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. Refactoring:
    a. Remove search algorithm code duplication;
    b. Remove code duplication in few other places;
    c. Replace recursion by iterative algorithm;
3. User binary search for pushing points into the limited queue;
4. Return iterators from the limited queue instead of vectors to eliminate memory allocation maybe even replace list by vector (profiling needed);

5. Make nice point class example that is not derived from std::vector;
6. Passing some shared user data to all points (this would allow user to create points that have only indexes of some external data structures);
7. Add support for passing allocators (should be easy since STL is used for memory management);
8. Implement addition tree creation strategies in addition to the median;
9. Implement limitation of the number visited bins during search;
10. Implement search with time limitation;
11. 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);
12. 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);
13. Separate coordinates and user data (in order to fit more coordinates in cache during search);
14. 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.