sciendo

TATRA MOUNTAINS
Mathematical Publications

DOI: 10.2478/tmmp-2023-0033
Tatra Mt. Math. Publ. **00** (xxxx), 1–40

DIRECT SIMPLE COMPUTATION OF MIDDLE SURFACE BETWEEN 3D POINT CLOUDS AND/OR DISCRETE SURFACES BY TRACKING SOURCES IN DISTANCE FUNCTION CALCULATION ALGORITHMS

Balázs Kósa — Karol Mikula

Department of Mathematics and Descriptive Geometry, Faculty of Civil Engineering, Slovak University of Technology in Bratislava, SLOVAKIA

ABSTRACT. In this paper, we introduce novel methods for computing middle surfaces between various 3D data sets such as point clouds and/or discrete surfaces. Traditionally the middle surface is obtained by detecting singularities in computed distance function such as ridges, triple junctions, etc. It requires to compute second order differential characteristics, and also some kinds of heuristics must be applied. Opposite to that, we determine the middle surface just from computing the distance function itself which is a fast and simple approach. We present and compare the results of the fast sweeping method, the vector distance transform algorithm, the fast marching method, and the Dijkstra-Pythagoras method in finding the middle surface between 3D data sets.

1. Introduction

Finding an optimal middle surface for a data set is a crucial task in many applications such as computational geometry, surface representation and reconstruction, image processing and computer vision or mesh generation. In optimal mesh generation [7], for example, the information about the middle surface can be used to densify or coarsen the computational grid in the computational domain. For this reason, having an efficient method that fulfils such needs is very

2020 Mathematics Subject Classification: 65M06, 65Y20, 68U05, 53A05.
Keywords: Middle surface, 3D point cloud, triangulated surface, fast sweeping method, fast marching method, vector distance transform, Dijkstra-Pythagoras method.
This work was supported by grants APVV-19-0460, and VEGA 1/0436/20.

important. Very often the middle surface, which algorithms are seeking for, is a middle axis of a closed curve or a surface, see e.g. [6, 9, 11]. Such algorithms can be complicated because they utilize the second order derivatives of the computed distance function in order to detect its ridges, junctions and other singularities which often requires some kinds of heuristics, see also [7]. In cases where we can distinguish individual separated or labelled shapes between which we want to find the middle surface, a much more straightforward approach can be derived. We show how algorithms designed for distance function calculation can be adjusted and utilized in these cases to obtain the middle surface already during the computations of the distance function itself. Opposite to methods that utilize second order derivatives of the computed distance function, we only adjust the distance function calculation algorithms. This makes our methods simple, efficient and easy to implement.

A distance function to an object is a useful tool in a variety of disciplines. For this reason, over the years many algorithms, which were optimized to obtain the most accurate result as fast as possible, have been developed, see e.g. [5]. We provide a short description of four such algorithms and show how they can be implemented to calculate the distance function on a uniform voxel grid for 3D objects represented either by point clouds or triangulated surfaces. To compare the algorithms, we applied them to several data sets and measured their accuracy and speed.

After providing a sufficient explanation of the methods with detailed pseudocodes for each of them, we describe how we use them to find the middle surface. We will see that all it needs is a few natural changes in the implementation to achieve this goal. We test our approaches on several experiments and present the results subsequently.

2. Numerical methods

In computational mathematics, the notion of distance function is used for the result of distance computation. In this section, we will discuss common numerical methods used for this task. Following [12] the presented methods are classified according to two criteria:

(1) Distance definition: The distance function can be calculated as a solution of the so called Eikonal equation or by the Euclidean distance computation.

(2) Voxel visit order strategy: We will analyse sweeping and wavefront methods.

Our goal is to demonstrate how methods falling under these categories can be used to find the middle surface between two or more input data sets. We will discuss and analyse these methods: the fast sweeping method (FSM) [14], the vector distance transform (VDT) algorithm [3], the fast marching method (FMM) [10] and the Dijkstra-Pythagoras (DP) method [12]. Table 1 shows the classification of the four studied methods.

TABLE 1. Overview of methods used to compute the distance function. Rows represent the distance definition. Columns represent the voxel visit order strategy.

	Sweeping	**Wavefront**
Eikonal equation	**Fast sweeping method**	**Fast marching method**
Euclidean distance	**Vector distance transform**	**Dijkstra-Pythagoras**

2.1. Basic definitions

The distance function will be calculated on the computational domain Ω, $\Omega \subseteq \mathbb{R}^n$. A data set Ω_0, to which we want to compute distance function d, will be a subset of Ω, $\Omega_0 \subseteq \Omega$. In this paper, we work with 3D objects so we limit the dimension of Ω to $n = 3$. With this notation of the domain, we can define the distance function as $d : \Omega \to \mathbb{R}$. On subset Ω_0 the distance should be 0, thus we get the boundary condition

$$d(x) = 0, \quad x \in \Omega_0 \subseteq \Omega. \tag{1}$$

Then the task is to calculate

$$d(x), \ x \in \Omega \setminus \Omega_0.$$

2.1.1. Distance definition

For the numerical methods, the computational domain Ω will be discretized into a finite number of voxels with edge size h. The number of voxels will be denoted as N_i along the x axis, N_j along the y axis, and N_k along the z axis. In the obtained computational grid, the function d will be calculated at the center of every voxel, the so-called grid points.

The **eikonal equation** is given by

$$|\nabla d(x)| = 1, \quad x \in \Omega. \tag{2}$$

This equation will be coupled with the boundary condition (1). For the discretization of (2), we denote grid points of Ω by $x_{i,j,k}$ and the numerical solution

of the distance function at $x_{i,j,k}$ as $d_{i,j,k}$. The discretization of (2) at interior grid points is done according to the Godunov upwind difference scheme [8]:

$$\begin{gathered}\left[(d_{i,j,k}-d_{x\ \min})^{+}\right]^2+\left[(d_{i,j,k}-d_{y\ \min})^{+}\right]^2+\left[(d_{i,j,k}-d_{z\ \min})^{+}\right]^2=h^2,\\ i=1,\ldots,I-1,\quad j=1,\ldots,J-1,\quad k=1,\ldots,K-1,\\ d_{x\ \min}=\min\left(d_{i,j-1,k},\ d_{i,j+1,k}\right),\\ d_{y\ \min}=\min\left(d_{i-1,j,k},\ d_{i+1,j,k}\right),\\ d_{z\ \min}=\min\left(d_{i,j,k-1},\ d_{i,j,k+1}\right),\end{gathered}\tag{3}$$

$$(x)^{+}=\begin{cases}x,\ x>0,\\ 0,\ x\le 0.\end{cases}$$

At the boundary of Ω we use one sided difference. This enforces that the solution at every voxel center is defined by the smaller values of neighbouring grid points. Eikonal-based methods calculate the distance function by applying the described numerical scheme (3).

Euclidean distance between two points will be defined according to the Pythagoras' theorem. For

$$a=(a_x,a_y,a_z)\in\Omega,\qquad b=(b_x,b_y,b_z)\in\Omega$$

we define

$$d\left(a,b\right)=\sqrt{(a_x-b_x)^2+\ (a_y-b_y)^2+\ (a_z-b_z)^2}.\tag{4}$$

2.1.2. Voxel visit order strategy

For algorithms with the **sweeping** approach, Gauss-Seidel iterations with alternating sweeping orderings are used. This allows the methods to pass through the voxels multiple times. For three dimensions we sweep the computational domain with eight alternating orderings:

1. $i = 1 : N_i,\quad j = 1 : N_j,\quad k = 1 : N_k;$
2. $i = 1 : N_i,\quad j = 1 : N_j,\quad k = N_k : 1;$
3. $i = 1 : N_i,\quad j = N_j : 1,\quad k = 1 : N_k;$
4. $i = 1 : N_i,\quad j = N_j : 1,\quad k = N_k : 1;$
5. $i = N_i : 1,\quad j = 1 : N_j,\quad k = 1 : N_k;$
6. $i = N_i : 1,\quad j = 1 : N_j,\quad k = N_k : 1;$
7. $i = N_i : 1,\quad j = N_j : 1,\quad k = 1 : N_k;$
8. $i = N_i : 1,\quad j = N_j : 1,\quad k = N_k : 1.$

To work with these sweeps, we will define the following sets:

$$\begin{aligned}
i_{\text{sweep}} =& \big\{\{0, N_i-1, 1\}, \{0, N_i-1, 1\}, \{0, N_i-1, 1\}, \{0, N_i-1, 1\}, \\
& \{N_i-1, 0, -1\}, \{N_i-1, 0, -1\}, \{N_i-1, 0, -1\}, \{N_i-1, 0, -1\}\big\} \\
j_{\text{sweep}} =& \big\{\{0, N_j-1, 1\}, \{0, N_j-1, 1\}, \{N_j-1, 0, -1\}, \{N_j-1, 0, -1\}, \\
& \{0, N_j-1, 1\}, \{0, N_j-1, 1\}, \{N_j-1, 0, -1\}, \{N_j-1, 0, -1\}\big\} \\
k_{\text{sweep}} =& \big\{\{0, N_k-1, 1\}, \{N_k-1, 0, -1\}, \{0, N_k-1, 1\}, \{N_k-1, 0, -1\}, \\
& \{0, N_k-1, 1\}, \{N_k-1, 0, -1\}, \{0, N_k-1, 1\}, \{N_k-1, 0, -1\}\big\}.
\end{aligned} \tag{5}$$

The different algorithms analyse a certain set of neighbouring voxels in every iteration. This can be the set of 6 closest neighbours

$$P^1 = \big\{(r,s,t)\,;\ r,s,t \in \{-1,0,1\}\,;\ |r|+|s|+|t| = 1\big\}, \tag{6}$$

or the set including also the diagonal voxels, the set of all 26 neighbours

$$P^2 = \big\{(r,s,t)\,;\ r,s,t \in \{-1,0,1\}\,;\ |r|+|s|+|t| = c;\ c \in \{1,2,3\}\big\}. \tag{7}$$

In the **wavefront methods** at every grid point, we assign the final value already in the first pass. To ensure this, the algorithms have to be set up in a way that every voxel is visited in the correct order, starting with the voxel nearest to Ω_0 and ending with the furthest. For this, a data structure called *min-priority-heap* [2] is utilized. In this structure whenever a change occurs the elements are rearranged so the element with the smallest value is on top. For wavefront algorithms at the beginning, we store all grid points that enforce the boundary condition in such a heap, with their distance value d and their location in the grid. In every iteration, we can immediately obtain the grid point with the smallest value of $d\,(x)$. As the front moves on, new elements are added to the heap. For easy updates of distance values at grid points already saved in the heap, additional information about their location in the heap should be maintained.

In the next subsections, we will go through the implementation of the mentioned methods, so we will be able to describe how to change them for the task of computing the middle surface. To that goal, we start with the description of how to implement the initialization of the distance function to ensure the boundary condition (1).

2.2. Initialization

For every point x of the input data set Ω_0 the function d should fulfill (1). When we implement a method for the calculation of d, we need to find a way to fulfill this condition. If point x would coincide with the voxel center, in an array representing d we could just set the values to 0 for every such point x. Unfortunately, this is usually not the case.

While working with point cloud data, to fulfill (1), we initialize the function d as follows. We find the 8 nearest grid points to every point in the cloud and calculate the exact distance for these points from the corresponding point cloud element. The smallest possible distance will be saved at grid points when exploring subsequently all point cloud elements. These initialized values will be fixed in further calculations. Some of the algorithms described in the following sections use the cloud points as "sources" to calculate the distance function at other grid points. For this reason, in the initialization, we will keep track of this information as well. We can easily do this by setting the index of the source cloud point to the fixed grid points which will refer to the coordinates of the source. At other than fixed grid points, we set d to a high enough number, which is bigger than the biggest possible distance in the grid. To simplify this, we can use $+\infty$, which for example when we implement the algorithm in C or C++ can be substituted by the maximum *double* value.

In Alg. 1, we show how the described initialization can be easily implemented.

2.3. Fast sweeping method

The fast sweeping method (FSM) [14] is an iterative algorithm with alternating sweeps (5) used for the numerical solution of the Eikonal equation (3). It can be applied in any number of dimensions for a rectangular computational grid. The value of $d\,(x)$ at any grid point will never increase because an update rule is implemented by which the new value of the distance function is saved only if it is smaller than the current value. This enforces the correct value not to change at later iterations.

Let us denote in equation (3) the unknown as $x = d_{i,j,k}$ and the coefficients as

$$a_1 = d_{x\ \min}, \quad a_2 = d_{y\ \min}, \quad a_3 = d_{z\ \min}.$$

Then the unique solution, denoted by $\bar{x}$, to the equation

$$\left[(x-a_1)^+\right]^2 + \left[(x-a_2)^+\right]^2 + \left[(x-a_3)^+\right]^2 = h^2 \tag{8}$$

can be found as follows. We order a_1, a_2, a_3 in the increasing order. For generality we assume $a_1 \leq a_2 \leq a_3$. There is an integer p, $1 \leq p \leq 3$, such that $\bar{x}$ is the unique solution that satisfies

$$(x-a_1)^2 + (x-a_2)^2 + (x-a_3)^2 = h^2$$

and

$$a_p < \bar{x} < a_{p+1}. \tag{9}$$

Algorithm 1 Initialization of distance function to the point cloud data

Input: Point cloud data:
pc_l - (x, y, z) coordinates of the lth point,
N - number of points.
Input: 3D grid with voxel edge size h and dimensions N_i, N_j, N_k.
Declaration: Arrays:
$d_{i,j,k}$ - value of distance function at grid point (i, j, k),
$c_{i,j,k}$ - (x, y, z) coordinates of grid point (i, j, k),
$f_{i,j,k}$ - determines if $d_{i,j,k}$ is fixed at (i, j, k),
$s_{i,j,k}$ - source for $d_{i,j,k}$ calculation at (i, j, k).

Set: $d_{i,j,k}$ to $+\infty$, $f_{i,j,k}$ to $false$, $s_{i,j,k}$ to $unknown$
Calculate: $c_{i,j,k}$
for $(l = 0; l < N; l = l + 1)$ **do**
  $i_{\text{first}} = RoundDown\Big(\big(pc_l.x - \min\big(c_{i,j,k}.x\big)\big)/h\Big)$
  $j_{\text{first}} = RoundDown\Big(\big(pc_l.y - \min\big(c_{i,j,k}.y\big)\big)/h\Big)$
  $k_{\text{first}} = RoundDown\Big(\big(pc_l.z - \min\big(c_{i,j,k}.z\big)\big)/h\Big)$
  for $(i = i_{\text{first}}; i \le i_{\text{first}} + 1; i = i + 1)$ **do**
    for $(j = j_{\text{first}}; j \le j_{\text{first}} + 1; j = j + 1)$ **do**
      for $(k = k_{\text{first}}; k \le k_{\text{first}} + 1; k = k + 1)$ **do**
        $d_{\text{new}} = d\left(pc_l, c_{i,j,k}\right)$ ▷ Calculated by (4).
        if $d_{\text{new}} < d_{i,j,k}$ **then**
          $d_{i,j,k} = d_{\text{new}}$
          $f_{i,j,k} = \text{true}$
          $s_{i,j,k} = pc_l$
        end if
      end for
    end for
  end for
end for

To find $\bar{x}$ we start with $p = 1$. If $\tilde{x} = a_1 + h \le a_2$ then $\bar{x} = \tilde{x}$. Otherwise we have to find the solution of the quadratic equation

$$(x - a_1)^2 + (x - a_2)^2 = h^2$$

that satisfies $\tilde{x} > a_2$. We always take the maximum of the two solutions as our $\tilde{x}$. If $\tilde{x} \le a_3$ then $\bar{x} = \tilde{x}$. If we still do not have $\tilde{x}$ which satisfies all the conditions as the third step we compute the solution of the quadratic equation

$$(x - a_1)^2 + (x - a_2)^2 + (x - a_3)^2 = h^2$$

which will satisfy (2.3).

Only a finite amount of iterations is needed to obtain the solution, thus the complexity of the method is $O(N)$, where N is the total number of grid points in the computational domain. This method is simple to implement, as it can be seen in the provided pseudo-code Alg. 2.

Algorithm 2 Fast sweeping method

Input: `From Alg.1: 3D grid,` $d_{i,j,k}$, $f_{i,j,k}$

for $(l = 0; l < 8; l = l + 1)$ **do**
  for $(i = i_{\text{sweep}}[l, 0]; i \leq i_{\text{sweep}}[l, 1]; i = i + i_{\text{sweep}}[l, 2])$ **do**
    for $(j = j_{\text{sweep}}[l, 0]; j \leq j_{\text{sweep}}[l, 1]; j = j + j_{\text{sweep}}[l, 2])$ **do**
      for $(k = k_{\text{sweep}}[l, 0]; k \leq k_{\text{sweep}}[l, 1]; k = k + k_{\text{sweep}}[l, 2])$ **do**
        if $f_{i,j,k}$ `is not true` **then**
          $a_1 = \min\left(d_{i+1,j,k}, d_{i-1,j,k}\right)$
          $a_2 = \min\left(d_{i,j+1,k}, d_{i,j-1,k}\right)$
          $a_3 = \min\left(d_{i,j,k+1}, d_{i,j,k-1}\right)$ ▷ Use $+\infty$ if (i, j, k) is out of bounds.
          `Sort` $\{a_1, a_2, a_3\}$ `from lowest to highest.`
          $d_{\text{new}} = a_1 + h$
          if $d_{\text{new}} > a_2$ **then**
            $d_{\text{new}} = \underset{x}{MaxSolution}\left((x - a_1)^2 + (x - a_2)^2 = h^2\right)$
            if $d_{\text{new}} > a_3$ **then**
              $d_{\text{new}} = \underset{x}{MaxSolution}\left((x - a_1)^2 + (x - a_2)^2 + (x - a_3)^2 = h^2\right)$
            end if
          end if
          if $d_{\text{new}} < d_{i,j,k}$ **then**
            $d_{i,j,k} = d_{\text{new}}$
          end if
        end if
      end for
    end for
  end for
end for

2.4. Vector distance transform

For the implementation of the vector distance transform (VDT) [3] algorithm, we follow the implementation used in [12] and extend it to 3D calculations. Comparing the pseudo-code of this method, Alg. 3 with Alg. 2, we can immediately see that the algorithm also uses Gauss-Seidel iterations alternating the sweeping ordering (5). This shows that the information propagates in the same manner, and we can use the same update rules for the values of $d(x)$.

The main difference between VDT and FSM lies in the method of how the values of $d(x)$ are calculated at the not fixed grid points. While FSM calculates new distance values from the values of neighbouring grid points, VDT only checks the source of the neighbours to calculate the smallest possible exact Euclidean distance (4) at the current grid point. For this reason, we need to keep track of the sources, and every time we calculate a smaller distance value we update this information. This method yields $O(N)$ complexity as well.

Algorithm 3 Vector distance transform

Input: From Alg.1: 3D grid, $d_{i,j,k}$, $c_{i,j,k}$, $f_{i,j,k}$, $s_{i,j,k}$

for $(l = 0; l < 8; l = l + 1)$ **do**
  for $(i = i_{\text{sweep}}[l, 0]; i \leq i_{\text{sweep}}[l, 1]; i = i + i_{\text{sweep}}[l, 2])$ **do**
    for $(j = j_{\text{sweep}}[l, 0]; j \leq j_{\text{sweep}}[l, 1]; j = j + j_{\text{sweep}}[l, 2])$ **do**
      for $(k = k_{\text{sweep}}[l, 0]; k \leq k_{\text{sweep}}[l, 1]; k = k + k_{\text{sweep}}[l, 2])$ **do**
        if $f_{i,j,k}$ is not true **then**
          for all $\{(i + r, j + s, k + t); (r, s, t) \in P^1\}$ not out of bound **do**
            if $s_{i+r,j+s,k+t}$ is known **then**
              $d_{\text{new}} = d(s_{i+r,j+s,k+t}, c_{i,j,k})$ ▷ Calculated by (4).
              if $d_{\text{new}} < d_{i,j,k}$ **then**
                $d_{i,j,k} = d_{\text{new}}$
                $s_{i,j,k} = s_{i+r,j+s,k+t}$
              end if
            end if
          end for
        end if
      end for
    end for
  end for
end for

2.5. Fast marching method

Similarly, as the FSM algorithm, the fast marching method (FMM) [10] gives results based on the solution of the Eikonal equation. While FSM tests the possible solutions of the alternatives of (2.3) by going through them in the right order, FMM sets up the solution immediately according to which coefficients are already calculated. In the construction of this algorithm, one-way propagation of information is utilized, secured by the upwind difference structure of discretization. To properly monitor this propagation the visiting of grid points is tracked throughout the execution of the algorithm. The solution is built outward from the smallest values, which, as seen in the initialization phase in Section 2.2, are at the grid points nearest to the points in the cloud. These elements are gathered in a *min-priority-heap* and marked as 'to be visited', while all others are marked 'unvisited'. In Alg. 4 we can see how the heap is used. While the solution from the initialized grid points is marched forward the values from the heap are finalized, marked as 'visited', and new points are brought into this set. FMM works, because we always select the grid point with the smallest value from the heap to calculate the values of the neighbouring elements, thus 'unvisited' grid points will not have any effect on the solution.

The complexity of the FMM algorithm is of order $O\left(N \log_2 N\right)$, because we visit every grid point once and the operations of the *min-priority-heap* have a complexity of $O\left(\log_2 N\right)$.

Algorithm 4 Fast marching method

Input: From Alg.1: 3D grid, $d_{i,j,k}$, $f_{i,j,k}$
Declaration: $v_{i,j,k}$ will hold the visiting values of grid points
'unvisited'=0, 'to be visited'=1, 'visited'=2
Declaration: *heap* container will be a *min-priority-heap*
Initialization: $\forall f_{i,j,k} = \text{true} : \left\{v_{i,j,k} = 1;\ heap.InsertNode(d_{i,j,k})\right\}$ else: $v_{i,j,k} = 0$

while *heap* is not empty **do**
  $(i,j,k) = heap.GetRoot()$ ▷ Obtain (i,j,k) with minimum d and delete from *heap*.
  for all $\left\{(i+r, j+s, k+t)\,; (r,s,t) \in P^1\right\}$, not out of bound **do**
    if ($f_{i+r,j+s,k+t}$ is $false$) and ($v_{i+r,j+s,k+t} = 0$ or $v_{i+r,j+s,k+t} = 1$) **then**
      $x = \min\left(d_{i+r+1,j+s,k+t}, d_{i+r-1,j+s,k+t}\right)$
      $y = \min\left(d_{i+r,j+s+1,k+t}, d_{i+r,j+s-1,k+t}\right)$
      $z = \min\left(d_{i+r,j+s,k+t+1}, d_{i+r,j+s,k+t-1}\right)$ ▷ Use $+\infty$ if $(i+r, j+s, k+t)$ is out of bounds.
      $a = b = c = 0$
      if $x \neq +\infty$ **then** $a = a + 1;\ b = b + x;\ c = c + x^2$
      if $y \neq +\infty$ **then** $a = a + 1;\ b = b + y;\ c = c + y^2$
      if $z \neq +\infty$ **then** $a = a + 1;\ b = b + z;\ c = c + z^2$
      $a = a * \left(1/h^2\right)$
      $b = (-2) * b * \left(1/h^2\right)$
      $c = c * \left(1/h^2\right) - 1.0$
      $d_{\text{new}} = \frac{-b+\sqrt{b^2-4*a*c}}{2*a}$
      if $d_{\text{new}} < d_{i+r,j+s,k+t}$ **then**
        $d_{i+r,j+s,k+t} = d_{\text{new}}$
        if $v_{i+r,j+s,k+t} = 0$ **then**
          $heap.InsertNode(d_{i+r,j+s,k+t})$
          $v_{i+r,j+s,k+t} = 1$
        else
          $heap.DecreaseKey\left((i+r, j+s, k+t), d_{\text{new}}\right)$
        end if
      end if
    end if
  end for
  $v_{i,j,k} = 2$
end while

2.6. Dijkstra-Pythagoras method

The Dijkstra-Pythagoras (DP) method was introduced in [12]. In [12], a gap was detected for a wave-front type method, like FMM, which has lead to a new DP method giving the results with the exact Euclidean distance. Dijkstra-Pythagoras algorithm uses visiting rules and a *min-priority-heap* as described in the FMM algorithm but utilizes the source tracking for distance

calculation as in the VDT method. In [12] the pseudo-code of the method was outlined in a 2D pixel grid with pixel edge size 1. We extend it to the 3D voxel grid and introduce a substantial modification. In the initial proposal, the algorithm analyses all neighbours of grid points. We changed this to include only the closest ones, which in 3D are the voxels from the set P^1 (6). We found that with this modification the method becomes much faster and its precision stays approximately the same. In Alg. 5 we show the detailed pseudo-code with our changes.

The logic of the method is based on a two-fold relaxation of $d(x)$ values. As in FMM, every cycle of the algorithm starts with the grid point of the smallest d value popped from a *min-priority-heap*. The distance value of this point is checked to the sources of all its 'visited' neighbours. From all 6 possibilities the value is adjusted to the minimum before it is marked as 'visited' as well. Its source is selected accordingly. Then this method attempts to relax the 'unvisited' and 'to be visited' neighbours in a Dijkstra way. The distances for these grid points are updated according to the Pythagoras rule if the new value is smaller than the value already stored. Their sources are set to the source of the grid point by which they were updated. The neighbours which are 'unvisited' will be added to the heap. The algorithm runs till the heap is empty.

Similarly to FMM the complexity of this method is $O(N \log_2 N)$.

3. Numerical experiments - methods comparison

In this section, we compare the efficiency of the described algorithms and show that they can be used for computing the distance function to objects represented by a 3D point cloud and triangulated surface.

3.1. Comparing methods

For the first experiment, we will work with a cube with an edge size of 1.0, and its vertex with minimum coordinates at $(0.0, 0.0, 0.0)$. Around it, we construct a rectangular computational domain which is in every direction 0.4 times larger than the Cube. In this experiment, we discretize the computational domain in a way that some of the grid points will always lie on the surface of the Cube. Thus, we can set the distance function at these points to 0 during initialization.

With this setup, we computed the distance function for the Cube with the four algorithms on the computational domain discretized to a grid by voxels with different edge sizes, namely 0.2, 0.1, 0.05, 0.025, 0.0125, 0.00625, 0.003125. We demonstrate how the distance function looks like on these grids in Figure 1 calculated by the FSM algorithm.

Algorithm 5 Dijkstra-Pythagoras method

Input: From Alg.1: 3D grid, $d_{i,j,k}$, $c_{i,j,k}$, $f_{i,j,k}$, $s_{i,j,k}$
Declaration: $v_{i,j,k}$ will hold the visiting values of grid points
'unvisited'=0, 'to be visited'=1, 'visited'=2
Declaration: *heap* container will be a *min-priority-heap*
Initialization: $\forall f_{i,j,k} = \text{true} : \{v_{i,j,k} = 1;\ heap.InsertNode(d_{i,j,k})\}$ else: $v_{i,j,k} = 0$

while *heap* is not empty **do**
  $(i,j,k) = heap.GetRoot()$ ▷ Obtain (i,j,k) with minimum d and delete from *heap*.
  for all $\{(i+r, j+s, k+t); (r,s,t) \in P^1\}$, not out of bound **do**
    if $f_{i+r,j+s,k+t}$ is *false* and $v_{i+r,j+s,k+t} = 2$ **then**
      $d_{\text{new}} = d\left(s_{i+r,j+s,k+t}, c_{i,j,k}\right)$ ▷ Calculated by (4).
      if $d_{\text{new}} < d_{i,j,k}$ **then**
        $d_{i,j,k} = d_{\text{new}}$
        $s_{i,j,k} = s_{i+r,j+s,k+t}$
      end if
    end if
  end for
  $v_{i,j,k} = 2$
  for all $\{(i+r, j+s, k+t); (r,s,t) \in P^1\}$, not out of bound **do**
    if ($f_{i+r,j+s,k+t}$ is *false*) and ($v_{i+r,j+s,k+t} = 0$ or $v_{i+r,j+s,k+t} = 1$) **then**
      $d_{\text{new}} = d_{i,j,k} + h$
      if $d_{\text{new}} < d_{i+r,j+s,k+t}$ **then**
        $d_{i+r,j+s,k+t} = d_{\text{new}}$
        $s_{i+r,j+s,k+t} = s_{i,j,k}$
        if $v_{i+r,j+s,k+t} = 0$ **then**
          $heap.InsertNode(d_{i+r,j+s,k+t})$
          $v_{i+r,j+s,k+t} = 1$
        else
          $heap.DecreaseKey((i+r, j+s, k+t), d_{\text{new}})$
        end if
      end if
    end if
  end for
end while

To compare the accuracy of the algorithms, we calculated the mean squared difference from the exact solution for all grids. If we denote the exact solution as $\bar{d}_{i,j,k}$ at $x_{i,j,k} \in \Omega$ the mean squared difference can be calculated as

$$\left[\sum_{i=0}^{N_1-1}\sum_{j=0}^{N_2-1}\sum_{k=0}^{N_3-1}\left(\bar{d}_{i,j,k} - d_{i,j,k}\right)^2\right] \Big/ \left(N_1 * N_2 * N_3\right) \tag{10}$$

We are listing these results in Table 2. In the first column, we list the number of grid points in x, y, z directions of our computational grid. In the second column, we see the length of the voxel edges. In the following columns, we see the mean squared difference for FSM, VDT, FMM, and DP methods. We can

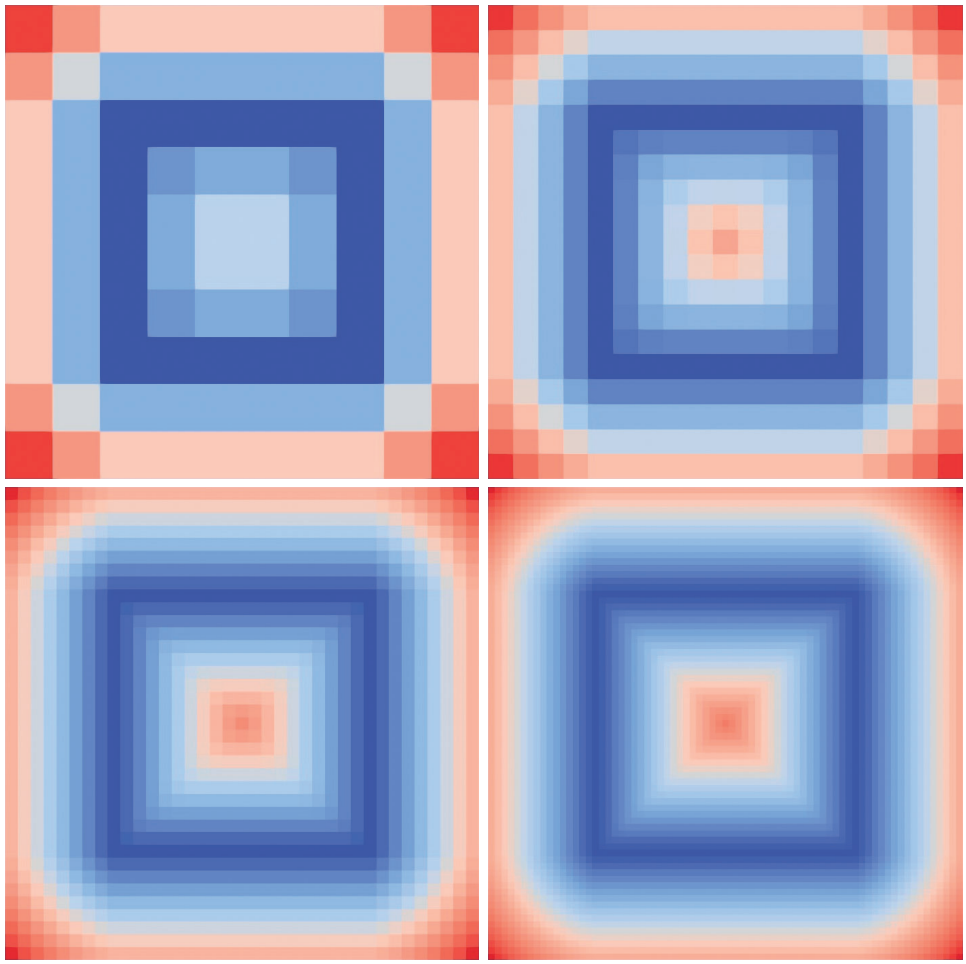

FIGURE 1. Distance function visualization for the Cube experiment. We visualize the section in a constant z plane for voxel edge sizes 0.2, 0.1, 0.05, 0.025. Values go from the highest dark red to the lowest dark blue. Results were calculated by FSM.

see that the results for the VDT and DP methods are basically 0, as we would have expected after stating the fact that they yield Euclidean distance results. The results of FSM and FMM are less accurate. We compare these results also visually in Figure 2 for computational grids with voxel edge size 0.1 and 0.025. We can see that the results for the pairs of FSM, FMM, and VDT, DP in this experiment are visually identical.

TABLE 2. Mean squared difference comparison for distance function calculation methods tested on the Cube experiment.

Number of grid points	Voxel edge size	FSM	VDT	FMM	DP
10^3	0.2	2.5692e-03	1.4791e-34	2.5692e-03	4.227801e-33
19^3	0.1	9.7901e-04	1.8869e-34	9.7902e-04	5.011068e-33
37^3	0.05	3.7697e-04	1.0579e-34	3.7697e-04	1.151981e-32
73^3	0.025	1.4352e-04	5.0275e-35	1.4352e-04	8.414407e-33
145^3	0.0125	5.3092e-05	2.5195e-35	5.3092e-05	9.454389e-33
289^3	0.00625	1.8949e-05	1.3057e-35	1.8949e-05	3.427338e-32
577^3	0.003125	6.5244e-06	6.5770e-36	6.5244e-06	1.404635e-31

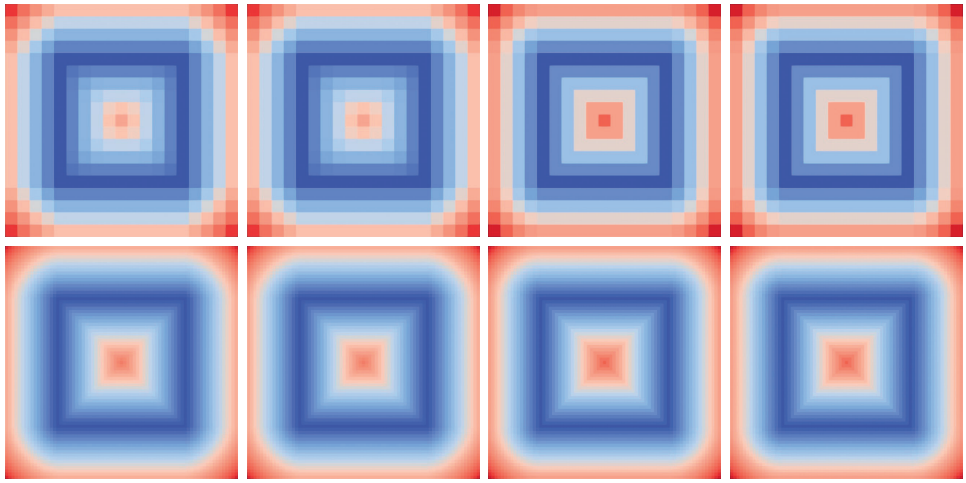

FIGURE 2. Visualization of results for distance function calculation in a constant z plane. In the first row, we see visualization for voxel edge size 0.1, in the second row for voxel edge size 0.025. In the first column, we see the result for the FSM algorithm, in second for FMM, in third for VDT and in the fourth for DP.

Besides the accuracy, for this experiment, we also measured the CPU time needed to calculate the distance function using different methods, reported in Table 3. Here again, first, the parameters of our grid are listed. In the third column, we list the CPU time for the initialization phase of the algorithms. The initialization is the same for all four methods. Because of the simplicity

of the experiment, this takes just a few seconds even for the finest grid. Comparing the results we see that concerning CPU time, the FSM algorithm outperforms all other methods.

TABLE 3. CPU time comparison for distance function calculation methods tested on the Cube experiment. CPU time was measured in seconds.

Number of grid points	Voxel edge size	Initialization	FSM	VDT	FMM	DP
10^3	0.2	0	0.001	0.002	0.001	0.001
19^3	0.1	0	0.002	0.017	0.002	0.002
37^3	0.05	0.001	0.008	0.029	0.016	0.015
73^3	0.025	0.014	0.059	0.186	0.18	0.138
145^3	0.0125	0.105	0.352	1.416	1.988	1.441
289^3	0.00625	0.816	2.881	11.352	26.109	15.425
577^3	0.003125	6.375	24.442	88.836	313.009	159.762

For further comparison of efficiency we choose a data set from [1] which will be used as a point cloud data and as a triangulated surface as well. This data set, seen in Figure 3, represents a teddy bear. Similarly, as in the previous experiment, we computed the distance function for the point cloud data with the four algorithms on computational grids with different voxel edge sizes 0.1, 0.05, 0.025, 0.0125, 0.00625, 0.003125, 0.0015625. Some of the results for distance functions calculated by the FSM algorithm can be seen in Figure 4. Visually there is no big difference between the results of the four algorithms.

We list the CPU time for calculation in Table 4. We added one more information in this table that was not listed in the previous experiment. In the third column, we list the number of fixed grid points produced by the initialization phase of the calculations. We will use this information for the comparison of distance function calculation in the case of the triangulated surface. In this experiment the points from the point cloud data do not coincide with points of the grid, thus the initialization was done by Alg. 1. The FSM algorithm is the fastest in this case as well.

3.2. Distance function to triangulated surfaces

With small changes, it is possible to easily modify the algorithm for the calculation of the distance function to triangulated surfaces. The most important changes which need to be applied concern the initialization phase. We demonstrate this in the pseudo-code Alg. 6. In this algorithm, we cycle through all

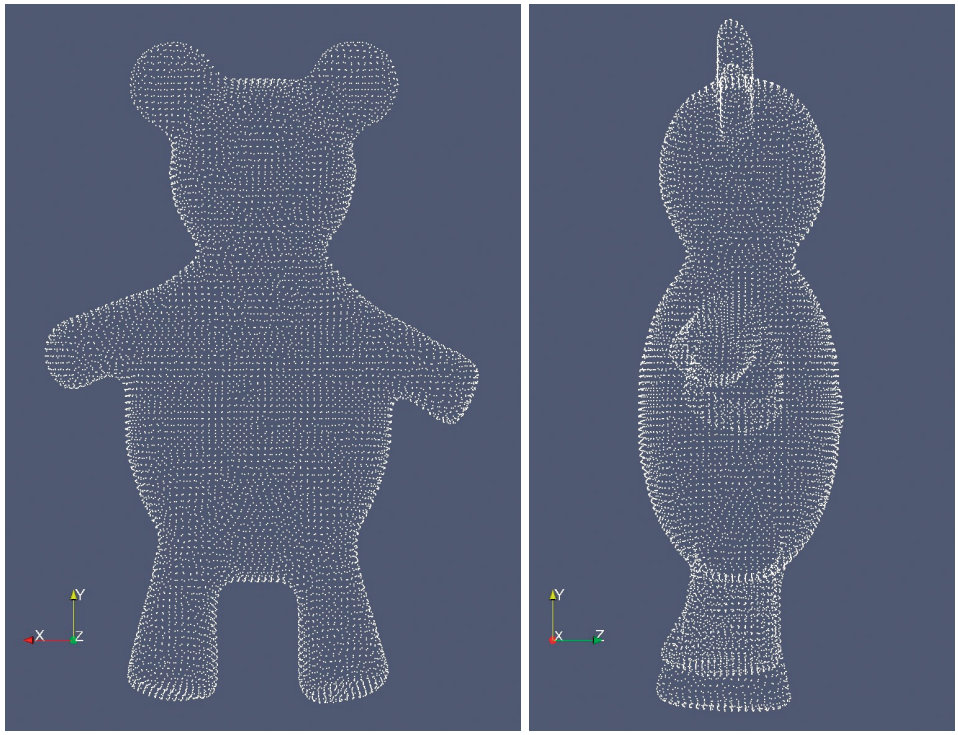

FIGURE 3. Teddy Bear point cloud data. In the left picture, we can see it from the front, in the right picture from the side.

TABLE 4. CPU time comparison for distance function calculation methods tested on the Teddy Bear point cloud data. CPU time was measured in seconds.

Number of grid points	Voxel edge size	Fixed points	Initial condition	FSM	VDT	FMM	DP
15 x 21 x 9	0.1	1003	0.001	0.002	0.002	0.001	0.001
29 x 41 x 17	0.05	3906	0.002	0.007	0.024	0.005	0.005
57 x 81 x 32	0.025	14428	0.007	0.036	0.083	0.059	0.054
113 x 161 x 62	0.0125	47575	0.042	0.253	0.562	0.658	0.605
224 x 321 x 122	0.00625	76231	0.293	1.891	4.135	9.079	7.638
447 x 640 x 242	0.003125	76384	2.235	15.335	33.126	121.185	103.515
893 x 1279 x 482	0.0015625	76384	19.085	118.572	254.895	1439.6	1221.13

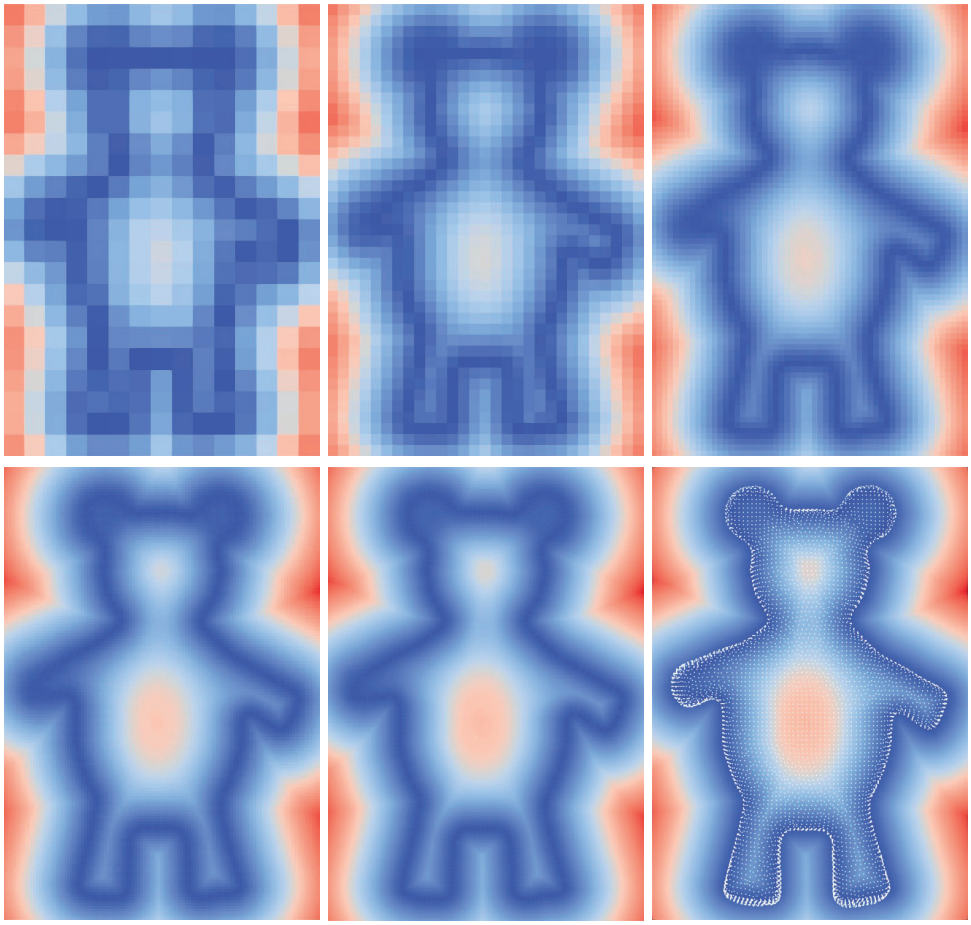

FIGURE 4. Distance function visualization of the Teddy Bear data set. Visualizing sections in a constant z plane for voxel edge sizes 0.1, 0.05, 0.025, 0.0125, 0.00625, 0.003125. In the last picture, we visualize the distance function with point cloud data. Values go from the highest dark red to the lowest dark blue. Results were calculated by FSM.

triangles in the triangulated surface. For every triangle, we find the grid points which are lying next to its surface. In these grid points, we calculate the distance from the triangle. For this, we use the method described in [4]. Similarly, as with the point cloud data, the values in these points will be fixed, but now as the source of distance computation, we will refer to the triangles. Regarding the algorithms FSM, VDT, FMM, and DP, the only changes will be in the pseudo-code Alg. 3 for VDT on the line 8 and in the pseudo-code Alg. 5 for DP on line 5 where the distance will be calculated between a point and a triangle.

Algorithm 6 Initialization of distance function to triangulated surface

Input: Triangulated surface:
tr_l - set of triangles,
N - number of triangles.
Input: 3D grid with voxel edge size h and dimensions N_i, N_j, N_k.
Declaration: Arrays:
$d_{i,j,k}$ - value of distance function at grid point (i,j,k),
$c_{i,j,k}$ - (x,y,z) coordinates of grid point (i,j,k),
$f_{i,j,k}$ - determines if $d_{i,j,k}$ is fixed at (i,j,k),
$s_{i,j,k}$ - source for $d_{i,j,k}$ calculation at (i,j,k).

Set $d_{i,j,k}$ to $+\infty$, $f_{i,j,k}$ to $false$, $s_{i,j,k}$ to $unknown$
Calculate: $c_{i,j,k}$
for $(l = 0; l < N; l = l + 1)$ **do**
    $gp_m = PointsAlongTriangle\,(tr_l)$ ▷ gp_m is a subset of the computational grid.
    $N_{gp} = NumberOfPointsIn(gp_m)$
    for $(m = 0; m < N_{gp}; m = m + 1)$ **do**
        $(i,j,k) = gp_m$
        $d_{\text{new}} = d\,(tr_l, c_{i,j,k})$ ▷ Distance of a point from a triangle.
        if $d_{\text{new}} < d_{i,j,k}$ **then**
            $d_{i,j,k} = d_{\text{new}}$
            $f_{i,j,k} = \text{true}$
            $s_{i,j,k} = tr_l$
        end if
    end for
end for

To demonstrate the results of these changes we will use again the Teddy Bear data set, but now as a triangulated surface as is seen in Figure 5. Similarly, as for the calculation to the point cloud data, we measured the CPU times and listed them in Table 5. If we compare this to Table 4 we can see the difference between the calculation of the distance function for point cloud data and a triangulated surface. The number of fixed points is much higher for the triangulated surface. This is because the initialization produces a "contiguous" volume around the triangles for every density of the grid, while around the point cloud data gaps can develop. We can see this also in Figure 6. Here we compare the distance function for both point cloud and triangulated surface by the results obtained by the FSM algorithm. (The difference in the visualization of the distance function calculated with the other algorithms is very small thus we provide just the visualization of the FSM algorithm.) The results for the triangulated surface, shown in the right column, are much smoother near the object as for the point cloud data, seen in the left column. While this difference has a little effect on the calculation time of the FSM and FMM algorithms, it increases the time for VDT and DP, significantly mainly for VDT method. This is because the implementation of FSM and FMM is independent of the initial data, but in VDT and

DP we work with the source as well and the calculation of the distance between a point and a triangle takes more time than the calculation between two points.

One may also observe that the increase in time is proportionally much higher for VDT than DP . This fact directly correlates to how the algorithms work and how many times we need to calculate the distance from a triangle to a point in each algorithm. From Alg. 3 for VDT , we see that potentially this operation is executed 8 (number of sweeps) $\times$ $\{N_i \times N_j \times N_k\}$ (number of grid points) $\times$ 6 (number of closest neighbours) times. Compared to this from Alg. 5 for DP we can determine that this number is much lower, namely $\{N_i \times N_j \times N_k\}$ (number of grid points) $\times$ 6 (number of closest neighbours), because every grid point goes through the *heap* container only once. In both cases the exact number of executions depends on the topology of the data set to which we calculate the distance function. We counted the number of calls of exact distance calculation for the Teddy Bear data set and present them in Table 6 to support our reasoning and compare this number also to the number of grid points.

To demonstrate a further example of distance function calculation on a triangulated surface we applied the algorithm on an additional data set. We obtained it from [13]. In Figure 7 we can see the triangulated surface of hand bones. With its many details and small parts, it is a good data set to show the accuracy of the results. We can see these in Figure 8. Here we choose planes in the computational domain in which we can see the most details.

TABLE 5. CPU time comparison for distance function calculation methods tested on Teddy Bear triangulated surface data. CPU time was measured in seconds.

Number of grid points	Voxel edge size	Fixed points	Initial condition	FSM	VDT	FMM	DP
15 x 21 x 9	0.1	1283	0.039	0.001	0.007	0.001	0.001
29 x 41 x 17	0.05	5147	0.048	0.007	0.056	0.005	0.008
57 x 81 x 32	0.025	20526	0.067	0.04	0.446	0.057	0.08
113 x 161 x 62	0.0125	82756	0.139	0.255	3.408	0.659	0.828
224 x 321 x 122	0.00625	331874	0.504	1.966	26.054	9.475	11.02
447 x 640 x 242	0.003125	1332140	2.877	16.211	203.035	120.117	131.252
893 x 1279 x 482	0.0015625	5346482	19.807	126.331	1578.07	1441.91	1394.25

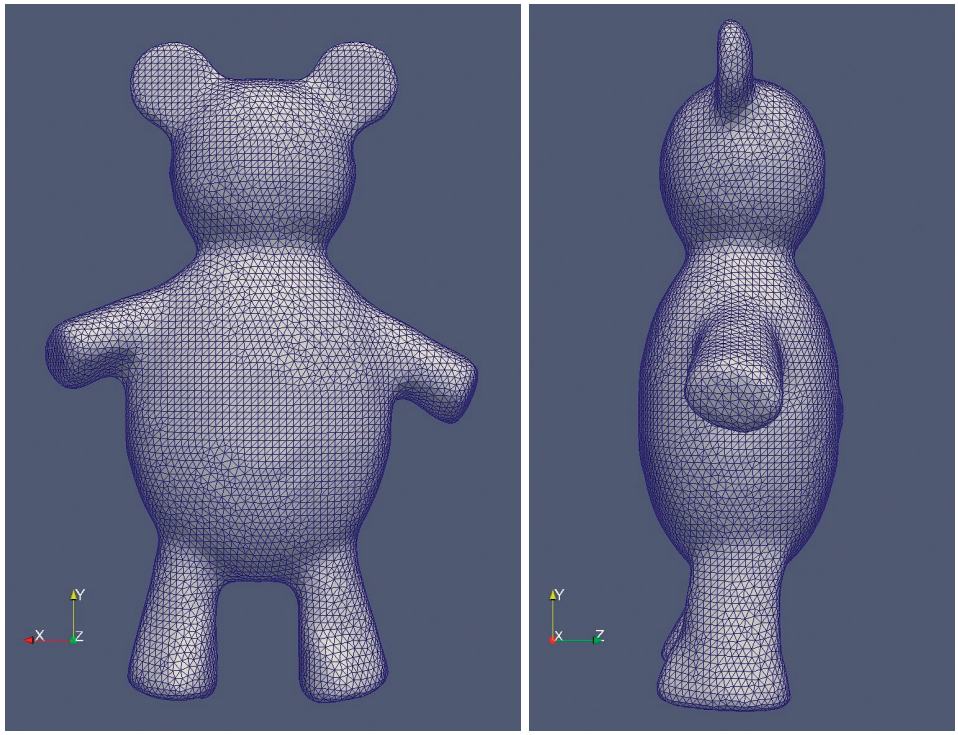

FIGURE 5. Teddy Bear triangulated surface data. In the left picture, we can see it from the front, in the right picture from the side.

TABLE 6. The number of execution for calculating the distance from a triangle to a point in VDT and DP algorithms for Teddy Bear triangulated surface data.

Number of grid points (NOGP)	Voxel edge size	Fixed points	VDT	VDT/ NOGP	DP	DP/ NOGP
15 x 21 x 9	0.1	1283	61043	21.5319	3605	1.2716
29 x 41 x 17	0.05	5147	629890	31.1626	40776	2.01732
57 x 81 x 32	0.025	20526	5359252	36.2739	364370	2.46623
113 x 161 x 62	0.0125	82756	43690983	38.7343	3067140	2.71918
224 x 321 x 122	0.00625	331874	349466298	39.8375	25049348	2.85551
447 x 640 x 242	0.003125	1332140	2808035440	40.5602	202613176	2.92661
893 x 1279 x 482	0.0015625	5346482	22508797856	40.8868	1631172583	2.96299

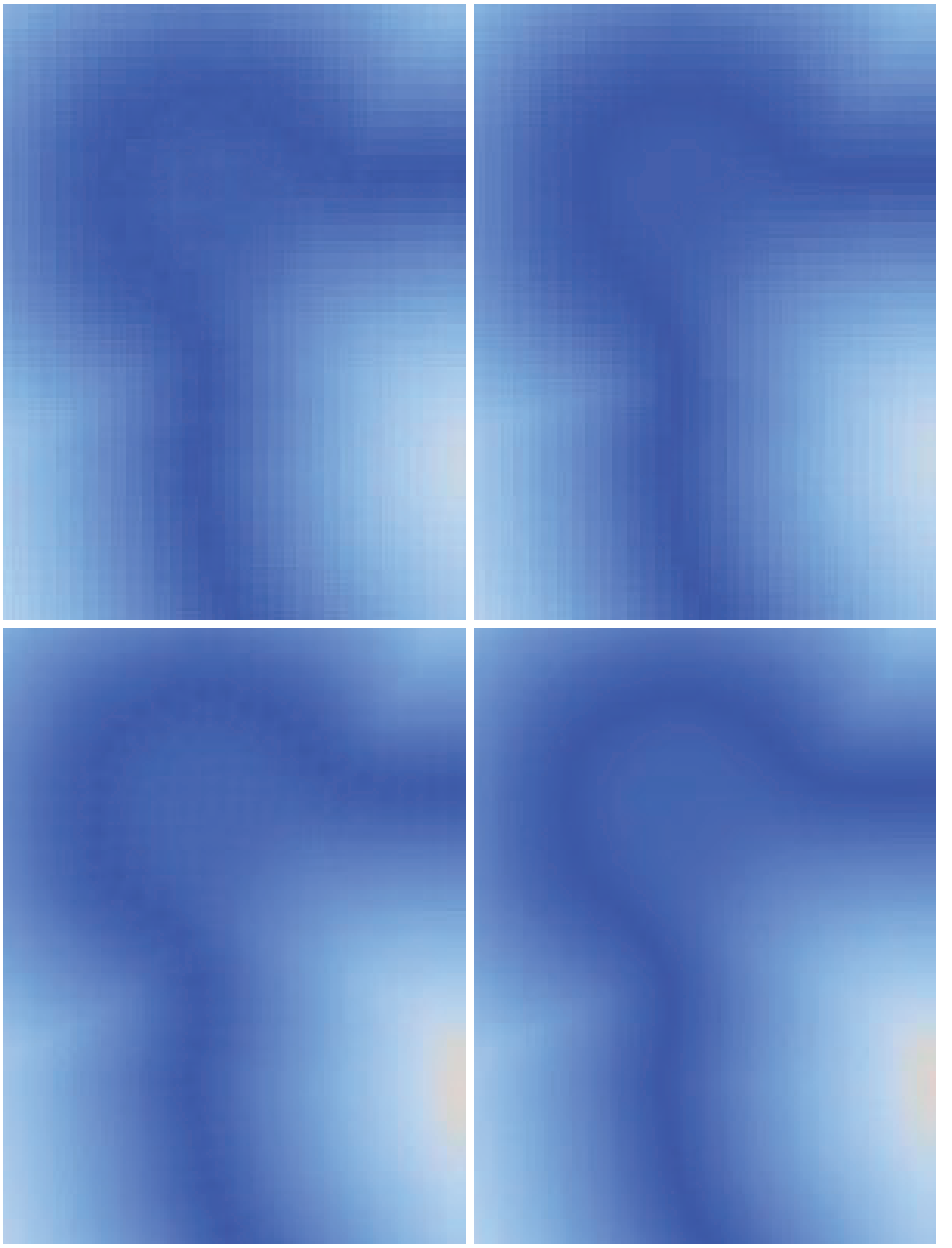

FIGURE 6. Comparing the results of distance function calculation from point cloud data (first column) and triangular surface (second column). Voxel edge size for results in the first row is 0.0125, in the second row is 0.003125. Results were calculated by FSM.

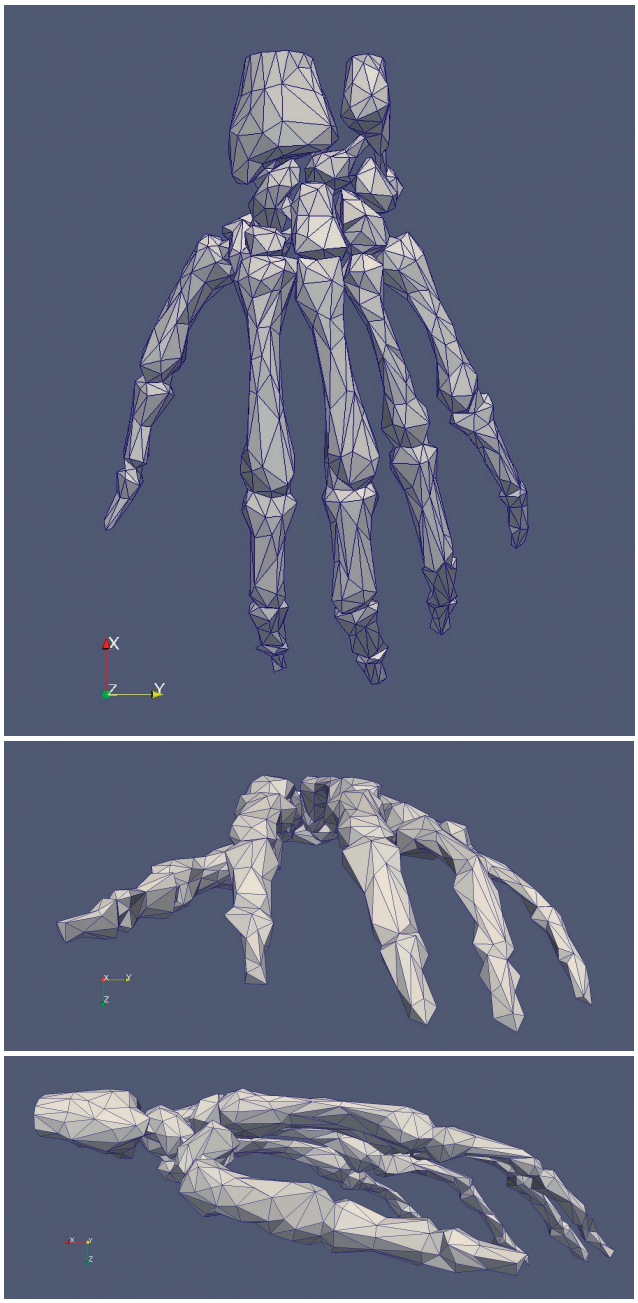

FIGURE 7. Hand Bones triangulated surface data. In the first picture, we can see it from above, in the second picture from the front, and the third picture from the side.

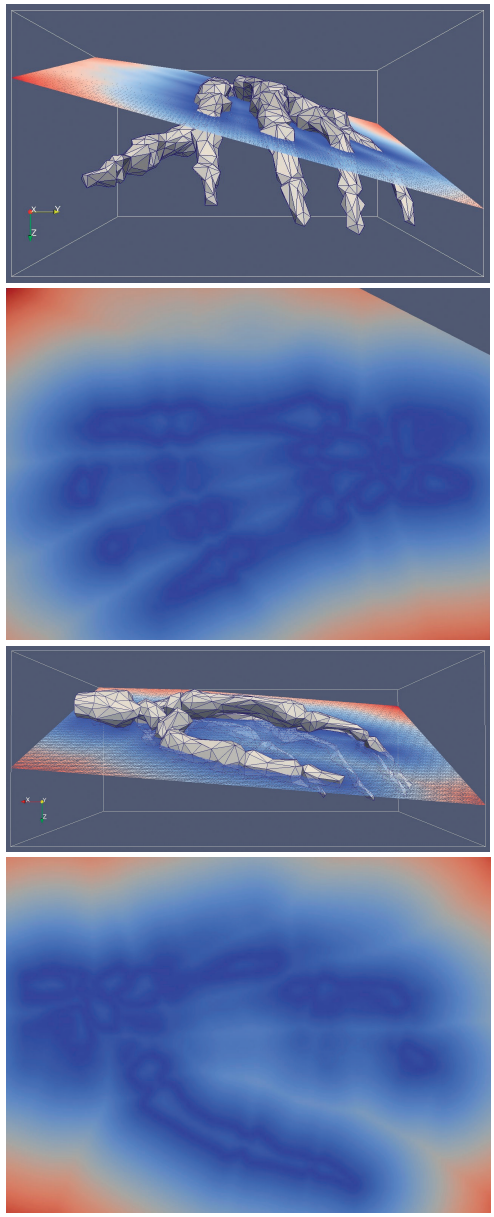

FIGURE 8. Visualization of slices of the distance function calculated to Hand Bones triangulated surface data. In the first and third picture, we can see the location of slices in the 3D computational domain, in the second and fourth picture the slices.

4. Numerical methods for computing the middle surface

While analysing the algorithms for distance function calculation, we realized that methods tracking the source of the distance, such as VDT and DP, can be straightforwardly modified for the search of middle surfaces between data sets. In fact, the main inspiration for us was DP method where we expected such modification should work. We propose how to adjust all previously described algorithms to find the middle surface for more data sets of various kinds. Our approach is based on information propagation through which we track the source of the information.

In the pseudo-codes Alg. 1 and Alg. 6 we showed how to initialize the distance function from one data set. When we have more data sets, we apply one of the algorithms for them separately on the same computational grid. A change which needs to be applied is that in the array $s_{i,j,k}$ for the source of $d_{i,j,k}$ calculation, we need to track also the information to which data set this source belongs to.

The change in the VDT and DP algorithms for our new purpose is very simple, because they already include source tracking. Again, what we need to change is to track also a label of the data set from which the information propagates. The modification of the FSM and FMM algorithms is not trivial. These methods originally do not contain any information about sources, thus we need to include it in a proper manner.

We display the modification of the FSM algorithm in the pseudo-code Alg. 7. In every iteration of the algorithm when we cycle through the grid points, we take the distance value from the neighbouring points to solve a quadratic equation. We need to keep track, from which neighbours the distance values enter the quadratic equation, thus we save the indexes (r, s, t), $r, s, t \in \{-1, 0, 1\}$, which identify them. When the solution is calculated for the equation we add up the indexes (r, s, t), see line 20 of Alg. 7, and this will show us which source to save for the current grid point from the sources of its 26 neighbours.

For the FMM algorithm, the modification is shown in the pseudo-code Alg. 8. In this modification after a nonfixed voxel is tagged as 'visited' (the visiting value is set to 2), we analyse all its neighbours, see line 26 of the pseudo-code. With the neighbours that also were 'visited', we calculate the current voxel's possible distance from the neighbours' sources, which for quick calculation will be determined by the neighbours' distance value plus the distance between the voxel and its neighbour. The source for which the calculated value is the smallest will be chosen as the source for the current voxel.

By using the modified algorithms, the grid points in the computational domain will be divided into subvolumes "belonging" to the different data sets by source information propagation. To obtain the middle surface between the data sets we just need to find the borders between these subvolumes.

To that goal, we use two methods. For any number of data sets, we can cycle through all points of the computational domain and find every point which has a neighbour belonging to a different subvolume. If we apply this for every data set separately, for each of them we obtain a set of points which are at a discrete border of the subvolume belonging to it. If we have just two data sets, we can treat the obtained information about which data set the grid points belong to, as a function of values 0 or 1, and visualize the isosurface of the function with the value 0.5. We demonstrate the two approaches of visualizing the results in the next subsection with the first numerical experiment for finding the middle surface. In Figure 11 in the second picture of the right column we see the representation of the middle surface as a discrete border of subvolumes belonging to a data set, and in the third picture of the right column as an isosurface of a function.

4.0.1. Experiment 1: Sponge & Sphere

Let us have two 3D point clouds, presented in Figure 9. The first is the "Sponge" point cloud data created by the parametric equations

$$\begin{aligned} x &= s_x + \left(0.207 + 2.003 \cdot \overset{2}{\sin}(\varphi) - 1.123 \cdot \overset{4}{\sin}(\varphi)\right) \cdot \cos(\varphi) \cdot \sin(\theta), \\ y &= s_y + \cos(\varphi) \cdot \sin(\theta), \\ z &= s_z + \sin(\varphi), \\ \varphi &\in \langle 0, 2\pi \rangle, \theta \in \langle 0, \pi \rangle. \end{aligned} \tag{11}$$

The second point cloud data is a sphere with a radius of 0.5. The distance between the centres of the two objects is 2.0. To create the point cloud data we used a step of $\frac{\pi}{10}$ for both angles in the parametric equations. We calculated the distance function on a grid voxel edge size 0.025. For the middle surface calculated by the VDT algorithm, we obtained the result seen in Figure 10 visualized as an isosurface. The FMM and DP algorithms yield a similar result.

With the application of the modified FSM algorithm for this experiment, we discovered that it can cause some issues in specific situations. When we initialize the distance function according to Alg. 1 on a grid with density higher than the point cloud density, we get an initial value that consists of separated subvolumes around the points. The problem is that these gaps in the initialized distance function do not contain any source information, and if we apply FSM, such lack of information can propagate through the computational grid. We can see that in the second picture of the left column in Figure 11. The orange dots indicate the grid points with no source information. This leads to errors when we are trying to detect grid points on the discrete borders of subvolumes belonging to the different point cloud data sets or when we want to visualize the middle

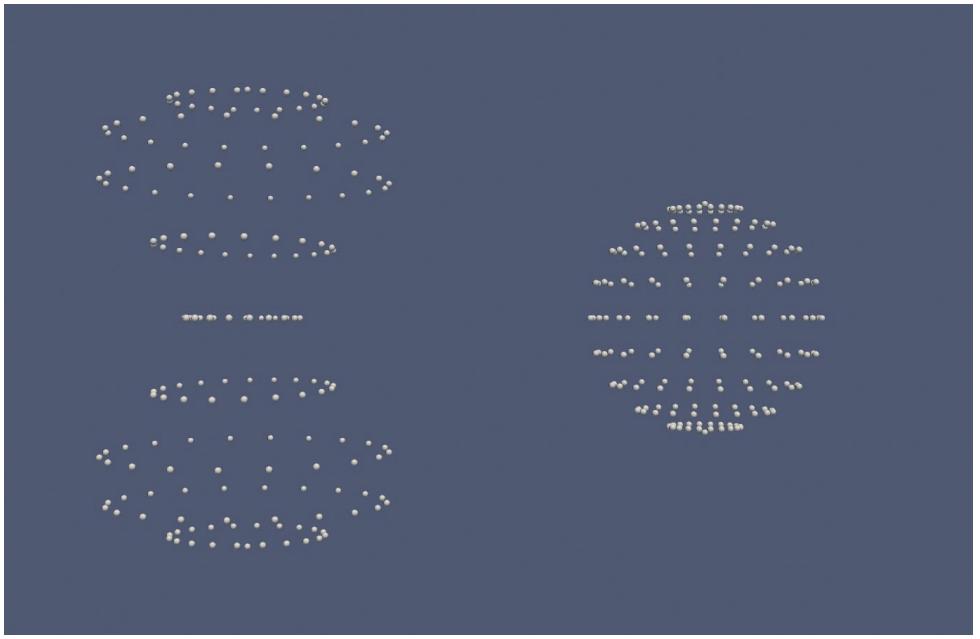

FIGURE 9. Experiment 1: Generated point cloud data of Sponge and Sphere.

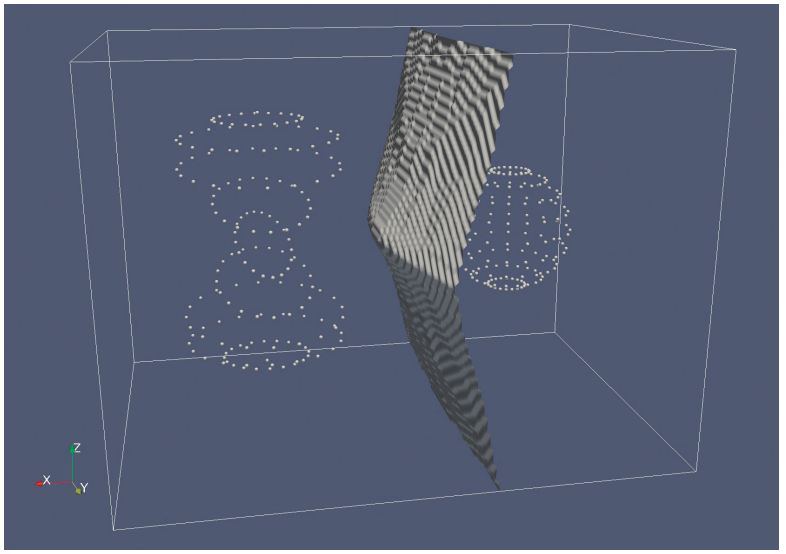

FIGURE 10. Experiment 1: Middle surface between Sponge and Sphere point cloud data calculated by the VDT algorithm.

Algorithm 7 Modified fast sweeping method including sources

Input: From correct initialization: 3D grid, $d_{i,j,k}$, $f_{i,j,k}$, $s_{i,j,k}$

for $(l = 0; l < 8; l = l + 1)$ **do**
  for $(i = i_{\text{sweep}}[l,0]; i \leq i_{\text{sweep}}[l,1]; i = i + i_{\text{sweep}}[l,2])$ **do**
    for $(j = j_{\text{sweep}}[l,0]; j \leq j_{\text{sweep}}[l,1]; j = j + j_{\text{sweep}}[l,2])$ **do**
      for $(k = k_{\text{sweep}}[l,0]; k \leq k_{\text{sweep}}[l,1]; k = k + k_{\text{sweep}}[l,2])$ **do**
        if $f_{i,j,k}$ is not true **then**
          The indexes (r,s,t), $r,s,t \in \{-1,0,1\}$, indicate from which neighbour the distance value comes from.
          $\left[a_1,(r,s,t)_{a_1}\right] = \min_d\left(\left[d_{i+1,j,k},(1,0,0)\right],\left[d_{i-1,j,k},(-1,0,0)\right]\right)$
          $\left[a_2,(r,s,t)_{a_2}\right] = \min_d\left(\left[d_{i,j+1,k},(0,1,0)\right],\left[d_{i,j-1,k},(0,-1,0)\right]\right)$
          $\left[a_3,(r,s,t)_{a_3}\right] = \min_d\left(\left[d_{i,j,k+1},(0,0,1)\right],\left[d_{i,j,k-1},(0,0,-1)\right]\right)$
          ▷ Use $+\infty$ if (i,j,k) is out of bounds.
          Sort $\{[a_1,(r,s,t)],[a_2,(r,s,t)],[a_3,(r,s,t)]\}$ from lowest to highest according to values $\{a_1,a_2,a_3\}$.
          $\left[d_{\text{new}},(r,s,t)_{d_{\text{new}}}\right] = \left[a_1,(r,s,t)_{a_1}\right] + [h,(0,0,0)]$
          if $d_{\text{new}} > a_2$ **then**
            $d_{\text{new}} = \underset{x}{MaxSolution}\left((x-a_1)^2 + (x-a_2)^2 = h^2\right)$
            $(r,s,t)_{d_{\text{new}}} = (0,0,0) + (r,s,t)_{a_1} + (r,s,t)_{a_2}$
            if $d_{\text{new}} > a_3$ **then**
              $d_{\text{new}} = \underset{x}{MaxSolution}\left((x-a_1)^2 + (x-a_2)^2 + (x-a_3)^2 = h^2\right)$
              $(r,s,t)_{d_{\text{new}}} = (0,0,0) + (r,s,t)_{a_1} + (r,s,t)_{a_2} + (r,s,t)_{a_3}$
            end if
          end if
          if $d_{\text{new}} < d_{i,j,k}$ **then** $\left\{d_{i,j,k} = d_{\text{new}};\ s_{i,j,k} = s_{(i,j,k)+(r,s,t)_{d_{\text{new}}}}\right\}$
        end if
      end for
    end for
  end for
end for

surface as an isosurface of a function. The isosurface with errors can be seen in the third picture of the left column in Figure 11.

To solve this problem, we need to modify also the initialization of the distance function to point cloud data for the FSM algorithm. The idea is to get a contiguous subvolume for the initialized grid points. For this, we need to increase the volume around the single points in which we initially calculate the distance function. We need to find the minimum size of this volume so that for two neighbouring cloud points the volumes will intersect. We found that for this minimum size we can use the maximum of all minimal distances between two cloud points. With its value, we build a cube around every cloud point which determines the volume in which we will calculate the exact distance values. We can see the result of this modification in the right column of Figure 11.

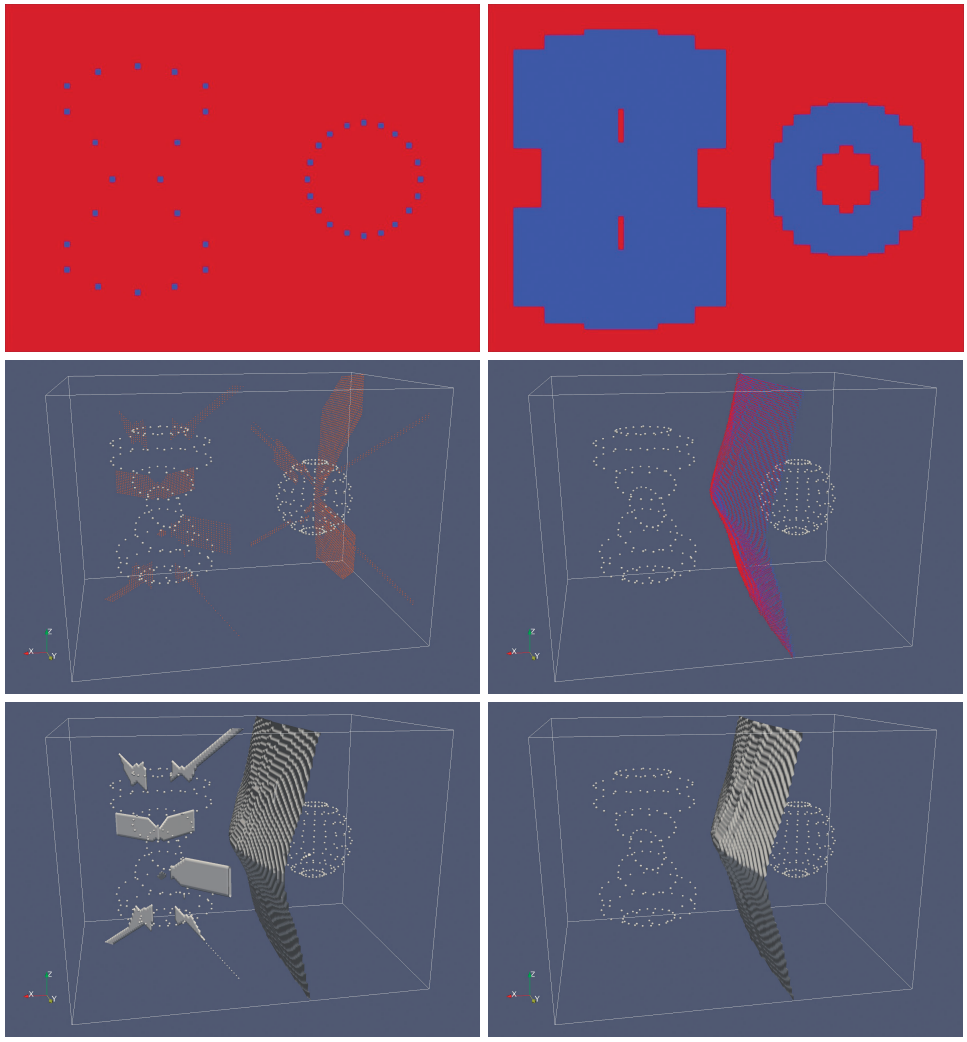

FIGURE 11. Experiment 1: Finding the middle surface between Sponge and Sphere point cloud data by the FSM algorithm. In the first picture of the left column, we see the section of the original initial condition in a constant y plane. In the second picture of the left column, the grid points with no source are visualized. In the third picture of the left column, the incorrect isosurface between subvolumes of the computational grid is visualized. In the first picture of the right column, we can see the corrected initial condition. In the second picture of the right column, the discrete borders of the subvolumes obtained by the corrected calculation are visualized. Red points "belong" to Sponge point cloud data, blue points "belong" to Sphere point cloud data. In the third picture of the right column, the correct isosurface between subvolumes of the computational grid is visualized.

Algorithm 8 Modified fast marching method including sources

Input: From Alg. 1: 3D grid, $d_{i,j,k}$, $f_{i,j,k}$, $s_{i,j,k}$
Declaration: $v_{i,j,k}$ will hold the visiting values of grid points
'unvisited'=0, 'to be visited'=1, 'visited'=2
Declaration: *heap* container will be a *min-priority-heap*
Initialization: $\forall f_{i,j,k} = \text{true} : \{v_{i,j,k} = 1;\ heap.InsertNode(d_{i,j,k})\}$ else: $v_{i,j,k} = 0$
1: **while** *heap* is not empty **do**
2: $(i, j, k) = heap.GetRoot()$ ▷ Obtain (i, j, k) with minimum d and delete from *heap*.
3: **for all** $\{(i + r, j + s, k + t) ; (r, s, t) \in P^1\}$, not out of bound **do**
.
.
. ▷ The pseudo-code is the same as in Alg. 4.
.
22: **end for**
23: $v_{i,j,k} = 2$
24: **if** $f_{i,j,k}$ is not true **then**
25: $[d_{\min}, (u, v, w)] = [\infty, (0, 0, 0)]$
26: **for all** $\{(i + r, j + s, k + t) ; (r, s, t) \in P^2\}$, not out of bound **do**
27: **if** $v_{i+r,j+s,k+t} = 2$ **then**
28: **if** $|r| + |s| + |t| = 1$ **then** $d_{test} = d_{i+r,j+s,k+t} + h$
29: **if** $|r| + |s| + |t| = 2$ **then** $d_{test} = d_{i+r,j+s,k+t} + \sqrt{2} * h$
30: **if** $|r| + |s| + |t| = 3$ **then** $d_{test} = d_{i+r,j+s,k+t} + \sqrt{3} * h$
31: **if** $d_{\min} > d_{test}$ **then**
32: $[d_{\min}, (u, v, w)] = [d_{test}, (i + r, j + s, k + t)]$
33: **end if**
34: **end if**
35: **end for**
36: $s_{i,j,k} = s_{(u,v,w)}$
37: **end if**
38: **end while**

In the first picture, visualizing a section of the new initial condition, we can see that now we have a contiguous subvolume of grid points. In the second picture, we can see that the discrete border of the subvolumes belonging to a data set can be detected correctly, and in the third picture that the isosurface is obtained without any error.

To have a sense of the overall CPU times required for finding the middle surface with the mentioned algorithms we conducted some measurements on this experiment and present these in Table 7 for FSM, Table 8 for VDT, Table 9 for FMM and Table 10 for DP. As in the previous tables, the parameters of our grid are listed in the first columns. After that we see the number of fixed points for the initialization of our grid and how long it took to perform it. These values are higher for the FSM algorithm as we needed to get the contiguous subvolumes. For the other three algorithms these numbers are the same. In each table the fifth column shows CPU times required to calculate just the

distance function, and the sixth column CPU times for middle surface calculation. This way, we can compare how tracing the sources and the objects for every grid point impacted the speed of the algorithms. The modification of algorithms VDT and DP was minimal, thus CPU times increased only about 10–20 %, while for FSM and FMM, with more complex modifications, the CPU times increased about 30–40 %.

In the following experiments, we will show various cases of how we can apply the described algorithms and discuss possible differences in the results of the methods.

TABLE 7. CPU time comparison for original and modified FSM algorithms applied to Experiment 1.2.

Number of grid points	Voxel edge size	Fixed points	Initial condition	Original FSM	Modified FSM
43 x 32 x 32	0.1	12357	0.003	0.007	0.011
85 x 62 x 62	0.05	86306	0.018	0.049	0.07
168 x 123 x 123	0.025	628799	0.13	0.371	0.525
334 x 245 x 245	0.0125	4781922	1.003	2.9	4.189
667 x 488 x 488	0.00625	38269939	8.071	23.829	33.078

TABLE 8. CPU time comparison for original and modified VDT algorithms applied to Experiment 1.

Number of grid points	Voxel edge size	Fixed points	Initial condition	Original VDT	Modified VDT
43 x 32 x 32	0.1	1890	0.001	0.028	0.02
85 x 62 x 62	0.05	2716	0.014	0.143	0.145
168 x 123 x 123	0.025	2892	0.09	0.922	1.126
334 x 245 x 245	0.0125	2912	0.702	7.563	8.942
667 x 488 x 488	0.00625	2912	5.62	59.763	72.474

4.0.2. Experiment 2: Subsets of the Cube

We return to the Cube data set that has coinciding points with the computational grid. We will use a computational grid with voxel edge size 0.05. The points on every subset of the Cube (vertex, edge, wall) will be treated as a separate data set. In Figure 12 we visualize with colors how the points are distributed into sets of sources. We can see that the vertices are treated as one-point data sets, the edges do not contain the vertices and the walls do

TABLE 9. CPU time comparison for original and modified FMM algorithms applied to Experiment 1.

Number of grid points	Voxel edge size	Fixed points	Initial condition	Original FMM	Modified FMM
43 x 32 x 32	0.1	1890	0.001	0.015	0.023
85 x 62 x 62	0.05	2716	0.014	0.144	0.208
168 x 123 x 123	0.025	2892	0.09	1.707	2.696
334 x 245 x 245	0.0125	2912	0.702	24.61	36.103
667 x 488 x 488	0.00625	2912	5.62	332.377	430.497

TABLE 10. CPU time comparison for original and modified DP algorithms applied to Experiment 1.

Number of grid points	Voxel edge size	Fixed points	Initial condition	Original DP	Modified DP
43 x 32 x 32	0.1	1890	0.001	0.012	0.014
85 x 62 x 62	0.05	2716	0.014	0.126	0.133
168 x 123 x 123	0.025	2892	0.09	1.516	1.619
334 x 245 x 245	0.0125	2912	0.702	21.333	23.499
667 x 488 x 488	0.00625	2912	5.62	274.257	290.603

not contain either the edges or the vertices. Now in this setup, we apply the algorithms for computing the middle surface.

First, we analyse the results for VDT . In Figure 13 we can see how the grid points are assigned to the different subsets. For clearer visualization, we show just some of the separate volumes with the outlines of the Cube by white lines. We can identify by color to which subset of the Cube the points belong to. Let us notice that to the interior of the Cube only the information from the walls propagates. From the vertices and edges, the information only propagates outwards. For this reason, the discrete borders of the subvolumes inside of the Cube are not "uniform". We can see it more clearly in Figure 14 where we visualize only the borders of the subvolumes. For FMM and DP we obtain similar results.

Let us compare the previous result to the results of the FSM algorithm visualized in Figure 15. We can identify by the colors that the information propagates inward from all subsets of the Cube. Inside of the Cube, the grid points belonging to vertices are along a line, for edges, the grid points are confined to a triangle, and for the walls, they are inside a pyramid. In these results, the borders of the separated volumes are much clearer and sharper which we can identify easier in Figure 16.

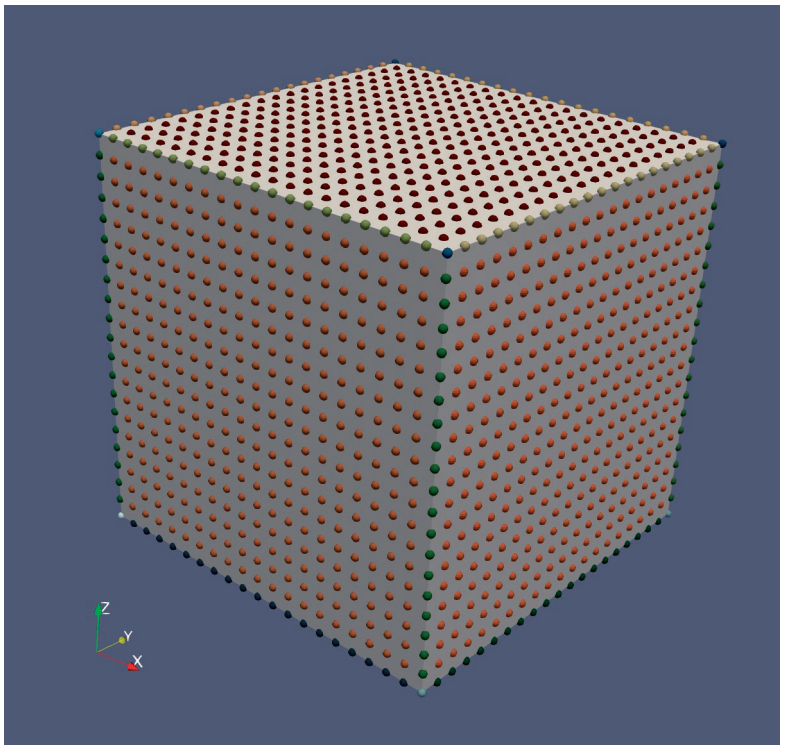

FIGURE 12. Experiment 2: Visualization of source labels on the Cube data set.

4.0.3. Experiment 3: Cube & Sphere

In the next experiment, we consider a cube with the same parameters but now we will work with it as a triangulated surface. As we demonstrated in Section 3.2, we can use the algorithms for distance function calculations on triangulated surfaces as well if we use the Alg. 6 for the initialization. This type of initialization produces contiguous subvolumes of grid points thus it does not need any changes to be applicable for the Modified FSM algorithm as it was in the case of point cloud data. Inside of the Cube, we have the Sphere with radius 0.25 and center point the same as the center of the Cube. We can see their relative location in the first picture of Figure 17. In the second picture, we see the computed middle surface with the objects. In the next pictures of this figure, we can see the results for the VDT, DP, FSM, and FMM algorithms, in this order from left-up to right--down. In the detailed view of the results, we can see the fine differences between them.

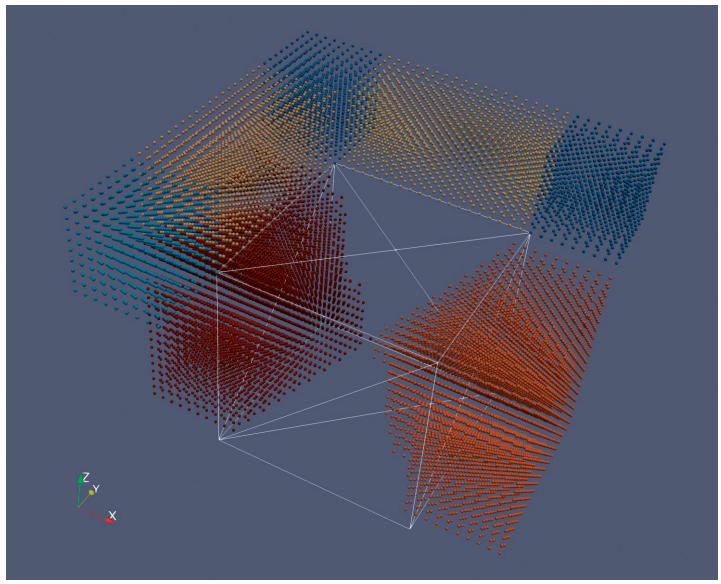

FIGURE 13. Experiment 2: Visualization of source tracking result on the Cube data set for the VDT method.

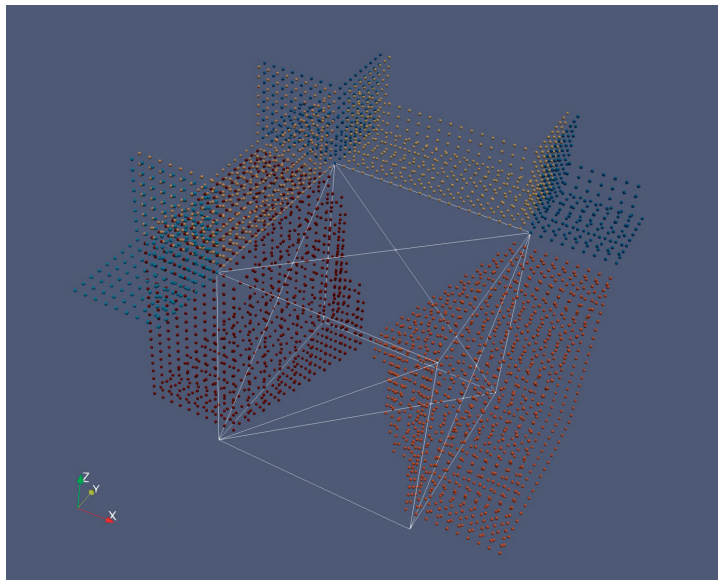

FIGURE 14. Experiment 2: Visualization of discrete borders of subvolumes belonging to different sources on the Cube data set for the VDT method.

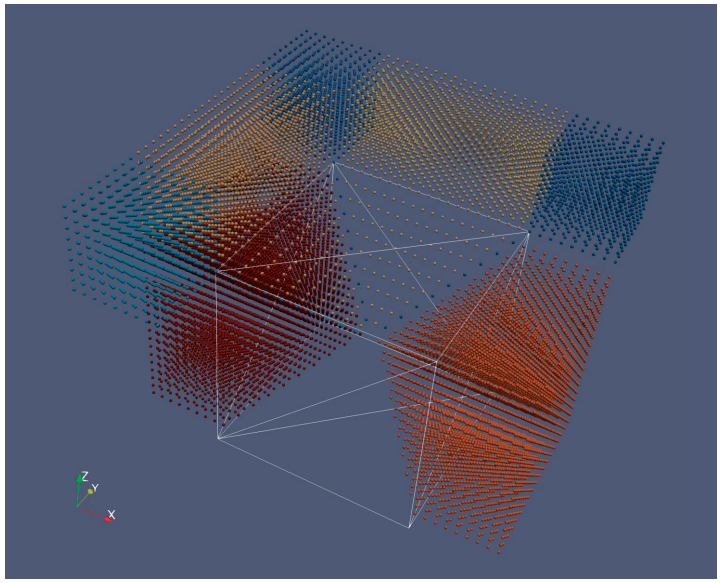

FIGURE 15. Experiment 2: Visualization of source tracking result on the Cube data set for the FSM algorithm.

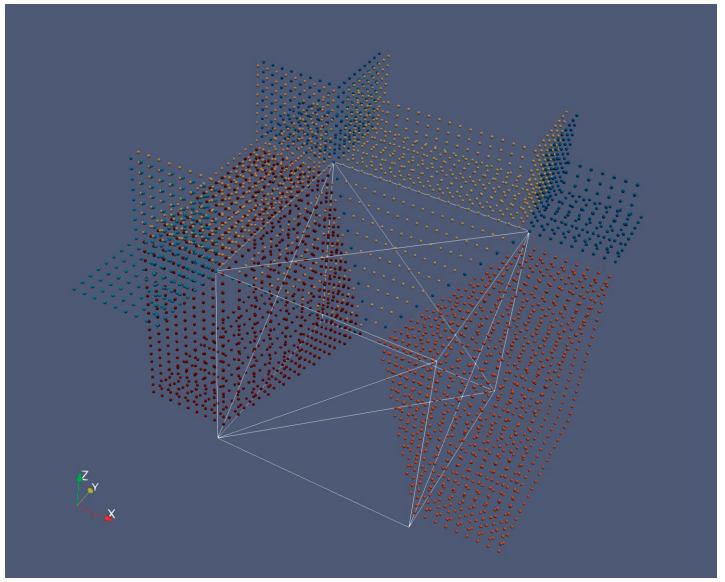

FIGURE 16. Experiment 2: Visualization of discrete borders of subvolumes belonging to different sources on the Cube data set for the FSM algorithm.

For a quantitative comparison of the methods, we calculate the volume and area of the isosurfaces computed on computational grids with different voxel edge sizes, equal to

$$0.2, 0.1, 0.05, 0.025, 0.0125, 0.00625, 0.003125.$$

We list these results in Table 11. By comparing the values in this table and looking at the pictures of the middle surface we can see that the results from the pairs VDT, DP, and FSM, FMM are very similar.

TABLE 11. Experiment 3: Comparing volume and area for middle surface between the Cube and the Sphere data sets.

Number of	Voxel	VDT		DP		FSM		FMM	
grid points	edge size	Volume	Area	Volume	Area	Volume	Area	Volume	Area
10^3	0.2	0.418667	2.92008	0.418667	2.92008	0.418667	2.92008	0.418667	2.92008
19^3	0.1	0.3005	2.39785	0.2855	2.34128	0.244167	1.98998	0.2645	2.21841
37^3	0.05	0.291396	2.40964	0.291396	2.40964	0.271396	2.29456	0.273396	2.31799
73^3	0.025	0.286294	2.39177	0.286326	2.39452	0.277992	2.34297	0.274508	2.34427
145^3	0.0125	0.287682	2.40421	0.287686	2.40489	0.282912	2.39768	0.277739	2.37683
289^3	0.00625	0.287828	2.42233	0.287828	2.42245	0.284826	2.40956	0.281192	2.40664
577^3	0.003125	0.287996	2.42093	0.287995	2.42094	0.286224	2.42284	0.283977	2.41481

4.0.4. Experiment 4: Five Ellipsoids

The following experiment is done with five different Ellipsoids point cloud data sets, for which the center points all lie on the plane $z = 0$. In Figure 18 we visualize the results of the algorithms in the plane $z = 0$ as the discrete border of subvolumes together with the distance function and the original data. Here we show the results of the FSM algorithm with red lines, of VDT with dark blue lines. Because of the overlapping of the results for DP and FMM are almost not visible. We can just see the result for DP with a green line in the upper left corner. In this experiment, we can see that with our algorithms the obtained results are a good approximation of the Voronoi diagram.

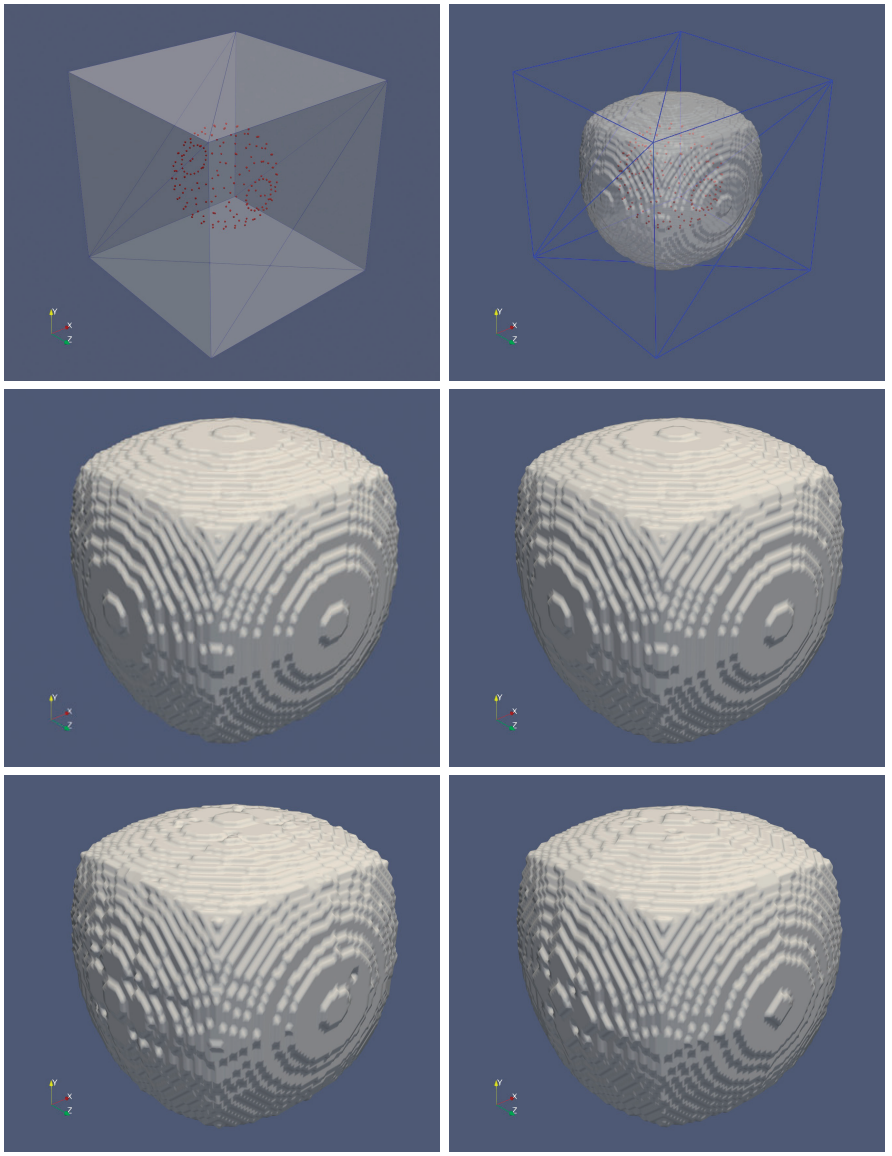

FIGURE 17. Experiment 3: Finding the middle surface between Sphere point cloud data inside a Cube triangulated surface. In the first picture, we see the two objects. In the next picture, we visualize the middle surface together with the objects. In the following pictures, we show the resulting isosurfaces for every algorithm in more detail. They are visualized from left-up to down-right in the following order: VDT, DP, FSM, FMM. The visualized results were computed on a grid with 181^3 elements and a voxel edge size of 0.01.

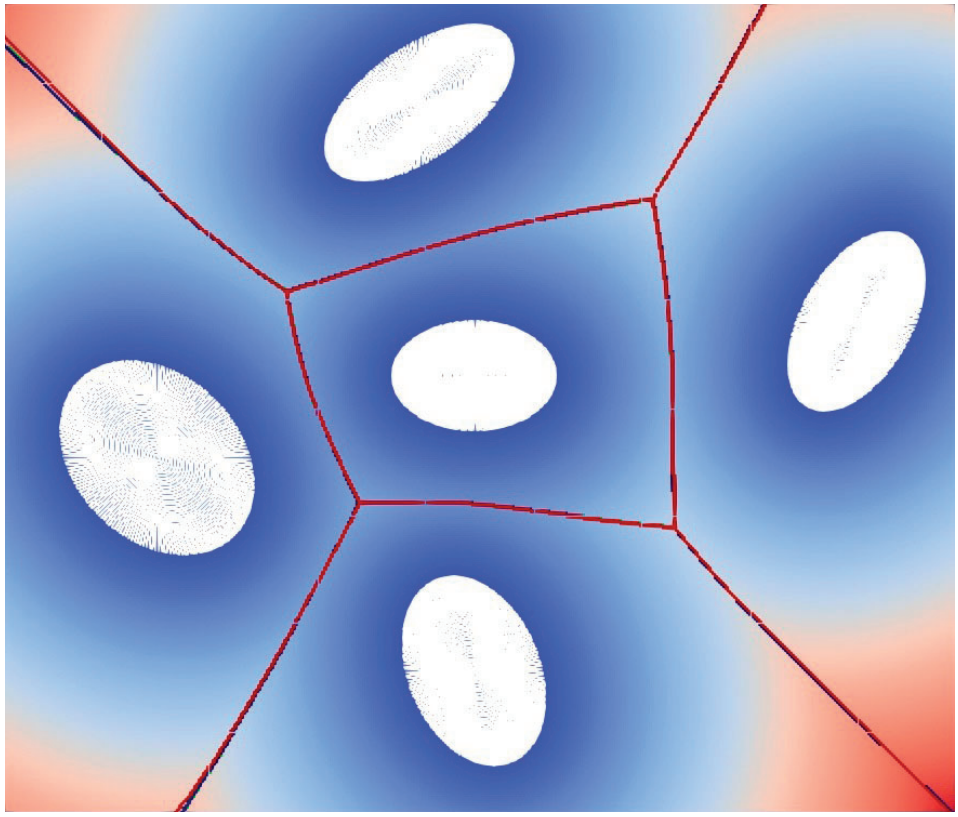

FIGURE 18. Experiment 4: Finding the border between five Ellipsoid point cloud data. In the picture, the border points between divided volumes are visualized in the plane $z = 0$ together with the distance function and the data sets. We show the results for FSM algorithm with red lines, for VDT with dark blue lines.

4.0.5. Experiment 5: Two parallel surfaces

For the last experiment, we want to show how accurately the algorithms can find the middle surface between two parallel data sets. For this purpose, we will use wave-like surfaces generated as point cloud data by functions

$$\begin{aligned} f(x,y) &= 0.2 * \cos(x * y) + 0.5, \\ f(x,y) &= 0.2 * \cos(x * y) - 0.5, \\ (x,y) &\in < -5.0, 5.0 > \times < -5.0, 5.0 > \end{aligned} \tag{12}$$

with a step of 0.05 for both x and y variables.

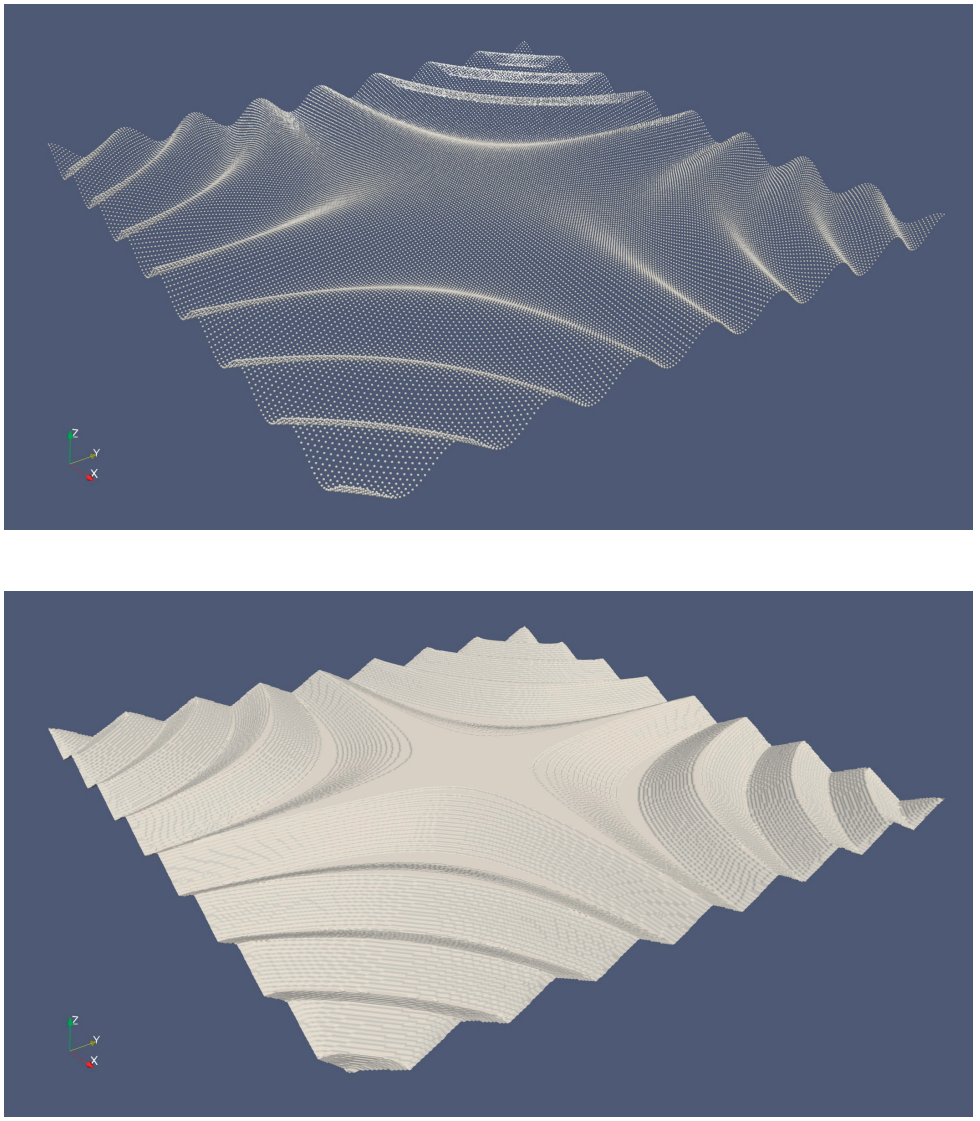

FIGURE 19. Experiment 5: Finding the middle surface between two parallel wave-like point cloud data sets generated by equations (4.0.5). In the first picture, we see the visualization of one point cloud. The other one is identical just shifted along the z axis. In the second picture, we visualize the middle surface which divides the computational domain between the two point cloud data sets.

We can see the visualization of the point cloud data generated by the first equation of (4.0.5) in the first picture of Figure 19. In the second picture, we can see the result of the calculations by the FSM algorithm on a computational grid with voxel edge size 0.025 represented as an isosurface. This isosurface lies between the two parallel point cloud data sets. Visually the results for the four methods do not show noticeable differences, thus we show only the results of FSM .

Acknowledgement.
We would like to thank Prof. Zuzana Krivá for pointing out the possibility to use 6 voxel neighbours instead of 26 in the Dijkstra-Pythagoras method.

REFERENCES

[1] CHEN, X.—GOLOVINSKIY, A.—FUNKHOUSER, T.: *A benchmark for 3D mesh segmentation*, ACM Transactions on Graphics **28** (2009). pp. 1–12. https://doi.org/10.1145/1531326.1531379

[2] CORMEN, T. H.—LEISERSON, C. E.—RIVEST, R. L.—STEIN, C.: *Introduction to Algorithms* Third Edition. MIT Press, Cambridge, MA, 2009.

[3] DANIELSSON, P.-E.: *Euclidean distance mapping*, Computer Graphics and Image Processing **14** (1980), 227–248.

[4] EBERLY, D.: *Distance between point and triangle in 3D*, Geometric Tools, Redmond WA 98052, (1999). https://www.geometrictools.com/Documentation/DistancePoint3Triangle3.pdf

[5] JONES, M. W.—BAERENTZEN, J. A.—SRAMEK, M.: *3D distance fields: a survey of techniques and applications*, IEEE Transactions on Visualization and Computer Graphics **12** (2006), no. 4, 581–599.

[6] KIMMEL, R.—SHAKED, D.—KIRYATI, N.—BRUCKSTEIN, A. M.: *Skeletonization via distance maps and level sets*, Computer Vision and Image Understanding **62** (1995), 382–391.

[7] PERSSON, P.-O.: *Mesh Generation for Implicit Geometries*. PhD Thesis, Department of Mathematics, Massachusetts Institute Of Technology, 2005.

[8] ROUY, E.—TOURIN, A.: *A viscosity solutions approach to shape-from-shading*, SIAM Journal on Numerical Analysis **29** (1992), 867–884.

[9] RUMPF, M.—TELEA, A.: *A continuous skeletonization method based on level sets*, Proceedings of the Symposium on Data Visualisation **2002** (2002), 151–ff.

[10] SETHIAN, J. A.: *A fast marching level set method for monotonically advancing fronts*, Proc. Nat. Acad. Sci. U.S.A. **93** (1996), no. 4, 1591–1595.

[11] SIDDIQI, K.—BOUIX, S.—TANNENBAUM, A.—ZUCKER, S.: *The hamilton-jacobi skeleton*. In: Proc. of the Seventh IEEE International Conference on Computer Vision (ICCV)(September 1999) Vol. 2 (1999), pp. 828-834.

[12] SMÍŠEK, M.: *Analysis of 3D and 4D Images of Organisms in Embryogenesis*. PhD Thesis, Faculty of Civil Engineering, Slovak University of Technology Bratislava, 2015.

[13] TURK, G.—MULLINS, B.: *Large Geometric Models Archive*. Georgia Institute of Technology, 1999.

[14] ZHAO, H.: *A fast sweeping method for Eikonal equations*, Math. Comput. **74** (2005), 603–627.

Received November 13, 2022
Revised March 23, 2023
Accepted October 9, 2023
Publ. online December 20, 2023

Department of Mathematics and Descriptive Geometry
Faculty of Civil Engineering
Slovak University of Technology in Bratislava
Radlinského 11
810 05 Bratislava
SLOVAKIA

E-mail: kosabalu@gmail.com
mikula@math.sk