Signals for Point Clouds

Samuel Dean

Contents

1 Signals

Points in a point cloud possess attributes. These may include, and may not be limited to:

- A position (x, y, z) in 3D space. (Three 32-bit integers.)
- An intensity. (An unsigned 16-bit integer.)
- Return number and number of returns (An unsigned 8-bit integer)
- Classification. (An unsigned 8-bit integer.)
- Time. (A 64-bit float.)

We see a **signal** as any mathematical function which may be generated by these attributes.

Our 3D signals will be stored as a point cloud with the value of our signal represented as a in intensity.

Remark 1.1. Some of our signals are given by regression coefficients. These statistics only attain values between 0 and 1, so it's important to remain aware of the caveat that if we store these raw numbers as intensities (i.e. 16-bit integers), they will be set to zero. Therefore, we will multiply these signals by 1000 before plotting them. When we colour our point clouds in Displaz, we will divide by 1000 again to get a number between 0 and 1, which will allow us to represent a colour, in order to make the value of the signal visibly high or low.

1.1 Intensity histograms

A simple 1-dimensional signal is the intensity histogram for a point cloud. For the point cloud in Figure 15, we can make a histogram of the intensities in a python shell as follows.

```
>>> import matplotlib.pyplot as plt
>>> import numpy, laspy
>>> inFile = laspy.file.File("RealDataClipped.las", mode = "r")
>>> N
4450
>>> plt.hist(inFile.intensity, bins = N)
(array([ 6., 10.,  9., ...,  0.,  0.,  1.]), array([  25.        ,
   25.99977528
,   26.99955056, ..., 4472.00044944,
       4473.00022472, 4474.        ]), <a list of 4450 Patch objects>)
>>> plt.title("Intensity␣Histogram")
Text(0.5, 1.0, 'Intensity␣Histogram')
>>> plt.show()
```

The resulting plot is shown in Figure 1.

Figure 1: An intensity histogram of a large tile

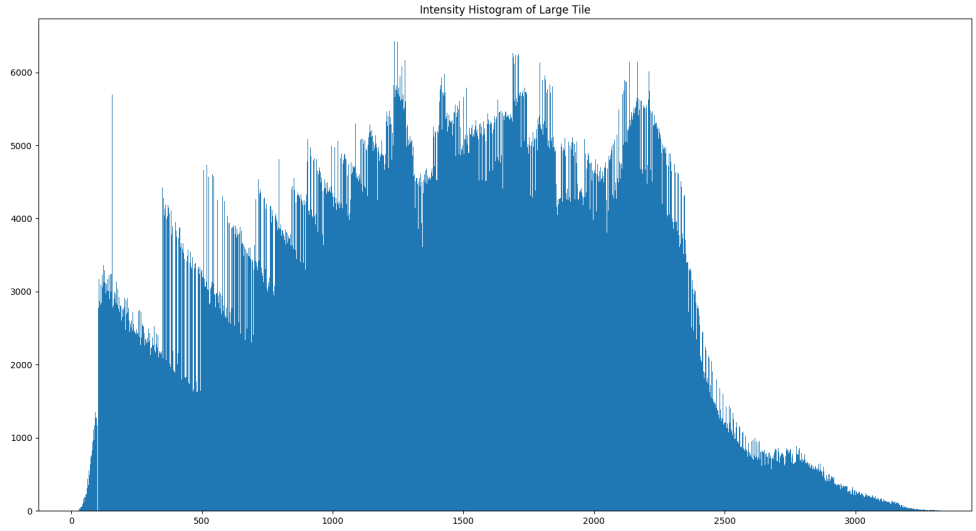
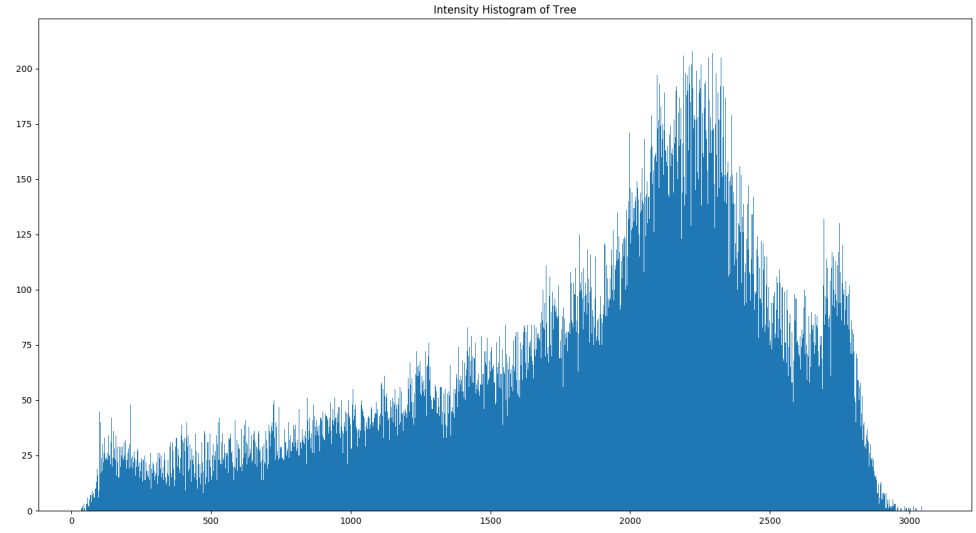


Figure 2: An intensity histogram of a tree



1.2 Statistical signals

Suppose we have a collection of 3-vectors

$$\{\mathbf{r}_1 = (x_1, y_1, z_1), \mathbf{r}_2 = (x_2, y_2, z_2), \ldots, \mathbf{r}_n = (x_n, y_n, z_n)\}$$

which constists of positions in the point cloud. We can compute the **covariance** of the vector $(x_1, x_2, \ldots, x_n)$ and $(y_1, y_2, \ldots, y_n)$, given by

$$c_{\mathrm{xy}} = \mathrm{cov}((x_1, x_2, \ldots, x_n), (y_1, y_2, \ldots, y_n)) = \frac{1}{n^2} \sum_i \sum_{j>i} (x_i - x_j)(y_i - y_j).$$

We could also take yz-covariance and zx-covariance. So, we get the covariances

$$c_{\mathrm{xy}},\ c_{\mathrm{yz}},\ c_{\mathrm{zy}}$$

and local standard deviation signals,

$$\sigma_{\mathrm{x}}^2 = c_{\mathrm{xx}},\ \sigma_{\mathrm{y}}^2 = c_{\mathrm{yy}},\ \sigma_{\mathrm{z}}^2 = c_{\mathrm{zz}}.$$

Many of the signals herein are built out of statistics such as these. But we can't call a single statistic a "signal". We'll instead take statistics at a local scale. Most of the signals will have an associated radius, which we will denote by ϵ. Given any statistic which makes sense for a family of 3-vectors, we can build a signal f by using the following recipe to evaluate $f(\mathbf{r})$ at any point $\mathbf{r} = (x, y, z)$:

- Take the ball
$$B_\epsilon(\mathbf{r}) = \{\mathbf{r}_i : i = 1, 2, \ldots, n, \|\mathbf{r} - \mathbf{r}_i\| < \epsilon\}.$$
- Take the statistic inside the ball $B_\epsilon(\mathbf{r})$, and let this be the value $f(\mathbf{r})$.

Therefore, for any point cloud, we obtain covariance signals $c_{\mathrm{xy}}(\mathbf{r})$, $c_{\mathrm{yz}}(\mathbf{r})$, $c_{\mathrm{zy}}(\mathbf{r})$ and standard deviation signals $\sigma_{\mathrm{x}}(\mathbf{r})$, $\sigma_{\mathrm{y}}(\mathbf{r})$, and $\sigma_{\mathrm{z}}(\mathbf{r})$. (Note that we can't actually evaluate these signals at points where $B_\epsilon(\mathbf{r}) = \varnothing$, but at these points we can just set $f(\mathbf{r}) = \mathrm{NAN}$ ("not a number").)

1.3 Boolean correlation

In this section we'll discuss correlation, whereby we loop through all of the points in the data set, and change their classificiation to 4 (in Displaz, this means we colour them green) if

$$|c_{\mathrm{xy}}(x,y,z)| \geqslant k \cdot \sigma_{\mathrm{x}}(x,y,z)\sigma_{\mathrm{y}}(x,y,z)$$
$$|c_{\mathrm{yz}}(x,y,z)| \geqslant k \cdot \sigma_{\mathrm{y}}(x,y,z)\sigma_{\mathrm{z}}(x,y,z)$$
$$|c_{\mathrm{zx}}(x,y,z)| \geqslant k \cdot \sigma_{\mathrm{z}}(x,y,z)\sigma_{\mathrm{x}}(x,y,z)$$

at their position (x,y,z). Here k is a chosen clip, close to one. One of the goals of this section is to demonstrate the limits of geometric intuition when dealing with point clouds. We'll use $k = 0.95$, looking only for strong correlation. We take a moment to emphasise that this procedure is itself still a signal: It's just very discrete, where we give a positition (x,y,z) a boolean value – TRUE or FALSE – depending on whether or not it does lie in an area of high or low correlation[1].

Each of these tests will depend on our choice of radius ϵ of the ball $B_\epsilon(x,y,z)$ which we draw around each point (x,y,z) before taking the covariances inside that ball. We'll plot this signal along an artificial conductor which has a sag of three meters, and hangs from (absent) pylons which are 50 meters apart. The las file for this artificial conductor is pictured in Figure 3. The code to generate this artificial conductor is given in Section **??**.

We'll now generate plots of binary correlation along this fictitious cable, using spheres of radius $\epsilon = 0.15, 0.5, 1, 2$ (using a correlation clip $k = 0.95$). To do this in a python shell, do the following:

```
>>> import laspy, numpy, master
>>> inFile = laspy.file.File("50MeterFalseConductor.las", read = "r")
>>> for radius in [0.15, 0.5, 1, 2]:
...     master.crltd_pts(inFile,radius,"CorrelationColourCable"+str(int
    (radius)).zfill(3)+"_"+str(1000*(radius-int(radius))).zfill(3))
```

The output for $\epsilon = 0.15$ is in Figure 4. Since any analytical curve is arguably "locally straight", we would naïvely expect that zooming in on this curve to such a fine scale would reveal something indistinguishable from a straight line. In a perfect platonic mathematical universe, maybe so, but here we are doing something more dangerous. Each point on the catenary is 0.05 units apart in the x-direction. Therefore, near its critical point, a sphere which is 30cm in diameter will contain seven points which are, because we are working on a 1cm scale, in a straight line, which leads us to colour lots of points. However, as we move up the catenary, the curve experiences 1cm (0.01 unit) step changes, meaning we can't quite draw a straight line through the local neighbourhoods of points. As the catenary continues outwards, further from the critical point, its radius of curvatures increases dramatically, and we start to colour in the points very consistently. Figures 5 ($\epsilon = 0.5$) shows a situation similar to 4 but less extreme due to the increase in scale, with some instability remaining. Figures 6 ($\epsilon = 1$), and 7 ($\epsilon = 2$) show a dramatically improved situation, but we can't get the points near the critical points because we those points have the lowest radius of curvature and we are now looking at the curve at too high a scale. The only way we could detect every point on a catenary when looking for correlations (with $k = 0.95$) would be to take very small scales – but that would require our data to be more heavily populated.

[1] A small caveat, for edge cases: We should ensure we use $\geqslant$ here rather than $>$, because it may be that the x and y values are constant in our chosen region, but there is still a vertical straight line. If we required $|c_{\mathrm{xy}}(x,y,z)| > k \cdot \sigma_{\mathrm{x}}(x,y,z)\sigma_{\mathrm{y}}(x,y,z)$ then we may require that $0 > 0$ (not much chance of this being true!), and accidentally preclude points from this region!

Figure 3: Artificial conductor, 50 meters wide, 3 meter sag.

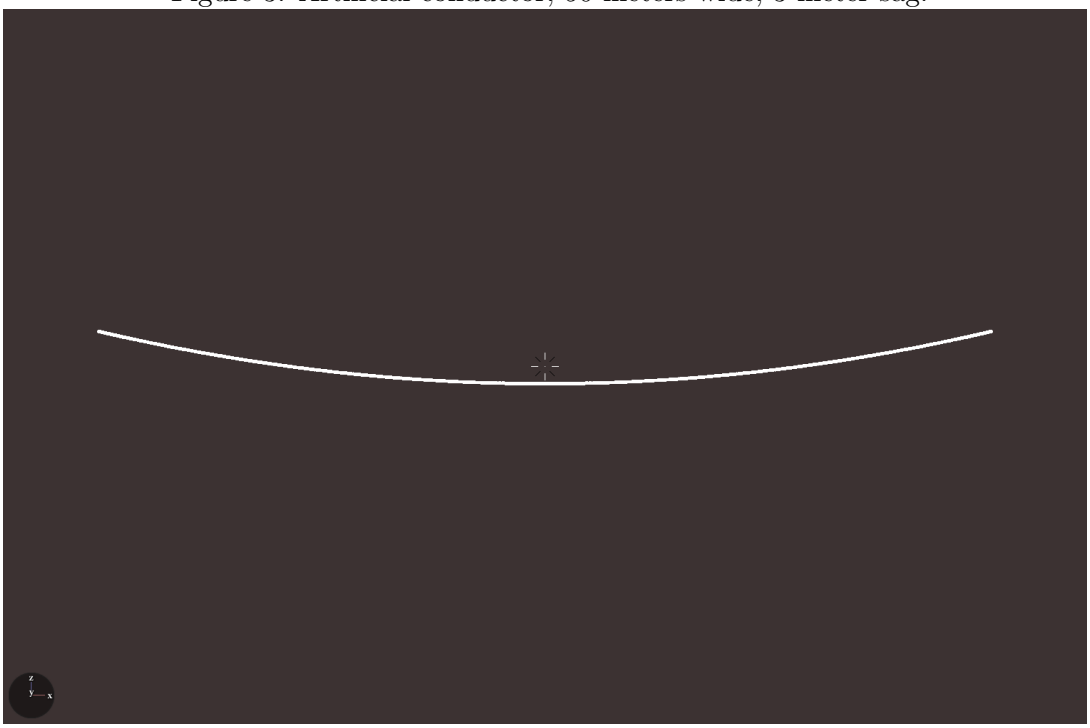


Figure 4: $\epsilon = 0.15$.

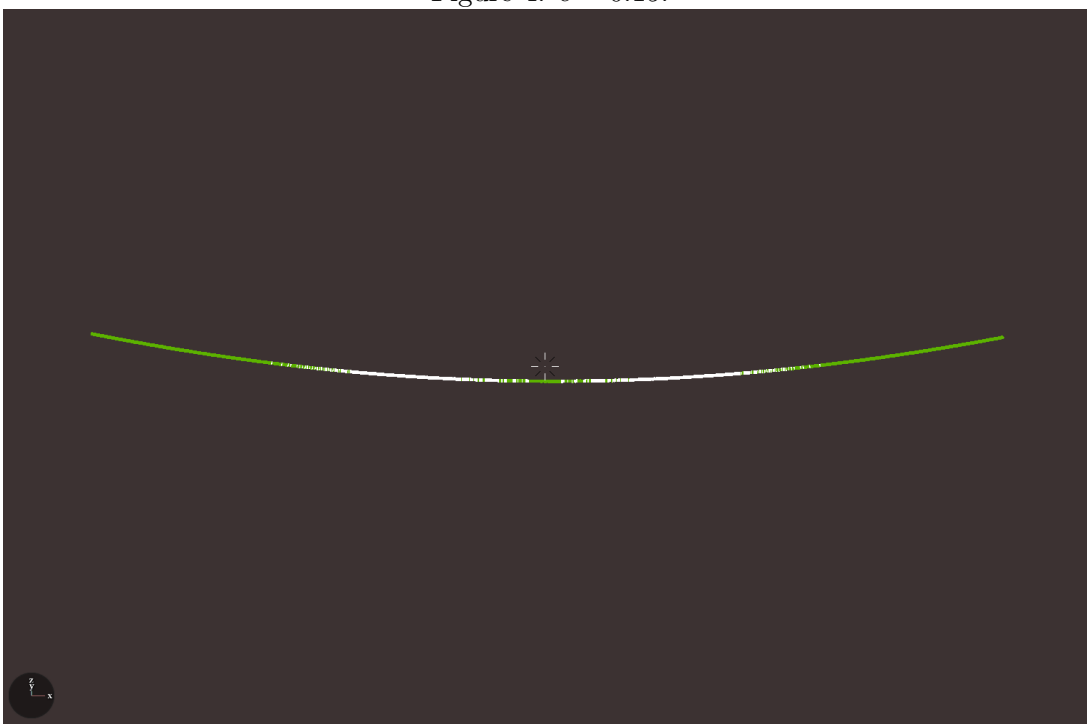


Figure 5: $\epsilon = 0.5$.

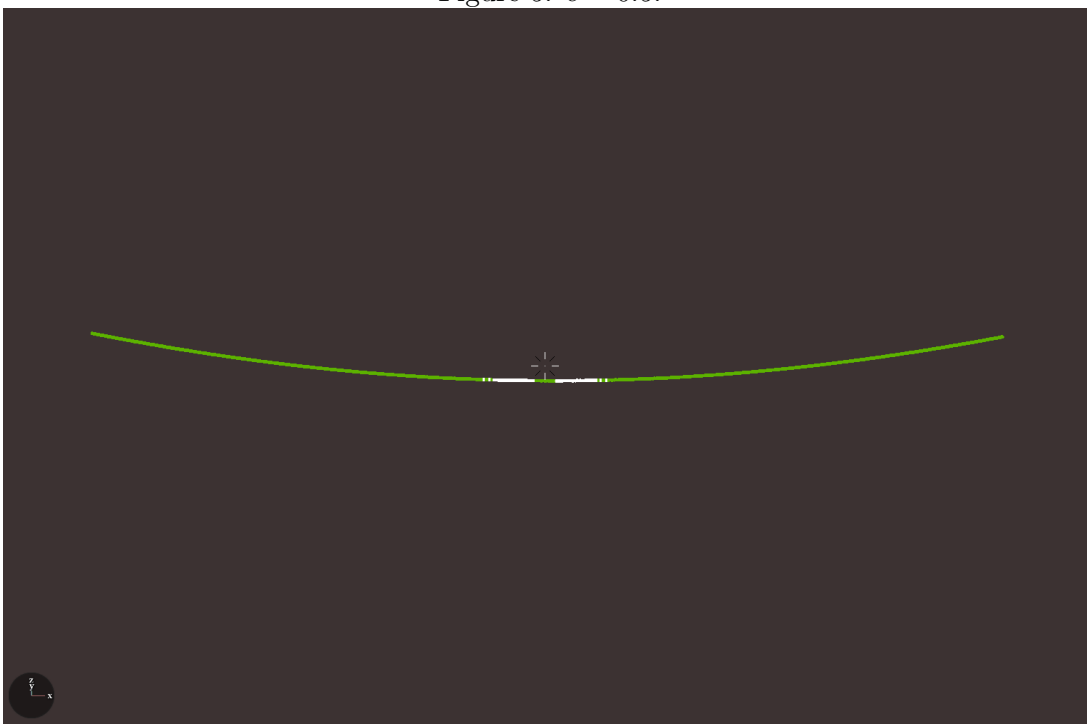


Figure 6: $\epsilon = 1$.

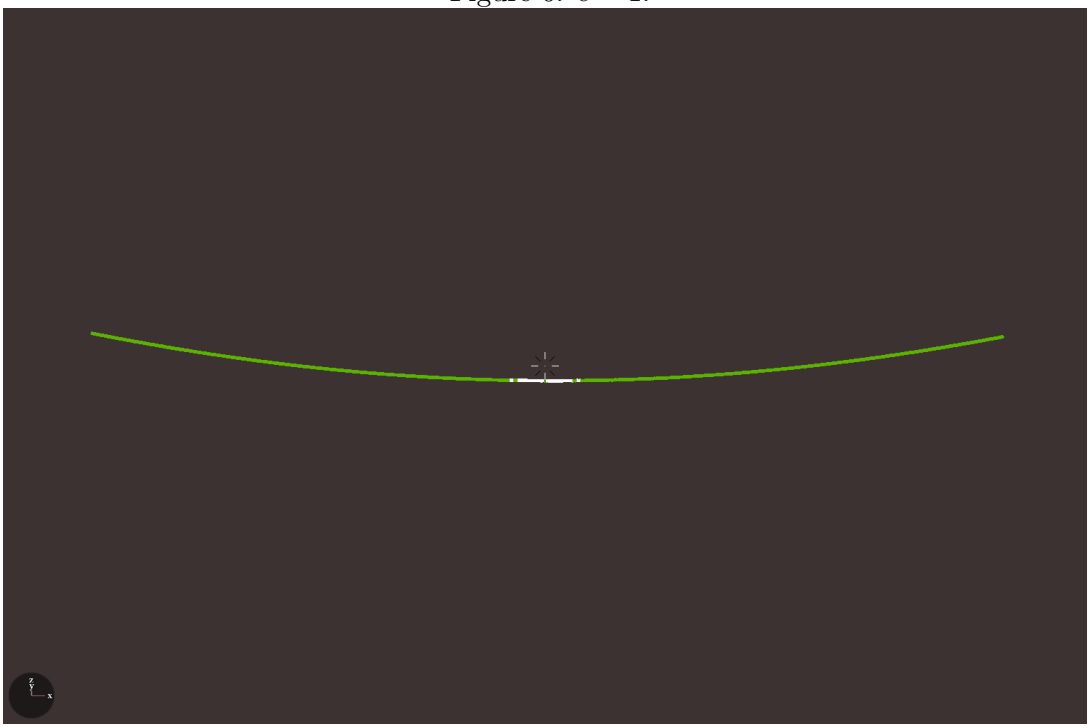
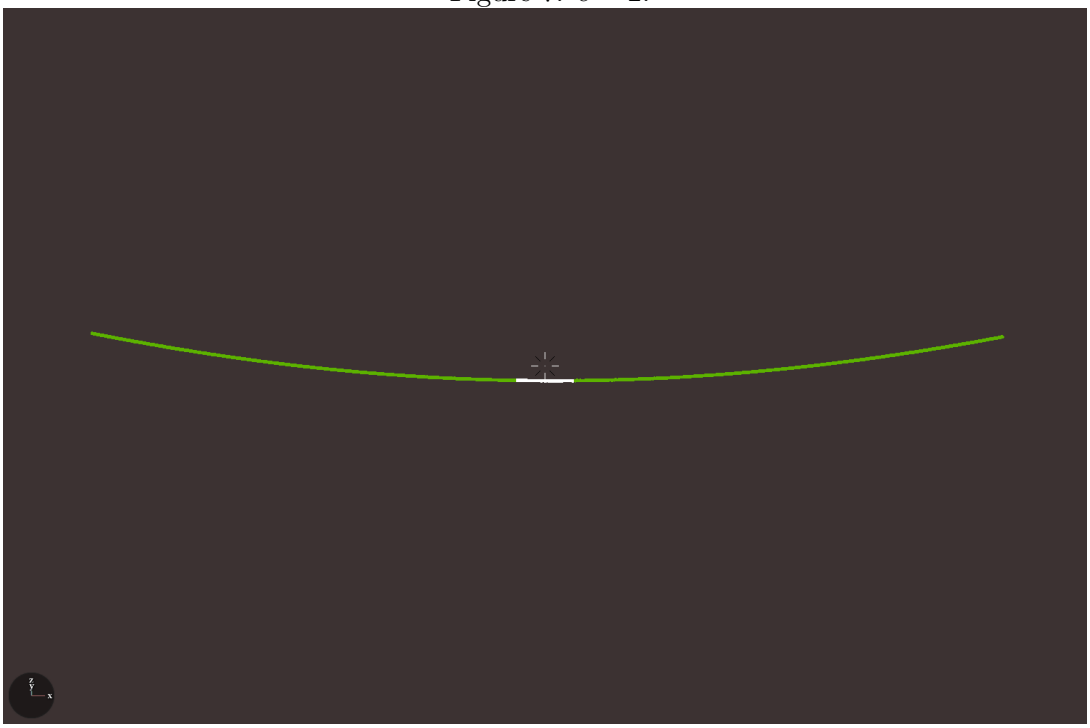


Figure 7: $\epsilon = 2$.



1.4 Local regression

Let ϵ be a chosen radius. At each point (x, y, z), if the ball $B_\epsilon(x, y, z)$ contains more than one position from the point cloud, we may assign a "regression" to the sphere. If

$$\sigma_{\mathrm{x}}(x, y, z)\sigma_{\mathrm{y}}(x, y, z)\sigma_{\mathrm{z}}(x, y, z) > 0$$

the regression is equal to

$$\frac{|c_{\mathrm{xy}}(x, y, z)c_{\mathrm{yz}}(x, y, z)c_{\mathrm{zx}}(x, y, z)|}{\sigma_{\mathrm{x}}(x, y, z)\sigma_{\mathrm{y}}(x, y, z)\sigma_{\mathrm{z}}(x, y, z)}.$$

If this statistic is close to 1, we can say that we can draw a straight line through the ball $B_\epsilon(x, y, z)$ with most of the points in the ball lying quite close to the line. If any one of the standard deviations $\sigma_\bullet(x, y, z)$ are equal to zero, a two-dimensional analogue is used. If two of them are zero, we can safely say the points lie on a straight line parrallel to the axes, so we just set the correlation equal to 1. If all three standard deviations are zero, all the points inside the sphere have the same position, and we can't say much, so we disregard regression, setting it to NAN ("not a number"). For the purposes of plotting the function.

Example 1.2. We use a point from a pylon in some real lidar data. We use a radius of one meter.

```
>>> import numpy, laspy, master
>>> inFile = laspy.file.File("TestArea.las", mode = "r")
>>> a=385684.080
>>> b=208241.38
>>> c=88.59
>>> master.lreg(inFile,a,b,c,1)
0.5983202923699099
```

Asking for the regression contained in a sphere 100 meters above that same point is ridiculous because it contains no points, so we don't get a numerical answer.

```
>>> master.lreg(inFile,a,b,c+100,1)
nan
```

Example 1.3. We can plot local regression at regularly spaced point, using a function in our master module. For a plot of local regressions with $\epsilon = 1$ of our false conductor, do the following.

```
>>> import laspy, numpy, master
>>> inFile = laspy.file.File("TestArea.las", mode = "r")
>>> master.lcorrplot(inFile, 1,"TestAreaLocalCorrPlot")
```

Compare Figures 8 and 9 to see the results. In the plot in Figure 8, "nan" values are replaced by zero, and then all the zeros are deleted. Values closer to one (correlated) are coloured with more red, and values closer to zero (uncorrelated) are coloured with more blue. As expected, the conductors are the reddest areas. The second conductor seems less red, but that's because of the quality of the test area used.

Figure 8: Test area.

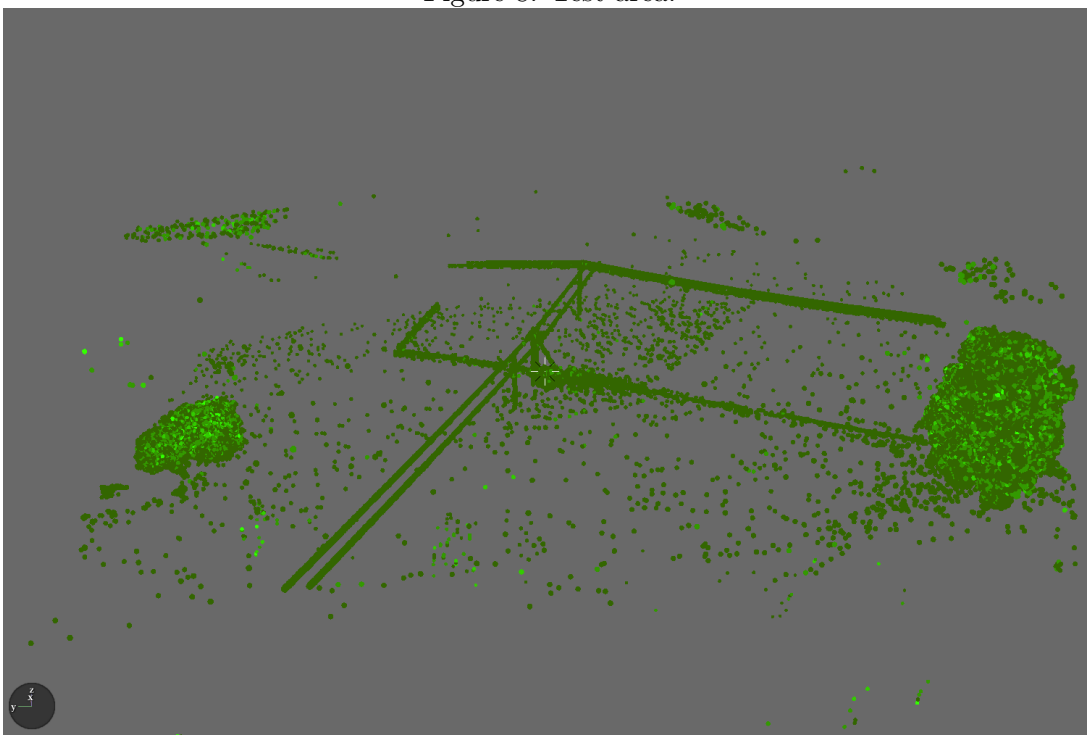
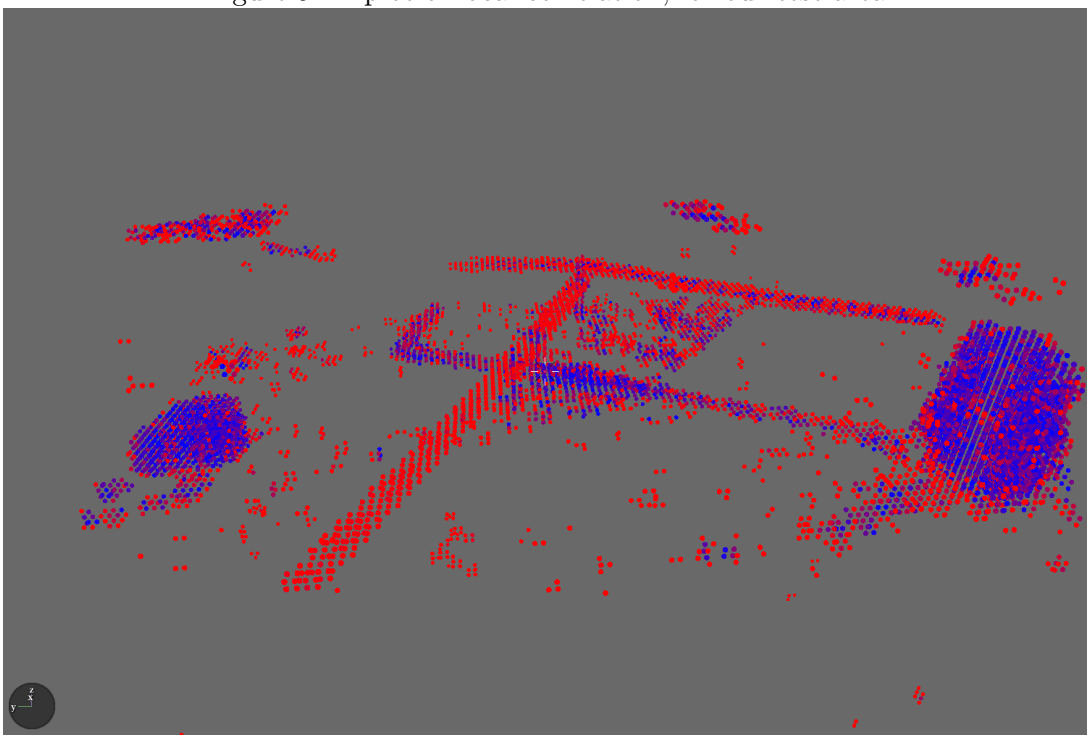


Figure 9: A plot of local correlation, for our test area.



1.5 Standard deviation of attributes

Choose an attrinute: Intensity is a natural one for lidar point clouds. When we take our chosen radius ϵ and a position (x, y, z), the ball $B_\epsilon(x, y, z)$ contains a number, n say, of points from the point cloud, which then generate a vector $(a_1, a_2, \ldots, a_n)$ for the chosen attribute. We can then calculate the standard deviation of that vector, and have that be our signal. Calling our signal f,

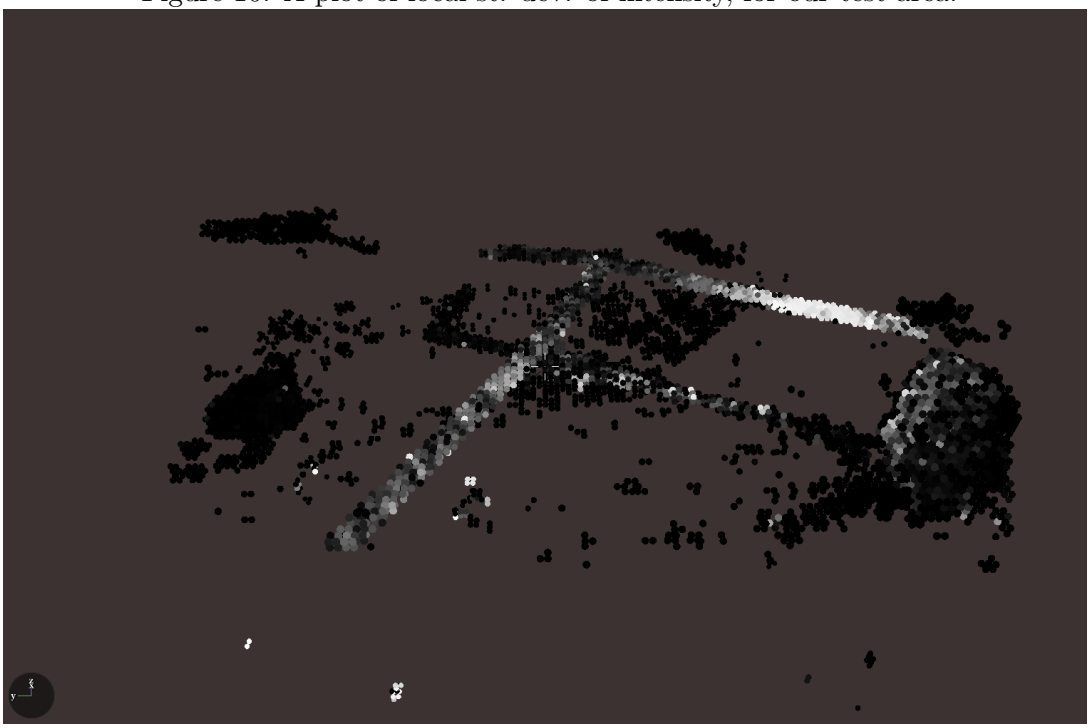
$$f(x, y, z) = \frac{1}{n}\sqrt{\sum_{i=1}^{n}(a_i - \overline{a})^2}$$

where

$$\overline{a} = \frac{1}{n}\sum_{i=1}^{n} a_i.$$

For the test area in Firgure 8, a high-contrast low-exposure plot of local standard deviation of intensity is shown in Figure 10, with this signal plotted as intensity itself. Again, all the zeros have been removed. The areas which have the highest values for this signal are clearly the conductors, with some vegetation.

Figure 10: A plot of local st. dev. of intensity, for our test area.



1.6 Coplanarity

Suppose we have a family of points $\{(x_i, y_i, z_i) : i = 0, 1, , 2 \ldots, n-1\}$. We would like a measure for the coplanarity of these points. Normal linear regression (the type we've used so far) only gives a measure of colinearity. If the standard deviation of the x_i in this family, σ_{x}, is zero then the x_i are all equal, and the n points clearly lie in a vertical plane, and similarly for the y_i and the z_i. Therefore, we assume that the standard deviations satisfy

$$\sigma_{\mathrm{x}}\sigma_{\mathrm{y}}\sigma_{\mathrm{z}} > 0.$$

Since the matrix

$$C = \begin{pmatrix} \sigma_{\mathrm{x}}^2 & c_{\mathrm{xy}} & c_{\mathrm{xz}} \\ c_{\mathrm{yx}} & \sigma_{\mathrm{y}}^2 & c_{\mathrm{yz}} \\ c_{\mathrm{zx}} & c_{\mathrm{zy}} & \sigma_{\mathrm{z}}^2 \end{pmatrix}$$

is symmetric, it is orthogonally diagonalisable – so says the spectral theorem.

For any vector

$$v = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

we have

$$v^T C v = \overline{w}$$

where

$$w_i = (a(x_i - \overline{x}) + b(y_i - \overline{y}) + c(z_i - \overline{z}))^2.$$

Now, the points (x_i, y_i, z_i) lie in a plane if and only if, for some vector $v \neq 0$, the resulting vector w is zero, which occurs precisely when $\overline{w} = 0$ because $w_i \geqslant 0$.

Since C is orthogonally diagonialisable, there is an orthogonal basis $\{v_1, v_2, v_3\}$ of $\mathbb{R}^3$ which consists of eigenvectors of C. Let $\{\lambda_1, \lambda_2, \lambda_3\}$ be the correpsonding eigenvalues. If $v = a_1 v_2 + a_2 v_2 + a_3 v_3$ then the corresponding vector w is such that

$$\overline{w} = v^T C v = a_1^2 \lambda_1 + a_2^2 \lambda_2 + a_3^2 \lambda_3.$$

Since C generates a non-negative bilinear form, $\lambda_1, \lambda_2, \lambda_3 \geqslant 0$. The points (x_i, y_i, z_i) lie in a plane if and only if the matrix C is *not* positive definite, i.e. if $\det C = \lambda_1 \lambda_2 \lambda_2 = 0$.

Now,

$$\det C = (\sigma_{\mathrm{x}}^2 \sigma_{\mathrm{y}}^2 \sigma_{\mathrm{z}}^2 + 2 c_{\mathrm{xy}} c_{\mathrm{yz}} c_{\mathrm{zx}}) - (\sigma_{\mathrm{x}}^2 c_{\mathrm{yz}}^2 + \sigma_{\mathrm{y}}^2 c_{\mathrm{xz}}^2 + \sigma_{\mathrm{z}}^2 c_{\mathrm{xy}}^2) \geqslant 0$$

so

$$\frac{\sigma_{\mathrm{x}}^2 c_{\mathrm{yz}}^2 + \sigma_{\mathrm{y}}^2 c_{\mathrm{xz}}^2 + \sigma_{\mathrm{z}}^2 c_{\mathrm{xy}}^2}{\sigma_{\mathrm{x}}^2 \sigma_{\mathrm{y}}^2 \sigma_{\mathrm{z}}^2 + 2 c_{\mathrm{xy}} c_{\mathrm{yz}} c_{\mathrm{zx}}} \leqslant 1$$

with equality if and only if the points (x_i, y_i, z_i) are coplanar. Note that if the denominator is zero, so is the denominator, and this ensures that the points are coplanar or colinear anyway. Therefore, we define the statistic

$$p_{\mathrm{xyz}} = \begin{cases} 1 & \sigma_{\mathrm{x}}^2 \sigma_{\mathrm{y}}^2 \sigma_{\mathrm{z}}^2 + 2 c_{\mathrm{xy}} c_{\mathrm{yz}} c_{\mathrm{zx}} = 0 \\ \dfrac{\sigma_{\mathrm{x}}^2 c_{\mathrm{yz}}^2 + \sigma_{\mathrm{y}}^2 c_{\mathrm{xz}}^2 + \sigma_{\mathrm{z}}^2 c_{\mathrm{xy}}^2}{\sigma_{\mathrm{x}}^2 \sigma_{\mathrm{y}}^2 \sigma_{\mathrm{z}}^2 + 2 c_{\mathrm{xy}} c_{\mathrm{yz}} c_{\mathrm{zx}}} & \text{otherwise} \end{cases}$$

which we call **coplanarity** or **planar regression**.

Now lets take this new signal and plot it locally, with a radius $\epsilon = 1$.

In Figures 11 and 12, the local coplanarity of a tree is taken. As we would expect, there aren't many flat areas, so there is a lot of blue in the picture (recall that the bluer the point, the smaller the value of our signal at that point). One might expect that the ground would give higher values, but zooming in on a displaz image of the ground (Figure 15), we see that the ground is not actually flat.

In Figures 13 and 14, the local coplanarity of a house is taken. Most of the points are red, which represents values closer to 1, because the house is made of flat planar surfaces. We can even detect the corner of the house, because it shows up as having very small coplanarity – observe the blue band down the center of Figure 14.

Figure 11: A tree!

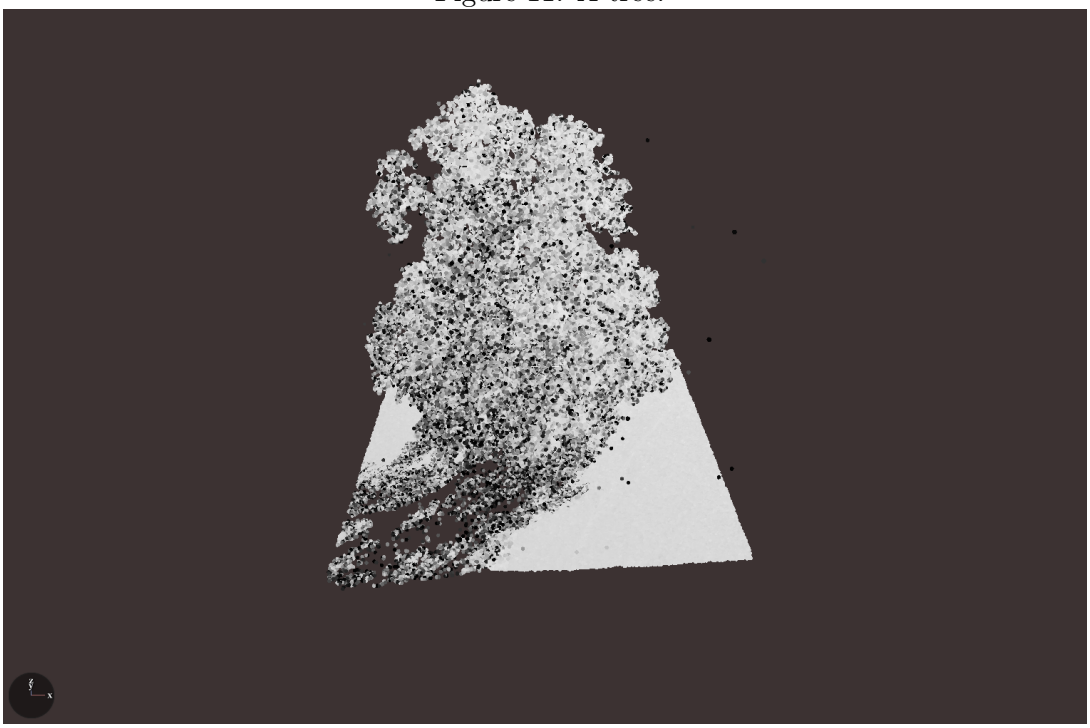


Figure 12: Local coplanarity of a tree.

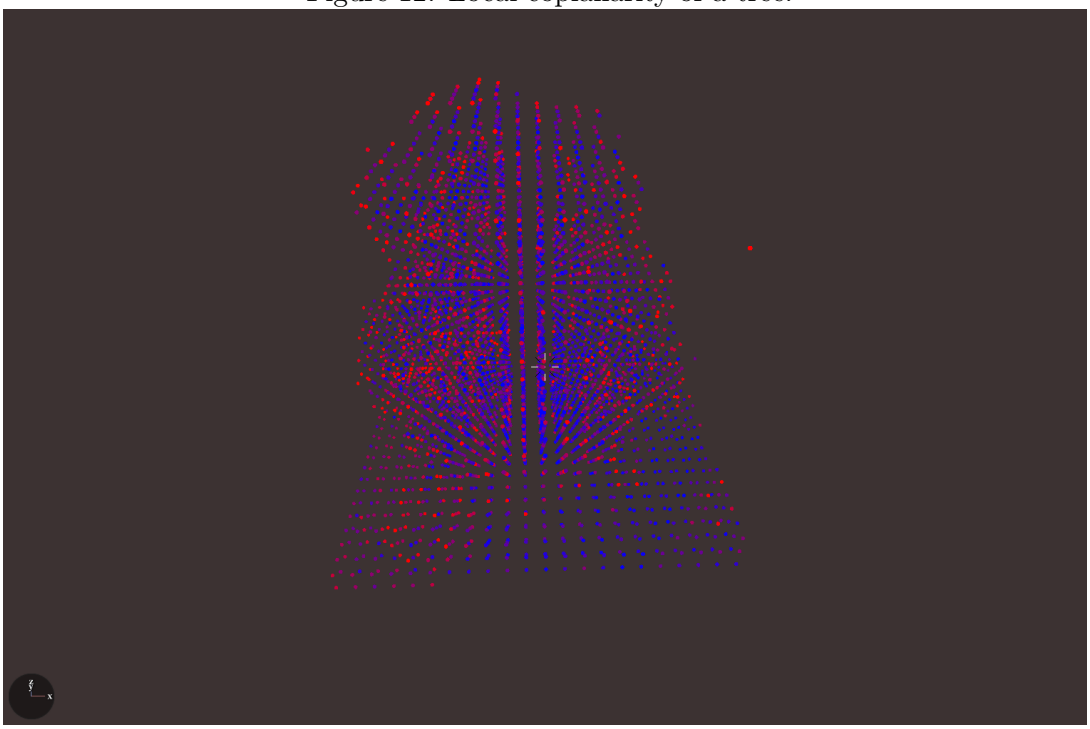


Figure 13: A house!

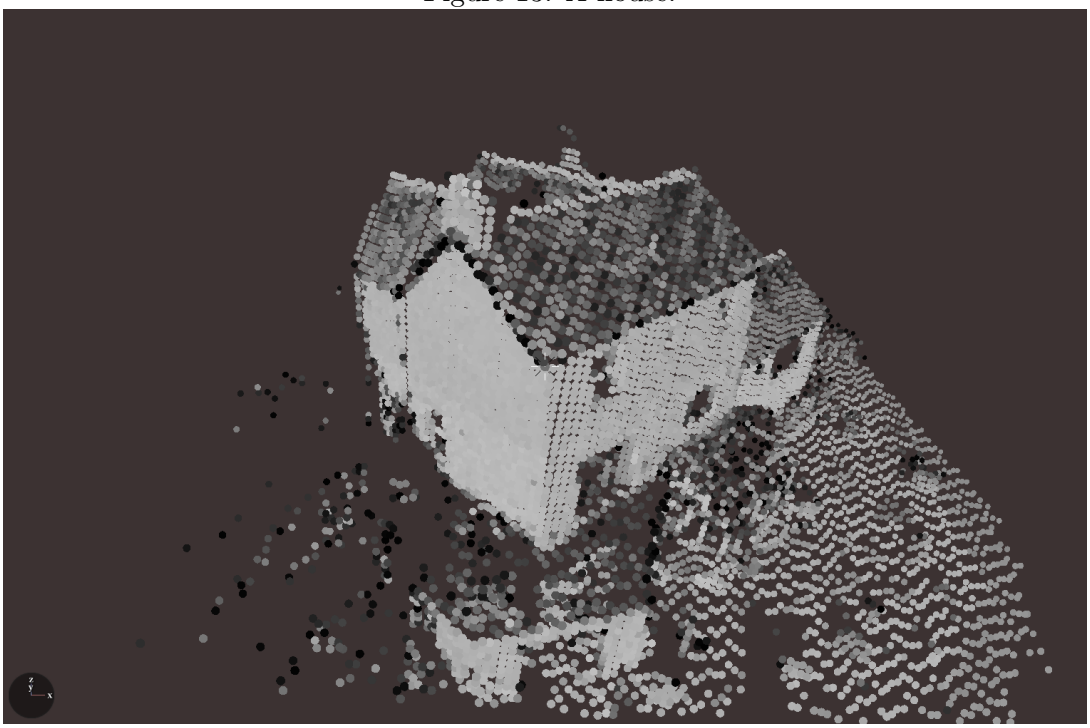


Figure 14: Coplanarity of a house – observe the “corner” is coloured blue

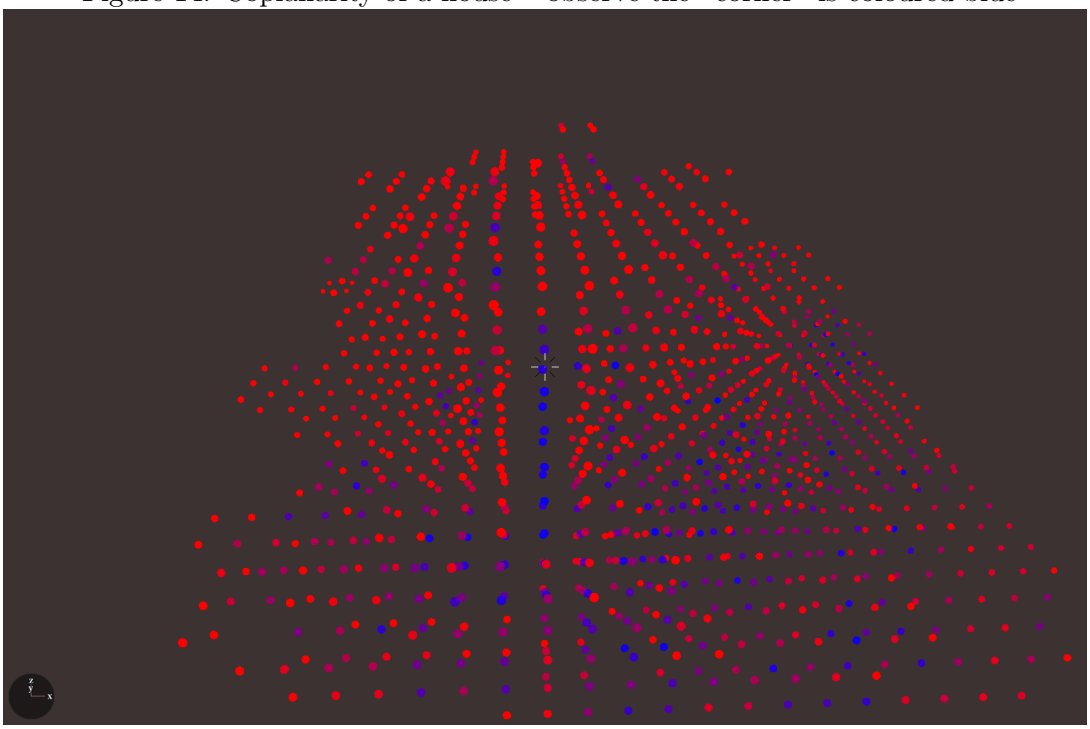
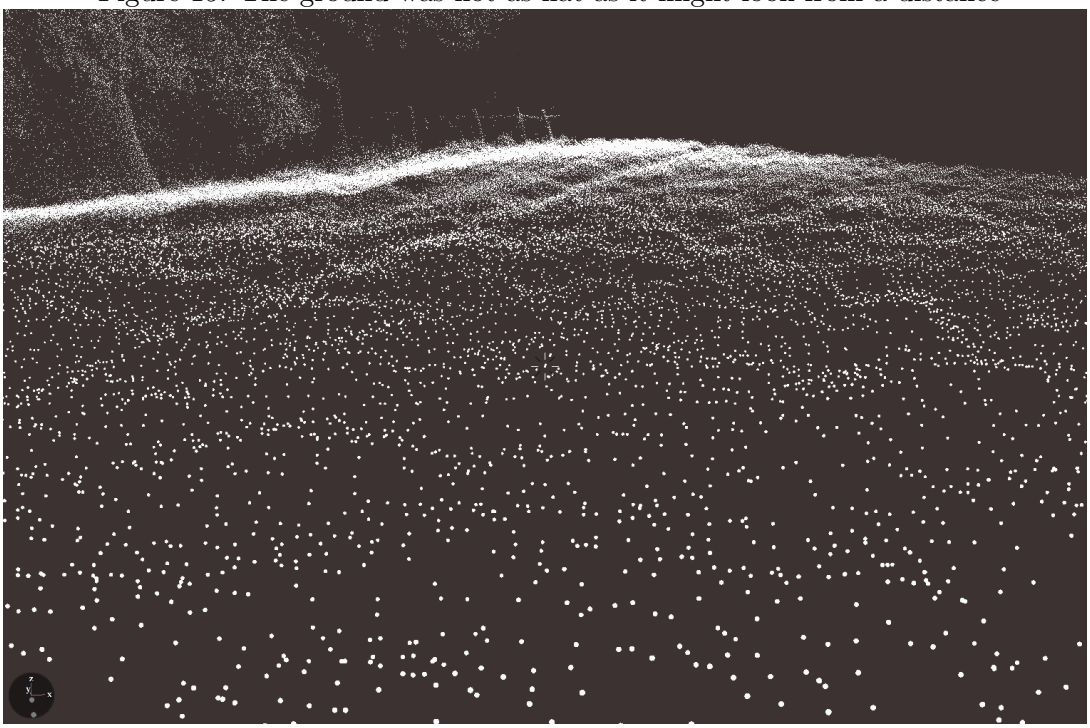


Figure 15: The ground was not as flat as it might look from a distance



1.7 Signals based on the center of intensity

Let points be given with position vectors $\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_n$ and corresponding intensities $I_1, I_2, \ldots, I_n$. Their **center of intensity** is

$$\sum_{i=1}^{n} I_i \mathbf{r}_i \Big/ \sum_{i=1}^{n} I_i$$

and their **center of mass** is

$$\sum_{i=1}^{n} \mathbf{r}_i \Big/ n.$$

When we calculate the center of intensity for lidar data, we must be careful. Indeed, we should calculate it as mean($I\mathbf{r}$)/mean(I), as sum($I\mathbf{r}$)/sum(I) is problematic.

For the test area given in Figure **??**, these two centers are extremely far apart, as shown below.

```
>>> import numpy, laspy
>>> inFile = laspy.file.File("RealDataClipped.las", mode = "r")
>>> xs = inFile.x
>>> ys = inFile.y
>>> zs = inFile.z
>>> CoM = numpy.array([mean(xs),mean(ys),mean(zs)])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'mean' is not defined
>>> CoM = numpy.array([numpy.mean(xs),numpy.mean(ys),numpy.mean(zs)])
>>> CoI = numpy.array([sum(xs*Is)/sum(Is),sum(ys*Is)/sum(Is),sum(zs*Is)
   /sum(Is)]
)
>>> CoM
array([3.85232195e+05, 2.07812059e+05, 8.10124502e+01])
>>> CoI = numpy.array([numpy.mean(Is*xs)/numpy.mean(Is), numpy.mean(Is*
   ys)/numpy.mean(Is), numpy.mean(Is*zs)/numpy.mean(Is)])
>>> numpy.sqrt(sum((CoI - CoM)**2))
6.988626965596175
```

2 Code

2.1 Code to generate artificial conductor

```
import numpy as np
import laspy
from math import cosh

a = 104.663 #a carefully chosen parameter to ensure we have three meter
    dip
l = 25 #half as long as we want the cable

#give an example las data for the header
in_file = laspy.file.File("TestArea.las", mode = "r")

#make a new one for our false conductor
output = laspy.file.File("FalseConductor.las", mode = "w", header =
   in_file.header)

#create some x values that are 5cm apart (not atypical for a conductor)
xs = [-0.05*n for n in range(int(l/0.05)+1)]+[0.05*n for n in range(int
   (l/0.05)+1)]
#make some artificial points and attributes
output.x = np.array(xs)
output.y = np.array([0*x for x in xs])
output.z = np.array([a*cosh(x/a)-a-3 for x in xs])
output.return_num = np.array([1]*len(xs))
output.num_returns = np.array([2]*len(xs))
output.raw_classification = np.array([1]*len(xs))
output.intensity = np.array([1000]*len(xs))

#close and save
output.close()
```

2.2 Master code

```
# master.py -- A MODULE OF ALL FUNCTIONS USED

# This module requires numpy and laspy to be installed
import numpy
import laspy

# RETURNS THE ABSOLUTE VALUE OF LOCAL COVARIANCE ABOUT TWO AXES
# 0 is x, 1 is y, 2 is z
# returns nan if chosen sphere is empty
def labscov(num_1,num_2,in_file,a,b,c,radius):
        indices_for_calc = (in_file.x-a)**2 + (in_file.y-b)**2 + (
           in_file.z-c)**2 < radius**2
        if not(numpy.isin(True, indices_for_calc)):
                return numpy.nan
        else:
                x_for_calc = in_file.x[indices_for_calc]
                y_for_calc = in_file.y[indices_for_calc]
                z_for_calc = in_file.z[indices_for_calc]
                coords_for_calc = numpy.vstack((x_for_calc,y_for_calc,
                   z_for_calc))
                cov_mat = numpy.cov(coords_for_calc)
                if len(x_for_calc) == 0:
                        return 0
                else:
                        return abs(cov_mat[num_1,num_2])

# FINDS OUT WHETHER A LOCAL NBHD OF A POINT CONTAINS A CHOSEN NUMBER OF
    PTS
#Returns true if (x,y,z) has more than N pts within a sphere of given
   radius
def nhbrs(N,in_file,a,b,c,radius):
        indices_for_calc = (in_file.x-a)**2 + (in_file.y-b)**2 + (
           in_file.z-c)**2 < radius**2
        pts = in_file.points
        pts_in_sphere = pts[indices_for_calc]
        if len(pts_in_sphere)>N:
                return True
        else:
                return False

# LOCAL STANDARD DEVIATION
# computes local standard deviation along a chosen axis
def lstd(num,in_file, a, b, c, radius):
        return numpy.sqrt(labscov(num,num,in_file,a,b,c,radius))

# COMPUTES LOCAL ABSOLUTE REGRATION (close to 1 if in a region of
   correlation, else close to 0)
def lreg(in_file, a, b, c, radius):
        axes = []
        for axis in range(2):
                if lstd(axis,in_file,a,b,c,radius) > 0:
                        axes = axes + [axis]
        if len(axes) == 0:
                return numpy.nan
```

```
        if len(axes) == 1:
                return 1
        if len(axes) == 2:
                return labscov(axes[0],axes[1],in_file,a,b,c,radius)/(
                    lstd(axes[0],in_file,a,b,c,radius)*lstd(axes[1],
                    in_file,a,b,c,radius))
        if len(axes) == 3:
                return labscov(0,1,in_file,a,b,c,radius)*labscov(1,2,
                    in_file,a,b,c,radius)*labscov(2,1,in_file,a,b,c,
                    radius)/(lstd(0,in_file,a,b,c,radius)*lstd(1,in_file
                    ,a,b,c,radius)*lstd(2,in_file,a,b,c,radius))


# THROW AWAY FLIGHT LINE AND KEEP ONE OF MANY RETURNS
# in_file is a LAS file loaded into laspy
# returns a file in read mode where flight line is taken away and only
    one of manys kept
# produces a las file along the way, with the chosen name
# name should not have .las extension
def one_of_many(in_file, name):
        off_flight  = in_file.classification != 10
        out_put_las = laspy.file.File(name+".las", mode = "w", header =
             in_file.header)
        one_of_manys = numpy.logical_and(in_file.return_num == 1,
            in_file.num_returns!=1)
        indices_kept = numpy.logical_and(off_flight,one_of_manys)
        out_put_las.points = in_file.points[indices_kept]
        out_put_las.close()
        out_put_to_be_read = laspy.file.File(name+".las", mode = "r")
        return out_put_to_be_read

#I NPUT A LAS FILE AND COLOUR PTS IN REGION OF CORRELATION
# in_file is the las file to be coloured as class 4
# the function simply creates a las file with those points coloured
# radius relates to the size of spheres used for local correlation
# name should be a string *without* .las extension
def crltd_pts(in_file, radius, name, corr_clip = 0.95):
        out_file = laspy.file.File(name+".las", mode = "w", header =
            in_file.header)
        points = in_file.points
        xs = in_file.x
        ys = in_file.y
        zs = in_file.z
        out_file.points = points
        N = len(points)
        for marker in range(N):
                condition_01 = abs(labscov(0,1,in_file,xs[marker],ys[
                    marker],zs[marker],radius))>=corr_clip*lstd(0,
                    in_file,xs[marker],ys[marker],zs[marker],radius)*
                    lstd(1,in_file,xs[marker],ys[marker],zs[marker],
                    radius)
                condition_12 = abs(labscov(1,2,in_file,xs[marker],ys[
                    marker],zs[marker],radius))>=corr_clip*lstd(1,
                    in_file,xs[marker],ys[marker],zs[marker],radius)*
                    lstd(2,in_file,xs[marker],ys[marker],zs[marker],
```

```
                    radius)
                condition_20 = abs(labscov(2,0,in_file,xs[marker],ys[
                    marker],zs[marker],radius))>=corr_clip*lstd(2,
                    in_file,xs[marker],ys[marker],zs[marker],radius)*
                    lstd(0,in_file,xs[marker],ys[marker],zs[marker],
                    radius)
                if condition_01 and condition_12 and condition_20 and
                    nhbrs(3,in_file,xs[marker],ys[marker],zs[marker],
                    radius): #disregards pts with fewer than 3
                    neighbours, as
                        out_file.raw_classification[marker]=4
                else:
                        pass
        out_file.close()

def three_mesh(L_1,L_2,L_3):
        for i in L_1:
                for j in L_2:
                        for k in L_3:
                                yield [i,j,k]

#OUTPUTS LOCAL CORRELATION PLOT FROM A LAS FILE PLOTTED AS A LAS FILE
# in_file is the las file to be taken as our input data set
# the function creates a las file which forms a lattice, with the local
    correlation at each lattice point shown
# the function also takes in a minimum detail, which is the resolution
   of our grid
# name should be a string *without* .las extension
def lcorrplot(in_file, radius, name):
        x_array = in_file.x
        y_array = in_file.y
        z_array = in_file.z
        max_x = max(x_array)
        max_y = max(y_array)
        max_z = max(z_array)
        min_x = min(x_array)
        min_y = min(y_array)
        min_z = min(z_array)
        N_x = int((max_x-min_x)/radius)+1
        N_y = int((max_y-min_y)/radius)+1
        N_z = int((max_z-min_z)/radius)+1
        N = N_x*N_y*N_z
        h_x = (max_x-min_x)/N_x
        h_y = (max_y-min_y)/N_y
        h_z = (max_z-min_z)/N_z
        lattice = three_mesh(range(N_x),range(N_y),range(N_z))
        xs = []
        ys = []
        zs = []
        for dex in lattice:
                xs.append(min_x+h_x*(dex[0]+0.5))
                ys.append(min_y+h_x*(dex[1]+0.5))
                zs.append(min_z+h_x*(dex[2]+0.5))
        out_file = laspy.file.File(name+".las", mode = "w", header =
            in_file.header)
```

```
        out_file.x = numpy.array(xs)
        out_file.y = numpy.array(ys)
        out_file.z = numpy.array(zs)
        out_file.return_num = numpy.array([1]*N)
        out_file.num_returns = numpy.array([2]*N)
        out_file.raw_classification = numpy.array([1]*N)
        for index in range(N):
                correlation = lreg(in_file, out_file.x[index], out_file
                   .y[index], out_file.z[index], radius)
                if numpy.isnan(correlation):
                        out_file.intensity[index]=0
                else:
                        out_file.intensity[index]=1000*correlation
        out_file.close()

# LOCAL STANDARD DEVIATION OF INTENSITY
# computes local standard deviation of intensity
def lstdint(in_file, a, b, c, radius):
        indices_for_calc = (in_file.x-a)**2 + (in_file.y-b)**2 + (
           in_file.z-c)**2 < radius**2
        if not(numpy.isin(True, indices_for_calc)):
                return numpy.nan
        else:
                intsty_for_calc = in_file.intensity[indices_for_calc]
                return numpy.std(intsty_for_calc)

# OUTPUTS LOCAL STANDARD DEVIATION OF INTENSITY PLOT FROM A LAS FILE
   PLOTTED AS A LAS FILE
# in_file is the las file to be taken as our input data set
# the function creates a las file which forms a lattice, with the local
    correlation at each lattice point shown
# the function also takes in a minimum detail, which is the resolution
   of our grid
# name should be a string *without* .las extension
# nans are set to zero -- afterall, our signals are meant to be
   detecting something!
def lstdintplot(in_file, radius, name):
        x_array = in_file.x
        y_array = in_file.y
        z_array = in_file.z
        max_x = max(x_array)
        max_y = max(y_array)
        max_z = max(z_array)
        min_x = min(x_array)
        min_y = min(y_array)
        min_z = min(z_array)
        N_x = int((max_x-min_x)/radius)+1
        N_y = int((max_y-min_y)/radius)+1
        N_z = int((max_z-min_z)/radius)+1
        N = N_x*N_y*N_z
        h_x = (max_x-min_x)/N_x
        h_y = (max_y-min_y)/N_y
        h_z = (max_z-min_z)/N_z
        lattice = three_mesh(range(N_x),range(N_y),range(N_z))
        xs = []
```

```
        ys = []
        zs = []
        for dex in lattice:
                xs.append(min_x+h_x*(dex[0]+0.5))
                ys.append(min_y+h_x*(dex[1]+0.5))
                zs.append(min_z+h_x*(dex[2]+0.5))
        out_file = laspy.file.File(name+".las", mode = "w", header =
           in_file.header)
        out_file.x = numpy.array(xs)
        out_file.y = numpy.array(ys)
        out_file.z = numpy.array(zs)
        out_file.return_num = numpy.array([1]*N)
        out_file.num_returns = numpy.array([2]*N)
        out_file.raw_classification = numpy.array([1]*N)
        for index in range(N):
                newintensity = lstdint(in_file, out_file.x[index],
                   out_file.y[index], out_file.z[index], radius)
                if numpy.isnan(newintensity):
                        out_file.intensity[index]=0
                else:
                        out_file.intensity[index]=newintensity
        out_file.close()

# LOCAL COVARIANCE
# not absolute value
def lcov(num_1,num_2,in_file,a,b,c,radius):
        indices_for_calc = (in_file.x-a)**2 + (in_file.y-b)**2 + (
           in_file.z-c)**2 < radius**2
        if not(numpy.isin(True, indices_for_calc)):
                return numpy.nan
        else:
                x_for_calc = in_file.x[indices_for_calc]
                y_for_calc = in_file.y[indices_for_calc]
                z_for_calc = in_file.z[indices_for_calc]
                coords_for_calc = numpy.vstack((x_for_calc,y_for_calc,
                   z_for_calc))
                cov_mat = numpy.cov(coords_for_calc)
                if len(x_for_calc) == 0:
                        return 0
                else:
                        return cov_mat[num_1,num_2]

# LOCAL COPLANARITY
# computes local coplanarity/planar regression
def lcoplan(in_file, a, b, c, radius):
        exp1 = (lstd(0,in_file,a,b,c,radius)*lstd(1,in_file,a,b,c,
           radius)*lstd(2,in_file,a,b,c,radius))**2+2*lcov(0,1,in_file,
           a,b,c,radius)*lcov(1,2,in_file,a,b,c,radius)*lcov(2,0,
           in_file,a,b,c,radius)
        exp2 = (lstd(0,in_file,a,b,c,radius)*lcov(1,2,in_file,a,b,c,
           radius))**2+(lstd(1,in_file,a,b,c,radius)*lcov(2,0,in_file,a
           ,b,c,radius))**2+(lstd(2,in_file,a,b,c,radius)*lcov(1,0,
           in_file,a,b,c,radius))**2
        if exp1 == 0:
                return 1
```

```
        else:
                return exp2/exp1

# OUTPUTS LOCAL COPLANARITY PLOT FROM A LAS FILE PLOTTED AS A LAS FILE
# in_file is the las file to be taken as our input data set
# the function creates a las file which forms a lattice, with the local
    correlation at each lattice point shown
# the function also takes in a minimum detail, which is the resolution
   of our grid
# name should be a string *without* .las extension
# nans are set to zero -- afterall, our signals are meant to be
   detecting something!

def lcoplanplot(in_file, radius, name):
        x_array = in_file.x
        y_array = in_file.y
        z_array = in_file.z
        max_x = max(x_array)
        max_y = max(y_array)
        max_z = max(z_array)
        min_x = min(x_array)
        min_y = min(y_array)
        min_z = min(z_array)
        N_x = int((max_x-min_x)/radius)+1
        N_y = int((max_y-min_y)/radius)+1
        N_z = int((max_z-min_z)/radius)+1
        N = N_x*N_y*N_z
        h_x = (max_x-min_x)/N_x
        h_y = (max_y-min_y)/N_y
        h_z = (max_z-min_z)/N_z
        lattice = three_mesh(range(N_x),range(N_y),range(N_z))
        xs = []
        ys = []
        zs = []
        for dex in lattice:
                xs.append(min_x+h_x*(dex[0]+0.5))
                ys.append(min_y+h_x*(dex[1]+0.5))
                zs.append(min_z+h_x*(dex[2]+0.5))
        out_file = laspy.file.File(name+".las", mode = "w", header =
           in_file.header)
        out_file.x = numpy.array(xs)
        out_file.y = numpy.array(ys)
        out_file.z = numpy.array(zs)
        out_file.return_num = numpy.array([1]*N)
        out_file.num_returns = numpy.array([2]*N)
        out_file.raw_classification = numpy.array([1]*N)
        for index in range(N):
                val = lcoplan(in_file, out_file.x[index], out_file.y[
                   index], out_file.z[index], radius)
                if numpy.isnan(val):
                        out_file.intensity[index]=0
                else:
                        out_file.intensity[index]=1000*val
        out_file.close()
```

```
def distbetweencenters(in_file, radius, name):
        x_array = in_file.x
        y_array = in_file.y
        z_array = in_file.z
        max_x = max(x_array)
        max_y = max(y_array)
        max_z = max(z_array)
        min_x = min(x_array)
        min_y = min(y_array)
        min_z = min(z_array)
        N_x = int((max_x-min_x)/radius)+1
        N_y = int((max_y-min_y)/radius)+1
        N_z = int((max_z-min_z)/radius)+1
        N = N_x*N_y*N_z
        h_x = (max_x-min_x)/N_x
        h_y = (max_y-min_y)/N_y
        h_z = (max_z-min_z)/N_z
        lattice = three_mesh(range(N_x),range(N_y),range(N_z))
        xs = []
        ys = []
        zs = []
        for dex in lattice:
                xs.append(min_x+h_x*(dex[0]+0.5))
                ys.append(min_y+h_x*(dex[1]+0.5))
                zs.append(min_z+h_x*(dex[2]+0.5))
        out_file = laspy.file.File(name+".las", mode = "w", header =
           in_file.header)
        out_file.x = numpy.array(xs)
        out_file.y = numpy.array(ys)
        out_file.z = numpy.array(zs)
        out_file.return_num = numpy.array([1]*N)
        out_file.num_returns = numpy.array([2]*N)
        out_file.raw_classification = numpy.array([1]*N)
        for index in range(N):
                a, b, c = (out_file.x[index], out_file.y[index],
                   out_file.z[index])
                indices_for_calc = (in_file.x-a)**2 + (in_file.y-b)**2
                   + (in_file.z-c)**2 < radius**2
                if True in indices_for_calc:
                        xs = in_file.x[indices_for_calc]
                        ys = in_file.y[indices_for_calc]
                        zs = in_file.z[indices_for_calc]
                        Intss = in_file.intensity[indices_for_calc]
                        Is = Intss.astype(float)
                        CoM = numpy.array([numpy.mean(xs),numpy.mean(ys
                           ).numpy.mean(zs)])
                        CoI = numpy.array([numpy.mean(xs*Is)/numpy.mean
                           (Is),numpy.mean(ys*Is)/numpy.mean(Is),numpy.
                           mean(zs*Is)/numpy.mean(Is)])
                        out_file.intensity[index]=1000*numpy.sqrt(numpy
                           .sum((CoM-CoI)**2))/radius
                else:
                        out_file.points[index]=0
        out_file.close()
```

2.3 Trimming data

Real lidar files are huge and look bizarre as we go further from the flight line. We can use a shell like this one to bound the flight line in a box and obtain a new las file which takes only points close enough to the box. See Figures **??** and **??** for a comparison.

Figure 16: A large data set

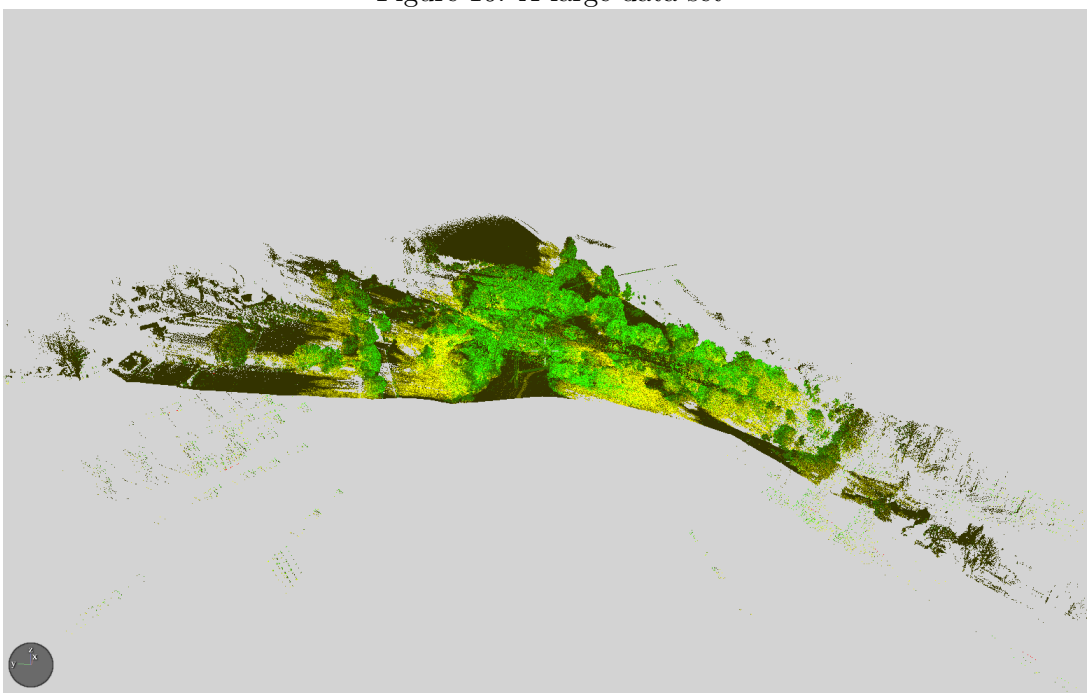
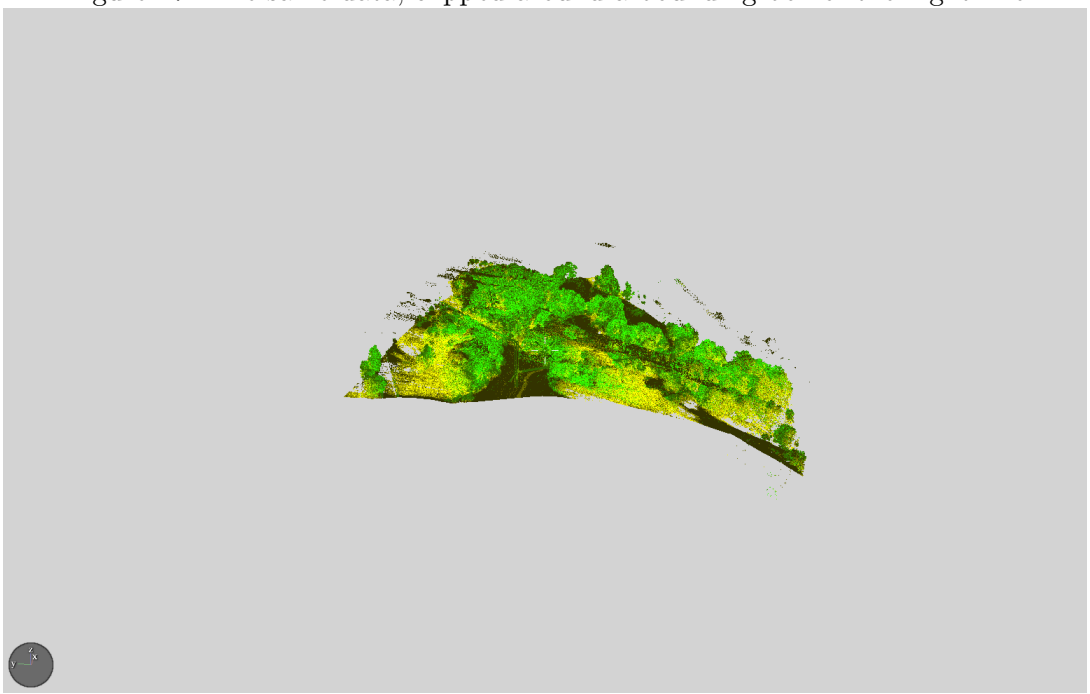


Figure 17: The same data, clipped around a bounding box of the flight line.



```
>>> inFile = laspy.file.File("RealData.las", mode = "r")
>>> class10 = inFile.classification==10
>>> ymax = max(inFile.y[class10])
>>> xmax = max(inFile.x[class10])
>>> zmax = max(inFile.z[class10])
>>> xmin = min(inFile.x[class10])
>>> ymin = min(inFile.y[class10])
>>> zmin = min(inFile.z[class10])
>>> xmax += 80
>>> ymax += 80
>>> zmax += 80
>>> xmin -= 80
>>> ymin -= 80
>>> zmin -= 80
>>> class1 = inFile.classification==1
>>> keepx = numpy.logical_and(inFile.x<xmax,inFile.x>xmin)
>>> keepy = numpy.logical_and(inFile.y<ymax,inFile.y>ymin)
>>> keepz = numpy.logical_and(inFile.z<zmax,inFile.z>zmin)
>>> keepxy = numpy.logical_and(keepx,keepy)
>>> keepz1 = numpy.logical_and(keepz, class1)
>>> keep = numpy.logical_and(keepxy,keepz1)
>>> outfile = laspy.file.File("RealDataClipped.las", mode = "w", header
    = inFile.header)
>>> outfile.points = inFile.points[keep]
>>> outfile.close()
```