UNIVERSITY OF CALIFORNIA, MERCED

Efficient Open Source Lidar for Desktop Users

A Thesis submitted in partial satisfaction of the requirements
for the degree of Master of Science

in

Environmental Systems

by

Jacob P. Flanagan

Committee in charge:

Professor Qinghua Guo, Chair
Professor Joshua Viers
Professor Thomas Harmon

2015

The Thesis of Jacob P. Flanagan is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Thomas Harmon

Joshua Viers

Qinghua Guo

Chair

University of California, Merced
2015

ACKNOWLEDGMENTS

This research herein was the result of processing lidar data for purpose of creating project deliverables as required by the Sierra Nevada Adaptive Management Project (SNAMP) and National Critical Zone Observatory (CZO) program. Funding was provided by the National Science Foundation (NSF).

Table of Contents

List of Tables

List of Figures

Abstract
Lidar -- **Li**ght **D**etection **a**nd **R**anging -- is a remote sensing technology that utilizes a device similar to a rangefinder to determine a distance to a target. A laser pulse is shot at an object and the time it takes for the pulse to return in measured. The distance to the object is easily calculated using the speed property of light. For lidar, this laser is moved (primarily in a rotational movement usually accompanied by a translational movement) and records the distances to objects several thousands of times per second. From this, a 3 dimensional structure can be procured in the form of a point cloud. A point cloud is a collection of 3 dimensional points with at least an *x*, a *y* and a *z* attribute. These 3 attributes represent the position of a single point in 3 dimensional space. Other attributes can be associated with the points that include properties such as the intensity of the return pulse, the color of the target or even the time the point was recorded. Another very useful, post processed attribute is point classification where a point is associated with the type of object the point represents (i.e. ground.).

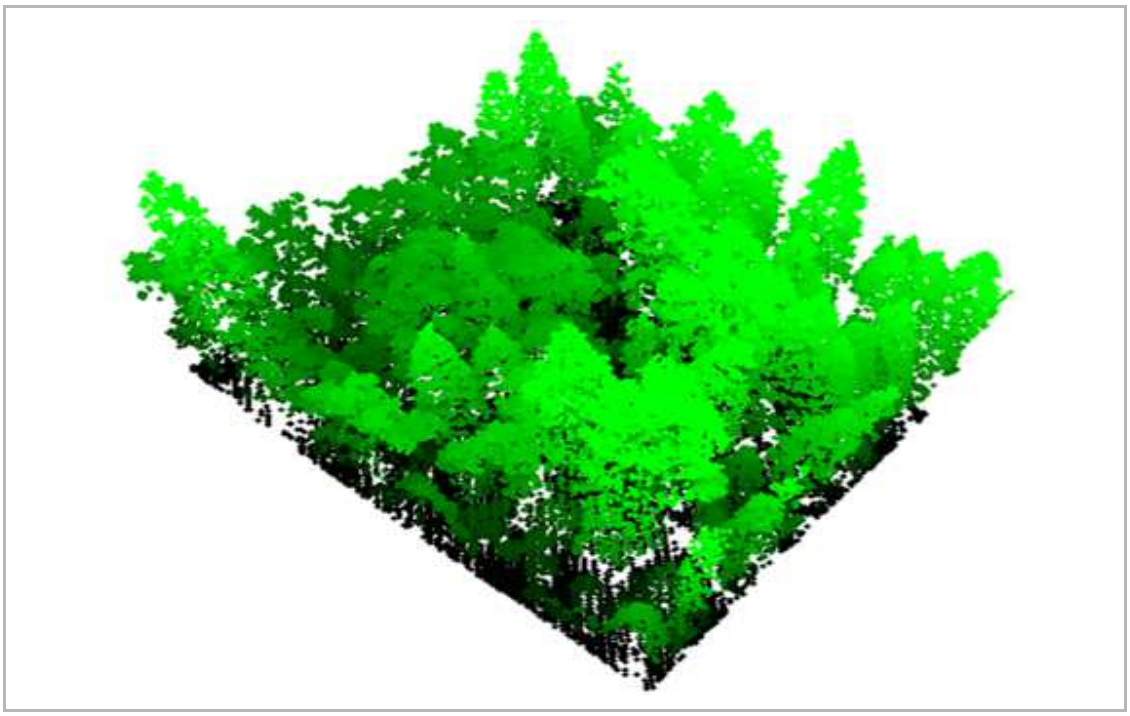


Figure 1: A Lidar Point Cloud

Lidar has gained popularity and advancements in the technology has made its collection easier and cheaper creating larger and denser datasets. The need to handle this data in a more efficiently manner has become a necessity; The processing, visualizing or even simply loading lidar can be computationally intensive due to its very large size. Standard remote sensing and geographical information systems (GIS) software (ENVI, ArcGIS, etc.) was not originally built for optimized point cloud processing and its implementation

is an afterthought and therefore inefficient. Newer, more optimized software for point cloud processing (QTModeler, TopoDOT, etc.) usually lack more advanced processing tools, requires higher end computers and are very costly. Existing open source lidar approaches the loading and processing of lidar in an iterative fashion that requires implementing batch coding and processing time that could take months for a standard lidar dataset. This project attempts to build a software with the best approach for creating, importing and exporting, manipulating and processing lidar, especially in the environmental field. Development of this software is described in 3 sections - (1) explanation of the search methods for efficiently extracting the “area of interest” (AOI) data from disk (file space), (2) using file space (for storage), budgeting memory space (for efficient processing) and moving between the two, and (3) method development for creating lidar products (usually raster based) used in environmental modeling and analysis (i.e.: hydrology feature extraction, geomorphological studies, ecology modeling, etc.).

1. Introduction

1.0 LASer Format (Lidar format)

The standard format for saving lidar to storage (file space) is using a binary file type known as LAS[1]. Since this format is what most people are comfortable with (for receiving, storing and sharing), this software sets out to preserve the original lidar files and utilize them without creating a separate, redundant, copy and thus it is important to understand the LASer format. This format has a particular structure that a reader/scanner must adhere to read the data properly. An illustration of the LAS file structured is shown in figure 2.

[1] "LASer (LAS) File Format Exchange Activities - asprs." 2011. 3 Jun. 2015 <http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html>

Figure 1.1: Outline of the LAS File Structure

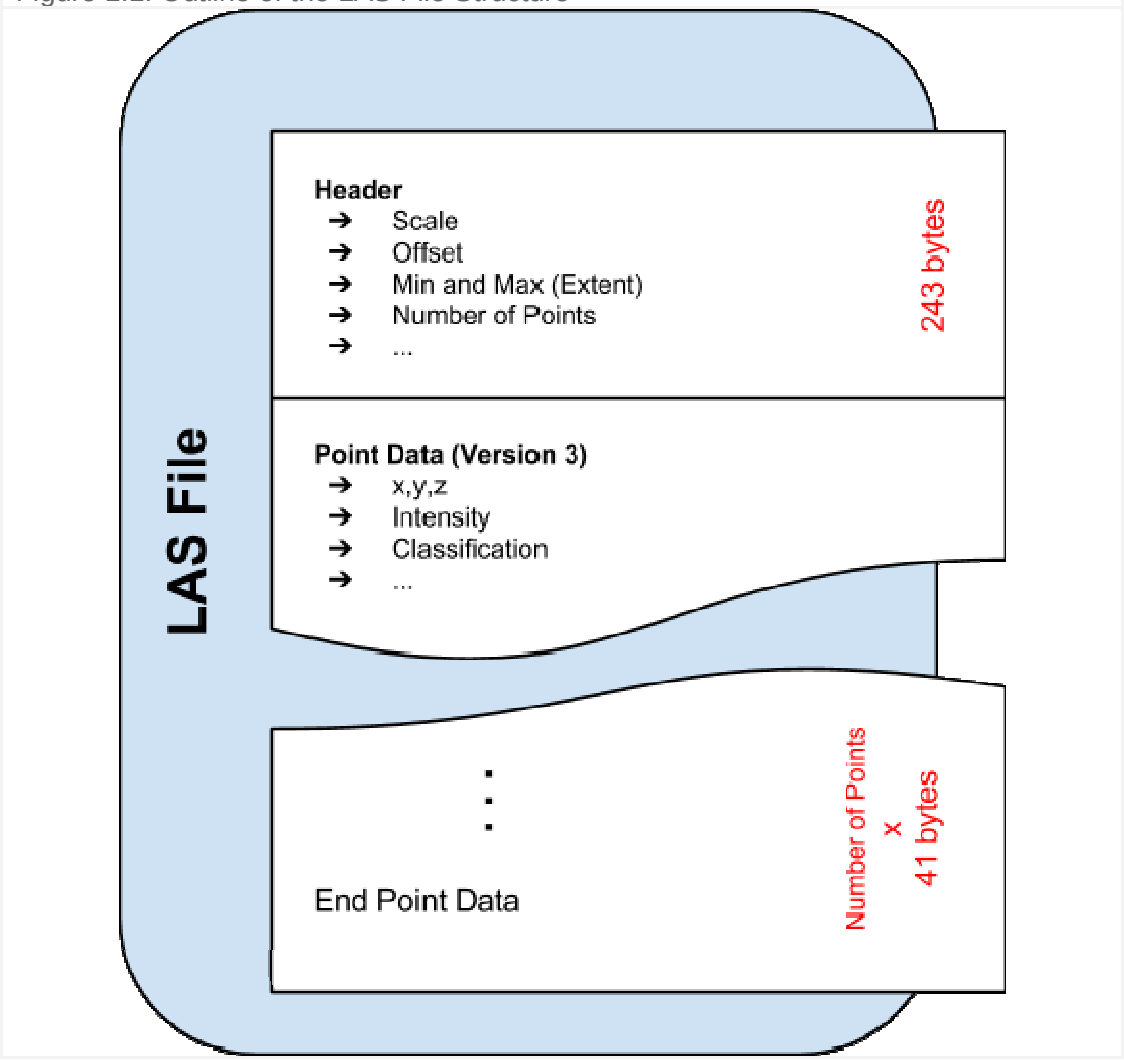


Figure 2: Outline of the LAS File Structure

The first part of the file is called the header block and contains information about the points contained within. This includes the extent (minimum and maximum in the x, y and z directions), the number of points, the scale and offset and other related information about the lidar point cloud. Table 1 defines the information contained within the header.

Table 1: LASer Header

Item	Format	Size
File Signature (“LASF”)	char[4]	4 bytes
File Source ID	unsigned short	2 bytes
Global Encoding	unsigned short	2 bytes

Project ID - GUID data 1	unsigned long	4 bytes
Project ID - GUID data 2	unsigned short	2 byte
Project ID - GUID data 3	unsigned short	2 byte
Project ID - GUID data 4	unsigned char[8]	8 bytes
Version Major	unsigned char	1 byte
Version Minor	unsigned char	1 byte
System Identifier	char[32]	32 bytes
Generating Software	char[32]	32 bytes
File Creation Day of Year	unsigned short	2 bytes
File Creation Year	unsigned short	2 bytes
Header Size	unsigned short	2 bytes
Offset to point data	unsigned long	4 bytes
Number of Variable Length Records	unsigned long	4 bytes
Point Data Format ID (0-99 for spec)	unsigned char	1 byte
Point Data Record Length	unsigned short	2 bytes
Number of point records	unsigned long	4 bytes
Number of points by return	unsigned long[7]	28 bytes
X scale factor	Double	8 bytes
Y scale factor	Double	8 bytes
Z scale factor	Double	8 bytes
X offset	Double	8 bytes
Y offset	Double	8 bytes
Z offset	Double	8 bytes
Max X	Double	8 bytes
Min X	Double	8 bytes
Max Y	Double	8 bytes
Min Y	Double	8 bytes
Max Z	Double	8 bytes
Min Z	Double	8 bytes
Start of Waveform Data Packet Record	Unsigned long	8 bytes

Following the header block is the point data comprised of multiple points. The format of a single points is outlined in table 2 (Version 3).

Table 2: LASer Point

Item	Format	Size
X	long	4 bytes
Y	long	4 bytes
Z	long	4 bytes
Intensity	unsigned short	2 bytes
Return Number	3 bits (bits 0,1,2)	3 bits
Number of Returns (given pulse)	3 bits (bits 3,4,5)	3 bits
Scan Direction Flag	1 bit (bit 6)	1 bit
Edge of Flight Line	1 bit (bit 7)	1 bit
Classification	unsigned char	1 byte
Scan Angle Rank (-90 to +90) – Left side	char	1 byte
User Data	unsigned char	1 byte
Point Source ID	unsigned short	2 bytes
GPS Time	double	8 bytes
Red	unsigned short	2 bytes
Green	unsigned short	2 bytes
Blue	unsigned short	2 bytes

To read this data, the header must be analyzed to determine the appropriate scaling and translations applied to the points. The following structure and method is used to read an LAS file:

```
struct LASreader( ) {

    file lasfile;
    Header h;     // Header structure that holds all header
                  //information

    //Method
    Point3 read( long i ){          // read at an index
        lasfile.seek( 243 + 41*i ); // change read position to
                                    // beginning
                                    // of the indexed point. Must
                                    // skip header (243 bytes) and
                                    // previous points
                                    // (41 bytes each)
        return populate_point( lasfile.read(41) ); // read 41 bytes
                                                   // and return a
                                                   // populated point
of
```

```
                                                    // Point3 type

};

// initialize the reader
LASreader LAS_read( file lasfile ){

    LASreader r = new LASreader;   // allocate space for the reader
    populate_header( r, lasfile ); // read all header information and
                                   // store into header structure
    r.lasfile = lasfile;           // associate the file with the
reader
}
```

1.1 Software

The “go to” GIS and remote sensing software like ArcGIS and ENVI have integrated lidar tools into its interface, and should probably be credited for introducing the lidar concept to a large number normal users. While these software are considered a standard and are very good for grid based analysis, their efficiently falls short when handling data that approaches sizes in the terabytes. Until version 10.1, ArcGIS was based on a 32-bit and a memory usage limitation of 2GB, even loading very large raster files can become difficult (lidar is even larger). ENVI allows users to create add-ons in IDL (interactive data language) making it easy to implement new tools, however, this is a high level language that suffers from the same performance issues as other high level languages. Both of these software are closed source, so lidar implementation is really dependent on the developers and it can be very difficult to integrate a new concepts into established software. Handling lidar is actually quite simple, just computationally intensive, so finding ways to make it more efficient would be ideal.

When lidar was young, processing datasets wasn’t as much of an issue because point clouds were less dense (we have lidar datasets from 2010 that had a point density of about 10 points per m^2). However, lidar equipment has become cheaper, collection has become easier and the demand for a finer datasets has increased causing datasets to be much larger in size (our new lidar sets can contain thousands of points per m^2). The classic (and simplest) method for processing lidar is to move over the point cloud iteratively (point by point), but higher point densities found in newer lidar data has made processing in this fashion nearly impossible due to the computational demand. Inherent of the first law of geography, like things are likely to be closer together[2], in most cases we process data with the spatial component as a priority when extracting a subset of points from a large point cloud. One solution is to implement a spatial database like postGIS that uses this concept, however, this is not a fully transparent solution; users

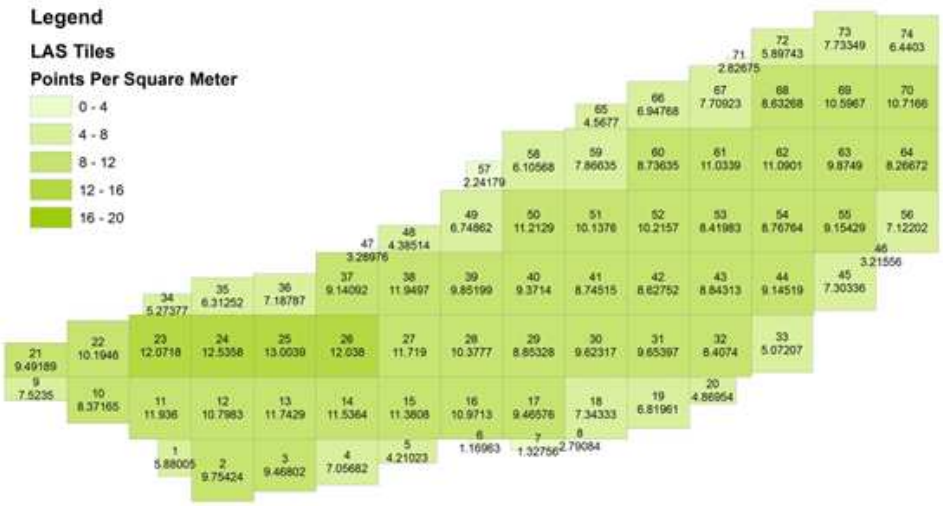
[2] Tobler, Waldo. "On the first law of geography: A reply." *Annals of the Association of American Geographers* 94.2 (2004): 304-310.

would essentially have to load their data (essentially creating a redundant copy) create tools to extract the data to a processing software and maintain the database. So development of extraction algorithm that utilizes an efficiency concept and is fully (or nearly) transparent while providing some necessary methods is ideal; the open source software outlined here attempts to do that.

1.2 LAS Tiling

Lidar files are usually very large and delivered as several smaller files called tiles. An interval extent is used to split the entire lidar dataset into the tiles. This smaller size makes it more convenient to load and search through individual files when trying to identify relevant data[3]. This process is simple enough, but can be time consuming as a through tiling involves reading every point (and every file if there are multiple files) to find points that fall within a particular tile extent. All points should be read at a low level (directly from disk) as the space requirements for loading into memory are usually too demanding. First, a common tile range should be established for every file (ex: 1000m x 1000m). Then the extent of the entire dataset is determined and the tiling range is used to create the smaller extents for the tiled files. Finally, each LAS file is read and distributed into the proper tile.

[3] Chen, Qi. "Airborne lidar data processing and information extraction." *Photogrammetric engineering and remote sensing* 73.2 (2007): 109.



Demonstrates the tiling of a larger lidar dataset.

Figure 3: LAS Tiles

2. Search Methods - Organized point clouds, grid searching and quadtrees

One method for improving the spatial searching across a point cloud is to create an organized point cloud from the lidar in which the points are restructured into something that resembles an organized image. This would give you speeds similar to processing an image, but potentially has the effect of losing useful information if downscaling many points to a single gridded point[4]. It is also sometimes hard to determine an appropriate resolution for an organized point cloud, especially across differently scaled projects that may be using the same lidar dataset (i.e.: a fire model using coarsely scaled lidar products vs a plot level terrain change analysis using finely scaled digital terrain models). In this case, several new organized datasets would have to exist which could be very time consuming to create and storage intensive. Instead, a hybrid method that combines the speed of image searching and the ability to utilize all points is ideal. Some

[4] Fabio, Remondino. "From point cloud to surface: the modeling and visualization problem." *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences* 34.5 (2003): W10.

indexing methods analyzed here are known as grid and quadtree indexing. These spatial index methods develop alternative ways of determining which points satisfy a spatial search and are likely faster. These index methods were chosen for this study because of their ease of use and implementation and their ability to refer to the original lidar file. Points are referenced using an index list that tracks their binary position in the original LAS file as illustrated by the figure 4.

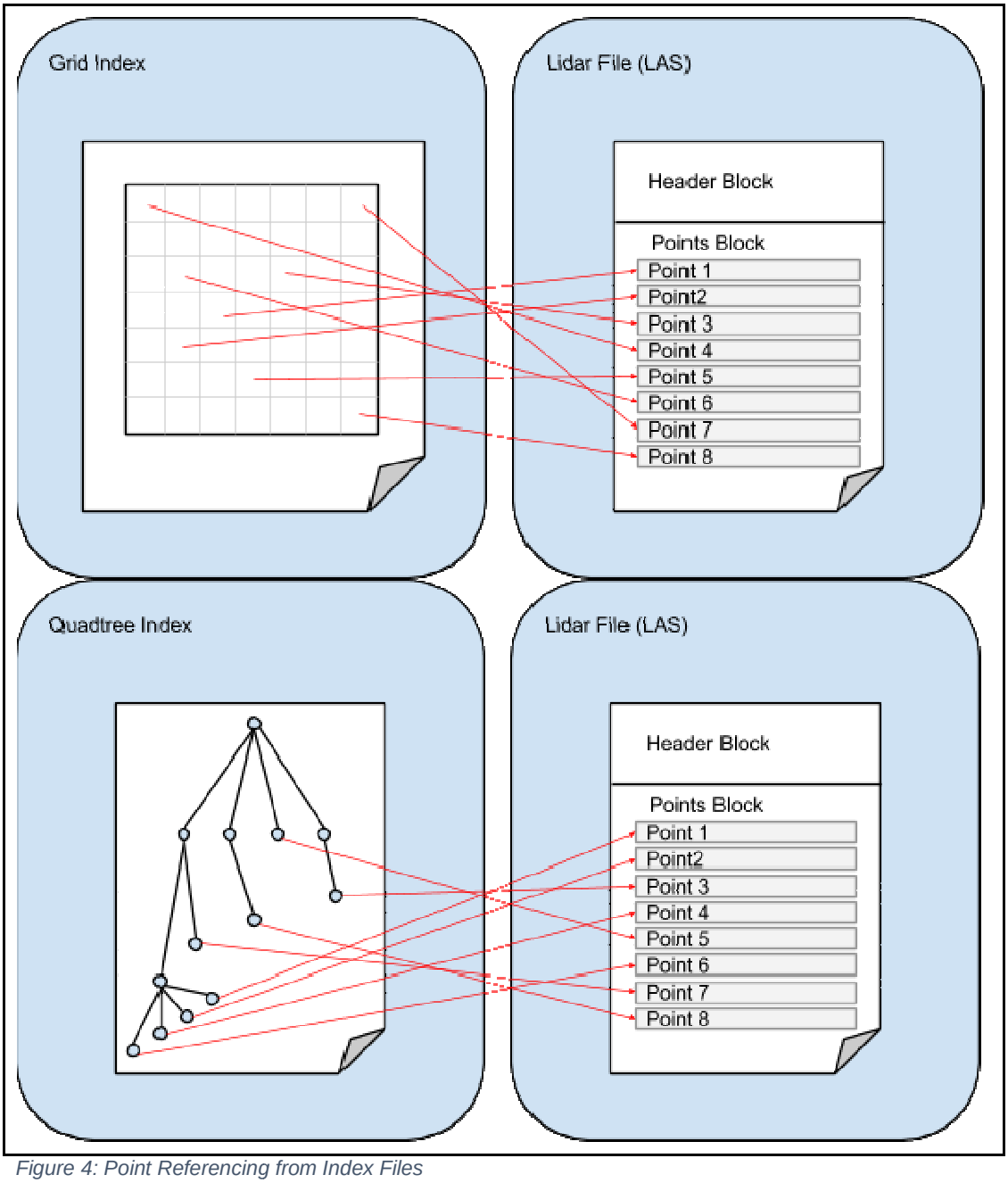


Figure 4: Point Referencing from Index Files

2.1 Grid-based Indexing

Grid-based indexing uses a grid to index the original points. This method overlays the extent of the points within an LAS file with a grid at a particular resolution[5]. A list of references for each point that fall within each grid cell is created. Figure 5 illustrates how the points are distributed within the grid index.

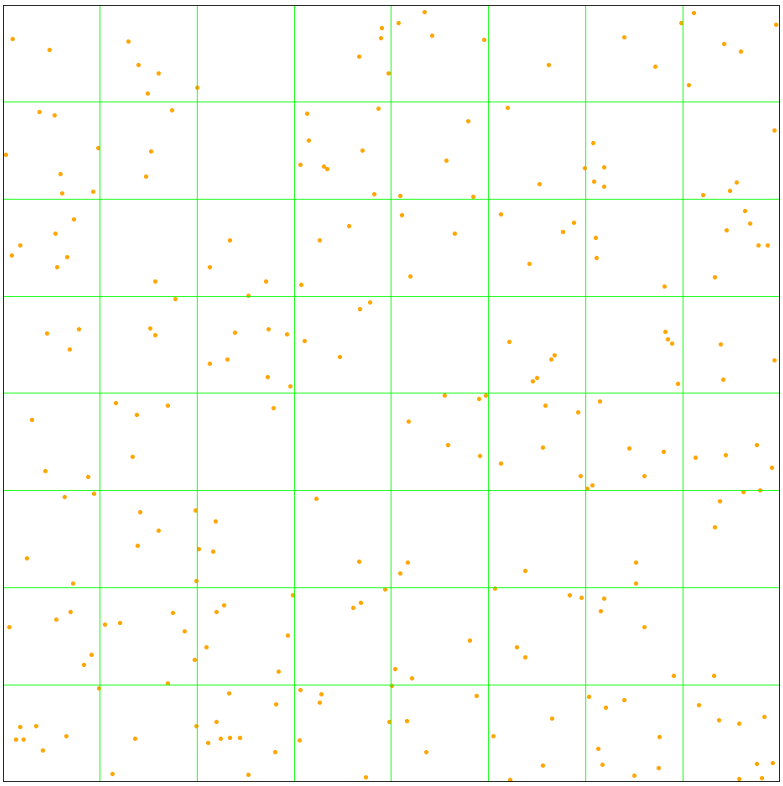


Figure 5: Grid Index Representation

Given a search location, the nearest cell can be quickly identified by traversing the rows and columns. However, choosing a proper resolution can be difficult (a good resolution should be based on the point density to select the maximum allowable points within a single cell) and grid sizes can grow quite large at very fine resolutions.
Programmatically, the grid-based index is structured as follows:

```
// Demonstrated as a constructor
```

[5] Sajjanhar, Atul, and Guojun Lu. "A grid-based shape indexing and retrieval method." *Australian Computer Journal* 29.4 (1997): 131-140.

```
struct grid_index( float x, float y, float res, long nx, long ny ) {

    float x = x;  // origin of grid
    float y = y;  // origin of grid
    float res = res;

    vector< unsigned long > grid[xn][yn];  // multidimensional
                                           // array of vectors
}
```

The grid variable within the grid index structure has 2 dimensions of type vector. These dimensions are x and y positions, respectively. The vector is of variable type unsigned long integer for storing the list of indexed points that fall within that particular cell. Vectors can be varied in length and are used because the number of points that fall within a grid cell are likely different from cell to cell. When creating the grid index, each point is analyzed to determine which cell it belongs to. The following function demonstrates the initialization of a grid index:

```
grid_index create_grid_index( LASreader r, float x, float y, float
res, int nx, int ny) {

    grid_index g = new grid_index( x, y, res, nx, ny ); // initialize
                                                        // the index

    for ( int i = 0; i < r.h.number_of_point_records; i++ ){

        Point3 p = r.read(i);      // load the current point
        int x = round(p.x);        // determine position in grid the
        int y = round(p.y);        // point should belong to
        g.grid[x][y].push_back(i); // track the index

    }

    return g;
}
```

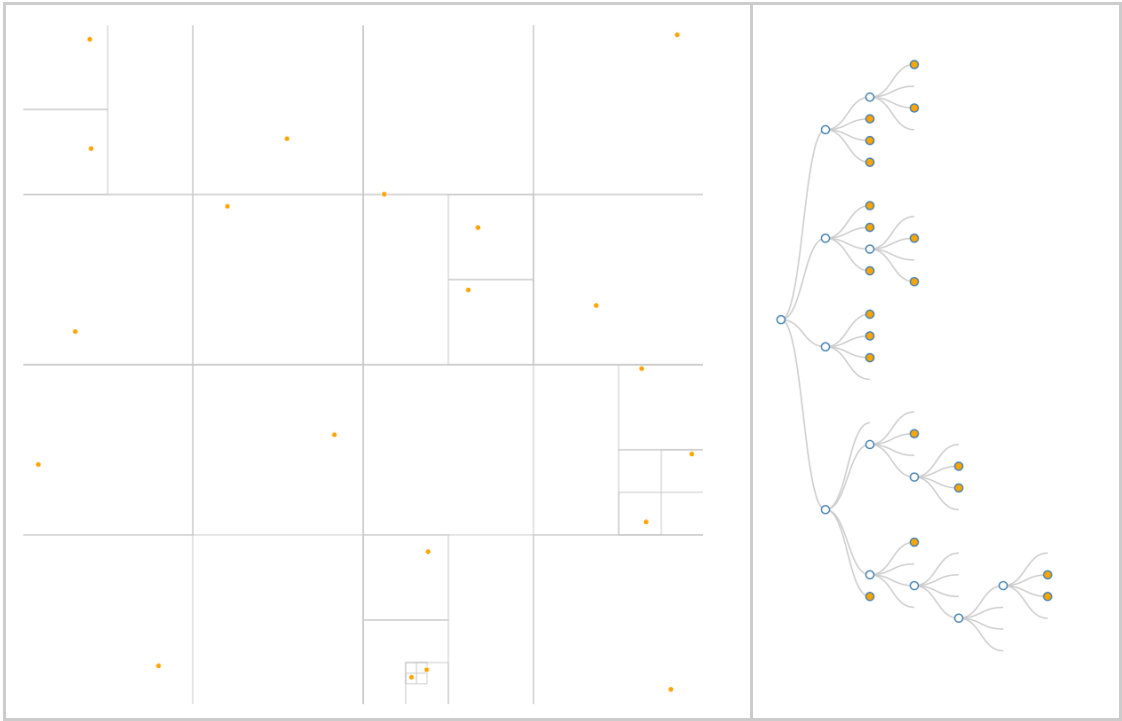
Here, The Lidar reader structure is used to start reading points from the LAS file. After the grid index is initialized, every point is iterated through to determine which cell is should be associated with.

2.2 Quadtrees

Another method for spatial searching is known as quadtree indexing[6]. It works by recursively splitting the extent of a lidar file into 4 quadrants until a particular hierarchical level (depth) is reached or until a desired amount of points fall within each quadrant,

[6] Tayeb, Jamel, Özgür Ulusoy, and Ouri Wolfson. "A quadtree-based dynamic attribute indexing method." *The Computer Journal* 41.3 (1998): 185-200.

depending on the implementation. The quadtree implementation is illustrated in the figure 6.



Quadtree representation for 20 points and the corresponding tree structure.

Figure 6: Quadtree Quadrants and Tree Structure

2.2.1 Bin and Depth Termination

For the purposed of this study, there are two forms of termination for a point-based quadtree (the base case for the recursion). The first has a threshold on the depth (or number of levels) the quadtree can have referred to here as depth termination. This means that the deepest quadrants can contain multiple and differing amount of point references. For the second form, each terminal quadrant is considered a bin that can hold *n* number of elements (point references) and known here as bin termination. As opposed to depth termination, bin termination has an unrestricted depth. Programmatically, the quadtree structure can be the same for both types of termination, but are used differently. For the terminal quadrants, bin termination will be limited by how many point references are allowed inside a particular quadrant whereas depth termination will be limited by maximum depth. The structure is outlined below:

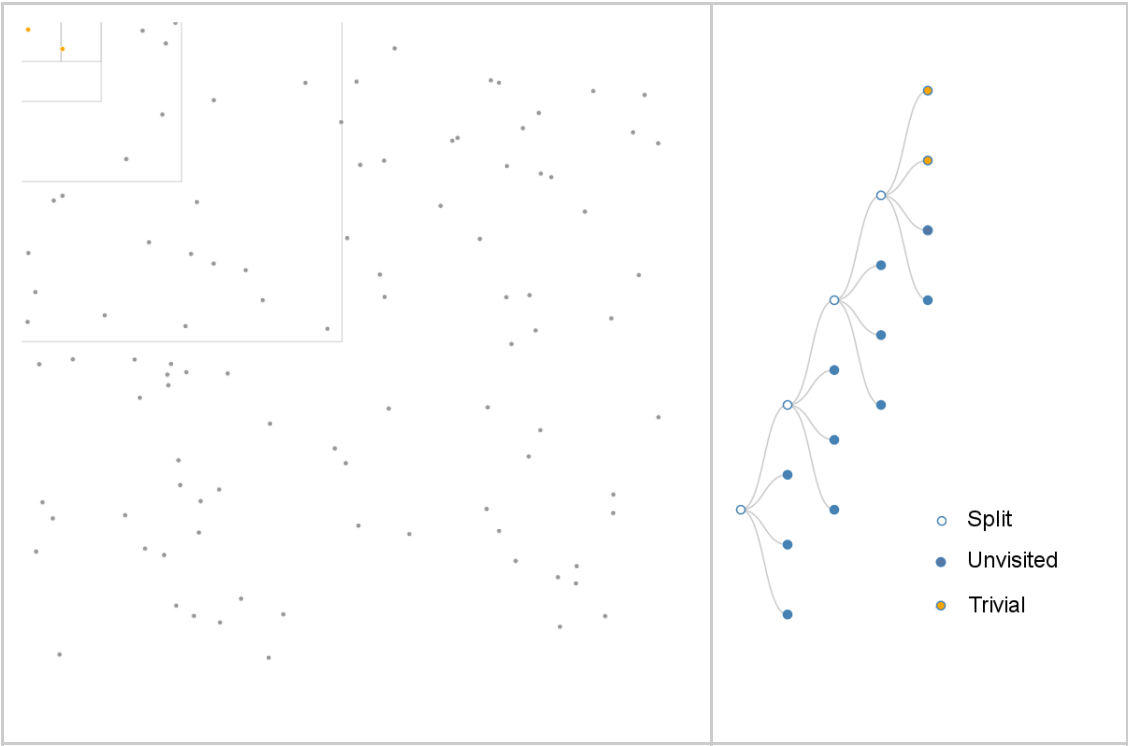
```
// quadrant node structure
struct quad( ) {
```

```
    //information about this quad
    extent e;
    int depth;

    quad * q1;
    quad * q2;
    quad * q3;
    quad * q4;

    bool is_terminal = false; //if this quad is another
                              // extension (false) or root (true)
    vector <long> * indices;  //for termination
};

// quadtree structure
struct quadtree( ) {

    extent e;  //the extent of the data being represented
    quad q;    //head of quadtree

    // methods
    bool is_in_quad( extent e, quad q ); //determine if extent
overlaps

};
```

In the case of bin termination, when creating the quadtree, the number of point references stored in the vector indices (under the quadrant structure) is limited by a user defined bin size. If the number of point references exceed the bin size, the quadrant will be split into the four quadrant pointers. In the case of depth termination, when the defined maximum depth is reached, all remaining points will stored into that quadrant.

Insertion Methods

Initializing the quadtree requires the point cloud be recursively partitioned into 4 quadrants until termination is satisfied. This is illustrated for the first two points in figure 7.



The first quadrant is recursively traversed until termination is determined	The resultant tree at the current traversal

Figure 7: Insertion Procedure

During insertion a subset of points are taken from the point cloud using the extent of each quadrant node. If the number of points within that subset is equal to or less than the threshold set on the bin size, those points are referenced in that particular node. The following code snippet demonstrates this insertion method.

```
void insert( LASreader r, quad q, int bin_size ){

    // Reference points that fall in extent e
    vector <long> * indices = new vector <long>;

    // fill the index vector
    for ( int i = 0; i < r.h.number_of_point_records; i++ ){

        if ( is_in_extent( r.read( i ), q.e ) )
            indices.push_back (i);

    }

    //check if extracted points will fit into the bin
    if ( indices.size() <= bin_size ) { // BASE CASE
```

```
        q.indices = indices;
        q.is_terminal = true;

    } else {

        // split quadrant into 4 new quadrants
        array < extent, 4 > e_arr = split_quad_extent ( e );

        //q1
        q.q1.extent = e_arr[0];
        q.q1.depth = q.depth+1;
        insert( r, q.q1, bin_size );

        //q2
        q.q2.extent = e_arr[1];
        q.q2.depth = q.depth+1;
        insert( r, q.q1, bin_size );

        //q3
        q.q3.extent = e_arr[2];
        q.q3.depth = q.depth+1;
        insert( r, q.q1, bin_size );

        //q4
        q.q4.extent = e_arr[3];
        q.q4.depth = q.depth+1;
        insert( r, q.q1, bin_size );

    }

}

quadtree create_bin_quadtree ( file lasfile, int bin_size ) {

    LASreader r = LAS_read( lasfile );       // initialize the reader
    quadtree qt = new quadtree();            // initialize the
quadtree

    //Kick off the recursion
    insert( r, qt.q, bin_size );              // start insertion
}
```

Reiterating, the insert function will recursively call itself until the number of points within a quadrant is less than the threshold set on the bin as defined by bin_size. This is decided by the conditional statement for base case; the base case is satisfied if the indices.size() (these are the point references that fall within the current quadrant) is less than the bin_size, and if true, the indices will be stored at the quadrant and the function will return. If the base case is unsatisfied, the current quadrant extent is split into 4 equal quadrants and the insertion function is recursively called on all 4 new quadrants.

For depth termination, the insert function is similar aside from checking on the bin size. Programmatically, this looks like the following:

```
void insert( LASreader r, quad q, int depth ){

    // Reference points that fall in extent e
    vector <long> * indices = new vector <long>;

    for ( int i = 0; i < r.h.number_of_point_records; i++ ){

        if ( is_in_extent( r.read( i ), q.e ) )
            indices.push_back (i);

    }

    //check if extracted points will fit into the bin
    if ( q.depth == depth || indices.size() < 2 ) { // base case

        q.indices = indices;

    } else {

        // create new quadrant extents
        array < extent, 4 > e_arr = split_quad_extent ( e );

        //q1
        q.q1.extent = e_arr[0];
        q.q1.depth = q.depth+1;
        insert( r, q.q1, bin_size );

        //q2
        q.q2.extent = e_arr[1];
        q.q2.depth = q.depth+1;
        insert( r, q.q1, bin_size );

        //q3
        q.q3.extent = e_arr[2];
        q.q3.depth = q.depth+1;
        insert( r, q.q1, bin_size );

        //q4
        q.q4.extent = e_arr[3];
        q.q4.depth = q.depth+1;
        insert( r, q.q1, bin_size );

    }

}

quadtree create_depth_quadtree ( file lasfile, int depth ) {
```

```
    LASreader r = LAS_read( lasfile );       // initialize the reader
    quadtree qt = new quadtree();            // initialize the
quadtree

    // Kick off the recursion
    insert( r, qt.q, depth );              // Start insertion
}
```

This time, the insert function tests whether the depth of the current quadrant node is equal to the maximum allowed depth to determine base case; base case is satisfied when `q.depth` (the current quadrant's depth) is equal to `depth` (the threshold), all indices that fall within this quadrant's extent will be saved to `q.indices` and the function will return or if there is just one point within the quadrant (we don't need to traverse any further if there is just one point). If the base case is unsatisfied, the current quadrant extent is split into 4 equal quadrants and the insertion function is recursively called on all 4 new quadrants.

It must be noted that it does take time to create an index. However, in all cases here, the index only needs to be created once and performance of the index methods are analyzed after they have been created.

2.3 Search Methods

To compare the different methods, the speed of each can be analyzed using a common search that extracts a subset of points that fall within a search extent[7]. Figure 8 illustrates a point cloud and a potential search extent.

[7] Piegl, Les A, and Wayne Tiller. "Algorithm for finding all k nearest neighbors." *Computer-Aided Design* 34.2 (2002): 167-172.

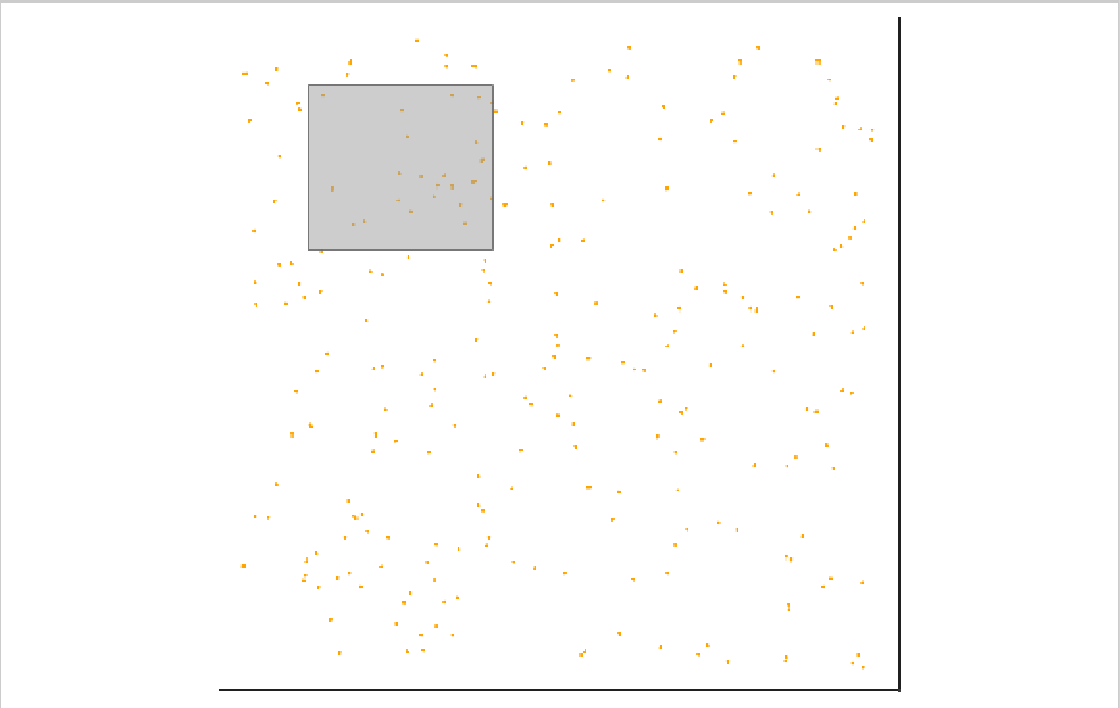


Figure 8: Point Cloud with a Search Extent

2.3.1 Qualitative Comparison

The algorithmic efficiency of each method can only be estimated, because there are too many factors that account for the speed of each which include the physical components of the computer (CPU speed, disk speed, RAM size and speed, etc.) and the point cloud spread (clustering). Each method takes advantage of different aspects about the cloud and the machine processing it to increase speed or save space. Analytically, computation methods are often described using the worst case scenario in Big-O notation[8]. For the indexing methods in this paper, this is outlined in table 3.

Table 3: Algorithmic Efficiency for Search - Big-O

Method	Iterative (non-indexed)	Grid Indexing	Quadtree	
			Level Termination	Bin Termination
Big-O speed	O(n)	O(1)	O(log n)	O(log n)

[8] Wing, Jeannette M. "Computational thinking." *Communications of the ACM* 49.3 (2006): 33-35.

Since the iterative approach requires a search to read each point every time the search is performed, it's worst case is O(n) where n is the number of points with a dataset. This is observed in the case of searching the dataset for points that fall within search extent; all points within the dataset must be analyzed. For a grid index search, points can be identified using cells that overlap with a search extent. This is done very quickly by simply traversing to the rows and columns of the overlapping search extent. This means if always using the same size grid for different size points clouds, the search speed will always be the same yielding a Big-O speed of just O(1)[9]. Searching a quadtree requires starting from the top node and traversing down the entire tree to find an element. This yields a Big-O speed operating at O(log n)[10] (in this case, since we're making a decision between 4 choices at each node, this would be log base 4). Comparing these speeds, it looks as though the iterative approach would perform the worse (as expected) whereas the grid approach would perform the best.

2.2.2 Quantitative Comparison

The above was just a theoretical comparison. However, a real world comparison will involve implementing and timing each method. The implementation for each is defined as follows.

Iterative search method

To search using the iterative approach, each point must be analyzed to determine whether or not it is within a search extent. The following function will determine if a point falls within an extent:

```
vector<long> iterative_search( LASreader r, extent e ) {

    vector<long> indices = new vector<long>;

    for ( int i = 0; i < r.h.number_of_point_records; i++ ) {
        if ( is_in_extent (r.read(i)), e )
            indices.push_back(i)
    }

    return indices;
}
```

The for loop uses the LASreader to iterate each point in the file. The function `is_in_extent` determines if a single point falls within a given extent, if true that point reference is added to the list.

[9] Garlasu, Dan et al. "A big data implementation based on Grid computing." *Roedunet International Conference (RoEduNet), 2013 11th* 17 Jan. 2013: 1-4.

[10] Tayeb, Jamel, Özgür Ulusoy, and Ouri Wolfson. "A quadtree-based dynamic attribute indexing method." *The Computer Journal* 41.3 (1998): 185-200.

Grid search method

The grid search method necessitates the identification of cells that overlap with the search extent then searching the list of references within each overlapping cell. This is done by converting the four points that make up the search extent into grid indices, then visiting those cells.

```
vector<long> grid_search ( LASreader r, grid_index g, extent e ) {

    vector<long> indices = new vector<long>;

    // transform positions into grid cell indices
    int x_min = round((e.x.min - grid_index.x) / grid_index.res);
    int x_max = round((e.x.max - grid_index.x) / grid_index.res);
    int y_min = round((e.y.min - grid_index.y) / grid_index.res);
    int y_max = round((e.y.max - grid_index.y) / grid_index.res);

    // iterate over overlapping grid cells
    for (int i = x_min; i < x_max; i++) {
        for (int j = y_min; j < y_max; j++) {

            // iterate through elements in cell
            for (iterator it = g.grid[i][j].begin();
                          it != g.grid[i][j].end(); ++it ){
                if ( is_in_extent (r.read( it ), e) )
                    indices.push_back(it);
            }

        }
    }

    return indices;
}
```

First, the points that make up the search extent are converted into grid cell positions. Using a nested for loop, each overlapping cell is iterated and the point preferences that fall with each cell are iterated again using the is_in_extent function to verify the point is in the extent.

Quadtree search method

The quadtree search method involves determining quadrants that fall within a search extent. Starting with the top node, the extents of the 4 quadrants are tested to determine if it overlaps with the search extent. Each overlapping extent is recursively traversed until the terminal nodes are reached. The index or list of indices at these nodes are then tested to determine if it falls within the search extent. (Note that the search is the same for both bin and depth termination)

```
// quadtree search
```

```
vector<long> search ( LASreader r, quadtree q, extent e ) {

    vector<long> indices = new vector<long>;

    // base case
    if (is_quad_terminal ( q )){

        for (iterator it = q.indices.begin();
            it != g.grid[i][j].end(); ++it ){
                if ( is_in_extent (r.read( it )) )
                    indices.push_back(it);
        }
    }
    else { // recursion
         vector<long> v; // temp vector to store returns
         // check for overlapping of each quadrant's extent
         // q1
         if( do_extents_overlap( e, q.q1.e ) ) {
             v = search( r, q.q1, e );
             for (iterator it = v.begin(); it != v.end(); +it)
                 indices.push_back(it); // push returned indices
         }
         // q2
         if( do_extents_overlap( e, q.q2.e ) ) {
             v = search( r, q.q2, e );
             for (iterator it = v.begin(); it != v.end(); +it)
                 indices.push_back(it);
         }
         // q3
         if( do_extents_overlap( e, q.q3.e ) ) {
             v = search( r, q.q3, e );
             for (iterator it = v.begin(); it != v.end(); +it)
                 indices.push_back(it);
         }
         // q4
         if( do_extents_overlap( e, q.q4.e ) ) {
             v = search( r, q.q4, e );
             for (iterator it = v.begin(); it != v.end(); +it)
                 indices.push_back(it);
         }
    }

    return indices;
}
```

The above function will first determine a base case by testing if the current node being tested is a terminal node by calling `is_quad_terminal`. Otherwise, each sub-quadrant contained within the current quadrant is tested whether its extent overlaps the search extent using `do_extents_overlap`. If true, the sub-quadrant is recursively searched then the point references that are returned are pushed into `indices`. The same search function applies to both bin and depth terminated quadtrees.

The Comparison

Given a search extent, the execution time for each method was recorded. This was performed on 11 separate datasets all having a square extent of 1000 meters and containing different number of points defined in table 4.

Table 4: Datasets

Dataset	Number of Points
1	1000
2	2000
3	4000
4	8000
5	16000
6	32000
7	64000
8	128000
9	256000
10	512000
11	1024000

A threshold of 1 was set on the bin terminated quadtree to resemble the most classic form of a quadtree. A threshold of ten depths were used for the depth terminated quadtree. For calibration, the extent size of the 10th depth was used for the resolution of the grid index. This resolution is determined by the following equation:

$resolution = \frac{1}{2^n} \cdot extent$, smallest resolution for depth termination

The *n* in the previous equation is the maximum depth allowed for depth termination and extent here is the extent of a particular direction. This equation comes from the fact that for every depth, the entire extent is recursively split in half in a given direction. Since ten levels are being used, this resolution would be:

$\frac{1}{2^{10}} \cdot 1000m = \frac{125}{128}m$, smallest possible resolution with maximum 10 levels for depth termination

So, $\frac{125}{128}m$ was the resolution used for the grid indexing.

Speed

The processor in the computer used was an AMD Phenom(tm) II X6 1100T at 3.31 GHz. A virtual machine (using Oracle VM[11]) was created running Ubuntu 12.04 64 bit set to use 1 central processing unit (CPU) and 512 megabyte base memory (RAM). The virtual machine operates at a much slower speed than the computer it resides and better illustrates the speed without having to record very small values or increasing the number of points to very large amounts. The speed results and size of each method are outlined in table 5.

Table 5: Speed Comparison

Method / # of Points	Iterative (non-indexed)	Grid Indexing	Quadtree	
			Depth Termination	Bin Termination
Dataset 1: 1000	2.810	0.089	0.134	0.259
Dataset 2: 2000	5.264	0.114	0.236	0.481
Dataset 3: 4000	10.327	0.164	0.305	1.017
Dataset 4: 8000	21.245	0.264	0.424	1.610
Dataset 5: 16000	45.543	0.464	0.644	2.258
Dataset 6: 32000	92.436	0.864	1.063	2.962
Dataset 7: 64000	179.945	1.664	1.882	3.721
Dataset 8: 128000	365.734	3.264	3.501	4.537
Dataset 9: 256000	750.976	6.464	6.721	5.022
Dataset 10: 512000	1534.987	12.864	13.140	5.933
Dataset 11: 1024000	2956.159	25.664	25.959	6.795

[11] "VM - Virtual Machine | Oracle." 2012. 5 Jun. 2015 <http://www.oracle.com/us/technologies/virtualization/oraclevm/overview/>

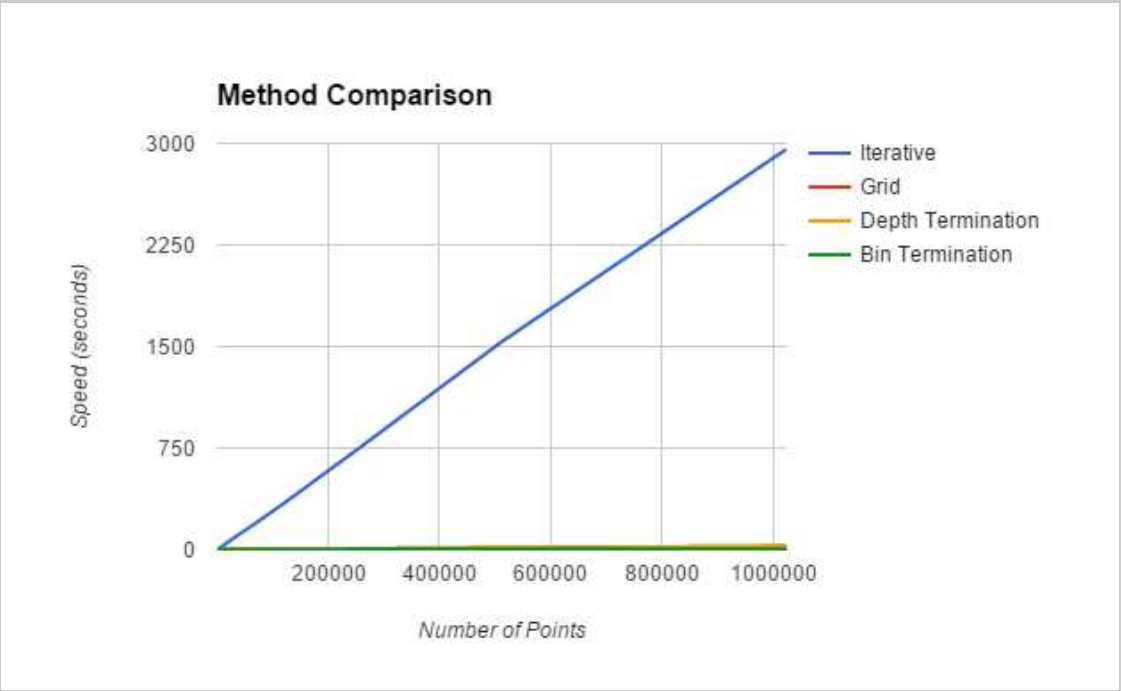


Figure 9: Methods Speed Comparison: Methods Speed Comparison

It is obvious the iterative approach is much slower than the index methods (although, following the trend backwards, as the number of points approach 0, an iterative search could as fast. However, since lidar projects usually contain a large number of points, this becomes unimportant). Omitting the iterative results and comparing the index methods yields the following graphs:

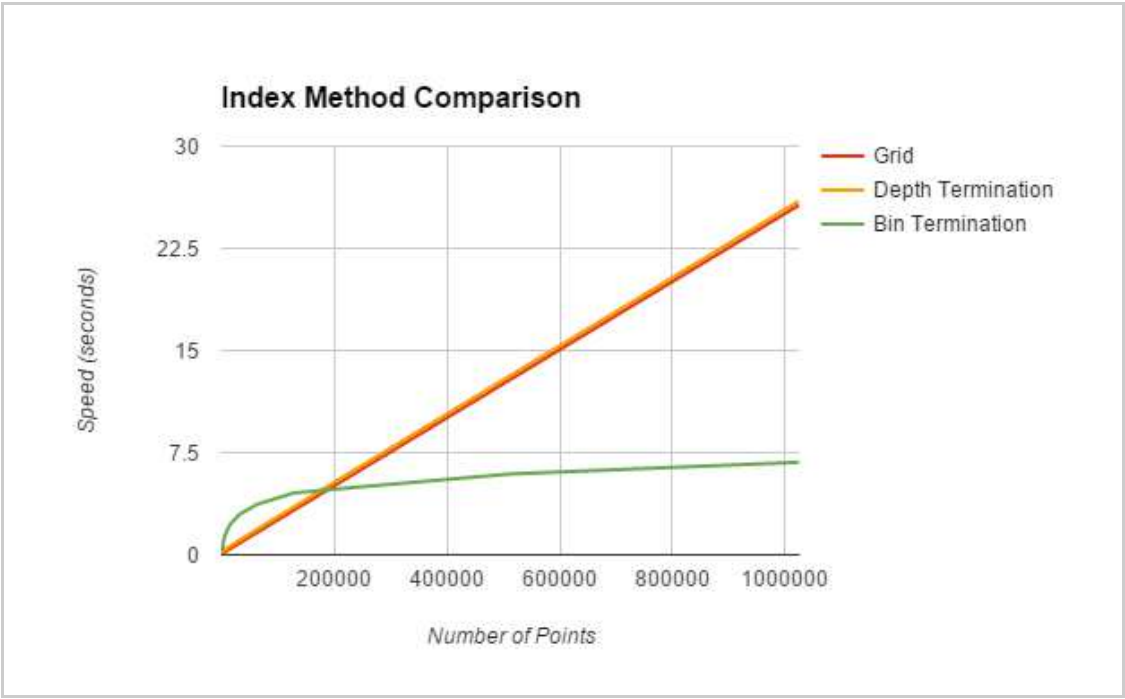


Figure 10: Index Speed Comparison (Iterative Method Omitted)

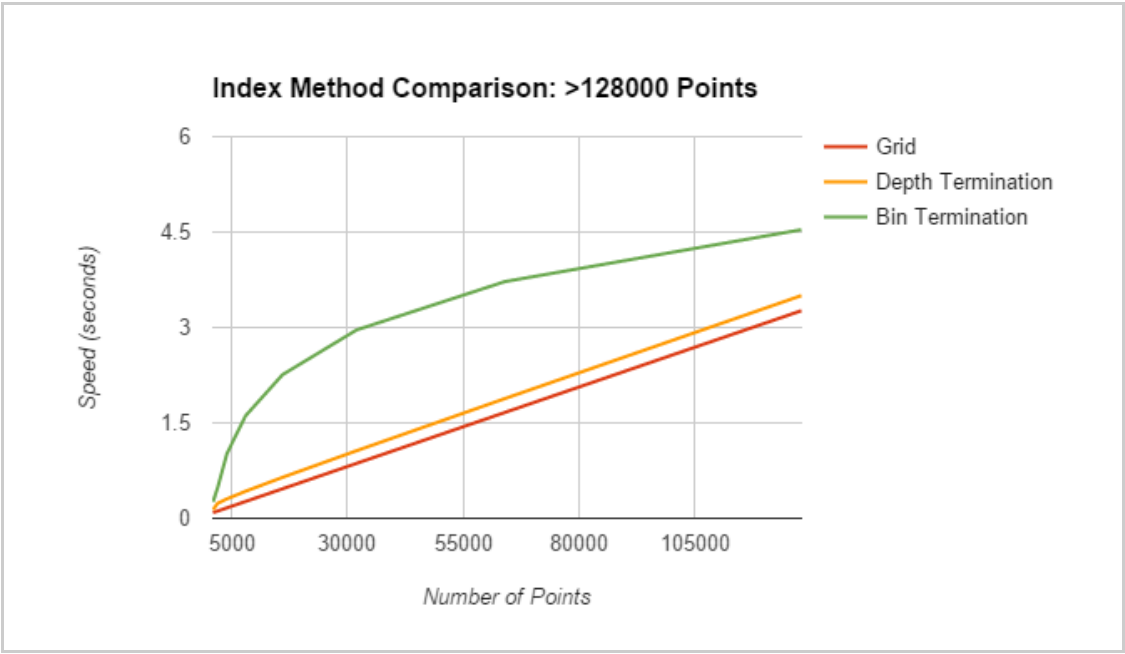


Figure 11: Index Speed Comparison - datasets under 150,000 point quantity

Comparing just the index methods, initially the bin terminated quadtree is the slowest but quickly becomes the fastest as the number of points increase across the datasets. This is due to the limitation put on grid index resolution and the threshold of the depth

terminated quadtree; as the number of points increase, more point references are put into each cell or terminal quadrant and have to be iterated. This is why a linear pattern emerges similar to the iterative (non-indexed) method. This emergence can be delayed by increasing grid resolution or the depth threshold (for grid and depth terminated quadtree, respectively) but has the effect of increasing the space required for the index. Figure 8 exhibits what happens right before bin termination becomes faster than the other methods; it's seen at lower values of points that both quadtrees have the similar curves until around 5000 points when the depth terminated quadtree seems to follows the same linear pattern as the grid index. It is likely that this is when all points are being stored at the maximum depth of the quadtree and would have the same gridded structure as the grid (this was inevitable as the resolution of the grid was defined at an equal size to the extent covered by a quadrant at the tenth depth). Bin termination is far less dependent on an iterative search at a terminal node (with no iterative searching being conducted at a bin size of 1), so the speeds are dependent solely on traversing the quadtree.

Size

Next, the size is analyzed for each method. These are outlined in table 6 (expressed in bytes).

Table 6: Size Comparison in Bytes (Index Size Only)

Method / # of Points	Iterative (non-indexed)	Grid Indexing	Quadtree	
			Depth Termination	Bin Termination
Dataset 1: 1000	41243	31936032	31768	32768
Dataset 2: 2000	82243	31936032	73543	75281
Dataset 3: 4000	164243	31936032	168045	172950
Dataset 4: 8000	328243	31936032	389095	397336
Dataset 5: 16000	656243	31936032	896435	912838
Dataset 6: 32000	1312243	31936032	2065152	2097152
Dataset 7: 64000	2624243	31936032	4753990	4817990
Dataset 8: 128000	5248243	31936032	10940834	11068834
Dataset 9: 256000	10496243	31936032	25173504	25429504
Dataset 10: 512000	20992243	31936032	33554432	58421659
Dataset 11: 1024000	41984243	31936032	33554432	134217728

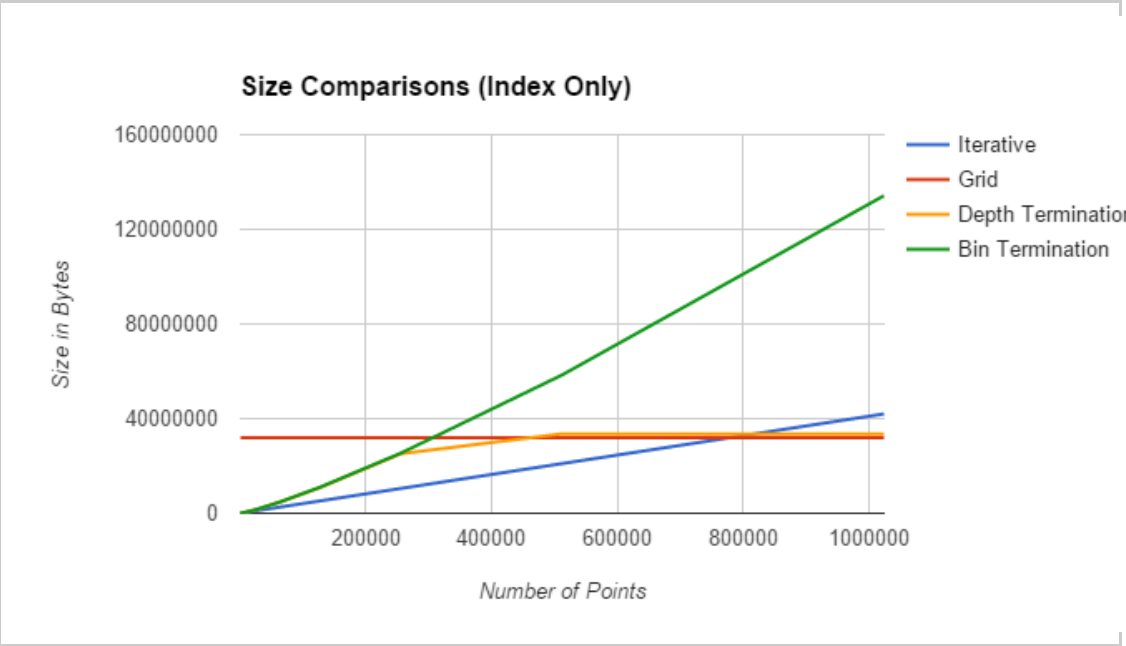


Figure 12: Index Size Comparisons

Table 6 and Figure 12 shows the size of just the index for grid and the depth terminated quadtree, omitting the actual point references. As these references are necessary for this type of indexing, Table 7 shows the size when taking them into account.

Table 7: Size Comparison in Bytes

Method / # of Points	Iterative (non-indexed)	Grid Indexing	Quadtree	
			Depth Termination	Bin Termination
Dataset 1: 1000	41243	31977275	32768	32768
Dataset 2: 2000	82243	32018275	75543	75281
Dataset 3: 4000	164243	32100275	172045	172950
Dataset 4: 8000	328243	32264275	397095	397336
Dataset 5: 16000	656243	32592275	912435	912838
Dataset 6: 32000	1312243	33248275	2097152	2097152
Dataset 7: 64000	2624243	34560275	4817990	4817990
Dataset 8: 128000	5248243	37184275	11068834	11068834
Dataset 9: 256000	10496243	42432275	25429504	25429504
Dataset 10: 512000	20992243	52928275	54546675	58421659
Dataset 11: 1024000	41984243	73920275	75538675	134217728

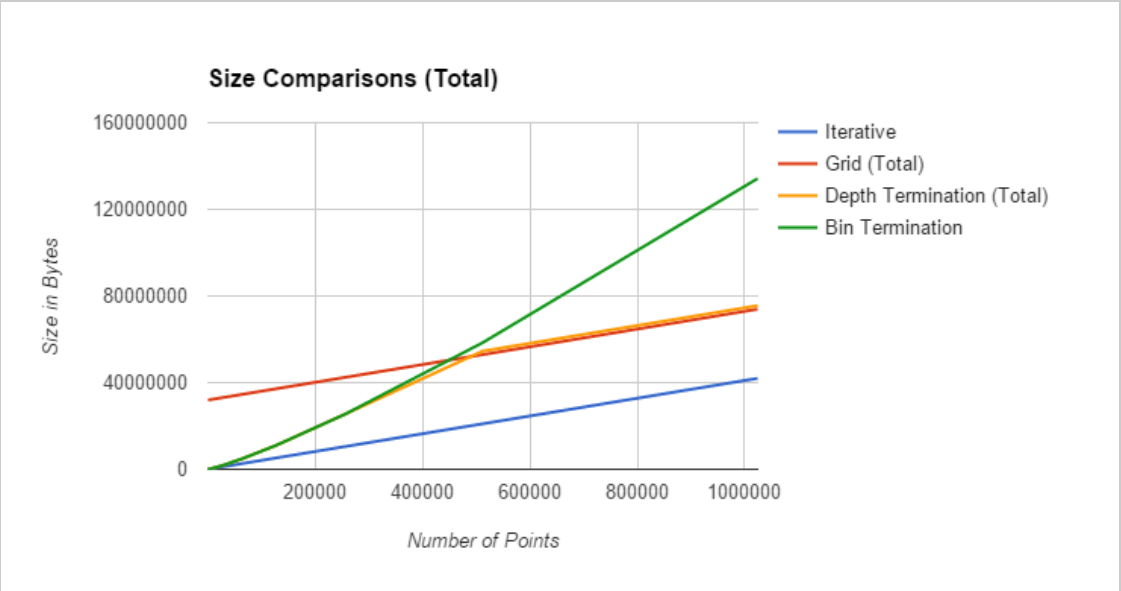


Table 8: Total Size Comparisons

For the iterative (non-indexed) method, the size it is just the size of the points (ex: the original LAS file) and will always be smaller than the indexing methods. The size of the grid index is dependent on the number of points referenced plus the allocated grid. The size of the quadtrees are the number of grid point references plus each node contained in a tree. Again, as discussed earlier, the depth terminated quadtree will eventually resemble a grid index (at a concurring resolution)...

In conclusion, a grid search could always perform the fastest if a resolution is selected that keeps cell density low enough. However, really fine resolutions mean allocating a large amount of space. This is where a quadtree is excels (namely, the bin terminated quadtree). To obtain the benefits of both approaches, a depth terminated quadtree should be used, and because of this, the rest of this paper will focus on this method.

2.3 Additional Index Methods

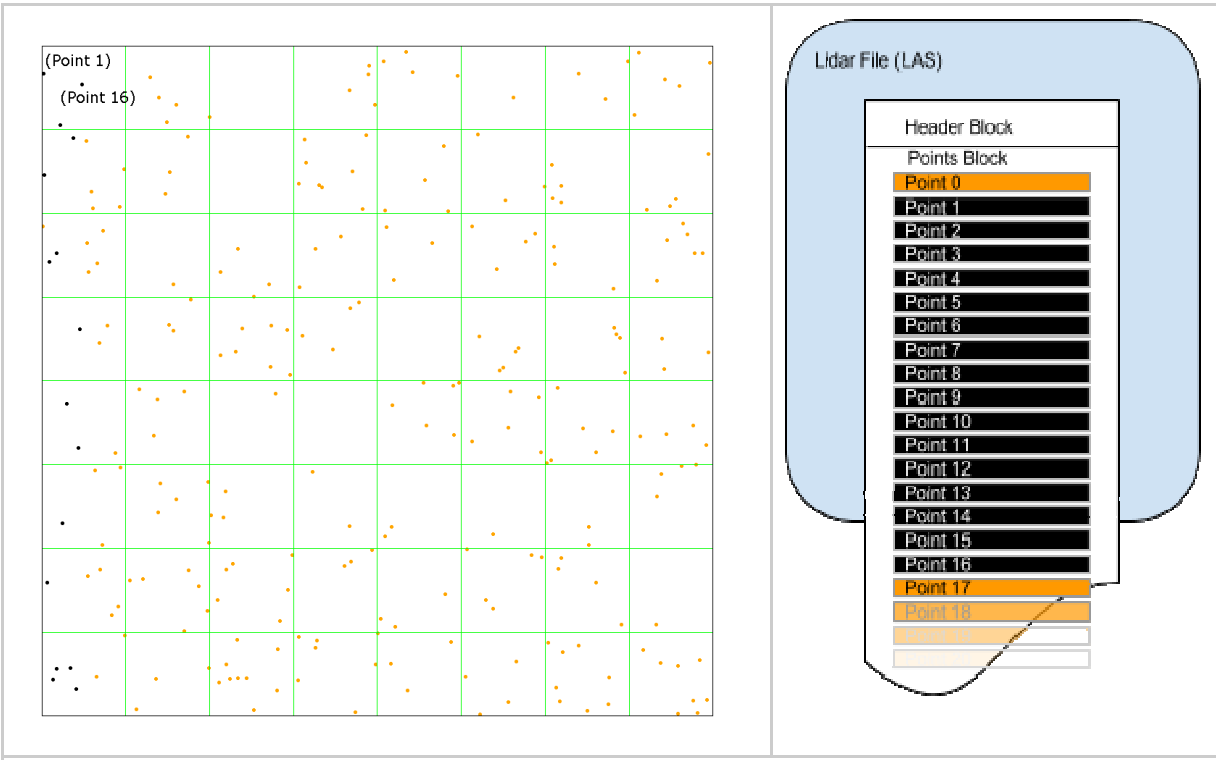
More indexing methods exist that should be tested. One such method is called an R-tree[12] that groups spatially similar objects together at different scales where each depth of the tree are the scales. Studies have shown R-trees to perform better than quadtrees, and the parameters for its creation are less complex[13], however, it should be noted that

[12] Beckmann, Norbert et al. *The R*-tree: an efficient and robust access method for points and rectangles*. ACM, 1990.
[13] Kothuri, Ravi Kanth V, Siva Ravada, and Daniel Abugov. "Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data." *Proceedings of the 2002 ACM SIGMOD international conference on Management of data* 3 Jun. 2002: 546-557.

R-trees are usually larger in size and require a cluster analysis (and therefore more complex to implement) for determining groups.

2.4 A solution to the space issue for index methods

A concern when using an indexing method is the size. Every point is indexed so the resultant index file is roughly as large as the original lidar file plus the size of the actual index (see size comparison in previous section); the original data and a corresponding index would likely consume twice the storage. One solution is to store the actual point data into the index and the remove the original to save space, however, it is often desirable to keep the original lidar file. A solution proposed here is to keep just the first and last reference points within a cell or terminal node and iterate all points in between from the original LAS file (this of course will not work on a bin terminated quadtree of size 1, however iteration of this method was never a real concern as it was with the others). This has the effect of likely having to iterate through points that fall within other cells or nodes. Figure 13 illustrates this.



When extracting points from the top left cell, points from other cells get iterated. The image on the right shows how the points are ordered, and highlights the points that have to be iterated.

Figure 13: Extracting Points in a Cell

In the previous figure, the LAS file was sorted by ascending x values. From the figure, the first cell of the grid uses point 1 as the beginning reference and point 16 as the ending reference. So, when extracting points from this cell, all points between 1 and 16 have to be extemporaneously iterated and determined to fall within the this cell.

Since we are only referencing 2 points per cell/terminal quadrant, the size can drastically be reduced (ex: if there are 20 points contained within a terminal quadrant and this method is applied, then the index size is reduced to 10 percent).

The above was an example of what would happen when using a grid index, but applying this to a quadtree would yield a similar outcome. Creating the quadtree uses the same insert structure as before, but slightly modified. This is seen in the following source code:

```
void insert( LASreader r, quad q, int depth ){

    // Reference points that fall in extent e
    vector <long> * indices = new vector <long>;

    for ( int i = 0; i < r.h.number_of_point_records; i++ ){

        if ( is_in_extent( r.read( i ), q.e ) )
            indices.push_back (i);

    }

    //check if extracted points will fit into the bin
    if ( q.depth == depth || indices.size() < 2 ) { // base case

        // Keep just first and last references
        g.indices.push_back( indices.begin() );
        q.indices.push_back( indices.end() );

    } else {

        // create new quadrant extents
        array < extent, 4 > e_arr = split_quad_extent ( e );

        //q1
        q.q1.extent = e_arr[0];
        q.q1.depth = q.depth+1;
        insert( r, q.q1, bin_size );

        //q2
        q.q2.extent = e_arr[1];
        q.q2.depth = q.depth+1;
        insert( r, q.q1, bin_size );

        //q3
        q.q3.extent = e_arr[2];
```

```
        q.q3.depth = q.depth+1;
        insert( r, q.q1, bin_size );

        //q4
        q.q4.extent = e_arr[3];
        q.q4.depth = q.depth+1;
        insert( r, q.q1, bin_size );

    }

}

quadtree create_depth_quadtree ( file lasfile, int depth ) {

    LASreader r = LAS_read( lasfile );       // initialize the reader
    quadtree qt = new quadtree();            // initialize the
quadtree

    // Kick off the recursion
    insert( r, qt.q, depth );              // Start insertion
}
```

This time, if base case is true, only the first and last references are stored.

2.4.1 Gaining speed through sorting

To eliminate the need to search through other cells, the LAS file can be sorted in such a way that all the points that fall within a cell/terminated quadrant are organized so they are consecutively stored. This has the effect of gaining speeds equal to a normal indexing method, however it requires the original LAS file to be altered.

2.5 Selecting Index Parameters for Index Creation

The speed of each method depend on so many factors (mainly the computer they reside) that making decisions for the parameters for initializing them can be difficult. For a uniformly distributed point cloud, grid indexing is an obvious choice with a resolution that encompasses a cell density of one. However, it is unlikely that lidar would be distributed this way and space is always a concern, especially for larger datasets.

Since space can be the real limitation (we never want to use parameters that directly and intentionally decrease the index speed), it is best to select parameters based on this. Generally (from experience), a value no larger than 10% of the original LAS file is typically acceptable. For the method that reduced the index size, it was mentioned that keeping just two points from 20 reduced the size of a cell/quadrant to 10%. By selecting a grid resolution that will yield a cell density of no more than 20 points, the size of the references in the index will never exceed 10% and the overall index will likely be smaller than 10% (in a worst case scenario, if there are 20 points distributed in each cell, the overall index would actually be the 10% plus the overhead of the index). This resolution is used to determine the maximum depth level for a depth terminated quadtree.

Test datasets using depth terminated quadtrees and the size reduction method have yielded an index file no larger than 10% (on average, they were only 1% of the original LAS file) and a speed that was about 10 times faster as compared to iteratively loading points.

3. File Space and Memory Space

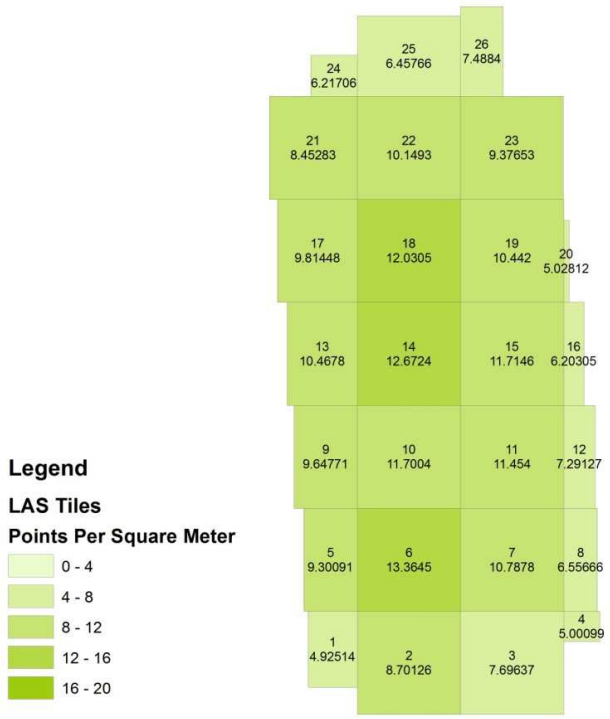
For budgeting memory allocation, processing lidar is done in two forms - file space and memory space. File space is the use of a physical hard drive to permanently store the lidar data. The lidar is stored as LAS files and loaded into memory space for processing since it is done there more efficiently. Loading entire LAS projects into memory can be difficult (or even impossible) due to memory limitations, so a combination of the two forms must be used for effectively processing the data. An ideal situation is to create a fully transparent procedure for a common user that allows them to load a portion of the lidar to memory in an efficient manner.

3.1 File Space

To maintain standards, lidar should always be stored as an LAS file. As demonstrated by the indexing method comparison, simply iterating through these files is the most inefficient approach. Software should take preliminary steps to make searching and loading relevant areas from lidar both faster and easier. Firstly, as demonstrated earlier, every file should have a quadtree index file to accompany it that will make searching within a single file more efficient. Secondly, since lidar files are usually tiled into smaller files according to specific extents, they should also be indexed so it can be quickly identified as to which overlap with a loading extent. However, lidar files can become mixed and can have overlapping or erratic extents which normal indexing methods do not account for (most index methods use a pattern to speed up searches). Instead, because there are usually far less files representing a project area then the points contained inside a typical LAS file, a simplified indexing method can be utilized using shapefiles.

A Shapefile is a standard Geographical Information System (GIS) format that stores shapes (or points) as vectors comprised of points and lines[14]. Multiple shapes can be stored into a single shapefile making it perfect to store individual LAS file extents. The open software called Geospatial Data Abstraction Library[15] (GDAL) contains the OpenGIS Simple Features Reference Implementation (OGR) that can be used to create, load and search across shapefiles using an extent search. Additionally, this software can be built into other software using the Application Programming Interface (API) provided.

[14] "ESRI Shapefile Technical Description." 2014. 8 Jun. 2015 <https://www.esri.com/library/whitepapers/pdfs/shapefile.pdf>
[15] "GDAL: GDAL - Geospatial Data Abstraction Library." 2004. 3 Jun. 2015 <http://www.gdal.org/>



Each tile labeled using the name of the LAS file. Additionally, each tile is labeled and colored by its average point density.

Figure 14: Shape File Representation

After the correct files are identified using a shape file search, the accompanying quadtree can be used to load the points that fall within the loading extent. The following figure outlines the process followed for loading corresponding points from a search extent, and at which step the files in a lidar directory are used.

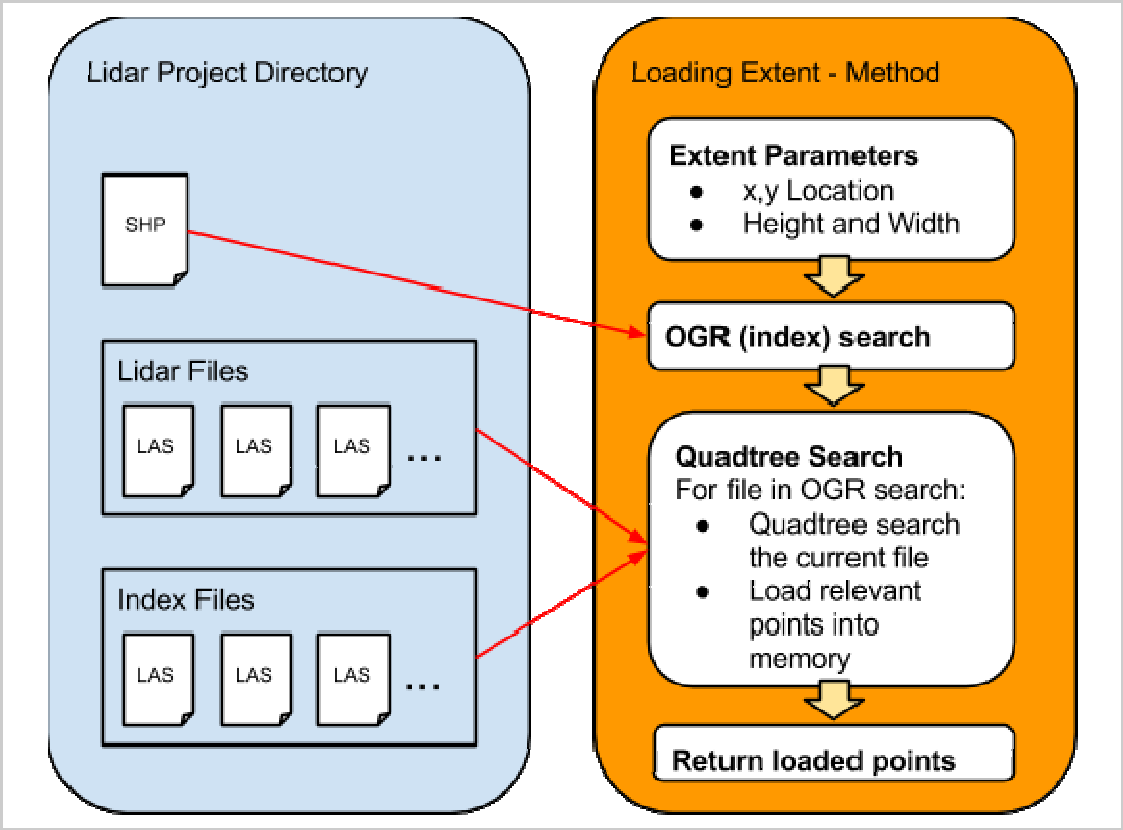


Figure 15: Loading data to Memory Space

3.2 Memory Space

When loading points into memory, a structure specific to lidar is used to hold point information. A single point structure should store the x,y,z location and additional attributes (as explained by the LAS point format) is used. For this, the Point Cloud Library with a tailored point type was created.

3.2.1 Point Cloud Library

The point cloud library (PCL) is a c++ package that attempts to standardize point structures and point clouds. It uses established structures and methods that are designed for optimizing memory operations while handling point clouds[16]. Additionally, updates to PCL are easily adopted into this software.

For efficiency, the individual point structure is static (attributes cannot be removed or added after a compilation) and for lidar (specifically, version 3 point format), this structure looks like the following (a custom PCL format created for this software):

[16] Rusu, Radu Bogdan, and Steve Cousins. "3d is here: Point cloud library (pcl)." *Robotics and Automation (ICRA), 2011 IEEE International Conference on* 9 May. 2011: 1-4.

```
struct Point3 {

	//LAS Point format - version 3
	unsigned short intensity;
	unsigned char return_number;
	unsigned char number_of_returns;
	unsigned char scan_direction_flag;
	unsigned char edge_of_flight_line;
	unsigned char classification;
	unsigned char scan_angle_rank;
	unsigned char user_data;
	unsigned short point_source_ID;
	double GPS_time;
	unsigned short red;
	unsigned short green;
	unsigned short blue;

};
```

For constructing the point cloud, a lidar structure is used to hold header information and the list of individual points. The header holds information about the points contained in a LAS file as well as unique information pertaining to it collection. This information should be maintained for other post processing tasks including writing a point cloud back to a file. The header structure is as follows:

```
struct Header{

	char file_signature[4];
	unsigned short file_source_ID;
	unsigned short global_encoding;
	unsigned short project_ID_GUID_data1;
	unsigned short project_ID_GUID_data2;
	unsigned short project_ID_GUID_data3;
	unsigned char project_ID_GUID_data4[8];
	unsigned char version_major;
	unsigned char version_minor;
	char system_identifier[32];
	char generating_software[32];
	unsigned short file_creation_day_of_year;
	unsigned short file_creation_year;
	unsigned short header_size;
	unsigned long offset_to_point_data;
	unsigned long number_of_variable_length_records;
	unsigned char point_data_format_ID;
	unsigned short point_data_record_length;
	unsigned long number_of_point_records;
	unsigned long number_of_points_by_return[5];
	double x_scale_factor;
	double y_scale_factor;
	double z_scale_factor;
	double x_offset;
```

```
      double y_offset;
      double z_offset;
      double max_x;
      double max_y;
      double max_z;
      double min_x;
      double min_y;
      double min_z;

};
```

As for the Lidar structure, which ties everything together (and uniquely identifies the point cloud as lidar) holds the header, the point cloud and relevant methods to modifying/accessing the points. This structure looks like the following:

```
struct Lidar {

      Point3 *pc;

      Header;
      int update_header( );

      double getX(int index);
      double getY(int index);
      double getZ(int index);
      void setX(int index, double x);
      void setY(int index, double y);
      void setZ(int index, double z);
};
```

3.2.2 LAS Importing

The simplest method for importing an LAS file is to read the entire file into memory by iterating through each point (this is seen in most lidar software).

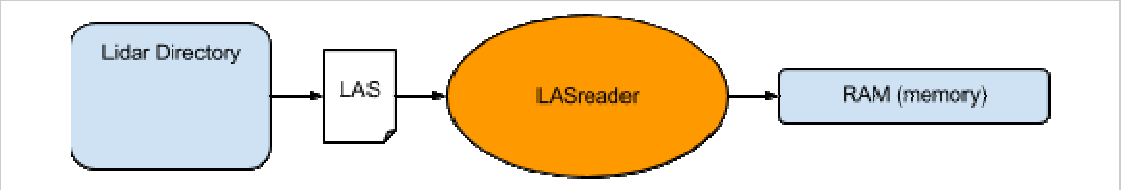


Figure 16: Loading an Entire LAS file

However, it is probably more useful to load a portion of the data defined by a search/loading extent. This is evident when processing a highly dense lidar dataset to a wall-to-wall grid product; the dataset maybe too large to store in memory, so it would be beneficial to iterate over the cells of the grid and extract points from LAS files as

necessary. To do this, the LASreader function can be overloaded to include an option to load a given extent using the search methods outlined earlier.

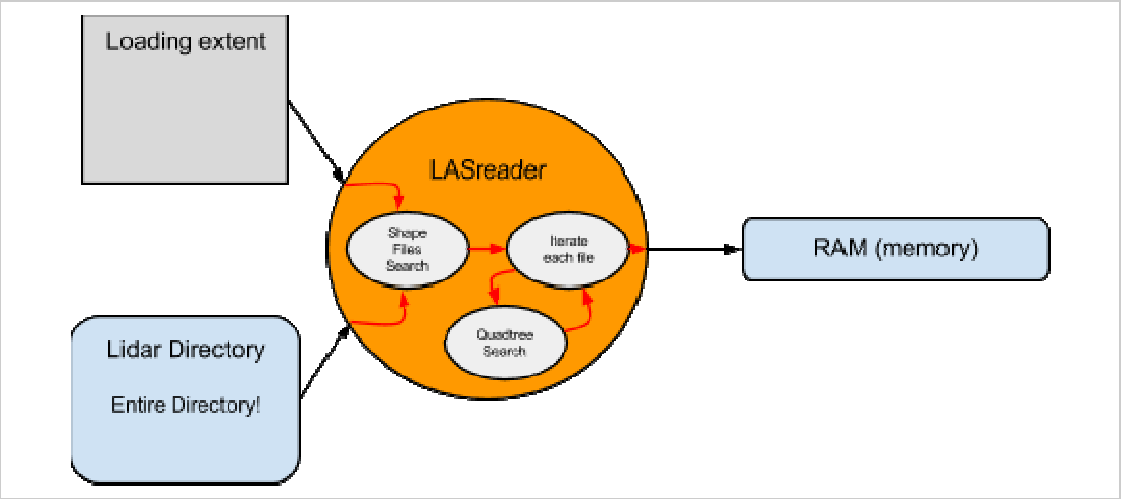


Figure 17: Extracting LAS Points using a Loading Extent

3.2.3 Octrees

Octrees are similar to quadtrees except they search in 3 dimensions (x, y and z). While not as useful for loading points from an LAS file (because a 2-dimensional spatial search is usually sufficient in file space and the size is *n* times larger depending on the resolution and range of the z direction), an octree can be useful for the points loaded into memory. This usefulness can be observed when analyzing neighboring points to a single points for something like cluster analysis where octrees are used to efficiently identify these points. PCL has octree structures and corresponding methods as part of its API. Octrees are created when needed and their setup is demonstrated in the code snippet below:

```
//octree setup
float resolution = 1.0f;  // smallest resolution of octree
pcl::octree::OctreePointCloudSearch<Point3> octree (resolution);  //
create
octree.setInputCloud (pc);  // set point cloud from Lidar structure
octree.addPointsFromInputCloud (); // Add points
```

PCL can create these octrees very efficiently, so speed in initializing one (for a smaller AOI dataset) is usually not a concern (it could be an issue if creating an octree for a very large point cloud, but this software avoids this situation).

4. Methods

The methods for processing lidar are separated into 2 sections: Preliminary methods and processing methods. The former is used to prepare the lidar for actual processing (the latter).

4.1 Thinning

Thinning the lidar can serve a few purposes. The most obvious benefit is that it reduces the amount of space required to store the files (or the amount to load into memory). Lidar is usually collected at a higher density than what is needed for the final lidar products; many of the points are redundant at certain resolutions and can be thinned to expedite the processing. Another benefit that is often observed after thinning is the reduction of noise.

4.1.1 Simple Thinning

Given a step of *n*, every *n*th point is removed from the lidar list. This is done by iterating each point from the point cloud and removing every *n*th point (ex: a step size of 2 would remove every other point thus reducing to about half the size).

4.1.2 Random Thinning

Given a number of points to remove *m*, *m* indices are chosen at random and the corresponding point is removed. This can also be given as a percentage, where *m* is determined by total points multiplied by the percentage.

4.1.3 Pixel Thinning

Given a resolution *r*, an overlapping grid is created with cell size of resolution *r* is used to select one point within each grid cell. There are several methods for choosing a single point to represent a grid cell.

- Bool - Determine if any points exist inside a grid cell. If true, the center of the pixel is used to determine the x,y location and all other attributes (including z) are averaged from all points in cell or by using the closest point to the center.
- Average - all points that fall within a grid cell are averaged to a single point.
- Random - A single point inside the grid cell is randomly selected to represent the grid cell.

4.1.4 Voxel Thinning

Voxels are similar to pixels and use the same methods for selecting a single points to represent the cell). However, this thinning method uses all 3 dimensions (x,y,z and is sometimes referred to as a 3D pixel). The result will be larger than the result from pixel thinning (at the same resolutions) but will preserve multiple z values at a corresponding 2D pixel. Additionally, this thinning method can be used to eliminate very small noisy points because it is more likely a thinned point will represent an object more accurately by using the techniques described in pixel thinning. This is because (in most cases) there are a larger number of the points that represent an object in a voxel accurately then there are noisy points.

4.1.5 Noise reduction

For highly dense but noisy lidar datasets, it can be assumed that most points are mostly correct and those points are likely to be closely grouped together. The noisy points are often identified as isolated points away from closely grouped points. Using a distance threshold that allows a point to exist if it has a given number of neighbors (neighbor threshold) within the distance threshold, the noisy points can be eliminated.

4.2 Processing Methods

4.2.1 Grid (Raster) Class

The Geospatial Data Abstraction Library (GDAL, mentioned earlier) is used to store raster information (a raster is a grid, however, here it is used to refer to grids being used as a product from lidar). The GDAL API provides standardized methods for creating, storing and saving raster data. Additionally, it also provides methods for interpolating points to a raster. Dependent grid definitions using GDAL used in this software can be viewed in the appendix. Source code for creating a grid is outlined below (where each function is contained with a structure “Grid”):

```
int Grid::create(double X, double Y, double cellsizeX, double
cellsizeY, int nX, int nY){

      //meta
      offsetX = X;
      pixelResX = cellsizeX;
      offsetY = Y;
      pixelResY = cellsizeY;
      nXSize = nX;
      nYSize = nY;

      //setup buffer
      buffer = (float*) CPLMalloc(sizeof(float)*(nXSize*nYSize));

      return 0;

}
```

Given the x,y orgin `X`, `Y`, the resolution, `cellsizeX`, `cellsizeY` and the number of columns and rows, `nX` and `nY` a raster can be properly defined. For setting and retrieving values, the following methods were used:

```
float Grid::get_value( float x, float y){

      int xOffset = ((x-offsetX)/pixelResX); //Values will get
truncated
      int yOffset = ((y-offsetY)/pixelResY); //Values will get
truncated
```

```
      return buffer[(nXSize*yOffset)+xOffset];

}

int Grid::set_value( float x, float y, float z ){

      int xOffset = ((x-offsetX)/pixelResX); //Values will get
truncated
      int yOffset = ((y-offsetY)/pixelResY); //Values will get
truncated
      int pos = (nXSize*yOffset)+xOffset;
      buffer[pos] = z;

      return pos; //in case this is useful to the user

}
```

For retrieving values, only an x and y are passed in the function, and the nearest cell's value will be returned (here as a float, but this can be any defined variable type). Similarly, an x and y are passed in but with an additional z parameter to set a cell's value.

4.2.2 Ground Point Classification

MCC-Lidar

The multiscale curvature classification algorithm (MCC) attempts to classify ground points with only a few model parameters and minimal processing[17]. The algorithm is optimized for a forest environment, however works in other types of terrains as well (ex: a meadow with downed logs or a post-file area with snags). MCC takes in 2 parameters: a scale parameter and a curvature tolerance. First, a raster is created using a resolution equal to the scale parameter and points are interpolated using the thin plate spline (TPS) method from GDAL. Next, the raster cells are iterated and a 3X3 average filter plus the curvature tolerance is used to create a new curvature raster. Finally, each point is iterated where each z value is compared to the curvature raster; if z is found to be greater than the corresponding value in the curvature raster, it is marked as a non-ground point, otherwise it is marked as a ground point.

```
void mcclidar( float s, float c, Lidar l) { //scale (s) and curvature
(c)

    Grid = new Grid<float>(l.header.min_x, l.header.min_y, s,
         s, (l.header.max_x-l.header.min_x) / s,
         (l.header.max_y-l.header.min_y) / s );
```

[17] Evans, Jeffrey S, and Andrew T Hudak. "A multiscale curvature algorithm for classifying discrete return lidar in forested environments." *Geoscience and Remote Sensing, IEEE Transactions on* 45.4 (2007): 1029-1038.

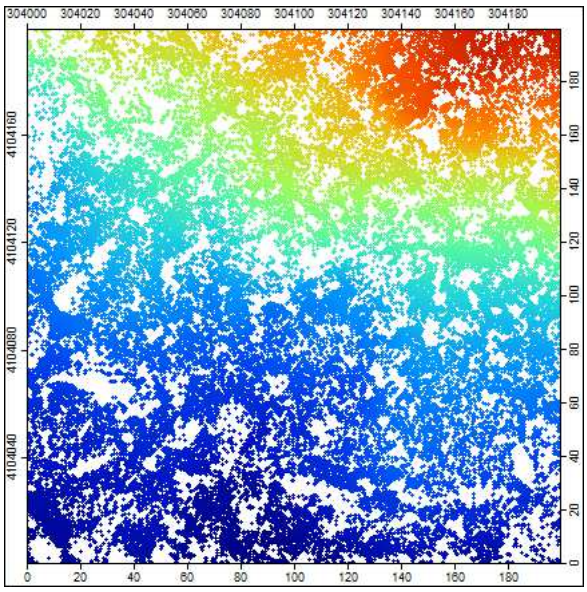
```
    // setup TPS, requires grid and number of points
    VizGeorefSpline2D vs = new VizGeorefSpline2D( grid,
        l->pc.size() );
    vs.set_toler(s,s);
    // add points to TPS
    for ( iterator it = l->pc.begin(); it == l->pc.end(); it++)
        vs.add_points( PointXYZ( it ) ); // cast to simple XYZ

    vs.solve();

    // apply 3x3 avg filter
    Grid curv = new Grid<float>(l.header.min_x, l.header.min_y, s,
         s, (l.header.max_x-l.header.min_x) / s,
         (l.header.max_y-l.header.min_y) / s )

    for ( i = 1; i < curv.nx - 1; i++ ){
        for ( j = 1; j < curv.ny - 1; j++ ){

            int tot = 0
            for ( k = -1; k <= 1; k++ ){
                for (l = -1; l <= 1; l++ ){
                    tot = tot + grid[i+k][j+l];
                }
            }

            curv[i][j] = (tot/9) + t; // add tension to avg value
        }
    }

    // run classification on points
    for ( iterator it = l->pc.begin(); it == l->pc.end(); it++){
        if ( it.z > curv.get_value( it.x, it.y ))
            it.classification = 0; // unknown classification
        else:
            it.classification = 2; // ground classification
    }
}
```

4.2.3 Digital Terrain Models

A digital terrain model (DTM, sometimes referred to as digital elevation model or DEM) is a representation of the bare earth surface (just terrain, no objects). This requires extracting points from a lidar dataset with a classification attribute set as 2. This classification indicates that a point is a ground point. First, a raster has to be created (given an extent and resolution) that corresponds to the lidar dataset. Second, an interpolation method needs to be selected to determine the value of each cell for the raster. These rasterizing methods include (but are not limited to, however these are the

only ones yet implemented in this software and have been selected because of their good speed) nearest neighbor, averaging and inverse distance weighting.

To compare the following rasterizing methods, a test dataset was used pictured in figure 18.

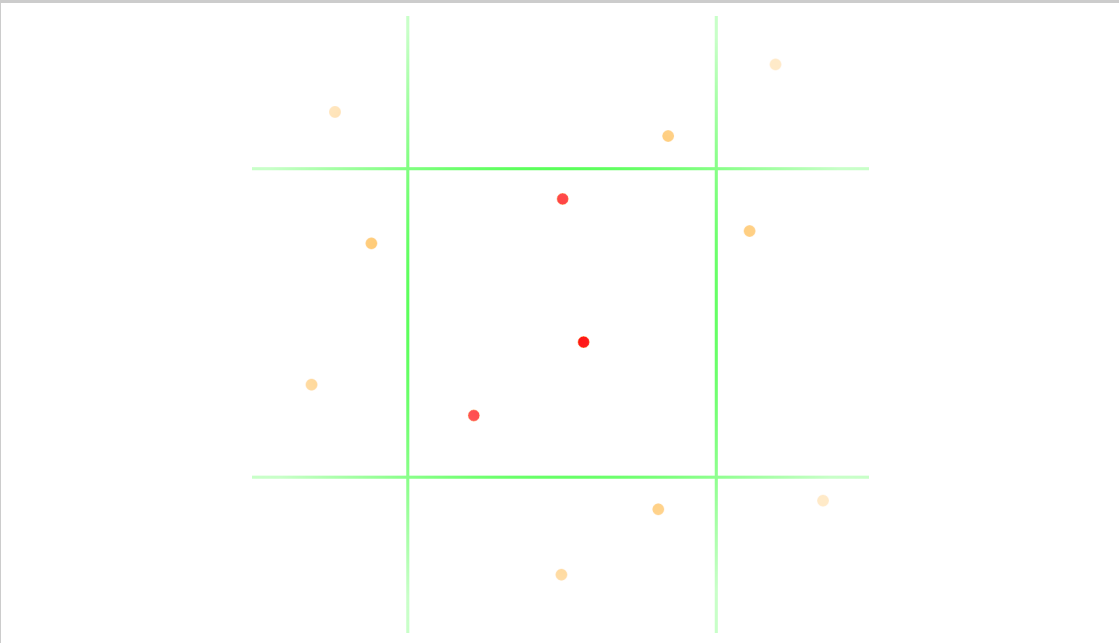


Each points is colored according to its z value.

Figure 18: Test Dataset

Averaging

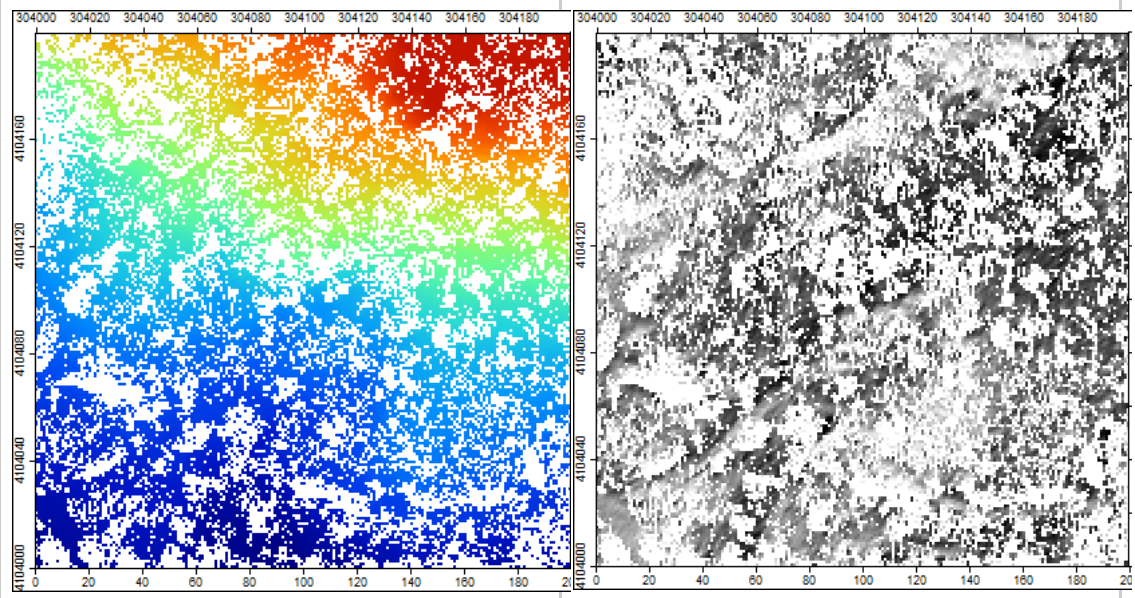
There are 2 ways to determine a cell value based on averaging: using the cell extent and using a distance extent. For the former, much like nearest neighbor, the extent of the cell is used to load points from an LAS file. This time, the z values from all points are averaged and assigned to the cell. For distance extent, a distance threshold is given and a circular extent with a radius equal to the distance threshold and the center positioned at the center of the cell is used to extract points from an LAS file. The points are then averaged as mentioned before.



The red points are used to determine an average and assigned to the cell value.

Figure 19: Average Cell Value Assignment

The results from applying this to the test dataset can be seen in figure 20.



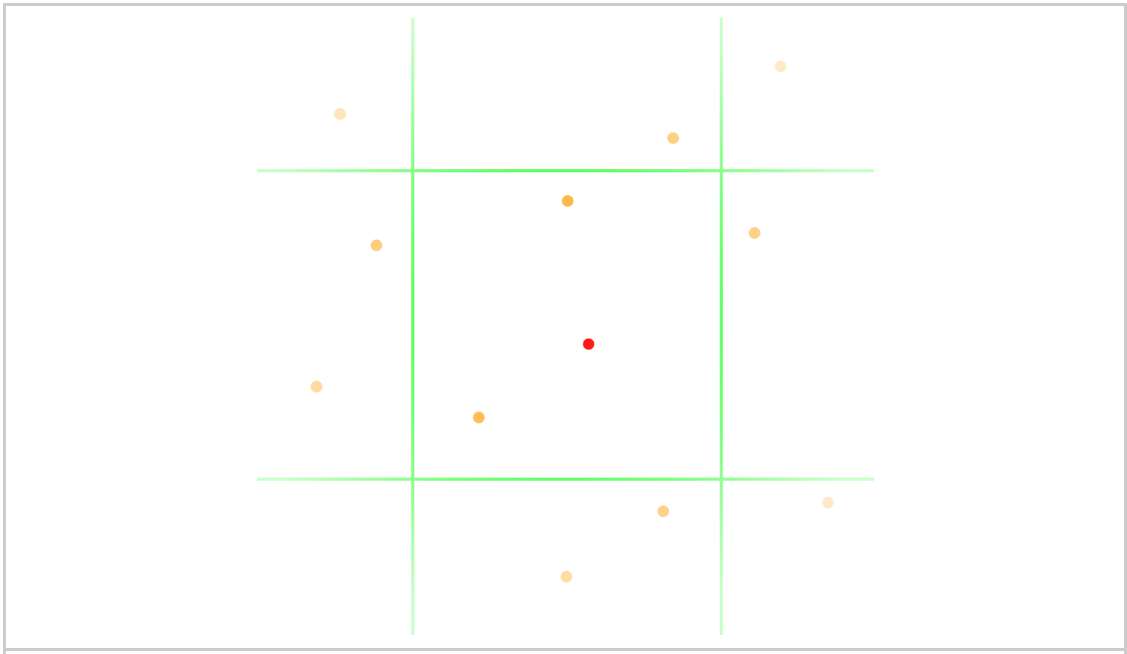
Rasterizing using averaging can contain holes if no points exist in a particular cell. Hillshade isn't much to look at in this case.

Figure 20: Averaging

Rasterizing using averaging does have the effect of missing cell values if there are no points within a cell. However, it should be noted that this method does perform the very fast and is highly accurate. It there is at least one point per cell, this is the ideal method.

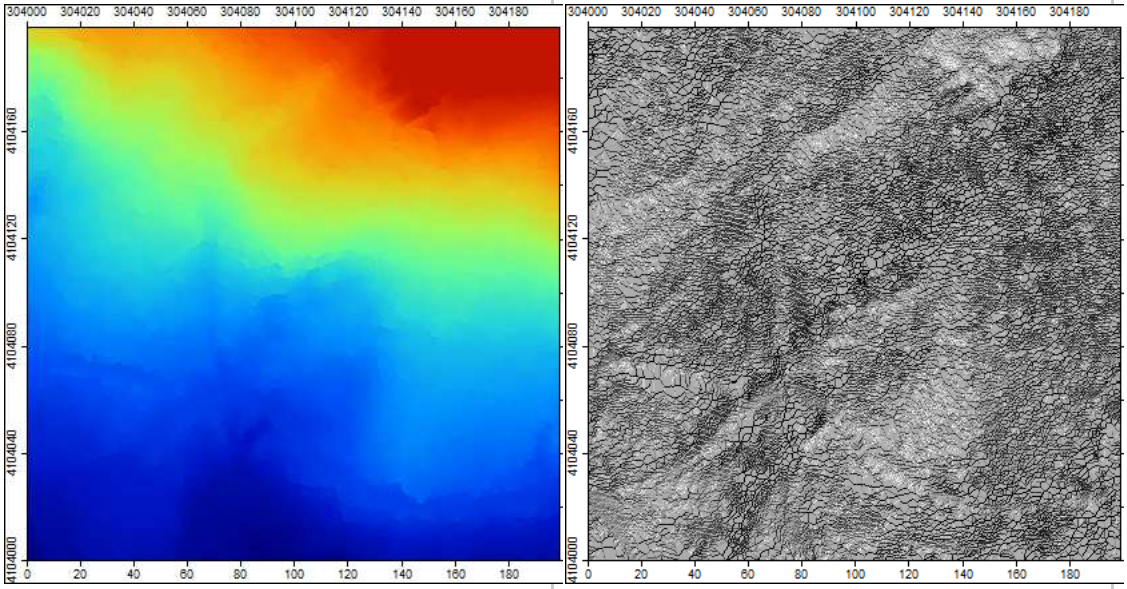
Nearest Neighbor

Nearest neighbor simply finds the point that is nearest the center of a raster cell to assign to that cell. This involves using a circular search extent centered at the center of the pixel and choosing a distance threshold for the radius. Using this search extent points are loaded from an LAS file then the point nearest the center of the pixel is identified.



The red point is the nearest to the center and will be used to represent the cell. This point could also fall outside the focused cell.

Figure 21: Nearest Neighbor Cell Value Assignment



Yields results for every cell, however cell value selections become more inaccurate as seen in the hillshade (a scaling or layering is evident).

Figure 22: Nearest Neighbor

Rasterizing using nearest neighbors could yield results for every cell, however, cells will a nearest point that is far away will be assigned an inaccurate value. This method is fast and it should be noted that it is likely the accuracy will increase as the point density increases.

Inverse distance weighting

Inverse distance weighting (IDW) is an interpolated rasterizing method that is fast, easy to compute and relatively straightforward. It works by weighting a point's value depending on its distance[18]. To determine a cell value, point are extracted from the LAS file using a circular extent with a center that corresponds to that of the pixel and a given distance threshold used as the radius. Each point is then iterated and the distance from the center of the pixel is calculated. Given a power threshold (i.e.: 2 would mean the 2nd power), an equation is applied to each distance to determine the point's influence on the cell's value. Mathematically, this is calculated 3 fold and expressed as follows:

$$numerator = \sum_{i=0}^{n} p.z_i / p.dist_i^m$$

$$denominator = \sum_{i=0}^{n} 1 / p.dist_i^m$$

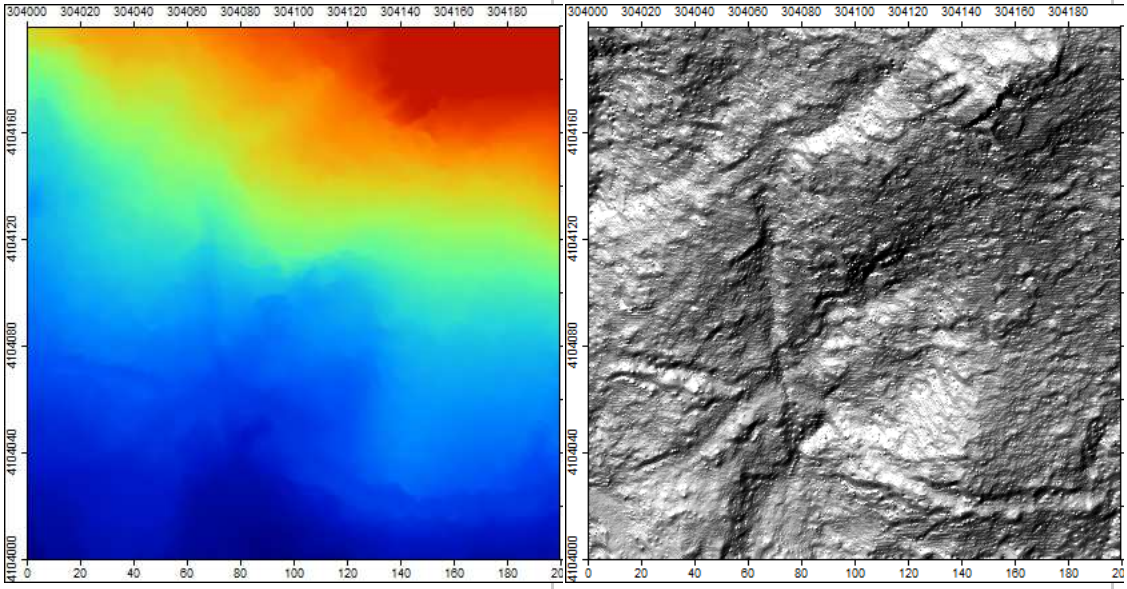
[18] Shepard, Donald. "A two-dimensional interpolation function for irregularly-spaced data." *Proceedings of the 1968 23rd ACM national conference* 1 Jan. 1968: 517-524.

$$cell\_value = numerator/denominator$$

where *n* is the number of points and *m* is the power placed on the distance to determine its influence. Programmatically, the source code is as follows:

```
float IDW (float x, float y, float power, Point3 * pc ) {

     //initialize summations
     float numerator, denominator = 0;

     for ( iterator it = pc.begin(); it != pc.end(); it++ ){
          // calc distance, return z in rare case of distance being
finite
          dist = distance(x,y,it.x,it.y); if(dist<0.00000001) return
it.z;
          numerator = numerator + it.z / pow(dist,power);
          denominator = denominator + 1 / pow(dist,power);
     }

     if ( denominator == 0 ) return -MAX_FLOAT

     return numerator / denominator;
}
```

The figure 23 demonstrates the results after applying IDW.



Yields similar results to nearest neighbor, however, as evident from the hillshade, the result is more accurate.

Figure 23: IDW

From analyzing the hill shade from IDW, it is obvious the results are much more accurate while maintaining the ability to fill missing cells.

DTM Conclusion

These 3 methods for rasterizing a continuous DTM surface are usually sufficient especially as the point density of lidar increases during collection.

4.2.4 Digital Surface Models

A digital surface model (DMS) represent the maximum height of objects at a given raster cell. This requires extracting the maximum point from points that fall within a raster cell; after points are loaded for a given cell, the points are iterated the largest z value is determined. This z value is assigned to the cell. Using the same dataset used in the DTM section (using all the points, not just points classifies as 2), the result is demonstrated in figure 24.

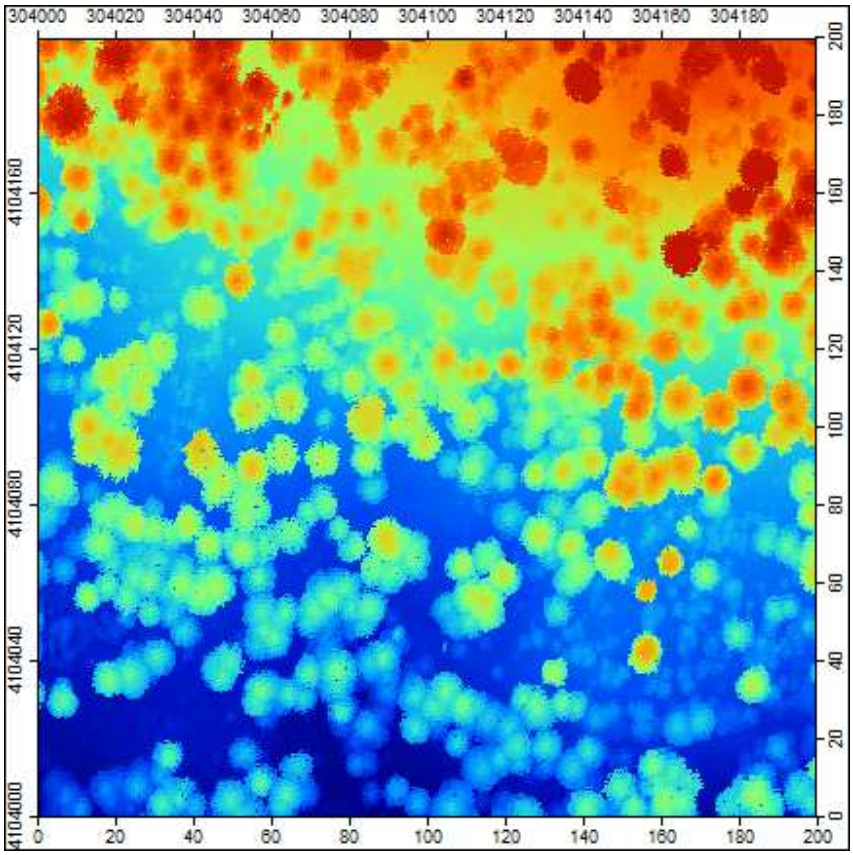


Figure 24: DSM

Since there is a high point return for the top of objects during collection, this method is usually sufficient enough to create a quality DSM.

4.2.5 Regression (a forced relationship)

The most common method for regressing a lidar dataset (specifically aerial based lidar that was evenly collected) is to analyze point densities at different height profiles and relate them to ground-based measurements[19]. The resultant products can be used to drive models in files of ecology, hydrology or fire science. Here, grids are produced at different user define percentiles and only these grids are saved; the actual relationship between plot measurements and these profiles are left to the user. Also, it has sometimes been noticed that simply using these percentile grids is enough (and in some cases perform better).

To create these grids, an extent and cell size are used initialize the grids. Then points are extracted at every cell and normalized using a DTM. Next, points sorted by z values and given a list of percentiles the z value is extracted at each corresponding percentile (ex: if 200 points, the 10th percentile would yield the z value of the 20th point) and assigned to the grid. Each percentile produces a grid.

4.2.6 Point Cloud (Conifer) Segmentation

Point cloud segmentation attempts to segment trees in a dense forest by identifying the tops of trees and iteratively identifying other points as belonging to a tree using a distance threshold[20]. First, the area of interest is loaded into memory from LAS files and a distance threshold is chosen. All points are then and normalized using a digital terrain model. Next, an octree is defined for the loaded points (for efficient searching). The points are then sorted into a list and by z value, highest to lowest, and the first point is marked as a new seed (the top of a tree). As the points are iterated, each is tested using the distance threshold; if it is found to be within that distance to a point that has already been marked as a tree, that point too gets marked as the same tree. Else, it is marked as a new tree. This method is illustrated in figure 25.

[19] Hudak, Andrew T et al. "Regression modeling and mapping of coniferous forest basal area and tree density from discrete-return lidar and multispectral satellite data." *Canadian Journal of Remote Sensing* 32.2 (2006): 126-138.

[20] Li, Wenkai et al. "A new method for segmenting individual trees from the lidar point cloud." *Photogrammetric Engineering & Remote Sensing* 78.1 (2012): 75-84.

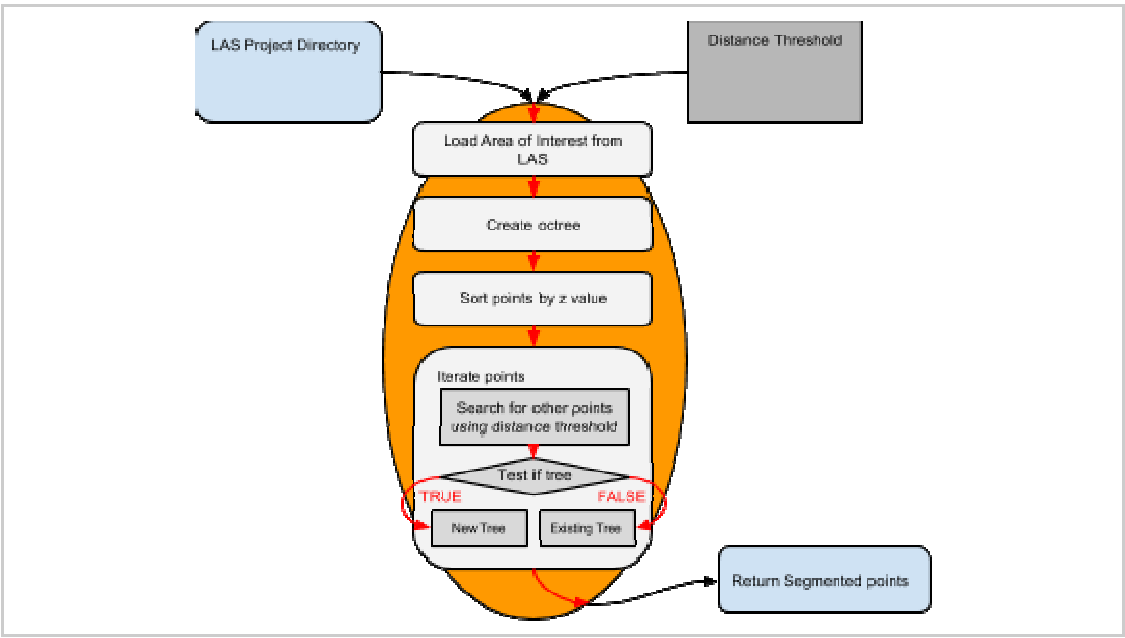


Figure 25: Segmentation Workflow

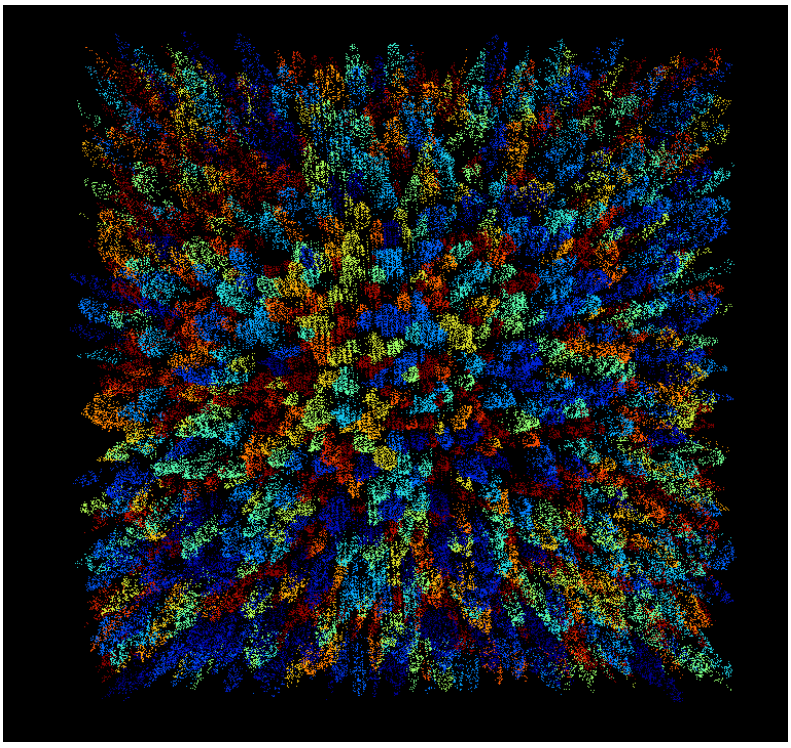
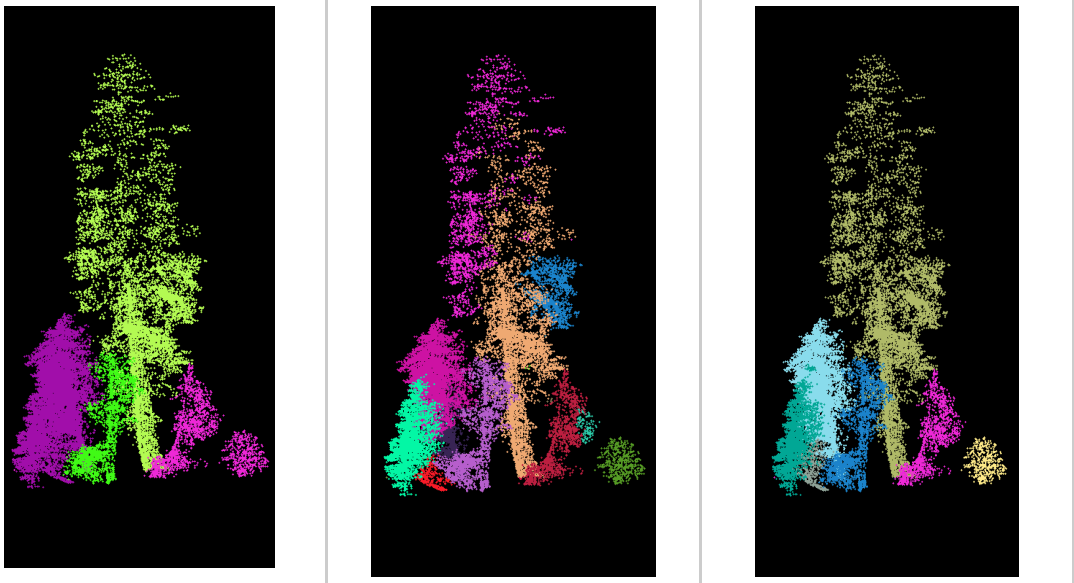


Figure 26: Results from tree segmentation method

This method can be also used to segment trees from the bottom[21] (it should be noted that a height threshold should be placed on the bottom of the loaded points to exclude the ground such that the trees can be identified using their trunks). However, this would likely yield many more seeds (the point that started the segmentation) due to the structure of the trees (i.e.: a long branch near the bottom of the tree could be considered a new seed) creating floating clusters. So, a hybrid method that combines the top-down segmentation approach with the bottom-up segmentation approach is used. After both segmentation methods have been executed, the clusters from the bottom-up approach are analyzed; if it cannot be traversed to the ground it needs to be associated with an existing cluster (that can be traversed to the ground). The results from the top-down approach help in identifying when clusters should be grouped. Figure 27 shows the results from a top-down, a bottom-up and the hybrid of the two.



Top-Down	Bottom-Up	Hybrid of the 2

Figure 27: A Comparison of Top-down, Bottom-up and Hybrid Segmentation

From the figure above, using just the top-down approach fails to segment some trees; this is due to the top of a smaller tree being below or very close to a larger neighboring tree. Using the hybrid method yields more segmented trees and thus is considered more accurate upon a visual inspection.

[21] Lu, Xingcheng et al. "A bottom-up approach to segment individual deciduous trees using leaf-off lidar point cloud data." *ISPRS Journal of Photogrammetry and Remote Sensing* 94 (2014): 1-12.

4.3 Exporting

Exporting data to an LAS file involves creation of a binary file, dumping header information into the file using the proper header format and writing each point to the file using the proper point format.

5. Conclusion

Lidar has become increasingly popular with its uses being applied in many disciplinarians. For many, it can be difficult to even get started with lidar, let alone have the knowledge on how to apply it to their workflow (especially if they have to develop software to process raw lidar). To reduce efforts and save time this software encapsulates useful product creation from lidar. Most lidar software are lacking in feature and/or are expensive and many times purchased to use only a select number of the tools provided that are built on algorithms found in key research papers.

The foundation and some key methods were outlined to create a completely open source software that is efficient and space conscious for the handling and processing of lidar datasets. This software sets out to be a fully transparent solution for loading data and executing common lidar products so users can concentrate on more essential issues pertaining to their disciplinary projects.

Appendix

Key Terms

AOI - Area of Interest
CPU - Central Processing Unit
RAM - Random Access Memory
DTM - Digital Terrain Model
DSM - Digital Surface Model

Referenced structures and functions

Extent Structure

```
// range structure
struct range( ) {
    float min;
    float max;
};

// extent structure
struct extent( ) {
    range x; // min and max x
    range y; // min and max y
};
```

Determine if a point is within an extent:

```
bool is_in_extent( point p, extent e ) {

    if ( p.x > e.x.min && p.x < e.x.max )
        if ( p.y > e.y.min && p.y < e.y.max )
            return true;
    return false;
}
```

Splits and extent into 4 equal quadrants.

```
// Splits an extent into four quadrants
array < extent, 4 > split_quad_extent( extent e ) {

    array < extent, 4 > arr; // allocated array with 4 extents
    float halfx = ((e.x.max-e.x.min)/2);
    float halfy = ((e.y.max-e.y.min)/2);
```

```
    array[0] = new extent(e.x.min,
                      e.y.min,
                      e.x.max - halfx,
                      e.y.max - halfy); // q1
    array[1] = new extent(e.x.min + halfx,
                      e.y.min,
                      e.x.max,
                      e.x.max - halfy); // q2
    array[2] = new extent(e.x.min,
                      e.y.min + halfy,
                      e.x.max - halfx,
                      e.y.max);         // q3
    array[3] = new extent(e.x.min + halfx,
                      e.y.min + halfy,
                      e.x.max,
                      e.y.max);        // q4
}
```

Grid methods built on GDAL:

```
struct Grid{

      GDALDataset* img;
      float offsetX, offsetY, pixelResX, pixelResY;
      int nXSize, nYSize;
      float *buffer;

      float get_value(float x, float y);
      int set_value( float x, float y, float z );
      std::vector<float> get_cell_extents(int x, int y);
      std::vector<float> Grid::get_cell_extents( int pos );
      int create(double X, double Y, double cellsizeX,
          double cellsizeY, int nX, int nY);
      int write(std::string filename);
};
```

This function analyzes whether the x and y locations of a point are within the respective ranges of the search extent. It does this by checking if the value of a particular location direction (x or y) is greater than its corresponding minimum value of the search extent but less than the maximum value of that same search extent. If it is found to be within the extent, the function returns true, else it returns false.

c++ Data Types

Table 9: c++ Data Types, Sizes and Ranges

Name	Description	Size	Range
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255

short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
float	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	Wide character.	2 or 4 bytes	1 wide character

References

1. "LASer (LAS) File Format Exchange Activities - asprs." 2011. 3 Jun. 2015 <http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html>
2. Tobler, Waldo. "On the first law of geography: A reply." *Annals of the Association of American Geographers* 94.2 (2004): 304-310.
3. Chen, Qi. "Airborne lidar data processing and information extraction." *Photogrammetric engineering and remote sensing* 73.2 (2007): 109.
4. Fabio, Remondino. "From point cloud to surface: the modeling and visualization problem." International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences 34.5 (2003): W10.
5. Sajjanhar, Atul, and Guojun Lu. "A grid-based shape indexing and retrieval method." *Australian Computer Journal* 29.4 (1997): 131-140.
6. Tayeb, Jamel, Özgür Ulusoy, and Ouri Wolfson. "A quadtree-based dynamic attribute indexing method." *The Computer Journal* 41.3 (1998): 185-200.
7. Piegl, Les A, and Wayne Tiller. "Algorithm for finding all k nearest neighbors." *Computer-Aided Design* 34.2 (2002): 167-172.
8. Wing, Jeannette M. "Computational thinking." *Communications of the ACM* 49.3 (2006): 33-35.
9. Garlasu, Dan et al. "A big data implementation based on Grid computing." *Roedunet International Conference (RoEduNet), 2013 11th* 17 Jan. 2013: 1-4.
10. Tayeb, Jamel, Özgür Ulusoy, and Ouri Wolfson. "A quadtree-based dynamic attribute indexing method." *The Computer Journal* 41.3 (1998): 185-200.
11. "VM - Virtual Machine | Oracle." 2012. 5 Jun. 2015 <http://www.oracle.com/us/technologies/virtualization/oraclevm/overview/>
12. Beckmann, Norbert et al. *The R*-tree: an efficient and robust access method for points and rectangles*. ACM, 1990.
13. Kothuri, Ravi Kanth V, Siva Ravada, and Daniel Abugov. "Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data." *Proceedings of the 2002 ACM SIGMOD international conference on Management of data* 3 Jun. 2002: 546-557.
14. "ESRI Shapefile Technical Description." 2014. 8 Jun. 2015 <https://www.esri.com/library/whitepapers/pdfs/shapefile.pdf>
15. "GDAL: GDAL - Geospatial Data Abstraction Library." 2004. 3 Jun. 2015 <http://www.gdal.org/>
16. Rusu, Radu Bogdan, and Steve Cousins. "3d is here: Point cloud library (pcl)." *Robotics and Automation (ICRA), 2011 IEEE International Conference on* 9 May. 2011: 1-4.

17. Evans, Jeffrey S, and Andrew T Hudak. "A multiscale curvature algorithm for classifying discrete return lidar in forested environments." *Geoscience and Remote Sensing, IEEE Transactions on* 45.4 (2007): 1029-1038.
18. Zhang, Keqi et al. "A progressive morphological filter for removing nonground measurements from airborne LIDAR data." *Geoscience and Remote Sensing, IEEE Transactions on* 41.4 (2003): 872-882.
19. Shepard, Donald. "A two-dimensional interpolation function for irregularly-spaced data." *Proceedings of the 1968 23rd ACM national conference* 1 Jan. 1968: 517-524.
20. Hudak, Andrew T et al. "Regression modeling and mapping of coniferous forest basal area and tree density from discrete-return lidar and multispectral satellite data." *Canadian Journal of Remote Sensing* 32.2 (2006): 126-138.
21. Li, Wenkai et al. "A new method for segmenting individual trees from the lidar point cloud." *Photogrammetric Engineering & Remote Sensing* 78.1 (2012): 75-84.
22. Lu, Xingcheng et al. "A bottom-up approach to segment individual deciduous trees using leaf-off lidar point cloud data." *ISPRS Journal of Photogrammetry and Remote Sensing* 94 (2014): 1-12.
23. Stein, Michael L. *Interpolation of spatial data: some theory for kriging*. Springer Science & Business Media, 1999.