Assumptions and decisions made during the design and implementation.

Design target:

The main API design goal for this library is to make the kd-tree component as flexible as possible without sacrificing performance too much. The API should allow the user to build both “easy to use” and “high performance” versions of the kd-tree by changing the point representation and defining distance computational function. As many tradeoffs as possible (including where to store point’s data) should resolved as late as possible without significant performance overhead.

The physical design goal is to be able to make a single-file library where all code is located inside of one header file and distribute it this way. Normally the library contains multiple files but it’s easy to write a script that combines them. The user should just include the library’s header file and don’t worry about building it.

Considering implementation targets, manual memory management and pointer operations should be avoided since it’s usually error-prone, STL should be used for this inside the kdtree implementation and user provided code should be used for storing point’s data (STL is used for the example Point implementation). Code should be platform independent.

Important notice: Not all design targets are achieved at this point.

Various assumptions:

- Modern compiler is used where there is no difference between i++ vs ++i, the return value optimization works and similar cases;
- Number of dimensions is selected at runtime;
- No need to remove points;
- Distance calculation algorithm is linked to the point type (quite bad assumption since there was really no reason to make it but it’s reasonably easy to fix);
- It’s ok to always store distance in double;
- Maximum number of points in the tree: 4,294,967,294;
- Maximum number of dimensions: 4,294,967,294;
- It’s possible to have a copy (inside the tree) of all pointers data in memory. (This assumption is for the example Point implementation);
- 32 and 64 bit platforms should be supported but no need to worry about big/little endian;
- The kdtree should not be fully thread safe;

Future improvements:

1. Profiling;
2. Improve interface for tree creation: pass iterators instead of vector;
3. Refactoring:
    a. Remove search algorithm code duplication;
    b. Remove code duplication in few other places;
    c. Replace recursion by iterative algorithm;
4. User binary search for pushing points into the limited queue;
5. During construction: sort once per dimensions and reuse sort results on each level

6. Return iterators from the limited queue instead of vectors to eliminate memory allocation maybe even replace list by vector (profiling needed);
7. Make nice point class example that is not derived from std::vector;
8. Passing some shared user data to all points (this would allow user to create points that have only indexes of some external data structures);
9. Add support for passing allocators (should be easy since STL is used for memory management);
10. Implement addition tree creation strategies in addition to the median;
11. Implement limitation of the number visited bins during search;
12. Implement search with time limitation;
13. Add an ability to select distance calculation algorithm at runtime independently from the point type (probably by passing a functor that takes 2 points and the number of dimensions);
14. Improve BBF algorithm, it looks like it's not faster than normal search (but profiling is needed really to figure out where is the problem. My guess is the sorting of the stack);
15. Separate coordinates and user data (in order to fit more coordinates in cache during search);
16. Implement SSE/AVX friendly distance computation;

Development process:

1. API;
2. Simple test application for creating tree and searching;
3. Limited priority queue;
4. Simple tree creation point by point;
5. Linear search in vector;
6. Simple knn search;
7. BBF search;
8. Tree construction from a vector of points;
9. 2 test application as specified;
10. Porting build scripts to GCC (from MSVS);
11. Unit tests;

Important decisions during the API design

Number of dimensions

Considered options:

1. Number of dimensions is hardcoded in code
    - MAINTAINABILITY: Simple code, easy to understand and maintain.
    - PERFORMACE: Compiler can do some optimizations since it knows number of dimensions (number of iterations in many loops) at compile time. Some manual algorithms fine tuning and optimizations are possible.
    - USABILITY: Limited possibility to reuse the component since the number of dimensions is hardcoded.
2. The number of dimensions is passed as the template parameter
    - MAINTAINABILITY: More complicated parameterized code.
    - PERFORMACE: Compiler can do some optimizations since it knows number of dimensions (number of iterations in many loops) at compile time. Manual optimization is more complicated since number of dimensions may vary.
    - USABILITY: It's possible to reuse the component for other projects but recompilation is required.
3. The number of dimensions is passed at runtime
    - MAINTAINABILITY: More complicated code.
    - PERFORMACE: Compiler has less freedom for optimization. More memory may be required for storing data (for example, 3 extra points, if points are stored in std::vector).
    - USABILITY: Easy to reuse and create tree with any number of dimensions at runtime

Selected option:

3. The number of dimensions is passed at runtime.

Reason:

It is unknown at this point what kind of test data will be used and the input data format allows to pass points with any dimensions, so the test application will know the number of dimensions at runtime only.

Comments:

If the number of dimensions is known upfront and it is known that it is not going to be changed (some parameters of the physical world for example), I would go with the second or even first approach to make everything faster and simpler.

Point type

Considered options:

1. Hardcoded point type with only coordinates
2. Library provided class template parameterized over the number of dimensions
3. Library provided class with the number of dimensions selected at runtime
4. User provided class derived from the library provided class that implements some interface
5. User provided class that implements some interface passed as the template parameter to the tree
6. Splitting user data and coordinates

7. Passing some shared user object to all points related operations

Selected option:

5. User provided class that implements some interface passed as the template parameter to the tree.

The user provided point class is responsible to reading/writing data to disk.

Reason:

Provides most flexibility (it would be even more flexible if some share user data would be passed to the point during all operations). The user is free to resolve the usability/memory usage tradeoff later. It is possible to use either very simple point representation that stores just coordinates or more complicated representation that stores additional data or allows to select the number of dimensions at runtime.

Performance evaluation

Theoretical algorithm complexity

Tree construction

The tree construction algorithm performs 2 main steps: sorts points by one of dimensions and constructs a bin (node) with at the median. Then it performs the same steps for all points left and right from the median. The node construction complexity is constant and the sort complexity is **N*Log(N)** where N is the number of points in the tree. So, each first iteration complexity is **N*Log(N)**, the second iteration complexity **is 2*(N/2*Log(N/2))** since each subtree contains only half of points, the third is **4*(N/4*Log(N/4))** and so on. So, the entire tree creation has the complexity of **N*Log(N)^2**.

Nearest point query

The nearest point query contains 2 main steps: going down the tree to the smallest bin with the query point and iteratively going around the tree and searching other bins if they can contain closer points. Complexity of the first part is **Log(N)** distance computations and coordinate comparisons (for 1 dimension) assuming the tree is balanced.

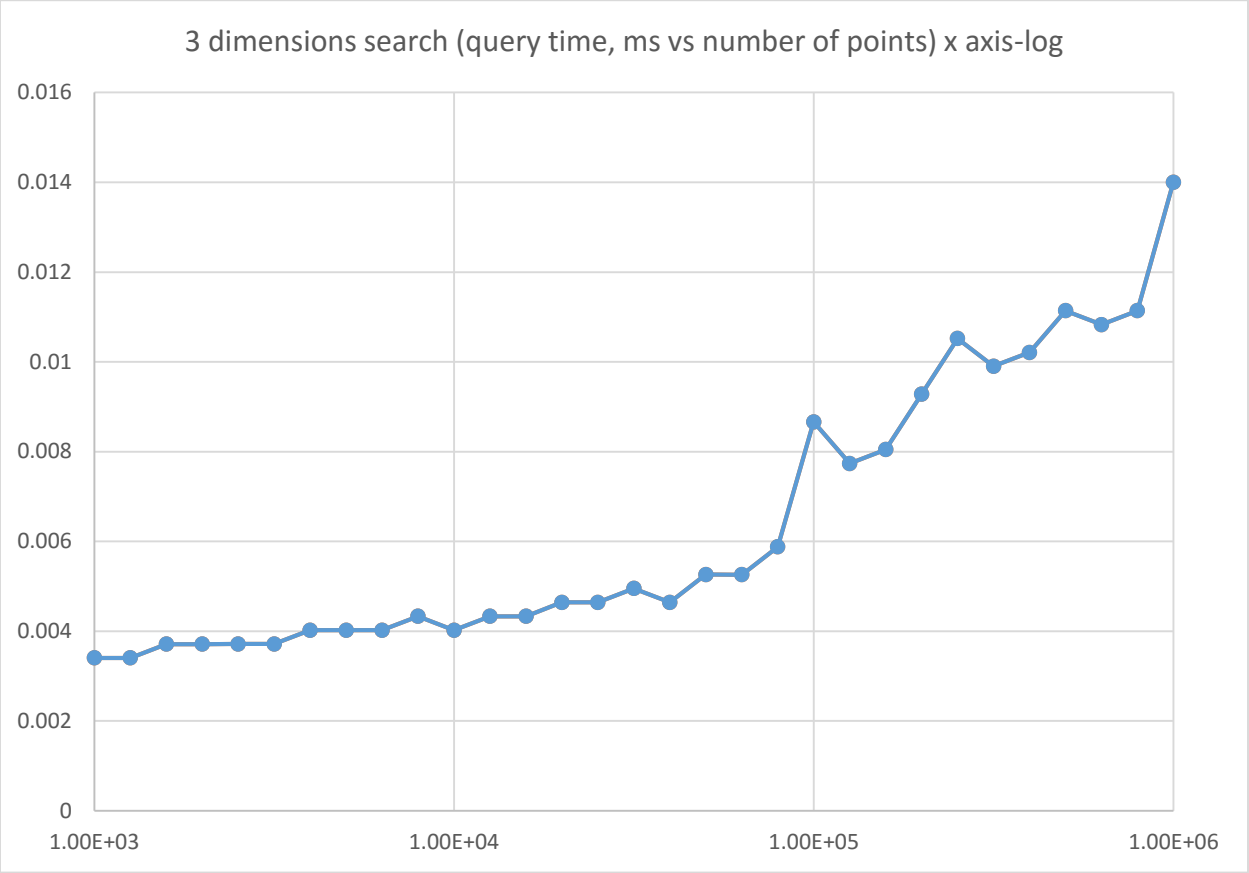
The second (backtracking part) is more interesting. In the best case scenario, the tree contains the point at the query point location and the closest distance found during the first part is **0**. In this case, the backtracking will perform **Log(N)** coordinate comparisons and **0** distance computations. In the worst case scenario, all bins are checked and the complexity is **N** distance computations and **N** coordinate comparisons. So, the complexity of the nearest point query is between **Log(N)** and **N**.

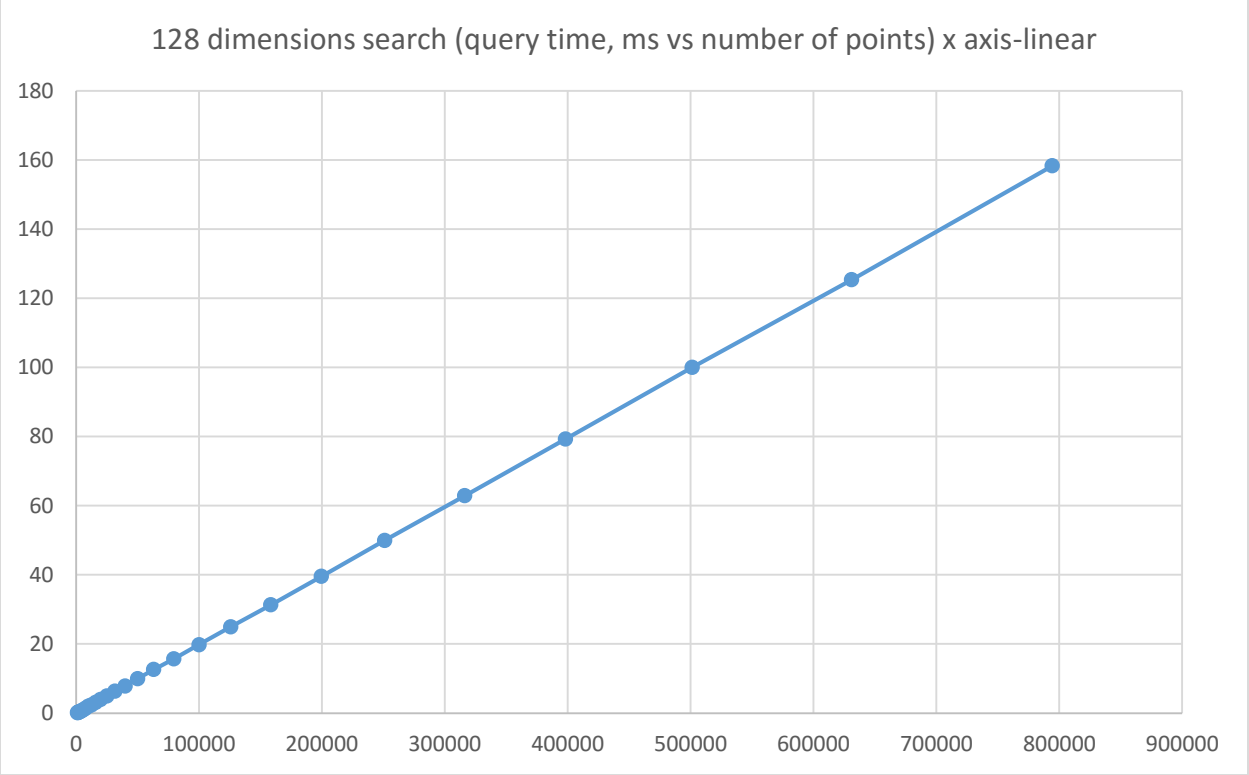
We assume here that only 1 nearest point is required. If more points are required, the result will be similar but more bins have to be checked, so the complexity is closer to the upper bound.

The average case complexity depends on how many bins should be checked during the second step of the algorithm. A bin is not checked only if the distance to that bin is larger than the distance to the closest known point. If many bins are skipped, the total complexity is closer to **Log(N),** if more bins have to be checked, the complexity is closed to **N**.

Assuming that points are evenly distributed in the space, all dimensions have the same length (distance between min and max coordinate), all coordinates are normalized to the range 0..1 and the number of points >> number of dimensions, it is possible to compute some statistics about distance to the closest point and distance to a bin boundary. A distance to a bin boundary will be between 0 and 1, and the average distance will be ½*(½^L) where the L is the bin depth level (assuming that the boundary is in the middle and the point is in the middle of the left or right bin sections). Since all points are evenly distributed, we can say that the distance to the closest point is equal Sqrt(D*(K/2)^2) where K is the average distance between 2 points on a single axis and D is the number of dimensions. We can compute K in terms on number of points and number of dimensions: K = 1/N^(1/D). So, the average distance to the closest point is ½*Sqrt(D)* 1/N^(1/D). It's easy to see that the average distance to the closest point quickly grows up with the number of dimensions and it's very likely that many (or all) bins will be checked if the number of dimensions is high (10+ dimensions for 1000 points, for example).

Performance testing results for the nearest point search algorithm





So, the practical performance measurement looks quite consistent with the theoretical expectations. The 3-dimensions search time grows logarithmically with the number of points (in the 1K-100K points at least), while search time for 128 dimensions grows linearly.

It's interesting that after 100k points, the 3dimensional search time grows faster but it's still possible to say that it's logarithmical. My guess about the reason: it's running out of CPU cache (at that point the tree takes few MB of memory which is similar size to the CPU cache).