Marching pentatopes for continuous morphing of isosurfaces from four dimensional data in HTML5/WebGL

A. R. Watters$^1$ 1Center for Computation Biology, Flatiron Institute of the Simons Foundation, New York, New York, USA

Abstract

Animations which show three dimensional volumes continuously changing over time facilitate the exploration and analysis of complex data sets such as calcium image data of neural activity and phase contrast magnetic resonance imaging of blood flows. This paper presents the marching pentatopes method for representing the iso-surfaces of a four dimensional data set as a triangulated surface smoothly deforming as time progresses. The morphing triangulations generated by this method may be rendered using the morph geometry capabilities provided by the three.js javascript library for cross platform HTML5/WebGL presentation in standard web browsers [Cab17].

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms

1. Introduction

Biological data is often four dimensional in raw form. For example phase-contrast magnetic resonance imaging [DBB+15] detects blood flow velocity in three spatial dimensions varying over a sequence of time samples and calcium imaging detects neuron activity in three dimensions varying over time [GKH07]. It is useful to be able to visualize the data directly using morphing 3 dimensional isosurfaces in order to identify characteristics, features, or measurement problems.

This paper presents the marching pentatopes method for deriving smoothly morphing triangulated geometries approximating iso-contours of scalar data $f(x,y,z,t)$ within a grid in a four dimensional volume as implemented in the contourist library [Wat17] for numeric Python [JOP*]. We present the marching pentatopes method as a generalization of the lower dimensional marching triangles method for deriving contour lines and the marching tetrahedra method for deriving iso-surfaces. The contourist package implements all of these methods using a common object oriented architecture designed to be used in conjunction with the three.js javascript library for WebGL [Cab17] which allows the geometric structures generated by contourist to be rendered in web browsers in combination with the features provided by three.js — such as lighting, shadowing, text rendering, various material implementations and interactive controls [Dir13].

2. Preliminaries

The contourist software computes implicit figures for any target value and any grid geometry. This presentation uses a simplified framework without loss of generality for notational convenience. Below we present some terminology and definitions useful in the explanation to follow.

We consider the problem of approximating zero valued iso-contours interpolated within grids limited to coordinates values $N = \{0,1,...,n-1\}$ for some fixed positive integer $n$ where the grid $Nd$ has dimension $d \in \{2,3,4\}$. The contour computation seeks to interpolate a fixed function $f : Nd \rightarrow \mathbb{R}$ approximating a hypothetical point set $\{p \in \mathbb{R}^d | f(p) = 0\}$ of the zeros of $f$ within the limits of the grid $Nd$.

We say a point $p \in Nd$ is a positive if $f(p) \geq 0$ or we say $p$ is a negative if $f(p) < 0$. A set of grid points $S \subset Nd$ crosses zero if $S$ contains at least one negative point and at least one positive point.

If $\{p, p'\} \subset N$ cross zero the interpolated zero $p \circ p'$ is defined as $p \circ p' = \frac{f(p)p' - f(p')p}{f(p) - f(p')}$ $\in \mathbb{R}^d$.

Two grid points $p, p' \in Nd$ are adjacent if $\max(|p_i - p'_i|) = 1$. A sequence of grid points $p_0, p_1, ..., p_m \in Nd$ where each $p_i$ is adjacent to $p_{i+1}$ define a grid path as the union of the line segments connecting each $p_i$ to $p_{i+1}$.

For any finite set of points $S = \{p_0, ..., p_m\} \subset Nd$ we define the convex closure of $S$, $C(S)$, to be the set of points generated by $\sum \alpha_i p_i$ for any set $\alpha_i \in [0,1]$ where $\sum \alpha_i = 1$. If the points $S = \{p_0, ..., p_m\}$ are linearly independent we say that the convex closure $C(S)$ defines a simplex of dimension $m-1$.

The voxel vertices $P(p)$ for a grid point $p \in Nd$ are the set of grid points including $p$ and also including all $p' \in Nd$ adjacent to $p$ where all $p'_i \geq p_i$.

The voxel $V(p) = C(P(p))$ is the convex closure of the voxel vertices $P(p)$ — a square if $d = 2$, a cube if $d = 3$, or a tesseract if $d = 4$.

A contour separating a positive seed point $p \in Nd$ and a nega-
tive seed point \( p' \in \mathbb{N}^d \) is a set of linearly independent point sets \( C = \{c_0, \ldots, c_n\} \) where each \( c_i \in \mathbb{N}^d \) of size \( d \) represents a simple-

In the 2 dimensional case the algorithm is a variant of the marching triangles contour algorithm which is a simplification of the marching squares contour algorithm [HSIW96, Map03]. In the 3 dimensional case the algorithm is the marching cubes contour method [WM97, LC87]. In the 4 dimensional case the algorithm, marching pentatopes, could be viewed as a simpli-

2.1. Other related work

Weigle and Banks [WB96] present a mesh generation approach that can be used for any number of dimensions which is similar to the approach presented here. Their method introduces midpoints in the interpolation step (step 4 below). The use of midpoints results in as many as 4 triangles dividing each crossing tetrahedron in three dimensions and as many as 5 tetrahedra dividing each pentatope in four dimensions. By contrast the methods described below are more parsimonious — producing at most 2 triangles for each crossing tetrahedron in three dimensions and at most 3 tetrahedra for each crossing pentatope in four dimensions. Other methods for interpo-

3. Marching methods for approximating implicit figures

All of the marching methods described here have a similar outline. All basic features of the method are illustrated for the simplest case of Marching Triangles in Figure 2.

3.1. Inputs and Outputs

Inputs: A grid \( \mathbb{N}^d \), a function \( f \) and a pair of seed points \( p, p' \in \mathbb{N}^d \) where \( p \) is negative and \( p' \) is positive.

Outputs: A contour \( C \) separating \( p \) from \( p' \) in \( \mathbb{N}^d \) and a post-proces-

3.2. Method outline

1: Locate initial crossing voxels. Use binary search between \( p \) and \( p' \) to find a grid point \( p_0 \) where \( V(p_0) \) crosses zero.

2: Find all adjacent crossing voxels. Find the smallest set of
grid points \( G = \{ p_i \} \) such that \( p_0 \in G \) and for any \( p, q \in N^d \) if \( p \in G \) and \( q \) is adjacent to \( p \) and \( V(q) \) crosses 0, then \( q \in G \).

3: Tile the crossing voxels into crossing simplices of dimension \( d \). For example for \( d = 3 \) each crossing cube is tiled into six tetrahedra. Collect the generated simplices that cross zero into the set \( T \).

4: Separate positive from negative vertices of simplices in \( T \). For each simplex \( s \in T \) find simplices of dimension \( d - 1 \) with vertices that interpolate the vertices if \( s \) separating the positive from the negative vertices in \( s \). The generated collection \( C \) of simplices of dimension \( d - 1 \) is the desired contour.

5: Post processing. Translate the contour \( C \) into a structure suitable for rendering.

Steps 1 and 2 are applications of the well known binary search [Ben75] and transitive closure [War75] techniques. Please see the contourist source code for additional details of the implementation. The steps 3, 4, and 5 vary between the cases \( d = 2, d = 3, \) and \( d = 4 \), with each case discussed in its own subsection below.

3.3. Marching triangles specialized steps

This section sketches the specialized steps 3, 4, and 5 for the marching triangles method where dimension \( d = 2 \).

Marching triangles step 3: tile crossing squares as 2 triangles. We divide each crossing square \( V(p) \) into two triangles (simplices of dimension 2). Of the generated triangles we select those that cross zero as the set \( T \).

Marching triangles step 4: Separate positive and negative vertices in crossing triangles with interpolated line segments. For each crossing triangle of the tile set \( T \) we find one line segment between interpolated points lying on crossing edges. We collect the interpolated line segments generated as the contour \( C \).

Marching triangles step 5: Assembling contour paths from line segments. Finally in order to render the contour curve in an efficient and appropriate manner using graphics libraries we must join connecting line segments into continuous paths.

![Figure 2: Summary diagram for the marching triangles method with dimension \( d = 2 \) and grid size \( n = 4 \). Here the contour approximates \( f(x, y) = y - 0.4x^2 - 0.2 = 0 \) in the grid \( \{0, 1, 2, 3\}^2 = N^d \). The seed line segment between the seed points \((1, 0)\) and \((2, 3)\) is in dark gray (Step 1). Crossing squares are in pink (Step 2). Crossing triangles tiling the squares are in light gray (Step 3). Positive vertices are in red. Line segment interpolation points are in black. Green line segments joining interpolation points separate positive from negative vertices in the crossing triangles (Step 4). Contour approximation line segments assemble into a path starting at the large yellow point and ending at the large green point (Step 5).](image1)

![Figure 3: Crossing cubes are tiled into six tetrahedra in marching tetrahedra method step 3. Each of the tetrahedra is associated with one of the permutations of "xyz".](image2)

![Figure 4: For a tetrahedron with two positive vertices \{a, b\} and two negative vertices \{c, d\} separate positive and negative vertices using two triangles with vertices at \{a \oplus d, b \oplus c, a \oplus c\} and \{a \oplus d, b \oplus c, b \oplus d\} in marching tetrahedra step 4.](image3)

3.4. Marching tetrahedra specialized steps

This section explains the specialized steps 3, 4, and 5 for the marching tetrahedra method where dimension \( d = 3 \).

Marching tetrahedra step 3: tile crossing cubes using 6 tetrahedra. We divide each crossing cube \( V(p) \) into six tetrahedra by assigning to each of the six permutations of "xyz" a tetrahedron where the dimension quantity order is defined by the permutation. For example for the permutation "zyx" we associate the tetrahedron \( \{(p_0 + x, p_1 + y, p_2 + z) \mid x, y, z \in [0, 1] \text{ and } z \leq y \leq x\} \).

The cube tiling is illustrated in Figure 3. The tetrahedral tiles that cross zero are collected as the tile set \( T \).
Marching tetrahedra step 4: Separate positive and negative vertices in each crossing tetrahedron using one or two triangles. For each crossing tetrahedron of the tile set \( T \) we determine the positive vertices and the negative vertices of the tetrahedron. For a tetrahedron with one positive vertex (or symmetrically one negative vertex) \( a \) and negative vertices \( b, c, d \), separate \( a \) from the negative points using a triangle with vertices at \( \{a \circ b, a \circ c, a \circ d\} \). Otherwise there are two positive and two negative vertices in the crossing tetrahedron: separate the positive and negative vertices using two triangles as shown in Figure 4. The set of triangles generated define the contour triangulation \( C \).

Marching tetrahedra step 5: Orient the triangles of the contour to be counterclockwise when viewed from the "outside".

In order to make sure that the surface normals are computed consistently for proper lighting interactions in \( \text{three.js} \) the triangle vertices must be provided in anti-clockwise order when viewed from the "outside" of the volume [Muk12]. Please consult the \text{contourist} source code for the implementation details of this operation.

**Figure 5:** The 24 pentatopes tiling a tesseract intersected with the hyperplane \( t = 0.4 \) from marching pentatopes step 3.

<table>
<thead>
<tr>
<th><img src="image1.png" alt="Diagram" /></th>
</tr>
</thead>
</table>

**Figure 6:** For a pentatope with two positive vertices \( \{a, b\} \), here projected into 3 dimensions, separate the positive vertices from the negative vertices using three tetrahedra with vertices \( \{a \circ e, a \circ c, a \circ d, b \circ e\} \), \( \{b \circ d, a \circ c, a \circ d, b \circ e\} \), \( \{b \circ e, a \circ c, b \circ d, b \circ e\} \). in marching pentatopes step 4.

### 3.5. Marching pentatopes specialized steps

This section explains the specialized steps 3, 4, and 5 for the marching pentatopes method where dimension \( d = 4 \).

**Marching pentatopes step 3: tile crossing tesseracts using 24 pentatopes.** We divide a crossing tesseract \( V(p) \) into 24 pentatopes by assigning to each of the 24 permutations of "xyzt" a pentatope where the dimension quantity order is defined by the permutation, illustrated in Figure 5. For example for the permutation "zytx" we associate the pentatope

\[
\{(p_0 + x, p_1 + y, p_2 + z, p_3 + t) \mid x, y, z, t \in [0,1] \text{ and } z \leq y \leq t \leq x\}
\]

The tiling pentatopes which cross zero define the tile set \( T \).

**Marching pentatopes step 4: Separate positive and negative vertices in each crossing pentatope using one or three tetrahedra.** For each crossing pentatope of the tile set \( T \) we separate the positive vertices and the negative vertices of the pentatope using interpolated tetrahedra. For a pentatope with one positive vertex (or symmetrically one negative vertex) \( a \) and negative vertices \( b, c, d, e \) separate \( a \) from the negative vertices using one tetrahedron with vertices \( \{a \circ b, a \circ c, a \circ d, a \circ e\} \). Otherwise there are two positive and three negative vertices (or symmetrically two negative and three positive vertices) in the crossing pentatope. In that case separate the positive and negative vertices using three tetrahedra as shown in Figure 6. The tetrahedra generated define the contour \( C \).

**Marching pentatopes step 5: Convert the contour tetrahedra into morphing triangles.** The \text{three.js} library implements morphing using "morphing triangles" where each vertex of the triangularization is associated with a three dimensional start position at a start time and a three dimensional end position at an end time. Translate \( C \) into morphing triangles using the following procedure for each tetrahedron in \( \tau \in \mathbb{C} \)

Sort the \( t \) values of the vertices of \( \tau \) with up to 4 unique values \( t_0, t_1, \ldots \). For each \( t \) interval between the vertex \( t \) values \( t_i, t_{i+1} \) compute the intersection \( \tau_{ij} \) of \( \tau \) with the hyperplane at the midpoint \( t = (t_i + t_{i+1})/2 \). The intersection either forms a triangle or a tetrahedron in three dimensions and each of the vertices of the intersection lies on a 4 dimensional line segment between two vertices of \( \tau \). If the intersection forms a triangle then generate a single morphing triangle using the vertices of \( \tau \) that correspond to the vertices of \( \tau_{ij} \) as the start and end positions of the morph. If the intersection \( \tau_{ij} \) forms a tetrahedron then generate two morphing triangles using the vertices of \( \tau \) that correspond to the vertices of \( \tau_{ij} \) as the start and end positions of the morph in a manner similar to Figure 4. The resulting set of morphing triangles are suitable for rendering using the \text{three.js} library, which uses WebGL and the GPU to render the morphs when available.

Above the choice of slicing in the \( t \) dimension is arbitrary and may be replaced with \( x, y \) or \( z \) as desired.

**Acknowledgements:** Tarmo Āijō pointed out many issues, errors, and possible improvements in this paper and other members of the Bonneau Laboratory for systems biology at New York University offered helpful comments and suggestions.

© 2017 The Author(s)

References


