KDTree Exercise

Uber Advanced Technologies Center seeks highly skilled C++ engineers to join us in achieving our ambitious objectives. This homework focuses on providing an assessment of your implementation skills. In this case, developing C++ software to implement a fast spatial data structure: KDTrees.

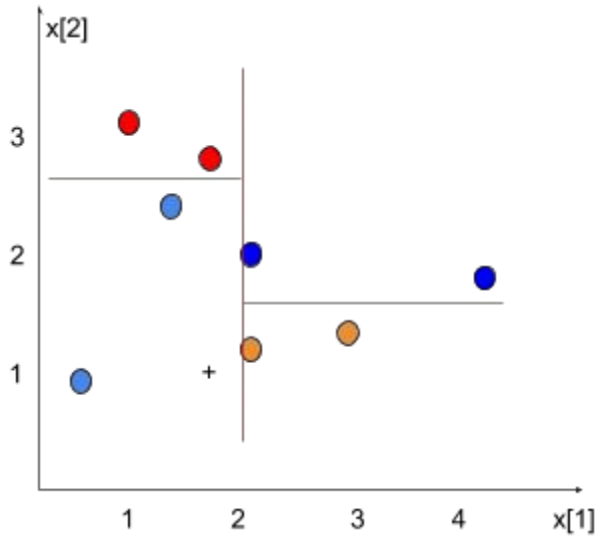
We are looking for candidates that are able to take such ideas and turn them into production quality code: high performance, well written, well tested, robust pieces of software written in C++.

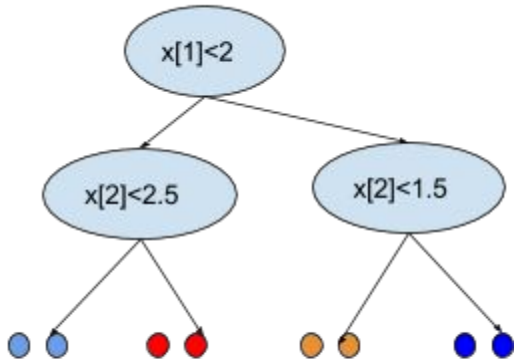
The exercise below is your opportunity to show us your skills in this area. To guide you, our grading rubric is roughly based on:

- Your solution should be your own code, or clearly attributed if you use other code sources.
- You should make use of the C++ standard library where appropriate.
- You should use templated classes and build a clean, usable and extensible API.
- You should use good coding practices (e.g. defensive programming, RAII etc.) to make your code production quality.

Remember, we're trying to assess your ability to produce solid *production quality* code.

Your mission, if you choose to accept it, is to build a KD-tree library for performing efficient nearest neighbor point queries given a prior dataset of points. That is, given a set of D-dimensional points $\{X_1, X_2, ...X_N\}$, a KD-tree defines a recursive binary space partitioning of the data. In short, it is a binary tree where each node in the tree defines a partitioning of the space, an axis aligned hyperplane that "splits" the X's associated with that node. The X's on the "left" of the split continue to be recursively split by the left child node and so on for the X's on the "right". Here we assume the nodes in the tree define splits: an axis or dimension of the vector to split on and a threshold that defines the splitting hyperplane and a left and right child to form the tree. All the data (the X's), or more usefully an index or reference to the data, is stored in the leaves of the tree. The figure below shows a very simple 2 dimensional example where the leaves store at most 2 points (the splitting could have continued of course to a single point at each leaf).





To greedily search for the nearest neighbor in the tree given a query point, e.g. Y, we descend through the tree. At each node we test whether the value of the query Y in the node's axis (e.g. Y[j]) is on the left side of the split (i.e. Y[j]<t) or not and descend to the left or right child accordingly until we hit a leaf. The problem with this approach is that for a query, there may be a closer point that was actually on the other side of the split. Consider a query in the above diagram shown as the '+' point. Greedily descending through the tree would lead to the light blue points where infact the orange point is closer. By backtracking through the ignored branches of the tree we can find the exact nearest neighbors. By searching through the hyperplanes that are closest to the query (i.e. keeping track of the nodes examined and sorting by the distance to the hyperplane) we can search the branches of the tree for the most promising candidates first. By keeping track of the best leaf nodes found to date, branches of the tree that are too far from the query point (i.e. the distance to the hyperplane is larger than the best found node to date) can be pruned/ignored entirely.

One of the keys to performing efficient nearest neighbor search over the X's, is to choose the splitting axis and the split threshold so that the resulting tree is well balanced. A simple heuristic is to choose the splitting axis that has the largest variation and to choose the split point that is the median of the values.

For this task, you must build a KD-Tree class that provides at least the following interface:

- Construction/destruction.
- Create a tree from a set of N-dimensional vectors (points). The tree should be created by recursively choosing the axis (dimension) with the largest range (i.e. $arg\ max_j (max\ X_i[j]\ -\ min\ X_k[j])$, and choosing the split point to be the median of the X[j]'s at that node. References or points should be stored in the leaves.
- Support efficient exact query the nearest point to a query point. That is search through the tree efficiently and find the nearest point and return the reference of the point.
- Support I/O to save/load the tree from disk.
- Lastly, your implementation should be templated allowing for specialization with float or real types. You should choose one of the above specializations for your testing and explain your choice.
- For a bonus, find a way to enable the same code to be extended with different ways of choosing the splitting axis or split point.

Using your library, you should build two applications:

- build_kdtree: An application to build the kdtree from a sample dataset (a simple CSV file – details below). It should build the tree and save it to a custom location.
- query_kdtree: The second application should read a file containing query points and the KDTree generated above and use it save out the exact nearest index (0-based) in the sample dataset and its corresponding distance. The output should be a CSV file with each line corresponding to a query.

We provide a set of test points and query points attached. The file format is a CSV file with a ',' for delimiter:

A00, A01, A02, ... A0N
A10, A11, A12, ... A1N

DO NOT DISTRIBUTE.

...

Each line contains a point with spacing between them. You can safely assume the file is formatted correctly and do not need to build an extensive parsing system. The point ID is the 0-based row number.

Your program should take in a set of query points and produce a file output that consists for each query point, the corresponding closest index of the sample data:

Query0_closest_index
Query1_closest_index
....

Implement the above operations and validate that your algorithm works. This exercise is not intended to take more than 8 hours, so try to size your plans accordingly. Please submit your source code, any associated build framework (e.g. cmake), testing, and a README clearly listing any other dependencies. Your implementation must be written in C++ (C++-11 is preferred) and compile with GNU g++. Your software should compile and run on a vanilla Ubuntu 14.04LTS 64-bit linux system. You may use additional dependencies, but clearly document them and make sure they are the vanilla install packages or provide source and make sure it builds as part of your build system. Your README should also detail how to build and how to run the software.

Additionally, please submit a brief pdf presentation that describes your implementation and design decisions undertaken. Detail the computational efficiency of your implementation and explain how you could make it better given more time.