with no optimization. The next four ($-O1$ to $-Ofast$) show the corresponding values for four levels of optimization – from basic to the most "aggressive". The last column $-Os$ represents a special level in which the compiler performs all optimizations that do not typically increase the target file size. It can be seen that the $-Ofast$ option allows for the

[9] With 16GB of DDR3L 1600 MHz RAM.

[10] Two compute units, each having 192 processing elements (CUDA cores), warp size 32, 1 GB of GDDR5 memory, 48 KB of local memory, 64 KB of constant memory.

[11] The compiler that was used in the experiments.

reduction of execution time by 36%, while $-Os$ gives a 32% lower execution time. This proves that the $-Os$ option is the recommended one, since it is only slightly less effective in reducing execution time (32% vs. 36%), but it achieves this without increasing the target file size ($-Os$: 206 kB vs. $-Ofast$: 210 kB).

5.2. OpenCL platforms comparison: CPU vs. GPU performance

According to the OpenCL execution model (see, for instance, [16]), a CPU host defines an N-dimensional computation domain over some region of an OpenCL device's DRAM memory. Each index of this N-dimensional domain at runtime corresponds to a work-item (which executes the same OpenCL kernel).

In the experiments performed, a two-dimensional domain was assumed, with each work-item mapped to one segment simulation. As there were C nodes in one row (see Fig. 3) and each node in a row was connected to all nodes in the subsequent row, the total number of connections (i.e., linear segments), which is equal to the total number of work-items, was

$$\#WI = C \times C = C^2 \tag{2}$$

The results for $\beta_0 = \pi/36$ are summarized by Table 2 and Figure 6.

Table 2

Average execution times m (in seconds) and standard deviations s, from 11 runs, for different numbers of mesh nodes in a row (the model with $\beta_0 = \pi/36$). The last column (t_f) contains the final results (in seconds).

C	#WI	t_s		$t_{ocl-CPU}$		$t_{ocl-GPU}$		t_f
		m	s	m	s	m	s	
8	64	0.21	0.002	0.05	0.001	0.24	0.002	1212.32
16	256	1.74	0.005	0.30	0.022	0.54	0.004	655.36
32	1024	13.76	0.060	1.97	0.022	1.08	0.012	411.25
64	4096	107.47	0.290	14.82	0.031	3.92	0.017	357.85
128	16384	874.43	3.358	128.63	0.466	21.33	0.120	352.17

In Table 2, the subsequent columns represent: C – the number of mesh nodes in one row; $\#WI$ – the corresponding number of work-items; t_s – the execution time of the serial version of the optimization algorithm (see Algorithm 3); $t_{ocl-CPU}$ and $t_{ocl-GPU}$ are the execution times of (parallel) Algorithm 4 run on the CPU and the GPU, respectively; and finally, for reference, t_f – the final result of the optimization.

Three facts are worth noting with regard to the parallel algorithm performance (see Fig. 6):

- On the GPU, there is a strong dependence of the speedup ratio[12] on the mesh-size), whereas on the CPU, this dependence is much weaker (almost negligible).

[12]Calculated in a classic way (i.e., as $speedup = t_s/t_p$).

This is a typical feature of computations performed on a GPU – the more work-items (which can be seen as threads) the GPU has to handle, the better the speedup ratio is.

- The overhead related to data transfers between the host and GPU can be sometimes so high that using the GPU as a general-purpose computing device has no benefit – for $C \leqslant 8$ (which corresponds to $\#WI \leqslant 64$), the execution times on the GPU were higher than the corresponding ones of the sequential algorithm!
- For a smaller number of work-items (in this example, it was for $C \leqslant 16$, which corresponds to $\#WI \leqslant 256$), using the CPU as the OpenCL (computing) device was more effective than using the GPU.

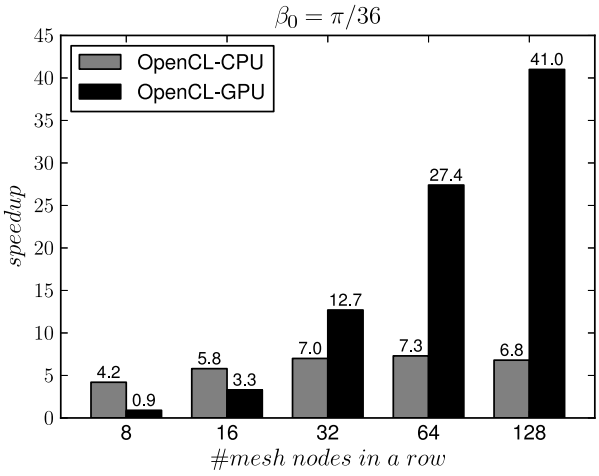

Figure 6. Speedups for the sailboat model with $\beta_0 = \pi/36$. *Note*: the values were calculated taking $t_s = t_{s-Os}$ (see Table 1).

Based on the above observations, one can see an important advantage of using OpenCL for programming heterogeneous computer systems – because the code is exactly the same for all OpenCL-capable devices, any[13] of the available OpenCL devices can be used with no additional programming cost, and the selection of the computing device can be done even at run-time.

The corresponding results for $\beta_0 = \pi/6$ are summarized by Table 3 and Figure 7. This parameter of the model was selected to demonstrate worse (than before) results of utilizing the GPU as a general-purpose computing device.

Figure 7 presents the parallel algorithm performance from the speedup point of view.

[13]Sometimes more than one OpenCL-capable device can be used at the same time.

Table 3

Average execution times m (in seconds) and standard deviations s, from 11 runs, for different numbers of mesh nodes in a row (the model with $\beta_0 = \pi/6$). The last column (t_f) contains the final results (in seconds).

C	#WI	t_s		$t_{ocl-CPU}$		$t_{ocl-GPU}$		t_f
		m	s	m	s	m	s	
8	64	0.22	0.001	0.06	0.002	0.33	0.002	331.35
16	256	1.79	0.007	0.33	0.011	0.64	0.004	331.35
32	1024	14.18	0.128	2.07	0.022	1.20	0.006	331.35
64	4096	108.02	0.171	15.27	0.181	4.16	0.028	331.35
128	16384	879.77	8.701	129.88	0.295	27.35	0.190	314.65

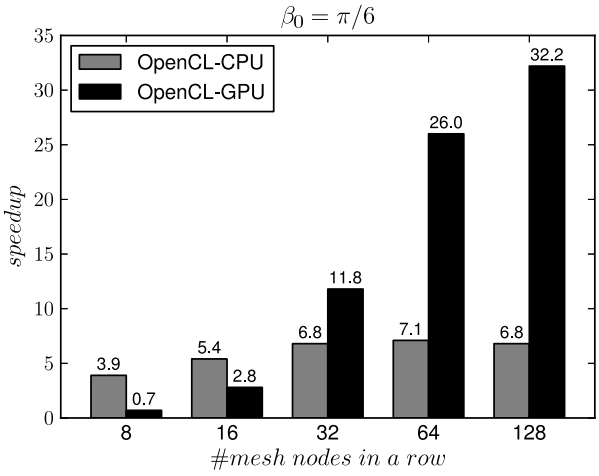

Figure 7. Speedups for the sailboat model with $\beta_0 = \pi/6$. *Note*: the values were calculated taking $t_s = t_{s-Os}$ (see Table 1).

The observations made previously (i.e., for the model with $\beta_0 = \pi/36$) are still valid, but the speedups related to the GPU were worse than before (especially for $C = 128$). For eight mesh-nodes in a row (which corresponds to 64 work-items, compare Eq.2), the GPU-accelerated computation *again took more time than the serial algorithm*. For 32 nodes, the speedup for the GPU was about 74% better that the corresponding one for the CPU; and finally, the biggest speedup ratio, observed as before for $C = 128$, was equal to 32.2. This worse performance was a result of an unbalanced (among work-items) computational load (the computation for a single mesh row was as long as the longest simulation of its single segment)

Finally, it is worth adding that, in the experiments performed, the OpenCL "auto mode" was used, which means that the number of work-items in one work-group was

managed by the OpenCL platform itself (see, for instance, [16]). This mode proved effective in similar computational tasks (see, for instance, [9]).

5.3. Optimization results analysis

As an example of the optimization results, the trajectories corresponding to each of the two steps of the iterative improvement algorithm (successive mesh refinements), run for $\beta_0 = \pi/36$, are presented in Figure 8.

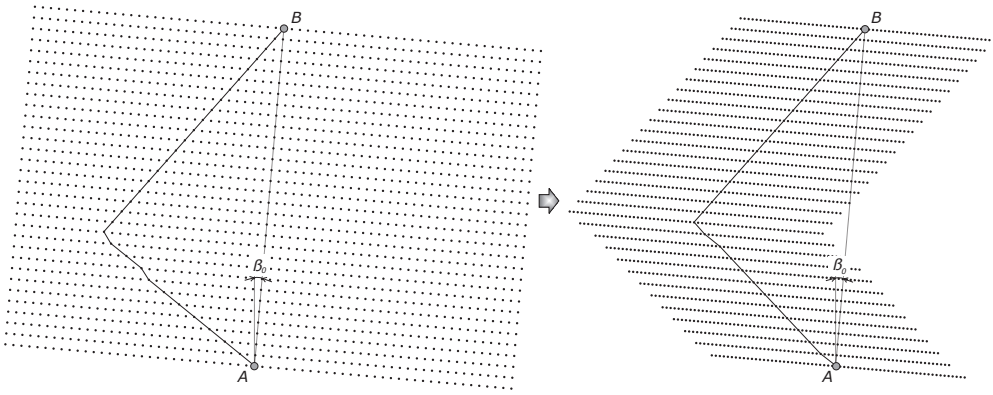

Figure 8. Optimization results for the model with $\beta_0 = \pi/36$: the final (second) step of the successive mesh refinement algorithm with $C = 64$. The corresponding result was $t_f = 352.17\,\mathrm{s}$.

In both steps, the number of grid nodes in one row (C) was set to 64. This corresponded in the first step to a row inter-node distance equal to approximately 12.7 m, whereas in the second step to 6.3 m.

During the experiments, $C = 64$, set for the initial step of the iterative improvement algorithm, was the minimum number of mesh nodes in one row that gave a good first approximation of the final solution. In the analyzed case (i.e., for $\beta_0 = \pi/36$), the second iteration (step) improved the final result only by about 1.6%, which was enough to stop the computation. In general, the stopping criterion is a trade-off between the required accuracy of the final result and the maximum (acceptable) computation duration.

The performance analysis presented in Section 5.2 and the above findings suggest that the iterative improvement algorithm (adapted to heterogeneous computing platforms) can start on the GPU, with the initial mesh size C set to 64 (or even 128) and, if necessary, perform the computation in single-precision. After a couple of iterations (mesh refinements), the search space size can be reduced; so when, for instance $C = 16$, the computation may be "switched" to the CPU[14].

[14]If necessary, performing the computation in double-precision.

Note: as an extension of this algorithm, an additional step may be added, in which the best solution found so far is "refined" by local optimization in continuous space.

6. Conclusion

An effective algorithm, adapted to on-board/mobile heterogeneous[15] computer systems, for simulation-based trajectory optimization has been studied using an example taken from high-performance sailing. The serial and parallel versions of the optimization algorithm have been presented in detail. Possible extensions of the basic algorithm have also been described. The presented iterative improvement algorithm can utilize all OpenCL-capable devices in an efficient way starting the computation (if necessary, in single-precision) on a GPU and finalizing it (if necessary, in double-precision) with the use of a CPU. As an additional step, the best trajectory found by the base algorithm can be refined by local optimization in continuous space.

The experimental results have shown that contemporary heterogeneous on-board (or mobile) computers can be treated as micro HPC platforms – they offer high performance (the effective use of the OpenCL-capable GPU accelerated the optimization routine up to 41 fold) while remaining energy and cost efficient (which is often crucial in many on-board and/or embedded systems). Therefore, they can be effectively used for solving simulation-based trajectory optimization problems.

The proposed approach (i.e., the use of OpenCL and black-box representation of performance measure) is very general from both the potential deployment point of view (mobile/on-board devices, modern embedded systems, but also HPC-clusters) and because of the scope of the optimization problems it covers. The simulation-based approach can be much more accurate because the mathematical model that the simulator is based on may be as complex as required (in contrast to the often-used "boat polar-based" approach, which gives only theoretical maximum boat speeds at various true winds.

Future research work could concentrate on:

- experimenting with the proposed extensions of the basic (parallel) algorithm,
- implementing a more-accurate sailboat model,
- experimenting with different ways of representing the trajectory,
- refining the iterative improvement algorithm itself (including the optimal resource usage, load balancing, and mesh-generation algorithms),
- verifying the presented algorithms in different trajectory optimization problems,
- verifying the presented algorithms in a distributed computing environment (including the *augmented cloud*[16]).

[15] Id est, composed of multiple processor types.
[16] See, for instance [5, 11, 12].

Acknowledgements

The research presented in this paper was partially supported by the Polish Ministry of Science and Higher Education under the AGH University of Science and Technology grant (statutory project) no. 11.11.230.124.

A. Simulation-based trajectory evaluator (the black-box simulator)

Consider a sailboat of mass m sailing upwind in direction s in true wind velocity (v_t) vector field as shown in Figures 1 and 9. We assume that the aerodynamic side force S_a generated by the wind pressure on the sails is matched instantaneously by S_h (the hydrodynamic side force), while the thrust T in general differs from the resistance R (compare, for instance, [24]).

The trajectory is approximated by a piecewise-linear function. This simplifies the problem significantly – instead of one (complex) two-dimensional problem, we have a series of (simple) one-dimensional ones [10]. Each of the sub-problems is related to one segment only.

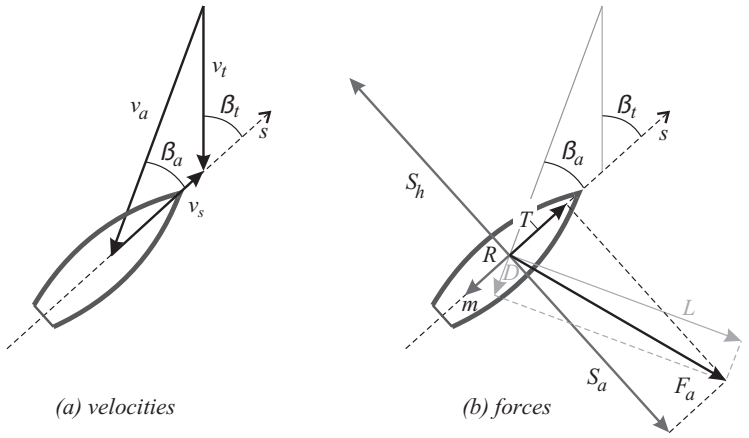

Figure 9. A sailboat (sailing upwind) model: β_t – true wind angle, β_a – apparent wind angle, s – (current) sailing direction, v_t – true wind velocity, v_a – apparent wind velocity, v_s – current sailboat velocity, F_a – aerodynamic force, L – lift (the aerodynamic force component perpendicular to the wind direction), D – drag (the aerodynamic force component in the direction of the wind), T – thrust, S_a – side force, S_h – the hydrodynamic side force, R – total (i.e., hydrodynamic plus aerodynamic) resistance.

Equation of motion. After applying Newton's second law for direction s, we receive[17]

$$m\dot{v}_s = m\ddot{s} = T - R \tag{3}$$

where

$$T = f(\beta_a, v_a) = (C_1\beta_a - C_2)v_a^2 = (mc_1\beta_a - mc_2)v_a^2 \tag{4}$$

is thrust[18],

$$R = k_1\dot{s}^2 = mk\dot{s}^2 \tag{5}$$

is the total (i.e. hydrodynamic plus aerodynamic) resistance, and

$$v_a = \sqrt{(\dot{s}\sin\beta_t)^2 + (v_t + \dot{s}\cos\beta_t)^2} \tag{6}$$

$$\beta_a = \beta_t - \arccos\left(\frac{v_t + \dot{s}\cos\beta_t}{\sqrt{(\dot{s}\sin\beta_t)^2 + (v_t + \dot{s}\cos\beta_t)^2}}\right) \tag{7}$$

are the apparent wind velocity and angle, respectively.

After dividing both sides of Eq.(3) by m and making use of Eqs.(4) and (5), for the i-th (linear) segment of the trajectory, we get

$$\ddot{s}^{(i)} + k\left(\dot{s}^{(i)}\right)^2 = \left(c_1\beta_a^{(i)} - c_2\right)\left(v_a^{(i)}\right)^2 \tag{8}$$

If we introduce

$$\boldsymbol{x}^{(i)} = \left(x_1^{(i)},\ x_2^{(i)}\right)^\top = \left(s^{(i)},\ \dot{s}^{(i)}\right)^\top \tag{9}$$

as the vector of the system state variables at time t, and

$$\boldsymbol{u}^{(i)} = \left(u_1^{(i)}\right) = \left(\beta_t^{(i)}\right) \tag{10}$$

as the vector of the system control inputs at time t, then the system can be described by the following two first-order differential equations:

$$\dot{\boldsymbol{x}}^{(i)}(t) = \boldsymbol{a}^{(i)}\left(\boldsymbol{x}^{(i)}(t),\ \boldsymbol{u}^{(i)}(t)\right), \tag{11}$$

where

$$\boldsymbol{a}^{(i)} = \begin{pmatrix} x_2^{(i)} \\ \left(c_1\beta_a^{(i)} - c_2\right)\left(v_a^{(i)}\right)^2 - k(x_2^{(i)})^2 \end{pmatrix}. \tag{12}$$

[17]Single-dotted and double-dotted values represent the first and second derivatives with respect to time.

[18]Approximated making use of experimental data taken from [20].

Boundary conditions. The first segment of the trajectory starts at point A and the last ends at B (see Fig. 1). The boundary conditions have to be written for each segment s of the (piecewise-linear approximated) trajectory taking into account angle δ representing the local value of the trajectory curvature as shown in Figure 10 (see also [10]).

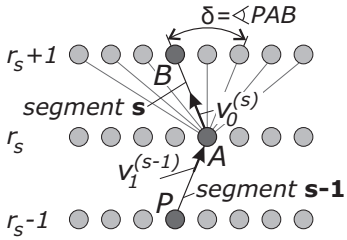

Figure 10. Angle δ representing the local value of the trajectory curvature.

The initial speed $|v_0^{(s)}|$ (i.e., the length of the velocity vector) for segment s is calculated in the following way:[19]

$$|v_0^{(s)}| = \begin{cases} |v_1^{(s-1)}| & \text{if } cos\delta \geqslant U_m, \\ |v_1^{(s-1)}| cos^4\delta & \text{if } L_m < cos\delta < U_m, \\ 0 & \text{otherwise.} \end{cases} \tag{13}$$

where $|v_1^{(s-1)}|$ is the final speed for segment $(s-1)$, U_m represents an arbitrarily chosen upper margin (assumed in experiments to be 0.98) and L_m stands for the corresponding lower margin (assumed in experiments to be 0.5). For the segment starting at point A, we assume that $|v_0^{(1)}| = 0$.

Performance measure. The analyzed optimization task is an example of a minimum-time problem – the optimal trajectory is the one for which the corresponding value of t_f (i.e., the time to reach point B) is minimal. For such problems (see Eq.1)

$$h\big(\boldsymbol{x}(t_f), t_f\big) = 0, \quad g\big(\boldsymbol{x}(t),\ \boldsymbol{u}(t),\ t\big) = 1, \tag{14}$$

so assuming that $t_0 = 0$ and taking into account the segmentation of the trajectory, we can rewrite the performance measure formula (Eq.1) as follows:

$$J = t_f = \sum_s t_f^{(s)}, \tag{15}$$

which means that, in order to find the time of motion from point A to point B, a series of simulations (one for each segment – s) have to be performed.

[19] To improve the accuracy of the model one should consider other approaches, see for instance [21] or [24].

References

[1] Arora N., Russell R.P., Vuduc R.W.: Fast Sensitivity Computations for Trajectory Optimization. *Advances in the Astronautical Sciences*, vol. 135(1), pp. 545–560, 2009.

[2] Bellman R.: The theory of dynamic programming. *Buletin of the American Mathematical Society*, vol. 60, pp. 503–515, 1954.

[3] Bellman R.: On a Routing Problem. *Quarterly of Applied Mathematics*, vol. 16, pp. 87–90, 1958.

[4] Betts J.T.: Survey of Numerical Methods for Trajectory Optimization. *Journal of Guidance Control and Dynamics*, vol. 21(2), pp. 193–207, 1998.

[5] Byrski A., Dębski R., Kisiel-Dorohinicki M.: Agent-based computing in an augmented cloud environment. *Computer Systems Science and Engineering*, vol. 27(1), pp. 7–18, 2012.

[6] Ceriotti M., Vasile M.: MGA trajectory planning with an ACO-inspired algorithm. *Acta Astronautica*, vol. 67(9–10), pp. 1202–1217, 2010.

[7] Crauser A., Mehlhorn K., Meyer U., Sanders P.: A parallelization of Dijkstra's shortest path algorithm. In: L. Brim, J. Gruska, J. Zlatuka, eds., *Mathematical Foundations of Computer Science 1998*, *Lecture Notes in Computer Science*, vol. 1450, pp. 722–731, Springer–Berlin–Heidelberg, 1998.

[8] Dalang R.C., Dumas F., Sardy S., Morgenthaler S., Vila J.: Stochastic optimization of sailing trajectories in an upwind regatta. *Journal of the Operational Research Society*, 2014.

[9] Dębski R.: Gradient-Based Algorithms in the Brachistochrone Problem Having a Black-Box Represented Mathematical Model. *Journal of Telecommunications & Information Technology*, vol. 2014(1), pp. 32–40, 2014.

[10] Dębski R.: High-Performance Simulation-Based Algorithms for Alpine Ski Racer's Trajectory Optimization in Heterogeneous Computer Systems. *International Journal of Applied Mathematics and Computer Science*, vol. 24(3), pp. 551–566, 2014.

[11] Dębski R., Byrski A., Kisiel-Dorohinicki M.: Towards an agent-based augmented cloud. *Journal of Telecommunications and Information Technology*, vol. 1, pp. 16–22, 2012.

[12] Dębski R., Krupa T., Majewski P.: ComcuteJS: A Web Browser Based Platform for Large-scale Computations. *Computer Science*, vol. 14(1), pp. 143–152, 2013.

[13] Dijkstra E.W.: A note on two problems in connexion with graphs. *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[14] Dramski M.: A comparison between Dijkstra algorithm and simplified ant colony optimization in navigation. *Zeszyty Naukowe / Akademia Morska w Szczecinie*, vol. 29 (101), pp. 25–29, 2012.

[15] Dussault J.P.: Solving trajectory optimization problems via nonlinear programming: the brachistochrone case study. *Optimization and Engineering*, vol. 15(4), pp. 819–835, 2014.

[16] Gaster B.R., Howes L.W., Kaeli D.R., Mistry P., Schaa D.: *Heterogeneous Computing with OpenCL – Revised OpenCL 1.2 Edition*. Morgan Kaufmann, 2013.

[17] Harish P., Narayanan P.: Accelerating large graph algorithms on the GPU using CUDA. In: *High performance computing – HiPC 2007*, pp. 197–208, Springer, 2007.

[18] Jasika N., Alispahic N., Elma A., Ilvana K., Elma L., Nosovic N.: Dijkstra's shortest path algorithm serial and parallel execution performance analysis. In: *MIPRO, 2012 Proceedings of the 35th International Convention*, pp. 1811–1815, IEEE, 2012.

[19] Lewis R.M., Torczon V., Trosset M.W.: Direct Search Methods: Then And Now. *Journal of Computational and Applied Mathematics*, vol. 124, pp. 191–207, 2000.

[20] Marchaj C.: *Aero-hydrodynamics of sailing*. Dodd, Mead, 1980.

[21] Marchaj C.: *Sail Performance: Techniques to Maximize Sail Power*. International Marine/McGraw-Hill, 2003.

[22] Park C., Pan J., Manocha D.: Real-time optimization-based planning in dynamic environments using GPUs. In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pp. 4090–4097, IEEE, 2013.

[23] Pêtres C., Romero-Ramirez M.A., Plumet F.: Reactive path planning for autonomous sailboat. In: *2011 15th International Conference on Advanced Robotics (ICAR 2011)*, pp. 112–117, IEEE, 2011.

[24] Philpott A., Henderson S., Teirney D.: A Simulation Model for Predicting Yacht Match Race Outcomes. *Operation Research*, vol. 52(1), pp. 1–16, 2004.

[25] Philpott A., Mason A.: Optimising yacht routes under uncertainty. In: *The 15th Cheasapeake Sailing Yacht Symposium*, 2001.

[26] Pontryagin L.S., Boltyanski V.G., Gamkrelidze R.V., Mischenko E.F.: The Mathematical Theory of Optimal Processes. *Interscience, NY*, 1962.

[27] Pošík P., Huyer W.: Restarted local search algorithms for continuous black box optimization. *Evolutionary Computation*, vol. 20(4), pp. 575–607, 2012.

[28] Pošík P., Huyer W., Pál L.: A Comparison of Global Search Algorithms for Continuous Black Box Optimization. *Evolutionary Computation*, pp. 1–32, 2012.

[29] Rippel E., Bar-Gill A., Shimkin N.: Fast graph-search algorithms for general aviation flight trajectory generation. *Journal of Guidance, Control, and Dynamics*, vol. 28(4), pp. 801–811, 2005.

[30] Singla G., Tiwari A., Singh D.P.: New Approach for Graph Algorithms on GPU using CUDA. *International Journal of Computer Applications*, vol. 72(18), pp. 38–42, 2013, published by Foundation of Computer Science, New York, USA.

[31] Stelzer R., Pröll T.: Autonomous sailboat navigation for short course racing. *Robotics and autonomous systems*, vol. 56(7), pp. 604–614, 2008.

[32] Stillwell J.: *Mathematics and its History*. Springer, third ed., 2010.

[33] Stryk von O., Bulirsch R.: Direct and indirect methods for trajectory optimization. *Annals of Operations Research*, vol. 37(1), pp. 357–373, 1992.

[34] Sussmann H.J., Willems J.C.: 300 years of optimal control: from the brachystochrone to the maximum principle. *Control Systems, IEEE*, vol. 17(3), pp. 32–44, 1997.

[35] Sussmann H.J., Willems J.C.: The brachistochrone problem and modern control theory. *Contemporary trends in nonlinear geometric control theory and its applications (Mexico City, 2000)*, pp. 113–166, 2002.

[36] Szłapczyński J.: Customized crossover in evolutionary sets of safe ship trajectories. *Int. J. Appl. Math. Comput. Sci*, vol. 22(4), pp. 999–1009, 2012.

[37] Szynkiewicz W., Błaszczyk J.: Optimization-based approach to path planning for closed chain robot systems. *International Journal Applied Mathematics and Computer and Science*, vol. 21(4), pp. 659–670, 2011.

[38] Vasile M., Locatelli M.: A hybrid multiagent approach for global trajectory optimization. *Journal of Global Optimization*, vol. 44(4), pp. 461–479, 2009.

[39] Wagner S., Kaplinger B., Wie B.: GPU Accelerated Genetic Algorithm for Multiple Gravity-Assist and Impulsive V Maneuvers. In: *AIAA/AAS Guidance Navigation and Control Conference AIAA*, vol. 4592, p. 2012, 2012.

Affiliations

Roman Dębski
Department of Computer Science, AGH University of Science and Technology, Krakow, Poland, e-mail: `rdebski@agh.edu.pl`

Received: 12.01.2016
Revised: 2.06.2016
Accepted: 2.06.2016