Scientific Visualization, 2024, volume 16, number 1, pages 1 - 18, DOI: 10.26583/sv.16.1.01

Modifications of Classical Surface Reconstruction Algorithms for Visualization of a Function Defined on a Rectangular Grid

N.V. Munts[1], S.S. Kumkov[2]

N.N. Krasovskii Institute of Mathematics and Mechanics of the Ural Branch of the Russian Academy of Sciences (IMM UB RAS), Yekaterinburg, Russia

[1] ORCID: 0000-0003-3234-1267, natalymunts@gmail.com
[2] ORCID: 0000-0002-2690-5380, sskumk@gmail.com

Abstract

In the paper, modifications of visualization algorithms for real-valued functions of two and three arguments given on a rectangular or parallelepipedal grid are considered. In the case of two arguments, the graph of the function is a surface embedded into the three-dimensional space. The majority of scientific visualization systems offer visualization procedures for such surfaces, but they construct them under the assumption that the functions are continuous. In the paper, for the case of a discontinuous function, a modification of this algorithm is proposed. In addition, the algorithm removes "plateaus" that occur after cutting the function at some level (in order to remove too large values).

Visualization of a function of three arguments implies showing its level sets, that is, regions of the space of arguments where the magnitudes of the function do not exceed a certain value. In the case of a grid function, such sets are "voxel" sets, that is, they are composed of grid cells. With that, some smoothing of the surface of such sets is required, which is carried out by the Marching Cubes algorithm and algorithms of the Laplacian family. A modification of the Marching Cubes algorithm is proposed, which preserves the symmetry of the set surface with respect to the coordinate planes, axes, or some point, if the rendered set has such a symmetry.

Keywords: human state visualization, human factor, pie charts, human resources management.

1. Introduction

Many modern computational methods are grid-based. A regular or irregular mesh is introduced in the space of function arguments to construct the function numerically. The grid nodes contain some values that approximate the value of the function to be computed. This raises the problem of presenting the results of computations to study the properties of the resultant function or to debug computational procedures. The representation obtained as the output of the computer program is a large volume of numbers (especially large in the case of usage of fine grids, which is typical when obtaining a good approximation of the desired function), which is poorly suited for perception by the researcher. The traditional way is to use visualization, that is, to transform the computed collection of numbers to some geometric objects that reflect the properties of the function being studied, and further show them to the researcher. However, of course, only two- or three-dimensional objects are available for effective visualization.

For the case of a two-dimensional argument, a natural representation of a function is its graph. Constructing the graph in this case is a very simple procedure. Each grid cell is triangulated (if the grid cells are not simplices), and linear interpolation is performed on each resultant simplex. This is what the most commonly used scientific visualization tools do

(Matlab, MathCAD, GNUPlot, etc). However, this approach produces a continuous graph, which does not correspond to the nature of the function if it has discontinuities. In addition, often, according to the essence of the problem, the function to be visualized can be *improper*, that is, it can have infinite values at some grid nodes, which is also not taken into account by standard procedures. (In numerical constructions, infinite values are replaced by very large values.) The authors are not aware of computer libraries containing algorithms for visualizing discontinuous graphs of improper functions. Traditional recommendations in such cases suggest identify manually continuous branches of the function with finite values and plotting each continuous branch independently. However, such proposals are inconvenient because they require manual processing of each graph, including situations when the graph is a single surface that has a discontinuity only at some its part.

If the function argument is three-dimensional, then the previous approach no longer works since the graph in this case is a four-dimensional object. Four-dimensional objects cannot be effectively visualized unless they have a regular structure (e.g., spheres, ellipsoids, regular polyhedra). Numerically constructed functions, as a rule, do not have regularity and, therefore, cannot be directly visualized. The traditional approach in this case is to visualize level sets of the function, that is, the regions of the space, in which the function does not exceed a certain value. Having level sets for different values, one can imagine the structure of the function as a whole. However, this also raises the problem of constructing the surface in the three-dimensional space as in the case of dealing with the graph of a function of two variables. The nice property here is that the surface of a level set is continuous, but the downside is that there is no structure on the given set of points. In fact, one has a cloud of points, which is a part of the grid nodes collection, on which the computations are performed. At best, this is a set of nodes of a parallelepipedal grid; at the worst case, the grid is irregular.

Showing a cloud of points in its natural form is relatively meaningless since this approach does not reveal the three-dimensional structure of the visualized set. Another traditional approach in the case of a parallelepipedal grid is to attach a parallelepiped of the corresponding grid cell to each point of the cloud. However, if the grid is not very fine, then the surface of the set turns out to be too “creased” and requires smoothing. The classic algorithm for smoothing such “voxel” surfaces is Marching Cubes [1]. Nevertheless, it often does not achieve sufficient visual smoothness of the surface, especially, if the mesh is not too fine and parallelepipeds of the grid cells are visually large enough. Another smoothing method involves a series of “Laplacian” algorithms [2,3,4,5]. The nature of these algorithms is that the curvature of the surface is reduced: convex parts are retracted, concave parts are squeezed out. These algorithms are iterative, and by choosing the number of iterations, a greater or lesser degree of smoothing can be achieved.

In recent years, the authors dealt with, in particular, formulation and theoretical justification of a numerical algorithm for constructing the value function of time-optimal differential games with lifeline [6,7]. The proposed numerical algorithm is namely grid-based and produces approximate values of the value function on a parallelepipedal grid. The primary evaluation of the quality of computations is associated exactly with a visual comparison of the graphs of the resultant functions with examples computed by other researchers by means of other methods. Such a comparison requires visualization. Visualization of the object being studied is burdened just by the circumstances mentioned above: the discontinuity and improper nature of the resultant function and, in the three-dimensional case, relatively large grid cells, which leads to large “bricks” and “creased” surfaces. As a result, the authors decided to develop their own algorithms for constructing a three-dimensional graph of a discontinuous function of two variables and to study the quality of smoothing of the resultant voxel surfaces using the Marching Cubes algorithm, Laplace algorithms, and their combined application.

In the case of visualization of level sets of functions of three variables, a combination of Marching Cubes and the Laplacian HC algorithm [5] is used. Primary smoothing is carried out using Marching Cubes, finer refinement is performed by means of the Laplacian algo-

rithm. However, when using this procedure for constructing level set surfaces, some problems arise: from a cloud of points that is symmetrical with respect to one or more coordinate planes, a surface is constructed that did not have the corresponding symmetry. The main problem here is that Marching Cubes, when processing some configurations of points, produces quadrangular surface elements that are triangulated in an asymmetrical manner. To overcome this problem, a modification of this algorithm is proposed that preserves the symmetry of the triangulated surface with an accuracy of individual triangles.

This article describes the proposed modifications of the algorithms. Examples of their work are given and a comparison of the resultant surfaces with those obtained by classical versions of the algorithms is shown. The paper is organized as follows. The second section discusses the visualization of a three-dimensional graph of a function of two arguments, describes the procedure for visualizing a graph that correctly represents discontinuities of the function, and provides several examples of graphs of discontinuous functions. The third section examines the visualization of three-dimensional sets, proposes a modification of the Marching Cubes algorithm that preserves the symmetry of the set, and provides some examples of three-dimensional sets. The article ends with a conclusion and a list of references.

2. Visualization of a three-dimensional graph of a discontinuous function of two arguments

2.1. Description of the surface reconstruction procedure

In popular scientific visualization systems (e.g., Matlab and GNUPlot), essentially the same procedure is used to reconstruct a three-dimensional graph of a function of two arguments. Its input information is a set of points in three-dimensional space specified at some nodes of a rectangular grid of the arguments. Also, some systems (for example, Matlab) additionally require that the set of nodes fill a rectangle:

$$\mathcal{L} = \{x_i = x_0 + i \cdot \Delta,\ y_j = y_0 + j \cdot \Delta,\ z_{i,j}\},\quad i = 0,\ \ldots,\ N,\quad j = 0,\ \ldots,\ M.$$

The surface to be drawn is generated using the following procedure.

Each rectangular grid cell with vertices at points (x_i, y_j), (x_{i+1}, y_j), (x_{i+1}, y_{j+1}), (x_i, y_{j+1}) is divided by a diagonal into two triangular cells with vertices at the points

$$\{(x_i, y_j), (x_{i+1}, y_j), (x_{i+1}, y_{j+1})\} \text{ and } \{(x_i, y_j), (x_{i+1}, y_{j+1}), (x_i, y_{j+1})\}.$$

Further, triangles are constructed in the three-dimensional space with the vertices at the points corresponding to the vertices of the resultant cells:

$$\{(x_i, y_j, z_{i,j}), (x_{i+1}, y_j, z_{i+1,j}), (x_{i+1}, y_{j+1}, z_{i+1,j+1})\} \text{ and}$$
$$\{(x_i, y_j, z_{i,j}), (x_{i+1}, y_{j+1}, z_{i+1,j+1}), (x_i, y_{j+1}, z_{i,j+1})\}.$$

The drawbacks of this algorithm are obvious. First, it implies that the reconstructed graph is continuous. If there are abrupt changes of the function value at neighboring grid nodes, then instead of the desired rupture of the graph, almost vertical "walls" appear in such places. Secondly, in the problems under consideration, the function to be visualized could be improper taking infinite values in some regions. In this case, infinite values are represented as some very large value. This structure of the function also gives breaks along the lines separating areas with finite and infinite function values. In addition, due to large values, a too wide range appears, in which the surface of the graph is constructed, while the most interesting parts of the graph are "squashed" and become small. Third, existing procedures imply that the grid nodes of the provided set $\mathcal{L}$ fill a rectangle. Generally speaking, this may not be the case; for example, the function is computed on a circle, that is, the set of grid nodes fills not a rectangle, but a circle.

These shortcomings lead to the need to create a new algorithm for constructing the graph surface:

1. The points are being read. During reading, the internal memory storage of the procedure does not include points, at which the function takes infinite (large) values. Filtering is

carried out based on the value of the function at a point exceeding a certain threshold v_∞. The threshold is selected visually or using knowledge about the computational algorithm (what value is used to replace infinity).

2. The points $(x_{i,j}, y_{i,j}, z_{i,j})$ remaining after filtering are run through. If for the current point all three nodes (x_{i+1}, y_j), (x_{i+1}, y_{j+1}), (x_i, y_{j+1}) are present in the set, then, same as in the classical procedure, two triangular cells are formed. If only two of these three nodes are present in the set, then one corresponding cell is formed. If only one node or none is present, no triangular cells are formed and the current point is skipped.

3. If, when processing a point, one or two triangular cells are formed, then these cells are processed. The processing consists of checking the absence of a "gap" in this cell, that is, it is checked that the function values at the vertices of the cell differ in pairs by no more than a given value ε. If this is the case, then it is considered that there is no gap, and a triangle with the vertices at the corresponding points is added to the formed surface. If at least one pair of values differs by more than some threshold ε, then a discontinuity is detected and the cell does not produce a triangle into the formed surface.

The generated surface is written to an output file in the .obj format and further viewed in one or another 3D viewer that works with this format. The authors use the free MeshLab system. The parameters v_∞ and ε are the input data of the construction procedure and are selected individually for each visualized graph by means of visual control of the quality of the resultant surface.

Note that due to the filtering of points performed at the step 1 of the above algorithm, the area, in which the graph is plotted, may be significantly non-rectangular, non-simple-connected, or even disconnected. However, this fact does not prevent the described procedure from constructing and displaying the required graph on the argument area of any shape.

2.2. Examples

Example 2.1. The controlled system «Dubins' car». It is said above that the authors are researching numerical procedures aimed at finding the value function of time-optimal differential games. However, the same procedures can also be applied to construct the optimal result function in control problems. To do this, a fictitious control of the second player is introduced into the description of the system, which is not included into the dynamics and is constrained by a one-point set (to reduce the enumeration of possible controls).

In this way, the classical controlled system "Dubins' car" (see, e.g., [8,9,10,11]) has been numerically studied. It describes the simplest version of the movement of a car (airplane, ship) in a horizontal plane with a constant linear speed and a turn radius bounded from below. The original system has a three-dimensional phase vector (X, Y, φ) where X, Y are the geometric coordinates of the object's position on the plane, φ is the current heading angle of motion (counted counterclockwise from the positive direction of the X axis). The dynamics of the system is described by the relations\

$$\dot{X} = \cos\varphi, \dot{Y} = \sin\varphi, \ \dot{\varphi} = u, \ |u| \le 1. \qquad (1)$$

The linear speed magnitude is considered equal to 1. The restrictions for the control u define the minimum turning radius (the maximum possible angular velocity of the linear speed vector turn). At the initial instant, the object is located at the origin and moving to the positive direction of the X axis: $X(0) = 0$, $Y(0) = 0$, $\varphi(0) = 0$. The aim of the control is to bring the object to a given point (X^*, Y^*) as quickly as possible with any direction of speed at the instant of reaching the target point.

Three-dimensional dynamics (1) can be reduced to a two-dimensional one by introducing a new moving coordinate system [15,16]. The origin of the system is aligned with the current position of the object, the y-axis is directed along the current direction of the object's velocity, and the x-axis is introduced to obtain a right-handed coordinate system. This eliminates the φ coordinate (the heading angle). The dynamics of the system in new coordinates has the form

$$\dot{x} = -yu,\ \dot{y} = xu - 1,\ |u| \le 1. \quad (2)$$

Here, the coordinates x, y have the meaning of the current relative position of the target point. This moving point must be brought to the origin as quickly as possible from the initial position $x(0) = Y^*$, $y(0) = -X^*$, that is, aligned with the current position of the object.

So far, this problem has been quite well studied both by analytical methods [8,12,13] and using numerical approaches [14]. Traditionally, the optimal result function in this case is represented as a set of level lines for various magnitudes (see Fig. 1). The inconvenience of this method of presentation is that it is impossible to consider the discontinuities of the function, but one can only suspect their presence at the places where the level lines are dense.

Fig. 2 shows a graph of the optimal result function of the "Dubins' car" problem for the situation of point reaching the origin (to a small neighborhood of the origin). On the left, a surface is shown, which is constructed using a procedure built into the GNUPlot system. On the right, one can see a surface constructed by means of the procedure proposed by the authors. As mentioned above, visualization of the graph surface in this case is made in the MeshLab system. The coloring of the graph denotes different levels of function value from zero (magenta) to the maximum available (red). Similar coloring of graphs is used for other figures in this section.

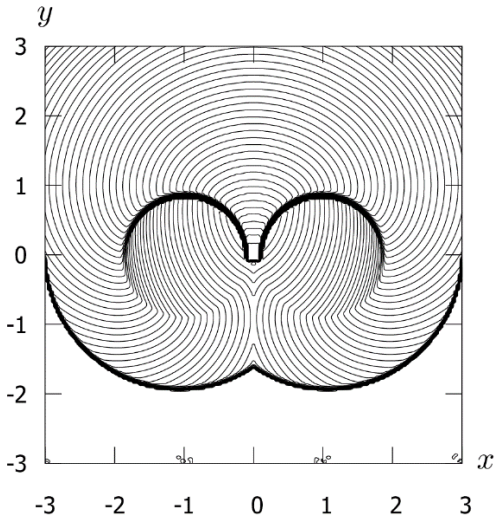

Fig. 1. One of the traditional representations of the optimal result function in the case of a two-dimensional phase vector in the form of a set of level lines (for some version of the "Dubins' car" problem)

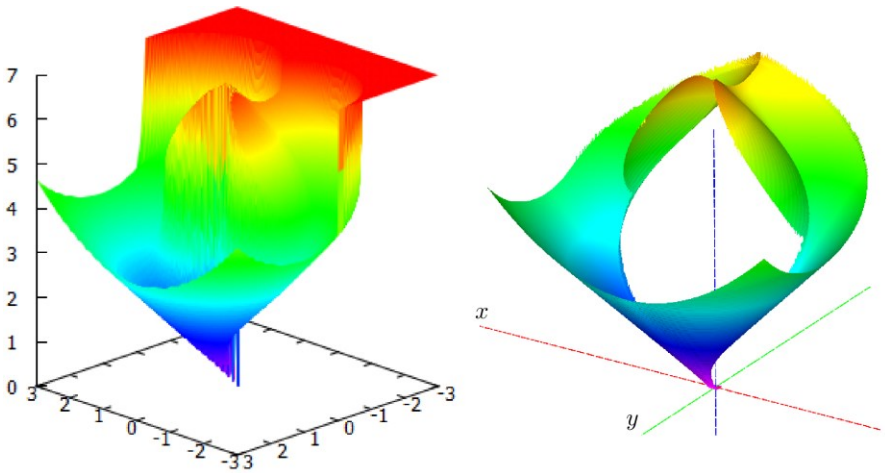

Fig. 2. Graph of the optimal result function for the "Dubins' car" problem with reduced dynamics. *On the left*: a graph of the value function visualized using GNUPlot system algorithms; *on the right*: the graph visualized using the MeshLab system after creating the surface

using the authors' algorithm. The coloring of the graph highlights different levels of the function magnitude from zero (magenta) to the available maximum (red)

One can see the absence of a "plateau" in the region of infinite function values, which in the left subfigure are cut off at level 7. There is no additional processing of the surface borders at the discontinuity lines of the green, yellow and orange parts of the surface on the right edge of the graph, as well as of the line between two continuous parts of the surface on the blue and the green parts of the surface to the right in the "hole". As a result, there is some "fringe" at the edges of the gap. It appears because the break line is located quite arbitrarily relatively to the grid nodes and the values at the "border" nodes correlate with each other quite arbitrarily (in the sense of which of them are larger and which are smaller, and by how much).

Despite the artifacts arise, the right image conveys the structure of the graph more adequately, and, therefore, of the function itself than the left one.

Example 2.2. Visualization of a graph on a non-rectangular set. The second example is also related to the Dubins' car problem with dynamics (2). However, now the graph of the optimal result function is computed for grid nodes belonging to a circle of radius 3 centered at the origin. The graph of the optimal result function is shown in Fig. 3.

It should be noted that GNUPlot is able to visualize a graph on a non-rectangular set. However, due to the complex nature of the function discontinuity, the resultant representation does not reflect the features of the graph at all. The surface reconstructed using the procedure proposed by the authors reproduces the nature of the discontinuities of the function much better. There is still a small "fringe" on the break lines.

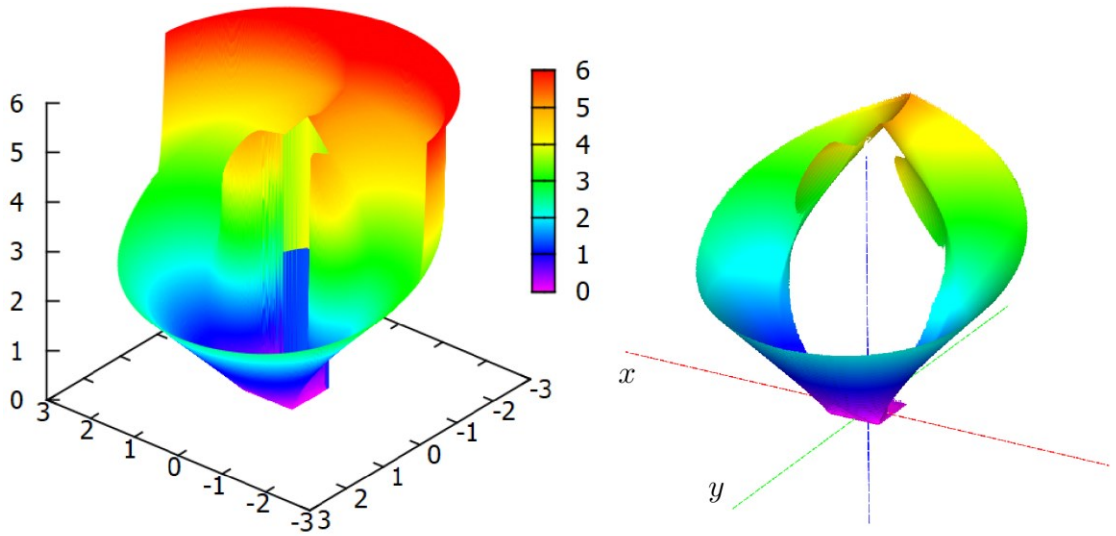

Fig. 3. Graph of the optimal result function for some version of the "Dubins' car" problem computed on a circular set; drawn by the GNUPlot system on the left, drawn by the MeshLab system on the right. The surface is constructed using the authors' algorithm. The coloring of the graph highlights different levels of the function value from zero (magenta) to the maximum available (red)

Example 2.3. The «homicidal chauffeur» game. Additional processing of the graph. At the step 1 of the algorithm, when reading and filtering points with finite values, or before it, additional processing of the graph is possible: filtering by additional criteria, recalculation of graph points (for example, scaling along the axis of function values), etc. Since the input data file is simply a list of three-dimensional points (x, y, function value), then filtering can be done (and is done) by executing a script that processes the original list of points into a new one. Procedures for any special filtering (except for cutting off large values) are not built into the converter created by the authors and used now. Additional filtering or data processing can

be implemented by additional scripts that process the point cloud file before submitting it to the graph surface generation procedure.

The third example is related to the classic "homicidal chauffeur" differential game proposed by R. Isaacs [15,16]. In this game, the pursuer (car), which has the dynamics of Dubins' car, pursues the evader (pedestrian) having the dynamics of simple motions, that is, it can choose the direction and magnitude of its speed at any given moment:

$$\begin{gathered} \dot{X_P} = w_P \cos\varphi, \quad \dot{Y_P} = w_P \sin\varphi, \quad \dot{\varphi} = \frac{w_P}{R} u, \\ \dot{X_E} = w_E v_X, \quad \dot{Y_E} = w_E v_Y, \\ |u| \le 1, \quad |(v_X, v_Y)| \le 1. \end{gathered} \tag{3}$$

Here, (X_P, Y_P), (X_E, Y_E) are the positions of the pursuer and the evader on the plane; φ is the heading angle, the direction of the speed of the pursuer; R is the minimum turning radius of the pursuer; w_P is the linear speed of the pursuer; w_E is the maximum speed value of the pedestrian.

The aim of the pursuer is to bring the evader to some given neighborhood as quickly as possible, that is, to fulfill the inequality $|(X_P, Y_P) - (X_E, Y_E)| \le l$ for a given l at the earliest possible instant. The evader tries to avoid the capture or, if this is not possible, to delay it as much as possible.

It can be observed that the original game has a five-dimensional phase vector with components X_P, Y_P, φ, X_E, Y_E. However, using the introduction of a moving coordinate system described above, it can be reduced to a game with a two-dimensional vector and dynamics

$$\begin{gathered} \dot{x} = -\frac{w_P}{R} yu + w_E v_x, \quad |u| \le 1, \\ \dot{y} = \frac{w_P}{R} xu - w_P + w_E v_y, \quad |(v_x, v_y)| \le 1. \end{gathered} \tag{4}$$

The graph of the value function for some variant of this game computed by the authors is shown in Fig. 4. The subfigure on the left shows a visualization in the GNUPlot system of data directly obtained from the program. Again, walls are visible at the breaks and red plateau in the region of infinite values. In addition, the output is produced equally scaled, which flattens the graph too much along the value axis. On the right, the surface constructed by the algorithm proposed by the authors is shown. In addition, the graph is additionally cropped along a certain circle centered at the origin through additional filtering at the first step of the algorithm.

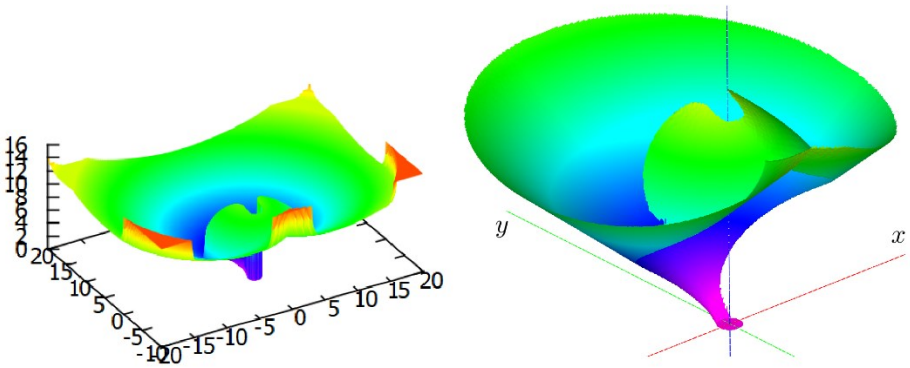

Fig. 4. Graph of the value function of some variant of the "homicidal chauffeur" game. *On the left*: the entire graph visualized in the GNUPlot system; *on the right*: the graph cropped to a circle and scaled along the axis of the function values constructed by the authors' procedure and visualized in the MeshLab system. The coloring of the graph emphasizes the different levels of the function magnitude from zero (magenta) to the existing maximum (red and yellow-green colors in the left and right sub-figures)

3. Visualization of three-dimensional level sets of three-argument function

3.1. Description of surface reconstruction procedure

The input data for this procedure is a cloud of points obtained by filtering the entire set of nodes of the original mesh under the condition that the magnitude of the function at the node does not exceed a given value. It is required to visualize the set “filled” with this point cloud.

As it is discussed in the introduction, the direct output of a point cloud does not represent the three-dimensional structure of the set. “Bricks” equal in size to a grid cell and attached to existing points result in an overly “creased” surface.

Traditionally, they use the Marching Cubes algorithm [1], which actually forms a “brick” surface and then slightly smoothens it inflating the set a bit. A nice property of this algorithm is that, in fact, when it works, the stage of creating the “bricks” is skipped: it immediately forms the resultant surface in each grid cell. To do this, the algorithm requires an information, which vertices of each cell belong to the set to be visualized and which do not. That is, for correct working of the algorithm, one also need to pass the grid parameters: the pivot point and steps along the axes. The algorithm is organized in such a way that surface fragments being constructed in neighboring cells automatically join each other for any configuration of the vertices of these cells. Fig. 5 shows the main configurations of the cell vertices. The remaining configurations are formed symmetrically. Note that there are no configurations describing cells that have five or more vertices belonging to the visualized cloud. In such cases, the cell is considered "internal", too surrounded by the set to contain elements of the generated surface.

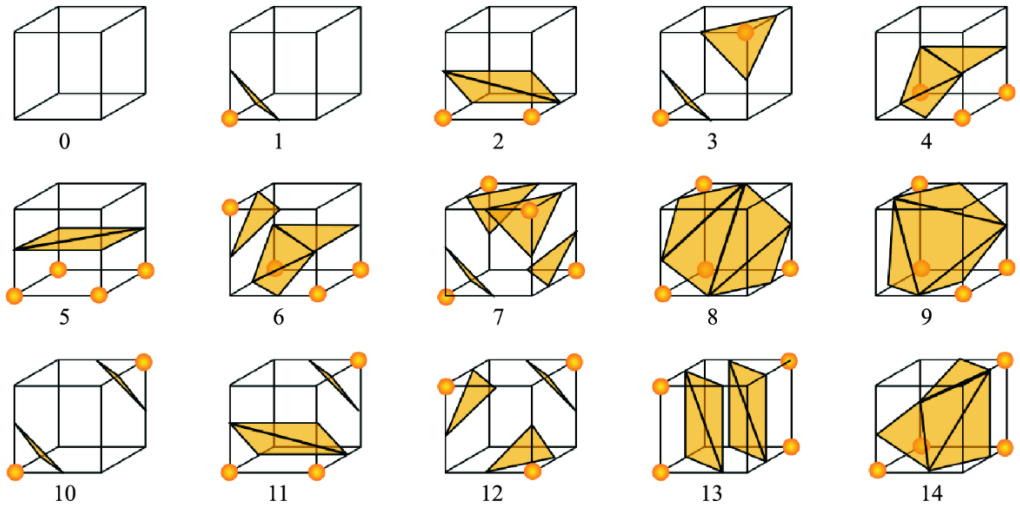

Fig. 5. Basic configurations of sets of cell vertices and elements of the surface to be constructed in the Marching Cubes algorithm. Orange dots mark nodes belonging to the input cloud (the figure is taken from https://ru.wikipedia.org/wiki/Marching_cubes)

Fig. 6b shows the result of the Marching Cubes algorithm when reconstructing the surface of a ball defined by a cloud of points. For comparison, Fig. 6a shows an image of the surface of a “voxel” set obtained by attaching a cube of the appropriate size to each point from the given cloud. Fig. 6c is discussed below.

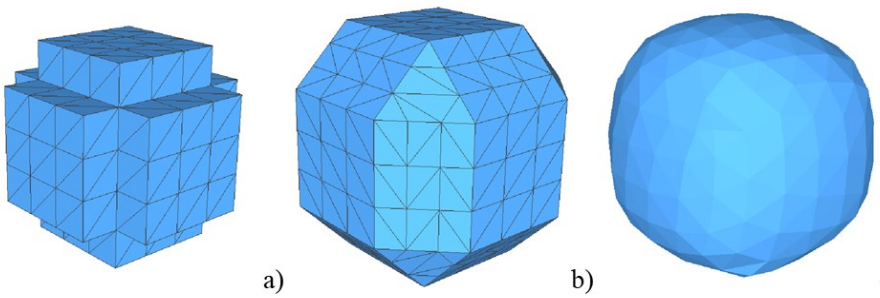

Fig. 6. Visualization of a three-dimensional ball defined by a cloud of points: a) a "voxel" set; b) the result of the Marching Cubes algorithm on the original cloud of points; c) the result of applying the Laplace algorithm to the surface obtained by the Marching Cubes algorithm

It can be seen that due to the small size of the point cloud, the size of one cube is relatively large in comparison with the size of the entire set, so the voxel set has a very "creased" surface. For the same reason, the result of the Marching Cubes algorithm also does not look smooth enough. Note that the consideration of sets defined by a small number of points is also important when studying control problems. This is due to the fact that the evolution of reachability sets at instants close to zero (the evolution of the value function level sets corresponding to near-zero values) is often especially interesting for researchers.

Further smoothing of the surface is possible by one or another Laplace algorithm [2,3,4,5]. The basic idea of this family of algorithms is as follows. For every vertex p_i of the surface, its "fan" $\{q_j\}_{j=1}^{N_i}$ is collected, that is, the set of vertices adjacent to p_i in a triangle of the surface. Then the vertex p_i is replaced by the arithmetic mean of the vertices from the fan:

$$p_i' = \frac{1}{N_i}\sum_{j=1}^{N_i} q_j$$

All surface vertices are replaced simultaneously. In fact, when this procedure works, the surface is "flattened": the convex parts are retracted, the concave parts are squeezed out. To obtain one or another degree of smoothing, the algorithm is applied iteratively several times. Some changes in the computational formula in one way or another are related to, in general, maintaining the volume of the visualized body. If such changes are not made, then repeated application of the procedure to the convex body deflates.

In this case, obviously, if the original surface itself is symmetrical with respect to any plane, line or point, and its triangulation is also symmetrical, then the result of the Laplace algorithm gives a result that preserves this symmetry. Otherwise, the symmetry may be broken, as it can be seen in Fig. 6c. Due to the asymmetry of surface triangulation in Fig. 6b, the result is asymmetrical. As a result, after applying the Laplace algorithm, the result in Fig. 6c is noticeably asymmetrical.

In connection with this, a desire appeared to modify the Marching Cubes algorithm in such a way that if the input cloud of points is distributed symmetrically with respect to certain directions associated with the mesh directions, then the triangulation of the resultant surface would have the same symmetry. The asymmetry of the triangulation produced by Marching Cubes is essentially due to the fact that in some situations an element that is not a triangle is actually added to the surface: the patterns 2, 4, 5, 6, 8, 11, 13 in Fig. 5. However, a non-triangular element itself cannot be added and must be triangulated at first. In Fig. 5, one can see that quadrilaterals, symmetrical in symmetrical configurations, are divided by the diagonal into two triangles, which, generally speaking, are asymmetrical. This leads to the asymmetry of triangulation, which is visible in Fig. 6b, where the surface triangles are highlighted with thin lines.

The natural solution is to change the triangulation of the quadrangular elements, dividing each of them into four triangles by both diagonals (see Fig. 7). In this case, symmetrical quadrilaterals are be triangulated symmetrically.

With this change in the algorithm, a special attention should be paid to the correct processing of pattern 8 and the ones symmetric to it. The remaining marked patterns add a quadrilateral element that is uniquely divided into four triangles. Triangulation of the hexagon, which is added in pattern 8 and in those symmetrical to it, can be done ambiguously.

Now, the same ball example as above is processed by the "symmetrical" algorithm. Fig. 8b shows that the surface triangulation is now symmetrical. As a consequence, the result of applying the Laplace algorithm to it gives a symmetric result (see Fig. 8c).

Fig. 9 shows surfaces from Figs. 6c and 8c, which are imposed over each other. The camera is located on one of the coordinate axes, and the direction of view is parallel to this axis. One can see how the blue surface obtained using the classic (asymmetrical) version of the Marching Cubes algorithm extends beyond the pink surface obtained using the symmetrical version. Moreover, the protrusion places are also located asymmetrically. The symmetry of the pink surface is more clearly visible in this figure than in Fig. 8.

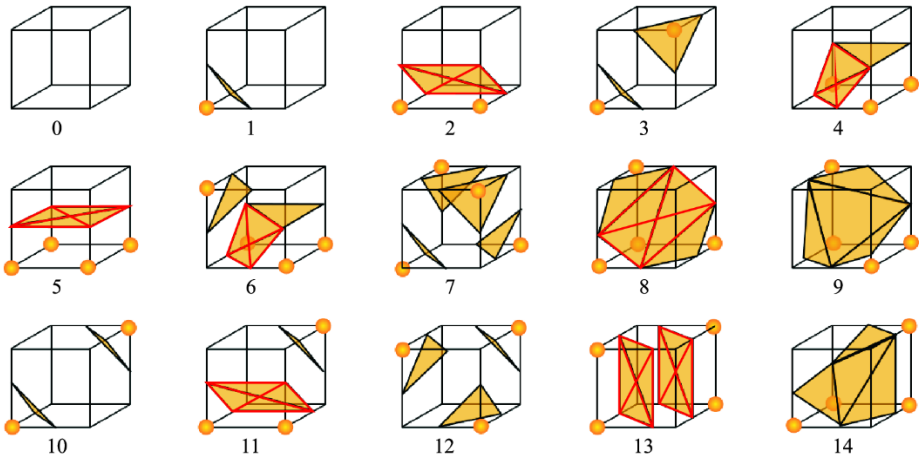

Fig. 7. Corrected Marching Cubes algorithm templates

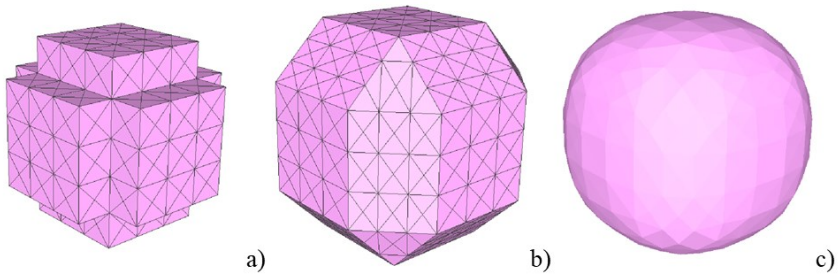

Fig. 8. Symmetric visualization of a three-dimensional ball defined by a cloud of points: a) a "voxel" set; b) the result of the modified Marching Cubes algorithm on the original cloud of points; c) the result of applying the Laplace algorithm to the surface obtained by the modified Marching Cubes algorithm

Fig. 9. Comparison of the surfaces from Figs. 6c (the blue surface) and 8c (the pink surface)

3.2. Examples

Below, examples of the proposed procedure for construction of the surface of three-dimensional voxel sets are given. Results are shown for the classic and "symmetrical" versions of the Marching Cubes algorithm. Same as in Figs. 6 and 8, the results of the classic version of the algorithm are depicted in blue and of the symmetric version in pink.

Example 3.1. The Dubins' car in the original coordinates. In Section 2, in Examples 2.1 and 2.2, the visualization of the graph of the optimal result function for the Dubins' car controlled system with reduced dynamics (2) is examined. However, it is also of interest to study the problem in its original form with a three-dimensional phase vector and dynamics (1). As mentioned, in this case the structure of the optimal result function can be studied by considering a collection of its level sets for various values. In terms of control theory, such sets are reachability sets up to instant, that is, sets of points in the phase space, in which the system can end up no later than a given instant starting from the initial position or from the initial set.

Regarding the visualization of the reachability sets of the Dubins' car, the question arises regarding the angular coordinate φ. By its meaning, its values are in the range $[0,2\pi)$ rad (that is, in the range $[0°,360°)$). However, in this case, the set has a very strange appearance, not too convenient for analysis. Therefore, it is supposed that the values of the coordinate φ can take any value from $-\infty$ to $+\infty$.

Figs. 10, 11 and 12 show the level sets of the value function for the magnitude 0.2 (the reachability set from the origin up to the instant 0.2 sec). The set is relatively small, and the size of the grid cells is quite large compared to the size of the entire set, and therefore the results of applying the classic and modified Marching Cubes algorithms have significant differences.

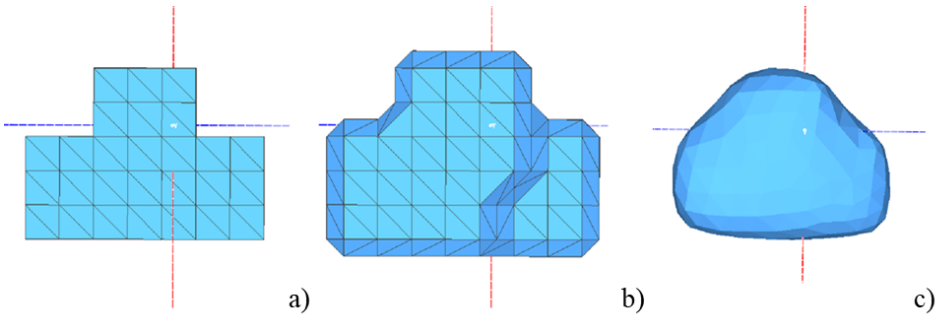

Fig. 10. "Dubins' car", reachability sets up to the instant 0.2: a) a "voxel" set; b) the result of the classic Marching Cubes algorithm on the original cloud of points; c) the result of applying the Laplace algorithm to the surface obtained by the classical Marching Cubes algorithm

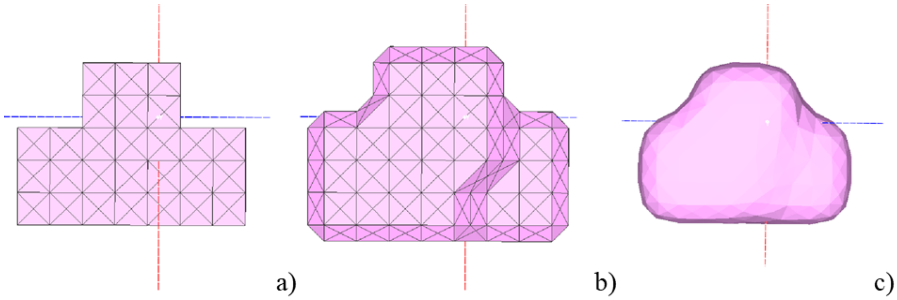

Fig. 11. "Dubins' car", reachability sets up to the instant 0.2: a) a "voxel" set; b) the result of the modified Marching Cubes algorithm on the original cloud of points; c) the result of applying the Laplace algorithm to the surface obtained by the modified Marching Cubes algorithm

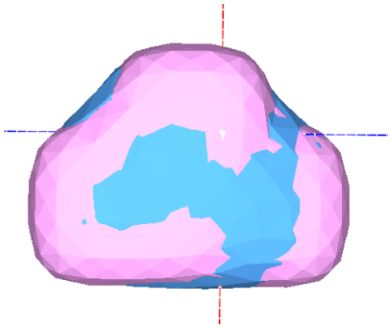

Fig. 12. "Dubins' car", comparison of surfaces from Figs. 10c and 11c

Fig. 13 shows the surfaces of the level set of the value function for the magnitude 3π, constructed by Laplace smoothing of the result of applying the classical and modified versions of the Marching Cubes algorithm.

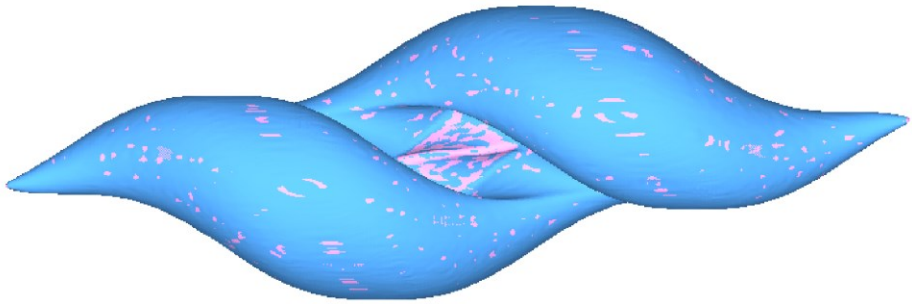

Fig. 13. "Dubins' car", the reachability sets up to the instant 3π: comparison of the surfaces obtained using the classic Marching Cubes algorithm (the blue surface) and the modified one (the pink surface)

Example 3.2. "Isotropic rockets" game. In his book [15,16], R. Isaacs considered also a game similar to the "homicidal chauffeur", but in which the pursuer controls his acceleration. The evader still has the dynamics of simple motions. He called this game "isotropic rockets". The dynamics in the original coordinates is described by the relations

$$\begin{gathered} \dot{X}_P = W_X, \quad \dot{W}_X = a\cos\Theta, \quad \dot{X}_E = w_E v_X, \quad a \in [\underline{a}, \overline{a}], \Theta \in [0, 2\pi), \\ \dot{Y}_P = W_Y, \quad \dot{W}_Y = a\sin\Theta, \quad \dot{Y}_E = w_E v_Y, \quad |(v_X, v_Y)| \le 1. \end{gathered} \tag{5}$$

Here, as in the case of dynamics (3), (X_P, Y_P) and (X_E, Y_E) are the positions of the pursuer and the evader on the plane. The vector (W_X, W_Y) is the speed of the pursuer. The maximum value of the speed of the evader is w_E, the vector (v_X, v_Y) is the control action of the evader, the instantaneous direction of its movement. The pursuer's control, the acceleration acting on it, is described by two parameters: a is the magnitude of the acceleration belonging to a given interval $[\underline{a}, \overline{a}]$; Θ is the angle between the positive direction of the x-axis and the instantane-

ous direction of the control acceleration counted counterclockwise and selected from the range $[0,2\pi)$.

This problem has a phase vector of dimension 6. However, by the same coordinates substitution that is used in the examples in the previous section (we superpose the origin with the pursuer and the ordinate axis with the direction of its instantaneous velocity), we can reduce the dimension of the phase vector to 3. Dynamics in new coordinates has the following form [15, p. 244], [16, p. 302]:

$$\begin{aligned} \dot{x} &= -\frac{y}{w_P}\cdot a\sin\theta + w_E v_x, & a &\in [\underline{a},\overline{a}], \\ \dot{y} &= \frac{x}{w_P}\cdot a\cos\theta + w_E v_y - w_P, & \theta &\in [0,2\pi), \\ \dot{w}_P &= a\cos\theta, & |(v_x,v_y)| &\le 1. \end{aligned} \tag{6}$$

Here, (x,y) are the relative coordinates of the evader with respect to the pursuer, w_P is the instantaneous value of the pursuer's speed, w_E is the maximum value of the evader's velocity, (v_x,v_y) is the vector of the instantaneous direction of the evader's speed, a is the magnitude of the control acceleration of the pursuer, θ is the angle describing the direction of acceleration of the pursuer. However, now it is convenient to count it from the direction of the pursuer's speed, that is, from the ordinate axis, and clockwise, not counterclockwise.

In paper [17], a variant of this problem is considered, in which the magnitude of the acceleration is constant $a = \underline{a} = \overline{a} = 1$, and the acceleration itself can be directed in such a way only to accelerate the pursuer, but not to slow it down: $-\pi/2 \le \theta \le \pi/2$. At the same time, to prevent the occurrence of a too high linear speed, a limitation $\underline{w} \le w_P \le \overline{w}$ is imposed on the speed magnitude. In other words, when the linear speed reaches the limit $w = \overline{w}$ only the controls $\theta = \pm\pi/2$ become valid, which no longer affect the value of the linear speed increasing it, but only change its direction.

Figs. 14, 15 show the level sets of the value function for the magnitude 5.7. In Fig. 14 on the left, there is a surface obtained using the asymmetrical Marching Cubes algorithm and on the right, a surface constructed by the modified one. Due to the small size of the grid cells, the differences are almost not visible. Fig. 15 compares surfaces obtained using the classical (blue surfaces) and symmetric (pink surfaces) Marching Cubes algorithms.

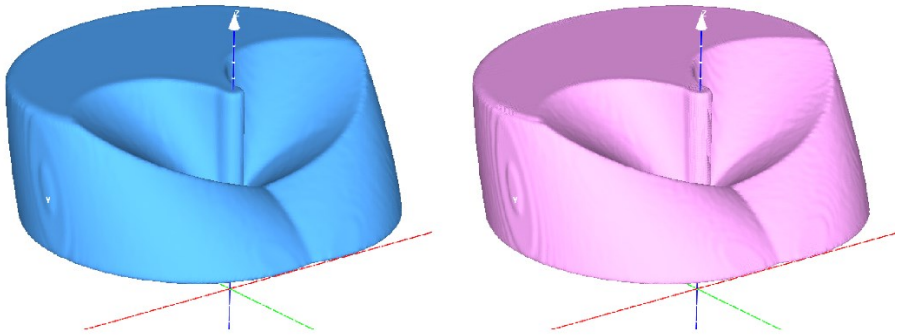

Fig. 14. "Isotropic rockets", the value function level set corresponding to the constant 5.7. *On the left*: a surface obtained using the asymmetric Marching Cubes algorithm; *on the right*: a symmetric algorithm is used

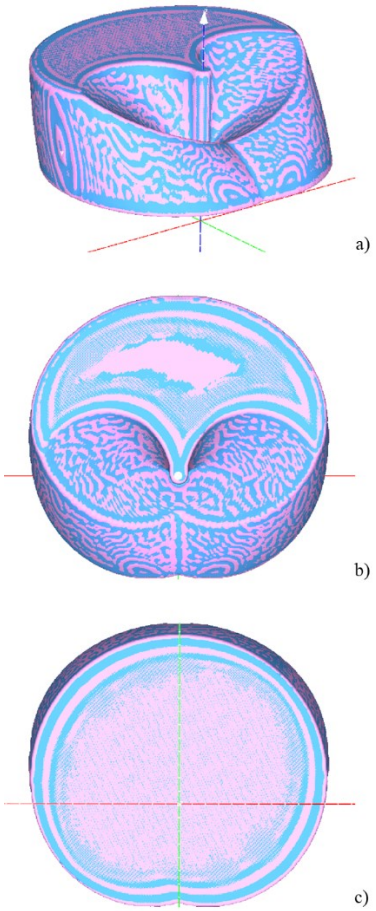

Fig. 15. “Isotropic rockets”, the value function level set corresponding to the constant 5.7. Comparison of surfaces obtained using the classic Marching Cubes algorithm (the blue surface) and the modified one (the pink surface): a) a side view, b) a top view (from the positive direction of the *Oz* axis), c) a bottom view (from the negative direction of the *Oz* axis)

4. Software implementation of three-dimensional sets smoothing

The computer program is implemented in C# for the .NET Framework 4.8 software platform under the Microsoft Windows operating system. However, it should be noted that over the two years passed after the program had been written, Microsoft has released a cross-platform .NET Core version of this environment. Because the program uses only standard language capabilities, it can, if necessary, be launched under other operating systems, for which there is a .NET Core implementation, for example, Linux or macOS.

The described algorithms are implemented as a set of libraries and console applications. Console applications take as input a file with data in the .txt format (a set of points and values at them). As output they generate a new file in the .obj format containing a set of points, triangles with vertices at these points, and normal vectors to these triangles. Files of this format can be opened and viewed, in particular, using the MeshLab program.

The algorithm for constructing a graph of a two-argument discontinuous function is implemented as described in Section 2.1. The implementation of the HC-algorithm repeats the description from article [5]. It is interesting to note some features of implementation of the symmetric version of the Marching Cubes algorithm.

The classic algorithm goes through the grid cells and looks for each cell, which of its vertices belong to the cloud of points to be visualized. In Fig. 5, it can be seen that the vertices of the added triangles can only be at the following points, the coordinates of which are specified in the local coordinate system of the parallelepipedal grid cell being processed:

```
Point[] vertices = new Point[]
{
  new Point(0.5, 0, 0),
  new Point(1, 0.5, 0),
  new Point(0.5, 1, 0),
  new Point(0, 0.5, 0),
  new Point(0.5, 0, 1),
  new Point(1, 0.5, 1),
  new Point(0.5, 1, 1),
  new Point(0, 0.5, 1),
  new Point(0, 0, 0.5),
  new Point(1, 0, 0.5),
  new Point(1, 1, 0.5),
  new Point(0, 1, 0.5)
};
```

Whether or not each of the eight vertices belongs to the visualized cloud can be determined using a byte variable. If a vertex belongs to the cloud, then the byte variable contains 1 in the corresponding bit; if it does not belong, then 0. Thus, each possible configuration of cloud of points at the vertices of the cell corresponds to some integer value from 0 to 255. This value is used as an index in the faces array, each element of which is also an array of indices of points from the vertices array, which contain the vertices of the triangles added with this configuration of the cloud. Each triple of indices defines its own triangle. The maximum number of triangles (four) in the added surface element determines the size of the array row. The last three elements on each line are entered to indicate the end of the line. The index -1 means “no triangle”:

```
int[,] faces = new int[256, 15]
{
  {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
  {0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
  ........................
  {9, 5, 7, 9, 7, 2, 9, 2, 0, 2, 7, 11, -1, -1, -1},
  ........................
  {6, 5, 9, 6, 9, 11, 11, 9, 8, -1, -1, -1, -1, -1, -1},
  ........................
  {0, 3, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
  {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}
};
```

The sequence of indices in each triple corresponds to a counterclockwise traversal of the corresponding triangle when viewed from the end of the normal external to the set described by the given cloud.

The modified scheme (Fig. 7) uses a similar idea for its implementation. In this case, an extended list of possible vertices of added triangles is used:

```
Point[] vertices = new Point[]
{
  new Point(0.5, 0, 0),
  new Point(1, 0.5, 0),
  new Point(0.5, 1, 0),
  new Point(0, 0.5, 0),
  new Point(0.5, 0, 1),
  new Point(1, 0.5, 1),
  new Point(0.5, 1, 1),
  new Point(0, 0.5, 1),
  new Point(0, 0, 0.5),
  new Point(1, 0, 0.5),
  new Point(1, 1, 0.5),
  new Point(0, 1, 0.5),
  new Point(0.5, 0.25, 0.25),
```

```
    new Point(0.75, 0.5, 0.25),
    new Point(0.334, 0.667, 0.167),
    new Point(0.5, 0.5, 0.5),
    new Point(0.25, 0.5, 0.25),
    new Point(0.667, 0.667, 0.167),
    new Point(0.5, 0.75, 0.25),
    new Point(0.667, 0.334, 0.167),
    new Point(0.334, 0.334, 0.167),
    new Point(0.25, 0.25, 0.5),
    new Point(0.25, 0.5, 0.5),
    new Point(0.167, 0.667, 0.667),
    new Point(0.334, 0.334, 0.834),
    new Point(0.75, 0.25, 0.5),
    new Point(0.334, 0.167, 0.667),
    new Point(0.834, 0.667, 0.667),
    new Point(0.667, 0.334, 0.834),
    new Point(0.5, 0.25, 0.75),
    new Point(0.667, 0.167, 0.334),
    new Point(0.334, 0.167, 0.334),
    new Point(0.667, 0.834, 0.334),
    new Point(0.334, 0.834, 0.334),
    new Point(0.5, 0.75, 0.75),
    new Point(0.75, 0.75, 0.5),
    new Point(0.834, 0.334, 0.667),
    new Point(0.334, 0.834, 0.667),
    new Point(0.667, 0.667, 0.834),
    new Point(0.667, 0.167, 0.667),
    new Point(0.834, 0.334, 0.667),
    new Point(0.75, 0.5, 0.75),
    new Point(0.834, 0.667, 0.334),
    new Point(0.834, 0.334, 0.334),
    new Point(0.167, 0.334, 0.334),
    new Point(0.167, 0.667, 0.334),
    new Point(0.25, 0.5, 0.75),
    new Point(0.334, 0.667, 0.834),
    new Point(0.667, 0.834, 0.667),
    new Point(0.167, 0.334, 0.667),
    new Point(0.25, 0.75, 0.5)
};
```

The array of faces is modified accordingly, which now has size of 256×24 since the maximum number of triangles in the added surface element is eight (the pattern 13 in Fig. 7).

Conclusion

The paper presents modifications of two classical algorithms for constructing surfaces in three-dimensional space. The input data for the algorithms is values of a real-valued function computed on a rectangular or parallelepipedal grid in a two- or three-dimensional space. The grid, possibly, fills a set of an arbitrary shape (not necessarily a rectangle or parallelepiped).

The first algorithm deals with constructing the surface of the graph of a discontinuous function of two arguments. Another feature of the function to be visualized is that it can take infinite values. During computations, infinite values are replaced by some very large constant. Algorithms presented in popular scientific systems, e.g. MatLab, MathCAD, GNUPlot, etc., construct the surface of the graph simply by connecting existing points at the grid nodes with triangles. This leads to appearance of vertical “walls” in places where the function is discontinuous, as well as to appearance of “plateaus” at large values where the function is infinite. This deforms the graph and makes poorly distinguishable parts of it, which are interesting to the researcher. To display adequately discontinuities, in the proposed algorithm, a triangular

element is not added to the formed surface if the absolute value of the difference between the applicates of at least two of its vertices is greater than a given threshold. To remove high "plateaus," the algorithm removes from the set those nodes, in which the function magnitude is greater than a specified value. Due to this filtering, the algorithm is adapted to a situation when the shape of the set, on which the graph is visualized, may be significantly non-rectangular. This helps to visualize also functions computed on non-rectangular sets.

If the grid cell sizes are not too small, a "fringe" may appear along the edges of breaks and transitions to plateaus. The break lines are located quite arbitrary with respect to grid nodes, somewhere closer, somewhere further, so the values at the nodes can differ from each other, and the corresponding points are located at different heights. Additional smoothing of the edge of the surface may be required, although with sufficiently fine grid steps this fringe is not too large.

The second algorithm deals with visualization of level sets of a function of three arguments, which are regions of the three-dimensional space where the function does not exceed a given value. A straightforward way of visualizing this kind of objects is associated with formation of "voxel" sets when a rectangular "brick" of the corresponding grid cell is attached to each point of the cloud. However, in this case, the surface of the set turns out to be too "creased", especially if the size of the cells is not too small in comparison with the size of the set itself. Traditionally, the Marching Cubes algorithm, algorithms of the Laplacian family, or their combinations are used for smoothing such a surface. However, even if the original set had symmetry with respect to coordinate planes, axes, or some point, the classic Marching Cubes algorithm generates an asymmetrical triangulation of the surface. Therefore, with further processing of such a surface, the asymmetry may increase and become obvious. A modification of the Marching Cubes algorithm is proposed, which triangulates the surface preserving the original symmetries if some present. However, the resultant symmetry violations, when applying the classic Marching Cubes algorithm, have the size of a grid cell and are not visible if the size of the grid cell is small compared to the size of the set. The proposed modification is useful primarily for visualizing small sets.

References

1. W.E. Lorensen, H.E. Cline, Marching Cubes: A high resolution 3D surface construction algorithm // ACM SIGGRAPH Computer Graphics, Vol. 21, No. 4, 1987, pp. 163–169.
2. S. Canann, M. Stephenson, T. Blacker, Optismoothing: An optimization-driven approach to mesh smoothing // Finite Elements in Analysis and Design, Vol. 13, No. 2–3, 1993, pp. 185–190.
3. L. Freitag, On combining Laplacian and optimization based mesh smoothing techniques // Proceedings of the 1997 Joint Summer Meeting of American Society of Mechanical Engineers (ASME) American Society of Civil Engineers (ASCE) and Society of Engineers Science (SES), 1997, pp. 37–44.
4. N. Amenta, M. Bern, D. Eppstein, Optimal point placement for mesh smoothing // Journal of Algorithms, Vol. 30, No. 2, 1999, pp. 302–322.
5. J. Vollmer, R. Mencl, H. Muller, Improved Laplacian Smoothing of Noisy Surface Meshes // Computer Graphics Forum, Vol. 18, No. 3, 1999, pp. 131–138.
6. N.V. Munts, S.S. Kumkov, Chislennyj metod resheniya differencial'nyh igr bystrodejstviya s liniej zhizni // Matematicheskaya teoriya igr i ee prilozheniya, V. 10, № 3, 2018, pp. 48–75. (in Russian)
7. N.V. Munts, S.S. Kumkov, Convergence of Numerical Method for Time-Optimal Differential Games with Lifeline // Annals of the International Society of Dynamic Games, Vol. 17, Games of Conflict, Evolutionary Games, Economic Games, and Games Involving Common Interest. Basel: Birkhäuser, 2020. pp. 103–132.

8. L.E. Dubins, On curves of minimal length with a constraint on average curvature and with prescribed initial and terminal positions and tangents // American Journal of Mathematics, Vol. 79, No. 3, 1957, pp. 497–516.
9. J.-P. Laumond (ed.), Robot Motion Planning and Control, Lecture Notes in Control and Information Sciences, Vol. 229. Springer-Verlag, Berlin Heidelberg, 1998. 354 p.
10. Y. Meyer, T. Shima, P. Isaiah, On Dubins paths to intercept a moving target // Automatica, Vol. 59, 2015, pp. 256–263.
11. E. Bakolas, P. Tsiotras, Optimal synthesis of the asymmetric sinistral/dextral Markov-Dubins problem // Journal of Optimization Theory and Applications, Vol. 150, No. 2, 2011, pp. 233–250.
12. E.J. Cockayne, G.W.C. Hall, Plane Motion of a Particle Subject to Curvature Constraints // SIAM Journal on Control, Vol. 13, 1975, pp. 197–220.
13. Y.I. Berdyshev, Nelinejnye zadachi posledovatel'nogo upravleniya i ih prilozhenie [Nonlinear problems of sequential control and their application]. Ekaterinburg: IMM UrO RAN, 2015. 193 p. (in Russian)
14. A.A. Fedotov, V.S. Patsko, Investigation of Reachable Set at Instant for the Dubins' Car // Proceedings of the 58th Israel Annual Conference on Aerospace Sciences, Tel-Aviv & Haifa, Israel, March 14–15, 2018, ThL2T1.4, pp. 1655–1669.
15. R. Isaacs, Differential Games. John Wiley and Sons, New York, 1965. 384 p.
16. R. Ajzeks, Differencial'nye igry [Differential games]. M.: Mir, 1967. 479 p. (in Russian)
17. N. Botkin, K.-H. Hoffmann, N. Mayer, V. Turova, Computation of value functions in nonlinear differential games with state constraints // D. Hömberg, F. Tröltzsch (eds.), System Modeling and Optimization. CSMO 2011. IFIP Advances in Information and Communication Technology, Vol. 391, 2013, pp. 235–244.