We present a randomized algorithm for the approximate nearest neighbor problem in d-dimensional Euclidean space. Given N points $\{\boldsymbol{x}_j\}$ in $\mathbb{R}^d$, the algorithm attempts to find k nearest neighbors for each of $\boldsymbol{x}_j$, where k is a user-specified integer parameter. The algorithm is iterative, and its CPU time requirements are proportional to $T \cdot N \cdot (d \cdot (\log d) + k \cdot (d + \log k) \cdot (\log N)) + N \cdot k^2 \cdot (d + \log k)$, with T the number of iterations performed. The memory requirements of the procedure are of the order $N \cdot (d + k)$.
A byproduct of the scheme is a data structure, permitting a rapid search for the k nearest neighbors among $\{\boldsymbol{x}_j\}$ for an arbitrary point $\boldsymbol{x} \in \mathbb{R}^d$. The cost of each such query is proportional to $T \cdot (d \cdot (\log d) + \log(N/k) \cdot k \cdot (d + \log k))$, and the memory requirements for the requisite data structure are of the order $N \cdot (d + k) + T \cdot (d + N)$.
The algorithm utilizes random rotations and a basic divide-and-conquer scheme, followed by a local graph search. We analyze the scheme's behavior for certain types of distributions of $\{\boldsymbol{x}_j\}$, and illustrate its performance via several numerical examples.

A Randomized Approximate Nearest Neighbors Algorithm - a short version

Peter W. Jones[†], Andrei Osipov[‡], Vladimir Rokhlin[⋆]
Research Report YALEU/DCS/TR-1439
Yale University
January 13, 2011

[†] This author's research was supported in part by the DMS grant #0602635 and the ONR grants #N000140910108, #N000140910340; [‡] this author's research was supported in part by the AFOSR grant #FA9550-09-1-02-41; [⋆] this author's research was supported in part by the ONR grant #N00014-10-1-0570 and the AFOSR grant #FA9550-09-1-02-41.

Approved for public release: distribution is unlimited.
Keywords: *Approximate nearest neighbors, randomized algorithms, fast random rotations*

1 Introduction

In this paper, we describe an algorithm for finding approximate nearest neighbors (ANN) in d-dimensional Euclidean space for each of N user-specified points $\{\boldsymbol{x}_j\}$. For each point $\boldsymbol{x}_j$, the scheme produces a list of k "suspects", that have high probability of being the k closest points (nearest neighbors) in the Euclidean metric. Those of the "suspects" that are not among the "true" nearest neighbors, are close to being so.

We present several measures of performance (in terms of statistics of the k chosen suspected nearest neighbors), for different types of randomly generated data sets consisting of N points in $\mathbb{R}^d$. Unlike other ANN algorithms that have been recently proposed (see e.g. [1]), the method of this paper does not use locality-sensitive hashing. Instead we use a simple randomized divide-and-conquer approach. The basic algorithm is iterated several times, and then followed by a local graph search.

The performance of any fast ANN algorithm must deteriorate as the dimension d increases. While the running time of our algorithm only grows as $d \cdot \log d$, the statistics of the selected approximate nearest neighbors deteriorate as the dimension d increases. We provide bounds for this deterioration (both analytically and empirically), which occurs reasonably slowly as d increases. While the actual estimates are fairly complicated, it is reasonable to say that in 20 dimensions the scheme performs extremely well, and the performance does not seriously deteriorate until d is approximately 60. At $d = 100$, the degradation of the statistics displayed by the algorithm is quite noticeable.

An outline of our algorithm is as follows:

1. Choose a random rotation, acting on $\mathbb{R}^d$, and rotate the N given points.

2. Take the first coordinate, and divide the data set into two boxes, where the boxes are divided by finding the median in the first coordinate.

3. On each box from Step 2, we repeat the subdivision on the second coordinate, obtaining four boxes in total.

4. We repeat this on coordinates $3, 4$, etc., until each of the boxes has approximately k points.

5. We do a local search on the tree of boxes to obtain approximately k "suspects", for each point $\boldsymbol{x}_j$.

6. The above procedure is iterated T times, and for each point $\boldsymbol{x}_j$, we select from the $T \cdot k$ "suspects" the k closest discovered points for $\boldsymbol{x}_j$.

7. Perform a local graph search on the collections of suspects, obtained in Step 6 (we call this local graph search "supercharging"). Among k^2 "candidates" obtained from the local graph search, we select the best k points and declare these "the suspected approximate nearest neighbors", or "suspects".

The data structure generated by this algorithm allows one to find, for a new data point $\boldsymbol{y}$, the k suspected approximate nearest neighbors in the original dataset. This search is quite rapid, as we need only follow the already generated tree structure of the boxes, obtained in

the steps listed above. One can easily see that the depth of the binary tree, generated by Steps 1 through 4, is $\log_2(N/k)$. This means that we can use the T trees generated, and then pass to Step 7.

Almost all known techniques for solving ANN problems use tree structures (see e.g. [1], [2], [3]). Two apparently novel features of our method are the use of fast random rotations (Step 1), and the local graph search (Step 7), which dramatically increases the accuracy of the scheme. We use the Fast Fourier Transform to generate our random rotations, and this accounts for the factor of $\log d$ that appears in the running time. Our use of random rotations replaces the usual projection argument used in other ANN algorithms, where one projects the data on a random subspace. As far as we know, the use of fast rotations for applications of this type appears first in [3] (see [4] and the references therein for a brief history). The use of random rotations (as in our paper) or random projections (as used elsewhere in ANN algorithms) takes advantage of the same underlying phenomenon; namely the Johnson-Lindenstrauss Lemma. (The JL Lemma roughly states that projection of N points on a random subspace of dimension $C(\varepsilon) \cdot (\log N)$ has expected distortion $1 + \varepsilon$, see e.g. [5].) We have chosen to use random rotations in place of the usual random projections generated by selecting random Gaussian vectors. The fast random rotations require $O(d \cdot (\log d))$ operations, which is an improvement over methods using random projections (see [6], [7]).

The $N \times k$ lookup table arising in Step 7 is the adjacency matrix of a graph whose vertices are the points $\{\boldsymbol{x}_j\}$. In Step 7 we perform a depth one search on this graph, and obtain $\leq k + k^2$ "candidates" (of whom we select the "suspects"). This accounts for the factor of k^2 in the running time. Due to degradation of the running time, we have chosen not to perform searches of depth greater than one.

The algorithm has been tested on a number of artificially generated point distributions. Results of some of those tests are presented below.

The paper is organized as follows. In the first section, we summarize the mathematical and numerical facts to be used in subsequent sections. In the second section, we describe the Randomized Approximate Nearest Neighbors algorithm (RANN) and analyze its cost and performance. In the third section, we illustrate the performance of the algorithm with several numerical examples.

2 Mathematical Preliminaries

In this section, we introduce notation and summarize several facts to be used in the rest of the paper.

2.1 Degree of contact

Suppose that $d \geq L > 0$ are positive integers. Suppose further that

$$\boldsymbol{\sigma} = \sigma_1 \dots \sigma_L, \quad \boldsymbol{\mu} = \mu_1 \dots \mu_L, \quad \sigma_i, \mu_j \in \{+, -\} \tag{1}$$

are two words of symbols $+, -$ of length L. We define the degree of contact $Con(\boldsymbol{\sigma}, \boldsymbol{\mu})$ between $\boldsymbol{\sigma}$ and $\boldsymbol{\mu}$ to be the number of positions at which the corresponding symbols are

different. In other words,

$$Con(\boldsymbol{\sigma}, \boldsymbol{\mu}) = |\{i \ : \ 1 \le i \le L, \quad \sigma_i \ne \mu_i\}| \,. \tag{2}$$

In a mild abuse of notation, we say that two disjoint sets $A_{\boldsymbol{\sigma}}$ and $A_{\boldsymbol{\mu}}$ (or their elements) have degree of contact j if $Con(\boldsymbol{\sigma}, \boldsymbol{\mu}) = j$. For example, x and y have degree of contact 1 if $x \in A_{\boldsymbol{\sigma}}$, $y \in A_{\boldsymbol{\mu}}$ and $\boldsymbol{\sigma}, \boldsymbol{\mu}$ differ at precisely one symbol.

2.2 Pseudorandom orthogonal transformations

In this section, we describe a fast method (presented in [6], [7]) for the generation of random orthogonal transformations and their application to arbitrary vectors.

Suppose that $d, M_1, M_2 > 0$ are positive integers. We define a pseudorandom d-dimensional orthogonal transformation Θ as a composition of $M_1 + M_2 + 1$ linear operators

$$\Theta = \left(\prod_{j=1}^{M_1} Q_j^{(d)} P_j^{(d)} \right) \cdot F^{(d)} \cdot \left(\prod_{j=M_1+1}^{M_1+M_2} Q_j^{(d)} P_j^{(d)} \right) . \tag{3}$$

The linear operators $P_j^{(d)} : \mathbb{R}^d \to \mathbb{R}^d$ with $j = 1, \dots, M_1 + M_2$ are defined in the following manner. We generate permutations $\pi_1, \dots, \pi_{M_1+M_2}$ of the numbers $\{1, \dots, d\}$, uniformly at random and independent of each other. Then for all $\boldsymbol{x} \in \mathbb{R}^d$, we define $P_j^{(d)} \boldsymbol{x}$ by the formula

$$\left(P_j^{(d)} \boldsymbol{x} \right)(i) = x(\pi_j(i)), \quad i = 1, \dots, d. \tag{4}$$

In other words, $P_j^{(d)}$ permutes the coordinates of the vector $\boldsymbol{x}$ according to π_j. $P_j^{(d)}$ can be represented by a $d \times d$ matrix P_j, defined by the formula

$$P_j(k, l) = \begin{cases} 1 & l = \pi_j(k), \\ 0 & l \ne \pi_j(k), \end{cases} \tag{5}$$

for $k, l = 1, \dots, d$. Obviously, the operators $P_j^{(d)}$ are orthogonal.

The linear operators $Q_j^{(d)} : \mathbb{R}^d \to \mathbb{R}^d$ with $j = 1, \dots, M_1 + M_2$ are defined as follows. We construct $(d-1) \cdot (M_1 + M_2)$ independent pseudorandom numbers, $\theta_j(1), \dots, \theta_j(d-1)$ with $j = 1, \dots, M_1 + M_2$, uniformly distributed in $(0, 2\pi)$. Then we define the auxiliary linear operator $Q_{j,k} : \mathbb{R}^d \to \mathbb{R}^d$ for $k = 1, \dots, d-1$ to be the planar rotation of the coordinates k and $k+1$ by the angle $\theta_j(k)$. In other words, for all $\boldsymbol{x} \in \mathbb{R}^d$

$$(Q_{j,k}(\boldsymbol{x})) \begin{pmatrix} k \\ k+1 \end{pmatrix} = \begin{pmatrix} \cos(\theta_j(k)) & \sin(\theta_j(k)) \\ -\sin(\theta_j(k)) & \cos(\theta_j(k)) \end{pmatrix} \cdot \begin{pmatrix} x(k) \\ x(k+1) \end{pmatrix} , \tag{6}$$

and the rest of the coordinates of $Q_{j,k}(\boldsymbol{x})$ coincide with those of $\boldsymbol{x}$. We define $Q_j^{(d)}$ by the formula

$$Q_j^{(d)} = Q_{j,d-1} \cdot Q_{j,d-2} \cdot \dots \cdot Q_{j,1}. \tag{7}$$

Obviously, the operators $Q_j^{(d)}$ are orthogonal.

The linear operator $F^{(d)} : \mathbb{R}^d \to \mathbb{R}^d$ is defined as follows. First suppose that d is even and that $d_2 = d/2$. We define the $d_2 \times d_2$ discrete Fourier transform matrix T by the formula

$$T(k,l) = \frac{1}{\sqrt{d_2}} \cdot \exp\left[-\frac{2\pi i(k-1)(l-1)}{d_2}\right], \tag{8}$$

where $k, l = 1, \ldots, d_2$ and $i = \sqrt{-1}$. The matrix T represents a unitary operator $\mathbb{C}^{d_2} \to \mathbb{C}^{d_2}$. We then define the one-to-one linear operator $Z : \mathbb{R}^d \to \mathbb{C}^{d_2}$ by the formula

$$Z\boldsymbol{x} = \begin{pmatrix} x(1) + i \cdot x(2), \\ x(3) + i \cdot x(4), \\ \cdots \\ x(2d_2 - 1) + i \cdot x(2d_2) \end{pmatrix} \tag{9}$$

for all $\boldsymbol{x} \in \mathbb{R}^d$. Eventually, we define $F^{(d)}$ by the formula

$$F^{(d)} = Z^{-1} \cdot T \cdot Z \tag{10}$$

for even d. If d is odd, we define $F^{(d)}\boldsymbol{x}$ for all $\boldsymbol{x} \in \mathbb{R}^d$ by applying $F^{(d-1)}$ to the first $d-1$ coordinates of $\boldsymbol{x}$ and leaving its last coordinate unchanged. Obviously, the operators T, Z, defined by (8), (9), respectively, preserve the norm of any vector $\boldsymbol{x} \in \mathbb{R}^d$. Therefore, $F^{(d)}$ is a real orthogonal transformation $\mathbb{R}^d \to \mathbb{R}^d$.

The cost of the generation of a random permutation (see e.g. [8]) is $O(d)$ operations. The cost of the application of each $P_j^{(d)}$ to a vector $\boldsymbol{x} \in \mathbb{R}^d$ is obviously d operations due to (4).

The cost of generation of $d-1$ uniform random variables is $O(d)$ operations. Also, the cost of application of each $Q_j^{(d)}$ to a vector $\boldsymbol{x} \in \mathbb{R}^d$ is $O(d)$ operations due to (7).

Finally, the cost of the fast discrete Fourier transform is $O(d \cdot \log d)$ operations, and the cost of the application of $F^{(d)}$ to a vector $\boldsymbol{x} \in \mathbb{R}^d$ is $O(d \cdot \log d)$ operations due to (8), (9) and (10).

Thus the cost of the generation of Θ defined via (3) is

$$\text{Cost}(\Theta) = O\left(d \cdot (M_1 + M_2 + \log d)\right). \tag{11}$$

Moreover, the cost of application of Θ to a vector $\boldsymbol{x} \in \mathbb{R}^d$ is also given by the formula (11).

Remark 1. *It was observed that if M_1 and M_2 in* (3) *are chosen such that $M_1 + M_2 \sim \log d$, then the distribution of Θ is close to the uniform distribution on the group $O(d, \mathbb{R})$ of orthogonal transformations from $\mathbb{R}^d$ to $\mathbb{R}^d$.*

Remark 2. *The use of the Hadamard matrix (without 2×2 rotations) appears in a related problem studied by Ailon and Liberty [9].*

3 Randomized Approximate Nearest Neighbors algorithm

In this section, we describe the Nearest Neighbor Problem and present a fast randomized algorithm for its solution.

3.1 The Nearest Neighbor Problem

Suppose that d and $k < N$ are positive integers and suppose that

$$B = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_N\} \subseteq \mathbb{R}^d \tag{12}$$

is a collection of N points in $\mathbb{R}^d$. We are interested in finding the k nearest neighbors of each point $\boldsymbol{x}_i$. The distance between the points is the standard Euclidean distance.

For each $\boldsymbol{x}_i$, one can compute in a straightforward manner the distances to the rest of the points and thus find the nearest neighbors. However, the total cost of the evaluation of the distances alone is $O(d \cdot N^2)$, which makes this naive approach prohibitively expensive when N is large. We propose a faster approximate algorithm for the solution of this problem.

3.2 Informal description of the algorithm

3.2.1 Initial selection

The key idea of our algorithm is the following simple (and well known) observation. Suppose that for each $\boldsymbol{x}_i$ we have found a small subset V_i of B such that a point inside V_i is more likely to be among the k nearest neighbors of $\boldsymbol{x}_i$ than a point outside V_i. Then it is reasonable to look for the nearest neighbors of each $\boldsymbol{x}_i$ only inside V_i and not among all the points. The nearest neighbors of $\boldsymbol{x}_i$ in V_i, which can be found by direct scanning, are referred to as its "suspected approximate nearest neighbors", or "suspects", as opposed to the true nearest neighbors $\left\{\boldsymbol{x}_{t(i,j)}\right\}$.

Of course, many of the k true nearest neighbors of $\boldsymbol{x}_i$ might not be among its suspects. However, one can re-select V_i to obtain another list of k suspects of $\boldsymbol{x}_i$. The initial guess is improved by taking the "best" k points out of the two lists. This scheme is iterated to successively improve the list of suspects of each $\boldsymbol{x}_i$.

The performance of the resulting iterative randomized algorithm admits the following crude analysis. Suppose that the size of V_i is $\alpha \cdot N$, with $\alpha \ll 1$. Suppose also that the number of the true nearest neighbors of $\boldsymbol{x}_i$ inside V_i is roughly $\beta \cdot k$, with $\alpha < \beta < 1$. If the choice of V_i is fairly random, then order $O(1/\beta)$ iterations of the algorithm are required to find most of the true nearest neighbors of each $\boldsymbol{x}_i$. Temporarily neglecting the cost of the construction of V_i, this results in $O\left((\alpha/\beta) \cdot d \cdot N^2\right)$ operations instead of $O\left(d \cdot N^2\right)$ operations for the naive algorithm. If $\alpha \ll \beta$, the improvement can be substantial.

Our construction of V_i's is based on geometric considerations. First, we shift all of the points to place their center of mass at the origin and apply a random orthogonal linear transformation on the resulting collection. Then, we choose a real number $y(1)$ such that the first coordinate of half of the points in B is less than $y(1)$. In other words,

$$|\{\boldsymbol{x} \in B : x(1) < y(1)\}| = \lfloor N/2 \rfloor. \tag{13}$$

We divide all the points into two disjoint sets

$$\begin{aligned} B_- &= \{\boldsymbol{x} \in B : x(1) < y(1)\}, \\ B_+ &= \{\boldsymbol{x} \in B : x(1) \geq y(1)\}. \end{aligned} \tag{14}$$

Obviously, the sizes of B_- and B_+ are the same if N is even and differ by one if N is odd. Next, we set $y_+(2)$ to be a real number such that the second coordinate of half of the points in B_+ is less than $y_+(2)$, i.e.

$$|\{\boldsymbol{x} \in B_+ : x(2) < y_+(2)\}| = \lfloor N/4 \rfloor. \tag{15}$$

We split B_+ into two disjoint sets B_{+-} and B_{++} by the same principle, e.g.

$$\begin{aligned} B_{+-} &= \{\boldsymbol{x} \in B_+ : x(2) < y_+(2)\}\,, \\ B_{++} &= \{\boldsymbol{x} \in B_+ : x(2) \geq y_+(2)\}\,. \end{aligned} \tag{16}$$

We construct B_{--} and B_{-+} in a similar fashion by using a real number $y_-(2)$ such that the second coordinate of half of the points in B_- is less than $y_-(2)$, i.e.

$$|\{\boldsymbol{x} \in B_- : x(2) < y_-(2)\}| = \lfloor N/4 \rfloor. \tag{17}$$

Each of the four boxes $B_{--}, B_{-+}, B_{+-}, B_{++}$ contains either $\lfloor N/4 \rfloor$ or $\lfloor N/4 \rfloor + 1$ points. Then we repeat the subdivision by splitting each of the four boxes into two by using the third coordinate, and so on. We proceed until we end up with a collection of 2^L boxes $\{B_{\boldsymbol{\sigma}}\}$ with k or $k+1$ points in each box. Here the box index $\boldsymbol{\sigma}$ is a word of symbols $+, -$ of length L as in (1) and L is a positive integer such that $k \cdot 2^L \leq N < k \cdot 2^{L+1}$. In other words, L is defined via the inequality

$$k \cdot 2^L \leq N < k \cdot 2^{L+1}. \tag{18}$$

Obviously, the sets $\{B_{\boldsymbol{\mu}}\}$ constitute a complete binary tree of length L, whose nodes are indexed by words $\boldsymbol{\mu}$ of symbols $+, -$ of length up to L. The set B is at the root of this tree, the sets B_- and B_+ are at the second level, and so on. At the last level of the tree, there are 2^L boxes. The depth of the tree equals to $L+1$.

The notion of degree of contact (2) extends to the collection $\{B_{\boldsymbol{\sigma}}\}$ of boxes. Suppose that $\boldsymbol{x}_i$ is in $B_{\boldsymbol{\sigma}}$. Obviously, the higher degree of contact of two boxes $B_{\boldsymbol{\sigma}}$ and $B_{\boldsymbol{\mu}}$ is, the less likely a point of $B_{\boldsymbol{\mu}}$ will be among the k nearest neighbors of $\boldsymbol{x}_i$. Motivated by this observation, we define the set V_i as

$$V_i = \left\{\boldsymbol{x} \in B_{\boldsymbol{\mu}} \;:\; Con(\boldsymbol{\sigma}, \boldsymbol{\mu}) \leq 1\right\}. \tag{19}$$

In other words, V_i is the union of the box $B_{\boldsymbol{\sigma}}$ containing $\boldsymbol{x}_i$ and L boxes whose degree of contact with $B_{\boldsymbol{\sigma}}$ is one. Thus for each $i = 1, \ldots, N$, the set V_i contains about $k \cdot (L+1)$ points.

3.2.2 "Supercharging"

In the previous subsections, we have described an iterative scheme for the selection of suspects (suspected approximate nearest neighbors) for each of the points $\boldsymbol{x}_i$ in B. Suppose now that after T iterations of this scheme, the list $\boldsymbol{x}_{s(i,1)}, \ldots, \boldsymbol{x}_{s(1,k)}$ of k suspects of each point $\boldsymbol{x}_i$ has been generated. This list can be improved by a procedure we call supercharging.

The idea of supercharging is based on the following observation. A true nearest neighbor of $\boldsymbol{x}_i$, missed by the scheme described above, might be among the suspects of one of $\boldsymbol{x}_{s(i,1)}, \ldots, \boldsymbol{x}_{s(1,k)}$. This leads to the following obvious procedure.

For each $\boldsymbol{x}_i$, we denote by A_i the list of suspects of all $\boldsymbol{x}_{s(i,1)}, \ldots, \boldsymbol{x}_{s(i,k)}$. A_i contains k^2 points, with possible repetitions. We compute the square of the distances from $\boldsymbol{x}_i$ to each point in A_i and find the k nearest neighbors $\boldsymbol{x}_{t(i,1,A_i)}, \ldots, \boldsymbol{x}_{t(i,k,A_i)}$ of $\boldsymbol{x}_i$ in A_i. Then we declare the (updated) suspects of $\boldsymbol{x}_i$ to be the best k points out of the two lists $\left\{\boldsymbol{x}_{s(i,j)}\right\}_{j=1}^{k}$ and $\left\{\boldsymbol{x}_{t(i,j,A_i)}\right\}_{j=1}^{k}$.

In other words, supercharging is a depth one search on the directed graph, whose vertices are the points $\{\boldsymbol{x}_i\}$ and whose $N \times k$ adjacency matrix is the suspects' indices $\{s(i,j)\}$, with $i = 1, \ldots, N$ and $j = 1, \ldots, k$.

3.2.3 Overview

We conclude this section with a list of the principal steps of the algorithm. Given the collection $\{\boldsymbol{x}_i\}_{i=1}^{N}$ of points in $\mathbb{R}^d$, we perform the following operations.

1. Subtract from each $\boldsymbol{x}_i$ the center of mass of the collection.
2. Choose a random orthogonal linear transformation Θ and set $\boldsymbol{x}_i = \Theta(\boldsymbol{x}_i)$ for all $i = 1, \ldots, N$.
3. Construct 2^L boxes $\{B_{\boldsymbol{\sigma}}\}$ as described above.
4. For each $\boldsymbol{x}_i$ define the set V_i via (19).
5. Update the suspects $\boldsymbol{x}_{s(i,1)}, \ldots, \boldsymbol{x}_{s(i,k)}$ of $\boldsymbol{x}_i$ by using its true nearest neighbors in V_i.
6. Steps 2-5 are repeated T times.
7. For each $\boldsymbol{x}_i$, perform supercharging.

3.2.4 Query for a new point

Suppose that we are given a new point $\boldsymbol{y} \in \mathbb{R}^d$, and we need to find its k nearest neighbors in $B = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$. In this subsection, we describe a rapid procedure to find k approximate nearest neighbors of $\boldsymbol{y}$. This procedure uses the following information, available on the jth iteration of the algorithm, for $j = 1, \ldots, T$:

1. The orthogonal linear transformation $\Theta^{(j)}$, generated on the jth iteration of the algorithm .
2. The collection of boxes $\left\{B_{\boldsymbol{\sigma}}^{(j)}\right\}$, generated on the jth iteration of the algorithm.

To find k approximate nearest neighbors of the new point $\boldsymbol{y}$ among the points of B, we perform the following operations. First, we apply $\Theta^{(1)}$ on $\boldsymbol{y}$, where $\Theta^{(1)}$ is the orthogonal linear transformation of the first iteration of the algorithm. The resulting vector is denoted by $\boldsymbol{y}^{(1)}$, in other words,

$$\boldsymbol{y}^{(1)} = \Theta^{(1)}(\boldsymbol{y}). \tag{20}$$

Next, in the collection of boxes $\left\{B_{\boldsymbol{\sigma}}^{(1)}\right\}$, generated on the first iteration of the algorithm, we find the box $B_{\boldsymbol{\sigma}(1)}^{(1)}$ that has degree of contact zero with $\boldsymbol{y}^{(1)}$. Note that if $\boldsymbol{y}$ had belonged to B in the first place, then on the first iteration of the algorithm $\boldsymbol{y}^{(1)}$ would have belonged to $B_{\boldsymbol{\sigma}(1)}^{(1)}$.

The box $B_{\boldsymbol{\sigma}(1)}^{(1)}$ has k or $k+1$ points. Suppose that $V_{\boldsymbol{\sigma}(1)}^{(1)}$ is is the union of the boxes having degree of contact zero or one with $B_{\boldsymbol{\sigma}(1)}^{(1)}$ (see (19)). Recall that $V_{\boldsymbol{\sigma}(1)}^{(1)}$ has about $(L+1)\cdot k$ points, where L is defined via (18).

We define the set $V_{\boldsymbol{\sigma}(2)}^{(2)}$ in a similar manner, by using the data of the second iteration of the algorithm. We apply the orthogonal transformation $\Theta^{(2)}$ of the second iteration on $\boldsymbol{y}^{(1)}$ to obtain $\boldsymbol{y}^{(2)}$, i.e.

$$\boldsymbol{y}^{(2)} = \Theta^{(2)}\left(\boldsymbol{y}^{(1)}\right) = \Theta^{(2)}\left(\Theta^{(1)}\left(\boldsymbol{y}\right)\right), \tag{21}$$

due to (20). In the boxes $\left\{B_{\boldsymbol{\sigma}}^{(2)}\right\}$ of the second iteration, we find the box $B_{\boldsymbol{\sigma}(2)}^{(2)}$, having degree of contact zero with $\boldsymbol{y}^{(2)}$. Similar to $V_{\boldsymbol{\sigma}(1)}^{(1)}$, the set $V_{\boldsymbol{\sigma}(2)}^{(2)}$ has about $(L+1)\cdot k$ points.

We repeat this procedure to construct the sets $V_{\boldsymbol{\sigma}(j)}^{(j)}$ for $j=3,4,\ldots,T$, where T is the number of the iterations of the algorithm. Each $V_{\boldsymbol{\sigma}(j)}^{(j)}$ contains roughly $(L+1)\cdot k$ points.

Finally, we define the set A to be the union of all the sets $V_{\boldsymbol{\sigma}(j)}^{(j)}$, in other words,

$$A = \bigcup_{j=1}^{T} V_{\boldsymbol{\sigma}(j)}^{(j)}. \tag{22}$$

The set A contains about $T\cdot k\cdot(L+1)$ points. The k nearest neighbors of $\boldsymbol{y}$ inside A are declared to be the approximate nearest neighbors of $\boldsymbol{y}$ inside B. We note that to construct A we need to store the corresponding data on each iteration of the algorithm.

Once the k suspects of $\boldsymbol{y}$ in B have been found, they can be improved by performing supercharging on $\boldsymbol{y}$ only.

3.3 Cost analysis

In this subsection, we analyze the cost of the algorithm in terms of number of operations. Also, we analyze the memory requirements of the algorithm. We recall that $\boldsymbol{x}_1,\ldots,\boldsymbol{x}_N$ is a collection of N points in $\mathbb{R}^d$ and $N\approx k\cdot 2^L$.

1. Centralizing the points costs $O(N\cdot d)$.
2. Generating a pseudorandom orthogonal transformation and applying it to N points costs $O(N\cdot d\cdot(\log d))$ (see (11)).
3. Constructing the binary tree of boxes of depth $L=\log_2(N/k)$ costs $O(N\cdot L)$.
4. The suspects of each point $\boldsymbol{x}_i$ are selected by computing the distances from $\boldsymbol{x}_i$ to $k\cdot(L+1)$ points in V_i (see (19)) and finding k minimal distances. Thus the cost of this step is $O(N\cdot L\cdot k\cdot(d+\log k))$.

5. In supercharging, k^2 points are scanned to update the suspects of each point. Thus the cost of supercharging is $O(N \cdot k^2 \cdot (d + \log k))$.

We conclude that the total cost of the algorithm is

$$\begin{aligned} C_{\text{RANN}} = O\left(T \cdot N \cdot (d \cdot (\log d) + k \cdot (d + \log k) \cdot (\log N))\right) + \\ O(N \cdot k^2 \cdot (d + \log k)), \end{aligned} \tag{23}$$

where T is the number of iterations. We observe that for fixed dimension d and number of required nearest neighbors k, the cost is $O(T \cdot N \cdot \log N)$, as opposed to $O(N^2)$ of the naive approach. Also, the cost of supercharging is quadratic in the number of nearest neighbors for fixed dimension d and number of points N, which makes supercharging expensive relative to a single iteration of the principal part of the algorithm even for moderate k.

Query for a new point $\boldsymbol{y}$ consists of performing all the steps of the algorithm on one point only. Consequently, due to (23) the total cost of query for a new point is

$$\begin{aligned} C_{\text{query}} = O\left(T \cdot (d \cdot (\log d) + k \cdot (d + \log k) \cdot (\log N))\right) + \\ O(k^2 \cdot (d + \log k)). \end{aligned} \tag{24}$$

To determine the memory requirements of the algorithm, we observe that to store N points in $\mathbb{R}^d$ we need $O(N \cdot d)$ memory, to store the indices of k nearest neighbors of each of N points we need $O(N \cdot k)$ memory, and to store a tree of boxes (with the corresponding orthogonal transformation) we need $O(N)$ memory.

We must distinguish between two cases. In the first case, given N points $\{\boldsymbol{x}_i\}$ in $\mathbb{R}^d$, we are interested in finding k nearest neighbors for each $\boldsymbol{x}_i$ only. In other words, no query for a new point will ever be requested. Then, the tree of boxes can be destroyed after each iteration. Hence, in this case the algorithm uses $O(N \cdot (d + k))$ memory. In other words, in this case the memory requirements are minimal, in the sense that most of the memory is spent on the storage of input and output of the algorithm only.

In the second case, we know in advance that queries for new points will be requested. Therefore we need to store T trees of boxes and the corresponding orthogonal transformations. Thus, when queries for new points are allowed, the total memory requirements are of the order

$$M_{\text{query}} = O\left(N \cdot (d + k + T)\right). \tag{25}$$

Remark 3. *The factor* $\log k$ *in* (23), (24) *can be omitted, if the suspects of every* $\boldsymbol{x}_i$ *are not required to be sorted according to their distance to* $\boldsymbol{x}_i$*.*

3.4 Performance analysis

In this subsection, we discuss the performance analysis of the Randomized Approximate Nearest Neighbor algorithm in the case of standard Gaussian distribution of the points.

To be more specific, we consider the collection of N independent standard Gaussian d-dimensional random vectors $\boldsymbol{x}_1, \dots, \boldsymbol{x}_N$, where the number of points is given by the formula

$$N = k \cdot 2^L \tag{26}$$

for some positive integer $L > 0$. We recall that for each point $\boldsymbol{x}_i$, the algorithm approximates the k true nearest neighbors $\{\boldsymbol{x}_{t(i,j)}\}_{j=1}^{k}$ of $\boldsymbol{x}_i$ by k suspects $\{\boldsymbol{x}_{s(i,j)}\}_{j=1}^{k}$. In order to analyze the quality of this approximation, we introduce a number of statistical quantities. First, we define the average square of the distance from $\boldsymbol{x}_i$ to its k true nearest neighbors by the formula

$$D_i^{true} = \frac{1}{k}\sum_{j=1}^{k} \|\boldsymbol{x}_i - \boldsymbol{x}_{t(i,j)}\|^2. \tag{27}$$

Next, we define the average square of the distance from $\boldsymbol{x}_i$ to its k suspects by the formula

$$D_i^{susp} = \frac{1}{k}\sum_{j=1}^{k} \|\boldsymbol{x}_i - \boldsymbol{x}_{s(i,j)}\|^2. \tag{28}$$

Finally, we define the proportion of the true nearest neighbors of $\boldsymbol{x}_i$ among its suspects by the formula

$$\text{prop}_i = \frac{1}{k}\left|\{\boldsymbol{x}_{t(i,j)}\}_{j=1}^{k} \cap \{\boldsymbol{x}_{s(i,j)}\}_{j=1}^{k}\right|. \tag{29}$$

We analyze a single iteration of the algorithm (selection of suspects without supercharging) by evaluating the expected values of (27), (28), (29) numerically (see Theorems 1, 2, 3 below). On the other hand, the performance of the algorithm can be studied empirically, by running the algorithm on artificially generated sets of points.

In the rest of this subsection, we outline the results of the analysis (see [10] for details). We start with introducing some notation.

Suppose that $d > 0$ is a positive integer, and $\lambda, \alpha, \beta > 0$ are positive real numbers. The distributions χ^2_d, $\chi^2(d,\lambda)$ and $B(\alpha,\beta)$ are defined by their probability density functions, given respectively via the formulas

$$f_{\chi^2_d}(t) = \frac{t^{d/2-1}\cdot e^{-t/2}}{2^{d/2}\cdot\Gamma(d/2)}, \qquad t > 0, \tag{30}$$

$$f_{\chi^2(d,\lambda)}(t) = e^{-\lambda/2}\sum_{j=0}^{\infty}\frac{(\lambda/2)^j}{j!} f_{\chi^2_{d+2j}}(t), \qquad t > 0, \tag{31}$$

$$f_{B(\alpha,\beta)}(t) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}\cdot t^{\alpha-1}(1-t)^{\beta-1}, \qquad 0 < t < 1. \tag{32}$$

Suppose that $a > 0$ is a positive real number. Suppose also that $d \ge L > 0$ are positive integers, and $\boldsymbol{a} \in \mathbb{R}^d$ is a vector all of whose coordinates are positive. We define the functions

$h_a^-, h_a^+, h_{\boldsymbol{a}} : \mathbb{R} \to \mathbb{C}$ via the formulas

$$h_a^-(x) = \exp\left[\frac{ia^2x}{1-2ix}\right] \cdot \operatorname{erfc}\left[-\frac{iax\sqrt{2}}{\sqrt{1-2ix}}\right] \cdot \frac{1}{\sqrt{1-2ix}}, \tag{33}$$

$$h_a^+(x) = \exp\left[\frac{ia^2x}{1-2ix}\right] \cdot \frac{2}{\sqrt{1-2ix}} - h_a^-(x), \tag{34}$$

$$h_{\boldsymbol{a}}(x) = \exp\left[\frac{ix}{1-2ix} \cdot \sum_{j=L+1}^{d} a(j)^2\right] \cdot \left(\frac{1}{\sqrt{1-2ix}}\right)^{d-L} \cdot \left(\prod_{j=1}^{L} h_{a(j)}^+(x) + \sum_{i=1}^{L} h_{a(i)}^-(x) \cdot \prod_{\substack{j=1\\ j\neq i}}^{L} h_{a(j)}^+(x)\right). \tag{35}$$

Also, we define the function $G_{\boldsymbol{a}} : (0,\infty) \to \mathbb{R}$ via the formula

$$G_{\boldsymbol{a}}(y) = \frac{1}{2\pi \cdot (L+1)} \int_0^y \int_{-\infty}^{\infty} e^{-ixt} \cdot h_{\boldsymbol{a}}(x)\, dx\, dt. \tag{36}$$

The following theorem provides an analytical formula for the expectation $\mathbb{E}\left[D_N^{true}\right]$ (see (27)).

Theorem 1. *Suppose that $d, k, N > 0$ are positive integers. Suppose further that D_N^{true} is defined by* (27)*. Then its expectation is given by the formula*

$$\mathbb{E}\left[D_N^{true}\right] = \int_0^{\infty} \left(\frac{1}{k}\sum_{i=1}^{k} \int_0^1 F_{\chi^2(d,\lambda)}^{-1}(t) \cdot f_{B(i,N-i)}(t)\, dt\right) \cdot f_{\chi_d^2}(\lambda)\, d\lambda, \tag{37}$$

where the functions $f_{\chi_d^2}, f_{B(i,N-i)}$ are defined respectively by (30)*,* (32)*, and $F_{\chi^2(d,\lambda)}^{-1}$ is the inverse of the cdf of $\chi^2(d,\lambda)$ (see* (31)*).*

The following theorem provides an approximation to the expectation $\mathbb{E}\left[D_N^{susp}\right]$ (see (28)). The error of this approximation is verified via numerical experiments (see [10] for more details).

Theorem 2. *Suppose that $k > 0$ and $d \geq L > 0$ are positive integers. We define the real number D_{appr}^{susp} by the formula*

$$D_{appr}^{susp} = \int_0^{\infty} \operatorname*{Avg}_{\boldsymbol{a}\in S_d^+(\lambda)} \left[\frac{1}{k}\sum_{i=1}^{k} G_{\boldsymbol{a}}^{-1}\left(\frac{i}{k\cdot L + k + 1}\right)\right] \cdot f_{\chi_d^2}(\lambda)\, d\lambda, \tag{38}$$

where the function $f_{\chi_d^2}$ is defined via (30)*, the function $G_{\boldsymbol{a}}^{-1}$ is the inverse of $G_{\boldsymbol{a}}$ defined via* (36)*, the set $S_d^+(\lambda)$ is the positive part of the d-dimensional hypersphere of radius $\sqrt{\lambda}$, defined by the formula*

$$S_d^+(\lambda) = \left\{\boldsymbol{x} \in \mathbb{R}^d : \|\boldsymbol{x}\|^2 = \lambda,\ \ x(j) > 0,\ \ 1 \leq j \leq d\right\}, \tag{39}$$

and the average (Avg) is taken with respect to the $(d-1)$-dimensional area. Then,

$$\mathbb{E}\left[D_N^{susp}\right] = D_{appr}^{susp} + O\left(k^{-1/2}\right), \quad k \to \infty. \tag{40}$$

In other words, (40) *holds, if we fix d, L and let $k \to \infty$.*

The following theorem provides an approximation to the expectation $\mathbb{E}\left[\text{prop}_N\right]$ (see (29)). The error of this approximation is verified via numerical experiments (see [10] for more details).

Theorem 3. *Suppose that $k > 0$ and $d \geq L > 0$ are positive integers. We define the real number P_{appr} by the formula*

$$P_{appr} = (L+1) \int_0^\infty \underset{\boldsymbol{a} \in S_d^+(\lambda)}{\text{Avg}} \left[G_{\boldsymbol{a}}\left(F^{-1}_{\chi^2(d,\lambda)}(2^{-L})\right)\right] \cdot f_{\chi^2_d}(\lambda)\, d\lambda, \tag{41}$$

where the function $f_{\chi^2_d}$ is defined via (30)*, the function $F^{-1}_{\chi^2(d,\lambda)}$ is the inverse of the cdf of $\chi^2(d,\lambda)$, defined via* (31)*, the function $G_{\boldsymbol{a}}$ is defined via* (36)*, the set $S_d^+(\lambda)$ is defined via* (39)*, and the average (Avg) is taken with respect to the $(d-1)$-dimensional area. Then,*

$$\mathbb{E}\left[\text{prop}_N\right] = P_{appr} + O\left(k^{-1/2}\right), \tag{42}$$

where the real random variable prop_N *is defined via* (29)*. In other words,* (42) *holds, if we fix d, L and let $k \to \infty$.*

4 Numerical Results

This section has two principal purposes. First, we numerically evaluate the expectations of (27), (28), (29) by using Theorems 1, 2, 3 above. Second, we demonstrate the performance of the algorithm empirically, by running it on sets of points, generated according to the Gaussian distribution. The choice of uniform or Hamming distributions instead of Gaussian results in very similar performance (see [10] for results and details).

The algorithm has been implemented in FORTRAN (Lahey 95 Linux version). The numerical experiments have been carried out on a Lenovo laptop computer, with DualCore CPU 2.53 GHz and 2.9GB RAM.

Experiment 1. In this experiment, we choose $k = 30$, and for $L = 10, 15, 20, 25$ set N via (26). Then, for different values of d between L and 110, we approximate the expectations $\mathbb{E}\left[D_N^{true}\right], \mathbb{E}\left[D_N^{susp}\right], \mathbb{E}\left[\text{prop}_N\right]$ via the numerical evaluation of (37), (38), (41), respectively. The approximations are denoted by D_{num}^{true}, D_{num}^{susp}, P_{num}, respectively, and are accurate to roughly two decimal digits. Also, we compute the ratio

$$\text{Ratio}_{true}^{susp} = \frac{D_{num}^{susp}}{D_{num}^{true}}. \tag{43}$$

The results of Experiment 1 are shown in Figure 1. In Figure 1 (left), we plot $\text{Ratio}_{true}^{susp}$ as a function of the dimensionality d, for each value of L. In Figure 1 (right), we plot P_{num} as a function of the dimensionality d, for each value of L (the number of points N is determined via (26)). This quantity estimates the average proportion of the suspects among the true nearest neighbors (found by one iteration of RANN without supercharging).

Several observations can be made from Experiment 1 (see Figure 1) and from more detailed experiments by the authors.

1. The ratio $\text{Ratio}_{true}^{susp}$, defined via (43), slowly increases with the number of points N, for fixed values of number k of nearest neighbors and dimensionality d (see Figure 1, left). In other words, the performance of RANN deteriorates with the number of points, if terms of $\text{Ratio}_{true}^{susp}$. However, as number of points is multiplied by 2, this ratio grows by a factor less than 1.1 for most values of d on Figure 1 (left).

2. The ratio $\text{Ratio}_{true}^{susp}$ actually decreases with dimensionality d, for fixed k and N (see Figure 1, left). For example, for $N \approx 10^6$, this ratio decays from the value of about 1.57 at $d = 20$ to roughly 1.3 at $d = 110$. This observation is not surprising, since the number of points within the same fixed relative distance (e.g. twice the distance to true nearest neighbors) grows with the dimensionality.

3. The proportion of true nearest neighbors among suspects decays with d for fixed k, N, as expected (see Figure 1, right). However, even for $d = 40$ and $N \approx 10^6$, RANN correctly finds about 2.7% of true nearest neighbors, on merely one iteration without supercharging. In other words, after 50 iterations without supercharging RANN will correctly detect about

$$75\% = 0.75 \approx 1 - (1 - 0.027)^{50} \tag{44}$$

true nearest neighbors, for $d = 40$ and $N \approx 10^6$ (see also Experiment 2 below).

Experiment 2. In this experiment, we run RANN with various parameters on different sets of points and report on the resulting statistics.

We choose the dimensionality d and the number of points N. Then we choose the number of nearest neighbors k, the number of iterations of the algorithm $T > 0$, and whether to perform the supercharging or not. Next, we generate N i.i.d. standard Gaussian vectors $\boldsymbol{x}_1, \dots, \boldsymbol{x}_N$ in $\mathbb{R}^d$.

We run RANN on $\boldsymbol{x}_1, \dots, \boldsymbol{x}_N$. For each $\boldsymbol{x}_i$, RANN finds its k suspects, $\boldsymbol{x}_{s(i,1)}, \dots, \boldsymbol{x}_{s(i,k)}$. Also, for all $i = 1, \dots, 2000$, we find the list $\boldsymbol{x}_{t(i,1)}, \dots, \boldsymbol{x}_{t(i,k)}$ of k true nearest neighbors of $\boldsymbol{x}_i$, by direct scanning. Then, we compute the quantities D_i^{true}, D_i^{susp}, prop_i, defined via (27), (28), (29), respectively, for $i = 1, \dots, 2000$. We define the average D_{algo}^{true} by the formula

$$D_{algo}^{true} = \frac{1}{2000} \sum_{i=1}^{2000} D_i^{true}. \tag{45}$$

The averages D_{algo}^{susp}, prop_{algo} are defined and computed by the same token.

Generation of the points and invocation of RANN are repeated 20 times, to obtain the values $D^{true}_{algo}(1), \ldots, D^{true}_{algo}(20)$, all defined via (45). Then, we define the sample mean of D^{true}_{algo} via the formula

$$\mathbb{E}_{smpl}\left[D^{true}\right] = \frac{1}{20}\sum_{i=1}^{20} D^{true}_{algo}(i). \tag{46}$$

The sample means $\mathbb{E}_{smpl}\left[D_{algo}\right]$ and $\mathbb{E}_{smpl}\left[\text{prop}_{algo}\right]$ are defined in a similar way. Finally, we compute the ratio

$$\text{ratio}_{smpl} = \frac{\mathbb{E}_{smpl}\left[D_{algo}\right]}{\mathbb{E}_{smpl}\left[D^{true}\right]}. \tag{47}$$

The parameters for Experiment 2 were chosen in the following way. The number of points was $N = 30 \cdot 2^{12} \approx 10^5$. The number of requested nearest neighbors was $k = 15, 30$ or 60. The dimensionality d was chosen to be a multiple of 5 between 15 and 200. The number of iterations of the algorithm was either $T = 1$ or $T = 10$. The supercharging step was either skipped or performed once after T iterations.

Most of the approximations have been computed with relative error up to 2%. The results are shown in Figures 2, 3 and in Table 1. In Figures 2, 3 (left), we plot ratio_{smpl} (see (47)) as a function of the dimensionality d for $k = 15$ and $k = 60$, respectively. In Figures 2, 3 (right), we plot $\mathbb{E}_{smpl}\left[\text{prop}_{algo}\right]$ as a function of the dimensionality d for $k = 15$ and $k = 60$, respectively. Each figure contains four curves, corresponding to one iteration without supercharging, one iteration with supercharging, ten iterations without supercharging and ten iterations with supercharging.

Table 1 has the following structure. The first column contains the dimensionality d. The next three columns contain the CPU time of 10 iterations of RANN without supercharging, with the requested number of nearest neighbors being $k = 15, 30, 60$, respectively. The last three columns contain the CPU time of the supercharging only (after 10 iterations of RANN), with the requested number of nearest neighbors being $k = 15, 30, 60$, respectively. The CPU time is shown in seconds.

Several observations can be made from Table 1 and Figures 2, 3 and from more detailed experiments by the authors. In these observations, we refer to ratio_{smpl} as "ratio", and to $\mathbb{E}_{smpl}\left[\text{prop}_{algo}\right]$ as "proportion". We recall that, roughly speaking, the proportion measures how many of the true nearest neighbors have been found by RANN. On the other hand, the ratio measures how much average distances to suspects differ from the average distances to true nearest neighbors.

1. As expected, for a fixed d the performance of RANN improves as the number T of iterations increases, both in terms of ratio and proportion.

2. As expected, for a fixed d the performance of RANN improves if supercharging is performed, both in terms of ratio and proportion.

3. For a fixed d, the performance of RANN improves as the number of requested nearest neighbors k increases, both in terms of ratio and proportion (at the expense of running time).

4. For a fixed d, the effects of supercharging (especially on proportion) increase as k grows. For example, in Figure 2, right ($k = 15$) we observe, that, for $T = 10$ and $d = 60$, supercharging increases the proportion from 22% to 32%. On the other hand, in Figure 3, right ($k = 60$) we observe that, for $T = 10$ and $d = 60$, supercharging increases the proportion from 43% to 74%.

5. As expected, the performance of RANN slowly deteriorates in terms of proportion, as d increases. On the other hand, there is no significant deterioration of performance in terms of ratio, as d increases.

6. For $T = 10$ the ratio is always below 1.1, with or without supercharging.

7. Even for as high a dimension as $d = 60$, as few as ten iterations with supercharging correctly determine at least 30% of the true nearest neighbors. Moreover, the error of this detection decays exponentially with number of iterations T.

8. The running time of RANN, with or without supercharging, grows roughly linearly with dimensionality d, as expected from (23) (see Table 1).

9. For fixed dimensionality d, the running time of RANN without supercharging grows roughly linearly with the requested number of nearest neighbors k, as expected from (23) (see Table 1). For example, 10 iterations of RANN in the case of $d = 200$ take about 80, 124 and 208 seconds for $k = 15, 30, 60$, respectively.

10. For fixed dimensionality d, the running time of the supercharging step grows roughly quadratically with the requested number of nearest neighbors k, as expected from (23) (see Table 1). For example, in the case of $d = 200$ supercharging takes about 15, 57 and 235 seconds for $k = 15, 30, 60$, respectively.

11. The algorithm has been tested on sets of points having non-Gaussian distribution, e.g. the uniform distribution in the $d-$dimensional cube $[0, 1]^d$ or Hamming distribution (i.e. uniform distribution on the discrete set of the vertices of $[0, 1]^d$). For both uniform and Hamming distributions, the performance of the algorithm was very similar to that in the Gaussian case (see [10] for details).

12. The algorithm has been tested on sets of points, having Gaussian distribution whose covariance matrix is not the identity matrix. These tests seem to indicate that the performance of RANN improves as the condition number of the covariance matrix grows. As an extreme example, if the points belong to a $q-$dimensional linear subspace of $\mathbb{R}^d$, the performance of the algorithm does not depend on d (though the running time obviously does).

References

[1] A. Andoni, P. Indyk (2008) *Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions*, Communications of the ACM, 51(1):117-122.

	10 iterations, without supercharging			supercharging after 10 iterations		
d	$k = 15$	$k = 30$	$k = 60$	$k = 15$	$k = 30$	$k = 60$
15	0.19641E+02	0.36516E+02	0.71135E+02	0.38584E+01	0.15013E+02	0.43673E+02
20	0.21340E+02	0.39109E+02	0.75629E+02	0.43306E+01	0.16881E+02	0.59338E+02
30	0.24441E+02	0.43465E+02	0.83369E+02	0.48159E+01	0.19481E+02	0.70798E+02
50	0.30864E+02	0.52813E+02	0.97469E+02	0.62415E+01	0.25946E+02	0.97832E+02
100	0.46909E+02	0.75247E+02	0.13236E+03	0.89168E+01	0.37370E+02	0.14492E+03
150	0.64509E+02	0.10082E+03	0.17305E+03	0.11773E+02	0.48810E+02	0.19105E+03
200	0.80588E+02	0.12427E+03	0.20787E+03	0.14595E+02	0.57456E+02	0.23531E+03

Table 1: CPU time of RANN, in seconds. Number of points: 122,880.

[2] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, A.Y. Wu (1998) *An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions*, Journal of the ACM, 45(6):891-923.

[3] N. Ailon, B. Chazelle (2009) *The Fast Johnson–Lindenstrauss Transform and Approximate Nearest Neighbors*, SIAM J. Comput., 39(1):302-322.

[4] N. Ailon, B. Chazelle (2010) *Faster Dimension Reduction*, Commun. ACM, 53(2):97-104.

[5] W. Johnson, J. Lindenstrauss (1984) *Extensions of Lipschitz mappings into a Hilbert space*, Contemporary Mathematics, 26:189-206.

[6] V. Rokhlin, M. Tygert (2008) *A fast randomized algorithm for overdetermined linear least-squares regression*, Proc Natl Acad Sci USA 105(36):13212-13217.

[7] E. Liberty, F. Woolfe, P.G. Martinsson, V. Rokhlin, M. Tygert (2007) *Randomized algorithms for the low-rank approximation of matrices*, Proc Natl Acad Sci USA 104:20167-20172.

[8] D. Knuth (1969) in *Seminumerical Algorithms, vol. 2 of The Art of Computer Programming*, Reading, Mass: Addison-Wesley.

[9] N. Ailon, E. Liberty (2010) *Almost Optimal Unrestricted Fast Johnson-Lindenstrauss Transform*, eprint arXiv:1005.5513.

[10] P.W. Jones, A. Osipov, V. Rokhlin (2010) *A Randomized Approximate Nearest Neighbors Algorithm*, Yale technical report.

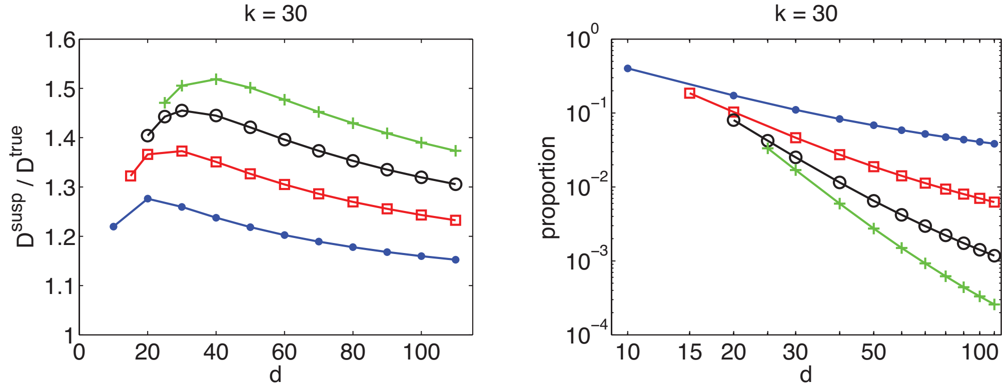

Figure 1: Number of points: $N = 30 \cdot 2^{10}$ (blue dots), $30 \cdot 2^{15}$ (red squares), $30 \cdot 2^{20}$ (black circles), $30 \cdot 2^{25}$ (green pluses).

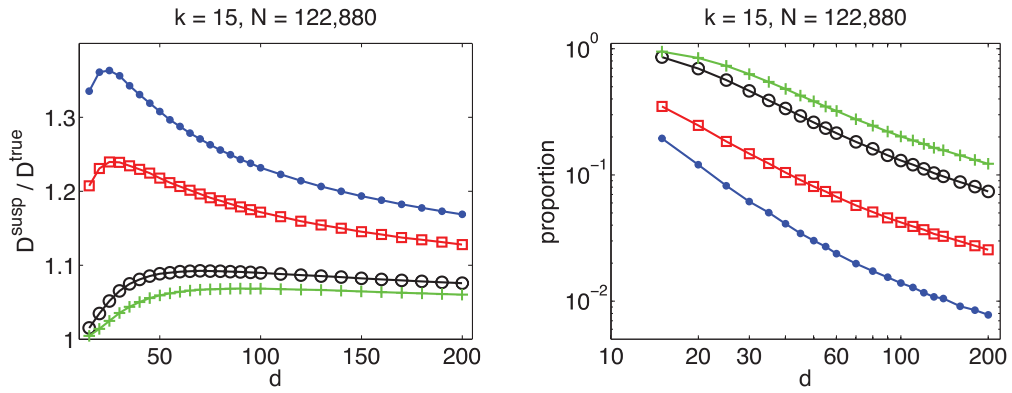

Figure 2: Without supercharging: one iteration (blue dots), ten iterations (black cirlces). With supercharging: one iteration (red squares), ten iterations (green pluses).

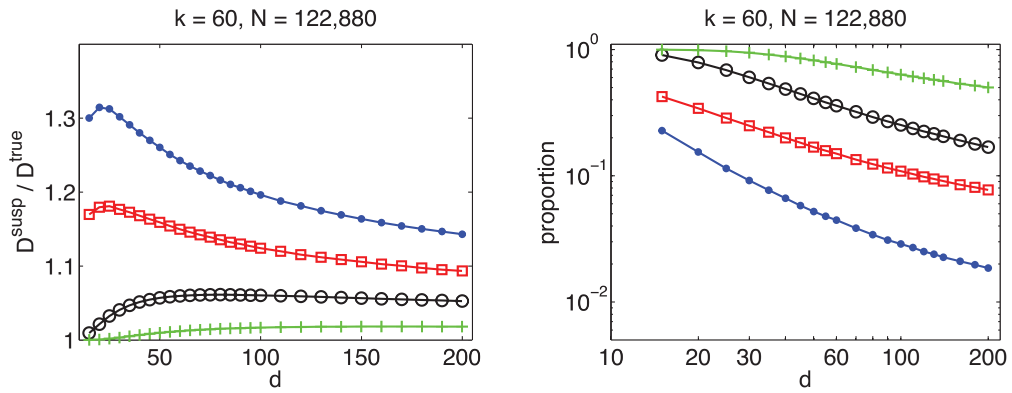

Figure 3: Without supercharging: one iteration (blue dots), ten iterations (black cirlces). With supercharging: one iteration (red squares), ten iterations (green pluses).