1. How much is too much?

a) Degree 2 fits the curve well (visually). Yes, noise is influencing higher degree polynomials.

b) Yes, we want to choose polynomial of degree > 1 if the cost is lower than the polynomial of degree 1. It's hard to choose degree just based on this graph because lower costs doesn't mean the model will perform well in validation. Higher degrees minimize the cost, but overfit the model.

2. OMP Exercise

$$\begin{bmatrix} 1 & 1 & 0 & -1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \approx \begin{bmatrix} 4 \\ 6 \\ 3 \end{bmatrix}$$

$\vec{c}_1\ \vec{c}_2\ \vec{c}_3\ \vec{c}_4\ \ \vec{x} \approx \vec{b}$

$\langle \vec{c}_1, \vec{b} \rangle = 10$

$\langle \vec{c}_2, \vec{b} \rangle = 7$

$\langle \vec{c}_3, \vec{b} \rangle = -3$

$\langle \vec{c}_4, \vec{b} \rangle = -1$

$\vec{c}_1$ is largest $\Rightarrow$ $\vec{e} = \vec{b} - \text{proj}_{\vec{c}_1}(\vec{b}) = \vec{b} - \frac{10}{2}\begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$

$$\vec{e} = \begin{bmatrix} 4 \\ 6 \\ 3 \end{bmatrix} - \begin{bmatrix} 5 \\ 5 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \\ 3 \end{bmatrix}$$

$\langle \vec{c}_1, \vec{e} \rangle = 0$

$\langle \vec{c}_2, \vec{e} \rangle = 2$

$\langle \vec{c}_3, \vec{e} \rangle = 2$

$\langle \vec{c}_4, \vec{e} \rangle = 4$

$\vec{c}_4$ is largest, so x_1 and x_4 are non-zero. x_2 and x_3 are zero.

$$\begin{bmatrix} 1 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_4 \end{bmatrix} \approx \begin{bmatrix} 4 \\ 6 \\ 3 \end{bmatrix}$$

$$\vec{x} = (A^T A)^{-1} A^T \vec{b} = \left(\begin{bmatrix} 1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \right)^{-1} \begin{bmatrix} 1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 \\ 6 \\ 3 \end{bmatrix} =$$

$$\vec{x} = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}^{-1} \begin{bmatrix} 10 \\ -1 \end{bmatrix} = \left[\begin{array}{cc|cc} 2 & -1 & 1 & 0 \\ -1 & 2 & 0 & 1 \end{array}\right] \begin{bmatrix} 10 \\ -1 \end{bmatrix} = \left[\begin{array}{cc|cc} -1 & 2 & 0 & 1 \\ 0 & 3 & 1 & 2 \end{array}\right] \begin{bmatrix} 10 \\ -1 \end{bmatrix} = \left[\begin{array}{cc|cc} -1 & 2 & 0 & 1 \\ 0 & 1 & \frac{1}{3} & \frac{2}{3} \end{array}\right] \begin{bmatrix} 10 \\ -1 \end{bmatrix}$$

$$\vec{x} = \left[\begin{array}{cc|cc} 1 & -2 & 0 & -1 \\ 0 & 1 & \frac{1}{3} & \frac{2}{3} \end{array}\right] \begin{bmatrix} 10 \\ -1 \end{bmatrix} = \left[\begin{array}{cc|cc} 1 & 0 & \frac{2}{3} & \frac{1}{3} \\ 0 & 1 & \frac{1}{3} & \frac{2}{3} \end{array}\right] \begin{bmatrix} 10 \\ -1 \end{bmatrix} = \begin{bmatrix} \frac{20}{3} - \frac{1}{3} \\ \frac{10}{3} - \frac{2}{3} \end{bmatrix} = \begin{bmatrix} \frac{19}{3} \\ \frac{8}{3} \end{bmatrix}$$

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} \frac{19}{3} \\ 0 \\ 0 \\ \frac{8}{3} \end{bmatrix}$$

4. Sparse Imaging

b) The image is Cal logo.

5. Trolls Revisited

a) No

b) $\vec{r}_1 = \alpha \vec{\ell}_1 + \vec{n}$

$\alpha = \frac{\langle \vec{r}_1, \vec{\ell}_1 \rangle}{\|\vec{\ell}_1\|^2}$ and $\vec{n} = \vec{r} - \alpha \vec{\ell}_1$

c) I do not successfully recover the lecture. It's still too noisy.

d) It works. The lecture: Well, A^TA is a very nice square matrix, but does that mean we can always invert it? No, right? A^TA might not be invertible and how do we get around that?

7. Homework Process

I worked on this alone. I watched the lectures and read the notes and so I was able to do it.

3. Greedy Algorithm for Calculating Matrix Eigenvalues

a) $Q = A^T A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 10 & 14 \\ 14 & 20 \end{bmatrix}$

$Q^T = \begin{bmatrix} 10 & 14 \\ 14 & 20 \end{bmatrix} = Q$

let $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$, so $A^T = \begin{bmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \end{bmatrix}$

$Q = A^T A = \begin{bmatrix} a_{11}^2 + a_{21}^2 & a_{11}a_{12} + a_{21}a_{22} \\ a_{12}a_{11} + a_{22}a_{21} & a_{12}^2 + a_{22}^2 \end{bmatrix}$

$Q^T = Q$, so <u>Q is symmetric</u> for any $A \in R^{2\times 2}$

b) $V^T V = \begin{bmatrix} \vec{v}_1 \\ \vec{v}_2 \\ \vdots \\ \vec{v}_N \end{bmatrix}\begin{bmatrix} \vec{v}_1 & \vec{v}_2 & \dots & \vec{v}_N \end{bmatrix} = \begin{bmatrix} \langle \vec{v}_1, \vec{v}_1 \rangle & \langle \vec{v}_1, \vec{v}_2 \rangle & \dots & \langle \vec{v}_1, \vec{v}_N \rangle \\ \langle \vec{v}_1, \vec{v}_2 \rangle & \langle \vec{v}_2, \vec{v}_2 \rangle & \dots & \langle \vec{v}_2, \vec{v}_N \rangle \\ \vdots & \vdots & & \vdots \\ \langle \vec{v}_1, \vec{v}_N \rangle & \langle \vec{v}_2, \vec{v}_N \rangle & \dots & \langle \vec{v}_N, \vec{v}_N \rangle \end{bmatrix}$

$V^T V = \begin{bmatrix} \|\vec{v}_1\|^2 & 0 & \dots & 0 \\ 0 & \|\vec{v}_2\|^2 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & \|\vec{v}_N\| \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix} = I$

c) Given that $\langle \vec{v}_i, \vec{v}_j \rangle = 0$ for $0 \le i,j \le N$ and $i \ne j$
We then know that there are N orthogonal (linearly independent) vectors in R^N.
So, $\{\vec{v}_1, \vec{v}_2, \dots, \vec{v}_N\}$ form a basis for R^N

d) $\langle \vec{v}_i, \vec{b} \rangle = \langle \vec{v}_i, \alpha_1\vec{v}_1 + \dots + \alpha_N\vec{v}_N \rangle$

$= \alpha_1 \langle \vec{v}_i, \vec{v}_1 \rangle + \dots + \alpha_N \langle \vec{v}_i, \vec{v}_N \rangle$

$= 0 + \dots + \alpha_i \langle \vec{v}_i, \vec{v}_i \rangle + \dots + 0$

$= \alpha_i \langle \vec{v}_i, \vec{v}_i \rangle$

$\boxed{\langle \vec{v}_i, \vec{b} \rangle = \alpha_i \|\vec{v}_i\|^2}$

e) $\hat{\vec{x}} = (V^TV)^{-1}V^T\vec{b}$

$\|\vec{e}\| = \|V\vec{x} - \vec{b}\| = \|VV^{-1}V^{T^{-1}}V^T\vec{b} - \vec{b}\| = \|II\vec{b} - \vec{b}\| = \boxed{0}$

f) $V_2 = \begin{bmatrix} \vec{V}_2 & \dots & \vec{V}_N \end{bmatrix}$

$\hat{\vec{x}} = (V_2^TV)^{-1} V_2^T\vec{b}$

$\|\vec{e}\| = \|V_2\vec{x} - \vec{b}\| = \|V_2(V_2^TV_2)^{-1}V_2^T\vec{b} - \vec{b}\|$

We cannot simplify because V_2^{-1} doesn't exist

g) 1) $V^T\vec{V}_1 = \begin{bmatrix} \vec{V}_1 \\ \vdots \\ \vec{V}_N \end{bmatrix} \vec{V}_1 = \begin{bmatrix} \|\vec{V}_1\|^2 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ \vdots \\ 0 \end{bmatrix}$

2) $\Lambda V^T\vec{V}_1 = \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & \lambda_N \end{bmatrix} \begin{bmatrix} 1 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} \lambda_1 \\ \vdots \\ 0 \end{bmatrix}$

3) $V\begin{bmatrix} \lambda_1 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} \uparrow & \uparrow & & \uparrow \\ \vec{V}_1 & \vec{V}_2 & \dots & \vec{V}_N \\ \downarrow & \downarrow & & \downarrow \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \vdots \\ 0 \end{bmatrix} = \boxed{\vec{V}_1 \lambda_1}$

EECS16A Homework 14

Question 1: How Much Is Too Much?

Some Setup Code

You do not need to understand how the following code works.

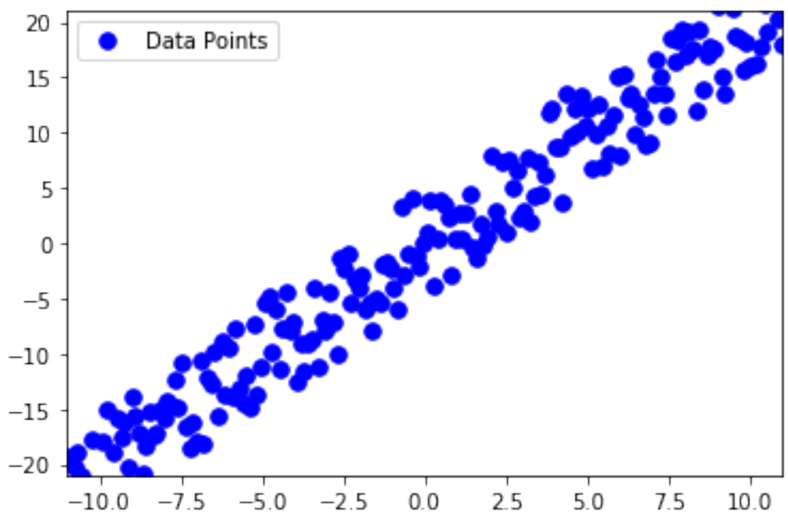
In [1]:
```
import numpy as np
import numpy.matlib
import matplotlib.pyplot as plt

%matplotlib inline

"""Function that constructs a polynomial curve for a set of
coefficients that multiply the polynomial terms and the x range."""
def poly_curve(params,x_input):
    # params contains the coefficients that multiply the polynomial te
    degree=len(params)-1
    x_range=[x_input[1], x_input[-1]]
    x=np.linspace(x_range[0],x_range[1],1000)
    y=x*0

    for k in range(0,degree+1):
        coeff=params[k]
        y=y+list(map(lambda z:coeff*z**k,x))
    return x,y

"""Function that defines a data matrix for some input data."""
def data_matrix(input_data,degree):
    # degree is the degree of the polynomial you plan to fit the data
    Data=np.zeros((len(input_data),degree+1))

    for k in range(0,degree+1):
        Data[:,k]=(list(map(lambda x:x**k ,input_data)))

    return Data

"""Function that computes the Least Squares Approximation"""
def leastSquares(D,y):
    return np.linalg.lstsq(D,y)[0]

np.random.seed(10)
```

Part a)

Some setup code to create our resistor test data points and plot them.

In [2]:
```
R = 2
x_a = np.linspace(-11,11,200)
y_a = R*x_a + (np.random.rand(len(x_a))-0.5)*10
fig = plt.figure()
ax=fig.add_subplot(111,xlim=[-11,11],ylim=[-21,21])
ax.plot(x_a,y_a, '.b', markersize=15)
ax.legend(['Data Points'])
```

Out[2]:
```
<matplotlib.legend.Legend at 0x113941a90>
```

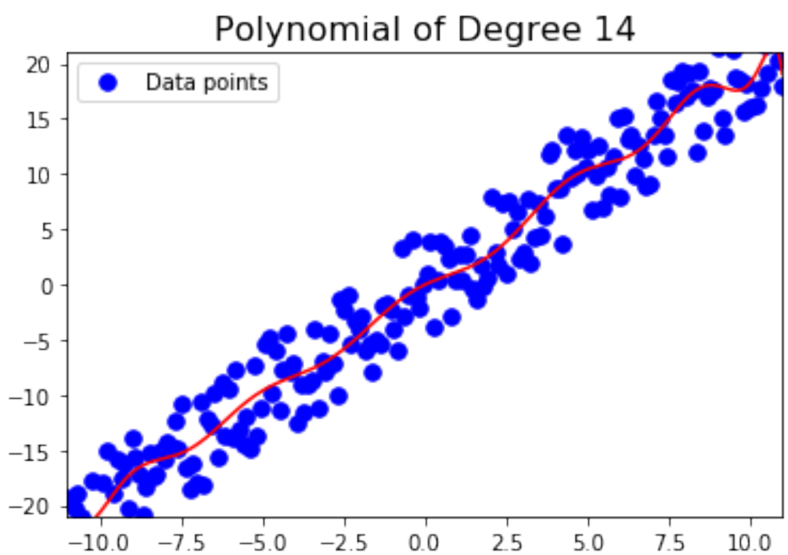


Let's calculate a polynomial approximation of the above device.

In [13]:
```
#Play around with degree here to try and fit different degree polynomi
degree=14 # change the degree here
D_a = data_matrix(x_a,degree)
p_a = leastSquares(D_a, y_a)

fig=plt.figure()
ax=fig.add_subplot(111,xlim=[-11,11],ylim=[-21,21])
x_a_,y_a_=poly_curve(p_a,x_a)
ax.plot(x_a,y_a,'.b',markersize=15)
ax.plot(x_a_, y_a_, 'r')
ax.legend(['Data points'])
plt.title('Polynomial of Degree %d' %(len(p_a)-1),fontsize=16)
```

```
/Users/manlai/miniconda3/lib/python3.7/site-packages/ipykernel_launch
er.py:33: FutureWarning: `rcond` parameter will change to the default
of machine precision times ``max(M, N)`` where M and N are the input
matrix dimensions.
To use the future default and silence this warning we advise to pass
`rcond=None`, to keep using the old, explicitly pass `rcond=-1`.
```

Out[13]: Text(0.5, 1.0, 'Polynomial of Degree 14')



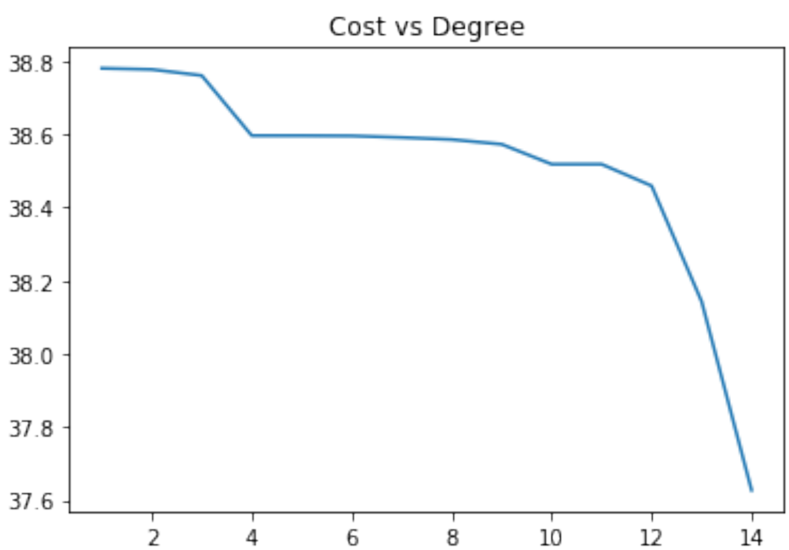
Part b)

In [9]:
```
def cost(x, y, start_deg, end_deg):
    """Given a set of x and y points, and a range of polynomial degree
    this function calculates polynomial fits to the data for polynomia
    of different degrees. It returns the "cost", i.e. the magnitude of
   The output is an array of the cost corresponding to each degree.
   """
    c = []
    for degree in range(start_deg, end_deg):
        D = data_matrix(x,degree)
        params = leastSquares(D,y)
        error = np.linalg.norm(y-np.dot(D,params))
        c.append(error)
    return c
```

In [10]:
```
start = 1
end = 15
fig=plt.figure()
ax=fig.add_subplot(111)
ax.plot(range(start, end), cost(x_a,y_a,start,end))
plt.title('Cost vs Degree')
```

```
/Users/manlai/miniconda3/lib/python3.7/site-packages/ipykernel_launch
er.py:33: FutureWarning: `rcond` parameter will change to the default
of machine precision times ``max(M, N)`` where M and N are the input
matrix dimensions.
To use the future default and silence this warning we advise to pass
`rcond=None`, to keep using the old, explicitly pass `rcond=-1`.
```

Out[10]: `Text(0.5, 1.0, 'Cost vs Degree')`



Question 4: Sparse Imaging

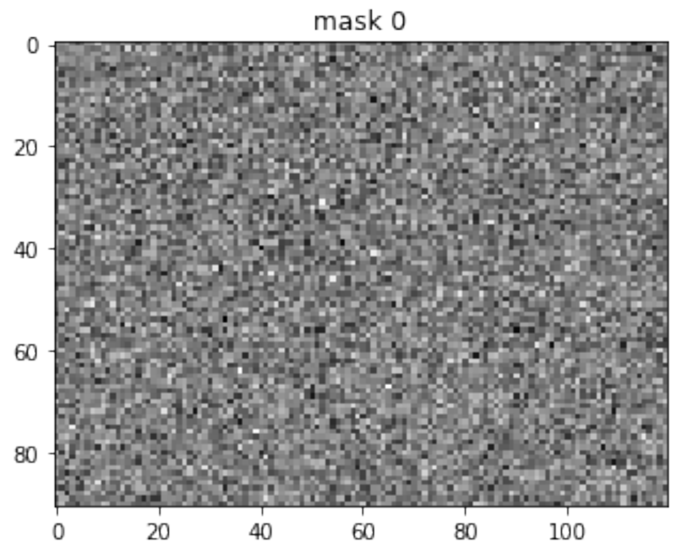
This example tries to reconstruct an image using the Orthogonal Matching Pursuit algorithm.

In [15]:
```
# imports
import matplotlib.pyplot as plt
import numpy as np
from scipy import misc
from IPython import display
import sys
import imageio
%matplotlib inline

def randMasks(numMasks, numPixels):
    randNormalMat = np.random.normal(0,1,(numMasks,numPixels))
    # make the columns zero mean and normalize
    for k in range(numPixels):
        # make zero mean
        randNormalMat[:,k] = randNormalMat[:,k] - np.mean(randNormalMa
        # normalize to unit norm
        randNormalMat[:,k] = randNormalMat[:,k] / np.linalg.norm(randN
    A = randNormalMat.copy()
    Mask = randNormalMat - np.min(randNormalMat)
    return Mask,A

def simulate():
    # read the image in grayscale
    I = np.load('helper.npy')
    sp = np.sum(I)
    numMeasurements = 6500
    numPixels = I.size
    Mask, A = randMasks(numMeasurements,numPixels)
    full_signal = I.reshape((numPixels,1))
    measurements = np.dot(Mask,full_signal)
    measurements = measurements - np.mean(measurements)
    return measurements, A
```

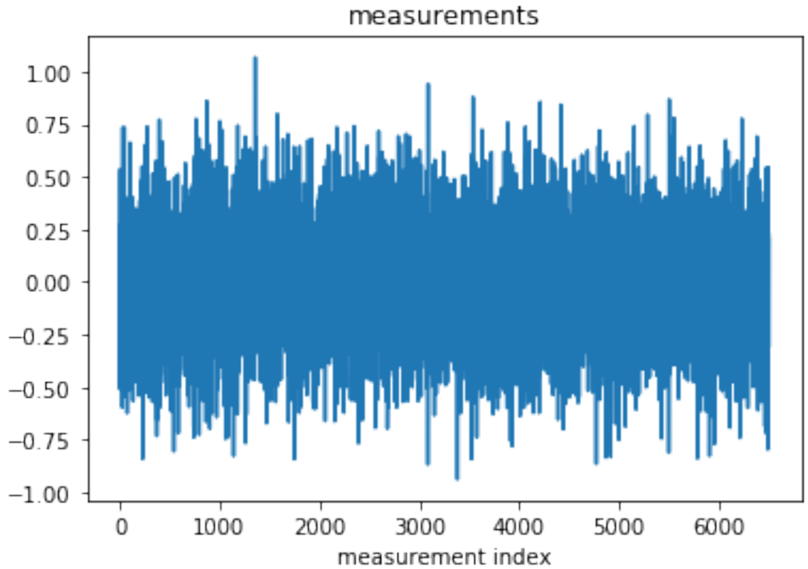
Part (a)

In [16]:
```
measurements, A = simulate()

# THE SETTINGS FOR THE IMAGE - PLEASE DO NOT CHANGE
height = 91
width = 120
sparsity = 476
numPixels = len(A[0])
```

In [17]:
```
# CHOOSE DIFFERENT MASKS TO PLOT
chosenMaskToDisplay = 0

M0 = A[chosenMaskToDisplay].reshape((height,width))
plt.title('mask %d'%chosenMaskToDisplay)
plt.imshow(M0, cmap=plt.cm.gray, interpolation='nearest');
```



In [18]:
```
# measurements
plt.title('measurements')
plt.plot(measurements)
plt.xlabel('measurement index')
plt.show()
```

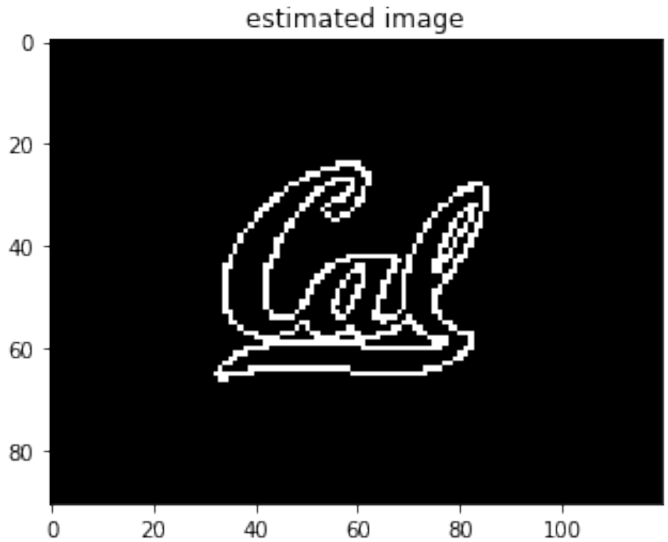


In [23]:
```
# OMP algorithm
# THERE ARE MISSING LINES THAT YOU NEED TO FILL
def OMP(imDims, sparsity, measurements, A):
    r = measurements.copy()
    indices = []

    # Threshold to check error. If error is below this value, stop.
    THRESHOLD = 0.1

    # For iterating to recover all signal
    i = 0

    while i < sparsity and np.linalg.norm(r) > THRESHOLD:
       # Calculate the inner products of r with columns of A
        print('%d - '%i,end="",flush=True)
        simvec = A.T.dot(r)

        # Choose pixel location with highest inner product and add to
        # COMPLETE THE LINE BELOW
        best_index = np.argmax(np.abs(simvec))
        indices.append(best_index)

        # Build the matrix made up of selected indices so far
        # COMPLETE THE LINE BELOW
        Atrunc = A[:,indices]
```

```
        # Find orthogonal projection of measurements to subspace
        # spanned by recovered codewords
        b = measurements
        # COMPLETE THE LINE BELOW
        xhat = np.linalg.lstsq(Atrunc, b)[0]

        # Find component orthogonal to subspace to use for next measu
        # COMPLETE THE LINE BELOW
        r = b - Atrunc.dot(xhat)

        # This is for viewing the recovery process
        if i % 10 == 0 or i == sparsity-1 or np.linalg.norm(r) <= THRE
            recovered_signal = np.zeros(numPixels)
            for j, x in zip(indices, xhat):
                recovered_signal[j] = x
            Ihat = recovered_signal.reshape(imDims)
            plt.title('estimated image')
            plt.imshow(Ihat, cmap=plt.cm.gray, interpolation='nearest'
            display.clear_output(wait=True)
            display.display(plt.gcf())

        i = i + 1

    display.clear_output(wait=True)

    # Fill in the recovered signal
    recovered_signal = np.zeros(numPixels)
    for i, x in zip(indices, xhat):
        recovered_signal[i] = x

    return recovered_signal
```

Part (b)

In [24]:
```
rec = OMP((height,width), sparsity, measurements, A)
```



PRACTICE: Part (c)

In [ ]:
```
# the setting

# file name for the sparse image
fname = 'figures/smiley.png'
# number of measurements to be taken from the sparse image
numMeasurements = 6500
# the sparsity of the image
sparsity = 400

# read the image in black and white
I = imageio.imread(fname, as_gray=True)
# normalize the image to be between 0 and 1
I = I/np.max(I)

# shape of the image
imageShape = I.shape
# number of pixels in the image
numPixels = I.size

plt.title('input image')
plt.imshow(I, cmap=plt.cm.gray, interpolation='nearest');
```

In [ ]:
```
# generate your image masks and the underlying measurement matrix
Mask, A = randMasks(numMeasurements,numPixels)
# vectorize your image
full_signal = I.reshape((numPixels,1))
# get the measurements
measurements = np.dot(Mask,full_signal)
# remove the mean from your measurements
measurements = measurements - np.mean(measurements)
```

In [ ]:
```
# measurements
plt.title('measurements')
plt.plot(measurements)
plt.xlabel('measurement index')
plt.show()
```

Question 6: Noise Cancelling Headphones (PRACTICE)

In [25]:
```
%matplotlib inline

import numpy as np
from matplotlib.pyplot import plot
from scipy.io import wavfile

from audio_support import wavPlayer
from audio_support import loadSounds
from audio_support import recordAmbientNoise
```

Part c)

In the following cell, implement the least squares solution to

$$min_{\vec{x}} \left|A\vec{x} - \vec{b}\right|$$

In [ ]:
```
def doLeastSquares(A,b):
    # BEGIN

    # END
    return x;
```

Part d)

Use your least squares solution to find the gamma that minimizes the effect of noise given:

$$\vec{n} = \begin{bmatrix} 0.10 \\ 0.37 \\ -0.45 \\ 0.068 \\ 0.036 \end{bmatrix}; \vec{r}_A = \begin{bmatrix} 0 \\ 0.11 \\ -0.31 \\ -0.012 \\ -0.018 \end{bmatrix}; \vec{r}_B = \begin{bmatrix} 0 \\ 0.22 \\ -0.20 \\ 0.080 \\ 0.056 \end{bmatrix}; \vec{r}_C = \begin{bmatrix} 0 \\ 0.37 \\ -0.44 \\ 0.065 \\ 0.038 \end{bmatrix}$$

In [ ]:

```
n1 = 0.10;
n2 = 0.37;
n3 = -0.45;
n4 = 0.068;
n5 = 0.036;

a1 = 0;
a2 = 0.11;
a3 = -0.31;
a4 = -0.012;
a5 = -0.018;

b1 = 0;
b2 = 0.22;
b3 = -0.20;
b4 = 0.080;
b5 = 0.056;

c1 = 0;
c2 = 0.37;
c3 = -0.44;
c4 = 0.065;
c5 = 0.038;

# BEGIN

'...'
gamma =

# END
print(gamma)
```

Report the results for your gamma-vector.

Part e)

First, we'll load the sounds from the included .wav files.

In [ ]:
```
[music_Fs, music_y, noise1_y, noise1_Fs, noise2_y, noise2_Fs] = loadSc
```

In [ ]:
```
noise1_y
```

We can use the following function to listen to our signals throughout this notebook.

Listen to each of the loaded sounds (`music_y` , `noise1_y` , and `noise2_y`). What do you hear?

In [ ]:
```
wavPlayer(music_y, music_Fs)
```

Add the first noise to the signal and listen to the result.

In [ ]:
```
# BEGIN

# END
```

Add the second noise to the signal and listen to the result.

In [ ]:
```
# BEGIN

# END
```

Part f)

Next, we will simulate the recording of `noise1` using a simulated microphone array.

In [ ]:
```
numberOfMicrophones = 3;
R = recordAmbientNoise(noise1_y,noise1_Fs,numberOfMicrophones);
```

In the cell below, calculate the gamma-vector using the least squares approach (you should calculate `gamma` from `R` and `noise1_y`).

In [ ]:
```
# BEGIN
gamma =
# END
```

In the cell below, create the noise cancellation signal by multiplying `R` and `gamma`. Add the result to `music_y` (with the right sign) to get `signalFromSpeaker`.

In [ ]:
```
# BEGIN
'...'
signalFromSpeaker =
# END
```

Part g)

Generate the signal at the listener's ear by adding the speaker signal (`signalFromSpeaker`) to the original noise signal (`noise1_y`).

In [ ]:
```
# BEGIN
signalAtEar =
# END
```

Listen to the noisy and noise-cancelled signal.

In [ ]:
```
wavPlayer(noisyMusic, music_Fs)
wavPlayer(signalAtEar, music_Fs)
```

What difference can you hear between these signals?

Part h)

Now, we'll see how well this gamma works for other noise.

We will run through the simulation again, but this time, we will just use the gamma from before instead of going through a training step.

In [ ]:
```
noisyMusic_2 = music_y + noise2_y;
R_2 = recordAmbientNoise(noise2_y,noise2_Fs,numberOfMicrophones);
# BEGIN
'...'
signalFromSpeaker_2 = '...'
signalAtEar_2 = '...'
# END

wavPlayer(noisyMusic_2, music_Fs)
wavPlayer(signalAtEar_2, music_Fs)
```

What do you hear in the noise-cancelled signal?

Part (a)

Listen to the recording you made, stored in the file `recording.wav` . You can load recordings using the `load_recording` function that we have written for you and imported. You can play recordings using the `play` function that we have also written and imported.

In [1]:
```
import numpy as np
from utils import load_recording, play, save_recording

RECORDING_FILE = "recording.wav"

r = load_recording(RECORDING_FILE)
play(r)
```

-0:10

Part (b)

Let $\vec{r}$ be your recording. Let us say you have access to the true lecture given by $\vec{l}$. You know that your received vector and the lecture have the relationship

$$\vec{r} = \alpha\vec{l} + \vec{n},$$

where α is an unknown constant. Estimate $\vec{n}$ by projecting $\vec{r}$ ontol $\vec{l}$ to recover α. What remains is $\vec{n}$. Assume that $\vec{l}$ is orthogonal to $\vec{n}$.

In [5]:
```
# Note that l and r are 1D arrays, not 2D arrays, so calling np.linalg
def projection(l, r):
    return np.dot(l, r) / (np.linalg.norm(l)**2) * l
```

In [6]:
```
def recover_noise(r, l):
    return r - projection(l,r)
```

In [7]:
```
#We use the technique above to recover candidate interference signals.

#noisy_lectures contains the lecture recordings with interference
noisy_lectures = [load_recording("noisy_lecture_{}.wav".format(i+1)) f

# lectures contains the clean lectures that you played to understand t
lectures = [load_recording("lecture_{}.wav".format(i+1)) for i in rang

# interferences is a matrix whose columns contain the possible interfe
interferences = np.column_stack([recover_noise(r_i, l_i) for r_i, l_i

#you can change the index 0 below to play different lectures and recor
play(lectures[0])
play(noisy_lectures[0])
play(interferences[:, 0])
```

-0:14

-0:20

-0:21

Part (c)

Now, given $\vec{r}$ and the $\vec{n}_i$, and the model

$$\vec{r} = \vec{l} + \sum_{i=1}^{s} \beta_i \vec{n}_i,$$

use least squares to recover $\vec{l}$. The $\vec{n}_i$ are computed from the $\vec{r}_i$ using your function from the previous part.

In [11]:
```
#r is the signal you have recorded
r = load_recording(RECORDING_FILE)

# Project r onto the interference signals to recover the component of
# What remains must be the lecture.

A = interferences
b = r

# Hint, use least squares
betas = np.linalg.lstsq(A, b)[0]
print(betas)

# This is the recovered lecture. Have you successfully recovered a
# noise-free signal? Or is it still noisy?
l = b - A.dot(betas)

play(l)
```

```
[-0.07080106 -0.09364032  0.11021623  0.02728798]
```

-0:16

Part (d)

Now, we will include the effect of the travel time of the noise signals, using the model

$$\vec{r} = \vec{l} + \sum_{i=1}^{s} \beta_i \vec{n}_i^{(k_i)}.$$

Recover $\vec{l}$ using this new model, using OMP, by filling in the blanks in the below code block.

In [51]:
```
from utils import cross_correlate

r = load_recording(RECORDING_FILE)
interferences = [recover_noise(r_i, l_i) for r_i, l_i in zip(noisy_lec

k = np.zeros(4, "int")

vecs = []

# the initial residual for OMP
residual = r

for _ in range(4):
    best_corr = float("-inf")
    best_vec = None
    # We first iterate over all the interferences n_i
    for i, n_i in enumerate(interferences):
        # for each interference, we look through its correlation with

        # Fill in the arguments to cross_correlate
        for k_i, corr in enumerate(cross_correlate(
            residual,
            n_i
        ) # This function returns a vector of cross correlation values
          # the residual/received signal with every possible delay of
        ):
            # we find the (noise, shift) pair that maximizes the corre
            if corr > best_corr:
                best_corr = corr
                best_vec = (i, k_i)
    i, k_i = best_vec
    k[i] = k_i

    # we shift the best noise by the best shift and add it to our list
    vecs.append(np.roll(interferences[i], k[i]))

    A = np.column_stack(vecs) # this is the matrix that captures all t

    # Use least squares to update the residual
    residual = residual - np.array(vecs).T.dot(np.linalg.lstsq(np.arra

l = residual
play(l)
```

-0:00

In [ ]: