Computing the Highest Density Region using Voronoi and Delaunay techniques

Bachelor thesis
Econometrics and Operational Research
Joeri Admiraal, 342478 *
Erasmus University Rotterdam

June 30, 2013

Abstract

Highest density regions have a great use as prediction regions. There are some methods to compute a highest density region, but they need information about the probability density function or are only suited for certain cases. In this paper we show a new method to approximate the highest density region using Voronoi and Delaunay techniques. First we use these techniques to create a graph, after which we compute the highest density region using two different graph algorithms: one that is faster and one that is more accurate. Compared to the symmetric prediction region, these algorithms perform very well and also for non-convex contour shaped densities both algorithms work as expected.

*Supervised by Dr. Wilco van den Heuvel

Contents

1 Introduction **2**

2 Methods **3**

2.1 Voronoi and Delaunay techniques . . . 3

2.1.1 Definition Voronoi diagram . . . 3

2.1.2 Relation to Delaunay triangulation . . . 3

2.1.3 Algorithms . . . 4

2.1.4 Data structure . . . 4

2.1.5 Creating the Voronoi diagram . . . 5

2.2 Graph theory . . . 7

2.2.1 Data structure . . . 8

2.2.2 Top-down algorithm . . . 8

2.2.3 Bottom-up algorithm . . . 10

3 Experimental results **11**

3.1 Independent bivariate normal distribution . . . 11

3.1.1 Theoretical symmetric prediction region . . . 11

3.1.2 Heuristics . . . 11

3.2 Dependent bivariate normal distribution . . . 14

4 Conclusion **16**

5 Discussion **16**

Appendix A Detailed pseudocode **18**

A.1 Main procedure . . . 18

A.2 Delaunay triangulation . . . 18

A.3 Voronoi diagram . . . 19

A.4 Graph heuristics . . . 20

A.4.1 Top-down algorithm . . . 20

A.4.2 Bottom-up algorithm . . . 20

A.4.3 Check connected subgraphs . . . 21

1 Introduction

For many statistical methods it is essential to get a summary of a given probability distribution in the form of a region in which an observation will be located with a certain probability. This is a so called prediction region (or forecast region in some literature). For example, a 95% prediction region contains approximately 95% of the observations. These prediction regions are of great use for testing the accuracy of a model.

Often a prediction region is calculated with the mean and standard deviation or with quantiles. These symmetric prediction regions are acceptable when prediction densities are normal. But when a model is non-linear or non-normal, prediction densities are often not normal and lead to asymmetric or possibly multimodal prediction regions [1, 2, 3]. Therefore another method to compute the prediction region is recommended, but there are many ways to construct one.

The most intuitive prediction region is the highest density region (HDR). In Bayesian analysis this is called the Highest Posterior Density Region (HPD) or Bayesian confidence set. We define the HDR with the following criteria, which are equal to the criteria in the definition by Box and Tiao [4]:

1. The surface of the region is minimized.
2. Every observation inside the region has a probability at least as large as every observation outside the region.

A more precise definition by Hyndman [6] is as follows: let $f(x)$ be the density function of a random variable X. Then the $100(1-\alpha)\%$ HDR is the subset $R(f_\alpha)$ of the sample space of X such that $R(f_\alpha) = \{x : f(x) \geq f_\alpha\}$, where f_α is the largest constant such that $Pr(X \in R(f_\alpha) \geq 1-\alpha)$.

For normal densities and any other unimodal symmetric distribution the HDR will lead to the same prediction region as the symmetric prediction region. For non-normal densities other prediction regions will fail, whereas the HDR will provide better results [5].

There are only few articles about HDRs. Hyndman [6] showed a method to approximate the HDR with a density quantile approach, using a kernel density estimator. However, the probability density function must be known for this method or should be approximated. Fadallah [7] provided another algorithm where no information about the probability distribution is needed, but works only for problems with convex contour shapes.

In this research we examine the possibility of a new algorithm for computing a HDR using Voronoi techniques that works not only for convex contour shaped probability distributions, but also for non-convex ones and does not need any information about the probability density function. In Section 2 we investigate the methods for creating a Voronoi diagram and using this to compute the HDR with graph theory. In Section 3 we test our methods, after which we can answer our research question.

2 Methods

The algorithms we propose work in two steps. First we divide the surface up into different cells using Voronoi techniques. Afterwards, we construct a HDR using graph theory.

2.1 Voronoi and Delaunay techniques

2.1.1 Definition Voronoi diagram

A Voronoi diagram is a way to split a surface up into different cells. This is done in a way that each Voronoi cell contains exactly one vertex. Furthermore, each point in the surface is located in the region of its closest vertex. Formal definition of a Voronoi cell: $R_k = \{x \in X | d(x, P_k) \leq d(x, P_j) \textit{ for all } j \neq k\}$, where $x \in X$ are all points in space, R_k is Voronoi cell k, P_k is vertex k in Voronoi cell k and *d(i,j)* is the distance between points i and j.

2.1.2 Relation to Delaunay triangulation

It is also noteworthy that the Voronoi diagram is the dual graph of the Delaunay triangulation [8]. A triangulation is made by subdividing a surface up into triangles and is a Delaunay triangulation if there are no other vertices inside the circumcircle of every triangle. When one obtains a Delaunay triangulation, one can easily transform it into a Voronoi diagram (or the other way around). Transforming a Delaunay triangulation into a Voronoi diagram is done by connecting the circumcenters of two triangles with a joint edge, for all edges (see Figure 1). In this research, this is very convenient as the Delaunay triangulation shows which Voronoi cells are connected to each other and can be used to create a graph.

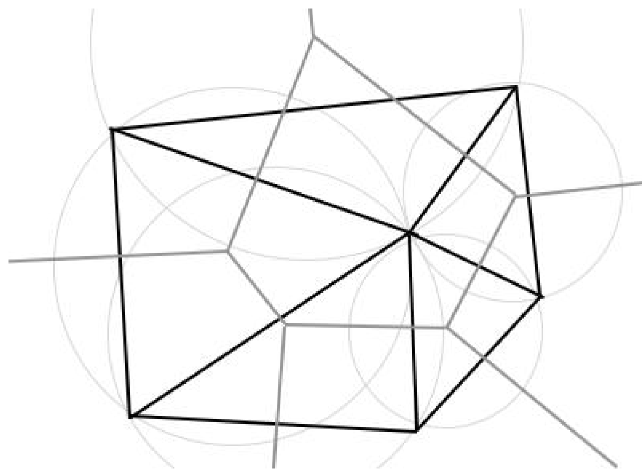

Figure 1: Duality between Voronoi diagram (in gray) and Delaunay Triangulation (in black)

2.1.3 Algorithms

There are several categories of algorithms to create a Voronoi diagram or Delaunay triangulation. First the only widely known Voronoi algorithm is given. Furthermore two Delaunay triangulation techniques are given, in which most of the popular Delaunay triangulation algorithms will fall.

Sweepline Voronoi algorithm This algorithm proposed by Fortune [9] creates a Voronoi diagram using a sweepline technique and works only for two dimensions. A line sweeps from one side of the surface to the other side, slowly including all vertices into the Voronoi diagram.

Divide and conquer Delaunay algorithms These algorithms split the surface recursively up into different regions and compute the Delaunay triangulation for each region. Later on these regions are merged. The first version of this algorithm was implemented by Lee and Schachter [10], was then improved by Guibas and Stolfi [11] and later by Dwyer [12].

Incremental Delaunay algorithms This is a method that adds one observation at a time and computes the Delaunay triangulation repeatedly for each step. The Bowyer-Watson algorithm [13, 14] is a very popular incremental algorithm. Lawson [15] has also created a widely used algorithm.

All algorithms run in $O(n \log n)$ time for non-uniform distributed observations [16]. Because the incremental algorithms are the easiest ones, we use one of these. At first, both mentioned algorithms are quite easy to understand. However, it is slightly more difficult to implement them. After some investigation the Bowyer-Watson algorithm seems to be somewhat easier to implement, so this is the algorithm we use. Both Delaunay triangulation methods can be implemented for higher dimensions than two, but in this research we focus on the two dimensional bivariate case.

2.1.4 Data structure

Almost all algorithms, including the Bowyer-Watson algorithm, use a quad-edge data structure, first proposed by Guibas and Stolfi [11]. This array-based data structure is in practice considerably faster than a pointer-based data structure [16] and is constructed with duality in mind, so it is easy to determine the Voronoi diagram afterwards. The quad-edge data structure has classes for vertices, edges and triangles, with emphasis on edges.

Vertices A vertex is the most essential data type, as edges and triangles depend on vertices. A vertex has two coordinates representing its horizontal and vertical position. We also give each vertex an id (unique number) and maintain a list of id's of edges the vertex is connected to.

Triangles Triangles have three vertices and also contain a list of id's of its three edges. Triangles also have a circumcircle: a circumcenter and a radius. Now we can check whether a given vertex is located in the circumcircle of this triangle. Triangles also have a list of id's of edges it contains.

Edges Edges contain two vertices, between which the edge is located. An edge also contains two triangles, one for each side of the edge. Edges also have an id: an unique number that can be used to store an edge in vertices and triangles.

2.1.5 Creating the Voronoi diagram

With the techniques mentioned above we can start constructing a graph. First we run our main procedure. For a given data set, all observations are transformed into vertices, with an x and y coordinate. Then we sort the vertices on x-coordinates. This could speed up the triangulation, as we could locate the vertex somewhat faster. For a detailed description of the main procedure, see Appendix A.1. Now we create a Delaunay triangulation and use that to obtain the Voronoi diagram.

Delaunay triangulation First we use the Bowyer-Watson algorithm and quad-edge data structure to create a Delaunay triangulation. A short pseudocode for the triangulation can be found in Algorithm 1. For a more detailed description, see Appendix A.2.

Algorithm 1 Bowyer-Watson Delaunay triangulation

```
create super triangle (enclosing all vertices)
for all vertices v do                                ▷ insert vertices one by one
    find triangles for which v is located in its circumcircle
    delete these triangles to get insertion polygon
    triangulate insertion polygon by creating edges from its vertices to v
end for
remove super triangle
```

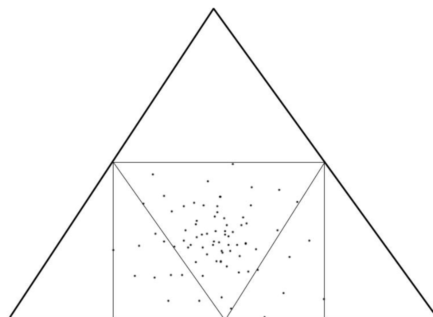

Figure 2: Creating the super triangle

First we create a super triangle, as illustrated in Figure 2. This is done by creating an enclosing rectangle and then creating a triangle enclosing this rectangle. The rectangle is easily made by calculating the minimum and maximum x- and y-coordinates. We now create a triangle as large as possible within this rectangle and duplicate this triangle three times to create a super triangle, inspired by the Sierpinski triangle.

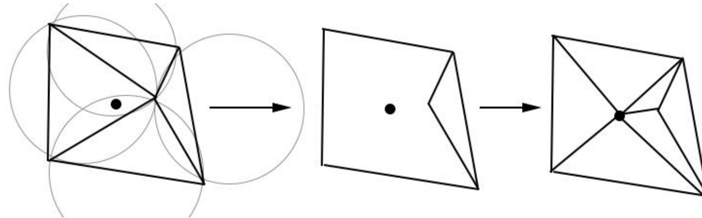

Figure 3: Finding faulty triangles and triangulate insertion polygon

Afterwards we insert the vertices one by one into the current triangulation, as illustrated in Figure 3. First we need to find all faulty triangles: triangles for which the inserted vertex is located in its circumcircle. It would be very time consuming to check this for every triangle, so we use another technique. First we locate the one faulty triangle, where we start with the last created triangles. Because we sorted the triangles beforehand, the inserted vertex is likely to be located near the last inserted vertices. Now we can find the remaining faulty triangles by checking adjacent triangles of all known faulty triangles, until there are no triangles left to check. We get the insertion polygon as a result of merging all faulty triangles. Then we can triangulate this insertion polygon by adding edges from its vertices to the inserted vertex.

Voronoi diagram We use this Delaunay triangulation to obtain the Voronoi diagram with Algorithm 2. For a more detailed pseudocode, see Appendix A.3.

Algorithm 2 Transformation Delaunay to Voronoi

Require: Delaunay triangulation
  for all vertices v in Delaunay triangulation **do**
    for all edges e connected to v **do**
      create edge between circumcenters of adjacent triangles of e
    end for
    calculate area of this Voronoi cell
  end for

As we used a quad-edge data structure in the construction of the Delaunay triangulation, it is quite easy to obtain the Voronoi diagram. To create a Voronoi cell around a vertex, one connects all circumcenters of the adjacent triangles (or adjacent triangles to adjacent edges). With this information it is easy to calculate the area of all Voronoi cells.

One problem however is how to determine the area of all boundary Voronoi cells: the cells that are on the edge of the Voronoi diagram. These cells are not

bounded with edges in all directions and theoretically they have an unlimited area. As it is most unlikely these cells are to be included in a HDR, we set the area to infinity so our algorithms also do not include them in the HDR.

2.2 Graph theory

With the Voronoi diagram and Delaunay triangulation, we can finally construct the HDR. We want to create a region with $100(1-\alpha)\%$ of the vertices, which have a weight equal to the area of its Voronoi cell. We now create a graph with the Delaunay triangulation and the Voronoi diagram. This graph $G = (V, E)$ consists of the vertices V from the observations and edges E from the Delaunay triangulation. Each vertex v has a weight $f(v)$, representing the area of its Voronoi cell.

Our goal is to obtain a subgraph with only $100(1-\alpha)\%$ of the vertices, with the smallest sum of areas. So we seek a subgraph $H = (W, E')$ with $W \subseteq V$ and $|W| \geq (1-\alpha)|V|$, minimizing $\sum_{i=1}^{|W|} f(w_i)$. This subgraph is easy to obtain by adding only the $(1-\alpha)|V|$ vertices with the lowest weight to H. In Figure 4 one can see an example outcome of this smallest subgraph, which might not be exactly what we want.

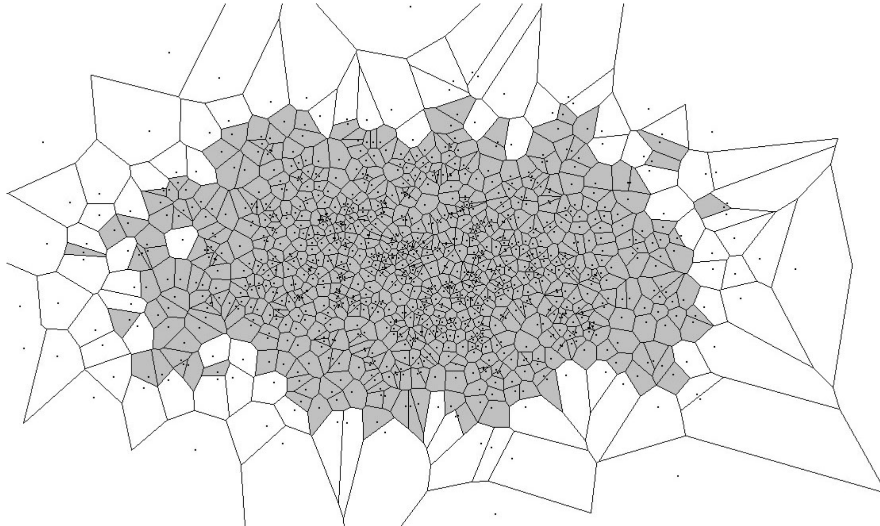

Figure 4: Smallest subgraph

First of all, there are gaps in the computed HDR. This is not reasonable for most probability distributions, so we might want to avoid this. We can add a restriction to our minimization problem in order to avoid these gaps. We start with graph G and end with subgraph H as HDR and another subgraph $I = (U, E")$ with $U = V - W$ and $E" = E - E'$. If this subgraph I with all vertices not in the HDR is a connected graph, then H does not contain gaps. Connected means that there exists a path $u_1, u_2, ..., u_k$ between each combination u_1, u_k in I (one might use a dummy vertex connected to all boundary vertices). In most cases it would be wise to add this restriction, but when stumbling upon

a multimodal probability distribution where gaps are expected, this could be relaxed.

Secondly, our computed HDR consists of multiple regions. Apart from the large central region, there are some cells not connected to this region. As this is also not a very logical result for most probability distributions, we want to prevent this from happening too. This can be done by ensuring that subgraph H is a connected graph. This means that there exists a path $w_1, w_2, ..., w_k$ between each combination w_1, w_k in H. This is preferable for unimodal probability distributions, but for multimodal probability distributions this might be relaxed.

In this research we will focus on the most common case: an unimodal probability distribution where we assume gaps are not present and the HDR is connected. First we will explain the data structure, then we will give two different algorithms: a top-down algorithm and a bottom-up algorithm.

2.2.1 Data structure

Before we can use one of our algorithms we have to create the data structure. Because the vertices hold most of the information, these vertices are the most important part of the graph. The vertices already exist after the Delaunay triangulation and have a weight equal to the area of its Voronoi cell. We added a list of id's of vertices this vertex is connected to. Now we can quickly find the adjacent vertices of each vertex, without relying on the edges from the Delaunay triangulation.

2.2.2 Top-down algorithm

The top-down algorithm begins with a full graph and removes the vertices with the highest weight one by one. This Algorithm 3 ensures the graph remains connected and without gaps. See Appendix A.4.1 for a more detailed pseudocode.

First we sort the vertices on weight from high to low and then calculate the number of vertices to be removed. For each removal, we search for the vertex with the highest weight that does not result in a disconnected subgraph H or I. The simplest way of checking whether a subgraph is connected is as follows. We start with one vertex of the subgraph and add it to a list. Now we add all adjacent vertices of this vertex to the list and continue with their adjacent vertices that are not already in this list. This goes on till we cannot find any new adjacent vertices. If this list now contains all vertices from our initial subgraph, this subgraph is connected. While this method is very straightforward, it might take some time when the subgraph contains a high number of vertices. Therefore we decided to use another method.

Checking whether H remains connected is done by looking at the neighboring vertices of the to be removed vertex. We count the amount of groups of neighboring vertices. If two adjacent neighbors are both in H or both in I, they are in the same group. When there are more than two groups, this means the

Algorithm 3 Top-down algorithm

Require: $G = (V, E)$
  sort V on $f(v)$
  create subgraph $H = (W, E')$, initially with $W = V$ and $E' = E$
  create subgraph $I = (U, E")$, initially empty
  for $\alpha \times |V|$ **do** ▷ remove α % of the vertices
    set $found$ as false
    while $found$ is false **do**
      try next not already tried $w \in W$ with highest $f(w)$
      remove w from H and add to I
      if H or I is not connected anymore **then**
        revert last step
      else
        set $found$ as true
      end if
    end while
  end for

to be removed vertex is the only connection between two regions of vertices and removal results in a disconnected subgraph H. Checking whether I is still connected can be done in almost the same way. Again we count the number of different groups the to be removed vertex has. When there is only one group of neighbors, this means that this vertex is fully surrounded by vertices in H and removal results in a gap. Detailed pseudocode can be found in Appendix A.4.3.

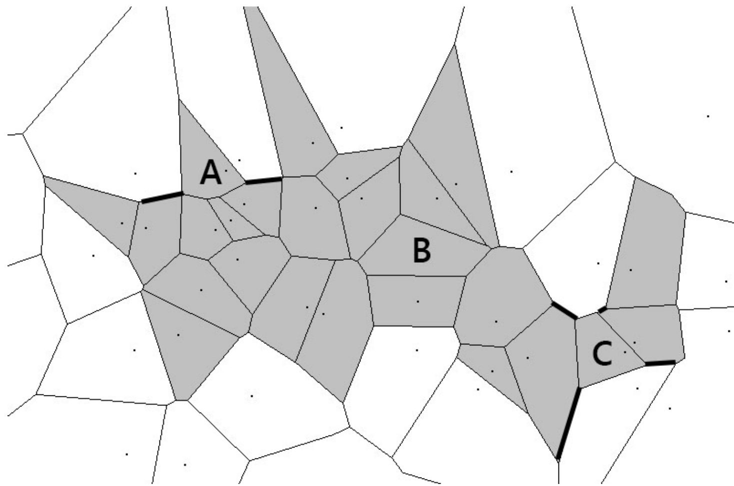

Figure 5: Ensuring connected graphs

For an example, see Figure 5. We can see vertex A has two different groups of neighbors, separated by the thick lines, and removal is approved. Vertex B has only one group of neighbors, so removal results in a disconnected graph I. Removing vertex C on the other hand, results in a disconnected graph H, because of the four different groups of neighbors.

The only exceptions are the boundary vertices, otherwise our algorithm would not run as all vertices have only one group of neighbors. So altogether, we only remove boundary vertices and vertices with two different groups of neighbors.

2.2.3 Bottom-up algorithm

This algorithm is the opposite from the top-down algorithm, where we start with the full graph and slowly remove vertices with a high weight. Here we start with an empty HDR subgraph H and add vertices with a low weight one by one. This is explained in Algorithm 4 and elaborated in more detail in Appendix A.4.2.

Algorithm 4 Bottom-up algorithm

Require: $G = (V, E)$
  sort V on $f(v)$
  create subgraph $H = (W, E')$, initially empty
  create subgraph $I = (U, E")$, initially with $U = V$ and $E" = E$
  for $(1 - \alpha) \times |V|$ **do** ▷ add α % of the vertices
    set $found$ as false
    while $found$ is false **do**
      try next not already tried $u \in U$ with lowest $f(u)$
      remove u from I and add to H
      if H or I is not connected anymore **then**
        revert last step
      else
        set $found$ as true
      end if
    end while
  end for

First we start with the vertex with the lowest weight. Then we add one by one the vertex with the lowest weight that does not result in a gap or a disconnected H. To check this we use the same method described in the top-down algorithm, but in this case we do not have to concern about the boundary vertices as exceptions.

3 Experimental results

We now test the aforementioned algorithms with a few experiments. First we examine the methods on an independent bivariate normal distribution, because these results can be compared with a symmetric prediction region. Afterwards we also test our algorithms on an unimodal bivariate non-convex contour shaped distribution.

We want to be sure the vertices on the edge of the graph are removed by our algorithm. As we set the weight of these vertices to infinity, they are the least likely to be included in the HDR. But for this to work, the number vertices to be excluded from the HDR must be sufficiently high. This depends not only on the amount of vertices and the percentage of vertices to be excluded (α), but also on the probability distribution of the vertices. The lower α is, the more vertices are needed. The kurtosis of the distribution also plays a role: the more vertices are located in the tails of the distribution, the more boundary vertices there are and the more vertices are required to compute the HDR (for a certain α). Otherwise, there is a chance some boundary vertices are included in the HDR, which results in an infinite area.

All testing is done on a computer with an i5-520M processor, using only one core at 2.40 GHz.

3.1 Independent bivariate normal distribution

The independent bivariate normal distribution consists of two independent univariate normal distributions. We can now generate two independent normal variables which can represent the x- and y-coordinate of a vertex i, with $x_i \sim \mathcal{N}(\mu_1, \sigma_1)$ and $y_i \sim \mathcal{N}(\mu_2, \sigma_2)$.

3.1.1 Theoretical symmetric prediction region

We use the formulas from Chew [17] to obtain a theoretical symmetric prediction region. The $100(1-\alpha)\%$-prediction region for a multivariate normal distribution is an ellipse with the equation $(x-\mu)'\Sigma^{-1}(x-\mu) \leq c^2 = \chi^2(\alpha, p)$. For the bivariate normal distribution, the chi-squared distribution can be simplified to an exponential distribution, which results in $c^2 = -2\ln(\alpha)$. As the two dimensions are independent, the covariance matrix is a diagonal matrix. The resulting ellipse has its center at (μ_1, μ_2), has a width of $2\sigma_1 c$ and a height of $2\sigma_2 c$. So now the area of this region can easily be calculated with $-2\ln(\alpha)\pi\sigma_1\sigma_2$.

3.1.2 Heuristics

Now we are going to test the top-down algorithm and bottom-up algorithm. We compare these to each other and to the theoretical prediction region. For this example we generated 10000 vertices with two independent standard normal variables, $x_i \sim \mathcal{N}(0, 2)$ and $y_i \sim \mathcal{N}(0, 1)$. Our chosen α is 0.1.

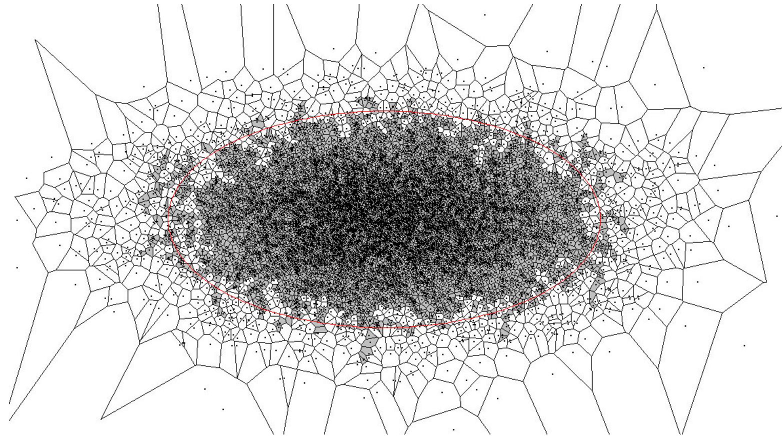

Figure 6: Top-down algorithm (convex experiment)

We can see in Figure 6 the top-down algorithm does a pretty good job, but has a downside. Slightly visible in picture, but becomes more clear when removing a higher percentage of the vertices, is that the HDR gets large so-called tentacles at the edges of the HDR. The reason is vertex with a low weight located at the end of the tentacle that does not get removed. Because of the algorithm, all other vertices in the tentacle cannot be removed too, as this causes the HDR to be disconnected.

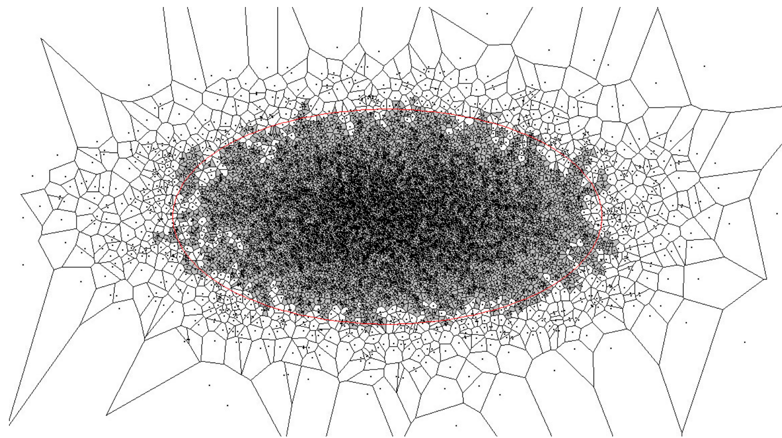

Figure 7: Bottom-up algorithm (convex experiment)

It is clear by Figure 7 that the bottom-up algorithm has this problem too, but to a lesser extent. This is now caused by vertices with a high weight near the center of the HDR. The difference is that the probability that a vertex with a low weight is located far from the mode of our probability distribution is higher than the probability that a vertex with a high weight is located close to the mode. This is because there are far more vertices near the mode and therefore will decrease the chance that weights greatly deviate from their theoretically expected weight.

Algorithm	Top-down	Bottom-up
Voronoi time (ms)	1355	1355
Theoretical area	28.94	28.94
Graph time (ms)	267	8233
HDR area	27.48	27.42
Similarity HDR and theory (%)	97.52	98.21

Table 1: Results with $|V| = 10000$ and $\alpha = 0.1$

In Table 1 we can compare the results of both algorithms in more detail. The areas of the HDRs computed by both algorithms are almost the same, with a small advantage for the bottom-up algorithm. More important is that the HDR computed by the bottom-up algorithm is more similar to the theoretical symmetric prediction region. This means that more vertices from the HDR are also in the symmetric region and could be a good indicator for the accuracy of the algorithms. This difference in similarity is present because the top-down algorithm has longer tentacles, which causes more vertices to be located outside the symmetric region. However, this slight advantage for the bottom-up algorithm comes at a cost: the top-down algorithm is much faster than the bottom-up algorithm.

$\|V\|$		100	1000	10000	100000
Voronoi time (ms)		81	408	1355	57561
Theoretical area		28.94	28.94	28.94	28.94
Graph time (ms)	Top-down	4	20	267	28741
	Bottom-up	14	202	8233	2342964
HDR area	Top-down	43.10	28.37	27.48	28.10
	Bottom-up	43.10	28.31	27.42	27.75
Similarity with theoretical (%)	Top-down	96.67	97.89	97.52	97.32
	Bottom-up	96.67	98.00	98.21	98.41

Table 2: Results with $\alpha = 0.1$ and variable $|V|$

In Table 2 we can see that increasing $|V|$ does not greatly improve the similarity of both algorithms (they are quite high anyway). And while the computation time for both algorithms greatly increases, the area also remains roughly the same. The only exception is the case with the smallest number of vertices. Because of the few vertices, all Voronoi cells have a quite large area. While many vertices are inside the theoretical area, their Voronoi cell is also partly outside of it.

As one can see in table 3, the difference between both algorithms changes as α varies. As α decreases, the top-down algorithm gets faster. The bottom-up algorithm should theoretically become slower when decreasing α, but that cannot be concluded from the table. What can be seen is that the similarity, and probably the accuracy too, decreases as α gets smaller. This has a smaller influence on the bottom-up algorithm and is again caused by the tentacles both algorithms produces.

α		0.01	0.05	0.1	0.25	0.5
Voronoi time (ms)		1355	1355	1355	1355	1355
Theoretical area		57.87	37.65	28.94	17.42	8.71
Graph time (ms)	Top-down	97	226	267	803	5004
	Bottom-up	10267	11107	8233	8725	7095
HDR area	Top-down	58.07	36.18	27.49	16.20	7.98
	Bottom-up	58.22	36.21	27.42	16.08	7.38
Similarity with theoretical (%)	Top-down	99.75	99.07	97.52	92.37	75.70
	Bottom-up	99.74	99.15	98.22	95.93	87.40

Table 3: Results with $|V| = 10000$ and variable α

3.2 Dependent bivariate normal distribution

To show how the algorithms perform on a distribution where the HDR is not a convex region, we test our algorithm on a bivariate distribution where $x_i \sim \mathcal{N}(0, 2)$ and $y_i \sim \mathcal{N}(|x_i|, 1)$.

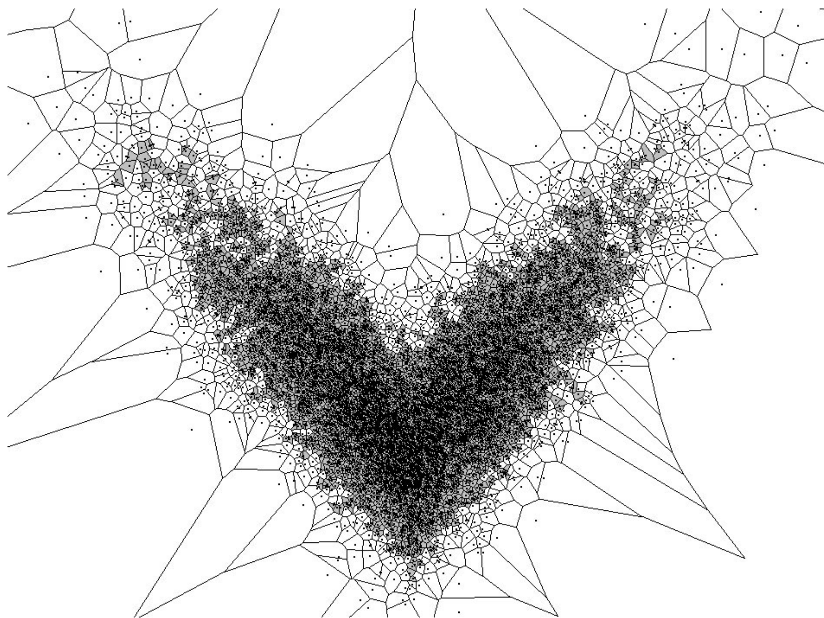

Figure 8: Top-down algorithm (non-convex experiment)

We can see in Figure 8 and 9 that both algorithms work as expected. Again does the top-down algorithm produce long tentacles, something the bottom-up algorithm does not encounter that much.

According to Table 4, also in this case does the bottom-up algorithm produce a HDR with a slightly smaller area, but again it takes more time than the top-down algorithm.

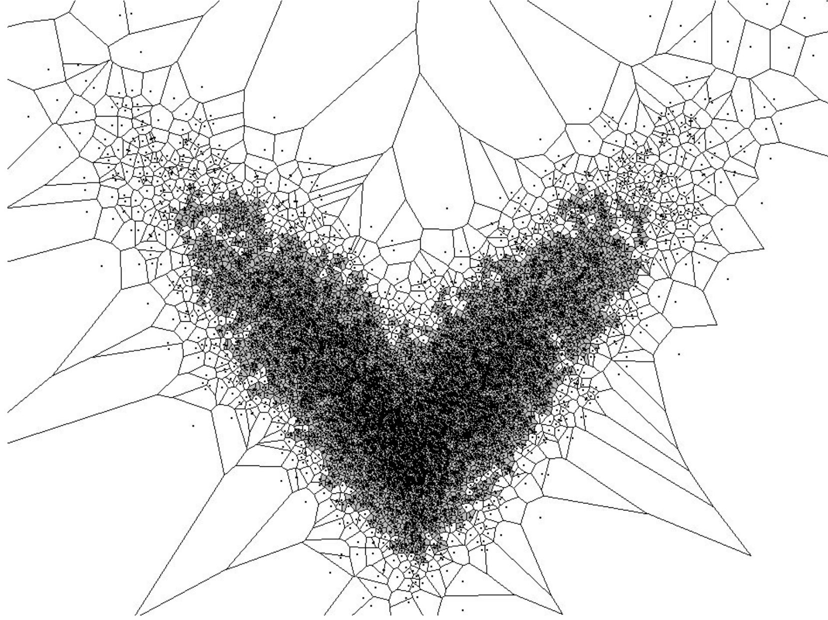

Figure 9: Bottom-up algorithm (non-convex experiment)

Algorithm	Top-down	Bottom-up
Voronoi time (ms)	1547	1547
Graph time (ms)	306	1808
HDR area	27.57	27.32

Table 4: Results with $|V| = 10000$ and $\alpha = 0.1$

4 Conclusion

In this paper we provided two algorithms to compute a HDR, which is essential to provide a decent prediction area for probability distributions with a non-normal density. Both algorithms use Voronoi and Delaunay techniques to compute a graph with weighted vertices, but differ in the graph algorithm. We tested these algorithms on two cases with unimodal bivariate probability distributions, one with a convex contour shape and one with a non-convex contour shape. In the convex case we could test these algorithms against the theoretical symmetric prediction region, where both algorithms performed quite well. Also for the non-convex contour shaped distribution the algorithms seem to work fine. Most of the time the top-down algorithm is much faster, while the bottom-up algorithm produces a slightly better HDR. As both algorithms are quite accurate, we recommend using the top-down algorithm unless α is high or extra accuracy is needed.

5 Discussion

In this paper we showed some possibilities to compute a HDR using Voronoi, Delaunay and graph techniques. However, there are some ways to improve these algorithms.

First of all, the Delaunay triangulation could be much faster with some optimization. By comparing processor speed with results in [16], one could speed up the Delaunay algorithm with a factor 100. To begin with, they use some kind of bucketing scheme which sorts the vertices in a way the algorithm is much faster in finding the faulty triangles upon insertion.

Furthermore, our algorithms have a problem with boundary vertices. We have to make sure that no boundary vertices are included in the HDR, because their area is set to infinity. This could be a problem when there are not enough vertices or when the α is very low. In our findings we see that the percentage of boundary vertices decreases when the number of vertices increases. This would require some kind of proof, but this means that increasing the number of vertices solves the problem.

Finally, as stated earlier, this is paper shows only some possibilities. In this research we focused solely on the unimodal bivariate case. But as Voronoi and Delaunay techniques can theoretically be used in all number of dimensions higher than one, these techniques could also be used to compute a HDR for multivariate density functions. Computing a HDR for multimodal density functions would require some adjusted graph algorithms, as the HDR might exists out of multiple regions.

References

[1] MS Al-Qassam and JA Lane. Forecasting exponential autoregressive models of order 1. *Journal of Time Series Analysis*, 10(2):95–113, 1989.

[2] R Moeanaddin and Howell Tong. Numerical evaluation of distributions in non-linear autoregression. *Journal of time series analysis*, 11(1):33–48, 1990.

[3] Howell Tong. *Non-linear time series: a dynamical system approach*. Oxford University Press, 1990.

[4] George EP Box and George C Tiao. *Bayesian inference in statistical analysis*. Wiley-Interscience, 1973.

[5] Rob J Hyndman. Highest-density forecast regions for nonlinear and non-normal time series models. *Journal of Forecasting*, 14(5):431–441, 1995.

[6] Rob J Hyndman. Computing and graphing highest density regions. *The American Statistician*, 50(2):120–126, 1996.

[7] Ahmed Fadallah. Highest density regions. Bachelor Thesis, July 2011.

[8] Henrik Zimmer. Voronoi and delaunay techniques. *Proceedings of Lecture Notes, Computer Sciences*, 8, 2005.

[9] Steven Fortune. A sweepline algorithm for voronoi diagrams. *Algorithmica*, 2(1-4):153–174, 1987.

[10] Der-Tsai Lee and Bruce J Schachter. Two algorithms for constructing a delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242, 1980.

[11] Leo J Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 221–234. ACM, 1983.

[12] Rex A Dwyer. A faster divide-and-conquer algorithm for constructing delaunay triangulations. *Algorithmica*, 2(1-4):137–151, 1987.

[13] Adrian Bowyer. Computing dirichlet tessellations. *The Computer Journal*, 24(2):162–166, 1981.

[14] David F Watson. Computing the n-dimensional delaunay tessellation with application to voronoi polytopes. *The computer journal*, 24(2):167–172, 1981.

[15] C. L. Lawson. Triangulation of plane point sets. *Jet Propulsion Laboratory Space Programs Summary*, IV(37-35):24–25, 1965.

[16] Peter Su and Robert L Scot Drysdale. A comparison of sequential delaunay triangulation algorithms. *Computational Geometry*, 7(5):361–385, 1997.

[17] Victor Chew. Confidence, prediction, and tolerance regions for the multivariate normal distribution. *Journal of the American Statistical Association*, 61(315):605–617, 1966.

Appendices

Appendix A Detailed pseudocode

See github.com/JoeriA/HDR for the actual Java code.

A.1 Main procedure

```
procedure MAIN
    generate observations or read from file
    triangulation = delaunay(observations)
    voronoi(triangulation)
    createHDR(triangulation)
end procedure
```

A.2 Delaunay triangulation

```
procedure TRIANGULATE(observations)
    initialize lists of vertices, edges and triangles in triangulation
    for all observations do
        initialize empty bounding rectangle
        if combination x- and y-coordinate does not exist then
            create new vertex
            if vertex is outside bounding rectangle then
                enlarge bounding rectangle
            end if
        else
            merge with existing vertex (increase nrDuplicates)
        end if
    end for
    createSuperTriangle(bounding rectangle)
    sort vertices on x-coordinate                    ▷ speeds up next step
    for all vertices do
        insert(vertex)
    end for
    remove superTriangle
end procedure

function CREATESUPERTRIANGLE(bounding rectangle)
    calculate width and height of rectangle
    first vertex is 1/2 width left of bottom left corner of rectangle
    second vertex is 1/2 width right of bottom right corner of rectangle
    third vertex is 1 height above middle of top edge of rectangle
    create edges of triangle and superTriangle itself
    return superTriangle
end function
```

```
procedure INSERT(vertex v)
    select last created triangle                ▷ higher chance it is located near v
    while toCheck is empty do
        if v is located in circumcircle of selected triangle then
            add triangle to toCheck
        else
            select next triangle
        end if
    end while
    while toCheck is not empty do
        select next triangle t from toCheck
        if v is in circumcirle of t then
            add adjacent triangles of t to toCheck
            remove t from toCheck
            add t to faultyTriangles
            add edges of t to faultyEdges
        else
            add t to okTriangles
        end if
    end while
    remove triangles in faultyTriangles from triangulation
    remove edges from triangulation that are twice in faultyEdges
    for all edges e once in faultyEdges do                ▷ insertion polygon
        for all vertices of e do
            if edge exists between this vertex and v then
                get edge
            else
                create edge from this vertex to v
            end if
        end for
        create triangle with edges and e
    end for
end procedure
```

A.3 Voronoi diagram

```
procedure VORONOI(triangulation)
    for all vertices v in triangulation do
        for all edge e connected to v do
            if e has two adjacent triangles then
                add circumcenters as Voronoi vertices to v
            else
                set v as boundary Voronoi cell
                add circumcenter as Voronoi vertex to v
            end if
        end for
        calculateArea(v)
    end for
end procedure
```

```
procedure calculateArea(v)
    if v contains less than three Voronoi vertices then
        add vertex v itself to Voronoi vertices
    end if
    sort Voronoi vertices in clockwise order
    calculate area of the Voronoi cell
    divide area with (1 + nrDuplicates)        ▷ harder to remove from HDR
    set this as weight of v
end procedure
```

A.4 Graph heuristics

A.4.1 Top-down algorithm

.

```
function TopDown(G = (V, E), nrToRemove)
    sort V on f(v)
    create subgraph H = (W, E′), initially with W = V and E′ = E
    create subgraph I = (U, E”), initially empty
    while nrToRemove > 0 do
        set found to false
        while found is false do
            try next not already tried w ∈ W with highest f(w)
            if w.nrDuplicates < nrToRemove then
                if w is bound then
                    set found to true
                    remove w from H and add to I
                else if checkSwitches(w) is true then
                    set found to true
                    remove w from H and add to I
                end if
            end if
        end while
        nrToRemove = nrToRemove − (1 + w.nrDuplicates)
    end while
    return H
end function
```

A.4.2 Bottom-up algorithm

```
function BottomUp(G = (V, E), nrToAdd)
    sort V on f(v)
    create subgraph H = (W, E′), initially empty
    create subgraph I = (U, E”), initially with U = V and E” = E
    while nrToAdd > 0 do
        set found as false
        while found is false do
```

```
            try next not already tried u ∈ U with lowest f(u)
            if w.nrDuplicates < nrToRemove then
                if |W| is zero then
                    set found to true
                    remove u from I and add to H
                else if checkSwitches(u) is true then
                    set found to true
                    remove u from I and add to H
                end if
            end if
            nrToAdd = nrToAdd − (1 + w.nrDuplicates)
        end while
    end while
    return H
end function
```

A.4.3 Check connected subgraphs

```
function checkSwitches(vertex v)
    sort neighbors of v in clockwise order
    initialize switches is zero
    for all neighbors do
        if next neighbor is not in same state then    ▷ state: in/outside HDR
            increase switches with 1
        end if
    end for
    if switches is 2 then
        return true
    else
        return false
    end if
end function
```