Exhibit 33

```
    h41166
    s 00000/00045/00437
    d D 1.2 00/07/26 15:47:10 erling 2 1
    c cleanup
    e
    s 00482/00000/00000
    d D 1.1 00/07/26 15:36:26 erling 1 0
    c date and time created 00/07/26 15:36:26 by erling
    e
    u
    U
    f e 0
    t
    T
    I 1
//----------------------------------------------------------------------
//      KD is a library for Approximate Nearest Neighbor searching,
//      based on the use of standard and priority search in kd-trees
//      and balanced box-decomposition (bbd) trees.  Here are some
//      references:
//
//      kd-trees:
//       Friedman, Bentley, and Finkel, ``An algorithm for finding
//              best matches in logarithmic expected time," ACM
//              Transactions on Mathematical Software, 3(3):209-226, 1977.
//
//      priority search in kd-trees:
//       Arya and Mount, ``Algorithms for fast vector quantization,"
//              Proc. of DCC '93: Data Compression Conference, eds. J. A.
//              Storer and M. Cohn, IEEE Press, 1993, 381-390.
//
//      approximate nearest neighbor search and bbd trees:
//         Arya, Mount, Netanyahu, Silverman, and Wu, ``An optimal
//              algorithm for approximate nearest neighbor searching,"
//              5th Ann. ACM-SIAM Symposium on Discrete Algorithms,
//              1994, 573-582.
//----------------------------------------------------------------------

#ifndef KD_H
#define KD_H

//----------------------------------------------------------------------
//  basic includes
//----------------------------------------------------------------------
#include <stdlib.h>                     // standard libs
#include <stdio.h>                      // standard I/O (for NULL)
#include <math.h>                       // math includes
#ifdef unix
#include <values.h>                     // special values
```

Exhibit 10
Date: 9/4/19 Name: WOLD
Lana L. Loper, RMR, CRR, CCP, CSR No. 9667

```
#else
#define MAXFLOAT HUGE_VAL
#endif
#include "mfErrors.h"

#define KDversion     "0.1"            // KD version number

//----------------------------------------------------------------------
// KDbool
//        This is a simple boolean type. Although ANSI C++ is supposed
//        to support the type bool, many compilers do not have it.
//----------------------------------------------------------------------

// KD boolean type (non ANSI C++)
typedef enum
{
   KDfalse = 0,
   KDtrue = 1
} KDbool;

//----------------------------------------------------------------------
// Basic Types: KDcoord, KDdist, KDidx
//        KDcoord and KDdist are the types used for representing
//        point coordinates and distances. They can be modified by the
//        user, with some care. It is assumed that they are both numeric
//        types, and that KDdist is generally of an equal or higher type
//        from KDcoord. A variable of type KDdist should be large
//        enough to store the sum of squared components of a variable
//        of type KDcoord for the number of dimensions needed in the
//        application. For example, the following combinations are
//        legal:
//
//                KDcoord        KDdist
//                ---------------------------------
//                short          short, int, long, float, double
//                int            int, long, float, double
//                long           long, float, double
//                float          float, double
//                double         double
//
//        It is the user's responsibility to make sure that overflow does
//        not occur in distance calculation.
//
//        The code assumes that there is an infinite distance, KD_DIST_INF
//        (as large as any legal distance). Possible values are given below:
//
//            Examples:
//            KDdist:        double, float, long, int, short
//            KD_DIST_INF:   MAXDOUBLE, MAXFLOAT, MAXLONG, MAXINT, MAXSHORT
```

```
//
//
//       KDidx is a point index.  When the data structure is built,
//       the points are given as an array.  Nearest neighbor results are
//       returned as an index into this array.  To make it clearer when
//       this is happening, we define the integer type KDidx.
//
//----------------------------------------------------------------------

typedef      float    KDcoord;          /* coordinate data type */
typedef      float    KDdist;           /* distance data type */
typedef int  KDidx;                 /* point index */

                                    /* largest possible distance */
/*const KDdistKD_DIST_INF = MAXFLOAT;*/
#define KD_DIST_INF (KDdist) MAXFLOAT
//----------------------------------------------------------------------
// Self match?
//       In some applications, the nearest neighbor of a point is not
//       allowed to be the point itself.  This occurs, for example, when
//       computing all nearest neighbors in a set.  By setting the
//       parameter KD_ALLOW_SELF_MATCH to KDfalse, the nearest neighbor
//       is the closest point whose distance from the query point is
//       strictly positive.
//----------------------------------------------------------------------

/*const KDbool        KD_ALLOW_SELF_MATCH= KDtrue;*/
#define KD_ALLOW_SELF_MATCH (KDbool) KDtrue

//----------------------------------------------------------------------
//  Norms and metrics:
//       KD supports any Minkowski norm for defining distance.  In
//       particular, for any p >= 1, the L_p Minkowski norm defines the
//       length of a d-vector (v0, v1, ..., v(d-1)) to be
//
//                (|v0|^p + |v1|^p + ... + |v(d-1)|^p)^(1/p),
//
//       (where ^ denotes exponentiation, and |.| denotes absolute
//       value).  The distance between two points is defined to be
//       the norm of the vector joining them.  Some common distance
//       metrics include
//
//                Euclidean metric      p = 2
//                Manhattan metric      p = 1
//                Max metric            p = infinity
//
//       In the case of the max metric, the norm is computed by
//       taking the maxima of the absolute values of the components.
//       KD is highly "coordinate-based" and does not support general
```

```
//      distances functions (e.g. those obeying just the triangle
//      inequality).  It also does not support distance functions
//      based on inner-products.
//
//      For the purpose of computing nearest neighbors, it is not
//      necessary to compute the final power (1/p).  Thus the only
//      component that is used by the program is |v(i)|^p.
//
//      KD parameterizes the distance computation through the following
//      macros.  (Macros are used rather than procedures for efficiency.)
//      Recall that the distance between two points is given by the length
//      of the vector joining them, and the length or norm of a vector v
//      is given by formula:
//
//              |v| = ROOT(POW(v0) # POW(v1) # ... # POW(v(d-1)))
//
//      where ROOT, POW are unary functions and # is an associative and
//      commutative binary operator satisfying:
//
//          **  POW:  coord         --> dist
//          **  #:    dist x dist   --> dist
//          **  ROOT:dist (>0)      --> double
//
//      For early termination in distance calculation (partial distance
//      calculation) we assume that POW and # together are monotonically
//      increasing on sequences of arguments, meaning that for all
//      v0..vk and y:
//
//      POW(v0) #...# POW(vk) <= (POW(v0) #...# POW(vk)) # POW(y).
//
//      Due to the use of incremental distance calculations in the code
//      for searching k-d trees, we assume that there is an incremental
//      update function DIFF(x,y) for #, such that if:
//
//                  s = x0 # ... # xi # ... # xk
//
//      then if s' is s with xi replaced by y, that is,
//
//                  s' = x0 # ... # y # ... # xk
//
//      can be computed by:
//
//                  s' = s # DIFF(xi,y).
//
//      Thus, if # is + then DIFF(xi,y) is (yi-x).  For the L_infinity
//      norm we make use of the fact that in the program this function
//      is only invoked when y > xi, and hence DIFF(xi,y)=y.
//
//      Finally, for approximate nearest neighbor queries we assume
```

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase®
Smarter legal research.