EDISON

Electromagnetic Design of
flexIble SensOrs

Localization of 3D point in tetrahedra mesh

Adam Dziekoński
April 4, 2018

The „EDISOn - Electromagnetic Design of flexIble SensOrs" project, agreement no TEAM TECH/2016-1/6, is carried out within the TEAM-TECH programme of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund.

Revision	Date	Author(s)	Description
1.0	22.03.2018	Adam Dziekoński	created
1.1	29.03.2018	Adam Dziekoński	created
1.2	30.03.2018	Adam Dziekoński	created
1.3	04.04.2018	Adam Dziekoński	created

Contents

1 Introduction **1**

2 Netgen **1**

3 Walking algorithms. Basic visibility and stochastic walk algorithms. **2**

4 Strategies how to improve the visibility and stochastic walk algorithms **7**
4.1 Strategy 1 - go to the first neighbour on the list 7
4.2 Strategy 2 - go to the neighbour with minimum value of face orientation test 7
4.3 Strategy 3 - go to the random neighbour 8

5 Determine the set of destination points (tetrahedra) **14**
5.1 List of destination points (tetrahedra) to be found 14
5.2 Coarse and fine mesh 14
5.3 Range of nodes 17
5.4 Destination point on a face of the tetrahedra 17

6 Jump and Walk **19**

7 Conclusions **27**

1 Introduction

In this report algorithms, implementations and performance of localization of 3D point in tetrahedral mesh are discussed.

Firstly, it is presented how the mesh is generated and read in Matlab (Section 2). Then, features of visibility and stochastic walk algorithms reported in [2, 3] and their implementations are described in Section 3. Strategies how to improve the visibility and stochastic walk algorithms are described and verified in Section 4. After selecting the most effective walking algorithm implemented in this report several examples are discussed in which it is presented how to find multiple destination points in mesh which can lie inside a tetrahedron or lie on a face of tetrahedron or be a node of a mesh (Section 5).

In test described in Sections 3-5 starting tetrahedra (in most cases) were selected in such a manner that finding a destination point by a walk algorithm is not trivial. The influence of the selection starting tetrahedron is crucial, thus in Section 6 an initialization phase (a *jump* to starting tetrahedron in a walk reported in [4]) is discussed.

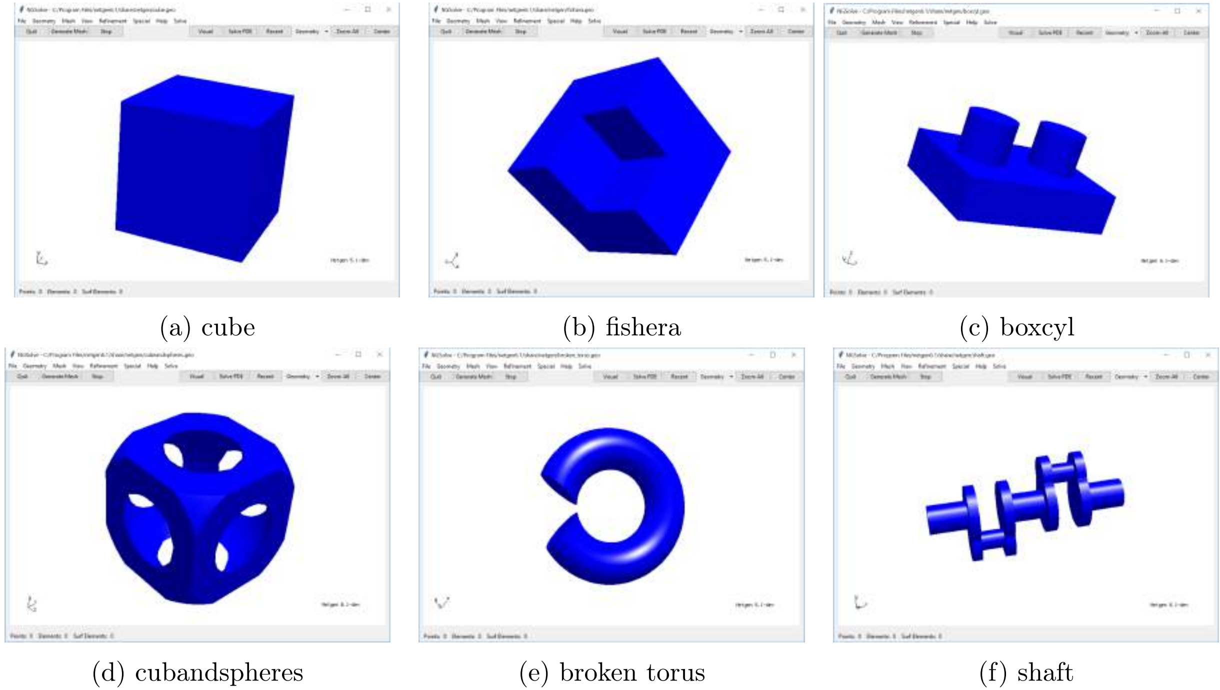
All computations in this report are performed in Matlab 2017a, operating system Windows 10, processor: Intel Xeon X5680 (i7) 3.33GHz.

2 Netgen

In this report *Experimental Netgen/NGSolve 6.1 featuring Python 3* was used [1]. To verify the effectiveness of all implemented walking algorithms several structures have been considered (Fig. 1). The procedure of generating a mesh and loading it in Matlab is as follows. One needs to load a geometry file (*.geo) in Netgen and exports a mesh file (*.mesh). Then, a mesh file is read in Matlab

(with a proper usage of fscanf, sscanf and fgetl build-in functions) and two arrays required for walking algorithms are extracted from (*.mesh) file:

- $Nodes$ - K x 3 array with x, y , z coordinates of K nodes,
- $Teta$ - N x 5 array with medium property, and four indices of nodes that represent N tetrahedra.



(a) cube (b) fishera (c) boxcyl

(d) cubandspheres (e) broken torus (f) shaft

Figure 1: Structures tested in this report.

3 Walking algorithms. Basic visibility and stochastic walk algorithms.

In this section two algorithms of walking in tetrahedral mesh are described. Firstly, the visibility and stochastic walk algorithms (Algorithm 3-4) were implemented after studying [2, 3]. Both of these algorithms are based on the 3D orientation test Eq. 1. This test is performed for each face of the tetrahedron and returns the position of a point (**w**) against a face given by a tree vertices (**t**,**u**,**v**). The side of face **tuv** on which the point **w** lies is given by the sign of the determinant. In 3D if the sign of determinant is positive it means that the next visited tetrahedron should be through this face. In other words the next visited tetrahedron should be a neighbour of a current tetrahedron which has a common face for which orientation test equals to 1.

$$orientation3D(\mathbf{t}, \mathbf{u}, \mathbf{v}, \mathbf{w}) = sign(\begin{vmatrix} u_x - t_x & v_x - t_x & w_x - t_x \\ u_y - t_y & v_y - t_y & w_y - t_y \\ u_z - t_z & v_z - t_z & w_z - t_z \end{vmatrix}) \tag{1}$$

In our algorithms the following mapping of tetrahedron nodes list generated by Netgen was assumed:

$$Q = \begin{bmatrix} 123 \\ 134 \\ 142 \\ 243 \end{bmatrix} \tag{2}$$

Matrix Q is not accidental, since such a mapping guaranties that each face of a tetrahedron is counter clock-wise (CCW) oriented. For example if a tetrahedron is defined by a list of nodes: 99 784 945 942. The four faces CCW oriented due to the Q-matrix mapping are as follows:

- Face 1: 99 784 945
- Face 2: 99 945 942
- Face 3: 99 942 784
- Face 4: 784 942 945

The difference between the visibility and stochastic algorithms is the order in which faces are verified with the orientation test. The visibility algorithm performs the orientation test face by face in order in which the four faces of the tetrahedron are defined, so orientation tests are performed for faces 1,2,3,4 as long as a sign of Eq. 1 is not equal to 1. In a stochastic walk algorithm there is a random selection of the order in which faces are tested (Algorithm 4, step 12).

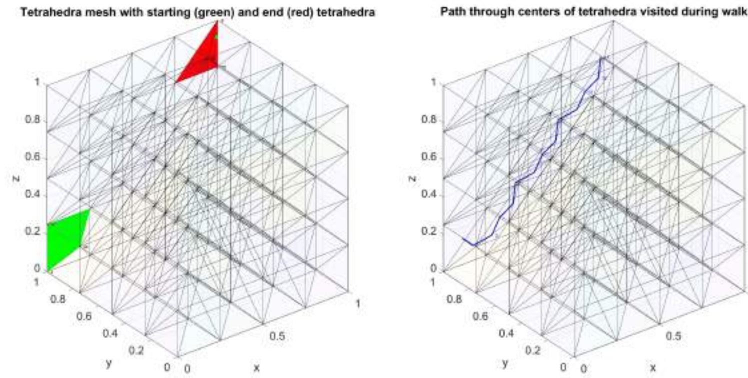
Both these algorithms performed well for simply connected structures such as a cube (cube.geo in Netgen, Fig. 1a). Figure 2 presents the path made by a visibility walk algorithm in mesh divided into 384 tetrahedra. For the same setup stochastic walk found the destination point (destination tetrahedron) in the same number of steps (14), however, three times selected different 'next' tetrahedra to visit. It is due to the random order of faces for which orientation tests have been made.

Both visibility and stochastic walks work properly for a structure named 'fichera' (fichera.geo from Netgen, Fig. 3). The mesh has 2368 tetrahedra. The starting and destination points were selected such as there is a straight line between starting and destination point, however, there are no tetrahedra on this line. In this case a destination point (tetrahedron) was found after 27 steps.

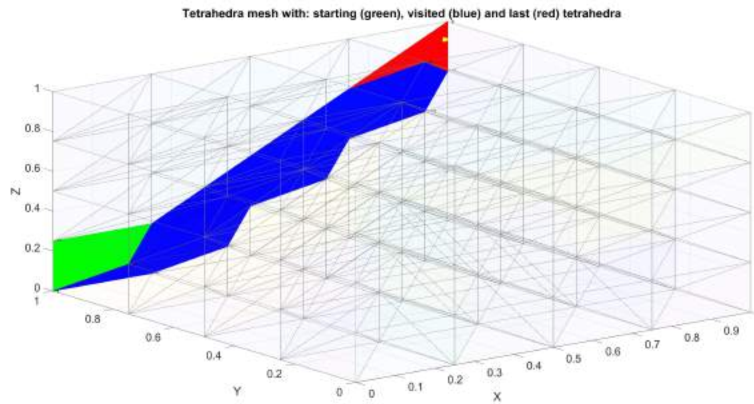
Table 2: List of tetrahedra visited in visibility walk (VW) and stochastic walk (SW).

Step	1	2	3	4	5	6	7	8	9	10	11	12	13	14
VW	1	344	349	348	346	43	48	47	357	85	90	89	87	6
SW	1	344	349	347	346	43	48	364	357	361	90	89	87	6

Behaviour of these two walking algorithms was also verified for another structure named 'boxcyl' (boxcyl.geo in Netgen, Fig. 4). The starting point was selected as a tetrahedron in top of the left cylinder and destination point was selected as a top of the right cylinder. Unfortunately, neither visibility nor stochastic walks manged to locate the tetrahedron that contains the destination point (Fig. 4). After dozen or so steps both algorithm are looped. They reach the tetrahedron for which a orientation test selects the only one good face through which the walk should be continued, however, the current tetrahedron has no neighbours of common face since this face is on boundary of the mesh.



(a)

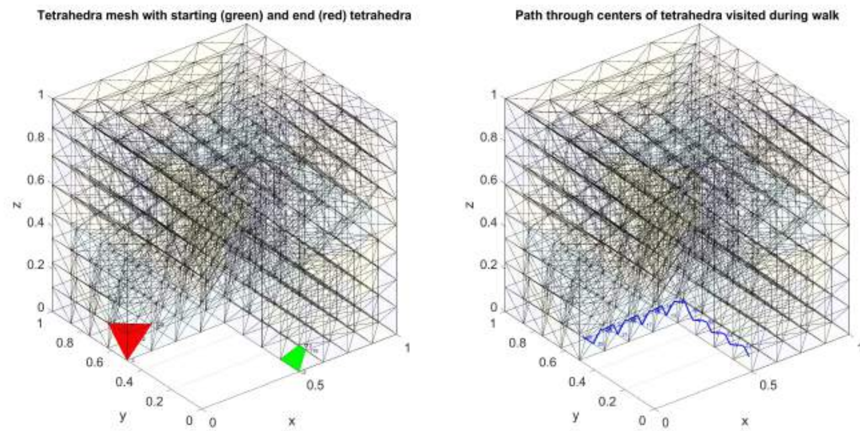


(b)

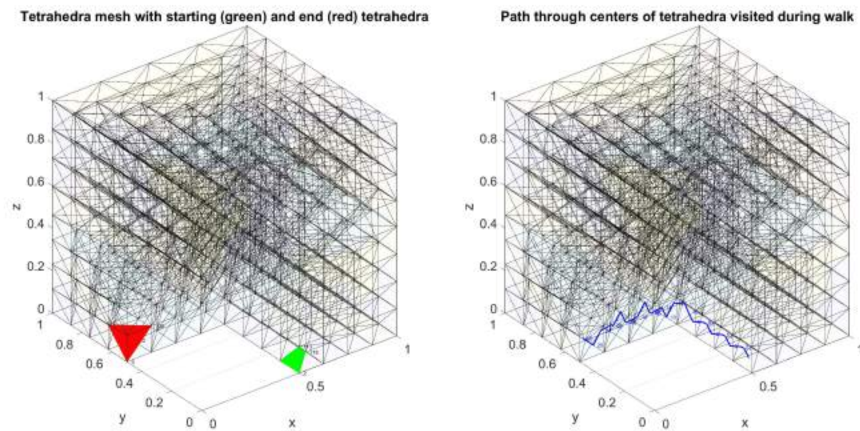
Figure 2: Structure: 'cube'. Visibility walk from starting tetrahedron (green) to tetrahedron that contains a destination point (red).

```
1. Inputs: Nodes, Teta % extracted from *.mesh file
2. Setup:
   max_steps % maximal number of steps during walk
   starting_teta % number of the first teta visited during walk
   destination_point % we find a teta which contains this point represented with coordinates [x,y,z]
   step % current iteration/step
3. [Teta _neigbours] = find4TetaNeigbours( Teta ) // find up to 4 neighbours of each Teta(i,:)
4. current_teta = starting_teta, step = 1
5. while ( step < max_steps)
6.    t = Teta (current_teta,:)
7.    [in_out] = checkPointInsideTeta(t,destination_point,...) ▷ test if the point is in current teta
8.    if (in_out==1)
9.       destination_teta = current_teta
10.      break
11.   end if
11.   for i=1:4 //loop over faces
12.      face_orientation = FaceOrientationTest(t,Nodes,destination_point,...) ▷ calc face orientation: -1,1
13.      if face_orientation == 1 ▷ go through these face to the next teta
14.         current_teta = findTetaNeighboursBy3Nodes(t, Teta_neighbours, current_teta,...) ▷ find a neighbour to visit
15.         break
16.      else
17.         continue
18.      end if
19.   end for
20.   step = step + 1;
21. end while
22. Output: destination_teta
```

Algorithm 1: Visibility walk in tetrahedral mesh

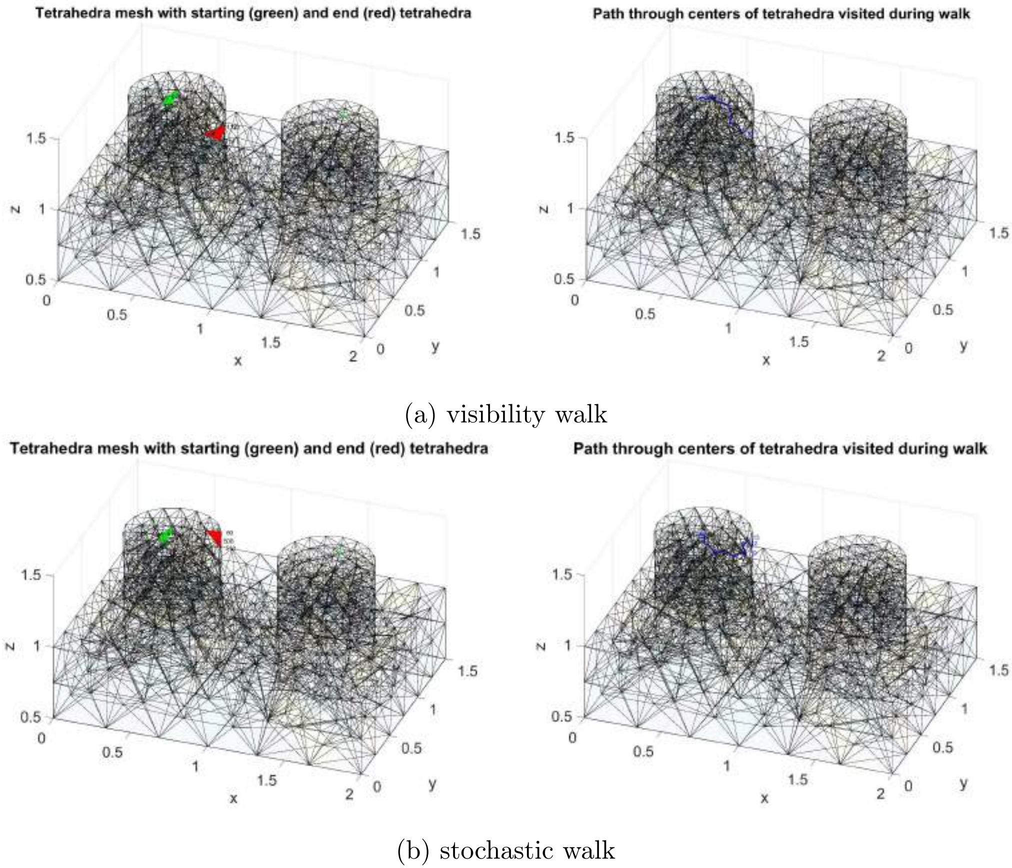


(a) visibility walk



(b) stochastic walk

Figure 3: Structure: 'fichera'. Visibility and stochastic walks from starting tetrahedron (green) to tetrahedron that contains a destination point (red).



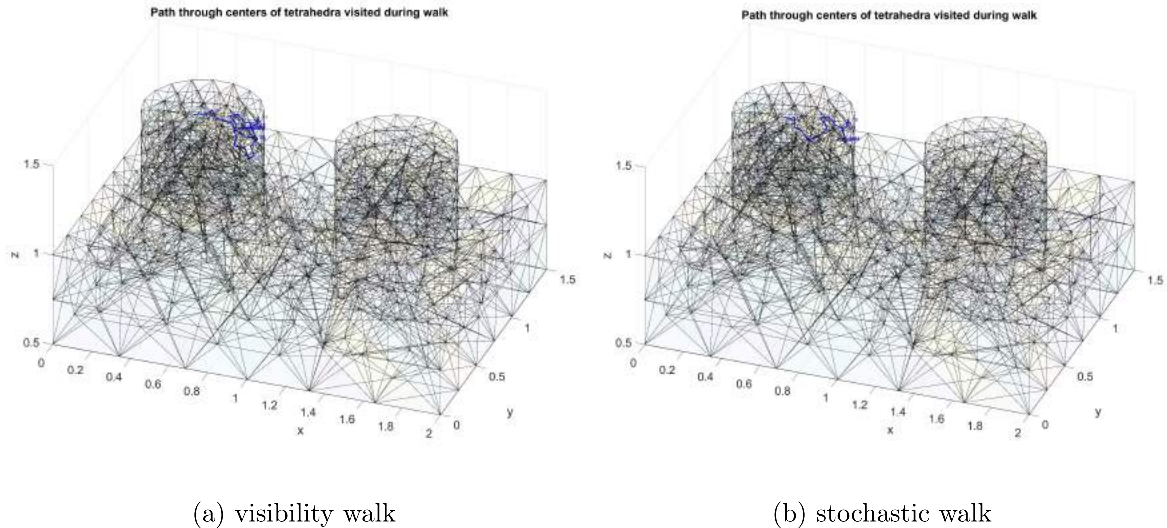
(a) visibility walk

(b) stochastic walk

Figure 4: Structure: 'boxcyl'. Visibility and stochastic walks from starting tetrahedron (green) to tetrahedron that contains a destination point (red).

```
1. Inputs: Nodes, Teta % extracted from *.mesh file
2. Setup:
   max_steps % maximal number of steps during walk
   starting_teta % number of the first teta visited during walk
   destination_point % we find a teta which contains this point represented with coordinates [x,y,z]
   step % current iteration/step
3. [Teta _neigbours] = find4TetaNeigbours( Teta ) // find up to 4 neighbours of each Teta(i,:)
4. current_teta = starting_teta, step = 1
5. while ( step < max_steps)
6.    t = Teta (current_teta,:)
7.    [in_out] = checkPointInsideTeta(t,destination_point,...) ▷ test if the point is in current teta
8.    if (in_out==1)
9.       destination_teta = current_teta
10.      break
11.   end if
12.   face_order = randperm(4) % random permuatation of [1,2,3,4]
13.   for j=1:4 //loop over faces
14.      i = face_order(j)
15.      face_orientation = FaceOrientationTest(t,Nodes,destination_point,...) ▷ calc face orientation: {-1,1}
16.      if face_orientation == 1 ▷ go through these face to the next teta
17.         current_teta = findTetaNeighboursBy3Nodes(t, Teta_neighbours, current_teta,...) ▷ find a neighbour to visit
18.         break
19.      else
20.         continue
21.      end if
22.   end for
23.   step = step + 1;
24. end while
25. Output: destination_teta
```

Algorithm 2: Stochastic walk in tetrahedral mesh



(a) visibility walk (b) stochastic walk

Figure 5: Structure: 'boxcyl', strategy #1. Visibility and stochastic walks from starting tetrahedron (green) to tetrahedron that contains destination point (red).

4 Strategies how to improve the visibility and stochastic walk algorithms

In this section 3 strategies how to improve the effectiveness of the walk algorithms are presented. For each of strategies the additional information about walking is collected in structure *history*:

- history.onPath - array that contains information of the tetrahedra visited during walk,
- history.wrong - array that contains information of the tetrahedra visited during walk that were selected as 'wrong' because an orientation test selected a face and a current tetrahedron does not have any neighbour through that face, while other faces were verified negatively.

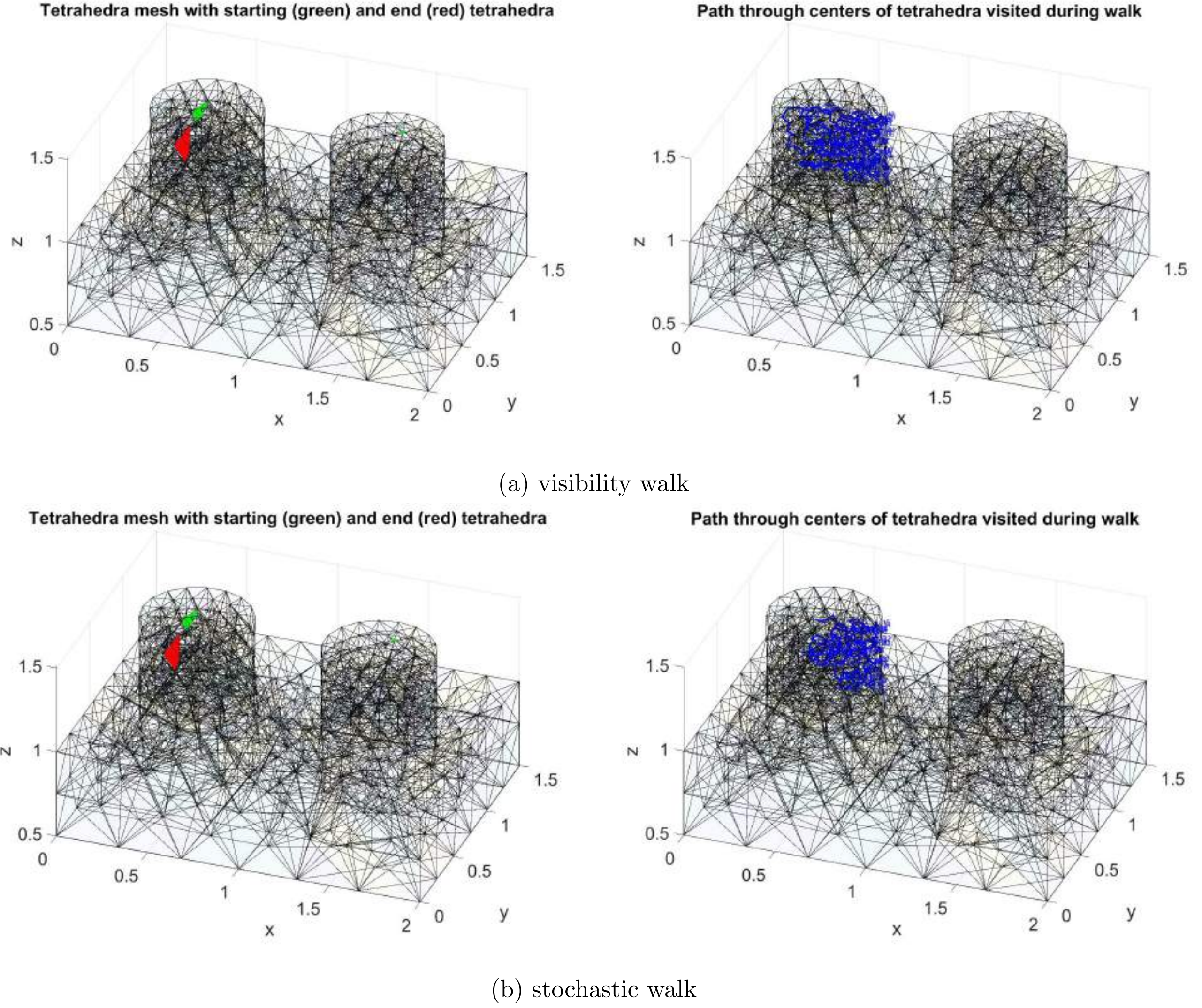
4.1 Strategy 1 - go to the first neighbour on the list

The first strategy is as follows: if you are in 'wrong' tetrahedron, visit the first tetrahedron from the list of neighbours, despite the fact that the orientation test could have verified it negatively (orientation3D=-1).

Figure 5 presents the result of application of the first strategy. This strategy did not pay off in both variants of walking. The tetrahedral mesh has 3072 elements, so the max_steps=3072 (the naive search for tetrahedron in entire mesh would require at most 3072 steps). During walk there were 3049 and 3056 'wrong' visited tetrahedra in visibility and stochastic walks, respectively. It means that more than 99% of tetrahedra were visited unnecessarily.

4.2 Strategy 2 - go to the neighbour with minimum value of face orientation test

The second strategy is based on the orientation test. However, despite the sign of the determinant, we also store in memory the value of the orientation test (Eq. 3). In this case if the currently visited tetrahedron is verified by the algorithm as 'wrong', then the next visited tetrahedron is the neighbour with **minimal value** of $orientation3D_value$. In other words: go in completely opposite direction that is suggested by the orientation test. Figure 6 presents the result of application of the second strategy. This strategy did not pay off in both variants of walking. During walk there were 2924 and 2974 'wrong' visited tetrahedra in visibility and stochastic walks, respectively. It means that more than 99% of tetrahedra were visited unnecessarily. An additional variant was also verified in which not only a minimal value of $orientation3D_value$ is considered but also a next visited tetrahedron cannot be a member of history.onPath (was not visited before). Unfortunately, this remembering concept did not help for the analyzed structure and algorithm quickly looped.



(a) visibility walk

(b) stochastic walk

Figure 6: Structure: 'boxcyl', strategy #2. Visibility and stochastic walks from starting tetrahedron (green) to tetrahedron that contains destination point (red).

$$orientation3D_value(\mathbf{t},\mathbf{u},\mathbf{v},\mathbf{w}) = \begin{vmatrix} u_x - t_x & v_x - t_x & w_x - t_x \\ u_y - t_y & v_y - t_y & w_y - t_y \\ u_z - t_z & v_z - t_z & w_z - t_z \end{vmatrix} \quad (3)$$

4.3 Strategy 3 - go to the random neighbour

The third strategy is as follows: if you are in 'wrong' tetrahedron, visit the **random** tetrahedron from the list of neighbours, despite the fact that the orientation test could have verified it negatively (orientation3D=-1) (Algorithms 3-4). This strategy payed off. Visibility walk (VW) and stochastic walk (VW) required 863 and 913 steps to reach the destination point. During the VW and SW walks 665 (77%) and 743 (81%) tetrahedra were classified as 'wrong'. Figure 7 confirms that the setup (*starting_teta*, *destination_point*) is very unfriendly, so both algorithms were reaching for the boundary of the left cylinder, however, thanks to a random selection of next tetrahedron (may be opposite direction than suggested by the orientation test) both algorithms managed to escape from the left cylinder and found the tetrahedron which contains a destination point.

The third strategy was also verified for more complicated and not simply connected structure: 'shaft' (shaft.geo in Netgen, Fig. 1f). Since the strategy #3 contains a random selection of the next tetrahedron in case the current tetrahedron is classified as a 'wrong' tetrahedron the number of steps in walk can be different. In Table 3 data from four tests are collected. One may observe that in average

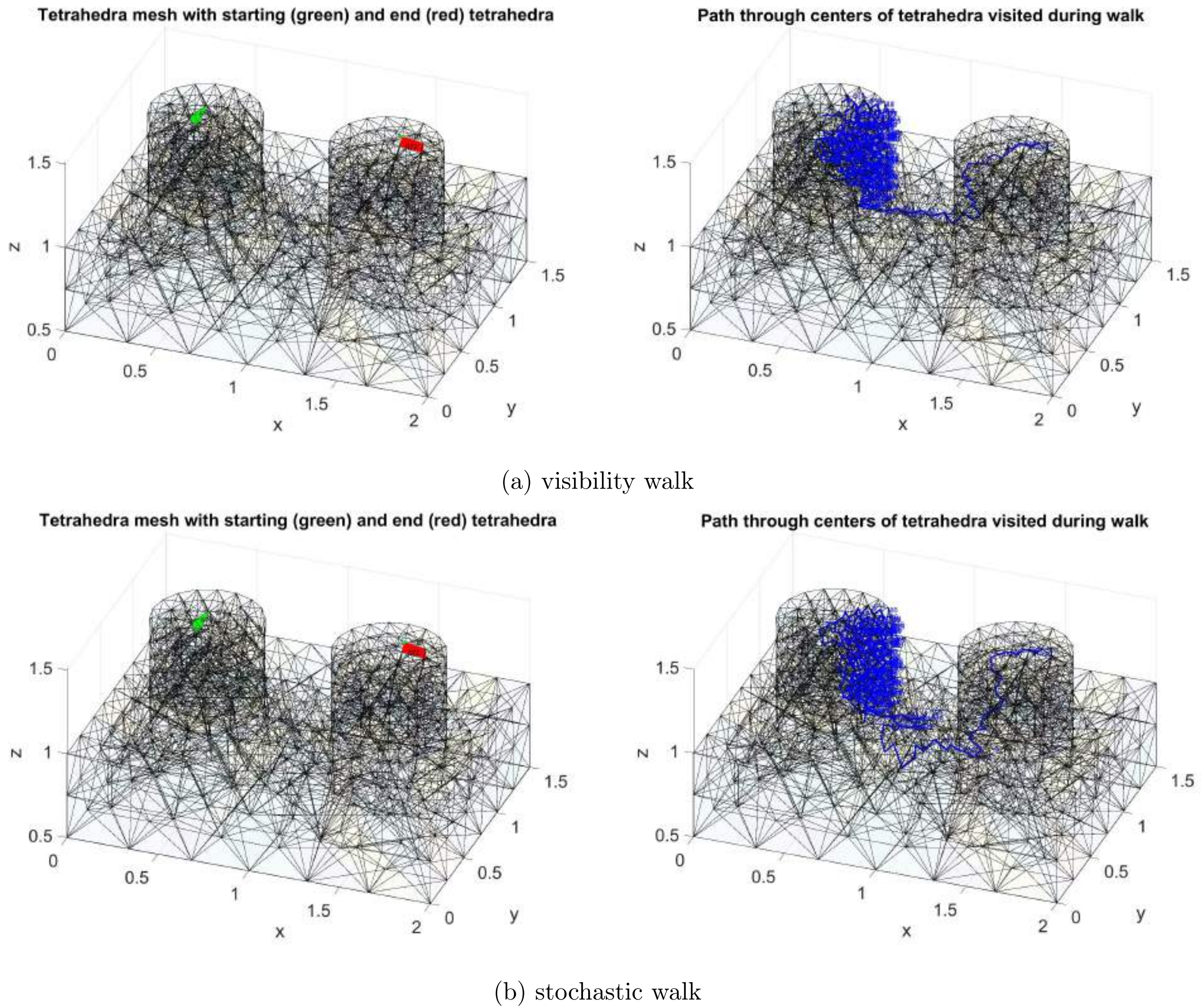


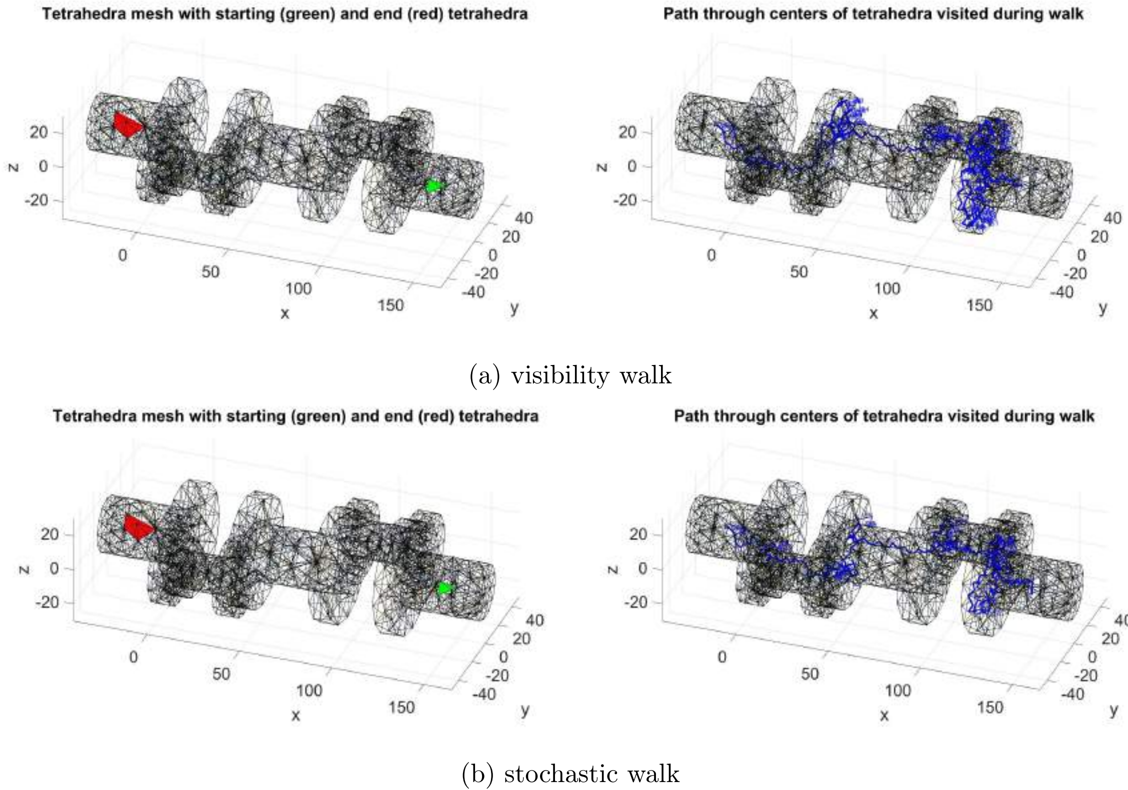
Figure 7: Structure: 'boxcyl', strategy #3. Visibility and stochastic walks from starting tetrahedron (green) to tetrahedron that contains destination point (red).

visibility walk (#3) requires more steps to find a destination point than a stochastic walk (#3). It is due to the fact that the number of 'wrong' tetrahedra is significantly lower. Figure 8 presents paths obtained with visibility and stochastic walks for which a strategy #3 was applied (test no. 1 from Tab. 3).

Table 3: List of tetrahedra visited in visibility walk (VW) and stochastic walk (SW) [Structure: 'shaft'].

Test number	VW (all steps)	VW (wrong)	SW (all steps)	SW (wrong)
1	389	209 (53%)	287	118 (41%)
2	545	328 (60%)	229	80 (35%)
3	349	169 (48%)	283	121 (43%)
4	533	314 (59%)	200	67 (34%)
Average	454	255 (56%)	250	97 (39%)

The last structure presented in this section is a 'broken torus' (Fig. 1e). Strategy #3 proposed for visibility and stochastic walk algorithms was verified for two setups:



(a) visibility walk

(b) stochastic walk

Figure 8: Structure: 'shaft', strategy #3. Visibility and stochastic walks from starting tetrahedron (green) to tetrahedron that contains destination point (red).

```
1. Inputs: Nodes, Teta % extracted from *.mesh file
2. Setup:
   max_steps % maximal number of steps during walk
   starting_teta % number of the first teta visited during walk
   destination_point % we find a teta which contains this point represented with coordinates [x,y,z]
   step % current iteration/step
3. [Teta _neigbours] = find4TetaNeigbours( Teta ) // find up to 4 neighbours of each Teta(i,:)
4. current_teta = starting_teta, step = 1, wrong_step = 1, history.path(1) = current_teta
5. while ( step < max_steps)
6.    t = Teta (current_teta,:)
7.    [in_out] = checkPointInsideTeta(t,destination_point,...) ▷ test if the point is in current teta
8.    if (in_out==1)
9.       destination_teta = current_teta
10.       break
11.   end if
12.   previuos_teta = current_teta
13.   for i=1:4 //loop over faces
14.      face_orientation = FaceOrientationTest(t,Nodes,destination_point,...) ▷ calc face orientation: -1,1
15.      if face_orientation == 1 ▷ go through these face to the next teta
16.         current_teta = findTetaNeighboursBy3Nodes(t, Teta_neighbours, current_teta,...) ▷ find a neighbour to visit
17.         break
18.      else
19.         continue
20.      end if
21.   end for
22.   if (previous_teta == current_teta)
23.         history.wrong(wrong_step) = current_teta
24.         neighbour_order = randperm(nnz(Teta_neighbours(current_teta,:)))
25.         current_teta = neighbour_order(1)
26.   end if
27.   step = step + 1;
28.   history.path(step) = current_teta
29. end while
30. Output: destination_teta
```

Algorithm 3: Visibility walk with strategy #3 in tetrahedral mesh.

```
1. Inputs: Nodes, Teta % extracted from *.mesh file
2. Setup:
   max_steps % maximal number of steps during walk
   starting_teta % number of the first teta visited during walk
   destination_point % we find a teta which contains this point represented with coordinates [x,y,z]
   step % current iteration/step
3. [Teta _neigbours] = find4TetaNeigbours( Teta ) // find up to 4 neighbours of each Teta(i,:)
4. current_teta = starting_teta, step = 1, wrong_step = 1, history.path(1) = current_teta
5. while ( step < max_steps)
6.   t = Teta (current_teta,:)
7.   [in_out] = checkPointInsideTeta(t,destination_point,...) ▷ test if the point is in current teta
8.   if (in_out==1)
9.     destination_teta = current_teta
10.      break
11.   end if
12.   previuos_teta = current_teta
13.   face_order = randperm(4) % random permuatation of [1,2,3,4]
14.   for j=1:4 //loop over faces
15.      i = face_order(j)
16.      face_orientation = FaceOrientationTest(t,Nodes,destination_point,...) ▷ calc face orientation: -1,1
17.      if face_orientation == 1 ▷ go through these face to the next teta
18.          current_teta = findTetaNeighboursBy3Nodes(t, Teta_neighbours, current_teta,...) ▷ find a neighbour to visit
19.          break
20.      else
21.          continue
22.      end if
23.   end for
24.   if (previous_teta == current_teta)
25.          history.wrong(wrong_step) = current_teta
26.          neighbour_order = randperm(nnz(Teta_neighbours(current_teta,:)))
27.          current_teta = neighbour_order(1)
28.   end if
29.   step = step + 1;
30.   history.path(step) = current_teta
31. end while
32. Output: destination_teta
```

Algorithm 4: Stochastic walk with strategy #3 in tetrahedral mesh

setup #1:
starting_teta = 2213 // a node with the lowest value of z-coordinate: [0.0545 -0.8966 2.0365]
destination_point = [0.1435 1.3673 -0.100]

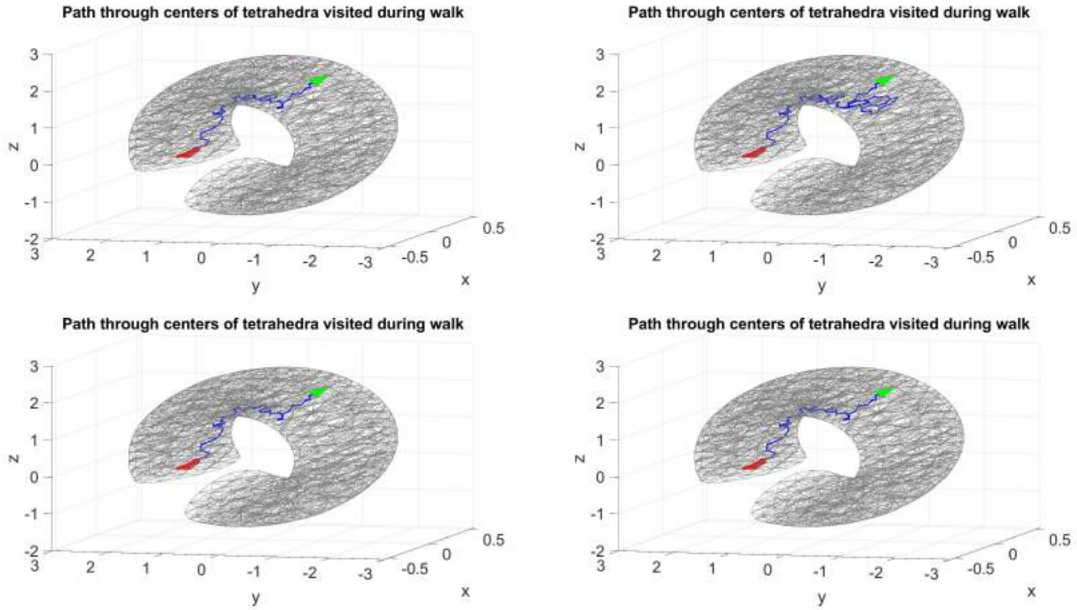
setup #2:
starting_teta = 1016 // a node with the lowest value of z-coordinate: [0.0525 -1.4876 0.6601]
destination_point = [0.1435 1.3673 -0.100]

For the first setup both visibility and stochastic walks performed very well (Figure 9, Table 4) and found the destination point (tetrahedron). It can be noticed that stochastic walk required fewer steps and there were lower number of 'wrong' tetrahedra on the path.

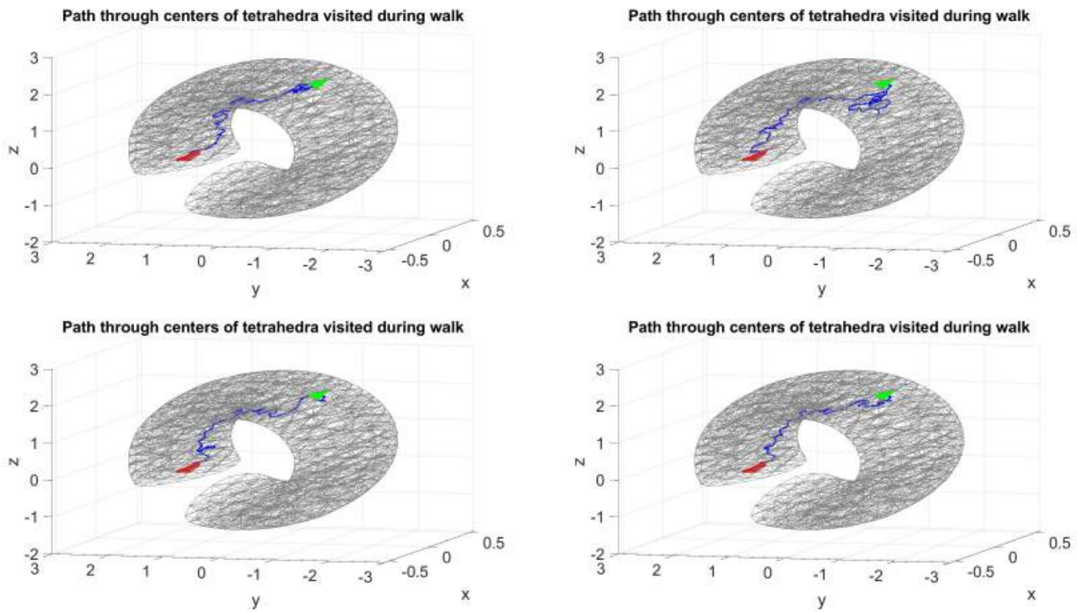
For the second setup, in which the starting tetrahedron has significantly lower z-coordinate (the difficulty level is higher), the visibility walk failed for all tests. Stochastic walk found the destination point in three of four tests (Figure 10, Table 5).

Conclusions of this section:

1. The best effectiveness of walks implemented by the author of this report was obtained for a stochastic walk with strategy #3 (referenced as SW#3 in the following sections).
2. For inconveniently located starting tetrahedron SW#3 requires many steps or fails to find a destination point.

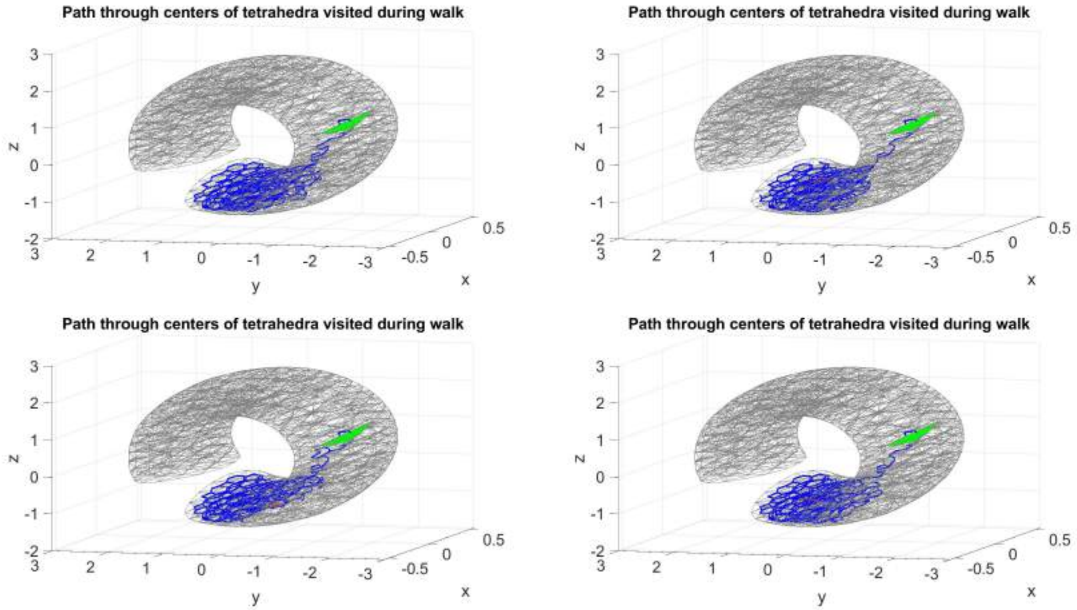


(a) visibility walks

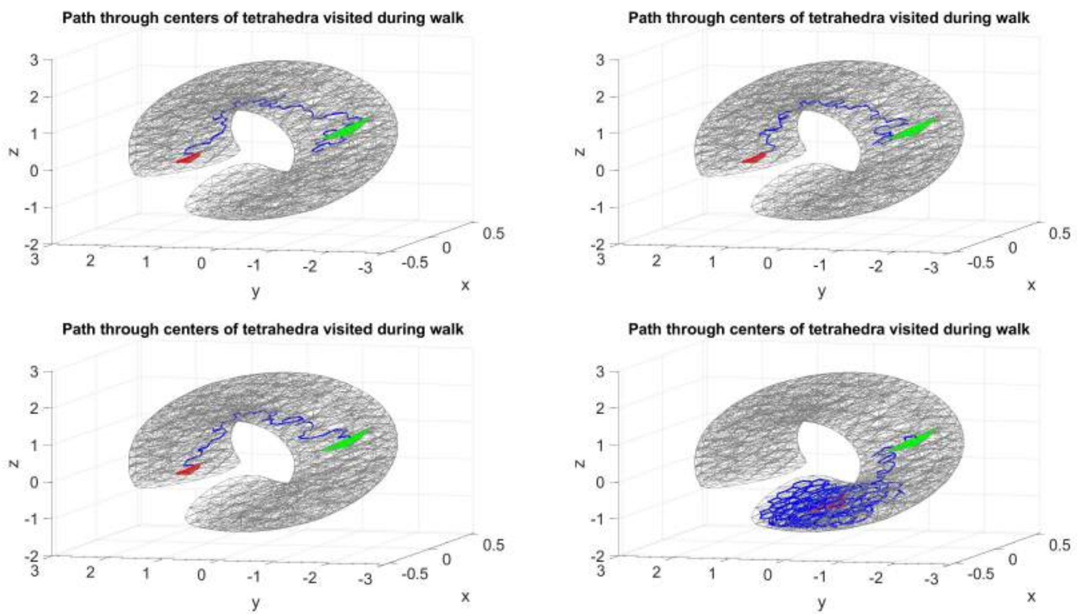


(b) stochastic walks

Figure 9: Structure: 'broken torus', strategy #3, setup #1. Four visibility (a) and stochastic (b) walks from starting tetrahedron (green) to tetrahedron that contains destination point (red).



(a) visibility walks



(b) stochastic walks

Figure 10: Structure: 'broken torus', strategy #3, setup #2. Four visibility (a) and stochastic (b) walks from starting tetrahedron (green) to tetrahedron that contains destination point (red).

Table 4: List of tetrahedra visited in visibility walk (VW) and stochastic walk (SW). Structure: broken torus, setup #1.

Test number	VW (all steps)	VW (wrong)	SW (all steps)	SW (wrong)
1	62	8 (13%)	71	12 (17%)
2	129	58 (45%)	82	20 (24%)
3	56	6 (11%)	57	4 (7%)
4	54	3 (6%)	51	4 (7%)
Average	75	19 (19%)	65	10 (14%)

Table 5: List of tetrahedra visited in visibility walk (VW) and stochastic walk (SW). Structure: broken torus, setup #2.

Test number	VW (all steps)	VW (wrong)	SW (all steps)	SW (wrong)
1	2650	2414 (91%)	114	44 (39%)
2	2650	2439 (92%)	97	33 (34%)
3	2650	2443 (92%)	72	16 (22%)
4	2650	2432 (92%)	2650	2391 (90%)
Average	-	- (-%)	-	- (-%)

5 Determine the set of destination points (tetrahedra)

In this section three examples are discussed in which the stochastic walk (SW#3) algorithm was employed to find multiple destination points in a mesh.

5.1 List of destination points (tetrahedra) to be found

In the first case the destination points come from the list of designation points:
DestinationPoints =
// x y z
0.2000 0.2000 0.2000
-10.0000 -8.1393 10.8942
-38.0000 -0.4678 0.2487
60.0000 -0.4678 0.2487

Figure 11 confirms that 4 tetrahedra including points from DestinationPoints list were found with stochastic walk (SW#3) algorithm.

5.2 Coarse and fine mesh

In this subsection we consider the same geometry however two meshes with different densities were generated. The tested geometry is 'fishera'. Coarse and fine meshes contain 37 and 2368 tetrahedra, respectively. The coarse mesh has 24 nodes.

Figure 12 presents tetrahedra which contain nodes of the coarse mesh that were found with stochastic walk (SW#3) algorithm. In case destination points are nodes of the mesh there are few tetrahedra which contain such a point (node). Thus after a single walk if a destination tetrahedron is found then an additional test is done in order to find all tetrahedra which contain the node. Result of this additional verification is presented in Fig. 13 where all tetrahedra that contain a node are displayed (blue - found by the walk, green - found in the additional verification after the walk).

The second tested geometry is 'boxcyl'. Coarse and fine meshes contain 384 and 24576 tetrahedra, respectively. The coarse mesh has 139 nodes (destination points/tetrahedra) and all 139 tetrahedra were found with stochastic walk (SW#3) (Figure 14). What is more, neighbours of found tetrahedra during a walk were tested (after a walk) and Fig. 15 presents all tetrahedra that contain 139 nodes.

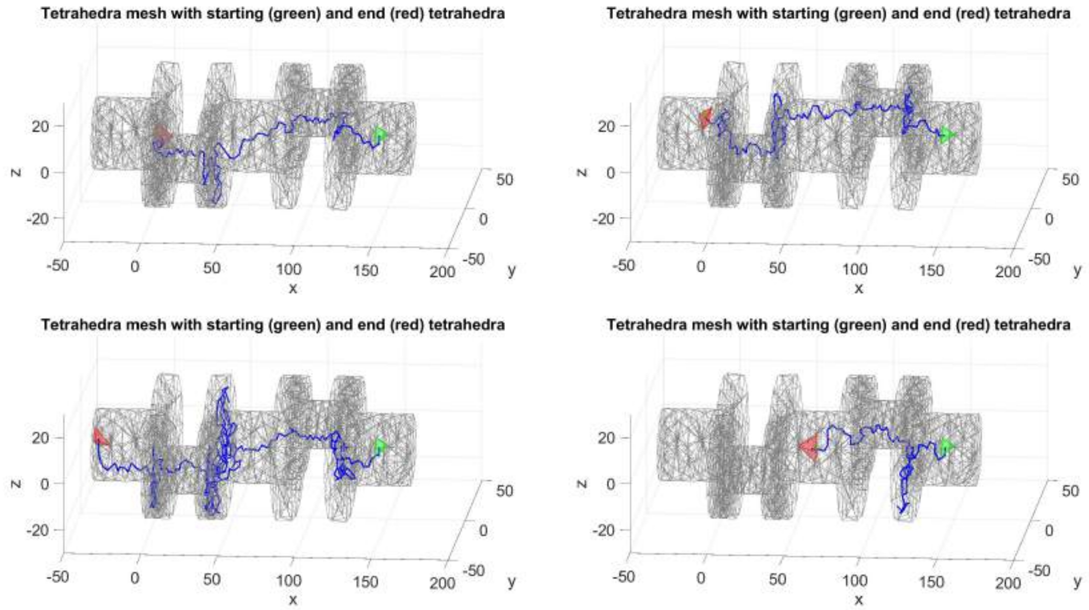


Figure 11: Structure: 'broken torus', strategy #3. Four calls of stochastic walks from starting tetrahedron (green) to four different tetrahedron that contain destination point (red).

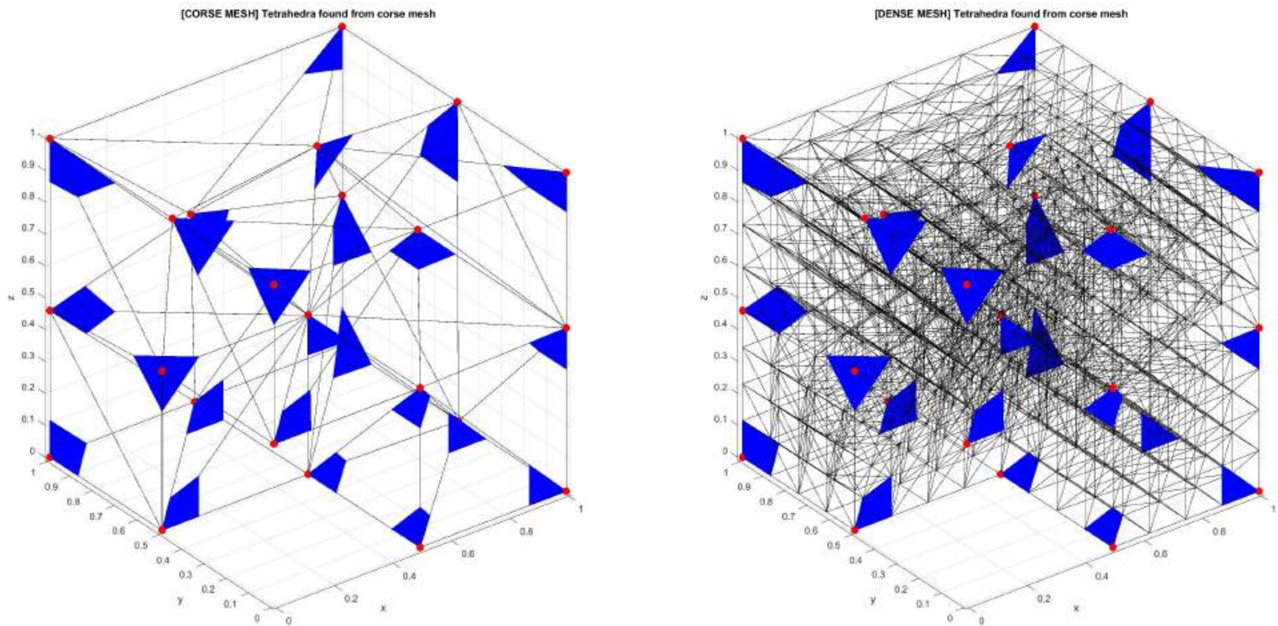


Figure 12: Structure: 'fishera', strategy #3. Result of 24 calls of stochastic walks performed in order to find 24 unique tetrahedra that contain nodes of the coarse mesh (blue).

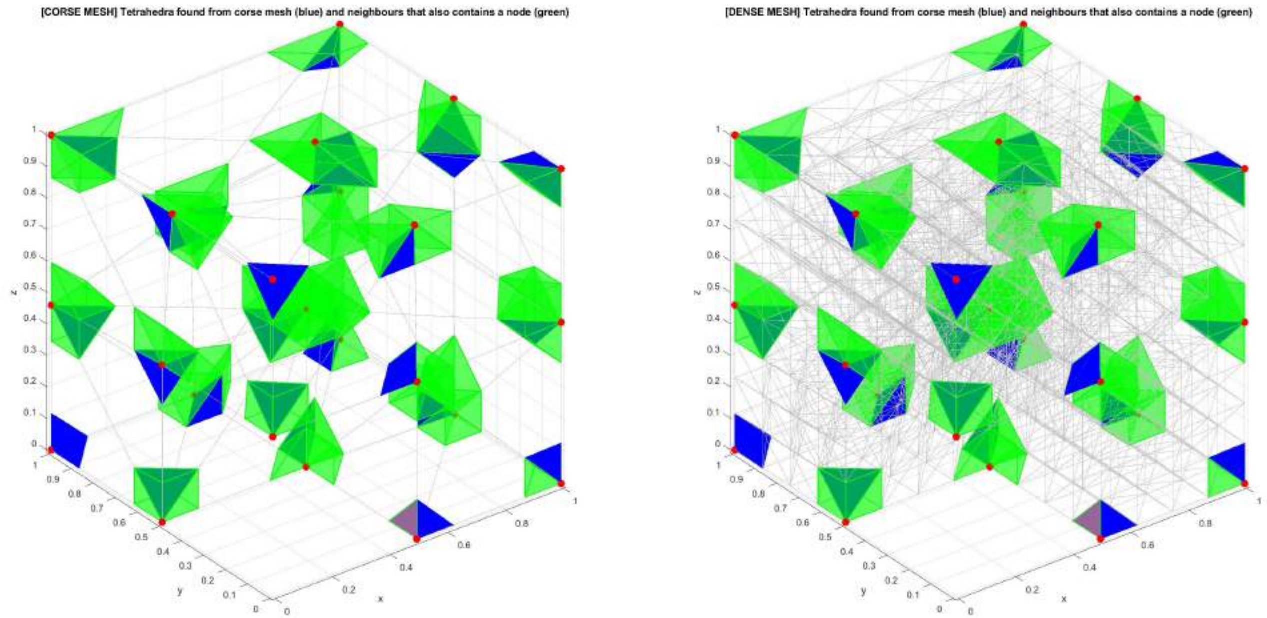


Figure 13: Structure: 'fishera', strategy #3. Result of 24 calls of stochastic walks performed in order to find 24 unique tetrahedra that contain nodes of the coarse mesh (blue) and tetrahedra found in additional test (green) performed after a walk.

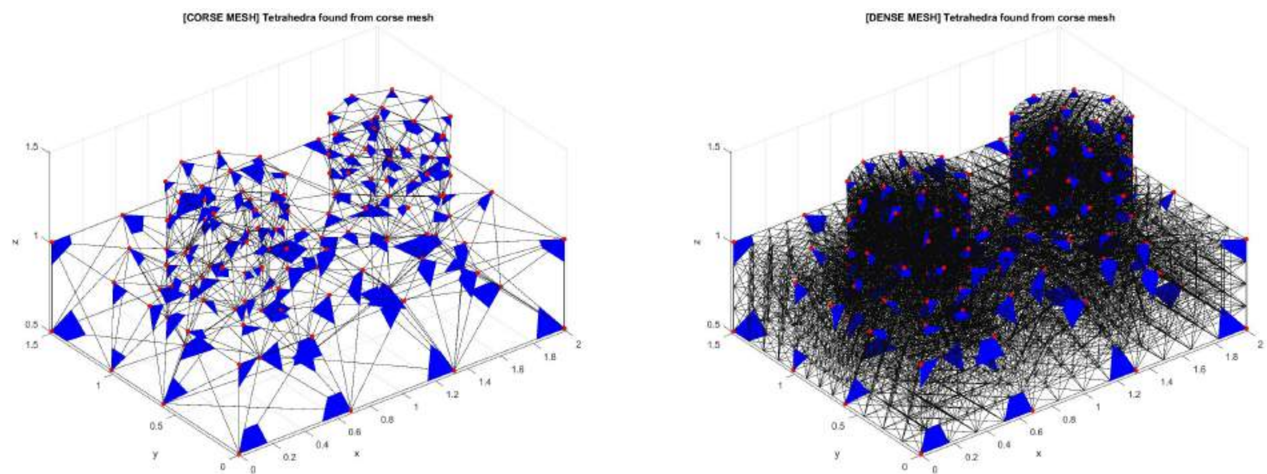


Figure 14: Structure: 'boxcyl', strategy #3. Result of 139 calls of stochastic walks performed in order to find 139 unique tetrahedra that contain nodes of the coarse mesh (blue).

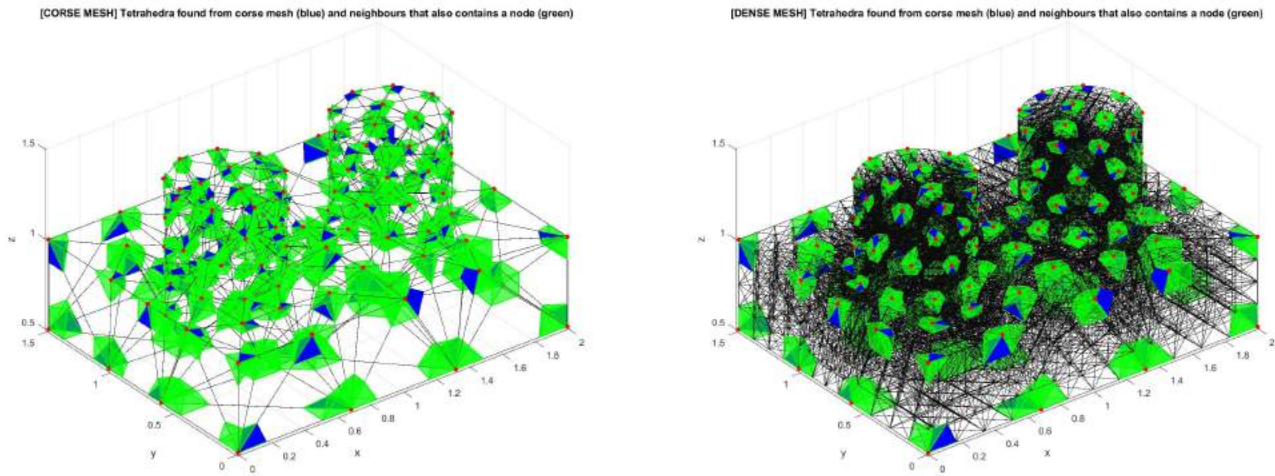


Figure 15: Structure: 'boxcyl', strategy #3. Result of 24 calls of stochastic walks performed in order to find 139 unique tetrahedra that contain nodes of the coarse mesh (blue) and tetrahedra found in additional test (green) performed after a walk.

The third geometry 'cubeandspheres' is the most unfriendly of verified in this subsection since there are not simply connected regions of the structure. Coarse and fine mesh contain 98 and 6272 tetrahedra, respectively. There are 66 nodes in a coarse mesh and 65 tetrahedra were found with stochastic walk (SW#3) (Figure 16). Unfortunately, one node was not found. What is more, neighbours of 65 found tetrahedra were tested and Figure 17 presents all tetrahedra that contain 65 nodes.

5.3 Range of nodes

In this subsection we consider an example in which the task is to find all tetrahedra which contain destination points that are nodes from given range of x-,y-,z- coordinates. Figures 18-19 present results of both the stochastic walks and an additional verification of tetrahedra for two ranges: $79<x<90$ and $15<x<20$, respectively. In the first case (Fig. 18) a stochastic walk found as many tetrahedra (65) as there were destination points (nodes) and an additional test upgraded these sets with a list of neighbours which contains the nodes as well. In the second case (Fig. 19) a stochastic walk found 15 tetrahedra (of 17 destination points), however, the two remaining were found among neighbours.

5.4 Destination point on a face of the tetrahedra

In order to verify if a destination point lies on the face of tetrahedron after each walk all neighbours of the found tetrahedron are verified. Concluding with the previous subsection the entire procedure is as follows :

1. A single stochastic walk (SW#3) finishes after finding a destination tetrahedron (single index of a row of Teta array), then
2. the first test is performed (loop over nodes) to verify if the destination point is a node of a found tetrahedron, if it is true then tetrahedra are verified to select which of them contain a destination point, then
3. the second test is performed (loop over faces) to verify if the destination point lies on a face of a tetrahedron, if it is true then neighbours are verified to select which neighbours contain a destination point (one neighbour if point lies on a face, few neighbours if point lies on an edge), then
4. output of such a walk and verification are: an index of found tetrahedra during walk (1), a list of neighbours 'by a node' (2), list of neighbours 'by a face' (3) and an unique list of all tetrahedra that contain the destination point.

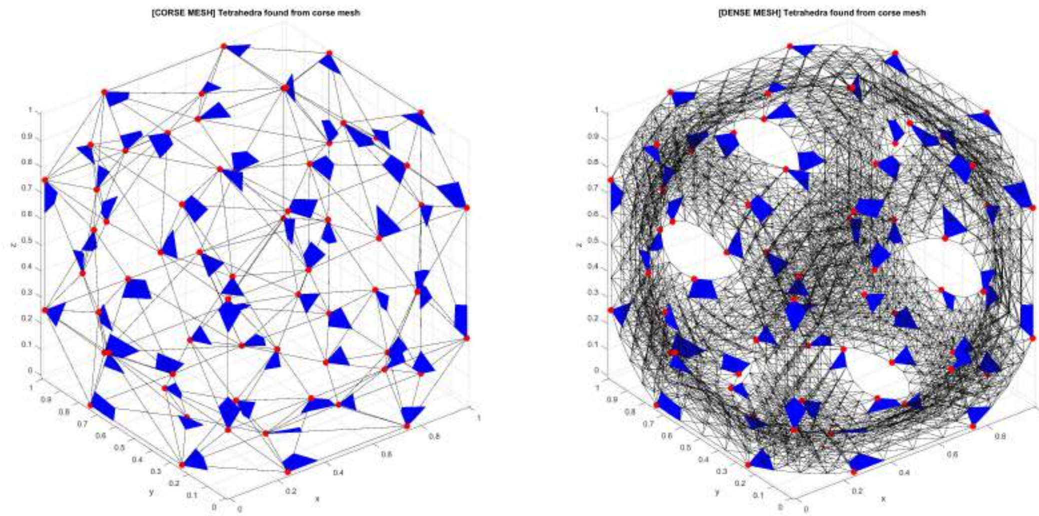


Figure 16: Structure: 'cubeandspheres', strategy #3. Result of 66 calls of stochastic walks performed in order to find 65 unique tetrahedra that contain nodes of the coarse mesh (blue).

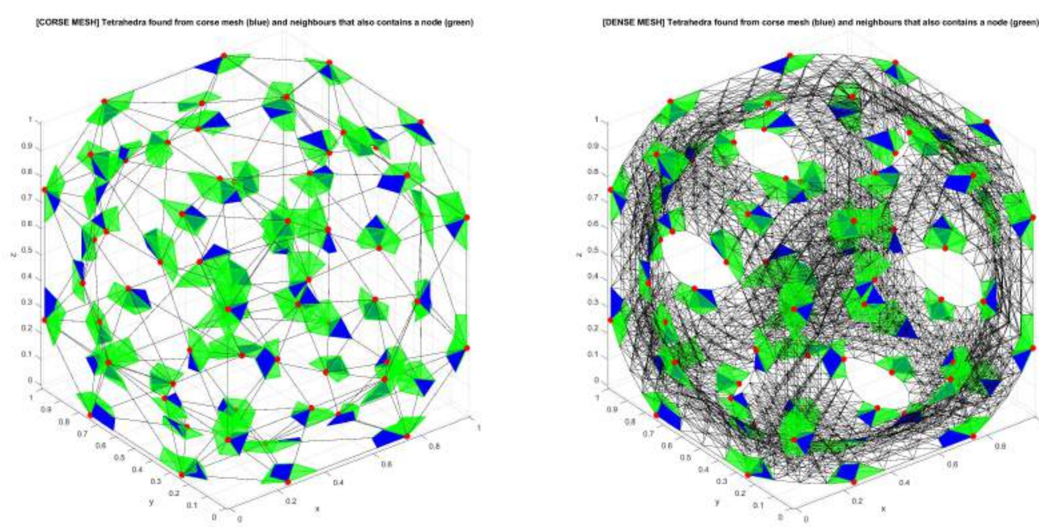


Figure 17: Structure: 'cubeandspheres', strategy #3. Result of 66 calls of stochastic walks performed in order to find 65 unique tetrahedra that contain nodes of the coarse mesh (blue) and tetrahedra found in additional test (green) performed after a walk.

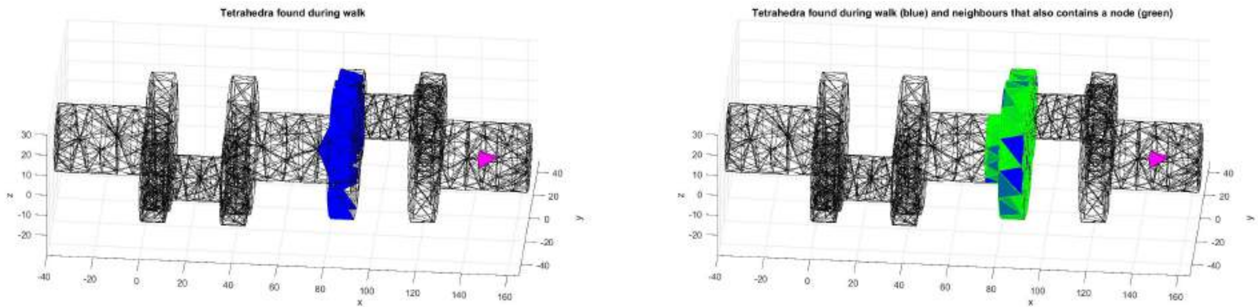


Figure 18: Structure: 'shaft', strategy #3. Result of 65 calls of stochastic walks performed in order to find 65 unique tetrahedra that contain nodes of the coarse mesh (blue) from a determined range and tetrahedra found in additional test (green) performed after a walk.

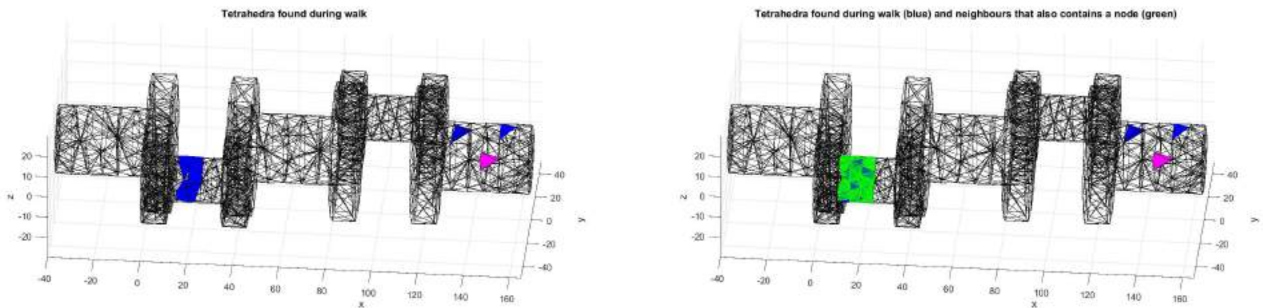


Figure 19: Structure: 'shaft', strategy #3. Result of 17 calls of stochastic walks performed in order to find 15 unique tetrahedra that contain nodes of the coarse mesh (blue) from a determined range and tetrahedra found in additional test (green) performed after a walk.

The procedure is repeated in case there are several destination points.

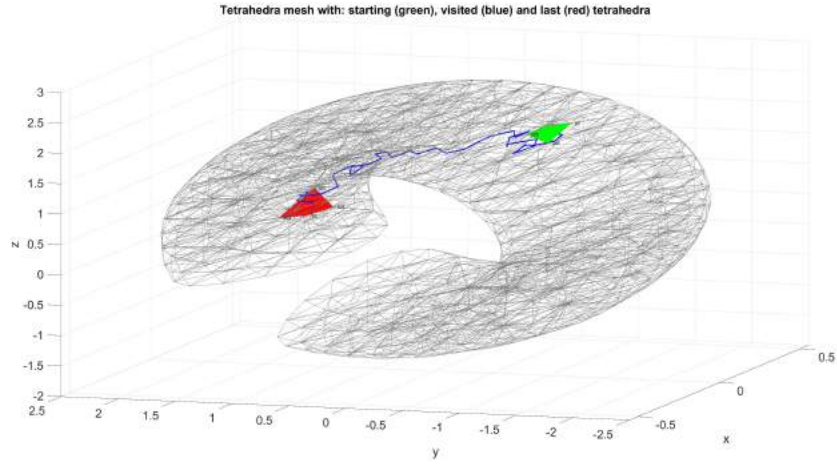
6 Jump and Walk

In [4] it is reported that supporting a walk algorithm with a *jump* to the starting tetrahedron is beneficial. Thus, after presenting the features of a stochastic walk (SW#3) in previous sections the impact of a selection of starting tetrahedron is described. In tests a shaft structure is used with two meshes with 51k and 409k tetrahedra. The goal is to find 324 destination points that come from the very coarse mesh of 800 tetrahedra.

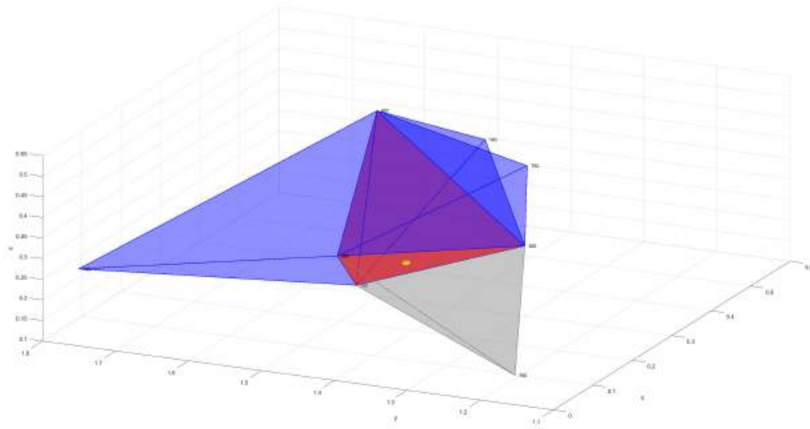
The jump process is as follows. A random set of $TetaJump$ tetrahedra is selected and distances between selected tetrahedra and destination points are calculated. Thus, in the implementation used in this report for each randomly selected tetrahedra a middle point of a simplex has to be calculated. A starting tetrahedron is the tetrahedron which has the smallest distance (as the crow flies) between its middle point and a destination point.

The influence of the number of randomly verified tetrahedra ($TetaJump$) in the jump phase is presented in Tables 6 and 7 which gather data collected from executions of 'jump-and-walk' with different setups ($TetaJump = \{5\%, 1\%, 0.5\%, 0.1\%, 0.05\%, 0.01\%, 0.005\%, 0.001\%\}$).

The more tetrahedra selected in a jump phase ($TetaJump$) then the distance between a starting tetrahedron and a destination point is shorter (Figs. 22-23) and as a result the number of steps in a walk should decrease (however, it should be kept in mind that in non simply connected regions a small distance may not guarantee a low number of steps). Moreover, Figure 21 shows the percentage share of jump and walk phases depending on the number of $TetaJump$, which suggests that for the entire localization algorithm is crucial to select an optimal balance between jump (number of $TetaJump$) and walk (number of steps in walk) phases. For a mesh with 51k tetrahedra it is preferable to chose a random set including 0.5% (256) tetrahedra in the jump phase. This selection guarantied both the



(a) Path



(b) destination point on a face

Figure 20: Structure: 'broken torus', strategy #3. Stochastic walk from starting tetrahedron (green) to tetrahedron that contains destination point (red) which lies on the face. Right figure visualize the verification of neighbours (grey - the neighbour that has a common face on which lies the destination point, blue - other neighbours).

Table 6: Statistics of a jump-and-walk algorithm for different setups of tetrahedra used in jump phase. [Structure: shaft, no. of tetrahedra: 51200, Destination points 324, $max_iters = 200$]

Tetrahedra in Jump [%]	5	1	0.5	0.1	0.05	0.01	0.005	0.001
Jump [s]	0.84	0.18	0.09	0.03	0.02	0.24	0.03	0.06
Walk [s]	0.69	0.88	0.85	1.46	1.93	4.22	6.03	8.71
Jump and Walk [s]	1.53	1.06	0.94	1.49	1.95	4.46	6.07	8.76
Jump [%]	55%	17%	10%	2%	1%	5%	1%	1%
Walk [%]	45%	83%	90%	98%	99%	95%	99%	99%
$TetaJump$	2560	512	256	52	26	6	3	1
Tetrahedra in Walk (onPath)	3	6.8	8.4	19.3	27.6	66.3	97	143.6
Tetrahedra in Walk (Wrong)	0	0	0	1.9	4.4	19.9	34.1	56.7
Success	324	324	324	**322**	**317**	**283**	**237**	**157**
Failed	0	0	0	**2**	**7**	**41**	**85**	**163**

Table 7: Statistics of a jump-and-walk algorithm for different setups of tetrahedra used in jump phase. [Structure: shaft, no. of tetrahedra: 409600, Destination points 324, $max_iters = 200$]

Tetrahedra in Jump [%]	5	1	0.5	0.1	0.05	0.01	0.005	0.001
Jump [s]	7.10	1.51	0.75	0.17	0.09	0.03	0.02	0.01
Walk [s]	1.71	2.42	2.32	2.64	2.88	4.06	4.92	7.53
Jump and Walk [s]	8.81	3.93	3.07	2.80	2.96	4.08	4.93	7.54
Jump [%]	81%	38%	24%	6%	3%	1%	0%	0%
Walk [%]	19%	62%	76%	94%	97%	99%	100%	100%
$TetaJump$	20480	4096	2048	410	205	41	21	5
Tetrahedra in Walk (onPath)	3	6.4	8.3	15.2	20.5	43.7	60.7	105.6
Tetrahedra in Walk (Wrong)	0	0	0	0.03	0.4	4.6	9.7	24.6
Success	324	324	324	324	**323**	**315**	**302**	**242**
Failed	0	0	0	0	**1**	**9**	**22**	**80**

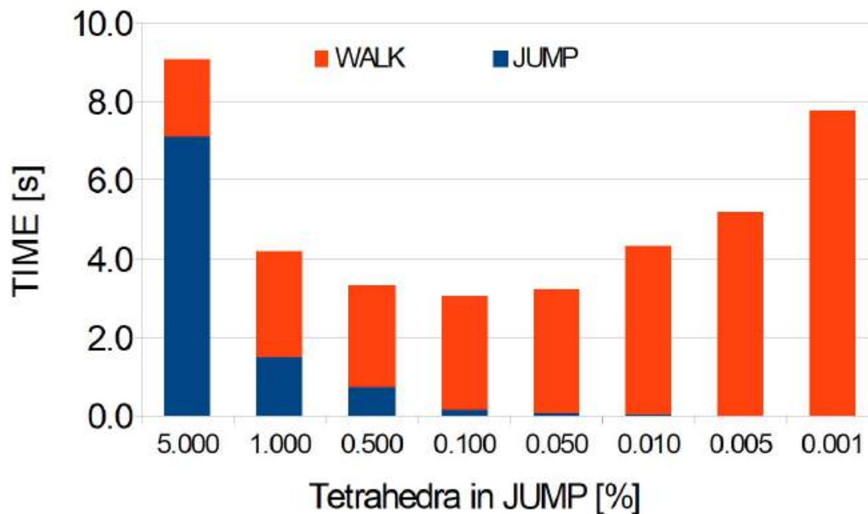
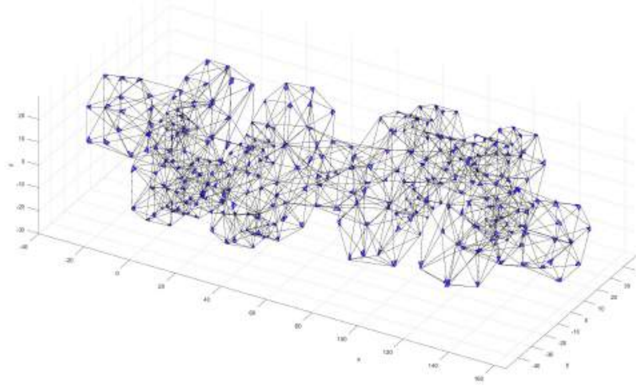


Figure 21: Times taken by jump (blue) and walk (red) phases for different setups of tetrahedra used in a jump phase. [Structure: 'shaft' with 409k tetrahedra, walk: stochastic walk with strategy #3.]

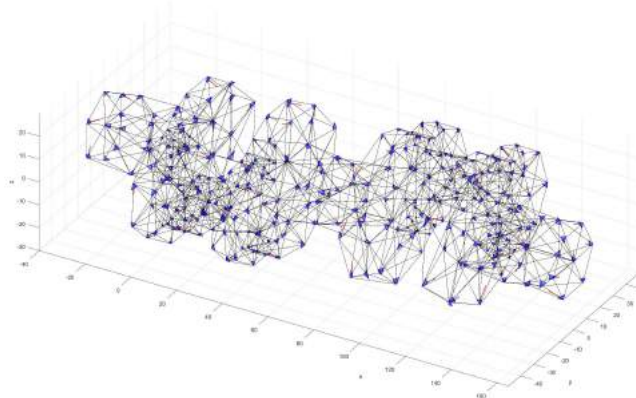
minimal time of 'jump-and-walk' and enabled to find all destination points. For a finer mesh with 409k tetrahedra it occurred that the optimal choice is a selection of 0.1% (410) tetrahedra in the jump phase. One may notice that with the decrease of the number of $TetaJump$ (0.05% and less) the number of failed walks grows[1]. Thus the jump-and-walk algorithm was modified in such a manner that in case the walk fails then the jump phase is run again (with significantly higher number of randomly selected tetrahedra) to select a better starting tetrahedron and walk is repeated. It is worth mentioning that in this looped variant the maximal number of steps (max_steps) is significantly lower, since we know that each step of walk is very time-consuming and we want to quickly verify if there is a need to repeat the walk with a more favorably located starting tetrahedron. Figures 24 presents a comparison of a 'jump-and-walk' algorithm and looped 'jump-and-walk' algorithm. The looped approach allowed finding all destination points (tetrahedra) that were not found previously.

Tests performed for a 'cubeandspheres' structure with 551k tetrahedra confirmed that 0.1% tetrahedra in a jump phase guaranties that all destination points are found in the shortest time.

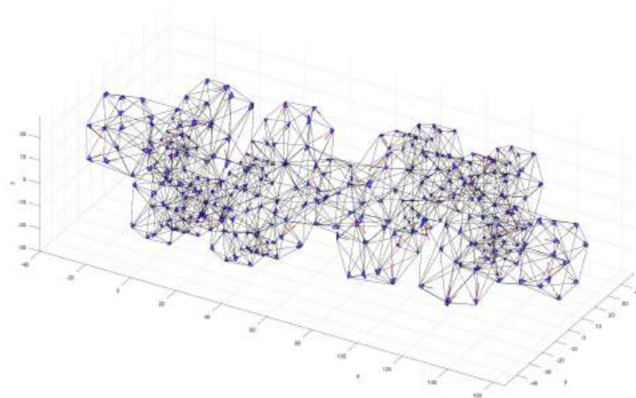
[1]It is worth noting, that here the limiter max_steps was set very restrictive to only 200 steps, according to previous section a stochastic walk (SW#3) enables to find a destination point in most of the cases, however, may require hundreds of steps



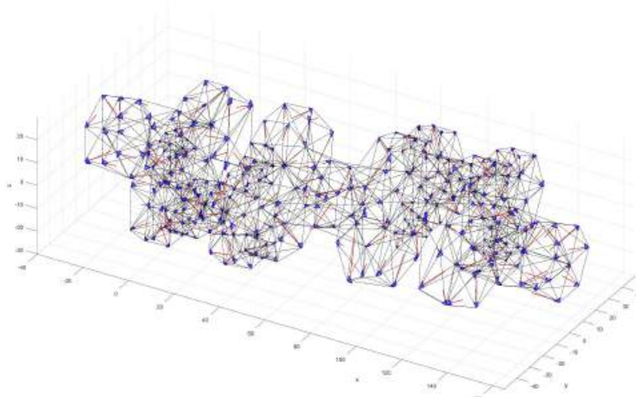
(a) Tetrahedra in Jump = 5%



(b) Tetrahedra in Jump = 1%

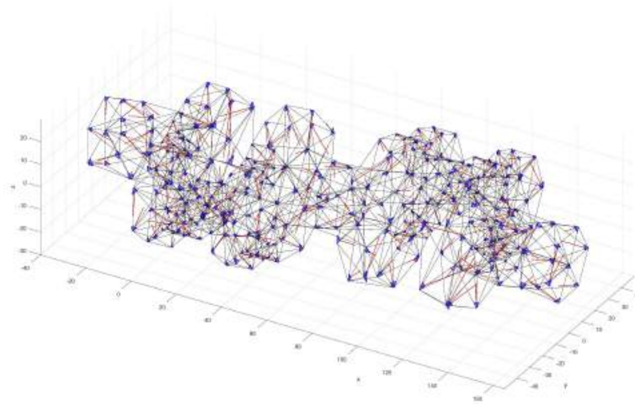


(c) Tetrahedra in Jump = 0.5%

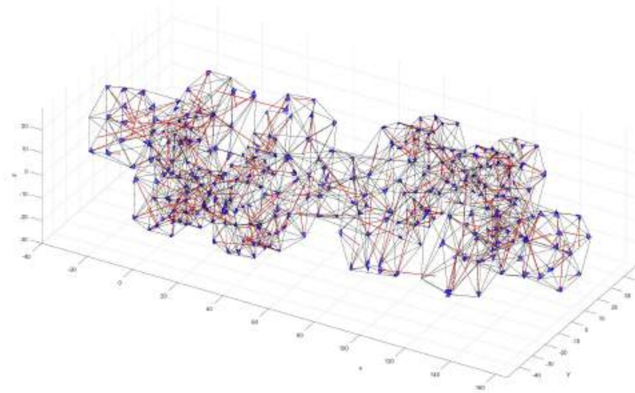


(d) Tetrahedra in Jump = 0.1%

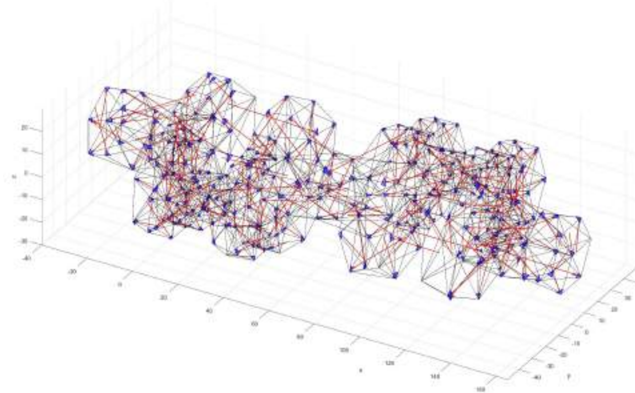
Figure 22: Distances between middle points of starting tetrahedra and and destination points. Randomly selected tetrahedra in a jump phase: $TetaJump = \{5\%, 1\%, 0.5\%, 0.1\%\}$.



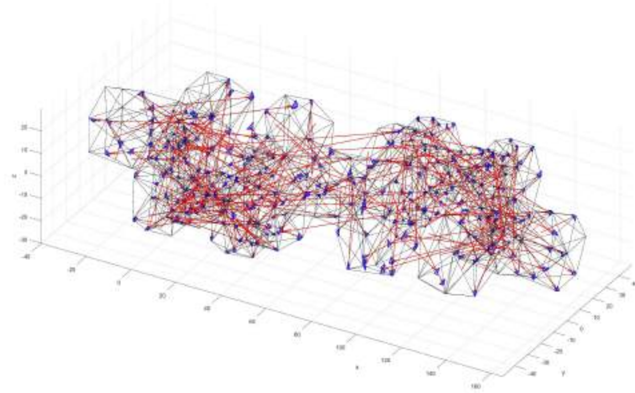
(a) Tetrahedra in Jump = 0.05%



(b) Tetrahedra in Jump = 0.01%

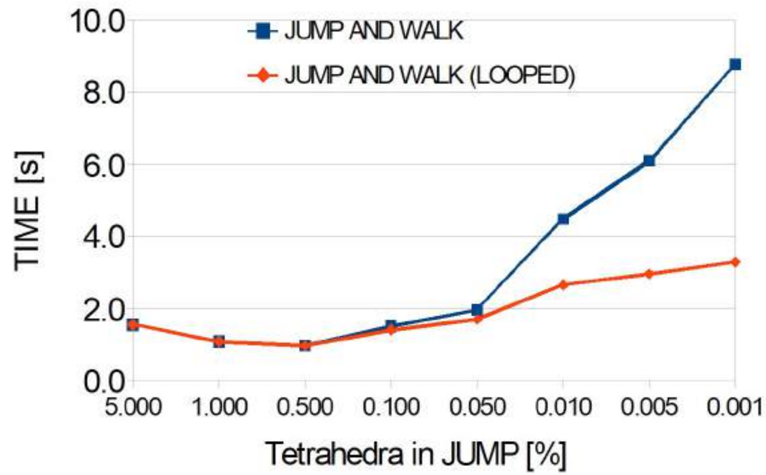


(c) Tetrahedra in Jump = 0.005%

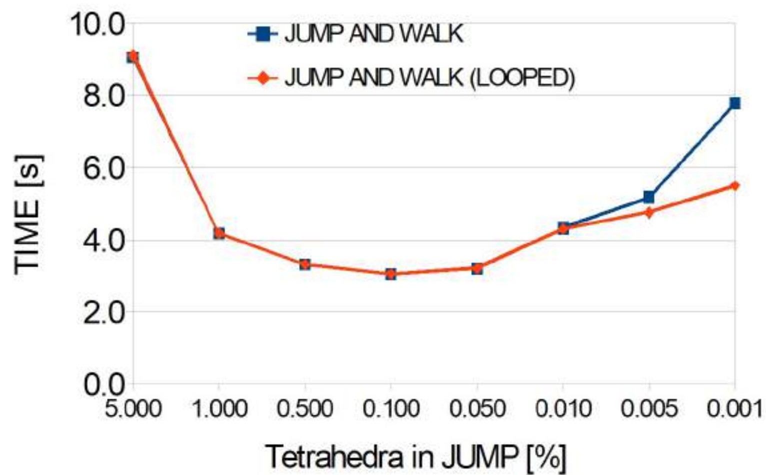


(d) Tetrahedra in Jump = 0.001%

Figure 23: Distances between middle points of starting tetrahedra and and destination points. Randomly selected tetrahedra in a jump phase: $TetaJump = \{0.05\%, 0.01\%, 0.005\%, 0.001\%\}$.



(a) No. of tetrahedra: 51k



(b) No. of tetrahedra: 409k

Figure 24: Times taken by jump-and-walk and looped jump-and-walk algorithms for different setups of tetrahedra used in jump phase. [Structure: 'shaft', walk: stochastic walk with strategy #3.]

Above in this section, the case in which destination points are nodes of a coarse mesh was discussed. Moreover, times presented in Tabs. 6-7 were total times of a walk phase. However, it is very important to present performance for a case when destination points are not nodes of a mesh as well as distinguish what is included in time of a walk phase. Figure 25 and Tabs. 8-9 present detailed times of a walk phase obtained for jump setups that allow finding all destination points ($TetaJums$ not higher than 0.1%). The time of a walk phase is a sum of times taken by the following steps:

1. 'Calc neighbours' - a quest for neighbours of all tetrahedra [2],
2. 'Walk (SW#3)' - a stochastic walk (SW#3),
3. 'After (Teta by Node)' - a verification if the destination point is a node of a tetrahedron and if it is true quest for other tetrahedra which also contain the destination point,
4. 'After (Teta by Neigh.)' - a verification if a destination point lies on a face and if it is true quest for other tetrahedra which also contain the destination point,
5. 'After (Teta unique)' - a creation of unique set of tetrahedra found in 2-4 steps

One may observe that in a case where destination points are nodes of a coarse mesh then the most time-consuming is step 3 (Fig. 25a, Tab. 8). It is due to the time-consuming search (in Teta array) to verify which tetrahedra contain a destination point (node).

If destination points are not nodes[3], then step 3 is inexpensive since there are only 4 checks if a destination point is any of nodes of a tetrahedron (which contains a destination point). It can be seen that time taken by a calculation of neighbours (step 1) cannot be neglected, however, a step (2) is the most time-consuming (Fig. 25b, Tab. 9).

Table 10 presents times taken by jump-and-walk algorithms for two setups of destination points: (A) destination points are nodes of a mesh, (B) destination points are not nodes of a mesh. It can be seen that a walk phase is about 3 times shorter in case destination points are not nodes of a mesh. As a result for the most optimal case (TetaJump=0.1%) a time taken by a jump-and-walk algorithm is 2.6 shorter (setup B over setup A).

[2] This step is performed with build-in Matlab functions: Teta_neighbours = neighbors(triangulation(Teta,Nodes));
[3] Here, destination points are slightly perturbed nodes of a coarse mesh.

Table 8: Times taken by a walk phase in a case that 324 destination points are nodes of a coarse mesh.

TetaJump	5%		1%		0.5%		0.1%	
Calc neighbours [s]	0.24	12%	0.24	9%	0.24	9%	0.24	8%
Walk (SW #3) [s]	0.21	11%	0.36	14%	0.43	17%	0.75	26%
After (Teta by Node) [s]	1.39	71%	1.94	73%	1.78	69%	1.78	62%
After (Teta by Neigh.) [s]	0.09	4%	0.09	3%	0.09	3%	0.09	3%
After (Teta unique) [s]	0.02	1%	0.03	1%	0.03	1%	0.02	1%
Walk (SUM) [s]	1.95	100%	2.66	100%	2.56	100%	2.88	100%

Table 9: Times taken by a walk phase in a case that 324 destination points are not nodes of a coarse mesh

TetaJump	5%		1%		0.5%		0.1%	
Calc neighbours [s]	0.24	39%	0.24	32%	0.24	29%	0.24	22%
Walk (SW #3) [s]	0.28	45%	0.42	54%	0.48	59%	0.78	70%
After (Teta by Node) [s]	0.01	1%	0.01	1%	0.00	0%	0.00	0%
After (Teta by Neigh.) [s]	0.08	12%	0.08	10%	0.07	9%	0.07	6%
After (Teta unique) [s]	0.02	4%	0.02	3%	0.02	2%	0.02	2%
Walk (SUM) [s]	0.63	100%	0.76	100%	0.82	100%	1.11	100%

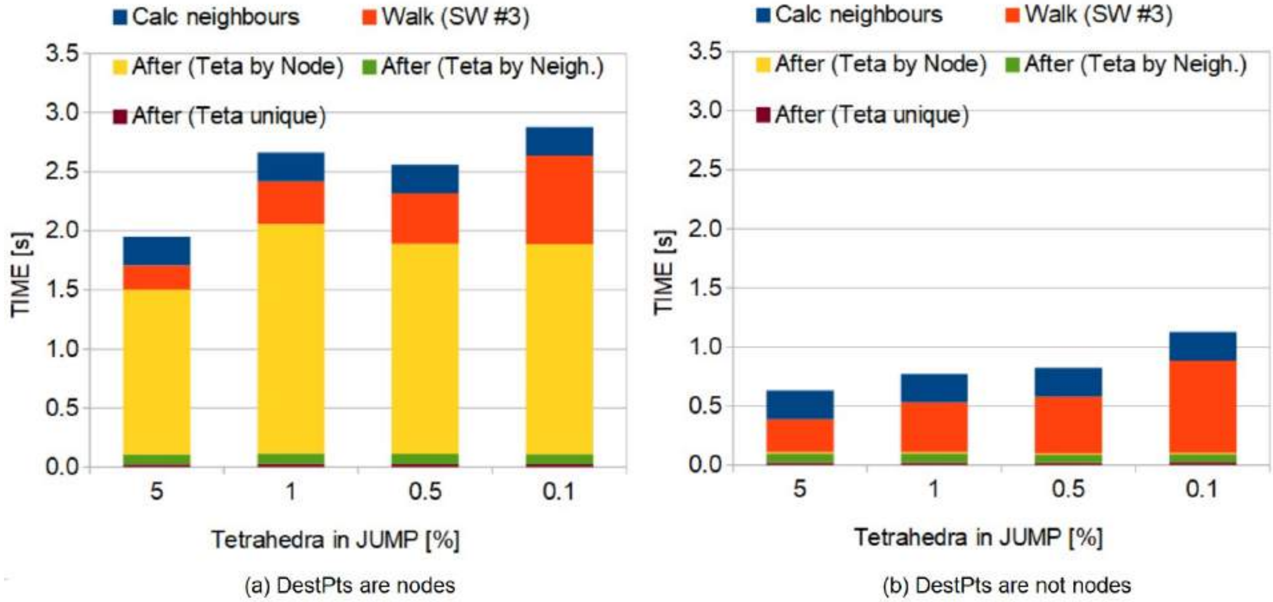


Figure 25: Times taken by steps in a walk phase for different setups of tetrahedra used in a jump phase (DestPts = 324 Destination Points). [Structure: 'shaft' with 409k tetrahedra, walk: stochastic walk with strategy #3.]

Table 10: Times taken by jump-and-walk algorithms for two setups of destination points: (A) 324 destination points are nodes of a mesh, (B) 324 destination points are not nodes of a mesh. (Structure: shaft, 409k tetrahedra)

Phase	TetaJump	5%		1%		0.5%		0.1%	
(A.1)	Jump [s]	7.1	(76%)	1.51	(34%)	0.75	(21%)	0.17	(5%)
(A.2)	Walk [s]	2.19	(24%)	2.9	(66%)	2.8	(79%)	3.12	(95%)
(A.3)	Jump-and-Walk [s]	9.29	(100%)	4.42	(100%)	3.55	(100%)	3.29	(100%)
(B.1)	Jump [s]	7.7	(92%)	1.55	(67%)	0.79	(49%)	0.17	(13%)
(B.2)	Walk [s]	0.63	(8%)	0.76	(33%)	0.82	(51%)	1.11	(87%)
(B.3)	Jump-and-Walk [s]	8.33	(100%)	2.32	(100%)	1.61	(100%)	1.28	(100%)
	(B.1) vs. (A.1)	0.9		1.0		0.9		1.0	
	(B.2) vs. (A.2)	3.5		3.8		3.4		2.8	
	(B.3) vs. (A.3)	1.1		1.9		2.2		2.6	

Table 11: Comparison of a jump-and-walk algorithms with a naive search of 324 tetrahedra (Structure: shaft, 409k tetrahedra). Two setups of destination points: (A) 324 destination points are nodes of a mesh, (B) 324 destination points are not nodes of a mesh. $TetaJump = 0.1\%$.

		(A)	(B)
1	Time naive [s]	489.9	2251.6
2	Jump and Walk [s]	3.3	1.3
2.1	Jump (0.1%) [s]	0.17	0.17
2.2	Walk [s]	3.12	1.11
	Speedup	**149x**	**1755x**

Table 11 presents a comparison of a jump-and-walk algorithm with a naive search of 324 tetrahedra

in a structure 'shaft'. Two setups of destination points are concerned in which 324 destination points are nodes of a mesh (A) and 324 destination points are not nodes of a mesh (B). In both cases a mesh has 409k tetrahedra. A reference (naive) search is a loop over all tetrahedra. If a currently verified tetrahedron contains a destination point, then the loop is not continued, however, additional verification is performed to check if the destination point is a node of a mesh or lies on a face. One may notice that a jump-and-walk algorithm needs several seconds while a reference implementation needs over 8 minutes and 38 minutes to locate 324 destination points for (A) and (B) setups, respectively. The reason why reference times differ is the fact that for a setup (A) 75% destination points lie in tetrahedra which have indices lower than 1000, as opposed to a setup (B) where 80% destination points lie in tetrahedra which have indices higher than 1000. Thus, in the former case the loop over tetrahedra breaks after a significantly lower number of iterations.

7 Conclusions

Walking (visibility and stochastic) algorithms are good for structures with simply connected regions. Better effectiveness of the walk can be achieved thanks to utilization of strategies which helps to run away from looped walks. It was presented that stochastic walk (SW#3) enables to find many (if not all) destination points for structures with not simply connected regions, however, many steps are required. Thus, in order to significantly reduce the number of expensive steps in a walk, it is preferable to support the walk algorithm with an inexpensive verification of a relatively small number of tetrahedra (0.1%) in a jump phase. With an optimal choice of the number of randomly selected tetrahedra the overhead due to a jump phase is relatively inexpensive, and a jump-and-walk approach bear fruits in significant reduction of the time (and steps) required to localize a point in a tetrahedral mesh.

References

[1] `https://sourceforge.net/projects/netgen-mesher/files/netgen-mesher/6.1-experimental/`

[2] Devillers, Olivier, Sylvain Pion, and Monique Teillaud. "Walking in a triangulation." International Journal of Foundations of Computer Science 13.02 (2002): 181-199.

[3] Soukal, Roman. "Walking location algorithms: technical report no. DCSE/TR-2010-03." (2010).

[4] Mücke, Ernst P., Isaac Saias, and Binhai Zhu. "Fast randomized point location without preprocessing in two-and three-dimensional Delaunay triangulations." Computational Geometry 12.1-2 (1999): 63-83.