

## **Příloha č. 5**

### **Implementace algoritmů vhodných pro kartografickou generalizaci SMD**



## Úvod

---

Na základě požadavku budoucího uživatele byly cizojazyčné texty stažené z webových stránek a z odborných publikací ponechány v anglickém jazyce a v původním grafickém layoutu.

## 1. Systém GRASS

[http://grasswiki.osgeo.org/wiki/V.generalize\\_tutorial](http://grasswiki.osgeo.org/wiki/V.generalize_tutorial)

### NAME

**v.generalize** - Vector based generalization.

### KEYWORDS

vector, generalization, simplification, smoothing, displacement, network generalization

### SYNOPSIS

**v.generalize**

**v.generalize help**

**v.generalize** [-cr] *input=name output=name* [*type=string[,string,...]*] *method=string*  
*threshold=float* [*look\_ahead=integer*] [*reduction=float*] [*slide=float*]  
[*angle\_thresh=float*] [*degree\_thresh=integer*] [*closeness\_thresh=float*]  
[*betweenness\_thresh=float*] [*alpha=float*] [*beta=float*] [*iterations=integer*]  
[*layer=integer*] [*cats=range*] [*where=sql\_query*] [--overwrite] [--verbose] [--quiet]

#### Flags:

**-c**

Copy attributes

**-r**

This does nothing. It is retained for backwards compatibility

**--overwrite**

Allow output files to overwrite existing files

**--verbose**

Verbose module output

**--quiet**

Quiet module output

#### Parameters:

**input=name**

Name of input vector map

**output=name**

Name for output vector map

**type=string[,string,...]**

Feature type

Options: *line,boundary,area*

Default: *line,boundary,area*

**method=string**

Generalization algorithm

Options:

*douglas,douglas\_reduction,lang,reduction,reumann,boyle,sliding\_averaging,di  
stance\_weighting,chaiken,hermite,snakes,network,displacement*

**douglas:** Douglas-Peucker Algorithm

**douglas\_reduction**: Douglas-Peucker Algorithm with reduction parameter  
**lang**: Lang Simplification Algorithm  
**reduction**: Vertex Reduction Algorithm eliminates points close to each other  
**reumann**: Reumann-Witkam Algorithm  
**boyle**: Boyle's Forward-Looking Algorithm  
**sliding\_averaging**: McMaster's Sliding Averaging Algorithm  
**distance\_weighting**: McMaster's Distance-Weighting Algorithm  
**chaiken**: Chaiken's Algorithm  
**hermite**: Interpolation by Cubic Hermite Splines  
**snakes**: Snakes method for line smoothing  
**network**: Network generalization  
**displacement**: Displacement of lines close to each other

**threshold**=*float*  
Maximal tolerance value  
Options: 0-1000000000

**look\_ahead**=*integer*  
Look-ahead parameter  
Default: 7

**reduction**=*float*  
Percentage of the points in the output of 'douglas\_reduction' algorithm  
Options: 0-100  
Default: 50

**slide**=*float*  
Slide of computed point toward the original point  
Options: 0-1  
Default: 0.5

**angle\_thresh**=*float*  
Minimum angle between two consecutive segments in Hermite method  
Options: 0-180  
Default: 3

**degree\_thresh**=*integer*  
Degree threshold in network generalization  
Default: 0

**closeness\_thresh**=*float*  
Closeness threshold in network generalization  
Options: 0-1  
Default: 0

**betweeness\_thresh**=*float*  
Betweeness threshold in network generalization  
Default: 0

**alpha**=*float*  
Snakes alpha parameter  
Default: 1.0

**beta**=*float*  
Snakes beta parameter  
Default: 1.0

**iterations**=*integer*  
Number of iterations  
Default: 1

**layer**=*integer*

Layer number

A single vector map can be connected to multiple database tables. This number determines which table to use.

Default: 1

**cats**=*range*

Category values

Example: 1,3,7-9,13

**where**=*sql\_query*

WHERE conditions of SQL statement without 'where' keyword

Example: income < 1000 and inhab >= 10000

## DESCRIPTION

*v.generalize* is a module for the generalization of GRASS vector maps. This module consists of algorithms for line simplification, line smoothing, network generalization and displacement (new methods may be added later). For more examples and nice pictures, see [tutorial](#)

If *type=area* is selected, boundaries of selected areas will be generalized, and the options *cats*, *where*, and *layer* will be used to select areas.

## NOTES

(Line) simplification is a process of reducing the complexity of vector features. The module transforms a line into another line consisting of fewer vertices, that still approximate the original line. Most of the algorithms described below select a subset of points on the original line.

(Line) smoothing is a "reverse" process which takes as input a line and produces a smoother approximate of the original. In some cases, this is achieved by inserting new vertices into the original line, and can total up to 4000% of the number of vertices in the original. In such an instance, it is always a good idea to simplify the line after smoothing.

Smoothing and simplification algorithms implemented in this module work line by line, i.e. simplification/smoothing of one line does not affect the other lines; they are treated separately. Also, the first and the last point of each line is never translated and/or deleted.

## SIMPLIFICATION

*v.generalize* contains following line simplification algorithms:

- Douglas-Peucker Algorithm
- Douglas-Peucker Reduction Algorithm
- Lang Algorithm
- Vertex Reduction
- Reumann-Witkam Algorithm
- Remove Small Lines/Areas

Different algorithms require different parameters, but all the algorithms have one parameter in common: the **threshold** parameter, given in map units (for latitude-longitude locations: in decimal degree). In general, the degree of simplification increases with the increasing value of **threshold**.

## ALGORITHM DESCRIPTIONS

- *Douglas-Peucker* - "Quicksort" of line simplification, the most widely used algorithm. Input parameters: **input**, **threshold**. For more information, see: [http://geometryalgorithms.com/Archive/algorithm\\_0205/algorithm\\_0205.htm](http://geometryalgorithms.com/Archive/algorithm_0205/algorithm_0205.htm).
- *Douglas-Peucker Reduction Algorithm* is essentially the same algorithm as the algorithm above, the difference being that it takes an additional **reduction** parameter which denotes the percentage of the number of points on the new line with respect to the number of points on the original line. Input parameters: **input**, **threshold**, **reduction**.
- *Lang* - Another standard algorithm. Input parameters: **input**, **threshold**, **look\_ahead**. For an excellent description, see: <http://www.sli.unimelb.edu.au/gisweb/LGmodule/LGLangVisualisation.htm>.
- *Vertex Reduction* - Simplest among the algorithms. Input parameters: **input**, **threshold**. Given a line, this algorithm removes the points of this line which are closer to each other than **threshold**. More precisely, if p1 and p2 are two consecutive points, and the distance between p2 and p1 is less than **threshold**, it removes p2 and repeats the same process on the remaining points.
- *Reuman-Witkam* - Input parameters: **input**, **threshold**. This algorithm quite reasonably preserves the global characteristics of the lines. For more information, see: [http://www.ifp.uni-stuttgart.de/lehre/vorlesungen/GIS1/Lernmodule/Lg/LG\\_de\\_6.html](http://www.ifp.uni-stuttgart.de/lehre/vorlesungen/GIS1/Lernmodule/Lg/LG_de_6.html) (german).

*Douglas-Peucker* and *Douglas-Peucker Reduction Algorithm* use the same method to simplify the lines. Note that

```
v.generalize input=boundary_county output=boundary_county_dp20
method=douglas threshold=20
```

is equivalent to

```
v.generalize input=boundary_county
output=boundary_county_dp_red20_100 \
    method=douglas_reduction threshold=20 reduction=100
```

However, in this case, the first method is faster. Also observe that *douglas\_reduction* never outputs more vertices than *douglas*, and that, in general, *douglas* is more efficient than *douglas\_reduction*. More importantly, the effect of

```
v.generalize input=boundary_county output=boundary_county_dp_red0_30 \
    method=douglas_reduction threshold=0 reduction=30
```

is that 'out' contains approximately only 30% of points of 'in'.

## SMOOTHING

The following smoothing algorithms are implemented in *v.generalize*:

- *Boyle's Forward-Looking Algorithm* - The position of each point depends on the position of the previous points and the point **look\_ahead** ahead. **look\_ahead** consecutive points. Input parameters: **input**, **look\_ahead**.
- *McMaster's Sliding Averaging Algorithm* - Input Parameters: **input**, **slide**, **look\_ahead**. The new position of each point is the average of the **look\_ahead** points around. Parameter **slide** is used for linear interpolation between old and new position (see below).
- *McMaster's Distance-Weighting Algorithm* - Takes the weighted average of **look\_ahead** consecutive points where the weight is the reciprocal of the distance from the point to the currently smoothed point. The parameter **slide** is used for linear interpolation between the original position of the point and newly computed position where value 0 means the original position. Input parameters: **input**, **slide**, **look\_ahead**.
- *Chaiken's Algorithm* - "Inscribes" a line touching the original line such that the points on this new line are at least *threshold* apart. Input parameters: **input**, **threshold**. This algorithm approximates the given line very well.
- *Hermite Interpolation* - This algorithm takes the points of the given line as the control points of hermite cubic spline and approximates this spline by the points approximately **threshold** apart. This method has excellent results for small values of **threshold**, but in this case it produces a huge number of new points and some simplification is usually needed. Input parameters: **input**, **threshold**, **angle\_thresh**. **Angle\_thresh** is used for reducing the number of the points. It denotes the minimal angle (in degrees) between two consecutive segments of a line.
- *Snakes* is the method of minimisation of the "energy" of a line. This method preserves the general characteristics of the lines but smooths the "sharp corners" of a line. Input parameters **input**, **alpha**, **beta**. This algorithm works very well for small values of **alpha** and **beta** (between 0 and 5). These parameters affect the "sharpness" and the curvature of the computed line.

One of the key advantages of *Hermite Interpolation* is the fact that the computed line always passes through the points of the original line, whereas the lines produced by the remaining algorithms never pass through these points. In some sense, this algorithm outputs a line which "circumscribes" the input line.

On the other hand, *Chaiken's Algorithm* outputs a line which "inscribes" a given line. The output line always touches/intersects the centre of the input line segment between two consecutive points. For more iterations, the property above does not hold, but the computed lines are very similar to the Bezier Splines. The disadvantage of the two algorithms given above is that they increase the number of points. However, *Hermite Interpolation* can be used as another simplification algorithm. To achieve this, it is necessary to set *angle\_thresh* to higher values (15 or so).

One restriction on both McMasters' Algorithms is that *look\_ahead* parameter must be odd. Also note that these algorithms have no effect if *look\_ahead* = 1.

Note that *Boyle's*, *McMasters'* and *Snakes* algorithm are sometimes used in the signal processing to smooth the signals. More importantly, these algorithms never change the number of points on the lines; they only translate the points, and do not insert any new points.



*Snakes* Algorithm is (asymptotically) the slowest among the algorithms presented above. Also, it requires quite a lot of memory. This means that it is not very efficient for maps with the lines consisting of many segments.

#### DISPLACEMENT

The displacement is used when the lines overlap and/or are close to each other at the current level of detail. In general, displacement methods move the conflicting features apart so that they do not interact and can be distinguished.

This module implements an algorithm for displacement of linear features based on the *Snakes* approach. This method generally yields very good results; however, it requires a lot of memory and is not very efficient.

Displacement is selected by **method=displacement**. It uses the following parameters:

- **threshold** - specifies critical distance. Two features interact if they are closer than **threshold** apart.
- **alpha, beta** - These parameters define the rigidity of lines. For larger values of **alpha, beta** ( $\geq 1$ ), the algorithm does a better job at retaining the original shape of the lines, possibly at the expense of displacement distance. If the values of **alpha, beta** are too small ( $\leq 0.001$ ), then the lines are moved sufficiently, but the geometry and topology of lines can be destroyed. Most likely the best way to find the good values of **alpha, beta** is by trial and error.
- **iterations** - denotes the number of iterations the interactions between the lines are resolved. Good starting points for values of **iterations** are between 10 and 100.

The lines affected by the algorithm can be specified by the **layer, cats** and **where** parameters.

#### NETWORK GENERALIZATION

Used for selecting "the most important" part of the network. This is based on the graph algorithms. Network generalization is applied if **method=network**. The algorithm calculates three centrality measures for each line in the network and only the lines with the values greater than thresholds are selected. The behaviour of algorithm can be altered by the following parameters:

- **degree\_thresh** - algorithm selects only the lines which share a point with at least **degree\_thresh** different lines.
- **closeness\_thresh** - is always in the range (0, 1]. Only the lines with the closeness centrality value at least **closeness\_thresh** apart are selected. The lines in the centre of a network have greater values of this measure than the lines near the border of a network. This means that this parameter can be used for selecting the centre(s) of a network. Note that if **closeness\_thresh=0** then everything is selected.
- **betweenness\_thresh** - Again, only the lines with a betweenness centrality measure at least **betweenness\_thresh** are selected. This value is always positive and is larger for large networks. It denotes to what extent a line is in between the other lines in the network. This value is large for the lines which

lie between other lines and lie on the paths between two parts of a network. In the terminology of road networks, these are highways, bypasses, main roads/streets, etc.

All three parameters above can be presented at the same time. In that case, the algorithm selects only the lines which meet each criterion.

Also, the outputted network may not be connected if the value of **betweenness\_thresh** is too large.

## EXAMPLES

### Simplification

Simplification of county boundaries with DP method (North Carolina sample dataset):

```
v.generalize input=boundary_county output=boundary_county_dp20 \
  method=douglas threshold=20
```

### Smoothing

Smoothing of road network with Chaiken method (North Carolina sample dataset):

```
v.generalize input=roads output=roads_chaiken method=chaiken \
  threshold=1
```

## SEE ALSO

[v.clean](#), [v.dissolve](#)

[v.generalize Tutorial](#) (GRASS-Wiki)

## AUTHORS

Daniel Bundala, Google Summer of Code 2007, Student  
Wolf Bergenheim, Mentor  
Fixes: Markus Metz

*Last changed: \$Date: 2015-02-09 07:09:02 -0800 (Mon, 09 Feb 2015) \$*

---

[Main index](#) - [vector index](#) - [Full index](#)

© 2003-2015 [GRASS Development Team](#)

## 2 CGAL Computed Geometry Algorithm Library

<http://www.cgal.org/>

### 2.1 Uživatelské licence

#### The LGPL

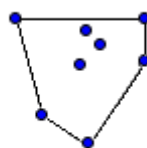
The [Lesser General Public License](#) (v3+) gives you the right to use and copy the code freely. It is also possible to modify the code under the condition that the resulting modification is released as source code under the LGPL with any binary distribution of your software that uses these LGPL parts.

#### The GPL

The [GPL](#) (v3+) is an Open Source license that, gives you the right to use, copy and modify the code freely. If you distribute your software based on GPLed CGAL data structures, you are obliged to distribute the modifications of CGAL you made, and you are furthermore obliged to distribute the source code of your own software under the GPL.

### 2.2 Convex Hull Algorithms

#### 2.2.1 2D Convex Hulls and Extreme Points



*Susan Hert and Stefan Schirra*

This package provides functions for computing convex hulls in two dimensions as well as functions for checking if sets of points are strongly convex or not. There are also a number of functions described for computing particular extreme points and subsequences of hull points, such as the lower and upper hull of a set of points.

[User Manual](#) [Reference Manual](#)

--

### 2.3 Polygons

#### 2.3.1 2D Polygons



*Geert-Jan Giezeman and Wieger Wesselink*

This package provides a 2D polygon class and operations on sequences of points, like bounding box, extremal points, signed area, simplicity and convexity test, orientation, and point location. The demo includes operations on polygons, such as computing a convex partition, and the straight skeleton.

[User Manual](#) [Reference Manual](#)

--

### 2.3.2 2D Polygon Partitioning



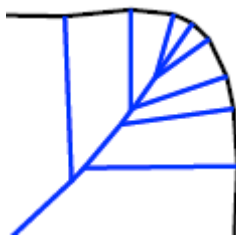
*Susan Hert*

This package provides functions for partitioning polygons in monotone or convex polygons. The algorithms can produce results with the minimal number of polygons, as well as approximations which have no more than four times the optimal number of convex pieces but they differ in their runtime complexities.

[User Manual](#) [Reference Manual](#)

--

### 2.3.3 2D Straight Skeleton and Polygon Offsetting



*Fernando Cacciola*

This package implements an algorithm to construct a halfedge data structure representing the straight skeleton in the interior of 2D polygons with holes and an algorithm to construct inward offset polygons at any offset distance given a straight skeleton.

[User Manual](#) [Reference Manual](#)

--

### 2.3.4 2D Minkowski Sums



*Ron Wein, Alon Baram, Eyal Flato, Efi Fogel, Michael Hemmer, Sebastian Morr*

This package consists of functions that compute the Minkowski sum of two simple straight-edge polygons in the plane. It also contains functions for computing the Minkowski sum of a polygon and a disc, an operation known as *offsetting* or *dilating* a polygon. The package can compute the exact representation of the offset polygon, or provide a guaranteed approximation of the offset.

[User Manual](#) [Reference Manual](#)

--

### 2.3.5 2D Polyline Simplification



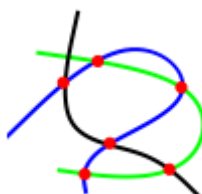
*Andreas Fabri*

This package enables to simplify polylines with the guarantee that the topology of the polylines does not change. This can be done for a single polyline as well as for a set of polyline constraints in a constrained triangulation. The simplification can be controlled with cost and stop functions.

[User Manual](#) [Reference Manual](#)

--

### 2.4 2D Intersection of Curves



*Baruch Zukerman, Ron Wein, and Efi Fogel*

This package provides three free functions implemented based on the sweep-line paradigm: given a collection of input curves, compute all intersection points, compute the set of subcurves that are

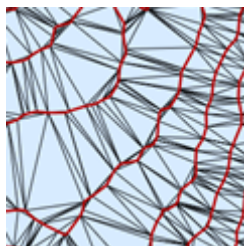
pairwise interior-disjoint induced by them, and check whether there is at least one pair of curves among them that intersect in their interior.

[User Manual](#) [Reference Manual](#)

--

## 2.5 Triangulations and Delaunay Triangulations

### 2.5.1 2D Triangulation



*Mariette Yvinec*

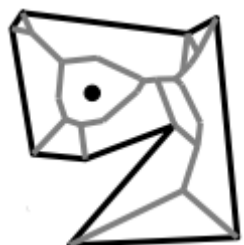
This package allows to build and handle various triangulations for point sets two dimensions. Any CGAL triangulation covers the convex hull of its vertices. Triangulations are built incrementally and can be modified by insertion or removal of vertices. They offer point location facilities. The package provides plain triangulation (whose faces depend on the insertion order of the vertices) and Delaunay triangulations. Regular triangulations are also provided for sets of weighted points. Delaunay and regular triangulations offer nearest neighbor queries and primitives to build the dual Voronoi and power diagrams. Finally, constrained and Delaunay constrained triangulations allows to force some constrained segments to appear as edges of the triangulation. Several versions of constrained and Delaunay constrained triangulations are provided: some of them handle intersections between input constraints segment while others do not.

[User Manual](#) [Reference Manual](#)

--

## 2.6 Voronoi Diagrams

### 2.6.1 2D Segment Delaunay Graphs



*Menelaos Karavelas*

An algorithm for computing the dual of a Voronoi diagram of a set of segments under the Euclidean metric. It is a generalization of the standard Voronoi diagram for points. The algorithms provided are dynamic.

## [User Manu](#)

--

### 2.6.2 L Infinity Segment Delaunay Graphs



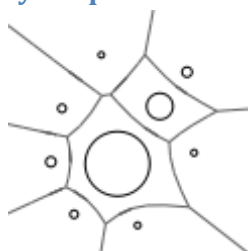
*Panagiotis Cheilaris, Sandeep Kumar Dey, Evanthia Papadopoulou*

Algorithms and geometric traits for computing the dual of the Voronoi diagram of a set of points and segments under the  $L^\infty$  metric.

[User Manual](#) [Reference Manual](#)

--

### 2.6.3 2D Apollonius Graphs (Delaunay Graphs of Disks)



*Menelaos Karavelas and Mariette Yvinec*

Algorithms for computing the Apollonius graph in two dimensions. The Apollonius graph is the dual of the Apollonius diagram, also known as the *additively weighted Voronoi diagram*. The latter can be thought of as the Voronoi diagram of a set of disks under the Euclidean metric, and it is a generalization of the standard Voronoi diagram for points. The algorithms provided are dynamic.

[User Manual](#) [Reference Manual](#)

--

### 2.6.4 2D Voronoi Diagram Adaptor



*Menelaos Karavelas*

The 2D Voronoi diagram adaptor package provides an adaptor that adapts a 2-dimensional triangulated Delaunay graph to the corresponding Voronoi diagram, represented as a doubly connected edge list (DCEL) data structure. The adaptor has the ability to automatically eliminate, in a consistent manner, degenerate features of the Voronoi diagram, that are artifacts of the requirement that Delaunay graphs should be triangulated even in degenerate configurations. Depending on the type of operations that the underlying Delaunay graph supports, the adaptor allows for the incremental or dynamic construction of Voronoi diagrams and can support point location queries.

[User Manual](#) [Reference Manual](#)

--

## 2.7 Mesh Generation

### 2.7.1 2D Conforming Triangulations and Meshes



*Laurent Rineau*

This package implements a Delaunay refinement algorithm to construct conforming triangulations and 2D meshes. Conforming Delaunay triangulations are obtained from constrained Delaunay triangulations by refining constrained edges until they are Delaunay edges. Conforming Gabriel triangulations are obtained by further refining constrained edges until they become Gabriel edges. The package provides also a 2D mesh generator that refines triangles and constrained edges until user defined size and shape criteria on triangles are satisfied. The generated meshes can be optimized using the Lloyd algorithm, also provided in this package. The package can handle intersecting input constraints and set no restriction on the angle formed by two constraints sharing an endpoint.

[User Manual](#) [Reference Manual](#)

--

### 2.7.2 Point Set Processing





*Pierre Alliez, Clément Jamin, Quentin Mérigot, Jocelyn Meyron, Laurent Saboret, Nader Salman, Shihao Wu*

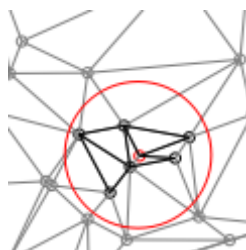
This CGAL component implements methods to analyze and process unorganized point sets. The input is an unorganized point set, possibly with normal attributes (unoriented or oriented). The point set can be analyzed to measure its average spacing, and processed through functions devoted to the simplification, outlier removal, smoothing, normal estimation, normal orientation and feature edges estimation.

[User Manual](#) [Reference Manual](#)

--

## 2.8 Spatial Searching and Sorting

### 2.8.1 2D Range and Neighbor Search



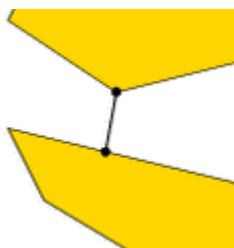
*Matthias Bäsken*

This package supports circular, triangular, and isorectangular range search queries as well as (k) nearest neighbor search queries on 2D point sets. In contrast to the spatial searching package, this package uses a Delaunay triangulation as underlying data structure.

[User Manual](#) [Reference Manual](#)

--

### 2.8.2 Optimal Distances



*Kaspar Fischer, Bernd Gärtner, Thomas Herrmann, Michael Hoffmann, and Sven Schönherr*

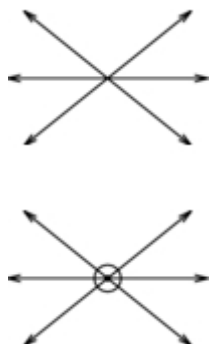
This package provides algorithms for computing the distance between the convex hulls of two point sets in d-dimensional space, without explicitly constructing the convex hulls. It further provides an algorithm to compute the width of a point set, and the furthest point for each vertex of a convex polygon.

[User Manual](#) [Reference Manual](#)

### 3 Stony Brook Repository

[http://www3.cs.stonybrook.edu/~algorith/major\\_section/1.6.shtml](http://www3.cs.stonybrook.edu/~algorith/major_section/1.6.shtml)

#### 3.1 1.6 Computational Geometry



[Robust Geometric Primitives](#)



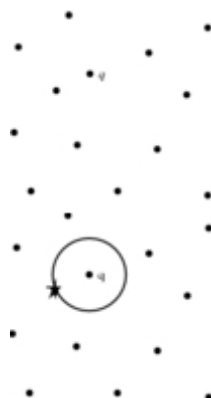
[Convex Hull](#)



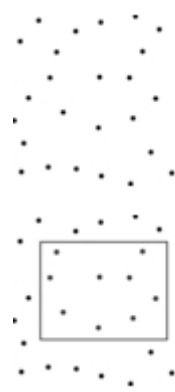
[Triangulation](#)



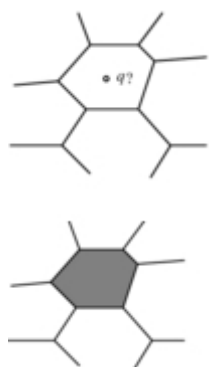
[Voronoi Diagrams](#)



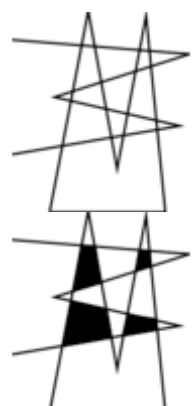
[Nearest Neighbor Search](#)



[Range Search](#)



[Point Location](#)



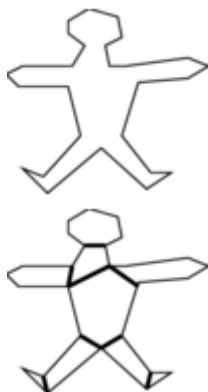
[Intersection Detection](#)



[Bin Packing](#)



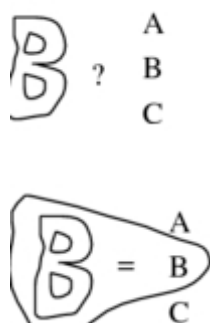
[Medial-Axis Transformation](#)



[Polygon Partitioning](#)



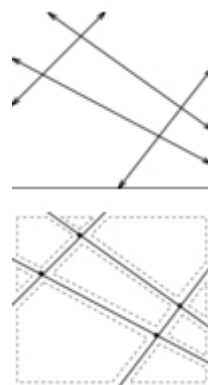
[Simplifying Polygons](#)



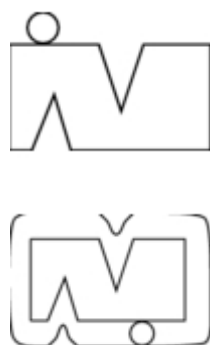
[Shape Similarity](#)



[Motion Planning](#)



[Maintaining Line Arrangements](#)



[Minkowski Sum](#)