Marching Cubes Re-implementation

Adrien Saouma

November 2025

Contents

1 Introduction

This project re-implements the marching cubes algorithm, first introduced in 1987 (Lorensen & Cline, 1987[1]). The goal of the algorithm is to extract a 3D isosurface from a sampled scalar field i.e a volumetric grid of intensity values. This surface is an isosurface because all points lying on it have the same intensity value, called the isovalue. Despite the fact that it was initially designed for biomedical visualization such as CT scans and MRI data, it has since become widely used in computer graphics. Indeed, its simplicity and efficiency make it suitable for surface generation in video games and simulation.

Two versions of the algorithm were implemented in this project. The first is a pure Python / Numpy prototype. The second is an optimized Cython based version integrated with `scikit-image.measure.marching_cubes`. The implementations were validated on synthetic volumetric datasets such as spheres, torus and ellipsoids. This enabled comparison between reconstructed meshes and known ground-truth surfaces. I thus used the following standard surface distance metrics: the Hausdorff distance and the average symmetric surface distance (ASSD). The second implementation was further validated on CT scans (Bladder, Bone, Kidney, Liver, and Lung).

A known limitation of the classical marching cubes algorithm is the presence of topological ambiguities. These include inconsistent face connections and interior cavity problems in certain cube configurations. Such ambiguities can lead to topologically incorrect surfaces, motivating the later development of alternative methods such as marching tetrahedra.

2 The Algorithm

This section outlines the full pipeline of the Marching Cubes algorithm. The process begins by decomposing the volume into elementary cubes. The number of cubes depends on the grid resolution: for an input volume sampled on a $N \times M \times P$ grid, the domain contains $(N-1) \times (M-1) \times (P-1)$ cubes in total. After the decomposition, each cube is processed independently following the steps below.

2.1 Cube Indexing Convention

The algorithm relies on a fixed indexing convention for both vertices and edges. Vertices are numbered from 0 to 7, and edges from 0 to 11. This indexing determines the binary ordering of the cube configuration as well as the mapping used by the lookup tables. The convention is shown in Figure 1A).

2.2 Corner Evaluation

For each cube, the algorithm evaluates its eight vertices with respect to the chosen isovalue. Each corner intensity is compared to this threshold: if the

scalar value exceeds the isovalue, the corner is classified as inside the surface and assigned a value of 1. Otherwise, it is assigned 0. Concatenating the eight binary outcomes yields an 8-bit index.

For instance, if only vertex 3 has a value greater than the isovalue, the index would be the following:

$$0000\ 1000.$$

2.3 Intersected Edges

The efficiency of the marching cubes algorithm comes greatly from its reliance on two predefined lookup tables rather than performing any geometric reasoning. Indeed, contrarily to machine learning models that need training, hyperparameter tuning, validation... the marching cubes makes all of its decisions using only the tables. The first one is the `edgeTable` (the second will be introduced in 2.4).

The input to the `edgeTable` is the 8-bit configuration index computed in 2.2. Its output is a 12-bit mask indicating which of the cube's twelve edges are intersected by the isosurface. A bit value of 1 means the isosurface crosses that edge. Using the previous example:

$$\texttt{edgeTable[0000 1000]} = \texttt{1000 0000 1100},$$

which means that edges 2, 3, and 11 are intersected. This is illustrated in Figure 1B).

After determining the intersected edges, the algorithm computes the precise intersection points using linear interpolation:

$$\mathbf{P} = \mathbf{P_1} + \frac{\text{isovalue} - \mathbf{V_1}}{\mathbf{V_2} - \mathbf{V_1}}(\mathbf{P_2} - \mathbf{P_1}),$$

where P_1 and P_2 are the endpoints of the edge, and V_1 and V_2 are the corresponding scalar values. These interpolated points form the vertices of the final triangular facets.

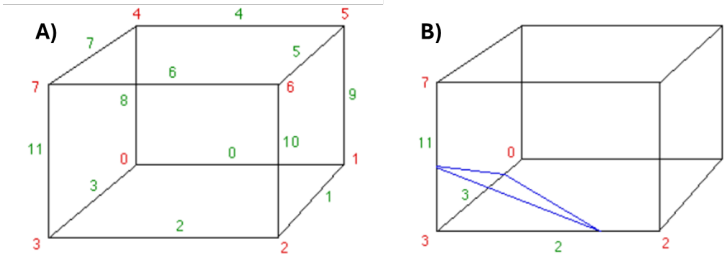

Figure 1: A) Vertex and edge indexing convention used throughout the algorithm. B) Example configuration where only the intensity at vertex 3 is greater than the isovalue.[2]

2.4 Forming the Triangular Faces

Once the intersection points are computed, the algorithm assembles them into triangular facets using the `TriTable`. In theory, each of the eight vertices can be either inside or outside the isosurface, leading to $2^8 = 256$ possible cube configurations.

Lorensen & Cline (1987)[1] showed that these 256 cases can be reduced to only 15 fundamental patterns through two symmetries:

- **Complementary symmetry:** The orientation of the surface does not change if inside and outside labels are swapped. This reduces 256 cases to 128.
- **Rotational symmetry:** Rotating the cube yields equivalent triangulations. Applying this symmetry reduces the remaining 128 cases to 15 patterns.

These 15 patterns (Fig.2) form the basis of the `TriTable`, which lists the correct triangles for every cube configuration.

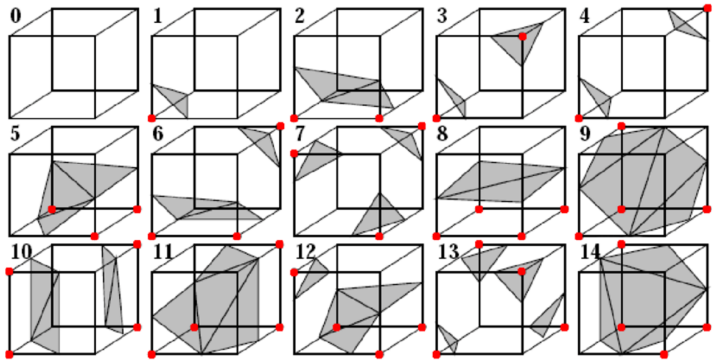

Figure 2: The 15 fundamental triangulation patterns of the Marching Cubes algorithm. [3]

Repeating these four steps for all cubes, you end up with your fully reconstructed isosurface.

3 Metrics

3.1 Hausdorff distance

The Hausdorff distance is a measure of the furthest point from the ground truth, i.e. the worst outlier. It is given by the following formula[4]:

$$d_H(X,Y) := \max\left\{\sup_{x\in X} d(x,Y),\ \sup_{y\in Y} d(X,y)\right\}$$

. With this distance one can analyze whether the implementation suffers from significant deviations at specific locations.

3.2 ASSD

The average symmetric surface distance is used to evaluate how close the marching cubes generated surface is from the ground truth overall. It is given by[5]:

$$\mathrm{ASSD}(S_1, S_2) = \frac{1}{|S_1| + |S_2|} \left(\sum_{x \in S_1} d(x, S_2) + \sum_{y \in S_2} d(y, S_1) \right).$$

This metric provides an overall measure of correspondence by averaging the surface distances in both directions, allowing a more balanced assessment of the reconstruction quality than the Hausdorff distance alone.

4 Implementations

4.1 Python/Numpy

The objective of the first implementation was to follow the full pipeline of the algorithm described in Section 2 and to reproduce it using pure Python and NumPy. I did this prototype first as a validation step, to be sure I understood the algorithm completely, but also as a reference implementation to assess performance limitations before moving toward an optimized version.

Because the implementation does not include any performance-oriented optimizations and has a computational complexity of $O(8n^3)$, the experiments were restricted to synthetic scalar fields such as spheres, tori, and ellipsoids. Another aspect why I chose these datasets was that they provide an analytical ground-truth surfaces, enabling direct quantitative evaluation (See 3).

A key parameter influencing reconstruction accuracy in marching cubes is the grid resolution. Increasing the sampling density of the scalar field gives more information and thus improves the reconstructed surface's fidelity to the ground truth.

Grid Resolution	8	16	32	64	128
ASSD (mm)	3.56	1.75	1.08	0.787	0.640
Hausdorff distance (mm)	11.4	4.91	3.88	3.08	3.60
Number of triangles	240	1104	5 104	19 856	82 928
Time (s)	0.014	0.106	0.783	6.13	47.0

Table 1: Effect of grid resolution on reconstruction accuracy and computational cost. Dataset: synthetic torus.

Table 1 shows that the ASSD decreases consistently as the resolution increases, illustrating convergence toward the analytical surface.

Moreover, the Hausdorff distance decreases initially but stabilizes at around 3 mm. This plateau reflects a limitation of the marching cubes algorithm rather than a sampling issue. Even with high grid resolutions, local geometric inaccuracies lead to persistent outliers.

Moreover, an important aspect to keep in mind is the impact on computational resources. Memory usage grows rapidly because higher resolutions imply a greater number of cubes to evaluate and therefore more triangles to generate. Regarding the execution time, it increases even more due to the implementation being unoptimized.

This prototype therefore demonstrates both the strengths and weaknesses of a direct implementation. While it successfully reconstructs surfaces with high fidelity at sufficiently fine resolutions, the computational and memory demands make it unsuitable for real medical data.

4.2 Scikit-image Implementation

4.2.1 Dataset

The objective of the second implementation was to evaluate the algorithm on real medical data. Thus, segmented CT volumes including the following organs were used: bladder, liver, lungs, kidneys, and skeleton. The dataset comes from the cancer imaging archives[6]. Some of the patients presented hepatic lesions. I applied the algorithm individually to each organ of the dataset in order to fully reconstruct the scans.

The dataset was also used to assess whether the reconstructed surface preserves relevant details such as the lesions on the livers. The algorithm was implemented using *skimage.measure.marching*()*_cubes*[7] on Python. The original paper (Lorensen & Cline, 1987) implemented it on C but the point of the project was not to learn a new programming language. Also the skimage version computes the surface in a reasonable time.

4.2.2 Results

The reconstruction results are shown in Fig. 3. The partial-body dataset produced in 16.38 seconds the detailed surface consisting of 3,886,250 triangles. The corresponding input label volume required 61.50 MB of memory, while the generated mesh representation required 20.04 MB. Also, the hepatic lesions were correctly reconstructed, showing that the algorithm can go in the details.

Regarding the full-body dataset, despite containing a much larger spatial extent, the reconstruction completed in 45.04 seconds and produced 3,107,814 triangles. The input label data required 268.00 MB of memory, whereas the final mesh occupied only 1.22 MB.

These results illustrate two observations:

- **Surface fidelity:** The algorithm can successfully reconstruct large structures while preserving details such as organ boundaries and lesions.

- **Memory characteristics:** Going from the volumetric data to the mesh representation saves a lot of memory. This is particularly evident in full-body reconstruction, where the mesh requires less than 1% of the memory of the input volume.

In addition, it is important to note that the partial-body reconstruction contains more triangles than the full-body reconstruction, even though its input volume is significantly smaller. This difference comes from the fact that the number of triangles generated by the algorithm depends primarily on the surface complexity rather than the size of the volumetric data. The partial-body dataset generates a more complex surface possibly because of the lesions, internal cavities and presence of the bladder. In contrast, the full-body dataset contains primarily skeletal structures with smoother surfaces. This results in fewer edge intersections and therefore fewer triangles. Thus, while the full-body volume requires more memory due to its larger grid, its simpler geometry leads to a lower triangle count.

Overall, this implementation demonstrates that a Cython-optimized Marching Cubes can handle full medical datasets fast while maintaining an accurate visualization of the organs.

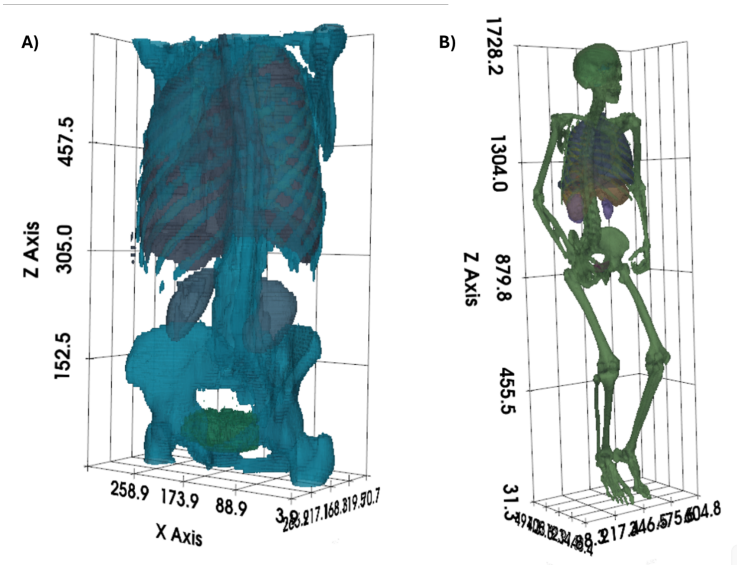

Figure 3: 3D reconstructions from real CT scans using the Marching Cubes implementation. A) Partial-body view showing bones from hips to shoulders, bladder, liver with lesions, and lungs. B) Full-body reconstruction showing the skeletal structure.

5 Discussion: Ambiguities and Solutions

The Marching Cubes algorithm sometimes fail to ensure global surface consistency. This comes from topological ambiguities that appear in specific voxel configurations.

A common example occurs when, on one face of the cube, two diagonally opposite vertices are above the isovalue and the other two are below. In this case, the surface can connect across either diagonal (Fig. 4a). The original algorithm resolves this by choosing only one of the two cases in the `edgeTable`. However, this technique does not work as neighbouring cubes may choose the opposite diagonal, which can create holes and cracks in the final surface (Fig. 4b).

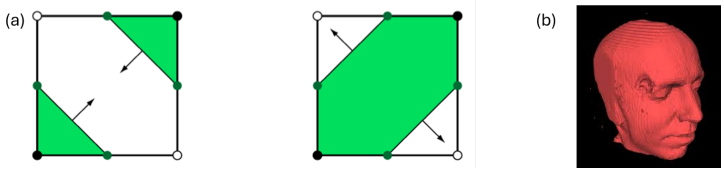

Figure 4: Case of ambiguity in the MC algorithm. a) shows two different possibles results of `edgeTable` for the same input voxels[2] b) shows the impact of these ambiguities in the global visualization.[8]

These ambiguous patterns exist in theory, but none of the medical datasets used in this project produced them. An explanation to this could be that the surfaces are too smooth to be triggering ambiguities.

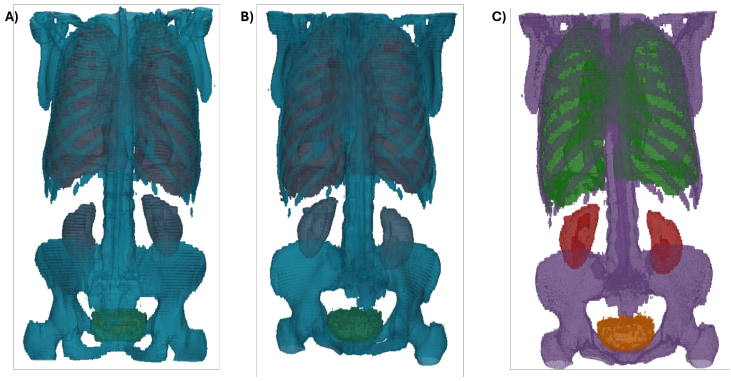

Figure 5: Comparison between three versions of the Marching Cube algorithm on partial body view. A) Original MC by Lorensen B) Improved MC33 by Lewiner C) Marching Tetrahedra

To solve the main issue of the marching cubes, improved versions of the original exists. In this project I will introduce two of them. The first is the **Marching Cube 33** algorithm, which extends the original 15 patterns of the

`edgeTable` to 33. Even if this solves ambiguous cases, this improved version suffers from skinny triangles which lead to poor mesh quality. The second is the **Marching Tetrahedra** (MT) algorithm. It splits each cube into five or six tetrahedrons. As tetrahedral cells cannot produce face ambiguities because they have only four vertices, MT therefore guarantees a consistent surface. Nevertheless, it tends to generate more triangles and is slower due to the larger number of cells.

Figure 5 shows the reconstructions produced by the three methods. The Marching Tetrahedra was implemented using the VTK library and the MC33 using skimage (it is an improved version of the MC33 as I did not find a library implementing the original MC33). The three algorithms generated correct and visually similar surfaces. There are possible local ambiguities in the original MC's reconstructed surface. However, they are not observable and did not expand into holes or cracks. The quantitative comparison is shown in Table 2.

Model	MC original	MT	MC33
Time (s)	17.04	306.61	17.32
Number of triangles	3,886,250	5,288,376	3,902,536

Table 2: Comparison of computation time and number of triangles for the three algorithms on the partial-body CT dataset.

The original MC and the MC33 versions show similar computation times and triangle counts. MC33 produces slightly more triangles but avoids ambiguous configurations. In contrast, MT creates many more triangles and is about 18 times slower. This comes from the tetrahedral subdivision, which increases the number of cases to process.

Overall, choosing the best algorithm is a compromise. As a matter of fact, Although MC33 resolves ambiguities without introducing the heavy computational cost of MT, it can lead to poor quality triangles.

6 Conclusion

This project re-implemented the Marching Cubes algorithm and tested it on synthetic data and real CT scans. The Python/NumPy version confirmed my understanding of the method and showed the limitations of an unoptimized implementation.The optimized scikit-image version showed that the algorithm can handle full medical datasets while preserving details such as liver lesions.

Even though no ambiguity cases appeared in the medical data used here, it remains an important issue of the original Marching cubes. MC33 and Marching Tetrahedra address this issue differently: MC33 removes ambiguities using larger lookup tables, while MT by dividing the cubes into tetrahedrons. On this dataset, both produced similar visual results, but because MT is slower and demands more triangles to fully reconstruct the surface, MC33 comes as the most balanced algorithm out of the three for our specific dataset.

Overall, the project shows that Marching Cubes remains practical when used with optimized libraries, and that the choice between MC, MC33, and MT depends on the required balance between speed and topological reliability. Other, more recent variants, such as the Flying Edge were not covered here but could also be explored in a future project.

References

[1] William E. Lorensen and Harvey E. Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm". In: *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*. Vol. 21. 4. ACM, 1987, pp. 163–169. DOI: 10.1145/37401.37422. URL: https://www.cs.toronto.edu/~jacobson/seminar/lorenson-and-cline-1987.pdf.

[2] Paul Bourke. *Polygonising a Scalar Field — An Algorithm for Creating a Polygonal Surface Representation of an Isosurface*. https://paulbourke.net/geometry/polygonise/. 1997.

[3] Yohan Payán. *The 15 Marching Cubes Patterns (image)*. https://www.researchgate.net/profile/Yohan-Payan/publication/314175176/figure/fig18/AS:668383179718663@1536366376440/The-15-marching-cubes-patterns-Red-points-can-be-inside-or-outside-the-target-isosurface.png. 2017.

[4] *Hausdorff distance*. URL: https://en.wikipedia.org/wiki/Hausdorff_distance.

[5] In: *Journal of Medical Imaging* (). URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC5817231/pdf/JMI-005-015006.pdf.

[6] The Cancer Imaging Archive. *CT-ORG - Collection of CT Abdomen Multiparametric Data*. URL: https://www.cancerimagingarchive.net/collection/ct-org/.

[7] scikit-image developers. *Example: Edges — Marching Cubes (scikit-image 0.25.x)*. URL: https://scikit-image.org/docs/0.25.x/auto_examples/edges/plot_marching_cubes.html.

[8] *Marching Cubes*. URL: https://fr.wikipedia.org/wiki/Marching_cubes.