[WS13/14] Mathematics for Robotics and Control: Assignment 002 - Matrices, Vectors, Eigenvalues and Eigenvectors

In [15]:
```
import IPython.core.display
import sys
if not "win" in sys.platform and not "linux" in sys.platform:
    %pylab
else:
    %pylab inline

'''
Worked with Stepan
'''
```

```
Welcome to pylab, a matplotlib-based Python environment [backend:
module://IPython.zmq.pylab.backend_inline].
For more information, type 'help(pylab)'.
```

Function and Module List

Each week, the assignment sheet will contain a list of Python modules and functions you are supposed to know. These modules and functions will help you to solve the assignments and many of them are required in later assignments. You are expected to familiarize yourself with each module / function listed here and to know when and how to apply them. Remember this is especially important in the exam!

Modules:

- glob
- matplotlib
- numpy
- numpy.linalg
- PIL.Image
- scipy.misc

Functions:

glob:

- glob, iglob

matplotlib:

- imshow

numpy:

- abs, argsort, cov, dot, empty_like, fliplr, fmax, fmin, loadtxt, min, max, nanmin, nanmax, sort, sum, sqrt, where

numpy.linalg:

- eig, eigh, eigvals, matrix_rank, norm

PIL.Image:

- open

scipy.misc:

- imresize, imfilter

Assignment 2.1 [L2]

Imagine you have programmed a robot to serve as an autonomous butler in a household environment. During the aftermath of a party, your

robot is given a verbal command to pick up all bottles from the living room. It navigates to the living room and starts looking around for empty bottles. For this exercise, we are assuming it was a rather civil (read: lame) party and everyone has been putting his or her bottle on a plain, even surface when they left at 10pm.

Now, how does your robot go about detecting all plane surfaces in the living room where a bottle could possibly be located? How does he distinguish between a bottle and a book, for example? This is where surface normals come in. If you're unable to recall your basic geometry knowledge about normals from school, go and read the linked Wikipedia article now, I'll wait.

Ok, having established that a normal is a vector perpendicular to a given plane (a surface), it becomes rather obvious how this helps in detecting possible bottle locations. If you have trouble seeing why this is the case, look at the image below and note the fancy colored tables. Each table has a normal vector pointing up, perpendicular to the surface of the table.

In [16]:
```
IPython.core.display.Image("images/livingroom.png")
```

Out[16]:

Given that the robot knows the world coordinate system and it's own orientation, it just needs to look at surfaces whose normal vectors are codirectional to the axis pointing up in the world coordinate system. Remember the direction cosine matrix from the last assignment? Why cosines? The answer is in the definition of the dot product. For two vectors $\mathbf{x}$ and $\mathbf{y}$, the dot product is defined as:

$$\mathbf{x} \cdot \mathbf{y} = ||\mathbf{x}|| \cdot ||\mathbf{y}|| \cdot cos(\theta)$$

...where θ is the angle between the two vectors. It is easy to see how the dot product can be used to determine if two vectors are codirectional or orthogonal, or how to get the angle between them. Now, remember the robot has access to the world coordinate system, so if the normal vectors of all surfaces of the living room would be available, it'd be easy to determine surfaces parallel to the floor (or the ceiling, for that matter) using the dot product.

So all that remains to find tables, shelves etc. is to obtain the surface normals. Your robot will most likely be equipped with a depth sensor, such as a laser scanner or a depth camera, e.g. the Microsoft Kinect. Both sensors can be used to acquire point clouds representing objects in the robot's perceived space. For an example of a point cloud, take a look at this. Now, a bunch of points does not immediately yield a plane and it's normal vector, but it is certainly possible to estimate the surface at any given point of a point cloud by considering the surrounding points. Remember that the normal of a surface at a point is identical to the surface of the tangent plane at that point (if you are wondering how you're supposed to "*remember*" that: it's in the Wikipedia article, really), so in order to obtain the surface normal, we can just fit a plane to a number of neighboring points and obtain the normal vector then. At this point in the lecture, we didn't talk about a number of things you need to know in order to do that, or even have an idea what "*fitting a plane to a set of points*" really means, but indulge me for now. If you want some visual clues to remember that "plane fitting" basically just means finding parameters of a plane in space such that it's orientation minimizes the distance to as many points as possible, and in the "best way possible" (under a given metric), take a look at these images: 1, 2, 3.

Anyway, for now, just accept the fact that if you....

I. Take a set of points $P = \left\{p | p = \in \mathbb{R}^3 \right\}, \left| \vert P \vert \right| = N and store them in a N \times 3 matrix A$
II. Create the covariance matrix of A, C_A
III. Obtain the eigenvectors and eigenvalues of C_A

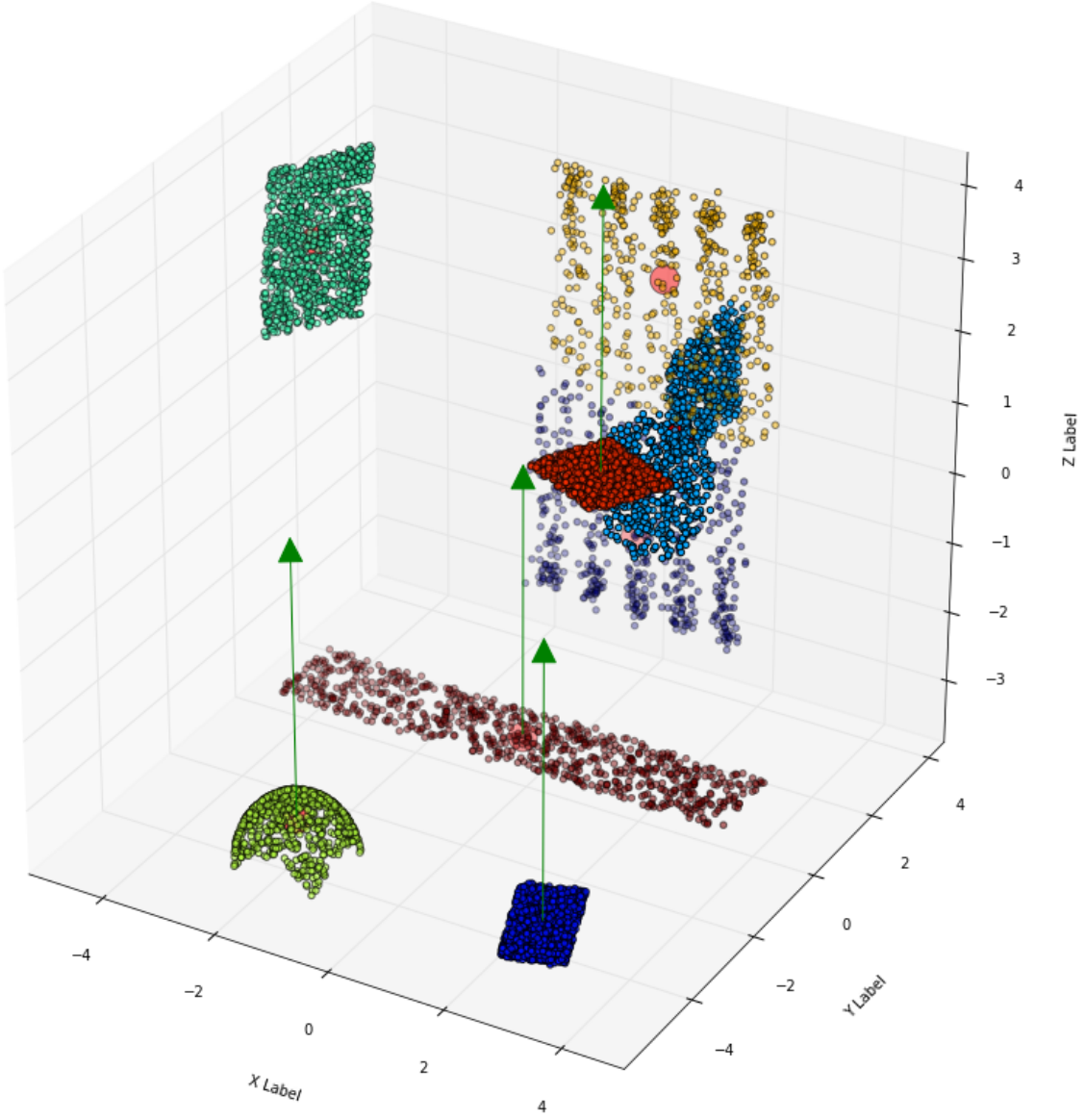
... the eigenvector with the smallest eigenvalue will be the normal vector of the plane fitted to P. If you want to know why that works and you need the math *right now*, this is a good starting point, but *reading that paper is not required*. We'll get back to that later. If the robot maintains an object database and knows the maximum height of a bottle, it can minimize the search space to the volume that is equal to a cube of that height, sitting on top of each flat surface. Bottles can then identified by their curved surface. See 4 for an example.

This assignment comes with a dataset of a simplified living room scene, named *point_cloud_001.npy*.

I. **Load the dataset and visualize it using matplotlib to get an idea of what the scene your robot sees looks like.**
II. **Device a simple(!) method for your robot to distinguish the different objects in that scene and write a Python function that, given the dataset, returns a number of sub-clouds, each representing a single object. The method does not need to be general, i.e. it's ok if it only works for this specific dataset and not all possible scenes. If you don't manage to do this, separate the objects manually and hardcode that information.**
III. **For each sub-cloud, create the covariance matrix C and obtain the eigenvalues and eigenvectors of C.**
IV. **Visualize the complete scene and draw the normal vector for each of the objects.**
V. **Write a function that identifies possible tables, shelves etc. and execute the function on the example dataset. Visualize your results.**

In [25]:
```
# Solution 2.1
# ...

## Note the magic offset of 1 here and there.

from sklearn.cluster import KMeans
import numpy as np
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from numpy import linalg as LA
#draw a vector
from matplotlib.patches import FancyArrowPatch
from mpl_toolkits.mplot3d import proj3d

class Arrow3D(FancyArrowPatch):
    '''
    This Arrow3D class was found on StackOverflow, to plot 3D vector Arrows
    http://stackoverflow.com/questions/11140163/
```

```
    '''
    def __init__(self, xs, ys, zs, *args, **kwargs):
        FancyArrowPatch.__init__(self, (0,0), (0,0), *args, **kwargs)
        self._verts3d = xs, ys, zs

    def draw(self, renderer):
        xs3d, ys3d, zs3d = self._verts3d
        xs, ys, zs = proj3d.proj_transform(xs3d, ys3d, zs3d, renderer.M)
        self.set_positions((xs[0],ys[0]),(xs[1],ys[1]))
        FancyArrowPatch.draw(self, renderer)

def possibleFlatSurface(norm):
    optimal_norm = np.array([0.0, 0.0, 1.0])
    angle = np.arccos(np.dot(norm, optimal_norm))
    if angle < 0.05:
        return True
    return False


# Load the Data
data = np.load('point_cloud_001.npy')

# The plot is by default a little hard to see, these fix that
ARROW_SCALE = 4
pylab.rcParams['figure.figsize'] = (15.0, 15.0)

# start a figure with 3D axes
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Using KMeans from SciKit Learn, cluster the data
km = KMeans().fit(data)

# get the center of the clusters and plot them as large red dots
xc, yc, zc = km.cluster_centers_.T
ax.scatter(xc, yc, zc, c='r', s=400, marker='o')

# append labels to the data
l_data = np.column_stack([data, km.labels_])
# sort the labeled data
sl_data = l_data[l_data[:,3].argsort()]

# plot the labeled data and color code by label
x, y, z, l = sl_data.T
ax.scatter(x,y,z, c=l.astype(np.float))

# find indices to split data, and group the data by labels
split_at = sl_data[:,3].searchsorted(np.unique(km.labels_)+1)
grouped_data = np.split(sl_data, split_at)

# for each group of data
for group in np.arange(len(grouped_data)-1):
    # create the covariance matrix C
    C = np.cov(grouped_data[group][:,0:3].T)
    # and obtain the eigenvalues and eigenvectors of C
    w, v = LA.eig(C)
    # the normal vector is eigenvector corresponding to lowest eigenvalue
    norm = v[np.argmin(w)]

    # If the norm indicates a possible flat surface show the normal vector
    if possibleFlatSurface(norm):
        # from the centre of each cluster, find endpoints of scaled normal vector
        i, j, k = km.cluster_centers_[group,:]
        xs = [i,i+norm[0]*ARROW_SCALE]
        ys = [j,j+norm[1]*ARROW_SCALE]
        zs = [k,k+norm[2]*ARROW_SCALE]

        # Plot the normal vectors using fancy function from stack overflow.
        a = Arrow3D(xs, ys, zs, mutation_scale=40, lw=1, arrowstyle="-|>", color="g")
        ax.add_artist(a)

#set the labels
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')
plt.show()
```



Assignment 2.1 took me *minutes.*

Assignment 2.2 L3

Eigenvectors have many applications which are not limited to obtaining surface normals from a set of point clouds. In this assignment, you are asked to write your own little facial recognition library. Take a look at the following image:

In [18]:
```
IPython.core.display.Image("images/ef0001.png")
```

Out[18]:

This is what is called an *eigenface*. An eigenface really is nothing else than an eigenvector, in this case reshaped for plotting. Eigenfaces can be used in facial recognition, allowing a robot to distinguish between different persons, but it can also be applied to other use cases such as voice or gesture recognition.

I. **Read the Wikipedia article about Eigenfaces.**
II. **Implement the eigenface algorithm described in the Wikipedia article in a separate Python module that exposes (at least) two functions:**
   A. **A function calculating eigenfaces given a number of images as parameter. Use img = numpy.array(PIL.Image.open(filename)) to load the images and resize every image to 58x69 (HxW) pixels.**
   B. **A function returning the most similar face, given a list of eigenfaces and a query face as parameters.**
III. **In this notebook, write a test case demonstrating your results by using your Python module to create the eigenface database of the images in the 'training' folder extracted from the facedb.zip file and displaying the most similar face for each of the faces in the 'test' folder in the facedb.zip file.**

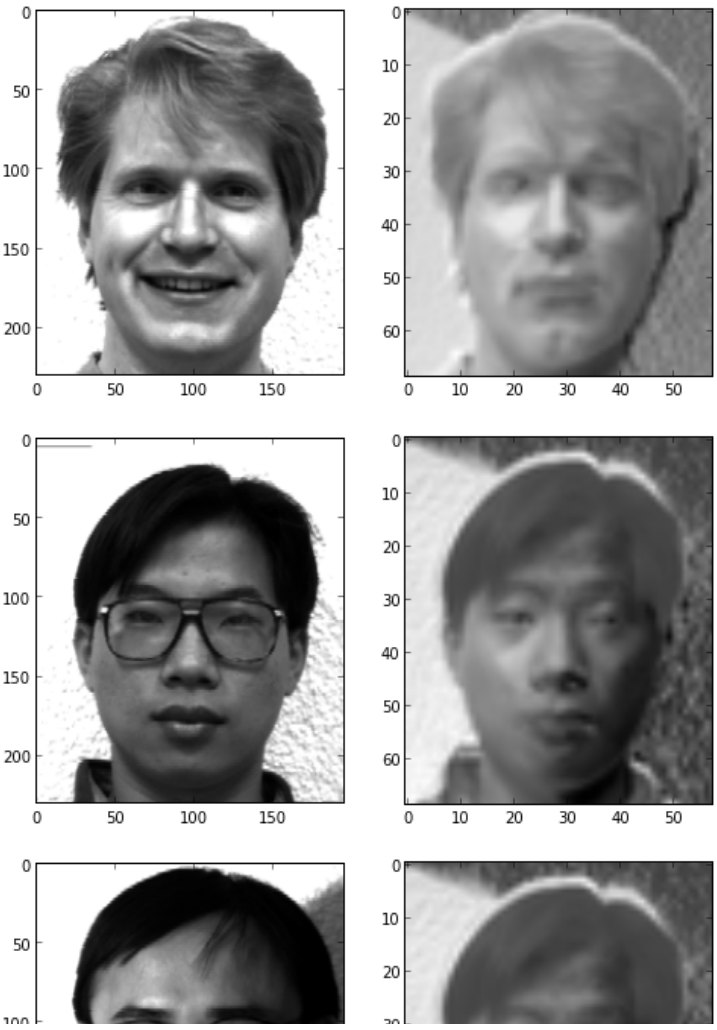
Use the function signatures shown below for your implementation.

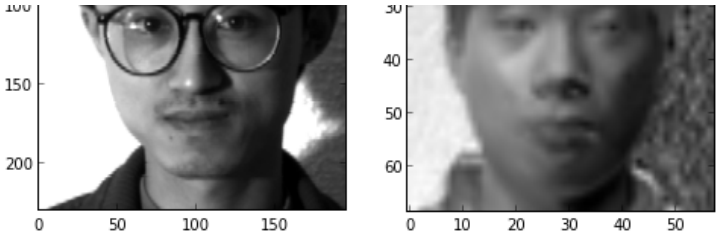
In [19]:
```
#def eigenfaces(image_filenames):
#    """ The eigenfaces function creates a set of eigenfaces from a list of grayscale images. """\
#    """ The eigenfaces are returned in a MxN matrix, where M = W*H of each single image, and N is the nu
#    raise NotImplementedError()

#def recognize_face(face, eigenfaces):
#    """ Given a query face as numpy.ndarray and a matrix of eigenfaces, returns the index of """\
#    """ the eigenface closest to the query face """
#    assert numpy.product(face.shape) == eigenfaces.shape[0]
#    raise NotImplementedError()

#from mrc_pkg.hw2 import face_detection as fd
import numpy

def eigenfaces(image_filenames):
    """ The eigenfaces function creates a set of eigenfaces from a list of grayscale images. """\
    """ The eigenfaces are returned in a MxN matrix, where M = W*H of each single image, and N is the numb
    # raise NotImplementedError()
    imgs = [PIL.Image.open(x) for x in image_filenames]
    new_width = 58
    new_height = 69
    ds_imgs = [x.resize([new_width, new_height], PIL.Image.ANTIALIAS) for x in imgs]
    sbjcts = [int(x.split("subject")[1][0:2]) for x in image_filenames]
    sbj_set = list(set(sbjcts))
    nSbjcts = shape(sbj_set)[0]
    cnt = collections.Counter(sbjcts)
    nImgs = shape(image_filenames)[0]
    ds_imgs_arr = numpy.zeros([nImgs, new_height, new_width])
    for x in arange(nImgs):
        ds_imgs_arr[x] = array(ds_imgs[x])
    inds = zeros([nSbjcts, 2], dtype=int)
    inds[0] = array([0, (cnt[sbj_set[0]]-1)])
    for x in range(1, nSbjcts):
        inds[x] = [inds[x-1, 1], inds[x-1, 1] + cnt[sbj_set[x]]]
    e_vecs = zeros([nImgs, new_height, new_width])
    for x in range(0, nSbjcts):
        tmp = eigenface(ds_imgs_arr[inds[x,0]:inds[x,1], ...])[...].T
        for y in xrange(tmp.shape[0]):
            e_vecs[inds[x,0]+y] = tmp[y].reshape([new_height, new_width])
    return e_vecs

def recognize_face(face, eigfaces):
    """ Given a query face as numpy.ndarray and a matrix of eigenfaces, returns the index of """\
    """ the eigenface closest to the query face """
    # assert numpy.product(face.shape) == eigenfaces.shape[0]
    # raise NotImplementedError()
    e_height = eigfaces.shape[1]
    e_width = eigfaces.shape[2]
    face = face.resize([e_width, e_height], PIL.Image.ANTIALIAS)
    face = array(face)
    dists = zeros([eigfaces.shape[0]])
    for i in arange(eigfaces.shape[0]):
        tmp = numpy.multiply(face - eigfaces[i], face - eigfaces[i])
        dists[i] = numpy.sqrt(numpy.sum(tmp))
    mini = dists.argmin()
    return eigfaces[mini]

def eigenface(images):
    nImgs = shape(images)[0]
    width = shape(images)[2]
    height = shape(images)[1]
    TM = zeros([nImgs, width*height])
    for j in range(0, nImgs):
        TM[j] = images[j].flatten()
    meanT = TM.mean(axis=0)
    for x in arange(nImgs):
        TM[x] -= meanT
    covT = cov(TM)
    eigVal, eigVec = eig(covT)
    eigVec = (TM.T).dot(eigVec)
    for i in arange(nImgs):
        eigVec[:,i] -= min(eigVec[:,i])
        eigVec[:,i] /= max(eigVec[:,i])
        eigVec[:,i] *= 255
    return eigVec
```

In [21]:
```
# Solution 2.2
```

```
# ...

# Solution 2.2
# ...

import PIL
import numpy
import glob
import collections
import matplotlib.pyplot as plt

pylab.rcParams['figure.figsize'] = (8.0, 8.0)

files = glob.glob("./YALEFaceDB/training/*.pgm")
evecs = eigenfaces(files)

test1 = PIL.Image.open("./YALEFaceDB/test/subject01.happy.pgm")
closest = recognize_face(test1, evecs)
fig = figure()
a = fig.add_subplot(1, 2, 1)
imshow(test1, cmap = 'gray')
a = fig.add_subplot(1,2,2)
imshow(closest, cmap='gray')

test2 = PIL.Image.open("./YALEFaceDB/test/subject04.glasses.pgm")
closest = recognize_face(test2, evecs)
fig = figure()
a = fig.add_subplot(1, 2, 1)
imshow(test2, cmap = 'gray')
a = fig.add_subplot(1,2,2)
imshow(closest, cmap='gray')

test3 = PIL.Image.open("./YALEFaceDB/test/subject08.leftlight.pgm")
closest = recognize_face(test3, evecs)
fig = figure()
a = fig.add_subplot(1, 2, 1)
imshow(test3, cmap = 'gray')
a = fig.add_subplot(1,2,2)
imshow(closest, cmap='gray')
```

Out[21]: AxesImage(350.545,80;225.455x496)





Assignment 2.2 took me *minutes.*

Use this button to create a .txt file containing the time in minutes you spent working on the assignments. Make sure to include your name in the textbox below. The file will be created in the current directory.

Student's name:

Save timings