1. Mechanical Gram-Schmidt.

a) let v_i be vectors in V and u_i be vectors in U where $i \in \mathbb{R}$.

Thus $u_1 = \text{norm}(v_1) = v_1$

$$u_2 = \text{norm}(v_2 - \langle v_2, u_1\rangle u_1) = \text{norm}(v_2 - \langle v_1, v_2\rangle v_1) = \text{norm}\left(\begin{bmatrix}1\\1\\1\\0\\0\end{bmatrix} - \begin{bmatrix}1\\0\\0\\0\\0\end{bmatrix}\right) = \text{norm}\begin{pmatrix}0\\1\\1\\0\\0\end{pmatrix}$$

$$= \frac{1}{\sqrt{2}}\begin{bmatrix}0\\1\\1\\0\\0\end{bmatrix} = \begin{bmatrix}0\\1/\sqrt{2}\\1/\sqrt{2}\\0\\0\end{bmatrix}$$

$$u_2 = \text{norm}(v_3 - \langle v_3, u_1\rangle u_1 - \langle v_3, u_2\rangle u_2)$$

$$= \text{norm}\left(\begin{bmatrix}1\\1\\1\\1\\1\end{bmatrix} - \begin{bmatrix}1\\0\\0\\0\\0\end{bmatrix} - \sqrt{2}\begin{bmatrix}0\\1/\sqrt{2}\\1/\sqrt{2}\\0\\0\end{bmatrix}\right) = \text{norm}\left(\begin{bmatrix}0\\1\\1\\1\\1\end{bmatrix} - \begin{bmatrix}0\\1\\1\\0\\0\end{bmatrix}\right) = n\left(\begin{bmatrix}0\\0\\0\\1\\1\end{bmatrix}\right)$$

$$= \frac{1}{\sqrt{2}}\begin{bmatrix}0\\0\\0\\1\\1\end{bmatrix} = \begin{bmatrix}0\\0\\0\\1/\sqrt{2}\\1/\sqrt{2}\end{bmatrix}$$

Thus $U = \begin{bmatrix}1 & 0 & 0\\0 & 1/\sqrt{2} & 0\\0 & 1/\sqrt{2} & 0\\0 & 0 & 1/\sqrt{2}\\0 & 0 & 1/\sqrt{2}\end{bmatrix}$

Thus

$$V(V^TV)^{-1}V^T\vec{w} = \begin{bmatrix}1&1&1\\0&1&1\\0&1&1\\0&0&1\\0&0&1\end{bmatrix} \times \left(\begin{bmatrix}1&0&0&0&0\\1&1&1&0&0\\1&1&1&1&1\end{bmatrix}\begin{bmatrix}1&1&1\\0&1&1\\0&1&1\\0&0&1\\0&0&1\end{bmatrix}\right)^{-1} \times \begin{bmatrix}1&0&0&0&0\\1&1&1&0&0\\1&1&1&1&1\end{bmatrix} \times \begin{bmatrix}1\\-1\\0\\-1\\0\end{bmatrix} = \begin{bmatrix}1\\-1/2\\-1/2\\-1/2\\-1/2\end{bmatrix}$$

$$U(U^TU)^{-1}U^T\vec{w} = \begin{bmatrix}1&0&0\\0&1/\sqrt{2}&0\\0&1/\sqrt{2}&0\\0&0&1/\sqrt{2}\\0&0&1/\sqrt{2}\end{bmatrix} \times \left(\begin{bmatrix}1&0&0&0&0\\0&1/\sqrt{2}&1/\sqrt{2}&0&0\\0&0&0&1/\sqrt{2}&1/\sqrt{2}\end{bmatrix} \times \begin{bmatrix}1&0&0\\0&1/\sqrt{2}&0\\0&1/\sqrt{2}&0\\0&0&1/\sqrt{2}\\0&0&1/\sqrt{2}\end{bmatrix}\right)^{-1} \times \begin{bmatrix}1&0&0&0&0\\0&1/\sqrt{2}&1/\sqrt{2}&0&0\\0&0&0&1/\sqrt{2}&1/\sqrt{2}\end{bmatrix} \times \begin{bmatrix}1\\-1\\0\\-1\\0\end{bmatrix} = \begin{bmatrix}1\\-1/2\\-1/2\\-1/2\\-1/2\end{bmatrix} \quad \{\text{same}\}$$

b) $u_1 = \text{norm}(v_1) = \frac{1}{\sqrt{3}}v_1 = \begin{bmatrix}1/\sqrt{3}\\0\\1/\sqrt{3}\\1/\sqrt{3}\\0\end{bmatrix}$

$$u_2 = \text{norm}(v_2 - \langle v_2, u_1\rangle u_1) = \text{norm}\left(\begin{bmatrix}1\\1\\0\\-1\\-1\end{bmatrix} - \left(\begin{bmatrix}1\\1\\0\\-1\\-1\end{bmatrix} \cdot \begin{bmatrix}1/\sqrt{3}\\0\\1/\sqrt{3}\\1/\sqrt{3}\\0\end{bmatrix}\right) \times \begin{bmatrix}1/\sqrt{3}\\0\\1/\sqrt{3}\\1/\sqrt{3}\\0\end{bmatrix}\right) = \overset{\text{norm}}{}\left(\begin{bmatrix}1\\1\\0\\-1\\-1\end{bmatrix}\right) = \frac{1}{2}\begin{bmatrix}1\\1\\0\\-1\\-1\end{bmatrix} = \begin{bmatrix}\frac{1}{2}\\\frac{1}{2}\\0\\-\frac{1}{2}\\-\frac{1}{2}\end{bmatrix}$$

$$u_3 = \text{norm}(v_3 - \langle v_3, u_1\rangle u_1 - \langle v_3, u_2\rangle u_2)$$

$$= \text{norm}\left(\begin{bmatrix}-1\\-1\\0\\1\\1\end{bmatrix} - \vec{0} - \begin{bmatrix}-1\\-1\\0\\1\\1\end{bmatrix}\right) = \text{norm}\left(\begin{bmatrix}0\\0\\0\\0\\0\end{bmatrix}\right) = \begin{bmatrix}0\\0\\0\\0\\0\end{bmatrix}$$

Thus $U = \begin{bmatrix}\frac{1}{\sqrt{3}} & \frac{1}{2} & 0\\0 & \frac{1}{2} & 0\\\frac{1}{\sqrt{3}} & 0 & 0\\\frac{1}{\sqrt{3}} & -\frac{1}{2} & 0\\0 & -\frac{1}{2} & 0\end{bmatrix}$

thus:

$$V(V^TV)^{-1}V^T\vec{w} = \begin{bmatrix}1&1&-1\\0&1&-1\\1&0&0\\1&-1&1\\0&-1&1\end{bmatrix} \times \left(\begin{bmatrix}1&0&1&1&0\\1&1&0&-1&-1\\-1&-1&0&1&1\end{bmatrix}\begin{bmatrix}1&1&-1\\0&1&-1\\1&0&0\\1&-1&1\\0&-1&1\end{bmatrix}\right)^{-1} \times \begin{bmatrix}1&0&1&1&0\\1&1&0&-1&-1\\-1&-1&0&1&1\end{bmatrix} \times \begin{bmatrix}1\\-1\\0\\-1\\0\end{bmatrix} = \begin{bmatrix}1/4\\1/4\\0\\-1/4\\-1/4\end{bmatrix}$$

$$U(U^TU)^{-1}U^T\vec{w} = \begin{bmatrix}\frac{1}{\sqrt{3}}&1/2&0\\0&1/2&0\\\frac{1}{\sqrt{3}}&0&0\\\frac{1}{\sqrt{3}}&-1/2&0\\0&-1/2&0\end{bmatrix} \times \left(\begin{bmatrix}\frac{1}{\sqrt{3}}&0&\frac{1}{\sqrt{3}}&\frac{1}{\sqrt{3}}&0\\1/2&1/2&0&-1/2&-1/2\\0&0&0&0&0\end{bmatrix}\begin{bmatrix}\frac{1}{\sqrt{3}}&1/2&0\\0&1/2&0\\\frac{1}{\sqrt{3}}&0&0\\\frac{1}{\sqrt{3}}&-1/2&0\\0&-1/2&0\end{bmatrix}\right)^{-1} \times \begin{bmatrix}\frac{1}{\sqrt{3}}&0&1/\sqrt{3}&1/\sqrt{3}&0\\1/2&1/2&0&-1/2&-1/2\\0&0&0&0&0\end{bmatrix}\begin{bmatrix}1\\-1\\0\\-1\\0\end{bmatrix} = \begin{bmatrix}1/4\\1/4\\0\\-1/4\\-1/4\end{bmatrix}$$

c) U & V have the same basis

Thus $V = QR \Rightarrow Q = U =$ the orthogonal matrix

Thus $R = Q^{-1}V = Q^TV = U^TV$

a) $\Rightarrow R = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1/\sqrt{2} & 1/\sqrt{2} & 0 & 0 \\ 0 & 0 & 0 & 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & \sqrt{2} & \sqrt{2} \\ 0 & 0 & \sqrt{2} \end{bmatrix}$

b) $\Rightarrow R = \begin{bmatrix} 1/\sqrt{3} & 0 & 1/\sqrt{3} & 1/\sqrt{3} & 0 \\ 1/2 & 1/2 & 0 & -1/2 & -1/2 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & -1 \\ 0 & 1 & -1 \\ 1 & 0 & 0 \\ 1 & -1 & 0 \\ 0 & -1 & 1 \end{bmatrix} = \begin{bmatrix} \sqrt{3} & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

2. Deconstructing Trolls.

a) Yes. for both conditions. the vectors are still orthogonal.

For example (or explain): scaling just change the length but not the position (relative to each other)

b) They are not necessarily to be orthogonal.

For example $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ as $\vec{a}$ and $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ as $\vec{b}$. and $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$ as $\vec{x}$ and $\begin{bmatrix} 2 \\ 2 \end{bmatrix}$ as $\vec{y}$

obviously $\vec{x}$ and $\vec{y}$ are not orthogonal since $\langle \vec{x}, \vec{y} \rangle \neq 0$

c) We can use the original $\vec{x}_1$ and $\vec{x}_2$ for any any scalings or $\vec{x}$ and $\vec{y}$ or row reduction forms of M. as long as keep the matrix as same basis and they are orthogonal

5. Fansheng Cheng 3033207855

Samuel Harreschou 23804699

prob8

August 9, 2017

1 EE16A Homework 8

1.1 Question 2: Deconstructing Trolls

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import wave as wv
        import scipy
        from scipy import io
        import scipy.io.wavfile
        from scipy.io.wavfile import read
        from IPython.display import Audio
        import warnings
        warnings.filterwarnings('ignore')
        sound_file_1 = 'm1.wav'
        sound_file_2 = 'm2.wav'
        rate1,corrupt1 = scipy.io.wavfile.read('m1.wav')
        rate2,corrupt2 = scipy.io.wavfile.read('m2.wav')
        corrupt1 = corrupt1.astype(np.float)
        corrupt2 = corrupt2.astype(np.float)
```

Just like last time, let's listen to the inputs.

```
In [2]: Audio(url='m1.wav', autoplay=False)

Out[2]: <IPython.lib.display.Audio object>

In [3]: Audio(url='m2.wav', autoplay=False)

Out[3]: <IPython.lib.display.Audio object>
```

In the cell below, complete the function to find the vectors $\vec{a}$ and $\vec{b}$. Make sure that $\vec{a}$ is the original speech and not the troll.

```
In [4]: def remove_troll(m1, m2):
            ## YOUR CODE HERE
            a = m1 - (np.dot(m1, m2)/ np.dot(m2, m2)) * m2
            b = m2
            return a, b
```

```
In [5]: a, b = remove_troll(corrupt1, corrupt2)

Out[5]: (array([  2.59899156,   4.70696241,   3.34705959, ...,  91.47055935,
                 86.64535883,  69.97438671]),
         array([ -211.,  -229.,  -169., ..., -1301., -1219.,  -996.]))

In [11]: result = np.dot(a, b)
         print(result)

-1.90734863281e-05
```

Run the cell below to test your function.

```
In [13]: # The result is really close to 0, and thus can be seen as orthogonal

In [6]: a, b = remove_troll(corrupt1, corrupt2)
        Audio(data=a, rate=rate1)

Out[6]: <IPython.lib.display.Audio object>
```

Let's now compare our output here to the output from Homework 1. Read through the block of code below and comment on it's output

```
In [7]: ## First, let's compute the original vectors representing the speakers using the techniq
        a_u = np.sqrt(2)/(1+np.sqrt(3))
        a_v = np.sqrt(6)/(1+np.sqrt(3))
        b_u = np.sqrt(2)/(1+np.sqrt(3))
        b_v = -1*np.sqrt(2)/(1+np.sqrt(3))
        s1 = a_u*corrupt1 + a_v*corrupt2
        s2 = b_u*corrupt1 + b_v*corrupt2

        ## Here, we will compute various dot products to see which vectors are orthogonal.
        ## Note that we normalize the vectors before we compare, this is because we want
        ## to get rid of any scaling.
        print("Dot product of the two original speaker outputs: ", np.dot(s1/np.linalg.norm(s1),
        print("Dot product of calculated a and b: ", np.dot(a/np.linalg.norm(a), b/np.linalg.nor

Dot product of the two original speaker outputs:  -0.00379040459598
Dot product of calculated a and b:  2.77555756156e-17

In [14]: # The result is really close to 0, and thus can be seen as orthogonal
```

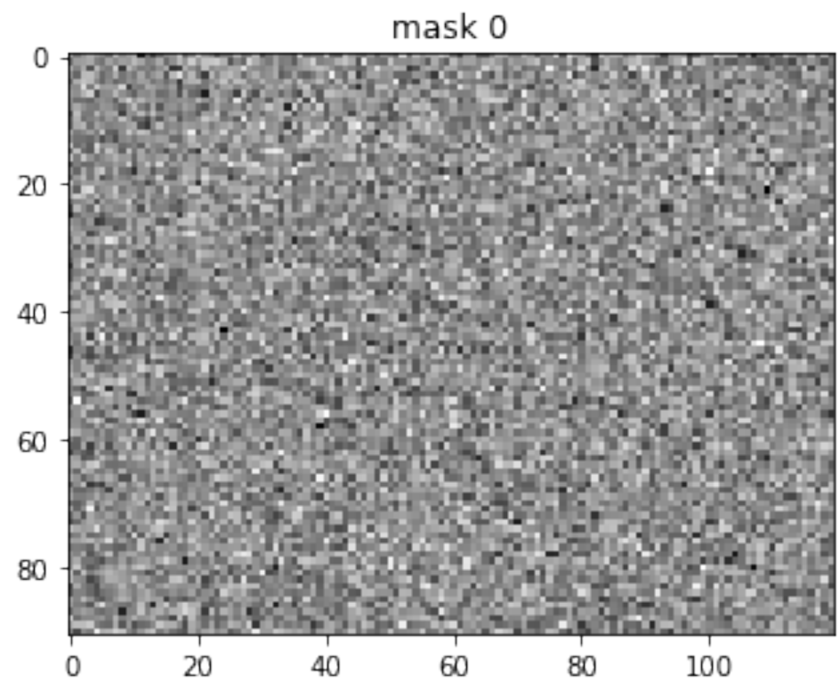
1.2 Question 3: Sparse Imaging

This example generates a sparse signal and tries to recover it using the orthogonal matching pursuit algorithm

```
In [8]: # imports
        import matplotlib.pyplot as plt
        import numpy as np
        from scipy import misc
        from IPython import display
        from simulator import *
        %matplotlib inline
```

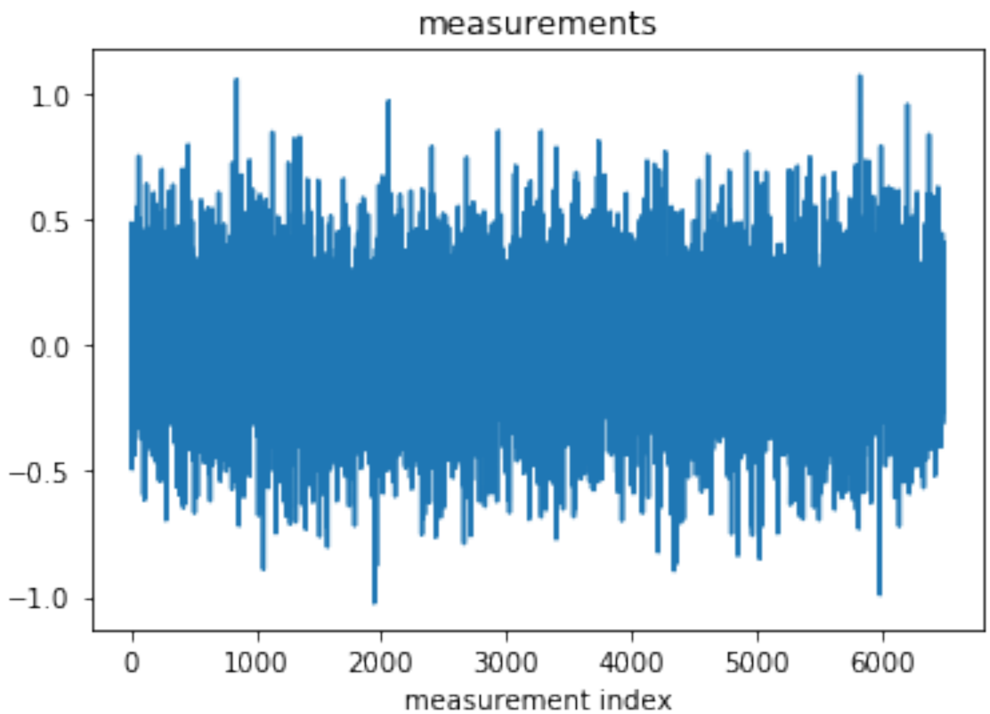
```
In [9]: measurements, A = simulate()

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

```
In [10]: # 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 [14]: # measurements
         plt.title('measurements')
```

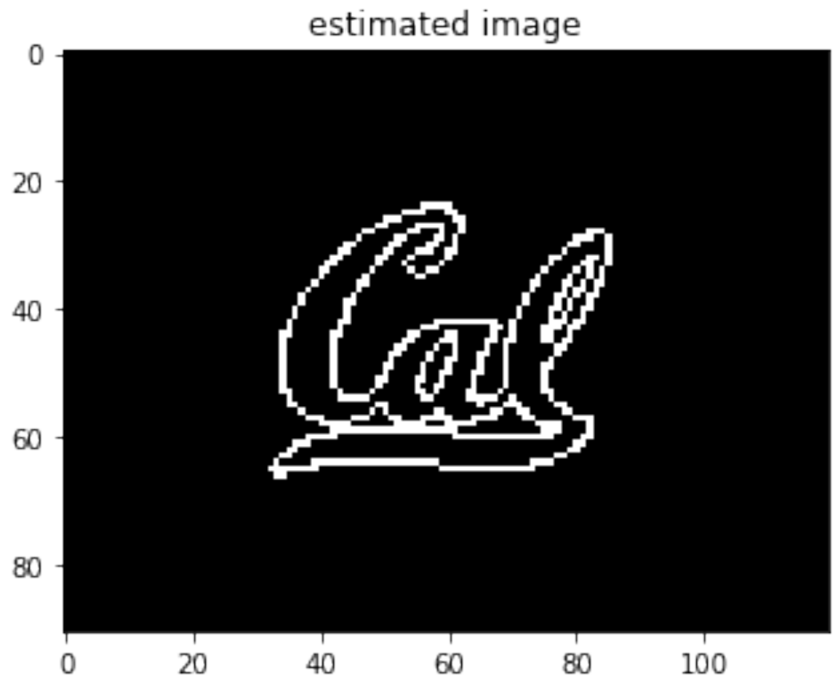
```
plt.plot(measurements)
plt.xlabel('measurement index')
plt.show()
```




```
In [15]: # 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 correlations
                 print('%d - '%i,end="",flush=True)
                 corrs = A.T.dot(r)

                 # Choose highest-correlated pixel location and add to collection
                 # COMPLETE THE LINE BELOW
                 best_index = np.argmax(np.abs(corrs))
```

```python
        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 measurement
        # 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) <= THRESHOLD:
            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
```

```python
In [25]: rec = OMP((height,width), sparsity, measurements, A)
```



1.2.1 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 = misc.imread(fname, flatten=1)
# 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()
```

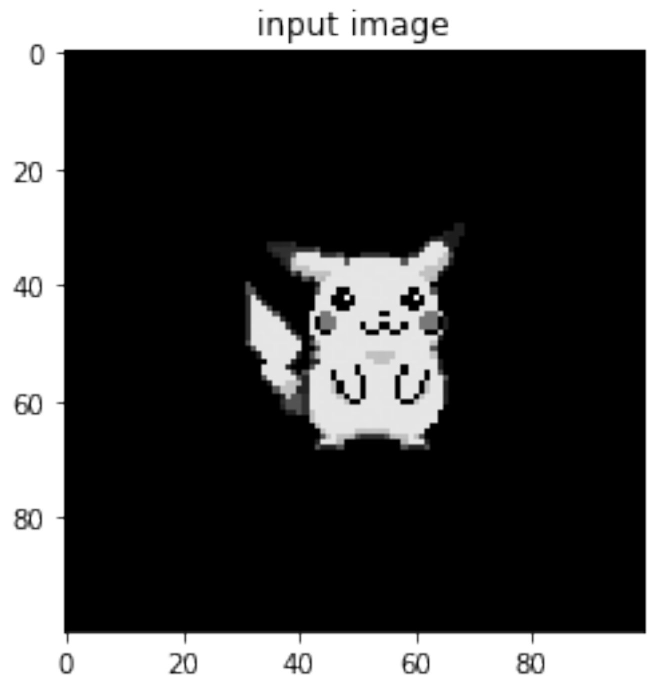
```
In [ ]: rec = OMP(imageShape, sparsity, measurements, A)
```

1.3 Question 4: Speeding Up OMP

This example generates a sparse signal and tries to recover it using the orthogonal matching pursuit algorithm

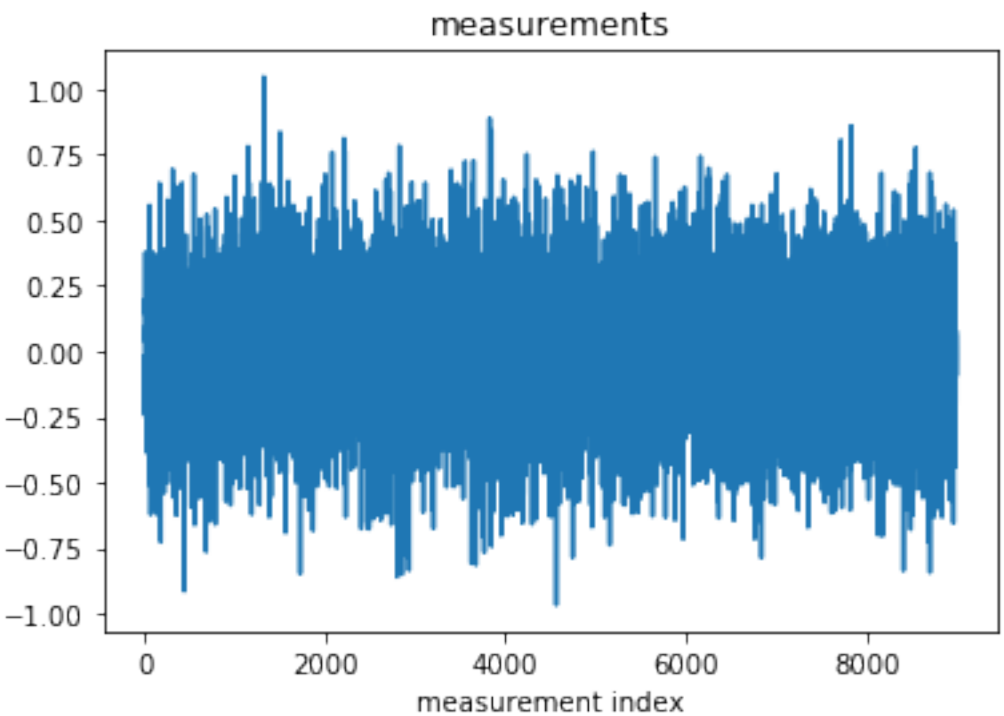
```
In [16]: # imports
         import matplotlib.pyplot as plt
         import numpy as np
         from scipy import misc
         from IPython import display
         from simulator import *
         %matplotlib inline
```

```
In [17]: # the setting

         # file name for the sparse image
         fname = 'figures/pika.png'
         # number of measurements to be taken from the sparse image
         numMeasurements = 9000
         # the sparsity of the image
         sparsity = 800

         # read the image in black and white
         I = misc.imread(fname, flatten=1)
         # 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 [18]:
```
# 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 [19]:
```
# measurements
plt.title('measurements')
plt.plot(measurements)
plt.xlabel('measurement index')
plt.show()
```




```
In [23]: # Write a function that returns a matrix U whose columns form
         # an orthonormal basis for the columns of the matrix A.
         def gramschmidt(A):
             the_A = np.transpose(A)
             n = np,linalg.norm(the_A[0])
             U = the_A / n

             for i in range(1, len(the_A)):
                 k = the_A[i]
                 for j in range(len(U)):
                     k -= np.dot(k, np.transpose(U[j]) * U[j])
                 m = np.linalg.norm(k)
                 if m > 0:
                     U = np.vstack((U, k / m))
             U = np.transpose(U)
             return U

         # A better option is to write a function that takes in a matrix U0
         # with orthonormal columns and a single new vector v and adds another
         # orthonormal column to U0 creating a new matrix U whose columns are orthonormal
         # and span the column space of {U0, v}.
         # Note: Using this function will make your code faster.
         def gramschmidt_addone(U0,v):
             if len(U0) == 0:
```

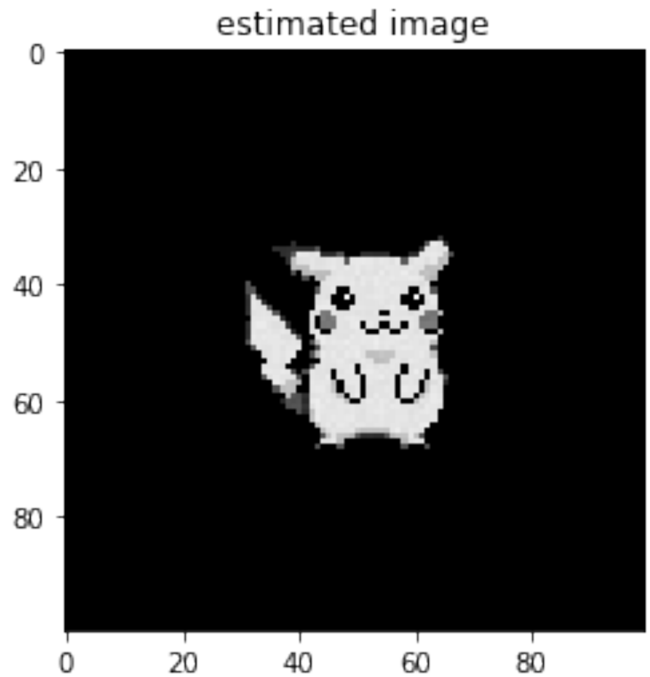
```
            U = v / np.linalg.norm(v)
        else:
            v2 = v - np.dot(U0, np.dot(np.transpose(U0), v))
            v2 = v2 / np.linalg.norm(v2)
            U = np.column_stack((U0, v2))
        return U
```

In [24]:
```
# 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

    ########################################
    ### THIS LINE INITIALIZES THE MATRIX U
    U = np.zeros([np.size(A,0),0])
    ########################################

    while i < sparsity and np.linalg.norm(r) > THRESHOLD:
        # calculate the correlations
        print('%d - '%i,end="",flush=True)
        corrs = A.T.dot(r)

        # Choose highest-correlated pixel location and add to collection
        # COMPLETE THE LINE BELOW
        best_index = np.argmax(np.abs(corrs))

        ###########################
        ### MODIFY THIS SECTION ###
        ## TO USE GRAM-SCHMIDT  ###
        ###########################

        indices.append(best_index)



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

        #############################
        ## CHOOSE ONE OF THESE LINES
#         U = gramschmidt(Atrunc)
```

```
    ### OR
    v = A[:,best_index]
    U = gramschmidt_addone(U,v)
    #############################

    # Find orthogonal projection of measurements to subspace
    # spanned by recovered codewords
    b = measurements

    ###################################
    ## COMPLETE THE LINES BELOW AND
    ## REWRITE THESE LINES USING GRAMSCHMIDT TO SPEED UP LEAST SQUARES
    xhat = np.linalg.lstsq(Atrunc, b)[0]
    r = b - Atrunc.dot(xhat)
    ####################################

    # Find component orthogonal to subspace to use for next measurement
    ## CHANGE THIS LINE

    ###########################
    ### ------------------- ###
    ###########################


    # This is for viewing the recovery process
    if i % 100 == 0 or i == sparsity-1 or np.linalg.norm(r) <= THRESHOLD:

        # RECOVERING xhat for plotting
        xhat = np.dot(np.linalg.inv(np.dot(Atrunc.T,Atrunc)),np.dot(Atrunc.T,b-r))

        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
```

```
In [25]: rec = OMP(imageShape, sparsity, measurements, A)
```




```
In [ ]:
```

```
In [ ]:
```