

# Morse-Smale segmentation of protein alpha-shapes

Sëma

## Abstract

Morse-Smale (MS) complex is a useful tool for molecular surface segmentation. We extend the for Morse-Smale construction and simplification algorithm for the case of protein alpha-shapes. Our software is freely available at <http://gila-fw.bioengr.uic.edu/~sema/alms>

## 1 Introduction

With constantly growing number of resolved protein structures there is an increasing importance of automated tools for the protein molecule shape analysis. Segmentation of the surface into regions that identify characteristic features is useful both for visual and computational studies of the molecule. It is essential for finding protein functional sites, predicting protein-protein interactions and evolutionary studies. While many methods for various surfaces segmentation have been proposed [8], the technique based on MS complex can be especially useful for segmentation of molecular surfaces [6].

## 2 Algorithm

We modify the algorithm described in [1]. Wherever *clustering* is mentioned in further text, we use a simple and fast union-find algorithm that can be found, for example, in the first chapter of [7].

### 2.1 Preliminary steps

We construct weighted Delaunay triangulation by incremental method, similar to the one described in [5], but using history DAG [3] for locating tetrahedra. The DAG is initialized by a tetrahedron of four *dummy* vertices that are located far enough to contain all other vertexes inside this tetrahedron. Our algorithm uses inexact (double precision) arithmetics for most of the computation, while in degenerate cases it switches to exact (integer) arithmetics and applies the Simulation of Simplicity (SoS) method [4] to resolve the degeneracy.

Weighted alpha-shape is built upon the triangulation data structure as described in [2]. While in general, alpha-shape is a family of polytopes depending

on parameter  $\alpha$ , in this application we only consider the case of  $\alpha = 0$ . Constructing alpha-shape classifies all simplexes in triangulation as *solid* (the ones that belong to alpha-complex), and *empty* (all others).

## 2.2 Constructing manifold

We run clustering algorithm on all empty tetrahedra in the triangulation data structure. If two empty tetrahedra have a common empty face they are clustered together. Any tetrahedron with at least one *dummy* vertex is *outside*, and so are all other tetrahedra in its cluster. All other empty tetrahedra represent voids that cannot be reached from the outside of the molecule. Solid triangular faces adjacent to *outside* tetrahedra are also classified *outside* and used for the manifold assembling. Each triangle in triangulation can have two, one, or none outside faces.

Once all outside faces are found, triangulated manifold is assembled by the following algorithm:

```
for each edge  $e$  of each outside face  $f$ :
  if  $f$  is not yet connected with another face at  $e$ :
    find the next outside face  $g$  adjacent to  $e$ ;
    connect faces  $f$  and  $g$  at  $e$ ;
  end if
end for
```

To traverse faces around edge  $e$ , we use the underlying triangulation structure that was built in previous step. Sometimes face  $g$  is an opposite side of face  $f$ .

Alpha shape may also contain single edges, not attached to faces, and single vertexes, but single vertexes never appear in alpha-shapes built for biomolecules, and single edges are rare if solvent accessibility values are used as atom radii. Both single edges and single vertexes are ignored in current implementation.

## 2.3 Vertex classification

We assume that no two vertices have equal function value. Degenerate cases can be resolved, for example, by comparing the vertex indexes. For every vertex  $\nu$  in the mesh triangulation, its *star*  $St(\nu)$  is defined as the collection of simplices containing  $\nu$ , and the *link*  $Lk(\nu)$  is the boundary of  $St(\nu)$ . The *lower link*  $LLk(\nu)$  is the subset of  $Lk(\nu)$  with the function values below that of vertex  $\nu$ . Vertex  $\nu$  is a *maximum* if its lower link is its entire link and a *minimum* if its lower link is empty. In all other cases the lower link consists of  $k \geq 1$  connected pieces. A vertex is *regular* if  $k = 1$ , *saddle* if  $k = 2$ , or *multiple saddle* if  $k > 2$ .

Multiple saddle  $s$  can be recursively split into the collection of simple saddles by the following procedure:

```
while the multiplicity of  $s$  is larger than 2 repeat:
```

```

find a single component  $C$  of its lower link;
find vertices  $a$  and  $b$  neighboring  $C$  in the link of  $s$ ;
create a new vertex  $u$  with  $f(u) = f(s)$ ;
delete all edges incident to  $C$  from  $s$  and connect them to  $u$ ;
add new edges  $u \rightarrow a$ ,  $u \rightarrow b$ , and  $s \rightarrow u$ ;
end while

```

Index for the new vertex  $u$  should be assigned in such a way, that a tie-breaking rule would consider the edge  $s \rightarrow u$  ascending. Each step creates a new saddle vertex  $u$  and decreases the multiplicity of saddle  $s$  by one. Before proceeding to the next part of the algorithm all vertices are classified into 4 types: *regular*, *minima*, *maxima*, *saddles*, but no *multiple saddles*.

## 2.4 Tracing descending paths

Descending paths start at saddles and end in local minima. Descending paths can merge, but not allowed to split. Computing descending paths is straightforward: first, mark all saddles and local minima as *descending* to acknowledge that they belong to some descending paths. Then, for each lower link of each saddle  $s$  search for the steepest descending edge  $s \rightarrow u$ . Mark the edge *descending*. If  $u$  is a local minimum, it is the end of the path. If  $u$  was already marked *descending*, the descending path is merged with the previous one. Otherwise, mark  $u$  *descending*, find the steepest descending edge beginning at this vertex, and continue the cycle until the path merges or reaches the local minimum.

A descending path hitting another saddle can pass it by either side. However, by convention, we assume that descending paths always keep the saddles they hit on the right side by their downward direction.

## 2.5 Tracing ascending paths

Ascending paths start at saddles and go to local maxima. Same as descending paths, ascending path can merge, but cannot split. In addition, ascending path cannot cross with existing descending paths. The last condition makes ascending path tracking a bit more tricky. Ascending edges may be marked *R-ascending* (default) or *L-ascending*, depending on which side they can merge with existing paths. For each upper link of each saddle  $s$  search for the steepest ascending edge  $s \rightarrow u$ . This edge starts a new path that by default is signed as *R-ascending*. Mark the edge with the same sign as the path. If  $u$  is a local maximum, it is the end of the path. If  $u$  was already marked with the same sign as the path, the path is merging with the previous one. If  $u$  does not lie on *descending* path, find the steepest ascending edge beginning at this vertex, sign the path *R-ascending*, and continue the cycle. Otherwise, the edge belongs to the part of the *star*  $St(u)$  bounded by *descending* edges. Find the steepest ascending edge in this part of the star. If the path approaches *descending*-marked edge from the left side, sign the path *L-ascending*, otherwise sign it *R-ascending*. Continue the cycle until the path merges or reaches the local maximum.

## 2.6 Ordering arcs

It's convenient to store the path information in the *arc* object that contains the references to both end points, and to the edge that starts the path from the saddle. Arc can be traced from the saddle point using one of the tracing algorithms described above. For convenience, we keep two reversely directed arcs for each descending and ascending path. Each arc *A* also keeps a pointer *A*->*Rev* to the reversely directed arc, and a pointer *A*->*Nxt* to the next side of the MS quadrangle. To assemble the MS complex from the set of loose paths, these pointers must be properly assigned.

The order of arcs around saddles coincides with the order of their path starting edges. The order of arcs around minima can be found by depth-first traversal of descendent edges around each minimum. For descendent paths following through saddles, we should use an earlier mentioned convention that paths always pass saddles by the same side. Path traversal around maxima is not necessary. Given the *Nxt*-pointers correctly assigned at minima and saddles, missing pointers can be easily found:

```
for each arc A starting at any maximum:
  A->Nxt->Nxt->Nxt->Nxt = A
end for
```

## 2.7 Simplification

Cancellation is the basic operation in simplifying MS complex. The extremum-saddle pair  $u, v$  is removed by merging four regions into two and extending all arcs ending at  $u$ . Some cancellations may be not valid, like, for example, cancellation of maximum-saddle pair where saddle is connected to the same maximum twice. Given an error metric it is easy to progressively simplify a MS complex, by greedy selection of valid cancellations with the smallest errors until the desired level of simplification is reached.

Instead of deleting arcs and critical points one can assign error values to them, that would mean that arc or critical point becomes inactive if its error is smaller than error threshold chosen for simplification. Thus, the whole family of simplified MS complexes can be retrieved from the same data structure for different error thresholds.

We use a *superarc* structure to run the simplification algorithm. Superarc starts at saddle, ends at minimum or maximum, and contains one or more arcs. There is a direct one-to-one correspondence between the arcs starting at saddles and the superarcs. Superarc object holds the references to its both ends and to its starting arc. A linked list structure is used to quickly find all superarcs that end at a given extremum.

Prior to simplification, all errors of arcs and critical points are set to the values larger than the largest possible error. Superarcs are initialized by corresponding arcs and arranged into a priority queue that returns superarc with the

smallest error. The following algorithm then produces simplification:

```

while priority que is not empty:
  pop superarc  $s \rightsquigarrow w$ ;
  if  $s \rightsquigarrow u$  is not valid cancellation: go to next superarc;
  set  $\varepsilon$  to the error of  $s \rightsquigarrow u$ ;
  if  $u$  is maximum:
    find another maximum  $w$  connected to  $s$  by superarc;
    find superarcs  $s \rightsquigarrow a$  and  $s \rightsquigarrow b$  going to minima;
  else ( $u$  is minimum):
    find another minimum  $w$  connected to  $s$  by superarc;
    find superarcs  $s \rightsquigarrow a$  and  $s \rightsquigarrow b$  going to maxima;
  end if
  deactivate  $u$  (set its error to  $\varepsilon$ );
  deactivate  $w$  (set its error to  $\varepsilon$ );
  deactivate superarc  $s \rightsquigarrow a$  (set the arc error to  $\varepsilon$ );
  deactivate superarc  $s \rightsquigarrow b$  (set the arc error to  $\varepsilon$ );
  for all superarcs ending at  $u$  (except of  $s \rightsquigarrow u$ ):
    set their end point to  $w$ ;
    recalculate their errors;
    update their positions in the priority que;
  end for
end while

```

Maintaining the lists of arcs within each superarc can be cumbersome and expensive, however, it is not necessary. To deactivate a superarc it is enough to deactivate only its first arc, rather than update them all.

## 2.8 Retrieving superarcs

Given a threshold value for the error, active superarcs can be found by depth-first search that starts at active minimum or maximum, and is allowed to traverse only active arcs and inactive critical points on its way to the active saddles. The search returns valid superarcs in correct order. Superarcs have the same order around the saddles as their corresponding arcs.

## 2.9 Patch assembling

Final step of algorithm is finding all vertices that lie inside each MS region. Due to high degeneracy, flood-fill procedure offered in [1] is not the method of choice for this case. MS region may consist of multiple disjoint patches of triangular mesh, connected only by singular boundary edges; boundary can pass through the same edge several times; edge can belong to more than two boundaries. The following method handles all these issues:

```

for each MS region  $R$ :

```

```

initialise an empty set  $S$  of edges;
for each edge  $a \rightarrow b$  on the patch boundary (going clockwise):
    if there is an edge  $b \rightarrow a$  in  $S$ : remove this edge from  $S$ ,
    otherwise: add  $a \rightarrow b$  into  $S$ .
end for
for each edge remaining in  $S$ :
    mark the triangle on its right side as belonging to  $R$ .
end for
end for
cluster the triangles (adjacent triangles are connected
if the edge between them does not belong to any boundary)

```

If any triangle in cluster belongs to region  $R$ , the whole cluster belongs to this region. Vertex belongs to region  $R$  if it belongs to a triangle in such a cluster, or if it lies on the region boundary. There can be some MS regions without boundaries, like, for instance, a sphere with minimum and maximum, but with no saddles. These regions correspond to clusters that are not associated with any boundaries.

### 3 Software

Our software is freely available at <http://gila-fw.bioengr.uic.edu/~sema/alms>. It computes Morse-Smale partition for the molecule's alpha-shape, and performs its simplification to the desired level of details.

Usage: `alms <parameter> <input>`

with one of the following *parameters*:

- e *float* - specifies the cancellation threshold;
- n *int* - specifies the number of pairs to cancel;
- p *float* - specifies the percent of pairs to cancel.

Options -e, -n, -p are mutually exclusive. Default *parameter* is -p 50.

The *input* should be a space- or tab- delimited text file with each line defining the parameters for one atom in the following format:

*Id(int)* *X(float)* *Y(float)* *Z(float)* *R(float)* *F(float)*

Here *Id* is the reference number for the atom to be used in the output; *X*, *Y*, *Z* are the atom coordinates, *R* is the atom radius (it's recommended to use the sum of atomic radius and the radius of solvent molecule to obtain smoother alpha-shapes); *F* is the value of the function of interest for the atom.

The program outputs MS complex as set of patches. For each patch it outputs its critical points: minimum, maximum, and two saddles (if the patch has those). After the critical points it outputs the patch boundary (if it exists) as an ordered sequence of atoms. In complicated cases same boundary can pass through the same atom more than one time. Finally, it outputs unordered list of all atoms belonging to the patch. Boundary atoms (including critical points) can belong to several patches simultaneously.

Users willing to obtain the neighborhoods around the function minima, max-

ima, or saddles, should then assemble together the patches sharing the same minimum, maximum, or saddle point.

## References

- [1] Peer-Timo Bremer and Valerio Pascucci. A practical approach to two-dimensional scalar topology. In Helwig Hauser, Hans Hagen, and Holger Theisel, editors, *Topology-based Methods in Visualization*, pages 151–169. Springer, 2007.
- [2] Herbert Edelsbrunner. Weighted alpha shapes. Technical report, Champaign, IL, USA, 1992.
- [3] Herbert Edelsbrunner. *Geometry and topology for mesh generation*. Cambridge University Press, New York, NY, USA, 2001.
- [4] Herbert Edelsbrunner and Ernst Peter Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9(1):66–104, 1990.
- [5] Ernst Peter Mücke. *Shapes and implementations in three-dimensional geometry*. PhD thesis, Champaign, IL, USA, 1994.
- [6] Vijay Natarajan, Yusu Wang, Peer-Timo Bremer, Valerio Pascucci, and Bernd Hamann. Segmenting molecular surfaces. *Comput. Aided Geom. Des.*, 23(6):495–509, 2006.
- [7] Robert Sedgwick and Christopher J. Van Wyk. *Algorithms in C++: Fundamentals, Data Structures, Sorting, Searching*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.
- [8] Ariel Shamir. A formulation of boundary mesh segmentation. In *3DPVT '04: Proceedings of the 3D Data Processing, Visualization, and Transmission, 2nd International Symposium on (3DPVT'04)*, pages 82–89, Washington, DC, USA, 2004. IEEE Computer Society.