Int. J. Appl. Math. Comput. Sci., *2016*, *Vol. 26*, *No. 2*, *351–365*
DOI: 10.1515/amcs-2016-0025

AN ADAPTIVE MULTI–SPLINE REFINEMENT ALGORITHM IN SIMULATION BASED SAILBOAT TRAJECTORY OPTIMIZATION USING ONBOARD MULTI–CORE COMPUTER SYSTEMS

ROMAN DĘBSKI [a]

[a]Department of Computer Science
AGH University of Science and Technology, Al. Mickiewicza 30, 30-059 Kraków, Poland
e-mail: `rdebski@agh.edu.pl`

A new dynamic programming based parallel algorithm adapted to on-board heterogeneous computers for simulation based trajectory optimization is studied in the context of “high-performance sailing”. The algorithm uses a new discrete space of continuously differentiable functions called the *multi-splines* as its search space representation. A basic version of the algorithm is presented in detail (pseudo-code, time and space complexity, search space auto-adaptation properties). Possible extensions of the basic algorithm are also described. The presented experimental results show that contemporary heterogeneous on-board computers can be effectively used for solving simulation based trajectory optimization problems. These computers can be considered micro high performance computing (HPC) platforms—they offer high performance while remaining energy and cost efficient. The simulation based approach can potentially give highly accurate results since the mathematical model that the simulator is built upon may be as complex as required. The approach described is applicable to many trajectory optimization problems due to its black-box represented performance measure and use of OpenCL.

Keywords: dynamic programming, black-box optimization, heterogeneous computing, micro HPC platform, cubic Hermite splines.

1. Introduction

Trajectory optimization is an important issue in the fields of robotics, aerospace engineering and optimal control. In most cases, however, it cannot be solved analytically because the corresponding mathematical model is too complex and, as a result, the optimization process has to be simulation based.[1] This approach allow us to obtain highly accurate results but—since the cost (time complexity) of a single simulation is often significant—also requires high computing power.

In most on-board (or embedded) systems, like sailboat or autonomous robot trajectory planners, a typical, cluster based HPC platform obviously cannot be used. A common way of addressing this issue is a simplification—sometimes radical, depending on the target platform capabilities—of the mathematical model, which can lead to very rough approximations of optimal trajectories (unacceptable in many situations).

[1] Since derivative related information is not available, meta-heuristics (like evolutionary or swarm intelligence based algorithms) are often used.

The aim of this paper is to present an alternative approach to the simulation based trajectory planning process, using contemporary onboard/mobile computers. The approach is very general both from the deployment point of view and because of the scope of the optimization problems it covers. The main contributions of this paper are the following:

- the concept of a *multi-spline*—a special discrete space of C^1-continuous functions, which can represent a solution space in many variational problems (Sections 4.2 and 4.1);
- the algorithm for simulation based trajectory optimization, built upon the concept of the *multi-spline*, massively parallel, and adapted to on-board heterogeneous computer systems (Sections 4.4, 4.3 and 4.5);
- experimental results which show that contemporary heterogeneous on-board computers can be treated as

micro HPC platforms—they offer high performance while remaining energy and cost efficient (Section 5).

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. Next, the proposed algorithm is described, and some remarks about augmenting the algorithm are also given. After that, experimental results are presented and discussed. The last section contains the conclusion of the study.

2. Related work

Johan Bernoulli originated the brachistochrone problem[2]—generally regarded as the first scientific formulation of the problem of trajectory optimization—in 1696. One of its first solutions, by Johan's brother Jakob, significantly contributed to the development of the calculus of variations (see, e.g., Stillwell, 2010)—the field of mathematics which played a substantial part in trajectory optimization for the next 250 years.

The 1950s saw the real breakthrough in this field with the development of the digital computer and introduction of dynamic programming (Bellman, 1954), effective shortest path algorithms (Bellman, 1958; Dijkstra, 1959) and the Pontryagin maximum principle (Pontryagin *et al.*, 1962). Together with non-linear programming (NLP), these have become the foundations for many effective trajectory optimization methods which are commonly classified as either *direct* or *indirect* (von Stryk and Bulirsch, 1992; Betts, 1998; Lewis *et al.*, 2000; Szynkiewicz and Błaszczyk, 2011). Trajectory optimization methods based on the shortest path algorithm (see, e.g., Crauser *et al.*, 1998; Bertsekas, 2000; Rippel *et al.*, 2005) can be considered a special case of the first approach.

A special kind of trajectory optimization problems are those having *black-box represented* performance measures. A typical example of this is when the performance measure values are received from computer simulation (see, e.g., Dębski, 2014a). In such an instance, most classic optimization methods cannot be used (at least not directly) and the optimization process is often based on *soft-computing/AI* methods (Vasile and Locatelli, 2009; Ceriotti and Vasile, 2010; Pošík *et al.*, 2012; Szłapczyński and Szłapczyńska, 2012; Zamuda and Sosa, 2014; Sun and Wu, 2011; Ćurković *et al.*, 2009; Li and Lü, 2014; Bai *et al.*, 2012; Kojic *et al.*, 2013; Zhou *et al.*, 2011).

Many existing trajectory optimization case studies are related to robotics or aerospace engineering (see, e.g., Rippel *et al.*, 2005; Krozel *et al.*, 2006; Ceriotti and Vasile, 2010), but there are also others (see, e.g., Dębski, 2014b), and a number of those address (autonomous) sailboat trajectory planing (Philpott and Mason, 2001; Philpott *et al.*, 2004; Böttner, 2007; Stelzer and Pröll, 2008; Pêtres *et al.*, 2011; Dalang *et al.*, 2015).

[2] Discussed in a broad sense by Sussmann and Willems (1997).

Another important research area in the context of this paper is related to the parallelization of trajectory optimization (and graph) algorithms (Crauser *et al.*, 1998; Jasika *et al.*, 2012) including the possibility of their GPU-acceleration (see, e.g., Harish and Narayanan, 2007; Arora *et al.*, 2009; Wagner *et al.*, 2012; Singla *et al.*, 2013; Park *et al.*, 2013; Dębski, 2014b).

3. Problem formulation

Consider a sailboat going from point $A(q_A, y_A)$ to $B(q_B, y_B)$, where (q_i, y_i) are the coordinates of the corresponding point in either the Cartesian or polar system. We assume that the true wind does not change significantly in time and therefore can be expressed by the following static vector field (see Fig.1):

$$\boldsymbol{v}_t(q,y) = M(q,y)\,\hat{\boldsymbol{q}} + N(q,y)\,\hat{\boldsymbol{y}}, \tag{1}$$

where $M(q,y)$, $N(q,y)$ are scalar functions, and $\hat{\boldsymbol{q}}$, $\hat{\boldsymbol{y}}$ are unit vectors representing the axes of the corresponding coordinate system.

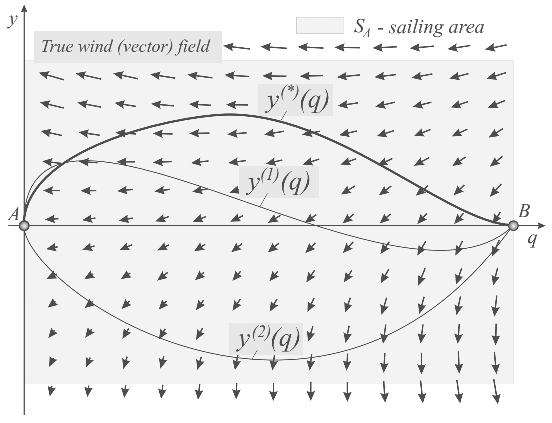

Fig. 1. Sailboat trajectory optimization problem: example admissible trajectories connecting points A and B, with $y^{(*)}(q)$ representing the optimal trajectory; the true wind vector field: $\boldsymbol{v}_t(q,y) = M(q,y)\,\hat{\boldsymbol{q}} + N(q,y)\,\hat{\boldsymbol{y}}$.

The set of admissible $\widetilde{AB}$ trajectories (i.e., the problem domain) consists of C^1-continuous functions which cover the given sailing area S_A (see Fig.1) and do not violate the constraints "embedded" in a sailboat model (these constraints can be related to the state and/or control variables). This model is used to evaluate each trajectory ($y^{(i)}$) through simulation; therefore,

$$J[y^{(i)}] = \mathit{SimulationFor}[y^{(i)}, \mathit{config}\,(\boldsymbol{v}_t, \ldots)], \tag{2}$$

where J represents the given performance measure and $\mathit{config}\,(\boldsymbol{v}_t, \ldots)$ is the simulator configuration.

The sailboat trajectory optimization problem under consideration is *to find, among all admissible trajectories, the one with the best value of the performance measure J*. The explicit formula of the performance measure is unknown, i.e., it is "opaque" (or black-boxed) to the optimization routine. In a special case, when

$$J[y^{(i)}] = \Delta t[y^{(i)}], \tag{3}$$

where Δt is a time interval (duration), we get the *minimum-time problem*.

4. Proposed algorithm

The approach proposed in this paper is based on the following two main steps:

1. transformation (using a grid based discretization scheme) of the continuous optimization problem into a search problem over a specially constructed finite graph; in this graph, vertices correspond to the grid nodes, and edges to pieces of the trajectory (see Sections 4.1, 4.2, 4.3),
2. application of dynamic programming to find the approximation of the optimal trajectory represented as a piecewise-defined function of class C^k, where $k \geq 1$ (see Sections 4.4, 4.5).

These two steps can be repeated several times—the next stage mesh (grid) can be generated through mesh refinement making use of the best trajectory found so far (see Fig. 5). This can be considered a form of *iterative improvement algorithm*.

4.1. Trajectory segments representation. In direct method based trajectory optimization—usually used when the process is simulation based—it is often necessary for candidate solutions to be *smooth*, and therefore a piecewise linear approximation (i.e., C^0-continuous) of the trajectory cannot be applied. In such cases, C^1 continuity (i.e., the first derivative is continuous) is usually sufficient, which means that we can represent trajectories as *cubic Hermite splines*. Each segment of such a spline is a third-degree polynomial specified by its values and the first derivatives at the end points of the corresponding interval (see Fig.2). Thus, we can think of each segment as having two degrees of freedom at both its ends, which makes it very "flexible" in the process of trajectory shaping.

If we assume the following representation of the segment[3] shown in Fig. 2:

$$s_H^{(i)}(q) = d^{(i)} + c^{(i)}q + b^{(i)}q^2 + a^{(i)}q^3, \tag{4}$$

[3] The i-th segment of a cubic Hermite spline; H stands for "Hermite".

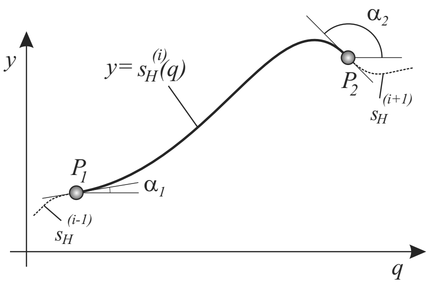

Fig. 2. i-th segment—connecting points $P_1(q_1, s_H^{(i)}(q_1))$ and $P_2(q_2, s_H^{(i)}(q_2))$—of the cubic Hermite spline $s_H^{(i)}(q)$; α_1 and α_2 represent slopes of tangents to $s_H^{(i)}(q)$ at points P_1 and P_2, respectively.

then, after expressing its boundary conditions as

$$\begin{cases} s_H^{(i)}(q_1) = y_1^{(i)}, \\ \dfrac{\mathrm{d}s_H^{(i)}}{\mathrm{d}q}(q_1) = \tan(\alpha_1) = \beta_1^{(i)}, \\ s_H^{(i)}(q_2) = y_2^{(i)}, \\ \dfrac{\mathrm{d}s_H^{(i)}}{\mathrm{d}q}(q_2) = \tan(\alpha_2) = \beta_2^{(i)}, \end{cases} \tag{5}$$

and solving the corresponding set of linear equations,

$$\begin{pmatrix} 1 & q_1^{(i)} & (q_1^{(i)})^2 & (q_1^{(i)})^3 \\ 0 & 1 & 2q_1^{(i)} & 3(q_1^{(i)})^2 \\ 1 & q_2^{(i)} & (q_2^{(i)})^2 & (q_2^{(i)})^3 \\ 0 & 1 & 2q_2^{(i)} & 3(q_2^{(i)})^2 \end{pmatrix} \begin{pmatrix} d^{(i)} \\ c^{(i)} \\ b^{(i)} \\ a^{(i)} \end{pmatrix} = \begin{pmatrix} y_1^{(i)} \\ \beta_1^{(i)} \\ y_2^{(i)} \\ \beta_2^{(i)} \end{pmatrix}, \tag{6}$$

we get

$$\begin{cases} a^{(i)} = \dfrac{(\beta_1^{(i)} + \beta_2^{(i)})(q_2^{(i)} - q_1^{(i)}) - 2(y_2^{(i)} - y_1^{(i)})}{(q_2^{(i)} - q_1^{(i)})^3}, \\ b^{(i)} = \dfrac{y_2^{(i)} - y_1^{(i)} - \beta_1^{(i)}(q_2^{(i)} - q_1^{(i)})}{(q_2^{(i)} - q_1^{(i)})^2} \\ \qquad - \dfrac{(q_2^{(i)})^2 - 2(q_1^{(i)})^2 + q_1^{(i)}q_2^{(i)}}{q_2^{(i)} - q_1^{(i)}}\, a^{(i)}, \\ c^{(i)} = \beta_1^{(i)} - q_1^{(i)}(2b^{(i)} + 3q_1^{(i)}a^{(i)}), \\ d^{(i)} = y_1^{(i)} - q_1^{(i)}\left[c^{(i)} + q_1^{(i)}b^{(i)} + (q_1^{(i)})^2 a^{(i)}\right], \end{cases} \tag{7}$$

which are the coefficients of the cubic Hermite spline segment that connects points $P_1(q_1^{(i)}, y_1^{(i)})$ and $P_2(q_2^{(i)}, y_2^{(i)})$ and has the first derivatives at $q_1^{(i)}$ and $q_2^{(i)}$ equal to $\beta_1^{(i)}$ and $\beta_2^{(i)}$, respectively.

4.2. Multi-spline: The solution space representation. A grid based discretization of the original (continuous) problem domain transforms the trajectory optimization problem into a search problem. An example of such a discretization (grid G) is shown in the upper part of Fig. 3. This grid is based on equidistant nodes, which are grouped

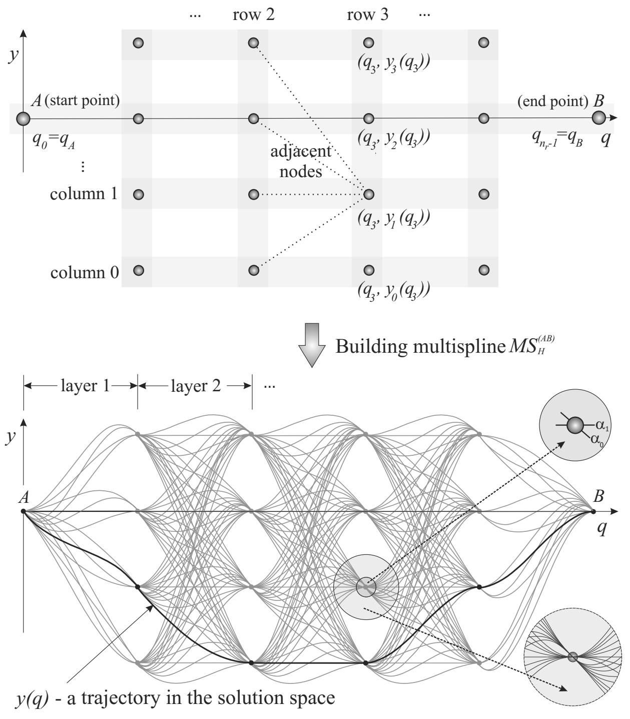

Fig. 3. Solution space representation: multi-spline $MS_H^{(AB)}$ built (spanned) on regular grid G.

in rows and columns: four regular rows plus two special ones—containing the start (A) and end (B) points—and four columns. The total number of nodes in such a grid is equal to

$$|G| = n_c\,(n_r - 2) + 2, \tag{8}$$

where n_c and n_r are the numbers of columns and rows (including the two special ones), respectively.

If we assign n_{ts} additional values to every node of grid G, we obtain a new structure (G_{ex}) called an *extended grid* (corresponding to grid G). The aim of these values is to store (at each node) n_{ts} different slopes (i.e., first derivatives) needed next to calculate the coefficients of cubic Hermite spline segments (see Eqns. (4) and (7)).

Joining the nodes (now treated as vertices) from subsequent rows of G_{ex} with the use of the corresponding cubic Hermite spline segments (treated now as edges), we get a directed graph that represents the new solution space. This collection of spline segments "spanned" on the nodes from subsequent rows of G_{ex} forms a discrete space of C^1-continuous functions called a MULTI-SPLINE (see Fig. 3). It has the following properties:

- its knots (i.e., the places where the spline segments connect) are defined by a special representation of the extended grid G_{ex} (see Fig. 3),
- when seen as a graph (knots are vertices, spline segments are edges), it is directed (from A to B or vice-versa), acyclic and "topologically sorted" (i.e., the edges in layer l are followed by those from layer $l+1$ and the vertices in row r are followed by those from row $r+1$, see Fig. 3),
- it is built from $n_c n_{ts}^2\left[(n_r-3)\,n_c+2\right]$ different spline segments;
- the discrete search space it spans represents $n_{ts}^{n_r} n_c^{n_r-2}$ different trajectories connecting points A and B; this value corresponds to the "inter-row complete" graph (i.e., the one in which all vertices from subsequent rows are connected);
- each of its internal layers (again, in the "inter-row complete" graph) consists of $n_c^2 n_{ts}^2$ spline segments,

where n_r, n_c and n_{ts} are the numbers of rows (including the two special ones), columns and tangent slopes, respectively.

4.3. Multi-spline adaptation.

Since a multi-spline can be built (spanned) on an arbitrary grid (including non-uniform ones), its structure can be adapted/optimized to the domain of the problem under consideration. Therefore, in iterative improvement method based computations, the multi-spline can be optimal at each iteration step (but the initial structure of the multi-spline has to be "guessed").

The adaptation/refinement process can utilize—at every step of the iterative improvement algorithm—the current approximation of the optimal trajectory (i.e., the best trajectory found so far). Two examples of such an approach are presented in Fig. 4. At each iteration step the current approximation of the optimal trajectory is used as the reference for the next step multi-spline. In some cases it can be sufficient to adapt/refine only the vertical positions of nodes (as shown in Fig. 4(a)), but most often this process has to include the tangent slopes as well (see the tangent slopes range reduction and rotation of the reference slope in Fig. 4(b)). In general, as the grid corresponding to G_{ex} can be very irregular, the multi-spline adaptation/refinement process can be highly complex.

The approach used in this paper (simple yet "effective enough" in experimental verification of the proposed algorithm) assumes the following for each iteration step i:

- the numbers of G_{ex} columns (n_c) and tangent slopes at each of its nodes (n_{ts}) are constant, i.e., $n_c(r) = const$ and $n_{ts}(r) = const$, where $r = 1, 2, \ldots, n_r -$

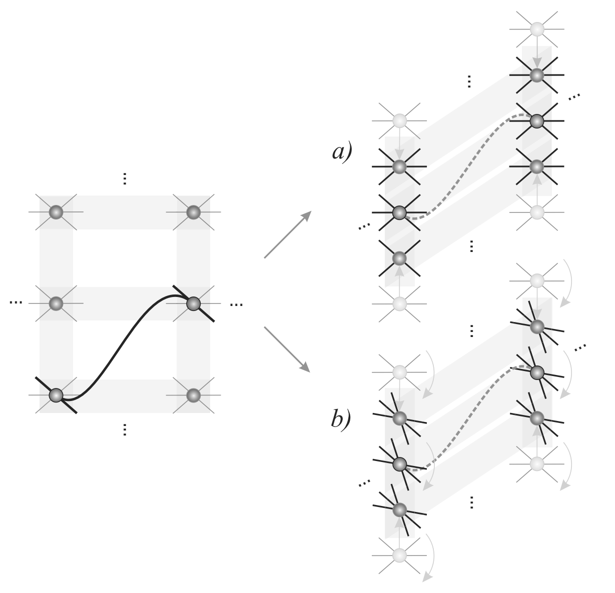

Fig. 4. Multi-spline adaptation in iterative improvement method based computation: (on the left) a segment of the current approximation of the optimal trajectory is used to build the next multi-spline—refining only vertical positions of nodes (a), or both vertical positions of nodes and tangent slopes (b). *Note*: for readability, only one multi-spline segment (the reference one) is drawn.

2 is the row index ($r = 0$ and $r = n_r - 1$ correspond to "special nodes" r_A and r_B, respectively),

- the range of vertical positions of the G_{ex} nodes can change with r,[4]

$$\Delta y_r^{(i)} = |y_{r,n_c-1}^{(i)} - y_{r,0}^{(i)}| \tag{9}$$

- the range of tangent slopes at each G_{ex} node change only with r, i.e., it does not depend on the column index c,

$$\Delta \alpha_r^{(i)} = |\alpha_{r,0,n_{ts}-1}^{(i)} - \alpha_{r,0,0}^{(i)}|; \tag{10}$$

- the refined ranges of both vertical positions of nodes and tangent slopes form geometrical sequences, i.e.,

$$\Delta y_r^{(i)} = k_y \, \Delta y_r^{(i-1)}, \quad 0 < k_y < 1, \tag{11}$$

$$\Delta \alpha_r^{(i)} = k_\alpha \Delta \alpha_r^{(i-1)}, \quad 0 < k_\alpha < 1; \tag{12}$$

- the current (i-th) vertical positions of nodes are calculated in the following way:

$$y_{r,c}^{(i)} = \widetilde{y}_r^{(i-1)} - k_y \frac{\Delta y_r^{(i-1)}}{n_c - 1} \left(\left\lceil \frac{n_c - 1}{2} \right\rceil - c \right), \tag{13}$$

where $r = 1, 2, \dots, n_r - 2$, $c = 0, 1, \dots, n_c - 1$ and $\widetilde{y}_r^{(i-1)}$ is the vertical position of the reference node for row r (the corresponding node from the current approximation of the optimal trajectory);

- the current (i-th) tangent slopes are calculated in the following way (they depend only on r):

$$\alpha_{r,c,s}^{(i)} = \widetilde{\alpha}_r^{(i-1)} - k_\alpha \frac{\Delta \alpha_r^{(i-1)}}{n_{ts} - 1} \left(\left\lceil \frac{n_{ts} - 1}{2} \right\rceil - s \right), \tag{14}$$

where r and c are as defined above, $s = 0, 2, \dots, n_{ts} - 1$ and $\widetilde{\alpha}_r^{(i-1)}$ is the reference tangent slope for row r (from the current approximation of the optimal trajectory). *Note*: the reference tangent slope can be also calculated as the central difference approximation of the first derivative:

$$\alpha_r^{(i)} = \operatorname{atan}\left(\frac{\widetilde{y}_{r+1}^{(i-1)} - \widetilde{y}_{r-1}^{(i-1)}}{\widetilde{q}_{r+1}^{(i-1)} - \widetilde{q}_{r-1}^{(i-1)}} \right). \tag{15}$$

Example results of this approach applied to two different multi-splines are presented in Fig. 5. These instances differ only in the number of tangent slopes of the underlying extended graph G_{ex} ($n_{ts} = 2$ vs. $n_{ts} = 4$); in both the cases, G_{ex} has $n_r = 6$ rows and $n_c = 8$ columns. The figure is organized as a 2×3 array, with each row showing two adaptation/refinement steps of the corresponding multi-spline: the upper one for $n_{ts} = 2$ and the lower one for $n_{ts} = 4$. Each adaptation step is based on the current approximation of the optimal trajectory, shown as a broad line in Fig. 5. The case for $n_{ts} = 2$ was selected just for two reasons: to obtain a figure with visible multi-spline segments[5] and, more importantly, to demonstrate the effect when the value of n_{ts} is too low.

4.4. Basic version of the algorithm. As discussed in Section 4.2, a multi-spline can be seen both as a discrete space of C^1-continuous functions and a special graph. This graph is directed, acyclic (DAG) and has a layered structure. In this new solution space the original continuous trajectory optimization problem may be expressed as an optimal path search problem and solved using dynamic programming.

At the beginning of the search/optimization process there is no *cost matrix*. The cost of each path in the graph is obtained from simulation using the *principle of optimality* (Bellman and Dreyfus, 1962). This principle can be expressed for an example path A-$N_{r,c,s}$ (see Fig. 6) in the following way:

$$\widetilde{J}_A^{N_{r,c,s}} = \min_{c_j, s_k} \left(\widetilde{J}_A^{N_{r-1,c_j,s_k}} + J_{N_{r-1,c_j,s_k}}^{N_{r,c,s}} \right), \tag{16}$$

[4]Subscripts are used to represent discrete function arguments, so $\Delta y_r^{(i)}$ corresponds to $\Delta y^{(i)}[r]$.

[5]Multi-splines used in real computations would be completely unreadable because of too many segments.

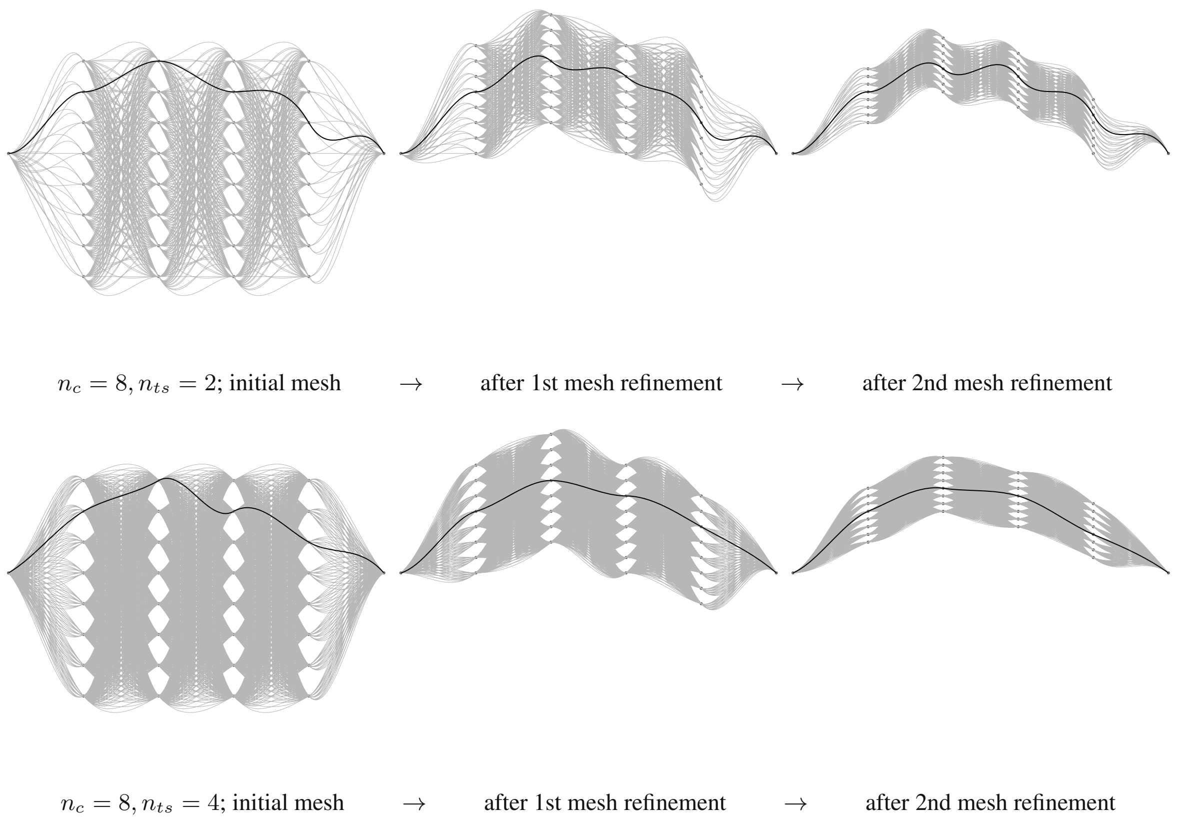

Fig. 5. Two steps of multi-spline (solution space) adaptation for different node (knot) degrees. Note that the convergence in the case of $n_{ts} = 2$ is much worse ("torsional rigid" multi-spline)—each spline segment can start and finish only with two tangent slopes; typically, we should use $5 \leq n_{ts} \leq 9$ (see Section 5).

where $c_j = (0, \dots, n_c - 1)$, $s_k = (0, \dots, n_{ts} - 1)$, $J_{N_s}^{N_e}$ is the cost corresponding to the path N_s-N_e (N_s—start node, N_e—end node), $\widetilde{J}$ represents the optimal value of J and $N_{r,c,s}$ is the node of G_{ex} with "graph coordinates" $(row, column, tangent\_slope) = (r, c, s)$.

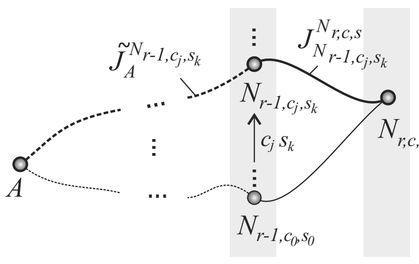
$\widetilde{J}_A^{N_{r\text{-}1,c_j,s_k}}$
$J_{N_{r\text{-}1,c_j,s_k}}^{N_{r,c,s}}$
$N_{r\text{-}1,c_j,s_k}$
$c_j\ s_k$
$N_{r,c,s}$
A
$N_{r\text{-}1,c_0,s_0}$

Fig. 6. Principle of optimality in dynamic programming: optimal path (trajectory) from point A to point $N_{r,c,s}$ (G_{ex} node); graphical representation of Eqn. (16) (with the notation $J_{\text{from}}^{\text{to}}$ and $\widetilde{J}$ representing the optimal J).

Figure 6 is a visualization of Eqn. (16). It presents the computation state in which the optimal costs of reaching all nodes in row $r - 1$ are known (they were calculated in previous stages of this multi-stage process). The optimal cost of path A-$N_{r,c,s}$ is calculated by performing simulations for all spline segments that join node $N_{r,c,s}$, which is located in row r, with nodes from the previous (i.e., $(r - 1)$-th) row. This simulation based multi-stage process can be visualized as a propagation of a "simulation-wave" presented in Fig.7. The computation begins from the start node (point A in layer 1), taking into account the corresponding initial conditions, and is continued (layer by layer) for the nodes in subsequent rows. Completing the simulations for the last layer (i.e., reaching the end node B), we get the optimal path (trajectory) and the corresponding value of the performance measure.

The special (layered) structure of the search space (i.e., multi-spline) and the dynamic programming based search algorithm allow us to organize the computation as a sequence of parallel simulations for multi-spline segments from the same layer (see Fig. 7). The sequential component of the computation—presented in the figure as a synchronization barrier—is a result of the layer-on-layer dependence (to start simulation for a segment, we have to know the corresponding initial conditions; see Eqn. (16)). This approach is presented as Algorithm 1. The pseudo code assumes the target execution platform to be an OpenCL-capable device (or a set of such

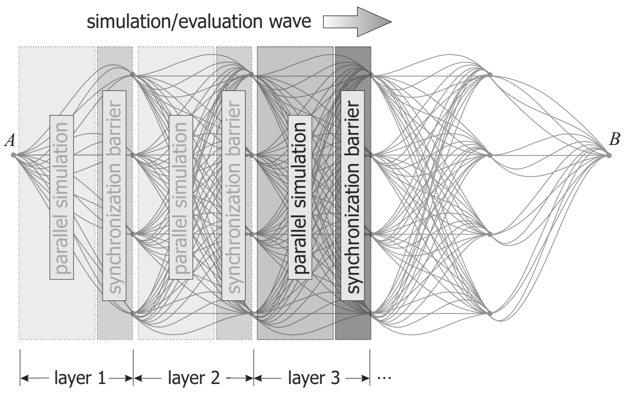

Fig. 7. Sequence of parallel simulations (a *simulation wave*) for multi-spline segments from the same layer; the synchronization barrier is needed due to layer-on-layer dependence (to perform simulation for a single segment, we need to know the initial conditions).

Algorithm 1. Parallel optimal path search.

```
1: {For all multi-spline segments starting at point A and ending at row 1}
2: @PARALLEL $(sim_{first}\_kernel(data_p))$
3: {For all multi-spline segments starting in row r and ending in row r+1}
4: **for all** r in [1..R-2) **do**
5:     @PARALLEL $(sim_{internal}\_kernel(r, data_p))$
6: **end for**
7: {For all multi-spline segments starting at row R-2 and ending at point B}
8: @PARALLEL $(sim_{last}\_kernel(R-2, data_p))$
9: **return** $(opt\_trajectory, performance\_measure)$
```

devices). Functions sim_{first}_kernel, $sim_{internal}\_kernel$ and sim_{last}_kernel are OpenCL kernels, $data_p$ represents memory buffers needed for data transfers between the computing devices, and annotation $@PARALLEL$ denotes[6] SPMD (*single program multiple data*) parts of the computation.

Depending on the size of differences in simulation times and the number of processing elements of the target platform (many-core system), the computation for a single multi-spline layer can be organized in one of the following ways:

- as a single-phase process (with simulation and aggregation interleaved): one processing element performs (sequentially) simulations for all multi-spline segments which end in the same node $N_{r,c,s}$ updating, if necessary, the best solution found so far;

- as a two-phase process (with simulation and aggregation separated): in the first phase, all simulations are performed (in parallel), in the second, the best solution for each node $N_{r,c,s}$ is calculated taking into account solutions for the segments that end in $N_{r,c,s}$ (this can be done in parallel as well).

[6]In pseudo-code only as there is no OpenCL correspondent.

The second approach can be more effective if the simulation times for segments in a given layer are similar (and the number of processing elements is big enough). In other cases, the first, simpler, approach is often more suitable.

If Algorithm 1 is combined with the multi-spline adaptation/refinement (discussed in Section 4.3) it forms an iterative improvement algorithm called *adaptive multi-spline refinement*. The stop condition for this algorithm can be defined in various ways. The first and simplest approach—used in the experimental verification of the algorithm—is to set a fixed number of iterations. This number can be calculated using Eqns. (11) and (12) having assumed the target ranges of y's and tangent slopes. Another possible approach would be to use a form of a derivative based criterion.

4.4.1. Algorithm complexity analysis.

The time complexity of the presented algorithm is determined by the number of its iterations, the simulation duration for a single segment, and the number of such segments (in the multi-spline). If we assume the average duration of this simulation to be $\bar{t}_{sim}^{(seg)}$, then the total duration of all simulations performed n_i times[7] sequentially (T_s) for a multi-spline built from $n_c n_{ts}^2 \left[(n_r-3)\, n_c + 2\right]$ segments (see Section 4.2) is proportional to

$$T_s \propto n_i\, \bar{t}_{sim}^{(seg)}\, n_c\, n_{ts}^2 \left[(n_r-3)\, n_c + 2\right], \quad n_r > 3, \quad (17)$$

or, using the big-O notation,

$$T_s = O\left(n_i\, \bar{t}_{sim}^{(seg)}\, n_r\, n_c^2\, n_{ts}^2\right). \quad (18)$$

In the single-phase version of Algorithm 1, the simulations for all nodes in a given row can be performed in parallel, thus

$$T_p = O\left(n_i\, \bar{t}_{sim}^{(seg)}\, n_r\, n_c\, n_{ts} \left\lceil \frac{n_c\, n_{ts}}{p} \right\rceil\right). \quad (19)$$

The algorithm space complexity is equal to $\Theta\left(n_r n_c n_{ts}\right)$.

4.5. Possible extensions of the basic algorithm.

In some onboard/embedded computer systems the proposed search space and/or algorithm may need to be slightly modified to meet certain constraints—for instance,

[7]n_i is the number of iterations/refinements in the adaptive multi-spline refinement algorithm.

regarding the time and/or space complexity, or the accuracy of the final solution. Worth consideration are the following:

- reduction in the number of connections between nodes from subsequent rows (cf., e.g., Dębski, 2014b)—this can decrease (significantly) both time and space complexities (fewer segments means fewer simulations to perform and less memory to store segments' data). *Note*: the effectivenesses of this modification is problem-dependent—in some cases we cannot reduce the search space in this way without the danger of losing good solution candidates;
- augmentation of the presented algorithm by a local search—in some cases this can improve the final result significantly since the search space of the local algorithm can be continuous, which means that there is no accuracy limit related to the mesh granularity. *Note*: the local search/optimization can be based on an indirect or direct method (cf., e.g., Dębski, 2014a),
- use of C^2-continuous splines in the final step of the iterative improvement algorithm—3-rd or 5-th order splines can be used. *Note*: the use of 3-rd order splines (with three boundary conditions for the start point and one for the end point) can make the algorithm unstable.

If necessary, these modifications can be combined.

5. Experimental verification

To demonstrate the effectiveness of the *adaptive multi-spline refinement algorithm*, two series of experiments were carried out. The first was related to a simple test problem—formulated in the polar coordinate system—with a known exact solution. The second, on the other hand, was related to a more complex problem formulated in the Cartesian coordinate system. In all experiments, a MacBook Pro[8] with OS X 10.10.3 and OpenCL 1.2, having three OpenCL-capable devices (processor Intel Core i7-3740QM @ 2.7 GHz and two graphics cards—integrated Intel HD Graphic 4000 and discrete (dedicated) nVidia GeForce GT 650m[9]) were used.

Example 1. (*Simple test problem, polar coordinates*) Consider a closed-trajectory simulation based optimization problem formulated in the polar coordinate system (O, r, θ) as shown in Fig. 8. We assume the

[8]With 16 GB of DDR3 1600 MHz RAM.

[9]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.

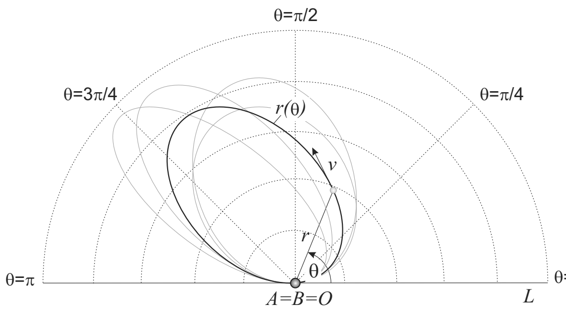

Fig. 8. Domain of the problem considered in Example 1 and several corresponding admissible trajectories.

problem domain (i.e., search space) D_1 to be

$$D_1 = \left\{ s_H^{\widetilde{AB}}(\theta) : \left(C^1 \ni s_H^{\widetilde{AB}} : [0,\pi] \to [0,5] \right) \right\} \tag{20}$$

and the boundary conditions as (see Fig. 8)

$$s_H^{\widetilde{AB}}(0) = s_H^{\widetilde{AB}}(\pi) = 0, \tag{21}$$

where $s_H^{\widetilde{AB}}$ is a cubic Hermite spline with endpoints A and B. The optimization goal is to find, among the admissible trajectories (see Eqn. (20)), the one that minimizes the performance measure, whose formula is unknown to the optimization routine.[10] Since in this problem the performance measure does not depend on a true wind field, it can be assumed that $\boldsymbol{v}_t(r,\theta) = \mathbf{0}$. The details of the simulation model (performance measure) are presented in Appendix A.

Algorithm convergence analysis: The impact of $\boldsymbol{n_{ts}}$. The number of tangent slopes n_{ts} significantly increases both the time and space complexity of the presented algorithm (see Eqn. (19)). At the same time, its impact on the rate of convergence of the algorithm is unclear, and because of that is investigated in this paragraph.

All computational experiments were performed for $k_\alpha = k_y = 0.3333$ and $\Delta\alpha/2 = \pi/5$ (see Eqns. (11), (12)). Having assumed the above values, the number of multi-spline refinements n_i needed to obtain the target inter-node distance δ_q was then calculated from the following formula:

$$n_i = \left\lceil \frac{1}{\ln k_q} \ln \frac{\delta_q (n_c - 1)}{\Delta q} \right\rceil, \tag{22}$$

valid both for $q = y$ and $q = \alpha$. For $n_i = 3$ and $n_c = 64$ the corresponding values of δ_y and δ_α were equal to 0.002939 and 0.003323, respectively.

Figure 9 presents two closed curves which are respectively the first and the fourth approximations of the optimal trajectory for the multi-spline defined by $n_r = 9$, $n_c = 64$ and $n_{ts} = 8$.

[10]The values of the performance measure are obtained from simulation.

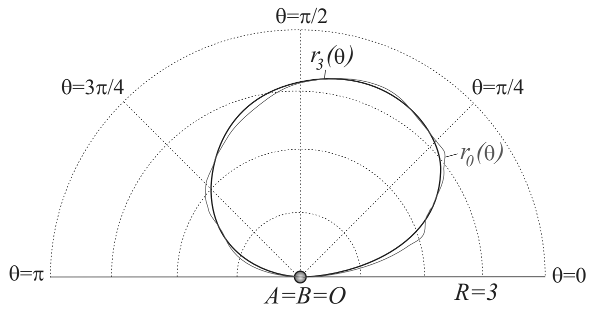

Fig. 9. Iterative improvement through multi-spline refinement in Example 1. Trajectories $r_0(\theta)$ and $r_3(\theta)$ are, respectively, the 1st (i.e., before the 1st multi-spline refinement) and 4th (i.e., after 3 refinements) approximations of the optimal trajectory. Multi-spline with $n_r = 9$, $n_c = 64$ and $n_{ts} = 8$. *Note*: the 4th approximation and the exact solution (i.e., the optimal trajectory) are indistinguishable (see Fig. 10 and Appendix A).

The corresponding values of the performance measure for each iteration are shown in Fig. 10. Two facts

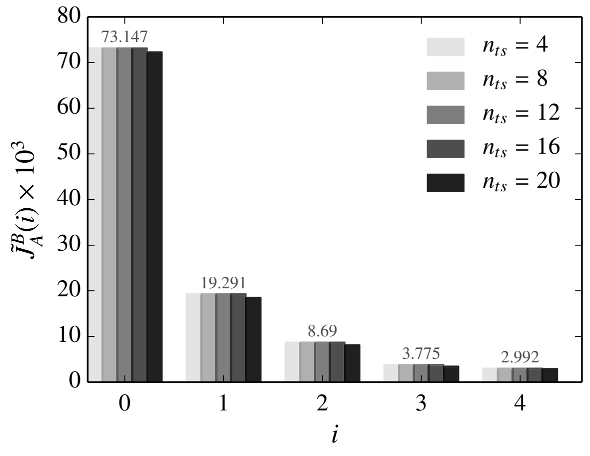

Fig. 10. Values of the performance measure ($\tilde{J}^B_A$) for subsequent refinements of five different multi-splines (defined respectively by $n_{ts} = 4, 8, \ldots, 20$). In all instances, $n_r = 9$ and $n_c = 64$.

are worth noting here:

- the first multi-spline refinement is the key step of the iterative improvement algorithm—in some cases (e.g., in time-consuming simulations) the search process can actually be stopped at this stage;
- the impact of the number of tangent slopes (n_{ts}) on the rate of convergence of the search process is negligible; therefore, the value of this parameter can be safely selected from the range $3 < n_{ts} < 9$. *Note*: this observation can justify regarding n_{ts} as a constant factor in expressions defining the time and space complexity of the algorithm.

Table 1. Average execution times $\bar{t}_{sim}$ (in milliseconds) and standard deviations σ, from eleven runs of the sequential version of the algorithm, for different optimization levels l set in the Apple LLVM 6.1 compiler. The fourth column contains the size s (in bytes) of the corresponding executables. The multi-spline with $n_r = 9$, $n_c = 64$, $n_{ts} = 32$; three multi-spline refinements.

l	$\bar{t}_{sim}$	σ	s	t_0/t_i	s_0/s_i
0	190461	2151	119192	-	-
1	62928	926	115044	3.03	1.04
2	57624	444	114100	3.31	1.04
3	56843	953	113972	3.35	1.05
s	56505	473	109996	3.37	1.08
fast	1460	144	113972	**13.05**	1.05

Table 2. As Table 1 but for the parallel version of the algorithm, different OpenCL devices, and different OpenCL optimization levels.

l	i7-3740QM		HD4000		GT650m		s
	$\bar{t}_{sim}$	σ	$\bar{t}_{sim}$	σ	$\bar{t}_{sim}$	σ	
0	3727	99	2073	34	3641	68	113276
1	3418	73	2063	30	3983	598	109548
2	3413	71	2074	20	3811	213	109948
3	3374	74	2069	31	3815	427	110076
s	13157	15	2075	26	3689	192	109996

Performance analysis: The impact of compiler optimization. Modern compilers (such as Clang/LLVM 6.1 used in the experiments) have optimization abilities which can significantly improve program performance. They often perform optimization at many stages (e.g., compile time, link time). The available optimization options usually allow the programmer to choose whether they prefer a smaller target file size, a faster code or faster build times. In the case of on-board computer systems (or embedded systems) a combination of the first two (i.e., fast and small) is usually the preferred option.

The impact of compiler optimization on the execution times of both sequential and parallel (OpenCL based) versions of the algorithm is summarized in Tables 1 and 2, respectively. Subsequent rows in the tables present results for all optimization levels (l) available in the Apple LLVM 6.1 compiler. The first row in both tables (i.e., for $l = 0$) shows the mean value ($\bar{t}_{sim}$) and the standard deviation (σ) of the execution times of the code compiled with no optimization;[11] additionally, the sizes s of the corresponding executables are also given. There are two special optimization levels available in the Apple LLVM 6.1 compiler: s (or $-Os$ option) and *fast* ($-Ofast$). For the first one (i.e., s-level), the compiler performs all optimizations that do not typically increase

[11]It corresponds to the $-O0$ command-line option.

the target file size (and, in many cases, reduce it). For the second, the compiler performs all possible optimizations, including 'relaxing IEEE compliance' (*-ffast-math* flag) and loop vectorization. The results show that

- non-OpenCL code optimizations can improve the execution time significantly (more than three times[12]);
- OpenCL code optimizations (see Table 2) have only a negligible effect on execution times; in addition, in some cases the optimization may produce very unexpected results (see row s for i7-3740QM in Table 2);
- the recommended optimization levels are s both for an OpenCL and non-OpenCL code (for a non-OpenCL code, whenever possible, $-Ofast$ should be used instead).

OpenCL platforms performance comparison. Contemporary mobile/embedded computers are usually heterogeneous, i.e., they are equipped with more than one type of processor—typically these are (multi-core) CPUs and (many-core) GPUs, but sometimes also others (e.g., FPGAs). OpenCL makes it possible to use these heterogeneous micro HPC platforms effectively since the same code can be executed on any OpenCL-capable processor. Effective processor allocation plans can be calculated before or during program execution (run-time) taking into account the capabilities of each of the available devices and the characteristics of a computation task.

The performance comparison (in the context of Example 1) of three OpenCL platforms is shown in Table 3 and Fig. 11. The results for $n_c < 16$ were omitted because the multi-spline they define is too coarse-grained. It is worth noting, however, that for such cases the sequential version was the fastest.

The performance comparison shows that

- the three platforms differ significantly so there is definitely room for optimization before or during code execution;
- the CPU (i7-3740QM) is capable of efficient execution of the OpenCL code for $n \leq 32$;
- the integrated graphics card (HD4000) is the best in the whole range of n_c;
- because of the high overhead of the host-GPU data transfer, the dedicated graphics card (GT650m) should not be used for $n_c \leq 128$.

♦

[12]In the case of $-Ofast$ even 13 times, but using this optimization level is not always possible because of the reduction in the floating point computations accuracy.

Table 3. OpenCL platforms comparison (Example 1): average execution times $\bar{t}_{sim}$ (in milliseconds) and standard deviations σ from seven runs of the parallel version of the algorithm for different n_c. The multi-spline with $n_r = 9$, $n_{ts} = 8$; no multi-spline refinements.

n_c	i7-3740QM		HD4000		GT650m	
	$\bar{t}_{sim}$	σ	$\bar{t}_{sim}$	σ	$\bar{t}_{sim}$	σ
16	50	1	56	1	489	2
32	92	1	83	3	625	4
64	249	2	149	12	752	1
128	867	6	236	16	1015	2
256	3392	82	590	17	1463	5
512	13061	18	1877	15	3241	154

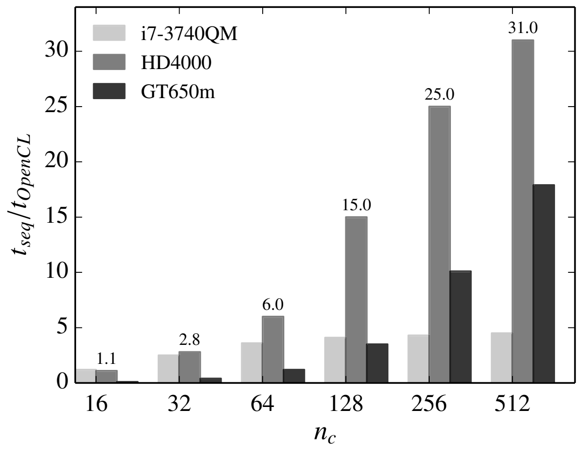

Fig. 11. OpenCL platforms comparison (Example 1): speed-ups for different n_c (t_{seq} measured with the $-Os$ flag). Multi-spline parameters as in Table 3.

Example 2. (*More complex problem, Cartesian coordinates*) Consider a sailboat going upwind from point A to point B with the true wind given by the following vector field (note that the field could be much more complex):

$$\begin{aligned} &\boldsymbol{v}_t(x, y) \\ &= v_0\Big[\Big(\cos\Big(\frac{x}{L}\Big) + \frac{y}{H}\Big)\hat{\boldsymbol{x}} + \Big(\frac{x}{L} - \sin\Big(\frac{y}{L}\Big)\Big)\hat{\boldsymbol{y}}\Big], \end{aligned} \tag{23}$$

shown in Fig. 12. We assume the problem domain (i.e., search space) D_2 to be

$$D_2 = \Big\{ s_H^{\widetilde{AB}}(x) : \Big(C^1 \ni s_H^{\widetilde{AB}} : [0, L] \to [-H, H] \Big) \Big\} \tag{24}$$

and the boundary conditions as

$$s_H^{\widetilde{AB}}(0) = s_H^{\widetilde{AB}}(H) = 0, \tag{25}$$

where $s_H^{\widetilde{AB}}$ is (as before) a cubic Hermite spline with endpoints A and B. The optimization goal is to minimize the performance measure which again is unknown (*black-boxed*) to the optimization routine. In the simulation

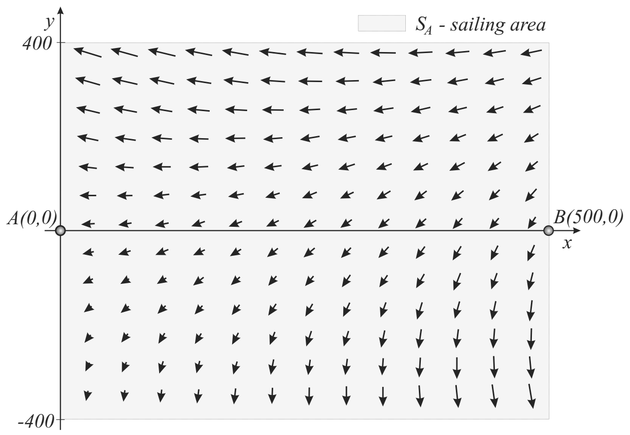

Fig. 12. Domain of the problem considered in Example 2 and the corresponding true wind vector field (defined by Eqn. (23)).

we assume that: $v_0 = -20$, $L = x_B - x_A = 500$ and $H = 0.5\,(y_{\max} - y_{\min}) = 400$ (the details of the simulation model and performance measure are presented in Appendix B).

Algorithm convergence analysis: The impact of n_{ts}. The computational experiments were performed for the same set of parameters as in Example 1, i.e., $k_\alpha = k_y = 0.3333$ and $\Delta\alpha/2 = \pi/5$ (see Eqns. (11), (12)). For $n_i = 3$ and $n_c = 32$ the corresponding values of δ_y and δ_α (see Eqn. (22)) were equal to 0.477754 and 0.003323, respectively.

Figure 13 shows two curves which are, respectively, the first and the fourth approximations of the optimal trajectory. The corresponding values of the performance measure for each iteration are shown in Fig. 14. Finally, Table 4 is the optimal route given in the time domain sampled once a second (i.e., $course(kT)$, $k = 0, 1, 2, \ldots,$ $T = 1\,\mathrm{s}$). The observations from Example 1 are still valid—both regarding the key role of the first multi-spline refinement and the negligible impact of the number of tangent slopes (n_{ts}) on the rate of convergence of the search process. These observations can help us to significantly reduce both the time and space complexity, which is often critical in many on-board/embedded computer systems.

Performance analysis: The impact of compiler optimizations. The execution times of both sequential (non-OpenCL) and parallel (OpenCL based) versions of the algorithm in the context of Example 2 for different optimization levels are summarized in Tables 5 and 6, respectively. Results are similar to those from Example 1—again we see the significant impact of optimizations on the non-OpenCL code and negligible

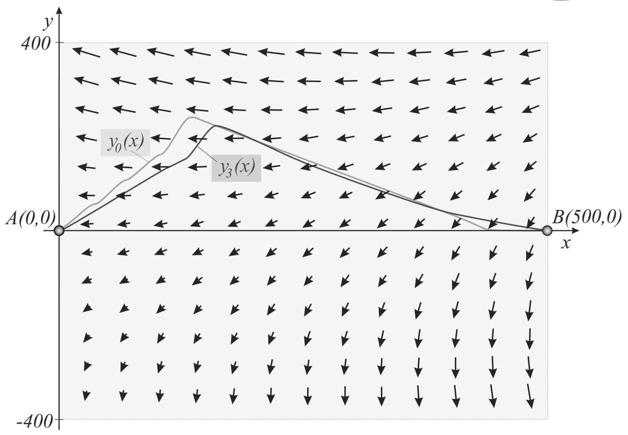

Fig. 13. Iterative improvement through multi-spline refinement in Example 2. Trajectories $y_0(x)$ and $y_3(x)$ are, respectively, the 1st (i.e., before the 1st multi-spline refinement) and 4th (i.e., after 3 refinements) approximations of the optimal trajectory; $n_r = 16$, $n_c = 32$ and $n_{ts} = 8$.

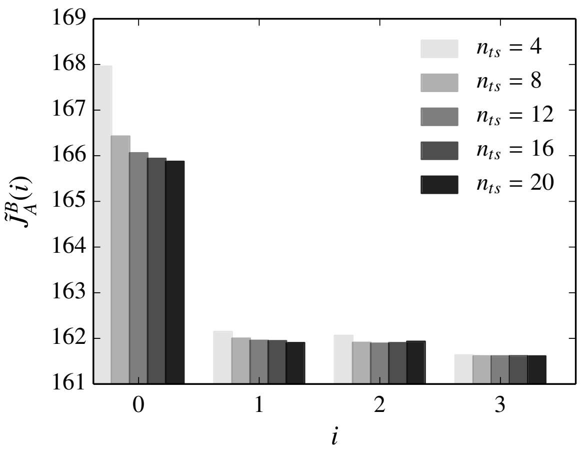

Fig. 14. Values of the performance measure ($\tilde{J}_A^B$) for subsequent refinements of five different multi-splines (defined respectively by $n_{ts} = 4, 8, \ldots, 20$). In all instances, $n_r = 16$ and $n_c = 32$.

impact (if we omit two "unexpected" cases in which run-time errors appeared) in the case of the OpenCL code. It is worth noting, however, that the $-Ofast$ option resulted this time in the corresponding speed-up equal to "only" 4.48 (compare with 13.05 in the previous example[13]) and that the OpenCL code compiled with no optimizations ($-O0$ flag) could not be executed on the HD4000 card due to a run-time error (this error does not appear in the simulator compiled by Apple LLVM 7.0).

[13]This huge change is a result of differences between simulation models used in each example (see Appendices A and B).

Table 4. Fourth approximation of the optimal route (planned trajectory) from point A to point B (see $y_3(x)$ in Fig. 13) in the time domain sampled once a second (i.e., with sampling period $T = 1$ s). *Note*: as the optimal route obtained using the proposed approach is C^1-continuous, the value of the sampling period can be set arbitrarily.

t_i	*course*
0	38.1°
1	38.3°
⋮	⋮
50	44.7°
⋮	⋮
100	130.8°
⋮	⋮
150	110.9°
⋮	⋮
161	107.0°

Table 5. Average execution times $\bar{t}_{sim}$ (in milliseconds) and standard deviations σ, from eleven runs of the sequential version of the algorithm for different optimization levels l set in the Apple LLVM 6.1 compiler. The fourth column contains the size s (in bytes) of the corresponding executables. The multi-spline with $n_r = 16$, $n_c = 32$, $n_{ts} = 8$; three multi-spline refinements.

l	$\bar{t}_{sim}$	σ	s	t_0/t_i	s_0/s_i
0	114751	864	136523	-	-
1	29398	138	124135	3.90	1.10
2	29346	110	127295	3.91	1.07
3	29518	281	127159	3.89	1.07
s	33055	327	123191	3.47	1.11
fast	25597	56	127159	4.48	1.07

Table 6. As Table 5 but for the parallel version of the algorithm, different OpenCL devices, and different OpenCL optimization levels.

l	i7-3740QM		HD4000		GT650m		s
	$\bar{t}_{sim}$	σ	$\bar{t}_{sim}$	σ	$\bar{t}_{sim}$	σ	
0	8701	53	–	–	29983	99	143143
1	8640	78	7619	22	46224	33	123879
2	8664	58	7624	25	46205	12	123127
3	8697	63	7626	21	46625	20	127879
s	10550	31	7627	29	46303	23	123191

OpenCL platforms performance comparison. Execution times of the OpenCL based implementation (run on each of the OpenCL enabled processors) for different values of n_c are shown in Table 7 and Fig. 15. The results are similar to those from Example 1 for the CPU and integrated GPU, but *significantly worse* in the case of the dedicated GPU. In addition, it was not possible to execute the test for $n_c = 128$ on the GT650m card due to a run-time error (this error still appears in the simulator compiled by Apple LLVM 7.0; it is, probably, 'out-of-memory' related). This huge performance loss (when compared with the CPU and HD4000) was due to the way the transcendental functions are processed by the GT650m card. The computation performed by the simulator used in Example 2 is dominated by the evaluation of transcendental functions (see Appendix B). This problem can be addressed by replacing (in the model) all instances of transcendental functions by their approximations (on some platforms using the *native_* versions can be sufficient).

Table 7. OpenCL platforms comparison in Example 2: average execution times $\bar{t}_{sim}$ (in milliseconds) and standard deviations σ from seven runs of the parallel version of the algorithm for different n_c. The multi-spline with $n_r = 16$, $n_{ts} = 8$; no multi-spline refinements.

n_c	i7-3740QM		HD4000		GT650m	
	$\bar{t}_{sim}$	σ	$\bar{t}_{sim}$	σ	$\bar{t}_{sim}$	σ
16	736	19	1363	19	8358	182
32	2514	24	2248	36	14660	13
64	9357	45	4651	17	25926	19
128	36943	1234	7869	22	-	-

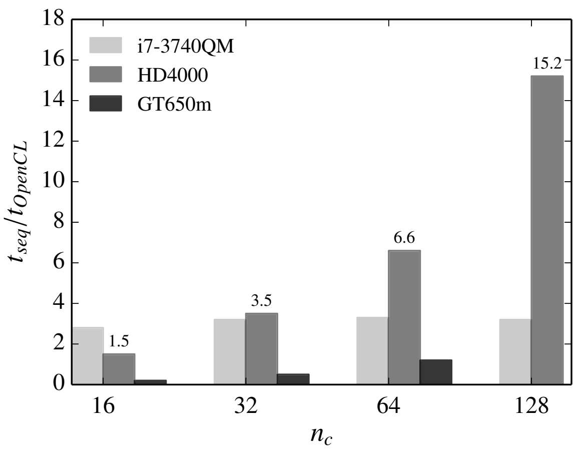

Fig. 15. OpenCL platforms comparison (Example 2): speed-ups for different n_c (t_{seq} measured with the $-Os$ flag). Multi-spline parameters as in Table 7.

♦

6. Conclusion

Adaptive multi-spline refinement—a new dynamic programming based parallel algorithm, adapted to on-board heterogeneous computers, for simulation based trajectory optimization—was studied with reference to two problems taken from "high-performance sailing". The algorithm uses a new discrete space of C^1-continuous functions called the *multi-spline* as its search space representation.

The basic version of the algorithm was presented in detail (pseudo-code, time and space complexity). Its possible extensions and search space (multi-spline) auto-adaptation properties were also described. To demonstrate the effectiveness of the algorithm, two numerical examples were studied (the first for the polar and the second for the Cartesian coordinate system).

The experimental results show that contemporary heterogeneous on-board (or mobile) computers can be effectively used for solving simulation based trajectory optimization problems. These computers can be considered micro HPC platforms—they offer high performance (the effective use of the OpenCL-capable GPU accelerated the optimization routine even up to 31 times) while remaining energy and cost efficient (which is often crucial in many on-board and/or embedded systems).

The simulation based approach can potentially give highly accurate results since the mathematical model that the simulator is based on may be as complex as required. It should be noted, however, that because the problem under consideration is formulated as time-invariant (the true wind is given by a static vector field) the functional scope of the proposed approach is limited to short-term trajectory planning (in the longer term, the wind almost always changes).

Future research work could concentrate on

- verifying the proposed approach with a more accurate sailboat model (taking into account currents and waves as the first step);
- comparing (experimentally) the effectiveness of the proposed approach to alternative ones (e.g., based on meta-heuristics like evolutionary algorithms, simulated annealing, particle swarm optimization);
- researching the proposed extensions of the algorithm;
- improving the algorithm itself (e.g., resource usage optimization, balancing the "simulation load", and more effective mesh-generation);
- studying the algorithm in other variational problems;
- verifying the algorithm in other computing environments (including the *augmented cloud* (Byrski *et al.*, 2012));
- building an optimizing OpenCL kernel scheduler.

Acknowledgment

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.

References

Arora, N., Russell, R.P. and Vuduc, R.W. (2009). Fast sensitivity computations for trajectory optimization, *Advances in the Astronautical Sciences* **135**(1): 545–560.

Bai, J., Chen, L., Jin, H., Chen, R. and Mao, H. (2012). Robot path planning based on random expansion of ant colony optimization, *in* Z. Qian *et al.* (Eds.), *recent Advances in Computer Science and Information Engineering*, Lecture Notes in Electrical Engineering, Vol. 125, Springer, Berlin/Heidelberg, pp. 141–146.

Bellman, R. (1954). The theory of dynamic programming, *Bulletin of the American Mathematical Society* **60**(1954): 503–515.

Bellman, R. (1958). On a routing problem, *Quarterly of Applied Mathematics* **16**(1958): 87–90.

Bellman, R. and Dreyfus, S. (1962). *Applied Dynamic Programming*, Princeton University Press, Princeton, NJ.

Bertsekas, D.P. (2000). *Dynamic Programming and Optimal Control*, 2nd Edn., Athena Scientific, Belmont, MA.

Betts, J.T. (1998). Survey of numerical methods for trajectory optimization, *Journal of Guidance Control and Dynamics* **21**(2): 193–207.

Böttner, C.-U. (2007). Weather routing for ships in degraded conditions, *International Symposium on Safety, Security and Environmental Protection, Athens, Greece*.

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

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

Crauser, A., Mehlhorn, K., Meyer, U. and Sanders, P. (1998). A parallelization of Dijkstra's shortest path algorithm, *in* L. Brim *et al.* (Eds.), *Mathematical Foundations of Computer Science 1998*, Lecture Notes in Computer Science, Vol. 1450, Springer, Berlin/Heidelberg, pp. 722–731.

Dalang, R.C., Dumas, F., Sardy, S., Morgenthaler, S. and Vila, J. (2015). Stochastic optimization of sailing trajectories in an upwind regatta, *Journal of the Operational Research Society* **66**(5): 807–821.

Dębski, R. (2014a). Gradient-based algorithms in the brachistochrone problem having a black-box represented mathematical model, *Journal of Telecommunications & Information Technology* **2014**(1): 32–40.

Dębski, R. (2014b). High-performance simulation-based algorithms for an alpine ski racer's trajectory optimization in heterogeneous computer systems, *International Journal of Applied Mathematics and Computer Science* **24**(3): 551–566, DOI: 10.2478/amcs-2014-0040.

Dijkstra, E.W. (1959). A note on two problems in connexion with graphs, *Numerische Mathematik* **1**(1): 269–271.

Harish, P. and Narayanan, P. (2007). Accelerating large graph algorithms on the GPU using CUDA, *in* S. Aluru *et al.* (Eds.), *High Performance Computing*, Springer, Berlin/Heidelberg, pp. 197–208.

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

Kojic, N., Reljin, I. and Reljin, B. (2013). Route selection problem based on Hopfield neural network, *Radioengineering* **22**(4): 1182–1193.

Krozel, J., Lee, C. and Mitchell, J.S. (2006). Turn-constrained route planning for avoiding hazardous weather, *Air Traffic Control Quarterly* **14**(2): 159–165.

Lewis, R.M., Torczon, V. and Trosset, M.W. (2000). Direct search methods: Then and now, *Journal of Computational and Applied Mathematics* **124**(1–2): 191–207.

Li, H. and Lü, M. (2014). A three dimensional route planning method based on improved ant colony optimization algorithm, *Xibei Gongye Daxue Xuebao/Journal of Northwestern Polytechnical University* **32**(4): 563–568.

Marchaj, C. (2004). *Sailing Theory: Aerodynamics of the Sail*, Alma-Press, Warsaw, (in Polish).

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

Pêtres, C., Romero-Ramirez, M.-A. and Plumet, F. (2011). Reactive path planning for autonomous sailboat, *2011 15th International Conference on Advanced Robotics (ICAR), Tallinn, Estonia*, pp. 112–117.

Philpott, A.B., Henderson, S.G. and Teirney, D. (2004). A simulation model for predicting yacht match race outcomes, *Operations Research* **52**(1): 1–16.

Philpott, A. and Mason, A. (2001). Optimising yacht routes under uncertainty, *15th Cheasapeake Sailing Yacht Symposium, Annapolis, MD, USA*, pp. 89–98.

Pontryagin, L.S., Boltyanski, V.G., Gamkrelidze, R.V. and Mischenko, E.F. (1962). *The Mathematical Theory of Optimal Processes*, Interscience, New York, NY.

Pošík, P., Huyer, W. and Pál, L. (2012). A comparison of global search algorithms for continuous black box optimization, *Evolutionary Computation* **20**(4): 509–541.

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

Singla, G., Tiwari, A. and Singh, D.P. (2013). New approach for graph algorithms on GPU using CUDA, *International Journal of Computer Applications* **72**(18): 38–42.

Stelzer, R. and Pröll, T. (2008). Autonomous sailboat navigation for short course racing, *Robotics and Autonomous Systems* **56**(7): 604–614.

Stillwell, J. (2010). *Mathematics and Its History*, 3rd Edn., Springer, New York, NY.

Sun, J. and Wu, S. (2011). Route planning of cruise missile based on improved particle swarm algorithm, *Beijing Hangkong Hangtian Daxue Xuebao/Journal of Beijing University of Aeronautics and Astronautics* **37**(10): 1228–1232.

Sussmann, H.J. and Willems, J.C. (1997). 300 years of optimal control: From the brachystochrone to the maximum principle, *IEEE Control Systems* **17**(3): 32–44.

Szłapczyński, R. and Szłapczyńska, J. (2012). Customized crossover in evolutionary sets of safe ship trajectories, *International Journal of Applied Mathematics and Computer Science* **22**(4): 999–1009, DOI: 10.2478/v10006-012-0074-x.

Szynkiewicz, W. and Błaszczyk, J. (2011). Optimization-based approach to path planning for closed chain robot systems, *International Journal of Applied Mathematics and Computer Science* **21**(4): 659–670, DOI: 10.2478/v10006-011-0052-8.

Ćurković, P., Jerbić, B. and Stipančić, T. (2009). Swarm-based approach to path planning using honey-bees mating algorithm and art neural network, *in* Z. Gosiewska and Z. Kulesza (Eds.), *Mechatronic Systems and Materials III*, Solid State Phenomena, Vol. 147, Trans Tech Publications, Pfaffikon, pp. 74–79.

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

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

Wagner, S., Kaplinger, B. and Wie, B. (2012). GPU accelerated genetic algorithm for multiple gravity-assist and impulsive δV maneuvers, *AIAA/AAS Guidance Navigation and Control Conference, Minneapolis, MN, USA*, Vol. 4592, p. 2012.

Zamuda, A. and Sosa, J.D.H. (2014). Differential evolution and underwater glider path planning applied to the short-term opportunistic sampling of dynamic mesoscale ocean structures, *Applied Soft Computing* **24**: 95–108.

Zhou, S., Zhu, G., Li, H., Wang, Y. and Liu, X. (2011). Real-time route planning for uav based on weather threat, *2011 International Conference on Remote Sensing, Environment and Transportation Engineering (RSETE), Nanijng, China*, pp. 2342–2345.

Roman Dębski works as an assistant professor at the Department of Computer Science at the AGH University of Science and Technology. He holds M.Sc. degrees in mechanics (1997) and computer science (2002), and a Ph.D. in computational mechanics (2002). Before joining the university (in 2013) he had worked in the IT industry for over 15 years. His current interests include mathematical modeling, computer simulations, parallel processing, heterogeneous computing and trajectory optimization.

Appendix A
Simulation model for Example 1

The *performance measure* for segment i:

$$J[s_H^{(i)}(\theta)] = \frac{1}{N}\left\{\sum_{j=1}^{N}\left[r_t(\theta_j) - s_H^{(i)}(\theta_j)\right]^2\right\}^{1/2}, \quad \text{(A1)}$$

where

$$r_t(\theta) = \frac{1+2\theta(1+\theta))}{1+4\theta^2}\theta(\pi-\theta). \quad \text{(A2)}$$

The *optimization goal* is to *minimize* $J[s_H^{\widetilde{AB}}(\theta)]$; hence, it is a function approximation in L^2-norm.

Appendix B
Simulation model for Example 2

The value of the *performance measure* for segment i:

$$\begin{aligned} J[s_H^{(i)}(x)] &= \Delta t[s_H^{(i)}(x)] \\ &= Sim[s_H^{(i)}(x), config\,(\boldsymbol{v}_t, c_1, c_2, k)]. \end{aligned} \quad \text{(B1)}$$

The *optimization goal* is to *minimize* $J[s_H^{\widetilde{AB}}(x)]$, where J is the time to reach point B (i.e., it is a minimum-time problem).

Applying Newton's second law for direction s (tangent to the current multi-spline segment) to a sailboat (treated as a material point) gives the sailboat's *equation of motion* (note that single-dotted and double-dotted values represent the first and second derivatives with respect to time),

$$m\dot{v}_s = m\ddot{s} = T - R, \quad \text{(B2)}$$

where

$$T = m(c_1\beta_a - c_2)v_a^2 \quad \text{(B3)}$$

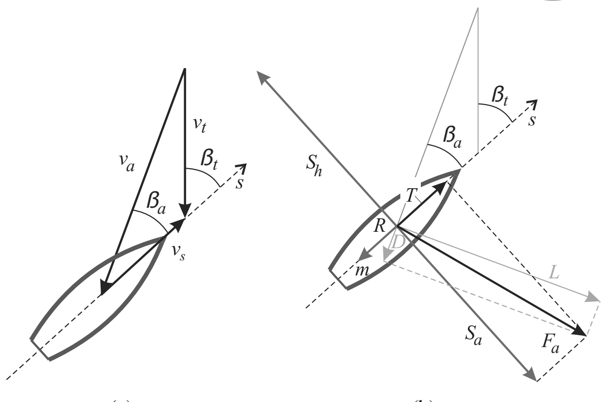

Fig. B1. Sailboat (sailing upwind) model used in Example 2. Here, β_t: true wind angle, β_a: apparent wind angle, s: (current) sailing direction, v_t: true wind velocity, v_a: apparent wind velocity, v_s: current sailboat velocity, F_a: aerodynamic force, L: lift (the aerodynamic force component perpendicular to the wind direction), D: drag (the aerodynamic force component in the direction of the wind), T: thrust, S_a: side force, S_h: hydrodynamic side force, R: total (i.e., hydrodynamic plus aerodynamic) resistance. Velocities (a), forces (b).

is thrust (approximation based on the result of Marchaj (2004)),

$$R = mk\dot{s}^2 \quad \text{(B4)}$$

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

$$v_a = \sqrt{\left(\dot{s}\sin\left(\beta_t\right)\right)^2 + (v_t + \dot{s}\cos\left(\beta_t\right))^2} \quad \text{(B5)}$$

and

$$\beta_a = \beta_t - \operatorname{acos}\left(\frac{v_t + \dot{s}\cos\beta_t}{\sqrt{(\dot{s}\sin\beta_t)^2 + (v_t + \dot{s}\cos\beta_t)^2}}\right) \quad \text{(B6)}$$

are the apparent wind velocity and apparent wind angle, respectively (see Fig. B1). In all experiments it was assumed that $c_1 = 0.03$, $c_2 = 0.005$ and $k = 0.7$ (approximation based on the result of Marchaj (2004)).

Received: 18 June 2015
Revised: 22 October 2015
Re-revised: 3 December 2015
Accepted: 4 February 2016