# Optimised KD-trees for fast image descriptor matching

Miguel Alcón Doganoc
Universitat Politècnica de Catalunya
Barcelona, Spain
miguel.alcon@est.fib.upc.edu

## 1 SUMMARY

This paper [1] focuses on improving the *KD-tree* for indexing a large number of *SIFT* (Scale invariant feature) and other types of image descriptors, which usually are large point vectors of high dimensionality. For instance, a *SIFT* descriptor is a 128-dimensional vector normalized to length one. In order to achieve their objective, the authors of the paper extended priority search to priority search among multiple trees, by creating them with the same data but with different structure. Also, they wanted the search of a point to be simultaneously independent among the trees.

They proposed 3 different types of *KD-trees* that have the properties described above.

- ***NKD-tree.*** Given $n$ and $m$, a *NKD-tree* is a group of $m$ trees constructed with the same data but, as said before, with different structure. This difference is achieved by rotating each point of the data with an arbitrary rotation matrix $R$, which is unique for each of the trees. Related with the search of a point, the authors propose to search the point among multiple trees in the form of a concurrent search with a pooled priority queue. After descending each of the trees to find an initial nearest-neighbor candidate, the best candidate from all the trees is selected. They then pool the node ranking by using one queue to sort the nodes from all $m$ trees. As a result, all of them are searched in the order of their distance from the query point simultaneously.
- ***RKD-tree.*** This kind of tree is almost the same than *NKD-tree*, but it achieves the difference of structures in the construction stage by selecting a random dimension in which to subdivide the data, instead of selecting it sequentially. This dimension is selected among a few dimensions in which the data has high variance. No rotation of the data is needed.
- ***PKD-tree.*** Again, the only difference between this kind of tree and the other two is how it achieves the difference of structure of all its trees. In this case, they align the data using *PCA* (Principal Component Analysis) and then build multiple trees using rotations that fix the space spanned by the top principal axes.

Finally, the authors test the proposed trees. They conclude that *NKD-trees* performs as well as *RKD-trees* (success rate from 75% to 88% with 3-times search speed-up with respect to the standard *KD-tree*), but that *PKD-tree* applying random *Householder*[1] transformations to the data points, in order to preserve the *PCA* subspace of appropriate dimension, leads to the highest success rate (95%).

## 2 IMPORTANCE

Matching image descriptors for an application like image recognition must be fast. So, increasing the speed of the query is really important in the *CV* (Computer Vision) field and the other ones that take advantage of it (like robotics).

However, this improvement is not only relevant in *CV*, since, at the end, the authors are trying to increase the speed of searching $d$-dimensional points within some dataset using *KD-trees*. As it is seen in the paper, this data structure performs really good in matching image descriptors quickly, which is not a surprise since it is a well-known space-partitioning data structure for organizing points in a $d$-dimensional space.

## 3 IMPLEMENTATION

Since I do not know how to perform rotations in a $d$-dimensional space (with $d > 3$) or creating the *Householder* matrices, again, in a $d$-dimensional space, I was not able to implement *NKD-tree* nor *PKD-tree*. Furthermore, some formulas related to their implementation are not complete in the paper. I also made some research on how to do it, but I did not achieve anything. So, my experiments are focused on comparing the basic *KD-tree* with the *RKD-tree* proposed in the paper. Before starting with the experiments, I implemented in *C++* both *KD-tree* and *RKD-tree*.

### 3.1 *KD-tree*

I only implemented the required operations of *KD-trees* to make the experiments, which are construction and nearest neighbor search (*NNS*). I based my implementation in [3], but I made the following changes.

- **Construction.** To select the pivot among a point list, in each recursion step, I used the *Quick Select* algorithm. Moreover, as the authors did in the paper, I implemented the random selection of the axis that is going to be used to create the splitting plane among a subset of the axis with high variance. In order to use less space (especially to use it in the *RKD-tree* implementation), instead of saving one point in each node of the tree, I saved the index of the point's position within the main vector.
- **NNS.** To know if the other side of the splitting hyperplane crosses the hypersphere around the search point (with radius equal to the current nearest distance), I followed what is explained in [2]. I also created a new type of search, the *limited search*, which, given a positive integer $n$, it limits the search to $n$ nodes.

### 3.2 *RKD-tree*

For the *RKD-tree*, I followed what is explained in the paper. In this case I want to explain what I did in detail.

- **Construction.** Given $n$, $m$, $k$ and a vector of points $V$, first the algorithm computes the top $k$ axis with highest variance, and then it generates the vector of the indices of its positions

[1]Explained in the paper, not necessary for my experiments.

within $V$ (I will refer to this vector as $HV$). Finally it creates $m$ *KD-trees* as explained before. In order to create the trees without using a lot more of space, it passes to them a pointer to $V$ and to $HV$. The algorithm also stores $n$ as the maximum number of nodes that the search algorithm can visit. At this moment, we have $m$ *KD-trees* with different structure, ready to be explored.

- **NNS.** Given a point $p$ that we want to search, first, for the root of each tree, the algorithm computes the distance between the root's point and $p$. It stores each root together with the computed distance inside a *priority queue*. This queue maintains the node with lower distance at its top. Then, while $n > 0$, while the queue is not empty and the algorithm does not find the exact point $p$, it takes the top element $e$ of the queue, it updates the closest point and its distance if $e$'s point is the closest one to $p$ visited so far. After that, it computes the distance between $p$ and each of $e$'s children, if possible, and add them to the queue. At the end of the algorithm, we have the closest point to $p$ that the algorithm can find visiting at most $n$ nodes of all the trees.

You can find the code of both *KD-tree* and *RKD-tree*in the 'src/' folder.

## 4 EXPERIMENTS

First, I tried to reproduce the experiment that is explained in the paper about *RKD-trees* to check its result. This experiment compares the expended time in searching one point in a *KD-tree* and a *RKD-tree*. I repeated it 100 times, in order to count how much times the *RKD-tree* does not reach the closest point to the query point. I used the same parameters as the authors to construct the trees, which are the following ones:

- A random vector of 128-dimensional points $V$ of size 20000, with real values between -10 and 10. In the case of the paper, points are normalized to unit length, but this just complicate things and is not relevant. It is also mentioned that tests with synthetic high-dimensional data led to even more dramatic improvements, up to 7-times diminished error rate with the *NKD-tree*.
- A vector $HV$ with the indices of the top ($k$) axes with highest variance of $V$. The parameter $k$ is not given in the paper, so I set it to 32 arbitrary.
- The number of trees $m$ is set to 6.
- The limitation of searched $n$ nodes is set to 1000;

The result of this experiment was bad. Although the searching time of the *RKD-tree* was 10 times better than *KD-tree* (*RKD-tree* lasts 0.543s in average while *KD-tree* lasts 5.114s), *RKD-tree* could find the closest point only 12 times of the 100. Because of this, I tried to increase the accuracy of the *RKD-tree*, so I made more experiments in order to reach the best possible accuracy with a good execution time. I tried different values of $n$, $m$ and $k$ to achieve the best combination.

## 5 RESULTS

The results of the experiments are shown in figure 2. Blue lines represent the results of the *KD-tree* and the other one (orange) of the *RKD-tree*. Now I am going to explain the meaning of this results, parameter by parameter.

- **n**. As you can see in figure 2a, when $n \approx 10000$, more or less a half of the total data, the *RKD-tree* starts to last more time than the *KD-tree*. The accuracy (successful search rate, figure 2b) in this point is roughly between 0.6 and 0.7. As I explained in section 1, the authors said that *RKD-tree* can reach a 0.88 accuracy with 3-times speed-up, with $n = 1000$. In my case this is not possible. For the rest of the experiments I fixed $n$ to 8000 since, with it, I obtained an acceptable accuracy with an acceptable search time.
- **m**. In figure 2c we can see that increasing the number of trees also increases the search time, but it seems that around $m = 30$ it stabilizes. Regarding the accuracy (figure 2d), it tends to increase, but the randomness of the experiments leads to significant accuracy differences in each iteration, so it is not so clear. Moreover, the construction time of the *RKD-tree* increases linearly (figure 1) because it has to build one *KD-tree* more in each iteration. For the rest of the experiments, I fixed $m$ to 24 since I achieved the best accuracy with it.

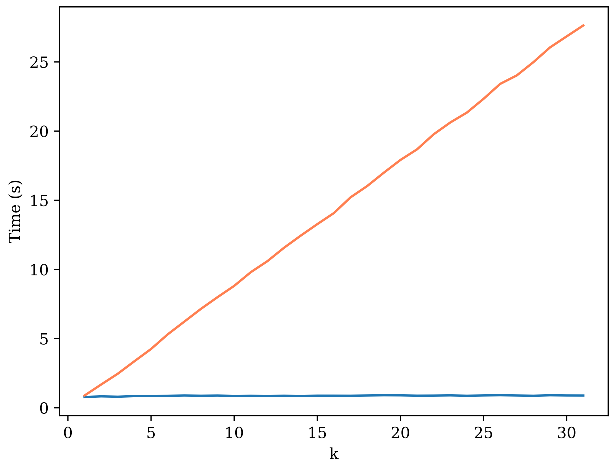


**Figure 1: Construction time**

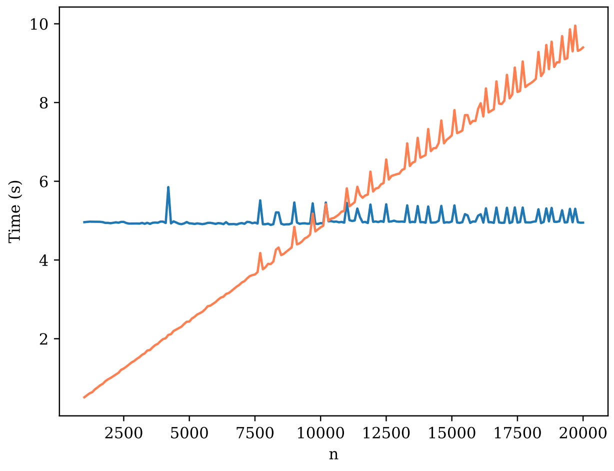
- **k**. As you can see in figure 2e, for the first time, one of the parameters affects the same in both kind of trees. The value of $k$ does not seem to be so relevant when $k > 10$, approximately. When it is lower than that, the performance is really bad, both in time and accuracy. I fixed $k$ to 41 because I achieved the best accuracy with it.

The *RKD-tree* with the tunned parameters ($n = 8000$, $m = 24$, $k = 41$) leads to 0.6 accuracy with a mean time of 4.189s, while *KD-tree* lasts 4.898s to obtain the nearest point. Also mention that the construction of the *RKD-tree* lasts 20.58s while *KD-tree* lasts 0.889*s*.

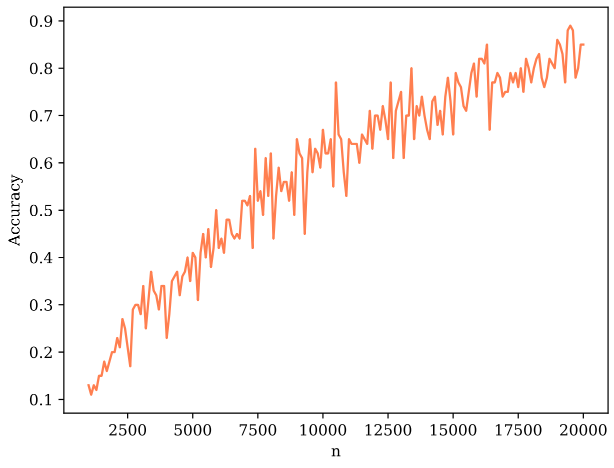
You can find all the data of the results in the 'data/' folder.

## 6 CONCLUSIONS

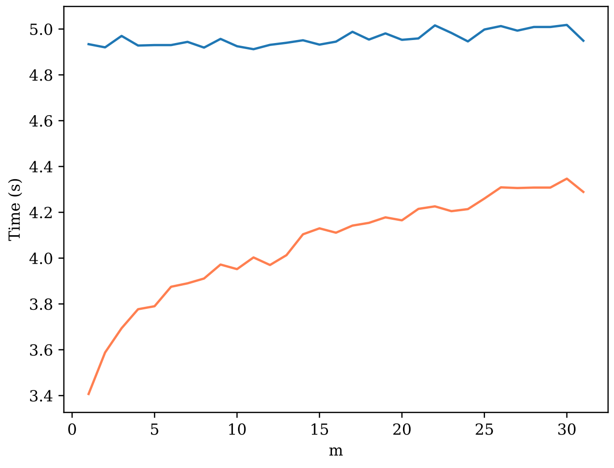
The difference in the results between the paper's experiments and mine is so huge. Maybe I did not understand the paper correctly, or maybe my implementation is poor compared with theirs. While



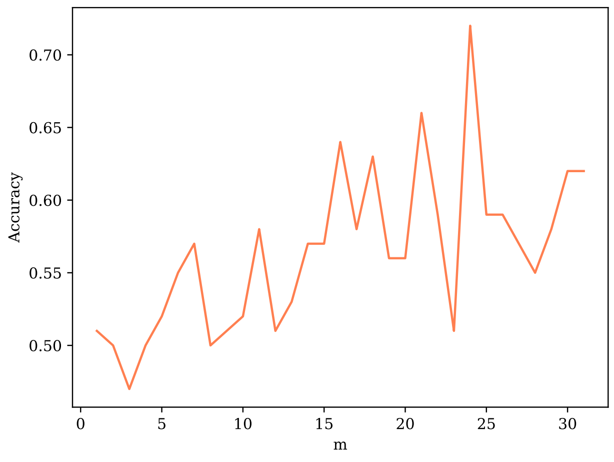
**(a) Search time**



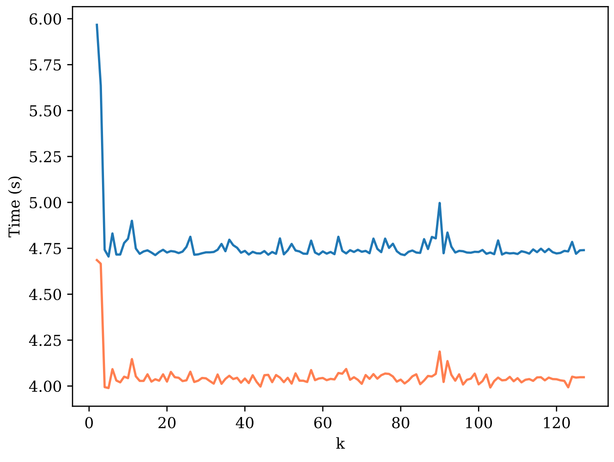
**(b) Accuracy**



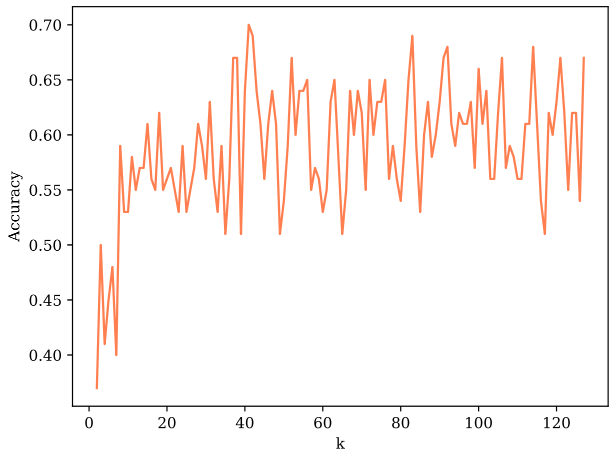
**(c) Search time**



**(d) Accuracy**



**(e) Search time**



**(f) Accuracy**

**Figure 2: Results of the experimentation**

they are obtaining 0.88 of accuracy with a 3-times speed-up with respect to the *KD-tree* search, I obtained 0.6 of accuracy with 1.1693-times speed-up. Since I am using synthetic high-dimensional data, I should obtain even better results than them, which is not happening. In the paper, they mention that with a standard *KD-tree* they obtain a 0.75 accuracy searching with a limit of 1000 searched nodes. I made a small experiment to test it, and in my case this accuracy is 0.04. As far as I know, searching only 1000 nodes over 20000 is just too few to obtain that high accuracy. Maybe they are searching for points that the tree contains, which is not what I am doing, so the algorithm can find it exactly and stop faster, but they do not let it clear in the paper. It can also be that their search algorithm cuts better the tree branches, so it needs to explore fewer nodes. But it should not be a problem for the *RKD-tree* search because its algorithm does not cut branches itself, it uses the priority queue to select the best node among all trees. However, if things showed in the paper are correct, *NKD-trees*, *RKD-trees* and *PKD-trees* represent high improvements with respect with the standard *KD-trees*.

As a final and personal conclusion, it was very challenging, and especially satisfactory, to implement the *KD-tree* class and then use it to implement the *RKD-tree* class. In order to extend this work, I would want to find the problem of my implementation, but also it would be interesting to parallelize the search algorithm of the *RKD-tree*, which it should not be so difficult, to see how much it improves.

## REFERENCES

[1] Hartley, R. R.: Optimised kd-trees for fast image descriptor matching. In *In CVPR, IEEE Computer Society* (2008).
[2] Nguyen, T. Nearest neighbor search (pg. 15). http://andrewd.ces.clemson.edu/courses/cpsc805/references/nearest_search.pdf, 2019.
[3] Wikipedia. kd-tree. https://en.wikipedia.org/wiki/K-d_tree, 2019.