Hochschule Bonn-Rhein-Sieg

Mathematics for Robotics and Control, SS17

Assignment 2 - Vectors, Matrices, Eigenvalues and Eigenvectors

This week's assignment is about matrices, particularly about their eigenvalues and eigenvectors, such that our robot will have the task of bringing a package to a conference room and placing it on a flat surface (e.g. a table).

Let us first setup this notebook so that figures and plots can be shown inside it.

In [2]:

```
try:
    shell = get_ipython()
    shell.enable_pylab("inline")
except NameError:
    pass

import numpy as np
import numpy.linalg as linalg
import matplotlib.pyplot as plt

from IPython.display import display
from IPython.core.pylabtools import figsize, getfigs
import IPython
```

Hint: Before you start solving the assignment, you might want to check the following *numpy* functions:

```
linalg.eig
linalg.eigh
argmax
argmin
argsort
cov
mean
where
einsum
```

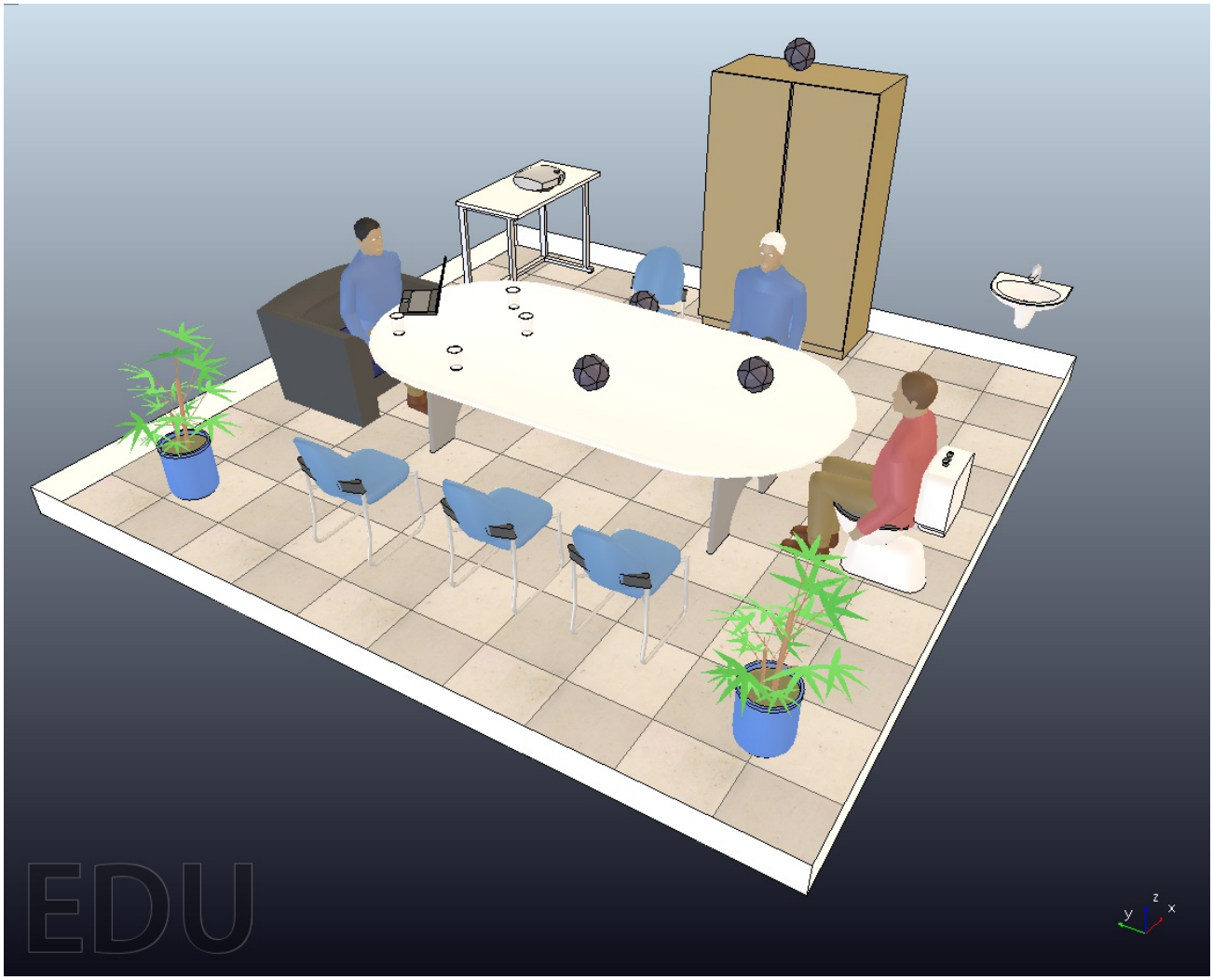
Placing a package [70 points]

Our robot has previously picked up a package and is now on its way to the office depicted below.

In [4]:

```
IPython.display.Image("images/isoview0001.png", width=800, embed=True)
```

Out[4]:



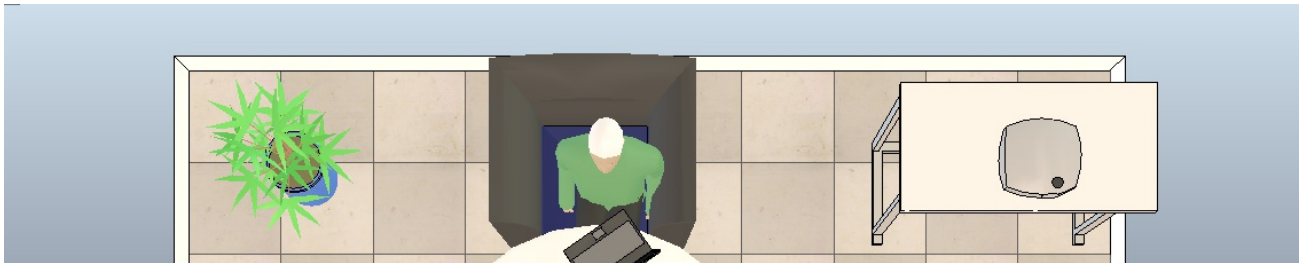
The offices in the building are equipped with sensors that help the robot obtain the required data for carrying out its tasks. The office to which the robot is going now has an RGB-D sensor - a Microsoft Kinect - mounted on the ceiling. RGB-D sensors can be used to obtain both a regular color image of a scene and what is called a depth image, in which each pixel encodes distances to the camera's sensor; these distances are typically given with respect to a reference frame whose origin is at the location of the camera's sensor.

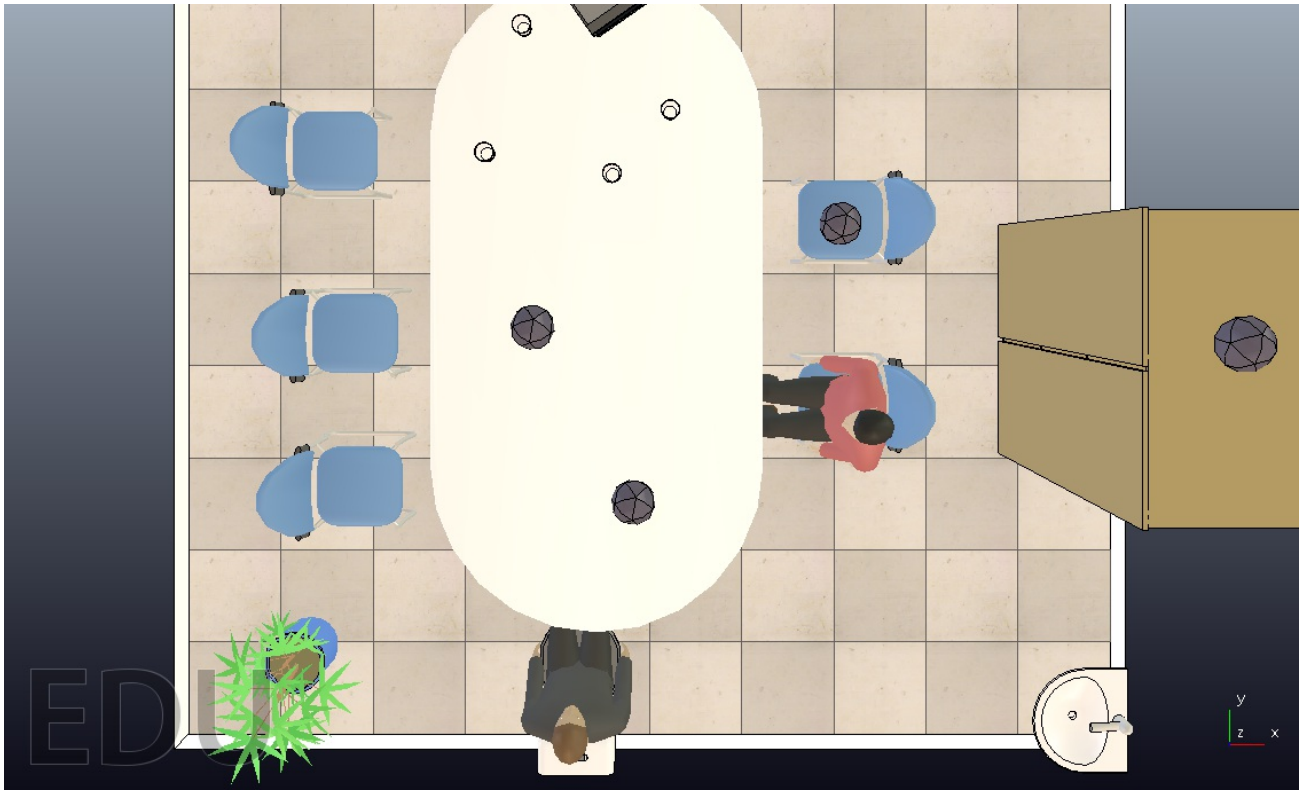
Both images obtained in the office are depicted below. Note that darker blue colors in the depth image represent pixels that are farther away from the sensor.

In [14]:

```
IPython.display.Image("images/rgb0001.png", width=800, embed=True)
```

Out[14]:

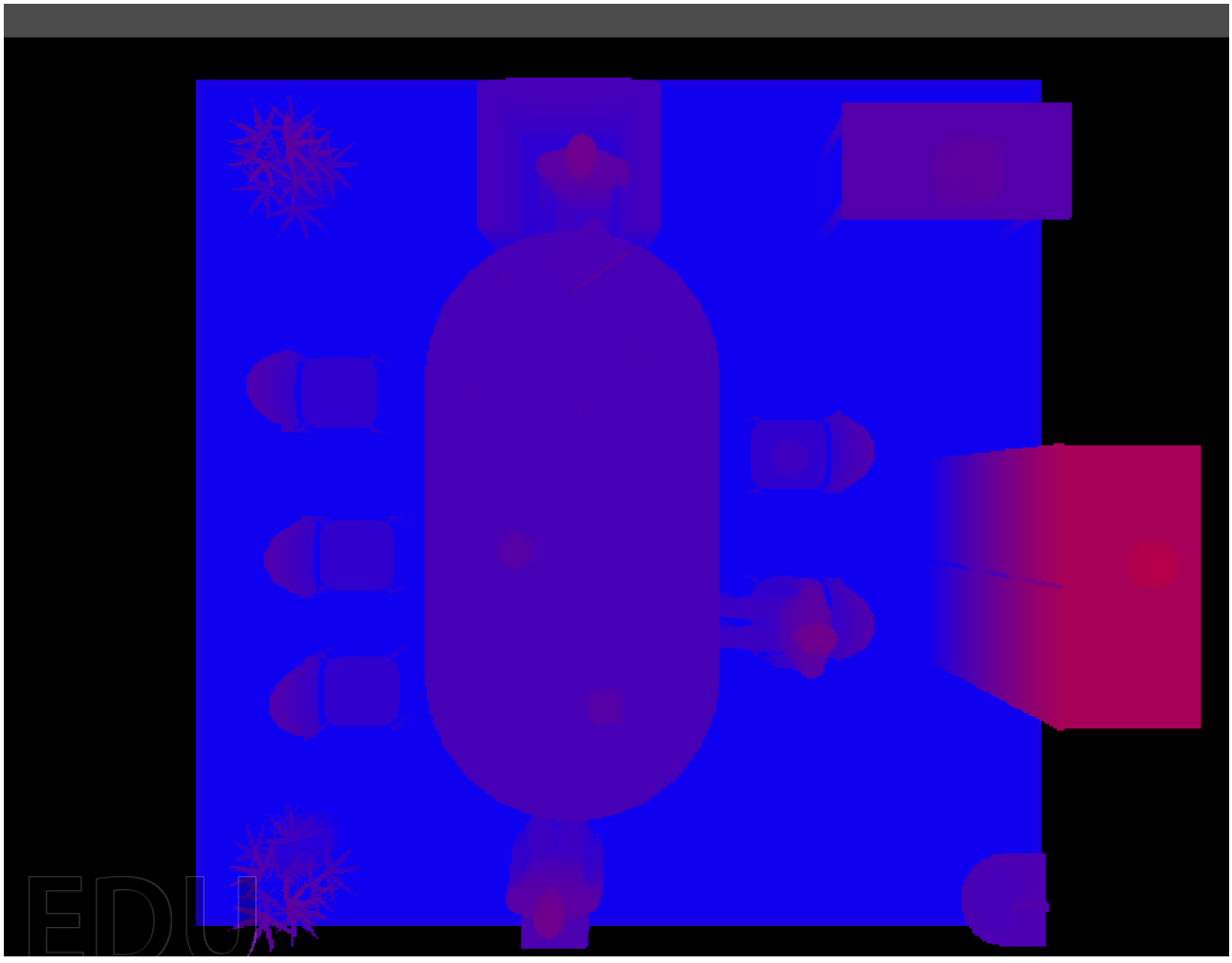




In [15]:

```
IPython.display.Image("images/depth0001.png", width=800, embed=True)
```

Out[15]:



Depth images can be converted to a different data format, called a *point cloud*. A point cloud is a

collection of 3-component tuples in which each component represents the coordinate of a point along a given axis. A formal definition of a point cloud P is given below:

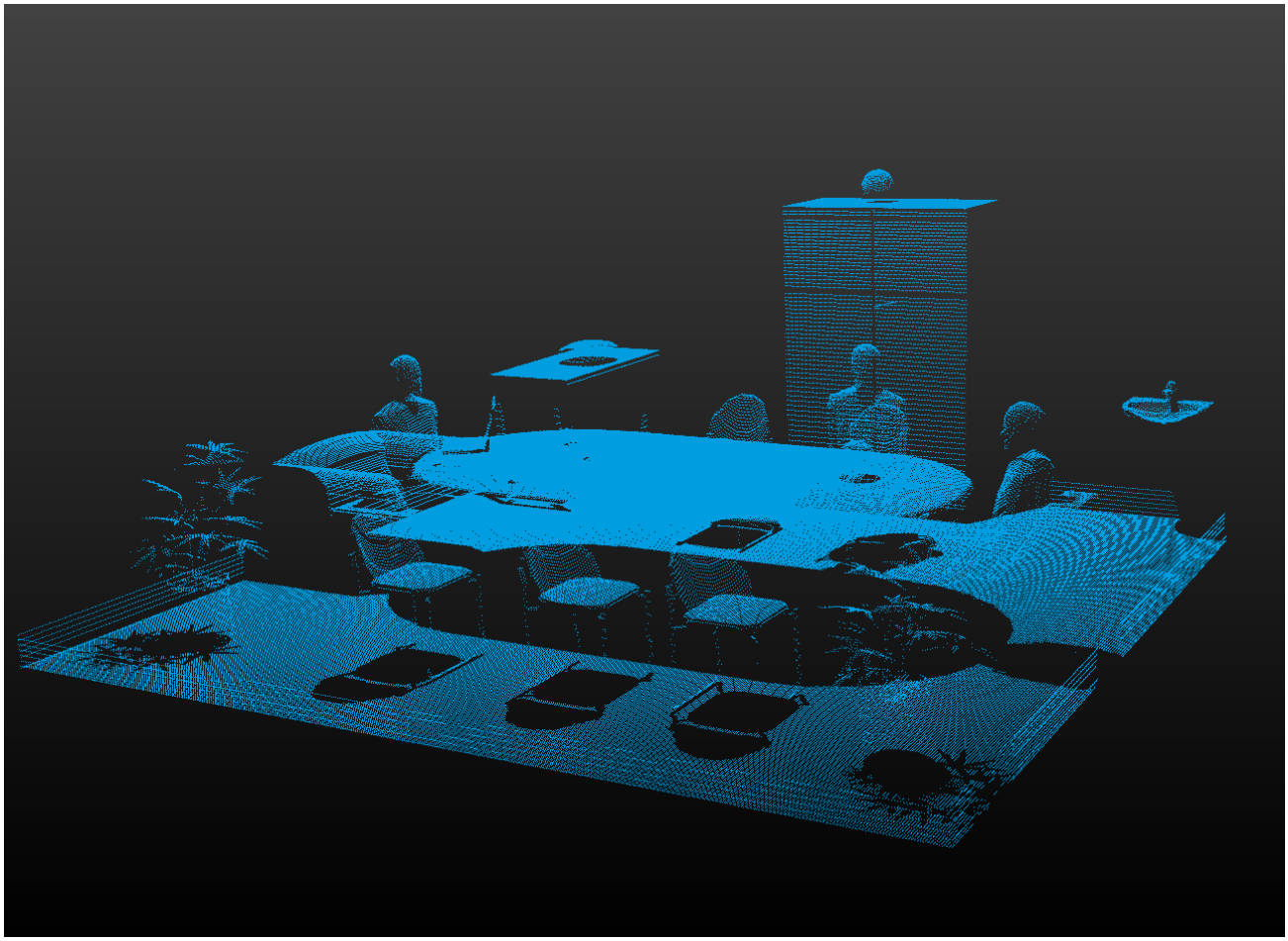
$$P = \{(x, y, z) \,|\, x, y, z \in \mathrm{R}\}$$

The picture below shows the aforementioned office scene as a point cloud obtained by the sensor mounted on the office's ceiling. Note that the unit of distance in the point cloud is **meters**.

In [3]:

```
IPython.display.Image("images/pointcloud0001.png", width=800, embed=True)
```

Out[3]:



If your browser supports WebGL and/or Adobe Flash, you can click this link for observing the point cloud in an interactive manner; otherwise, you can see the included *cloud.asc* or *cloud.ply* file in MeshLab, or the included *cloud.pcd* file using *pcl_viewer*, which should be included in your ROS installation.

Notice the object "shadows" in the image above. Given what you know about how the point cloud is generated, why do these shadows exist? What are these shadows really?

Now, back to our scenario. Our robot has already reached its destination and is now standing in front of the office door, after requesting sensor data from the central logistics unit and receiving the point cloud. **Your task is to identify possible drop points where the robot could place the package once inside the office**.

In order to achieve this goal, we need to evaluate each point with respect to whether dropping off a package there would make sense; for that purpose, we need to obtain a new point cloud in which

each point is associated with a quality measure. A simple quality measure would be the orientation of the normal vector of the underlying surface: given a point, if the surface to which the point belongs is not parallel to the floor, it would probably be unwise to place a package there. Note that normal vectors can be used for detecting tables, shelves and other furniture items commonly used for storing items.

Let us assume that our robot has knowledge of the world reference frame, i.e. it has an understanding of how the world is oriented; this is a reasonable assumption given a fixed robot construction and prior knowledge of the pose of the sensor used to obtain the point cloud data. In this case, we assume that the world frame is a subspace of R^3 and that the orthonormal basis of the coordinate space is $B = \left\{\begin{pmatrix}1 & 0 & 0\end{pmatrix}^T, \begin{pmatrix}0 & 1 & 0\end{pmatrix}^T, \begin{pmatrix}0 & 0 & 1\end{pmatrix}^T\right\}$.

To calculate the normal vectors of the points in the point cloud, we shall implement a method described in Surface reconstruction from unorganized points, a (rather old) publication by Hoppe et al. You don't have to study the complete paper for implementing this method, as all relevant information is contained in section **3.2 (Tangent Plane Estimation)** of it.

As you will notice by reading this section, we need to find point neighbourhoods (referred to as *Nbhd(x)* in the paper) for calculating the normal vectors. In order to obtain such neighborhoods, we shall use *scipy.spatial.cKDTree*, which is a spatial partitioning structure, and its method *query_ball_point*; we will use a radius of $4cm$ and the **L2 norm** as a distance measure when calling this method.

Note: For the purposes of this assignment, you don't have to understand how a k-d tree works; the only thing you should know about it is that it is a structure that can be used for speeding up spatial neighborhood queries. If you are curious, however, there are many resources on k-d trees available on the web, such as this collection of k-d tree visualisations.

After obtaining a normal vector of each point's neighborhood, we need to calculate the angle between this normal vector and the Z-axis. The reasoning behind this idea is that a surface has to be parallel to the floor for it to be level; given that every surface parallel to the floor will have a normal vector pointing in the same direction as the Z-axis, the angle between the normal vector of the surface and the Z-axis is a measure of how level the surface is and shows whether a point is suitable for releasing a package. While doing this step, we need to be careful about points that yield no usable neighborhoods, e.g. points that are too far away from the other points; such points should not be processed. You can pass any value for the resulting angles of these points (except for values that are close to zero, as we will then mistakenly consider the points as being part of a level surface), but do **not** omit them from the final result.

In [77]:

```
import scipy.spatial

def rate_placements(point_cloud):

    z_u = np.array([[0.0, 0.0,1.0]]).T           #coordinate of Z-axis (unit
length)
    tree =scipy.spatial.cKDTree(point_cloud)
    angles=[]                                    #Empty array to store angl

    for group in tree.query_ball_point(point_cloud, r=0.04):      #look for
nearest points within 0.04m

            cluster= tree.data[group]        #get the neighbour point coord
nates using the current index
```

```
nates using the current index
            if (np.shape(cluster)[0]>3):      #Checking if current point has
any usable neighbours (>3).

                cov = np.cov(cluster.T)       #Covariance of the current clus
ter of neighbour points
                eigenvalues, eigenvectors = np.linalg.eigh(cov)
                smallest_eigenvalue_idx = np.argsort(eigenvalues)[::1][0:1]

                W = eigenvectors[:,smallest_eigenvalue_idx]
eigen vector corresponding

igen vlaue

                w_u = (W/np.linalg.norm(W)).T
long the direction of the eigen vector

                """
                To get angle between normal and z axis, we can take cos inv
erse of the dot product
                of the two vectors.
                """

                theta=math.acos(np.clip(np.dot(w_u, z_u), -1.0, 1.0))

            else:
                """
                Setting angle as
any usable neighbour points
                i.e., for points with number of neighbours<=3
                """
                theta=np.pi/2

            angles.append(theta)     #appeding the angle to the array of ang
es.
    return angles


point_cloud = np.loadtxt('data/cloud.asc')
angles = rate_placements(point_cloud)

"""
Saving the array in a text file "angles.txt"
"""

numpy.savetxt('data/angles.txt', angles, fmt='%.18e')

print "Saved array to angles.txt"
```

Programming tips: argsort sorts the eigenvalues in increasing order, which is what you want; [::1] doesn't do anything to the resulting array, so you don't need that here. Also, you could simply use [0] instead of [0:1]

Your vectors are already unit vectors, so the dot product is going to be between -1 and 1 anyway; if they weren't, you would've had to divide the dot product by the product of their norms in order to get the angle (see how the dot product is defined)

```
Saved array to angles.txt
```

After obtaining the normal vectors of the points in the point cloud, assign a color to each point by mapping the angles to a linearly segmented color space ranging from green to red, where green means *perfectly parallel to the ground*. You can use the matplotlib.cm module for this task (see Matplotlib Cookbook: Show Colormaps Example for an example of using this module). Generally speaking, this step boils down to picking (or creating) a suitable colormap and applying the colormap to your angle values. Remember to convert the colors to a range of $[0, 255]$ before continuing with the next step.

next step.

The following cell shows a simple example code snippet on how to apply colormaps:

In [2]:

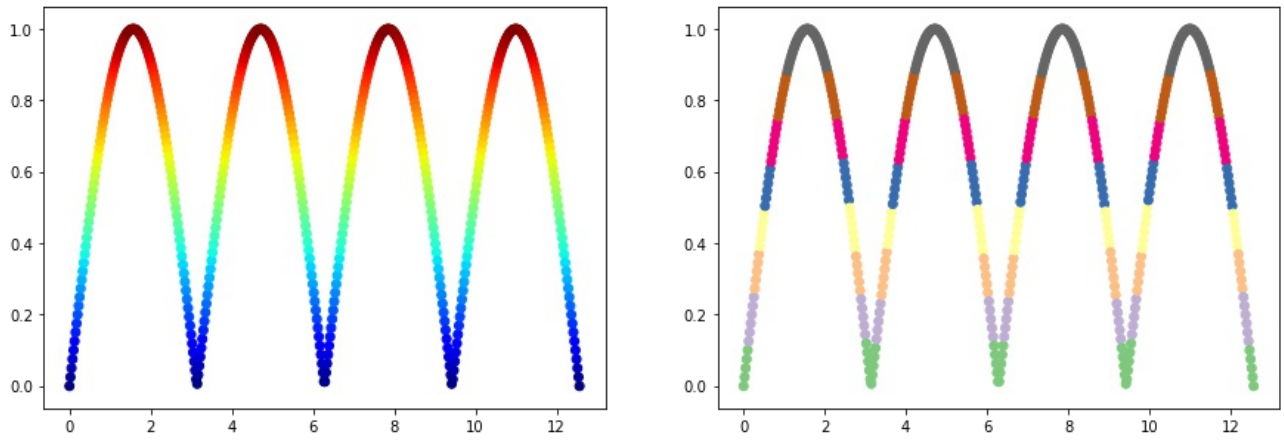
```
# colormap example
f = plt.figure(figsize=(15, 5))
x = np.linspace(0, 4*np.pi, 500)
y = np.abs(np.sin(x))

plt.subplot(1,2,1)
colors = plt.cm.jet(y)
plt.scatter(x, y, c=colors)

plt.subplot(1,2,2)
colors = plt.cm.Accent(y)
plt.scatter(x, y, c=colors)
```

Out[2]:

```
<matplotlib.collections.PathCollection at 0x7f5b6aaa7590>
```



In [78]:

```
### map the angles to a color space here

import matplotlib.colors as col

startcolor = '#006400'   # Green
endcolor = '#b22222'   # red
cmap = col.LinearSegmentedColormap.from_list('cloud_color_map',[startcolor,
endcolor])
```

After you have obtained color values for each point:

1. create a new $N \times M$ matrix, where N is the number of points, while $M == 6$ corresponds to the X, Y and Z coordinates of each point and its RGB color values (a tuple in the range $[0, 255]$), and
2. pass this matrix to the following function, which generates a file in the Stanford Triangle Format

In [80]:

```
def save_as_ply(filename, points):
    assert points.ndim == 2
```

```
    assert points.shape[1] == 6
    header = """ply
        format ascii 1.0
        element vertex {0}
        property float x
        property float y
        property float z
        property uchar red
        property uchar green
        property uchar blue
        element face 0
        property list uchar int vertex_indices\r\nend_header"""
    number_of_points = points.shape[0]
    with open(filename, "w") as ply_file:
        print >> ply_file, header.format(number_of_points)
        point_format = ("%.18f", "%.18f", "%.18f", "%d", "%d", "%d")
        np.savetxt(ply_file, points, delimiter=" ", fmt=point_format)

### your code here
points = point_cloud    #getting points from point cloud
color=[]

"""
We run a loop for the entire set of angles and map each angle to the
corresponding
color and points
"""
for i in range(len(angles)):

    """
    The angles within the range of (90,180] degrees are scaled to correspon
ding values between
    [0,90] degrees. This will result in a uniform gradient between green (l
evel surfaces) to
    red (perpendicular surfaces)
    """

    if(angles[i]<=np.pi and angles[i]>np.pi/2 ):
        angles_changed=np.pi-angles[i]
    else:
        angles_changed=angles[i]

    """
    In the below statement,
    1)The angle values are mapped to corresponding color in RGB: (cmap(angl
es_changed)[:3])
    2) This is then multiplied by 255 to bring the range from [0,1] to [0,2
55]
    3) This is then appended to the matrix "color".
    """
    color.append(np.multiply(255,(cmap(angles_changed)[:3])))

color=numpy.array(color)
final=np.hstack((points,color)) #finally, the color is stacked horizontally
with the point cloud data
save_as_ply('data/surfaces_ply.ply',final)
```

You can visualize your generated point cloud file using Meshlab or by using the command line utility **pcl_ply2pcd** and opening the resulting file in **pcl_viewer**. Both utilities come with an installation of

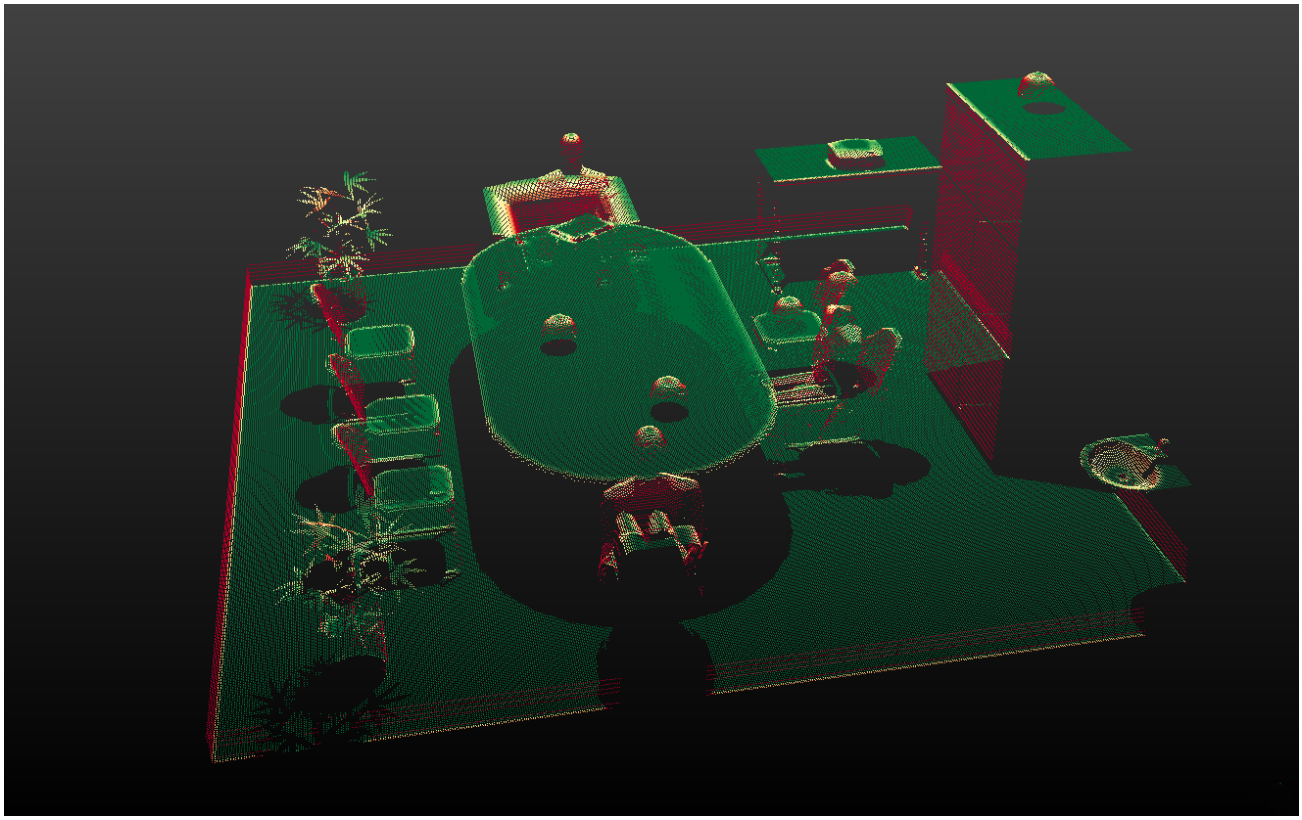
PCL and should be included in your ROS installation.

Your solution should looks similar to the one shown in the image below.

In [14]:

```
IPython.display.Image("images/result0001.png", width=800, embed=True)
```

Out[14]:



Your resulting image should've been included here

Condition number [10 points]

Exercise 1: 68/70 points

Consider the following set of equations:

$$4.5x_1 + 3.1x_2 = 19.249$$

$$1.6x_1 + 1.1x_2 = 6.843$$

1. Formulate the problem in the form $Ax = B$.
2. Determine whether the problem is ill-conditioned and if yes, make it well-conditioned.

In [12]:

```
A=np.array([[4.5,3.1],[1.6,1.1]])
B=np.array([[19.249,6.843]]).T
print "Matrix A:"
print A
print"``````````````````````````````````````````````````"
print "Matrix B:"
print B
print"``````````````````````````````````````````````````"
A_inv=np.linalg.inv(A)

X=A_inv.dot(B)
```

```
X=A_inv.dot(B)
print "X:"
print X
print"``````````````````````````````````````````"

print "Condition number:",
k= numpy.linalg.cond(A, p=inf)
print k
print "Since condition number is large, so the matrix is ill-condtioned"
print
"_____________________________________________________________________"
print
"_____________________________________________________________________"
print ""

print "We take a constant c and an identity matrix of the order of A in an
attempt to improve the condition of A"
print "we assume A'=A+cI, assuming c=2"


c=2 #assuming c as 2
I=np.array([[1,0],[0,1]])


A_1=A+c*I
print "Matrix A':"
print A_1
print"``````````````````````````````````````````"
"""
"""

k=numpy.linalg.cond(A_1, p=inf)
print "Condition number:",k
print ""

print "We can see that this improves the condtion number of the matrix
significantly"

print "Now, we need to change B so that the solution to the equations can
be maintained. So,"
print "B'=A`.(A_inverse).B"
print ""

B_1=A_1.dot(A_inv.dot(B))


print "Matrix B':"
print B_1
print"``````````````````````````````````````````"

print "X':"
A_inv_1=np.linalg.inv(A_1)
X_1=A_inv_1.dot(B_1)
print X_1
print"``````````````````````````````````````````"

print "Final system of equation:"
print A_1[0,0],"x1 + ", A_1[0,1], "x2 = ",B_1[0]
print A_1[1,0],"x1 + ", A_1[1,1], "x2 = ",B_1[1]
```

The thing about ill-conditioned matrices is that if you try to solve for x using the inverse, you may get a wrong solution. You are fully correct that we need to change the right side of the system if we change the matrix, but this is probably not the best way to do that.

```
Matrix A:
```

```
[[ 4.5  3.1]
 [ 1.6  1.1]]
```````````````````````````````````````````

```
Matrix B:
[[ 19.249]
 [  6.843]]
```````````````````````````````````````````

```
X:
[[ 3.94]
 [ 0.49]]
```````````````````````````````````````````

```
Condition number: 4636.0
Since condition number is large, so the matrix is ill-condtioned
```

```
We take a constant c and an identity matrix of the order of A in an attempt
to improve the condition of A
we assume A'=A+cI, assuming c=2
Matrix A':
[[ 6.5  3.1]
 [ 1.6  3.1]]
```````````````````````````````````````````

```
Condition number: 5.11915734036

We can see that this improves the condtion number of the matrix
significantly
Now, we need to change B so that the solution to the equations can be maint
ained. So,
B'=A`.(A_inverse).B

Matrix B':
[[ 27.129]
 [  7.823]]
```````````````````````````````````````````

```
X':
[[ 3.94]
 [ 0.49]]
```````````````````````````````````````````

```
Final system of equation:
6.5 x1 +  3.1 x2 =  [ 27.129]
1.6 x1 +  3.1 x2 =  [ 7.823]
```

Exercise 2: 10/10 points

Total: 78/80 points