LEPCC Version 1: Byte Stream Specification

LEPCC license information

Version 1.0, March 02, 2018

Contributors: Thomas Maurer, Ronald Poirrier

For additional information, contact:
Environmental Systems Research Institute, Inc.
Attn: Contracts and Legal Department
380 New York Street
Redlands, CA 92373
E-mail: contracts@esri.com

Contents

LEPCC Version 1: Byte Stream Specification

Overview

LEPCC stands for "Limited Error Point Cloud Compression". This new compression method extends LERC ("Limited Error Raster Compression") from 2D images to 3D point clouds.

In LERC the user specifies the maximum compression error per pixel or MaxZError he is willing to tolerate. The larger that number, the more compression can be expected.
In LEPCC the user specifies the maximum compression error per point in all 3 coordinates x, y, and z. For instance, 1 cm for all 3 coordinates. The larger that number, the more compression can be expected.

Points often carry more information than just the location in space (xyz).
Point clouds derived from images via photogrammetric methods usually also have color (rgb).
Point clouds derived from Lidar can have an intensity value, and other metadata such as a return flag (2 means 3rd point returned from this laser pulse), or a class flag (2 means point has been classified as a point on the ground etc.). For other possible metadata see the Las standard (e.g. http://www.asprs.org/wp-content/uploads/2010/12/LAS_1_4_r13.pdf).

Usually not all of the above data is needed at the same time. For instance, at one time the user wants to render the point cloud in color. Then he needs (xyz) and (rgb) data. At another time he may want to display the intensity instead of color. Then he needs (xyz) and (intensity) data.

Therefore it makes sense to not stuff all data into one compressed blob or file. This would require to always transmit and decompress all of the above data together although only part of it is needed. Instead, we follow a more modular approach. Each of the above kind of data gets compressed independently. Then the user can request and decompress only those portions of the data he really needs. Another advantage of compressing the data into separate packages is that each one of the

compression methods or formats can be changed or upgraded without affecting the others.

Note, that point clouds are usually tiled up into many tiles each covering a certain area in x and y. The compression described here is applied per tile.

The different kinds of data packaged

1. Point locations or (xyz). Almost always needed. Biggest part of the data. 3 double values or 24 bytes per point, uncompressed. Our approach: Compress with controlled loss.

2. Color or (rgb). Often needed. Smaller, 3 bytes per point, uncompressed. Our current approach: Convert to color map, currently 1 byte per point. Maximum error per point can be large.

3. Las intensity, from Las file. 2 bytes per point, uncompressed. Our approach: Compress lossless.

4. Return, class, and similar flags. 1 byte per point each. Usually small numbers. Our approach: Compress lossless.

Before we describe the byte stream specifications for each of the above compressed byte blobs, we describe headers, methods, and other elements that are shared among them.

Shared or common elements

1. Endianness: Little Endian. Same as LAS format.

2. All is done in memory. The data gets encoded into a memory blob, which can be saved to disk or transmitted over the network etc.

3. Top header. Every compressed data module has a top or primary header and a secondary header. The top header has the same structure and size for all compressed data modules. It is

Item	Datatype	Size in bytes
Data / file type specifier	char[10]	10
Version number	uint16	2
Checksum	uint32	4

   The 10 char data type specifier or key serves as an identifier for the compressed data module. It is human readable even in the binary file. The version number is the file or format version number. It is currently 1 for all modules. The checksum is the so called Fletcher32 checksum, see below. Its computation starts at the beginning of the secondary header and ends with the last byte of the compressed byte blob.

4. Checksum. Fast and efficient checksum computed on the compressed byte stream. Purpose is to detect any alteration or corruption of the compressed byte stream before it gets decoded. For details, see https://en.wikipedia.org/wiki/Fletcher%27s_checksum .

5. Simple bit stuffer. For an array of positive integer numbers (uint32), find the number of bits per element needed to encode lossless. The array usually contains 0. So if the maximum number in that array needs nBitsPerElem bits, stuff all N array elements into a byte array of size (N * nBitsPerElem + 7) / 8. The header for this bit stuffed byte array is kept as small as possible. It is usually 2 or 3 bytes long:

Item	Datatype	Size in bytes
Number of bits per element (bits 0-4). Bit 5 flags simple bit stuffing (0) or LUT (1) mode. Here always 0. Bits 6-7 encode the next datatype.	byte	1
Number of array elements	uint32 (0), uint16 (1), or byte (2)	1 or 2 (or 4, rare)

Next we describe the different modules.

LEPCC or compress xyz

This is the main module. It compresses the xyz coordinates of the points.
The data type specifier / identifier for this module is "LEPCC " containing 5 spaces adding up to 10 chars total. This is the first entry of the top header and the beginning of the compressed byte stream.

Fundamental considerations

Before we describe header and data structure, let us first explain the compression method. It is based on the following 3 ideas / observations:

1. The accuracy of the point coordinates is usually limited when captured. This accuracy is usually known. To store the floating point numbers with a precision higher than that means trying to encode or compress noise. In addition, the user may only need a certain accuracy even if the data precision is higher. So we let the user or caller specify the encoding error per point and for each coordinate x, y, and z that he finds sufficient for his purpose. For instance, he can specify [1 cm, 1 cm, 2 cm] as maximum encoding errors for x, y, and z. The error is always given in the same unit as the data. So if the data is in meter, then the correct input is [0.01, 0.01, 0.02].

2. A very common case and what we currently focus on is aerial data. Typical point clouds are the result of reflections from surfaces. This is not always true, and the methods described here may have to be extended accordingly as needed. Here we focus on the case that the point cloud describes a (potentially detailed and noisy) surface z(x, y), such as a world elevation tile. Of course there can be many points from trees, rocks, large buildings, overhanging structures. There can be multiple points for the same (x, y) location and we deal with it. Still we focus for now on the aerial use case (versus terrestrial) with the point cloud coming from a plane capturing the surface of the land, or the point cloud being derived from aerial images by photogrammetric methods. Therefore we are going to treat x and y differently from z. Extending this to the general case may be as easy as trying y(x, z) and x(y, z) in addition to z(x, y) and choosing the best of the 3 options.

3. This is more an observation than an idea. It looks like that natural data are inherently random and noisy, for the most part at least. Using compression methods based on prediction and modeling does not appear to yield much better compression on natural data than simpler methods. An example from 2D image compression is PNG and JPEG. While PNG can yield great compression results on artificial images such as computer screen shots it does usually poor on natural images as captured with a camera. For natural images JPEG is the compression format of choice. Therefore we will usually go for compression methods that treat the signal as a noisy one, and we will use for instance the knowledge about the amplitude of the noisy signal in order to compress it. The reward is much faster encoding and decoding compared to more complex prediction methods. Also the algorithms and code tend to be easier.

The encoding

After these more fundamental considerations, here are the steps of the encoding:

- The user specified tolerances for the x and y coordinates (MaxXError and MaxYError) together with the x and y extent of the point cloud tile to be compressed define a grid or raster in x and y. Each point falls into one xy cell. There can be multiple points in one cell.

- Sort the points in raster order top left to bottom right.

- Fill a first array of type uint32 with the row deltas. Let's call it DeltaRowArray. The first entry is usually 0 (as the first row must have at least one point). If all rows have points, then the remainder of this array contains only 1's. If the raster is very fine, has small cells, and / or there are only few points, then this array has as many elements as there are points. Typically the size of DeltaRowArray is much smaller than the number of points. And most of the elements are small, positive integer numbers.

- Fill a second array of type uint32 with the number of points per row for the non-empty rows. Let's call it NumPointsPerRowArray. Its size is the same as DeltaRowArray. Here, if the raster is densely populated with points, then the numbers in this array can be large. But its size is much smaller than the number of points. In the other extreme, if the raster is very big with small cells, and / or there are only very few points, then the elements of this array are all just 1's.

- Fill a third array of type uint32 with the column deltas. Let's call it DeltaColArray. For each non-empty row, we start counting at 0. But we accumulate all column deltas into the same array. If all raster cells are occupied with points, this array contains only 0's and 1's. If there are multiple points in one cell, the column deltas for the additional points are 0. This array has as many entries as there are points.

- Fill a fourth array of type uint32 with the z values after they got quantized from floating point to integer. The quantization is based on the user specified tolerance in z (MaxZError) and the z value range for this tile (zMax – zMin):

  z_quantized = (unsigned int)((z – zMin) / (2 * MaxZError) + 0.5) .

  This array has as many entries as there are points.

- The last step is to compress each of these 4 arrays using bit stuffing (see above paragraph "Shared or common elements"). As these arrays can be long, we split them into sections of 128 elements each. The last section can be shorter than 128. For each section, find the min value. Subtract the min value from all elements of this section. The min values for all sections are collected in another array of type uint32 called MinArray. Finally bit stuff

this MinArray and all section arrays into the same byte array. Repeat this for all 4 arrays above and we are done.

A detailed example to illustrate the encoding of x and y

Let's go through all steps of encoding the x and y coordinates for one point cloud tile in detail. Assume the user specified parameters MaxXError and MaxYError together with the xy extent of the point cloud tile lead to a xy grid of size (6 x 8). Note that these grids are usually much larger than that. Then let's place 12 points in that grid. The number in each cell denotes the number of points in that cell.

	1		1		
1	1	2			
			3		
					1
			1	1	

- DeltaRowArray = [0, 2, 1, 2, 2]. There are 5 rows with points or non-empty rows. Each delta tells how many rows to jump to get to the next non-empty row.

- NumPointsPerRowArray = [2, 4, 3, 1, 2]. This array has the same size as the previous one. It contains the number of points for each non-empty row.

- DeltaColArray = [1, 2, 0, 1, 1, 0, 3, 0, 0, 5, 3, 1]. This array has the size NumPoints. Each delta tells how many columns to jump to get to the next

non-empty cell. For each non-empty row, we start counting from 0 again. If a cell contains more than one point, delta is 0 for all other points in that cell but the first one.

All arrays above are encoded the same way. For each array, we

- Split up the full length array into sections of length 128 each. In our case, there is only one section.

- For each section, find the minimum value and subtract it from the data elements of that section. Add this min value to a new, smaller array containing the min values for the array being encoded. Here, these min arrays are all of size 1: [0], [1], and [0]. Encode them using bit stuffing. If the array is constant 0, only the header is written. So the sizes needed are 2, 3, and 2 bytes to encode these 3 MinArrays.

- Compress the sections with the data elements after the min value of that section got subtracted. Here we have for DeltaRowArray 2 bits per element or (2 + 2) bytes, for NumPointsPerRowArray the same, and for DeltaColArray 3 bits per element or (2 + 5) bytes.

So the total number of bytes needed for encoding the x and y coordinates is
(2 + 3 + 2) + 4 + 4 + 7 = 22.
Note that the concept of the MinArray looks like overhead here. The number of points is too small in our illustration example. The average number of bytes per point just for x and y coordinates is 1.8 which is still high. For more realistic data samples and settings, the biggest part of the compressed data volume is used for the z coordinate with the x and y parts mostly compressed away.

The secondary header

The LEPCC secondary header (following the top header) uses the following C structs:

```
struct Point3D { double x, y, z; };
struct Extent3D { Point3D lower, upper; };
```

Item	Datatype	Size in bytes
Blob size	int64	8
3D extent	Extent3D	48
Max error in x, y, z	Point3D	24
Number of points	uint32	4
reserved	uint32	4

The blob size is the size of the compressed byte blob in bytes.
The 3D extent has the xMin, xMax, yMin, yMax, zMin, zMax of this point cloud tile, as double values. The decoder ensures that no point gets reconstructed outside this extent.
The next entry is MaxXError, MaxYError, MaxZError, also double.
The number of points is stored as a uint32.
The last header entry is reserved for future use and must be set to 0.
The total header size is 88 bytes.

The compressed data

Here we only repeat what we already explained in the above paragraph “The encoding”. Each of the 4 arrays DeltaRowArray, NumPointsPerRowArray, DeltaColArray, and ZArray, gets encoded the same way. Each array gets split up into sections of 128 elements each. The last section can be smaller. For each section, the minimum value is found and subtracted from all other elements of that section. The minimum values are collected in another array called MinArray. The size of MinArray is equal to the number of sections. Finally the MinArray and all section arrays get bit stuffed as explained in the above paragraph “Shared or common elements”, sub paragraph bit stuffer, into the outgoing byte array or byte blob.

Compress color or rgb

This module compresses the rgb values of the points.

The data type specifier / identifier for this module is “ClusterRGB“. This is the first entry of the top header and the beginning of the compressed byte stream.

Fundamental considerations

Before we describe header and data structure, let us briefly explain the compression method. It is an implementation of the well-known Median-Cut Color Quantization. In a nutshell it works like this: Construct a 3D cube of width 256 in all 3 directions. Assign all input or point colors to its appropriate cell in that cube. Now we have a 3D point cloud in RGB space. (But it is not necessarily a surface or close to one.) Find M (set by the user) cluster centers such that the mean distance of all color points to their cluster centers gets minimized. Then replace each RGB color by the closest cluster center. Usually M = 256. The result is a color map with M colors, and each point gets an index pointing to the right entry in this color map. The compression is 3x.

This compression is not only lossy, the maximum error in RGB can be large. Therefore an alternative approach may be needed for applications other than rendering. However, there are 2 reasons why this method fits here:

- When this method is used on true color RGB images, the main artifacts are often color steps in areas of a small gradient. For instance, a large grass area where the tone of green slightly changes from left to right, can turn into a number of patches each of which represents one of the 256 color cluster centers. This kind of artifact is much harder to notice in a point cloud. A point cloud with all its discrete points is a lot noisier from the start than an RGB color image. So far we have not detected this kind of artifact in rendered point clouds.

- This method is still used today to filter out color noise from scanned color images. Especially scanned documents that originally had only few colors get turned into true RGB color images by the scanner. Using this clustering method forces the noisy color pixels back to the cluster centers and effectively cleans the scanned image from the color noise added by the scanner. In that sense this method fits the spirit of LEPCC and LERC: Cut off the noise.

If the compression errors in RGB should become too large, the option remains to increase M. The current method could be turned into a method of controlled loss as

follows. The user specifies a maximum error in RGB space he can tolerate (MaxRGBError). We repeat the clustering for increasing M until the maximum error in RGB is below MaxRGBError. If M and the color map should get too large, switch to raw binary or another lossless encoding method.

The secondary header

The ClusterRGB secondary header (following the top header) is

Item	Datatype	Size in bytes
Blob size	int64	8
Number of points	uint32	4
Number of color map colors	uint16	2
Color lookup method	byte	1
Color index compression method	byte	1

The blob size is the size of the compressed byte blob in bytes.
The number of points is stored as a uint32.
The number of color map colors or entries is stored as a uint16. A color map with more than 256 colors is possible here.
The color lookup method is currently one of these 3: 0 (none), 1 (lossless), or 2 (clustering). Further explained below.
The color index compression method is currently one of these: 0 (none), 1 (all const).
The total header size is 16 bytes.

The compressed data

The encoding starts with counting the number of points and the number of different colors present. As the color map needs some space (3 * number of color map entries), for a small number of points it may be better to just store the colors as is lossless. Here are the different cases:

- Number of points too small. Set color lookup method to 0 (none), and color index compression method to 0 (none). Omit the color map. Store the colors raw, 3 bytes RGB per point, starting behind the secondary header. Lossless.

- Number of points large enough to store color map. However, there is only one constant color. Set color lookup method to 1 (lossless), and color index compression method to 1 (all const). Store the color map having just one color entry, starting behind the secondary header. Omit the color indexes. Lossless.

- Number of points large enough to store color map. However, the number of different colors present is less or equal to 256. Set color lookup method to 1 (lossless), and color index compression method to 0 (none). Store the color map using 3 bytes RGB per color map entry, and then the point indexes using 1 byte per point. Lossless.

- Number of points large enough to store color map. The number of different colors present is larger than 256. Run the 3D clustering algorithm to find the best 256 RGB color cluster centers. Set color lookup method to 2 (clustering), and color index compression method to 0 (none). Store the color map using 3 bytes RGB per color map entry, and then the point indexes using 1 byte per point. Lossy.

It is probably possible to further compress the color indexes, lossless. The order of the colors in the color map can be changed any way we see fit. Reordering them may open up ways to better compress the color indexes. Then such a new method can be marked in the header as a new entry in the color index compression method field.

Compress LAS intensity

This module compresses the intensity values of the points.
The data type specifier / identifier for this module is “Intensity “ containing 1 space adding up to 10 chars total. This is the first entry of the top header and the beginning of the compressed byte stream.

Basic considerations

According to the LAS standard, the LAS intensity is stored as a uint16 or unsigned 16 bit integer. (Again, see http://www.asprs.org/wp-content/uploads/2010/12/LAS_1_4_r13.pdf).
It describes the strength or intensity of the returned laser pulse. The real data range of the intensity can vary, it depends on the sensor type. Quite often the range is less than 16 bit. In order to make the intensity values comparable across different sensor types, the LAS standard recommends scaling it up to 16 bit by left shifting the numbers accordingly.
Apparently this recommendation is more often not followed than followed. We have found many data sets with the intensity having a different range (e.g. 8 bit), or the intensity values got somewhat scaled up (e.g. factor 10) but not necessarily to full 16 bit and not by the recommended left shift.
We also found the intensity to be a very noisy signal. The numbers can jump a lot between very small (close to 0) and very large (close to maximum value).

The encoding

Is simple. First, we check if the intensity values have been scaled up or not. If yes, we revert the upscaling and store that scale factor. Second, we check if the number of bits per point (bpp) needed equals either 8 or 16. As mentioned before, 8 bpp appears to be a common case and therefore qualifies for further optimization. If bpp is 16, we encode the intensity array binary raw using 2 bytes per point. If bpp is 8, we encode the intensities as one byte array using 1 byte per point. If bpp is neither 8 nor 16, the downscaled intensity value array gets bit stuffed into a byte array using bpp bits per point. The compressed data volume is usually 1 to 1.5 bytes per point. The encoding is lossless.

The secondary header

The Intensity secondary header (following the top header) is

Item	Datatype	Size in bytes
Blob size	int64	8

Number of points	uint32	4
Scale factor	uint16	2
Bits per point (bpp)	byte	1
reserved	byte	1

The blob size is the size of the compressed byte blob in bytes.
The number of points is stored as a uint32.
The scale factor is to be multiplied with the encoded intensity values by the decoder.
Bits per point (bpp) tells the number of bits used per point or element. If 8 or 16, the intensity values are encoded raw binary as a byte or uint16 array, not bit stuffed.
The last header entry is reserved for future use and must be set to 0.
The total header size is 16 bytes.

The compressed data

Is just one single array, either a bit stuffed byte array, or, if bpp equals 8 or 16, the (potentially downscaled) intensities are written as byte or uint16 array. Due to the noisy nature of the intensity signal we could not find any extra compression gain in splitting up the bit stuffed array into sections, subtracting min values or similar. The intensity array, bit stuffed or not, is encoded directly behind the secondary header. The encoding is lossless.

Compress LAS flag bytes such as return, class, or mask flag

This module omitted at this time.

The input data is an array of bytes with one byte per point. It can be an array of return flag bytes, or class flag bytes, or mask bytes. Using standard gzip compresses such an array to typically 2 bits per point or less. The resulting compressed data volume is small compared to the other modules.

That’s why there is currently no interest in a custom compression module for this kind of data. If the dependency on gzip should become an issue, the Huffman encoder contained in the LERC package and the bit stuffer can be used in combination to aim at a similar compression performance, and at potentially higher speed.