NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer’s or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.
Creating, Searching, and Deleting KD Trees Using C++

by Robert J Yager
Weapons and Materials Research Directorate, ARL
K-dimensional (KD) trees use binary space-partitioning algorithms to organize and store data points in K-dimensional space. They are particularly useful for efficient nearest-neighbor search algorithms. This report presents a set of functions, written in C++, that is designed to be used to create, search, and delete KD trees. All of the functions are based on recursive algorithms. Tests for measuring function performance are included, as are examples for creating Voronoi diagrams.
Contents

List of Figures v

Acknowledgments vi

1. Introduction 1

2. Sorting Tables — the ColumnSort() Function 1
   2.1 ColumnSort() Code ................................................................. 1
   2.2 ColumnSort() Parameters ...................................................... 2
   2.3 ColumnSort() Example .......................................................... 2
   2.4 ColumnSort() Performance .................................................... 2

3. KD-Tree Nodes — the NODE Struct 4
   3.1 NODE Code ........................................................................ 5
   3.2 NODE Parameters ................................................................ 5

4. Creating KD Trees — the NewTree() Function 5
   4.1 NewTree() Code .................................................................... 6
   4.2 NewTree() Parameters ........................................................... 6
   4.3 NewTree() Return Value .......................................................... 6
   4.4 NewTree() Example — Creating a Simple KD Tree ..................... 6

5. Deleting KD Trees — the DeleteTree() Function 7
   5.1 DeleteTree() Code .................................................................. 7
   5.2 DeleteTree() Parameters .......................................................... 8

6. Searching KD Trees — the NNSearch() Function 8
   6.1 NNSearch() Code .................................................................... 8
   6.2 NNSearch() Parameters ........................................................... 8
   6.3 NNSearch() Example — Creating a Simple Voronoi Diagram ........... 9
   6.4 NNSearch() Performance ......................................................... 10
7. Example — Fermat’s Spiral 13

8. Example — Comparing Average Distances to Sources 14

9. Example — Optimizing Areal Coverage 16

10. Code Summary 17

11. References 19

Distribution List 20
List of Figures

Fig. 1  ColumnSort() performance compared with stable_sort() performance.........................4
Fig. 2  Tabulated data stored in a 2-index array .................................................................5
Fig. 3  A simple Voronoi diagram .........................................................................................10
Fig. 4  Scale for Voronoi diagrams .......................................................................................10
Fig. 5  Brute-force and KD-tree methods compared for $K = 2, 4, \text{ and } 6$ ......................12
Fig. 6  Voronoi diagrams of Fermat’s Spiral ($m = 100 \text{ left, } m = 200 \text{ center, } m = 600 \text{ right}$) ......14
Fig. 7  Voronoi diagrams (case 1 left, case 2 center, case 3 right) ...........................................15
Fig. 8  Voronoi diagrams of a table without and with a point added that minimizes $D$ ..........17
Acknowledgments

I would like to thank Tim Fargus of the US Army Armament Research, Development and Engineering Center’s System Analysis Division. Tim provided technical and editorial recommendations that improved the quality of this report.
1. Introduction

K-dimensional (KD) trees use binary space-partitioning algorithms to organize and store data points in K-dimensional space. They are particularly useful for efficient nearest-neighbor search algorithms. This report presents a set of functions, written in C++, that is designed to be used to create, search, and delete KD trees. All of the functions are based on recursive algorithms. Tests for measuring function performance are included, as are examples for creating Voronoi diagrams.

The functions that are described in this report have been grouped into the yKDTree namespace, which is summarized at the end of this report. The yKDTree namespace relies exclusively on standard C++ operations. However, example code that is included in this report makes use of the yRandom namespace for generating pseudorandom numbers and the yBmp namespace for creating Voronoi diagrams.

2. Sorting Tables — the ColumnSort() Function

The ColumnSort() function uses a stable merge-sort algorithm to sort tabulated data into ascending order. The data must be stored in a 2-index array, where the indices are arbitrarily referred to here as rows (first index) and columns (second index).

The ColumnSort() function is included in the yKDTree namespace as a helper function for the NewTree() function, which is described in section 4. However, the ColumnSort() function can also be useful on its own when an efficient means of sorting tables by columns is desired.

2.1 ColumnSort() Code

```cpp
template<class T> void ColumnSort(int& a, int& b, T** t, unsigned c) {
    if (b - a < 2) return;
    int m = a + (b - a) / 2;
    ColumnSort(a, m, t, c);
    ColumnSort(m, b, t, c);
    for (int i = m; i < b; i++)
        if (t[i] < t[i + m - m])
            t[i] ^= t[i + m - m], t[i + m - m] ^= t[i], t[i] ^= t[i + m - m];
}
```

//~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~
2.2 ColumnSort() Parameters

a  a points to the first row that will be included in the sort.
b  b points to one past the last row that will be included in the sort.
t  t points to temporary storage for the ColumnSort() function. t must point to a
d  array that is capable of storing at least b-a elements.
c  c specifies the column on which the sort will be based.

2.3 ColumnSort() Example

The following example uses the ColumnSort() function to sort a 2-column table, first by the
second column, then by the first.

```c
#include <cstdio>
#include "y_kd_tree.h"

int main(){
    int *T[9],A[9][2]={0,5,0,7,1,9,0,8,1,6,0,7,1,4,0,0,0,3},*B[9];
    for(int i=0;i<9;++i)B[i]=A[i];
    yKDTree::ColumnSort(B,B+9,T,1),yKDTree::ColumnSort(B,B+9,T,0);
    printf(" UNSORTED |  SORTED
---------|--------
   ");
    for(int i=0;i<9;++i)
        printf(" %d , %d  |  %d , %d 
",*A[i],A[i][1],*B[i],B[i][1]);
}
```

OUTPUT:

```
  UNSORTED |  SORTED
---------|--------
   0 , 5  |  0 , 0
   0 , 7  |  0 , 3
   1 , 9  |  0 , 5
   0 , 8  |  0 , 7
   1 , 6  |  0 , 7
   0 , 7  |  0 , 8
   1 , 4  |  1 , 4
   0 , 0  |  1 , 6
   0 , 3  |  1 , 9
```

2.4 ColumnSort() Performance

The following code measures the performance of the ColumnSort() function and compares it
with the performance of the stable_sort() function. The yRandom namespace is used to generate
pseudorandom numbers for the test. Time measurements represent the total time required to
perform 10^6 sorts for tables with 2^n rows, where n varies from 1 to 14.
The output was generated by compiling the code using Microsoft’s Visual Studio C++ 2010 Express compiler, with the output set to “release” mode. For this scenario, the ColumnSort() function outperforms the built-in stable_sort() function (Fig. 1).

```cpp
#include <algorithm>
#include <cstdio>
#include <ctime>
#include "y_kd_tree.h"
#include "y_random.h"

inline bool Compare(double*a, double*b){
  return a[0]<b[0];
}

int main(){
  const int N=1<<14, M=1000000;
  unsigned I[625];
  yRandom::Initialize(I,1);
  double s, t, *T[N], *A[N];
  for(int i=0; i<N; ++i) A[i] = new double[1];
  printf( "          |  std::stable_sort()  |     ColumnSort()
             |    count   |   time   | Z[m/2][0] |   time   | Z[m/2][0]
             | count     | avg.     | count   | avg.     
             |------------|----------|----------|----------
" );
  for(int m=2; m<=N; m*=2){
    s=0, t=clock();
    yRandom::Initialize(I,1);
    printf("%7d   |%9.3f |%10.7f |" , m, (clock()-t)/CLOCKS_PER_SEC, s/M);
    for(int k=0; k<M; ++k){
      for(int i=0; i<m; ++i) A[i][0] = yRandom::RandU(I,0,1);
      std::stable_sort(A, A+m, Compare), s+=A[m/2][0];
      printf("%9.3f |%10.7f |", m, (clock()-t)/CLOCKS_PER_SEC, s/M);
      for(int i=0; i<N; ++i) delete[] A[i];
    }
  }
  return 0;
}
```

---

The output was generated by compiling the code using Microsoft’s Visual Studio C++ 2010 Express compiler, with the output set to “release” mode. For this scenario, the ColumnSort() function outperforms the built-in stable_sort() function (Fig. 1).
OUTPUT:

<table>
<thead>
<tr>
<th>row count</th>
<th>std::stable_sort()</th>
<th>ColumnSort()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time (s)</td>
<td>Z[m/2][θ]</td>
</tr>
<tr>
<td></td>
<td>avg.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.046</td>
<td>0.6667224</td>
</tr>
<tr>
<td>4</td>
<td>0.078</td>
<td>0.6002211</td>
</tr>
<tr>
<td>8</td>
<td>0.219</td>
<td>0.5556498</td>
</tr>
<tr>
<td>16</td>
<td>0.546</td>
<td>0.5293284</td>
</tr>
<tr>
<td>32</td>
<td>1.316</td>
<td>0.5150998</td>
</tr>
<tr>
<td>64</td>
<td>3.401</td>
<td>0.5077139</td>
</tr>
<tr>
<td>128</td>
<td>7.675</td>
<td>0.5038248</td>
</tr>
<tr>
<td>256</td>
<td>17.456</td>
<td>0.5019407</td>
</tr>
<tr>
<td>512</td>
<td>39.059</td>
<td>0.5009805</td>
</tr>
<tr>
<td>1024</td>
<td>83.569</td>
<td>0.5004688</td>
</tr>
<tr>
<td>2048</td>
<td>186.202</td>
<td>0.5002375</td>
</tr>
<tr>
<td>4096</td>
<td>409.029</td>
<td>0.5001205</td>
</tr>
<tr>
<td>8192</td>
<td>890.498</td>
<td>0.5000599</td>
</tr>
<tr>
<td>16384</td>
<td>2061.582</td>
<td>0.500335</td>
</tr>
</tbody>
</table>

Fig. 1 ColumnSort() performance compared with stable_sort() performance

3. KD-Tree Nodes — the NODE Struct

NODE structs can be used to store the nodes that make up KD trees. Typically, NODE structs are created using the NewTree() function (see section 4) and deleted using the DeleteTree() function (see section 5).
3.1 NODE Code

```cpp
template<class T> struct NODE{
    // A KD-TREE NODE
    T*r; // POINTER TO A ROW IN A TABLE
    NODE*a,*b; // POINTERS TO SUBNODES
    int k; // NODE LAYER
};
```

3.2 NODE Parameters
- `r` points to a row in a table.
- `a` points to a subnode.
- `b` points to a subnode.
- `k` is used to identify a node’s layer.

4. Creating KD Trees — the NewTree() Function

If a 2-index array is used to store sortable tabulated data in the format that is shown in Fig. 2, then the NewTree() function can be used to create a KD tree for the tabulated data. In Fig. 2, values for both independent and dependent variables are stored in array `A`. Independent variables are stored in columns with subscripted `X` headers, while dependent variables are stored in columns with subscripted `Y` headers. Each row represents a single data point.

<table>
<thead>
<tr>
<th></th>
<th><code>X_0</code></th>
<th><code>X_1</code></th>
<th>...</th>
<th><code>X_k</code></th>
<th>...</th>
<th><code>X_{K-1}</code></th>
<th></th>
<th><code>Y_0</code></th>
<th><code>Y_1</code></th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>A_{0,0}</code></td>
<td><code>A_{0,1}</code></td>
<td>...</td>
<td><code>A_{0,k}</code></td>
<td>...</td>
<td><code>A_{0,K-1}</code></td>
<td></td>
<td><code>A_{0,K}</code></td>
<td><code>A_{0,K+1}</code></td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td><code>A_{i,0}</code></td>
<td><code>A_{i,1}</code></td>
<td>...</td>
<td><code>A_{i,k}</code></td>
<td>...</td>
<td><code>A_{i,K-1}</code></td>
<td></td>
<td><code>A_{i,K}</code></td>
<td><code>A_{i,K+1}</code></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td><code>...</code></td>
<td><code>...</code></td>
<td><code>...</code></td>
<td><code>...</code></td>
<td><code>...</code></td>
<td><code>...</code></td>
<td></td>
<td><code>...</code></td>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>m-1</code></td>
<td><code>A_{m-1,0}</code></td>
<td><code>A_{m-1,1}</code></td>
<td>...</td>
<td><code>A_{m-1,k}</code></td>
<td>...</td>
<td><code>A_{m-1,K-1}</code></td>
<td></td>
<td><code>A_{m-1,K}</code></td>
<td><code>A_{m-1,K+1}</code></td>
<td>...</td>
</tr>
</tbody>
</table>

Fig. 2 Tabulated data stored in a 2-index array
Note that the NewTree() function uses the “new” command to allocate memory for nodes. To avoid memory leaks, the DeleteTree() function (see section 5) should be used to deallocate memory when a KD tree is no longer needed.

4.1 NewTree() Code

```cpp
template<class T> NODE<T>* NewTree(  // CREATE A KD TREE
    T** a, T** b, // POINTERS TO STARTING AND ENDING ROWS
    int K, // NUMBER OF INDEPENDENT DIMENSIONS
    int k=0) { // NODE LAYER (SET BY RECURSIVE ALGORITHM)
    T** t = new T*[b-a]; /* ColumnSort(a,b,t,k); */& delete[] t;
    T** m = a+(b-a-1)/2;
    NODE<T>* N = new NODE<T>; /* ColumnSort(m,b,b-t,k); */& delete[] N;
    return N;
}
```

4.2 NewTree() Parameters

- **a**: points to the first row of the table that will be included in the new KD tree.
- **b**: points to one past the last row of the table that will be included in the new KD tree.
- **K**: specifies the number of independent dimensions that are included in the table that is specified by **a** and **b**.
- **k**: specifies the layer of the current node that is being created by the NewTree() function. **k** is set automatically, either by the default value, or by the NewTree() function when it calls itself recursively.

4.3 NewTree() Return Value

The NewTree() function returns a pointer to the root node of a newly created KD tree.

4.4 NewTree() Example — Creating a Simple KD Tree

The following example code uses the NewTree() function to create a simple KD tree, with \( K = 2 \) independent dimensions, then prints the nodes. Nodes represented by \((X,X)\) are empty.
```c
#include <cstdio>
#include "y_kd_tree.h"

inline void PrintNode(N) // <================================== PRINTS THE COORDINATES OF A NODE
yKDTree::NODE<int>*N){
N?printf("(%d,%d) \n",N->r[0],N->r[1]):printf("(X,X) \n");
}

int main(){
// <================================== A SIMPLE EXAMPLE FOR THE NewKDTree() FUNCTION
int*A[5],B[5][3]={4,1,0, 4,3,1,6,2,2, 2,4,3, 8,4,4};
printf("TABULATED DATA:\n");
for(int i=0;i<5;++i)printf("%13d,%d,%d \n",B[i][0],B[i][1],B[i][2]);
printf("\n\nKD TREE:\n");
yKDTree::NODE<int>*N=yKDTree::NewTree(A,A+5,2);
p yNode(N->a),printf(" \n"),yNode(N->b);
} // ~~~~~~~~~~ YAGENAUT@GMAIL.COM ~~~~~~~~~~~~ LAST~UPDATED~15JUL2014 ~~~~~~~
```

OUTPUT:

<table>
<thead>
<tr>
<th>TABULATED DATA:</th>
</tr>
</thead>
<tbody>
<tr>
<td>4,1,0</td>
</tr>
<tr>
<td>4,3,1</td>
</tr>
<tr>
<td>6,2,2</td>
</tr>
<tr>
<td>2,4,3</td>
</tr>
<tr>
<td>8,4,4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>KD TREE:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4,3)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>(4,1)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>(2,4)</td>
</tr>
<tr>
<td>(X,X)</td>
</tr>
</tbody>
</table>

5. Deleting KD Trees — the DeleteTree() Function

The DeleteTree() function can be used to delete a KD tree that has been created using the NewTree() function.

5.1 DeleteTree() Code
5.2 DeleteTree() Parameters

N  N points to the root node of a KD tree.

6. Searching KD Trees — the NNSearch() Function

The NNSearch() function can be used to search a KD tree for \( A_i \), a row in a table that specifies a location that minimizes the distance to \( X \), a user specified location where

\[
X = \{X_0, X_1, \ldots, X_k, \ldots, X_{K-1}\}.
\]

Thus, \( A_i \) is defined to be a row for which \( S \) in Eq. 2 is minimized.

\[
S = \sum_{k=0}^{K-1} (X_k - A_{i,k})^2.
\]

(2)

Note that \( A_i \) may or may not be unique.

6.2 NNSearch() Parameters

N  N points to the root node of a KD tree.

X  X points to \( X \), the coordinates of the location for the search. The coordinates that are pointed to by \( X \) must consist of \( K \) values that correspond to the \( K \) independent variables that are associated with the KD tree specified by \( N \).
**r**

points to \( A_i \), a row that specifies the coordinates of a location for which the distance to the location specified by \( X \) is minimized. \( r \) is calculated by the NNSearch() function.

**S**

\( S \) specifies \( S \), the squared distance between the locations specified by \( X \) and \( r \). Although \( S \) is calculated by the NNSearch() function, \( S \) should be initialized to some value that is larger than the expected final value of \( S \) (typically some very large value).

**K**

\( K \) specifies \( K \), the number of independent dimensions that are included in the KD tree.

### 6.3 NNSearch() Example — Creating a Simple Voronoi Diagram

The following example code uses functions from the yBmp namespace, along with the NNSearch() function to create the Voronoi diagram that is presented in Fig. 3. The black dots in Fig. 3 show the locations of the points that were used to create the Voronoi diagram. The colored sections represent sets of points that have a common nearest neighbor among the points that make up the KD tree (i.e., the black dots). For this particular Voronoi diagram, the colors themselves represent the row number of the table that was used to create the KD tree (see Fig. 4).

```cpp
#include <cmath>
#include "y_bmp.h"
#include "y_kd_tree.h"

inline void Rainbow(unsigned char C[3], double x, double min, double max) {
    if (x<min){C[0]=C[1]=C[2]=0; return;}
    if (x>max){C[0]=C[1]=C[2]=255; return;}
    x=(1-(x-min)/(max-min))*8;
    C[0]=int((3<x&&x<5||x>7 ? -fabs(x/2-3)+1.5:5<=x&&x<=7?1:0)*255);
    C[1]=int((1<x&&x<3||5<x&&x<7? -fabs(x/2-2)+1.5:3<=x&&x<=5?1:0)*255);
    C[2]=int(( x<1||3<x&&x<5? -fabs(x/2-1)+1.5:1<=x&&x<=3?1:0)*255);
}

int main() {
    double*A[5],B[5][3]={4,1,0, 4,3,1, 6,2,2, 2,4,3, 8,4,4};
    for(int i=0;i<5;++i)A[i]=B[i];
    yKDTree::NODE<N=yKDTree::NewTree(A,A+5,2);
    int n=1000; image size will be 2n x n pixels
    for(int p=0; p<2*n;++p)
        for(int q=0; q<n;++q){
            double X[2]=(p*10./(2*n-1),q*5./(n-1));
            double S,*r; yKDTree::NNSearch(N,X,r,S=1E9,2);
            if(S<.002)memcpy(yBmp::GetPixel(I,p,q),BLACK,3);
            else Rainbow(yBmp::GetPixel(I,p,q),r[2],-1.5,5.5);
        }
yBmp::WriteBmpFile("voronoi.bmp", I);
yKDTree::DeleteTree(N), delete[]I;
}
```
The following code measures the performance of the NNSearch() function and compares it with the performance of the NNSearchExhaustive() function, which is defined in the example. The NNSearchExhaustive() function takes a brute-force approach to determining $A_i$ and $S_i$.

The yRandom namespace is used to generate pseudorandom numbers for the test. Time measurements represent the total time required to perform $10^7$ searches on tables with $2^n$ rows, where $n$ varies from 1 to 14.

When $K = 2$, the test shows that for tables with very few rows (somewhere around 32 or fewer) the brute-force method outperforms the KD-tree method. Figure 5 shows that as the value of $K$ increases, the minimum number of rows required for the KD-tree method to be advantageous increases as well.
#include <cstdio>
#include <ctime>
#include "y_kd_tree.h"
#include "y_random.h"

template<class T>T NNSearchExhaustive(/*<=====EXHAUSTIVE NEAREST-NEIGHBOR SEARCH*/
    T**a,T**b,/*<-------------------POINTERS TO STARTING AND ENDING ROWS*/
    T*X,/*<-----------------------------------POINTERS TO SEARCH COORDINATES*/
    T*&r,/*<------------------------POINTER TO NEAREST-NEIGHBOR ROW (CALCULATED)*/
    int K){/*<----------------------------------NUMBER OF INDEPENDENT DIMENSIONS*/
    double S=0;/*<*/
    for(int i=0;i<K;++i)S+=(a[0][i]-X[i])*(a[0][i]-X[i]);
    for(r=*a;a<b;++a){/*<*/
        double s=0;/*<*/
        for(int i=0;i<K;++i)s+=(*a)[i]-X[i]);
        if(s<S)r=*a,S=s;}
    return S;}

int main(){/*<------------------MEASURE THE PERFORMANCE OF THE NNSearch() FUNCTION*/
    const int N=1<<14,M=10000000,K=2;
    unsigned I[625];/*<-*/
    yRandom::Initialize(I,1);/*<....state of Mersenne twister*/
    double*A[N],B[N][K];/*<-*/
    for(int i=0;i<N;++i)A[i]=B[i];
    for(int i=0;i<N;++i)
        for(int k=0;k<K;++k)A[i][k]=yRandom::RandU(I,0,1);
    printf("          | NNSearchExhaustive()|     NN2DInterp()
    row    |---------------------|---------------------
    count   |   time   |    x     |   time   |    x
    |    (s)   |   avg.   |    (s)   |   avg.
    ---------|----------|----------|----------|------");/*<*/
    for(int m=2;m<=N;m*=2){/*<*/
        double s=0,t=clock();/*<&*/
        yRandom::Initialize(I,1);/*<...NNSearchExhaustive()*/
        for(int k=0;k<K;++k){/*<*/
            double X[K];/*<-*/
            for(int i=0;i<K;++i)X[i]=yRandom::RandU(I,0,1);
            double*r;/*<-*/
            NNSearchExhaustive(A,A+m,X,r,K);
            s+=*r;}
        printf("%7d   |%8.3f  |%9.6f |",m,(clock()-t)/CLOCKS_PER_SEC,s/M);
        s=0,t=clock(),yRandom::Initialize(I,1);/*<.........begin NNSearch() test*/
        yKDTree::NODE<double>*R=yKDTree::NewTree(A,A+m,K);
        for(int k=0;k<K;++k){/*<*/
            double X[K];/*<-*/
            for(int i=0;i<K;++i)X[i]=yRandom::RandU(I,0,1);
            double S=1E9;
            double*r;/*<-*/
            yKDTree::NNSearch(R,X,r,S,K);
            s+=*r;}
        yKDTree::DeleteTree(R);
        printf("%8.3f |%9.6f\n",(clock()-t)/CLOCKS_PER_SEC,s/M);}
}

}///~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
<table>
<thead>
<tr>
<th>row count</th>
<th>NNSearchExhaustive()</th>
<th>NN2DInterp()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time (s)</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>0.320</td>
<td>0.577847</td>
</tr>
<tr>
<td>4</td>
<td>0.380</td>
<td>0.372263</td>
</tr>
<tr>
<td>8</td>
<td>0.552</td>
<td>0.354816</td>
</tr>
<tr>
<td>16</td>
<td>0.710</td>
<td>0.481116</td>
</tr>
<tr>
<td>32</td>
<td>1.050</td>
<td>0.491685</td>
</tr>
<tr>
<td>64</td>
<td>1.760</td>
<td>0.499086</td>
</tr>
<tr>
<td>128</td>
<td>3.104</td>
<td>0.499204</td>
</tr>
<tr>
<td>256</td>
<td>5.630</td>
<td>0.498995</td>
</tr>
<tr>
<td>512</td>
<td>11.131</td>
<td>0.499451</td>
</tr>
<tr>
<td>1024</td>
<td>21.392</td>
<td>0.499869</td>
</tr>
<tr>
<td>2048</td>
<td>43.283</td>
<td>0.499950</td>
</tr>
<tr>
<td>4096</td>
<td>88.706</td>
<td>0.499952</td>
</tr>
<tr>
<td>8192</td>
<td>177.382</td>
<td>0.499965</td>
</tr>
<tr>
<td>16384</td>
<td>382.268</td>
<td>0.499971</td>
</tr>
</tbody>
</table>

Fig. 5  Brute-force and KD-tree methods compared for $K = 2$, 4, and 6
7. Example — Fermat’s Spiral

According to Vogel\textsuperscript{4} Eqs. 3 and 4, which define a set of points in polar coordinates that all lie on Fermat's spiral, can be used to model the patterns of seeds in sunflowers.

\begin{align}
  r_i &= R \sqrt{\frac{i}{m-1}}, \\
  \theta_i &= (3 - \sqrt{5})i\pi,
\end{align}

where \( R \) is the radius of a circle that contains all of the points, \( m \) is the total number of points, and \( 0 \leq i < m \).

The following code uses functions from the yBmp namespace, as well as the Rainbow() function from section 6.3, to create three Voronoi diagrams (Fig. 6) that are based on Eqs. 3 and 4 (one with \( m = 100 \), one with \( m = 200 \), and one with \( m = 600 \)). For each, \( R \) has been chosen to be 1/2 the width of the image.

```cpp
#include <cmath>
#include "y_bmp.h"
#include "y_kd_tree.h"

int main(){
  //......CREATE VORONOI DIAGRAMS BASED ON FERMAT'S SPIRAL
  for(int m=100,j=0,n=1000;j<3;++j,m*=j+1){//....image size will be n x n pixels
    double**A=new double*[m];/*<
    for(int i=0;i<m;++i)A[i]=new double[3];
    for(int i=0;i<m;++i){
      double R=.5*sqrt(i/(m-1.)),theta=(3-sqrt(5.))*i*3.141592653589793;
      x=R*cos(theta),y=R*sin(theta);
    yKDTree::NODE<double>*N=yKDTree::NewTree(A,A+m,2);
    unsigned char*B=yBmp::NewImage(n,n,255),BLACK[3]={0};
    for(int p=0;p<n;++p)
      for(int q=0;q<n;++q){
        double X[2]={p/(n-1.),q/(n-1.)};
        double S,*r;yKDTree::NNSearch(N,X,r,S=1E9,2);
        if(S<.00002)memcpy(yBmp::GetPixel(B,p,q),BLACK,3);
        else Rainbow(yBmp::GetPixel(B,p,q),r[2],0,m);
    yBmp::WriteBmpFile(j==0?"spiral1.bmp":j==1?"spiral2.bmp":"spiral3.bmp",B);
    for(int i=0;i<m;++i)delete[]A[i];/*&
  }
}
```

//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
8. Example — Comparing Average Distances to Sources

Suppose that the effectiveness of some physical phenomenon of interest is reduced as the distance from a source increases (such as signal strength from radio antennas). If $D(x, y)$ is defined to be the distance to the nearest source, then Eq. 5 can be used to calculate the average distance to the nearest source ($\bar{D}$) for some area of interest:

$$\bar{D} \equiv \frac{\iint D(x, y)dx\,dy}{A},$$

(5)

where $A$ is the total area over which $D(x, y)$ is integrated. By comparing $\bar{D}$ values for different sets of points, the relative coverage effectiveness between sets can be evaluated.

The following example code uses functions from the yBmp namespace, as well as the Rainbow() function from section 6.3, to create 3 Voronoi diagrams (Fig. 7). The first 2 are based on tables containing randomly selected points, while the third is based on a set of points that was purposely chosen to result in a small value for $\bar{D}$.
```cpp
#include "y_kd_tree.h"
#include "y_bmp.h"
#include "y_random.h"

int main(){
    const int m=9, n=1000; // image size will be n x n pixels
    double*A[m]; /*< */for(int i=0; i<m; ++i) A[i] = new double[3];
    double d=1./6;
    double O[m][2] = {3*d, 3*d, 5*d, 3*d, 5*d, 3*d, 5*d, 3*d, 5*d, 5*d, 3*d, 5*d, 3*d, 5*d, 3*d, 5*d, 3*d, 5*d, 5*d, 3*d, 5*d};
    unsigned I[625]; /*<-*/ yRandom::Initialize(I, 1); // state of Mersenne twister
    unsigned char*B = yBmp::NewImage(n, n, 255), BLACK[3] = {0};
    for(int J=0; J<3; ++J){
        for(int i=0; i<m; ++i)
            A[i][0] = J == 2 ? O[i][0] : yRandom::RandU(I, 0, 1),
            A[i][2] = i;
        yKDTree::NODE<double>*N = yKDTree::NewTree(A, A+m, 2);
        double s=0;
        for(int p=0; p<n; ++p)
            for(int q=0; q<n; ++q){
                double X[2] = {p*1. / (n-1), q*1. / (n-1)};
                double S, r; /*<-*/ yKDTree::NNSearch(N, X, r, S = 1E9, 2);
                s+=sqrt(S);
                if(S < .00002) memcpy(yBmp::GetPixel(B, p, q), BLACK, 3);
                else Rainbow(yBmp::GetPixel(B, p, q), r[2], -1, m);
            }
        yBmp::WriteBmpFile(!J ? "coverage1.bmp" : J == 1 ? "coverage2.bmp" : "coverage3.bmp", B);
        printf("CASE %d: D_bar= %8.5f\n", J+1, s/n/n);
    }
    for(int i=0; i<m; ++i) delete[] A[i]; /*< */ delete[] B;
} // ~~~~~YAGENAUT@GMAIL.COM ~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~~~
```

**OUTPUT:**

<table>
<thead>
<tr>
<th>CASE 1</th>
<th>D_bar= 0.27243</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASE 2</td>
<td>D_bar= 0.17623</td>
</tr>
<tr>
<td>CASE 3</td>
<td>D_bar= 0.12766</td>
</tr>
</tbody>
</table>

Fig. 7 Voronoi diagrams (case 1 left, case 2 center, case 3 right)
9. Example — Optimizing Areal Coverage

The following example code begins by recreating the “CASE 1” KD tree from the example presented in section 8. Next, a random-walk method is used to determine the optimal placement of an additional point. Figure 8 shows the original Voronoi diagram compared with the Voronoi diagram with the additional point.

The purpose of this example is to show a type of problem that might benefit from the use of KD trees. For this particular case, since the number of points in the table being searched is so small, it would likely have been slightly faster to use a brute-force method.

```cpp
#include "y_kd_tree.h"
#include "y_bmp.h"
#include "y_random.h"

int main(){
    #<===========================================OPTIMIZE AREAL COVERAGE
    double*T[10],*A[10];/*<for(int i=0;i<10;++i)A[i]=new double[3];
    unsigned I[625];/*<*/yRandom::Initialize(I,1);////////state of Mersenne twister
    int n=100;////////image size will be n x n pixels
    unsigned char*B=yBmp::NewImage(n,n,255),BLACK[3]={0};
    for(int i=0;i<10;++i)
        //A[i][0]=yRandom::RandU(I,0,1),A[i][1]=yRandom::RandU(I,0,1),A[i][2]=i;
        double s=1E99,x=0,y=0,st,xt,yt;
    for(int J=0;J<100;++J){
        st=0,xt=*T[9]=x+yRandom::RandN(I,0,.1),yt=T[9][1]=y+yRandom::RandN(I,0,.1);
        if(xt<0||xt>1||yt<0||yt>1)
            continue;
        yKDTree::NODE<double>*N=yKDTree::NewTree(T,T+10,2);
        for(int p=0;p<n;++p)
            for(int q=0;q<n;++q){
                double X[2]={p*1./(n-1),q*1./(n-1)};
                yKDTree::NNSearch(N,X,r,D=1E9,2);
                if(D<.00002)
                    memcpy(yBmp::GetPixel(B,p,q),BLACK,3);
                else
                    Rainbow(yBmp::GetPixel(B,p,q),r[2],-1,9);}
        yBmp::WriteBmpFile("optimized_coverage.bmp",B);
        yKDTree::DeleteTree(N),delete[]B;
    };//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
```
OUTPUT:

| J= 4  | D_bar=0.262436 | x= 0.183841 | y= 0.031927 |
| J= 7  | D_bar=0.256637 | x= 0.217818 | y= 0.054442 |
| J= 8  | D_bar=0.241756 | x= 0.324290 | y= 0.017688 |
| J=12  | D_bar=0.239188 | x= 0.303139 | y= 0.112939 |
| J=13  | D_bar=0.238673 | x= 0.321162 | y= 0.061664 |
| J=14  | D_bar=0.209072 | x= 0.597562 | y= 0.007720 |
| J=18  | D_bar=0.189872 | x= 0.633593 | y= 0.136296 |
| J=20  | D_bar=0.187058 | x= 0.876571 | y= 0.212578 |
| J=23  | D_bar=0.183453 | x= 0.797085 | y= 0.401234 |
| J=26  | D_bar=0.179227 | x= 0.711760 | y= 0.272756 |
| J=28  | D_bar=0.179046 | x= 0.769481 | y= 0.303093 |
| J=62  | D_bar=0.178932 | x= 0.752994 | y= 0.268364 |

Fig. 8   Voronoi diagrams of a table without and with a point added that minimizes $\bar{D}$

10. Code Summary

A summary sheet is provided at the end of this report. It presents the yKDTTree namespace, which contains the ColumnSort(), NewTree(), DeleteTree(), and NNSearch() functions and the NODE struct.
**yKDTree Summary**

### ColumnSort Performance

<table>
<thead>
<tr>
<th>Row Count</th>
<th>time</th>
<th>x</th>
<th>time</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0.0</td>
<td>0.5</td>
<td>0.5</td>
<td>2.5</td>
</tr>
<tr>
<td>2</td>
<td>2.5</td>
<td>5.5</td>
<td>4</td>
<td>4.5</td>
</tr>
</tbody>
</table>

### NNSearch Performance

![NNSearch Performance Chart](chart.png)

**Example - Creating a Simple Voronoi Diagram**

```cpp
#include <algorithm>//<-------------------------------stable_sort() FUNCTION
#include <ctime>    //<-------------------------------clock() FUNCTION
#include <y_kd_tree.h>    //<---yKDTree

inline bool Compare(pair<Point,Point> a, pair<Point,Point> b){
  return a.first < b.first;
}

double* a, double* b = NULL;

//<-----------------------------CREATE A VORONOI DIAGRAM USING NNSearch()
double S = 2.5, K = 3;

for (int i = 1; i <= S; i++) {
  double u = yRandom::RandU(I,0,1);
  double v = yRandom::RandU(I,0,1);
  Node* N = yKDTree::NewNode(u, v);
  N->x = u * 255;
  N->y = v * 255;
}

yKDTree::DeleteTree(R);
```

---

**y_kd_tree.h**

```cpp
#include <algorithm>
#include <algorithm>
inline double x, y;
inline void rand2d(double a, double b, double& x, double& y)
{
  x = a + (b - a) * yRandom::RandU(I, 0, 1);
  y = a + (b - a) * yRandom::RandU(I, 0, 1);
}
```
11. References


