Computing the Probabilities of Operations in Vector Models for Uncertain Spatial Data

Erlend Tøssebro
Department of Electrical and Computer Engineering, University of Stavanger, NO-4036 Stavanger, Norway
erlend.tossebro@uis.no

Mads Nygård
Department of Computer and Information Science, Norwegian University of Science and Technology, NO-7491 Trondheim, Norway
mads@idi.ntnu.no

Abstract

Many papers on modeling uncertain or fuzzy spatial data are based on probability functions or fuzzy set functions. However, few of them specify how to actually compute these in a real system. This paper presents a way to use Delaunay triangulations to compute the values of such functions for uncertain spatial objects represented in a vector model. The paper also contains an algorithm that uses this Delaunay triangulation to compute the probability that two uncertain regions intersect.

1. Introduction

Many models for uncertain or vague geographical data, such as the ones presented in [2], [12], [18], assume that objects are defined using some form of probability function or fuzzy set membership function. For raster models, such a function might be implemented by storing a fuzzy set value for each raster cell. However, it is more difficult for vector models.

In a vector model of spatial data, point objects are represented by a coordinate pair. Line objects are represented as a series of connected straight line segments. Area objects (faces) are represented by their border lines. To model spatial objects with uncertain position, we presented an approach in which it is assumed that an uncertain object is known to be inside a given area in [15]. This area is called the **support**. This term comes from fuzzy set theory where the support is the set of points in the universe whose fuzzy set membership is above 0. For faces, there may be areas that are known to be part of the face. These areas are called the **core**. If nothing is known for certain, the core will be empty. This way of modeling faces is similar to egg-yolk models [3], [6]. A similar model for uncertain lines is given in [5].

The main advantages of a vector model compared to a raster model are that they take less storage space, they scale better for display at varying resolutions, and you can represent topology, using methods such as the one in [17]. These are among the reasons why vector models are usually preferred for storing discrete geographical objects.

We also presented a method for computing the probabilities that various operations are true in [15]. This method is based on the distance from the point in question to the core and to the edge of the support. For uncertain points a point of maximum probability is defined. For uncertain lines a central line of maximum probability is defined. The uncertain types from [15] are shown in Figure 1.

However, the way of computing the probability function in that paper is not suitable to generate iso-lines of probability. As described in Sections 2 and 6, such iso-lines can be used to compute the probabilities of many spatial predicates such as whether two regions overlap. The problem is that while the method from [15] can compute the probability value in a given point, there is no easy way to calculate which points have a given probability value. The iso-lines may also be curved even if the bordering line segments are straight. The last property could easily lead to the following type of inconsistency: When computing the function value for the point p, the result is 0.51. When computing the iso-line with a function value of 0.5, the point p is outside this line (that is, it is grouped with the points that have values below 0.5).

The goal of this paper is therefore to develop a new method for computing such probability functions or fuzzy set membership functions for a vector model, as well as to develop some algorithms that use this function to compute some predicates for uncertain

geographical objects. The method assumes that a one-dimensional function is used to compute the probabilities because such functions are much easier to handle both for those entering the data and for the computer. If the data provider does not know the exact probability distribution, using a simple one-dimensional function such as the linear function is much easier that trying to create a two-dimensional function. Models like the fuzzy set model for soil boundaries from [8] already use a one-dimensional function for this.

Such functions can also be used to estimate the probability that two faces overlap each other or the probability that object *A* is inside face *B*.

In the next section we describe related work as well as the model of spatial objects that the rest of this paper is based on. In Section 3 we describe our method for creating a triangulation of the uncertain object. The beginning of this section discusses triangulations. Section 4 describes our basic algorithm and how it works on uncertain faces. Uncertain lines and points are a straightforward extrapolation from faces and are not treated in detail due to space considerations. Section 5 briefly describes our implementation of the method described in this paper. Section 6 shows an example operation that uses the triangulation to compute the probability that two uncertain regions overlap. Section 7 shows how much additional storage space our method uses compared to a basic spaghetti model. Section 8 concludes the paper and describes how this approach can be extended for use with fuzzy points and regions.

2. Related work

In [12] and [21], models for fuzzy regions based on infinite point sets are described. Such models are called “abstract models” in the rest of this paper. These models also contain means of computing topological relationships between fuzzy regions. This means, for instance, that they can compute the fuzzy overlap between two regions. This may be interpreted as the probability that the two regions overlap.

Such fuzzy topological relationships are computed by checking a set of alpha-cut values. An alpha-cut of a fuzzy set is a crisp set containing all elements that are in the fuzzy set with membership value above a given alpha.

These alpha-cut values must be computed somehow, either using some type of function or by explicitly storing the alpha-cut regions. Storing the alpha-cuts takes a lot of space, so if storage is at a premium, the alpha-cuts should be computed when needed. The method presented in this paper is well suited for such computations. It can compute an alpha-cut in linear time as long as the required triangulation is stored with the object.

In [2], a way of computing topological relationships between fuzzy regions that include those that deal with the border is presented. He computes the border as a separate fuzzy set using the fuzzy set values from the region and computes the various relationships by taking the appropriate intersections and checking the maximum fuzzy set value in each result. For instance, to check whether two regions share a border, one needs to check the intersection of their boundaries and the intersection of their interiors. If the first is large and the second is small, it is likely that they share a border rather than overlap.

None of the three papers presented so far give any means of computing the fuzzy set values on which they base themselves.

Three algorithms for computing the probability functions of uncertain objects are described in [18]. All three have the basic property that they use one-dimensional functions to compute the fuzzy set values for two-dimensional objects. The fuzzy set value of a point depended on the distance from the expected boundary and the fuzziness of the boundary in [8]. In the same manner, the methods for computing probability [18] depend on the distance to the core and the distance to the outside border.

The first method in [18] uses the distance from the core and the distance from the edge. This algorithm has the advantage over the second method from that paper that it can be used for objects with supports and cores of any shape. However, it cannot be used for

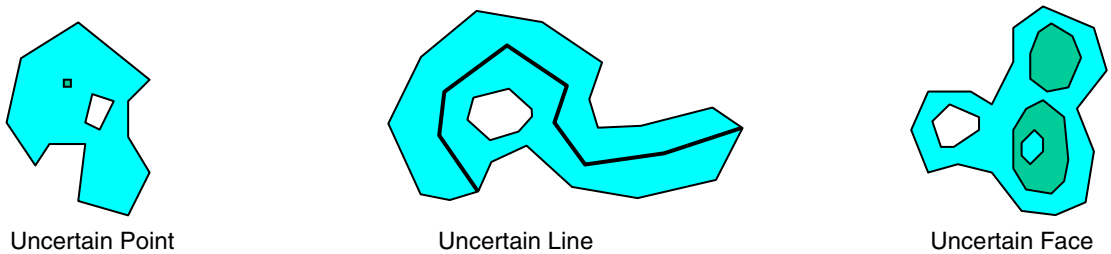

Figure 1: Uncertain Spatial Objects

computing iso-lines of probability because there is no easy way to compute all points with a given probability value [15]. This also means that to get an accurate result for a spatial predicate, all possible points have to be computed individually, which is infeasible in practice.

The second method uses the interpolation algorithm from [13] to generate a triangulation and uses that to compute the probability function. Each corner of the triangulation is assigned a value of 1.0 or 0.0 depending on whether it is on the core boundary or the support boundary, respectively. The probability can then be computed with linear interpolation inside the triangle. This produces consistent results with a reasonable amount of computation because iso-lines only need to be computed along the edges of the triangles. However, the interpolation algorithm in [13] can only be used for objects for which the core and support have similar shapes and for which there are no holes in the support that are not shared by the core.

The third method is an extension of the second in which the user manually inserts the extra points that are needed in the triangulation in order to make it work for an arbitrary object. The problem with this is that it requires some work by the user, and a fairly experienced user.

This paper presents a new method for automatically generating such a triangulation that is also capable of automatically generating the necessary points to make it work for arbitrary objects.

Another approach [16] solves the probability computation problem by simplifying the model and thereby limiting its expressiveness. That model is capable of computing probability values reliably, but it cannot represent certain types of objects that our earlier model [15] can. For instance, it cannot represent holes in the support like the one in the example in Figure 2. Despite this, the two models consume roughly equal amounts of space.

In [9], a method for indexing multidimensional uncertain points is introduced. Their method is based on a grid with probability values in each point of the grid and linear interpolation. Linear interpolations in a square area do not necessarily produce straight surfaces and it is therefore difficult to know which points have a particular probability value. This makes producing alpha-cuts difficult.

The papers [3], [10], [13] deal with spatial uncertainty in point objects resulting from the uncertain movement of objects and present more specific solutions for this problem.

3. Triangulating uncertain objects

For digital terrain models, the Delaunay triangulation is often used. The Delaunay triangulation is a triangulation that maximizes the smallest angle in the triangulation. The Delaunay triangulation can be computed reasonably fast (O(n*log(n))) and produces a triangulation with few long, thin triangles. Long, thin triangles are not desirable because then the interpolation may use points far from the point of interest. Points are more relevant to the interpolation the closer they are to the point of interest. Thus the Delaunay triangulation ensures a reasonably accurate interpolation.

A Delaunay triangulation can be created for the uncertain border of any spatial object by using the end points of the line segments as the points from which to create the triangulation. For this triangulation to be valid, it must contain the actual lines that make up the boundary and the core of the object. If it does not, it contains lines that cross this boundary, and such lines cause errors in the interpolation. To ensure that it contains all the lines, a constrained Delaunay triangulation algorithm, such as the one in [7], may be used. A constrained Delaunay triangulation is one in which it is required that the triangulation contains a certain set of line segments. In this paper this set of line segments is the boundaries of the core and support. Making a constrained Delaunay triangulation takes the same asymptotic time as making an ordinary one [7].

After this triangulation is computed, there is one important problem that needs to be solved. There may be lines that go through the support yet both ends are on the outer boundary. Such a line will have a probability value of 0 along its entire length because it has a value of 0 at both ends. Because it goes through the support, it ought to have a probability value above 0 in its interior. To remove this error, a new point may be inserted in the middle of this line. The probability value of this point may be computed using another method such as comparing the distance to the outside border and the distance to the core. As long as the point is inside the support, this will yield a probability value between 0.0 and 1.0. This point is then added to the triangulation using a dynamic Delaunay triangulation algorithm.

A similar problem occurs for lines that go from one point along the core to another but through the boundary area. The solution here is the same, except if the points are inside an uncertain hole, because then the distance to the outer edge is meaningless. However, uncertain holes have a given probability of existing, and that can be used as the probability value of these points.

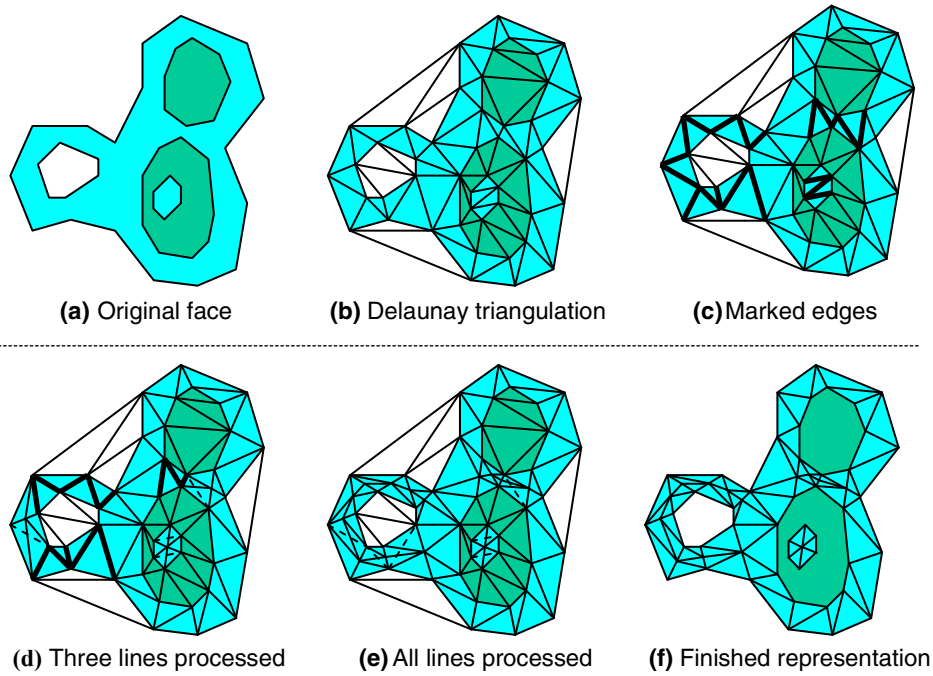

Figure 2. Triangulation of an uncertain face

Triangular Irregular Networks have been shown to produce consistent results for linear interpolations [19]. However, the probability function is not necessarily linear. The input of the function is the relative distance to the core and the outer support. Non-linear functions are not a problem because one can use the triangulation to generate this input rather than the output. The relative distance from the core and the outer edges is always linearly interpolated. As the function is assumed to be strictly rising, and the same value *in* always yields the same value *out*, this ensures consistency for any legal function.

4. Basic algorithm and uncertain face example

Our algorithm for creating the triangulation of an uncertain face *f* is shown below. In this algorithm, a **Point** is a crisp point stored with a pair of coordinate values. A **Line Segment** is a straight line stored with its two end points. A **Cycle** is a set of connected line segments forming a closed loop. A **Face** is a connected area represented with one outer cycle and a set of hole cycles. A **Region** is a disjoint set of faces. An **Uncertain Face** is stored with a *Face* as support and a *Region* as core. A **Graph** is a set of vertices (*points*) and a set of edges (*line segments*) between them.

This algorithm is illustrated in Figure 2. Each part of Figure 2 shows the result at a particular stage of the algorithm. This is indicated by letters in parentheses referring to the letter of the part that represents the result at that particular point. The algorithms in this paper use indentation to indicate the level of nesting. So a for-loop for instance ends when the indentation is decreased rather than with an explicit "end" statement.

Algorithm Triangulate_Face
(a) **Input**: The Uncertain Face UF
Output: A triangulation of the support of the face
Method:
Let *P* be an empty set of points
Let *C* be an empty set of cycles
Let *E* be an empty set of line segments
Let *Sf* be the support of *UF*
Let *Cr* be the core of *UF*
Let *Dg* be an empty graph
// Part 1: Generating the list of cycles and list of points, and the probability values of the points
For each cycle *c* that is part of the boundary of *Sf* **do**
$C := C \cup \{c\}$
Let *value*(c) = 0
For each $f \in Cr$ do
For each cycle *c* that is part of the boundary of *f* **do**
$C := C \cup \{c\}$
Let *value*(c) = 1
For each $c \in C$ do
For each $l \in C$ **do**
P := P ∪ {l.start}

```
            Let value(p) = value(c)
// Part 2: Creating the initial triangulation
(b)     Dg := Constrained_Delaunay_Triang(P, C)
// Part 3: Finding edges with constant value
        For each s ∈ Edges(Dg) do
            If value(s.start) = value(s.end) and
               s overlaps (interior(Sf)) and
               s overlaps (exterior(Cr)) do
(c)                E := E ∪ {s}
// Part 4: Inserting the extra points
        For each e ∈ E do
            If e ∉ Dg do
               E := E – {e}
            Else
               Let mp be the midpoint of e
               Dg := Constr_Del_Insert(mp, C, Dg)
(d, e)         E := E – {e}
// Part 5: Removing unnecessary edges
        For each s ∈ Edges(Dg) do
            If s ∩ support(UF) = ∅ do
               Dg.removeEdge(s)
            If s ∩ core(UF) = s do Dg.removeEdge(s)
(f)     Return (Dg)
    End Triangulate_Face
```

The *Constrained_Delaunay_Triang* (set of points, set of cycles) function creates a constrained Delaunay triangulation in which all the edges in the set of cycles are required to be in the triangulation. This is done using for instance the algorithm in [7].

The *Constr_Del_Insert* (point, set of cycles, constrained Delaunay triangulation) algorithm inserts a single point into an existing constrained Delaunay triangulation.

Figure 2d shows what the triangulation looks like after three extra points have been inserted in part 4 of the algorithm. The dashed lines show those marked lines from Figure 2c that have been removed by the dynamic Delaunay triangulation algorithm. Figure 2e shows the result after all the lines have been processed.

The order in which the lines are taken in part 4 is usually arbitrary. The results may vary depending on the order, but the quality of the result is not that different, and creating any form of optimized solution would take exponential time as all possible orders would have to be investigated. Tests by re-triangulating the same figures many times have shown that the number of extra points inserted typically varies by 12%, at worst 20%. The quality in this case is the number of triangles (fewer triangles are better, as more triangles need more storage space) and how narrow the triangles are (the closer the triangles are to equilateral, the better. If they are very long and thin, the interpolation may be done using points that are far away from the point of interest).

One exception to this is uncertain holes. In an uncertain hole, one should choose the line that passes closest to the center of the hole first. If the hole is close to circular in shape, it is likely that all other lines that need to be taken are eliminated by choosing that single line first, so no other lines need to be processed.

It may seem that the new lines that are inserted in part 4 may violate the condition that no line through the support should have values of 0.0 or 1.0 at both ends. In fact they do not, because every line that is inserted or modified when inserting a new point into the Delaunay triangulation will either begin or end in the new point (see the dynamic Delaunay triangulation algorithm in [1]). The new point has a value between 0.0 and 1.0.

Part 5 of the algorithm removes those line segments from the triangulation that are unnecessary. Lines that are outside the actual object have a function value of 0 along their entire length. Line segments that are entirely within the core have a function value of 1. These lines are therefore not needed.

The running times of the five parts of the algorithm are as follows:

P1. O(n) where *n* is the number of points because each point is visited once. Each cycle is also visited once, but this is a far smaller number than the points.

P2. O(n*log(n)) using a standard algorithm for constrained Delaunay triangulation such as the one in [7].

P3. O(n) because one must check each point once whether it is on the core or the support. Checking whether the line is outside or inside a polygon is also easy. Each end point must at this stage be either on the support or on the core. If the end point is on the support, one may compare the angle of the line to the angles of the lines to the previous and next points along the support. If one also knows which way is inside and outside, this angle determines whether the line is inside or outside the support. If both end points are on the core, one knows that the line is entirely within the support. The same method can be used to find whether the line is outside the core or not.

P4. O(n*log(n)) using a dynamic Delaunay triangulation algorithm.

P5. O(n) because all n-2 segments in the triangulation have to be checked. One may use the same method as given in point 3 to check whether a line segment is inside a polygon in constant time. Any point that is not on either core or support is known to be inside the support and outside the core.

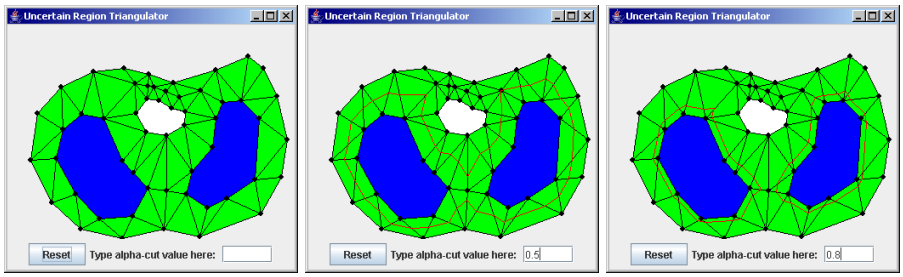

Figure 3. Screenshots from the application

Because these parts are done in sequence, the running time of the algorithm is the sum of the running times of the parts, or the asymptotic running time of the part with the highest running time. This is O(n*log(n)) in this case.

5. Implementing the algorithm

The triangulation algorithm for uncertain regions has been implemented in Java, along with algorithms that use the triangulation to compute alpha-cuts and to compute the probability that a give point is inside the region. The running time of the alpha-cut algorithm currently implemented is O(t + l*l) time where t is the number of triangles in the triangulation and l is the number of lines in the alpha-cut. If the Java implementation of TreeSet contained some additional functionality this could be reduced to O(t + l*log(l)) time. Some screenshots from the Java implementation are shown in Figure 3.

The algorithm for computing the probability of an individual point uses a tree of triangles constructed during the triangulation process to find the triangle that contains the point. The worst-case running time is O(t), while the average-case running time is O(log(t)).

6. Using the triangulation

In this section, an example operation that uses the triangulations is described: How to compute the probability that two uncertain faces overlap.

This operation computes the probability that two faces overlap. The first step in presenting it is to describe a simpler operation: The operation to compute the probability that an uncertain face overlaps with a crisp face. A lower boundary of this probability can be calculated by computing the intersection of the support of the uncertain face with the crisp face and using the triangulation to compute the maximum probability. This is done by computing the probability value at each point in which the crisp face crosses an edge of the triangulation and in each vertex of the border of the crisp face that is inside the support of the uncertain face, and then choosing the highest value from those computed.

If the boundary were constrained to follow one of the iso-lines of probability this would yield a correct solution. In other cases it yields a lower bound.

The positions of the border of a region at two different points are not necessarily independent of each other. However, one might describe how curved the border can be by defining a distance *indist* and assuming that this far away, the position of the border of the uncertain face is for all practical purposes independent. Then in addition to computing the probability at the point where it is highest, compute the probability at other points along the boundary of the crisp face with distance *indist* between them. The two faces overlap if they overlap at at least one of these points.

To do this for two uncertain faces one needs to take two independent probability functions into consideration. To do this exactly is quite complex, but it is possible to find a simple approximate solution using a method similar to the method from [12]. Create *n* alpha-cuts for the face with the least uncertain border (with the narrowest gap between the support and the core). Then use the method from the previous paragraph with each of these alpha-cuts as the crisp face. Multiply the probability from each alpha-cut with the difference in probability between the cuts. Add these multiplied probabilities together to get the final probability. A large *n* yields a more accurate result but takes more time.

Algorithm UncertainOverlap
Input: A crisp face *cf* and an uncertain face *uf*
Output: The probability that they overlap
Method:
  Let *pcf* be the points that define the boundary of *cf*
  Let *probSet* be an empty set of probabilities
  Foreach $p \in pcf$ do
    If (Inside(*p*, Support(*uf*))) **do**
      If (Inside(*p*, Core(*uf*))) **do Return**(1.0)
      Find which triangle in the triangulation of *uf* that *p* is inside and use that to compute the

```
            probability prob that that point is inside uf
        probSet := probSet ∪ {prob}
    Foreach (line tl that is a part of the triangulation)
    do
        If (Cross(tl, Border(cf))) do
            Let cp be the point at which tl crosses the
            border of cf
            Compute the probability prob that the point
            cp is inside uf by using the probabilities at
            each end of tl
    probSet := probSet ∪ {prob}
    Return (The highest element of probSet)
End UncertainOverlap
```

Running time: The second Foreach sentence can be done in O((n + I)*log(n)) time where I is the number of intersection points by using the line intersection algorithm from section 2.1 of [1]. This algorithm also returns the lines that cross at that point so computing the probability takes O(1) time.

The first Foreach sentence can be done in the same time using the MapOverlay algorithm from section 2.3 of [1]. I here is the complexity of the overlay itself. Using the resulting overlay one can discover which points from *cf* are inside *uf* and which triangle each point is inside (as well as whether any points are inside the core).

7. Analysis of the method

The number of points inserted is typically much lower than the number of lines in the original triangulation. In the example in Figure 2, the number of new points is approximately 1/3 the number of points in the original figure.

To compute how much additional storage would be needed to store the triangulation, the storage space needed to store a spaghetti model will be presented first. This is $pv + rh$ where p is the storage needed for a point, h is the number of holes in the region, and r is the storage needed for a pointer.

As the algorithms that use the triangulation mostly are based on triangles and not lines, it makes sense to store the triangles instead of the additional lines, especially as the storage space needed is almost the same. Using Euler's formula for planar graphs one can find that the maximum number of lines in a triangulation is $3v - 3$, where v is the number of points in the triangulation, and the maximum number of triangles is $2v - 2$. For each line one would have to store a pointer to its end points. For each triangle one must store pointers to its three corners. Thus the storage space needed for the lines is $2r*(3v - 3) = r*(6v - 6)$, and the storage space needed for the triangles is $3r*(2v - 2) = r*(6v - 6)$.

The total storage space needed is $pv + rh + r*(6v - 6)$. Remember also that the number of points v is slightly larger for the triangulated version than for the spaghetti version.

To see how much storage space is increased, some sample data has been inserted into the formulae. The OpenGIS model uses double-precision floating-point numbers to store the coordinates of points. P is thus 16 bytes. If one assumes that an object cannot have more than 64000 points, a pointer can be stored with 2 bytes. If one assumes that the number of holes is small compared to the number of points, that term can be ignored as it contributes little. The increase in storage space is then $(pv + r*(6v - 6)) / pv$. For large numbers of points, the constant 6 contributes little, so it is usually safe to ignore it. The formula then becomes $(pv + 6rv)/pv$, which becomes $(p + 6r) / p$. 16 + 12 / 16 = 28/16 = 7/4 = 1.75. Multiply this by the ratio of points in the two cases to get the final answer, on average 4/3 * 7/4 = 7/3 = 2.3.

8. Conclusions

This paper has presented an algorithm that is capable of creating a triangulation of an arbitrary uncertain object for the purpose of computing probability values. The triangulation is generated in O(n*log(n)) time. Additional points may increase the storage requirement somewhat if the support has large holes or concavities not shared by the core, or the support of an uncertain point is highly elliptical.

Methods based on two-dimensional probability theory or stochastic simulation can be used to generate these probabilities for relatively simple spatial objects [20], but two-dimensional probability theory becomes unwieldy for vector objects consisting of more than three points. Stochastic simulation needs a lot of processing power and may therefore only be used if one has few objects, but not in a large spatial database.

Our approach is an improvement over previous approaches. Compared to the approach in [15], the present approach can generate consistent results for arbitrary objects, it can easily generate alpha-cuts, and it is also easier to generate other function-dependent information such as the area of the uncertain object. In [9], the area of an uncertain object is defined as the integral over space of its fuzzy set value function. Generating this value is much easier for a set of triangles than for an arbitrarily shaped object.

While our earlier approach [15] had some limitations, the present approach works for all kinds of faces.

Both probabilities and fuzzy set values are values between 0 and 1. Although our paper has described how to compute probability values, the triangulation could equally well be used to compute fuzzy set values. The only difference would be in which operations are implemented. For instance computing the probability that two fuzzy regions meet is meaningless whereas using an alpha-cut as a threshold makes as much sense for fuzzy regions as for uncertain ones. The triangulation can also be used to compute the fuzzy topological relationships between regions as described in [2].

One step in determining the topological relationship between two fuzzy regions in [2] is to find the fuzzy overlap between them. The fuzzy overlap between two faces can be computed in a way similar to the method from Section 6. Fuzzy sets use maximum and minimum values rather than multiplication. Thus the goal of a fuzzy overlap operation is to find the largest value of $\min(f_a(x, y), f_b(x, y))$ where f_a is the fuzzy set function for region a. This value can be approximated by for each of the n alpha-cuts of b, compute the largest value of f_a. Then return the largest value of all alpha-cuts.

References

[1] M. de Berg, M. van Krevald, M. Overmars and O. Schwarzkopf: *Computational Geometry: Algorithms and Applications, 2nd edition.* Springer-Verlag, 2000.

[2] J. T. Bjørke: "Topological relations between fuzzy regions: derivation of verbal terms". *Fuzzy Sets and System*, 141(3): pp. 449-467, 2004.

[3] R. Cheng et al.: "Querying Imprecise Data in Moving Object Environments". In *IEEE Transactions on Knowledge and Data Engineering,* Vol. 16, No. 9, pp. 1112-1127, Sep 2004

[4] E. Clementini and P. Di Felice: "An Algebraic Model for Spatial Objects with Indeterminate Boundaries". In *Geographic Objects with Indeterminate Boundaries*, P. Burrough and A. Frank (eds.), Taylor & Francis, 1996, pp. 155-169.

[5] E. Clementini: "A model for uncertain lines". In *Journal of Visual Languages and Computing*, 16, pp. 271-288, 2005.

[6] A. G. Cohn and N. M. Gotts: "The 'Egg-Yolk' Representation of Regions with Indeterminate Boundaries". In *Geographic Objects with Indeterminate Boundaries*, P. Burrough and A. Frank (eds.), Taylor & Francis, 1996, pp. 171-187.

[7] L. P. Chew: "Constrained Delaunay Triangulations". *Proceedings, 3rd International. Symposium on Computational Geometry*, Waterloo, Canada, ACM Press, 1987, pp. 215-222.

[8] P. Lagacherie, P. Andrieux and R. Bouzigues: "Fuzziness and Uncertainty of Soil Boundaries: From Reality to Coding in GIS". In *Geographic Objects with Indeterminate Boundaries*, P. Burrough and A. Franks (eds.), Taylor & Francis, 1996, pp. 275-298.

[9] V. Ljosa and A. K. Singh: APLA: "Indexing arbitrary probability distributions". In *Proceedings of the 23rd International Conference on Data Engineering, 2007*, pp. 946-955.

[10] D.Pfoser and C. Jensen: "Capturing the uncertainty of moving-objects representations". In *Proc. SSDBM*, 1999, pp. 111-131.

[11] M. Schneider: "Metric Operations on Fuzzy Spatial Objects in Databases". *Proceedings, 8th ACM Symposium on Geographic Information Systems*, Washington D.C., ACM Press, 2000, pp. 21-26.

[12] M. Schneider: "Fuzzy Topological Predicates, Their Properties, and Their Integration into Query Languages". *Proceedings, 9th ACM Symposium on Geographic Information Systems*, Atlanta, ACM Press, 2001, pp. 9-14.

[13] P. A. Sistla, et al.: "Querying the uncertain position of moving objects", In *Temporal Databases: Research and Practice*, Lecture Notes in Computer Science, 1998, pp. 310-337.

[14] E. Tøssebro and R. H. Güting: "Creating Representations for Continuously Moving Regions from Observations". *Proceedings, 7th International Symposium on Spatial and Temporal Databases*, Lecture Notes in Computer Science 2121, Springer, 2001, pp. 321-344.

[15] E. Tøssebro and M. Nygård: "An Advanced Discrete Model for Uncertain Spatial Data". *Proceedings, 3rd International Conference on Web-Age Information Management*, Lecture Notes in Computer Science 2419, Springer, 2002, pages 37-51.

[16] E. Tøssebro and M. Nygård: "Medium Complexity Discrete Model for Uncertain Spatial Data". *Proceedings, 7th International Database Engineering and Applications Symposium*, IEEE Computer Society, 2003, pp. 376-384.

[17] E. Tøssebro and M. Nygård: "Representing Topological Relationships for Moving Objects". *Proceedings, 4th International Conference on Geographic Information Science*, LNCS 4197, Springer Verlag, 2006, pages 383-399.

[18] E. Tøssebro: *Representing Uncertainty in Spatial and Spatiotemporal Databases.* Ph.D Thesis nr. 2002:88 at the Norwegian University of Science and Technology (NTNU).

[19] M. Worboys: *GIS: A Computing Perspective.* Taylor & Francis, 1995.

[20] J. Zhang and M. Goodchild: *Uncertainty in Geographic Information*. Taylor & Francis, 2002.

[21] F. B. Zhan: "Approximate analysis of binary topological relations between geographic regions with indeterminate boundaries". *Soft Computing* 2, 1998, pages 28-34.