Assignment 1: Adam Thompson

SYDE556/750 Assignment 1: Representation in Populations of Neurons

- Due Date: January 23rd at midnight
- Total marks: 20 (20% of final grade)
- Late penalty: 1 mark per day
- It is recommended that you use a language with a matrix library and graphing capabilities. Two main suggestions are Python and MATLAB.
- *Do not use any code from Nengo*

In [388]:
```
%pylab inline
import numpy as np
```

```
Populating the interactive namespace from numpy and matplotlib
```

1) Representation of Scalars

1.1) Basic encoding and decoding

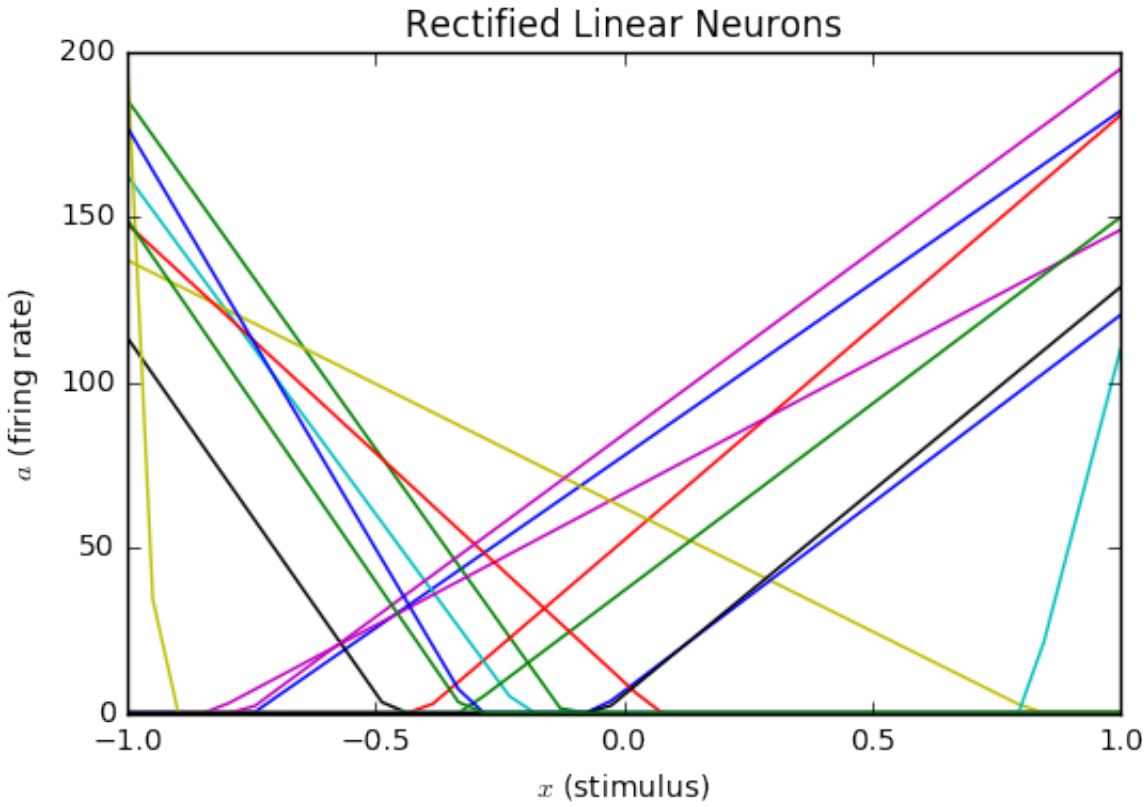
Write a program that implements a neural representation of a scalar value x. For the neuron model, use a rectified linear neuron model ($a = max(J, 0)$). Choose the maximum firing rates randomly (uniformly distributed between 100Hz and 200Hz at x=1), and choose the x-intercepts randomly (uniformly distributed between -0.95 and 0.95). Use those values to compute the corresponding α and J^{bias} parameters for each neuron. The encoders e are randomly chosen and are either +1 or -1 for each neuron. Go through the following steps:

In [389]:
```
class LinearNeuron(object):
    max_fr = 0
    alpha = 0
    jbias = 0
    encoder = 0

    def __init__(self):
        self.max_fr = np.random.uniform(low = 100, high = 200) # a @ x =
        xint = np.random.uniform(low = -0.95,high = 0.95) # x @ a = 0

        self.alpha = self.max_fr/(1-xint) #slope
        self.jbias = -self.alpha * xint #yint
        self.encoder = np.random.uniform(low = -1, high = 1)
        if self.encoder > 0: self.encoder = 1
        else: self.encoder = -1

def initLinearNeurons(n,X):
    A = np.zeros([n,len(X)])
    for i in range(0,n):
        neu = LinearNeuron()
        for j,x in enumerate(X): # for each step dx
            a = neu.encoder * neu.alpha * x + neu.jbias
            A[i][j] = max(a, 0)
    return A
```

a. [1 mark] Plot the neuron responses a_i for 16 randomly generated neurons. (See Figure 2.4 in the book for an example, but with a different neuron model and a different range of maximum firing rates).

- Since you can't compute this for every possible x value between -1 and 1, sample the x-axis with $dx = 0.05$. Use this sampling throughout this question)

In [390]:
```
n = 16
stepSize = 0.05
X = np.linspace(-1,1,2/stepSize)

A = initLinearNeurons(n,X)

for i in range(0,n): #for each neuron
    plot(X, A[i])
xlabel('$x$ (stimulus)')
ylabel('$a$ (firing rate)')
title('Rectified Linear Neurons');
```



b. [1 mark] Compute the optimal decoders d_i for those 16 neurons (as shown in class). Report their values.

- The easiest way to compute d is to use the matrix notation mentioned in the course notes. A is the matrix of neuron activities (the same thing used to generate the plot in 1.1a).

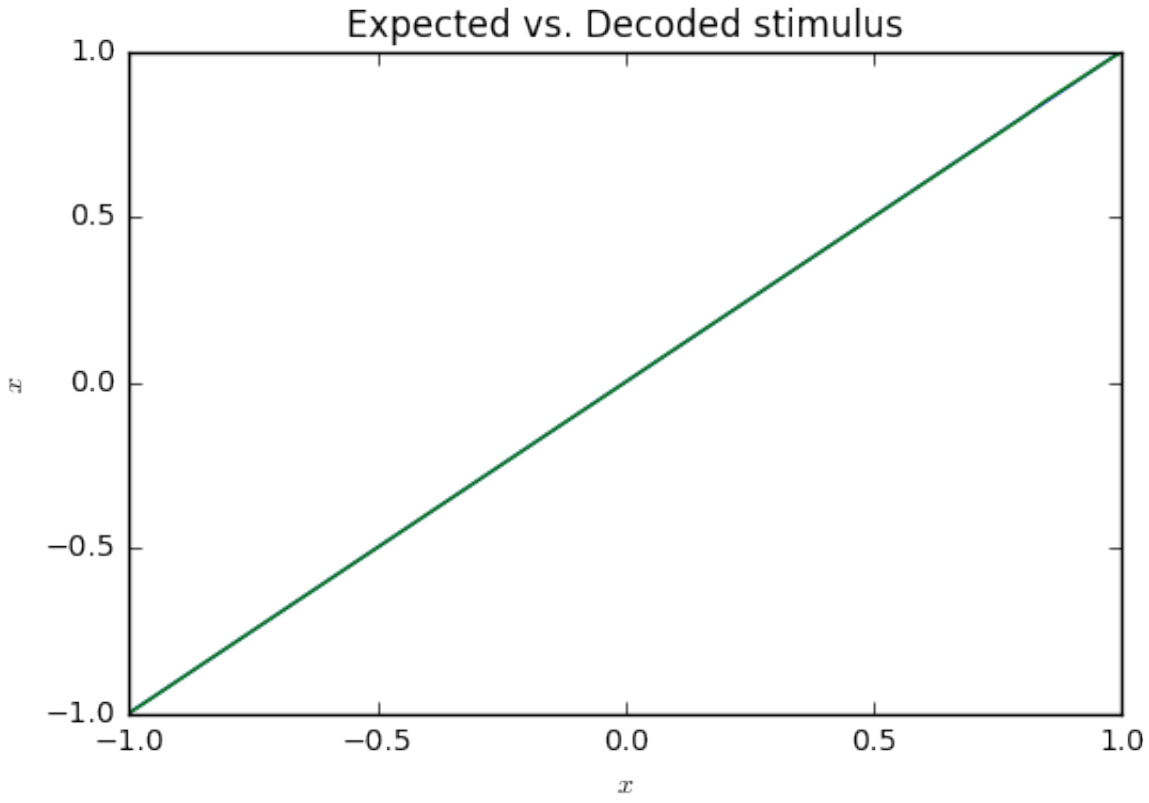
In [391]:
```
def decodeNeurons(A):
    Ypsilon = stepSize * np.matmul(A,X)
    Gamma = stepSize * np.matmul(A, np.transpose(A))
    Gamma_inv = np.linalg.inv(Gamma)
    d = np.dot(Gamma_inv, Ypsilon)
    return d

d = decodeNeurons(A)
print "Decoders: \n\n d = " + str(d)
```

```
Decoders:

 d = [ -3.72240696e-04  -1.17929235e-04  -9.32460364e-05   4.70247105e
-05
  -3.56935402e-04  -3.59500495e-03   3.39206371e-05   2.39995665e-04
   6.43436423e-03  -8.24604550e-05   4.14753155e-04   4.99407030e-04
   6.66941375e-06  -2.63410239e-06   1.80870143e-03  -5.41447006e-03]
```

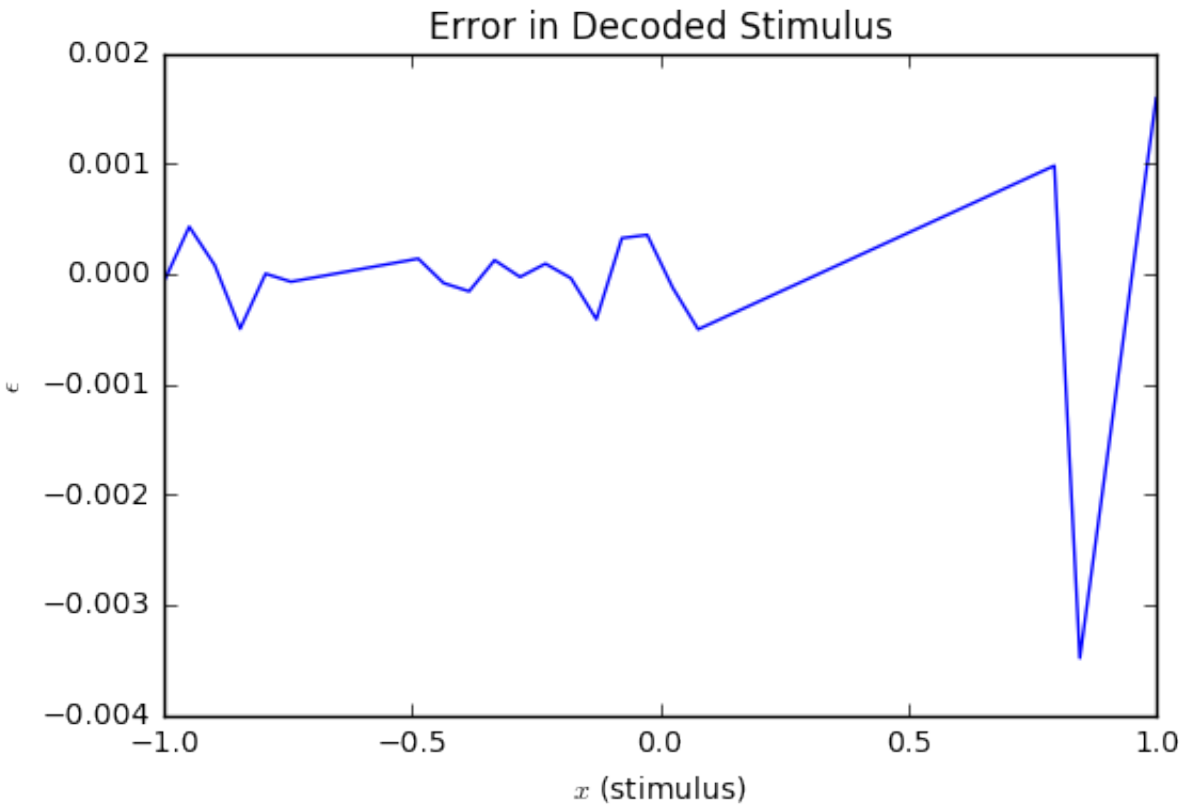
c. [1 mark] Compute and plot $\hat{x} = \sum_i d_i a_i$. Overlay on the plot the line $y = x$. (See Figure 2.7 for an example). Make a separate plot of $x - \hat{x}$ to see what the error looks like. Report the Root Mean Squared Error value.

In [392]:
```
X_hat = np.matmul(A.transpose(),d)

def getRMS(X, X_hat):
    RMS = 0
    Diff = X - X_hat
    for x in range(0,len(X)):
        RMS += np.power(Diff[x],2)
    RMS = np.sqrt(RMS/len(X))
    return RMS, Diff

RMS1, Diff = getRMS(X, X_hat)
print "RMS error = " + str(RMS1)
plot(X, X)
plot(X, X_hat)
xlabel('$x$')
ylabel('$x$')
title('Expected vs. Decoded stimulus');
```

```
RMS error = 0.000755688588185
```

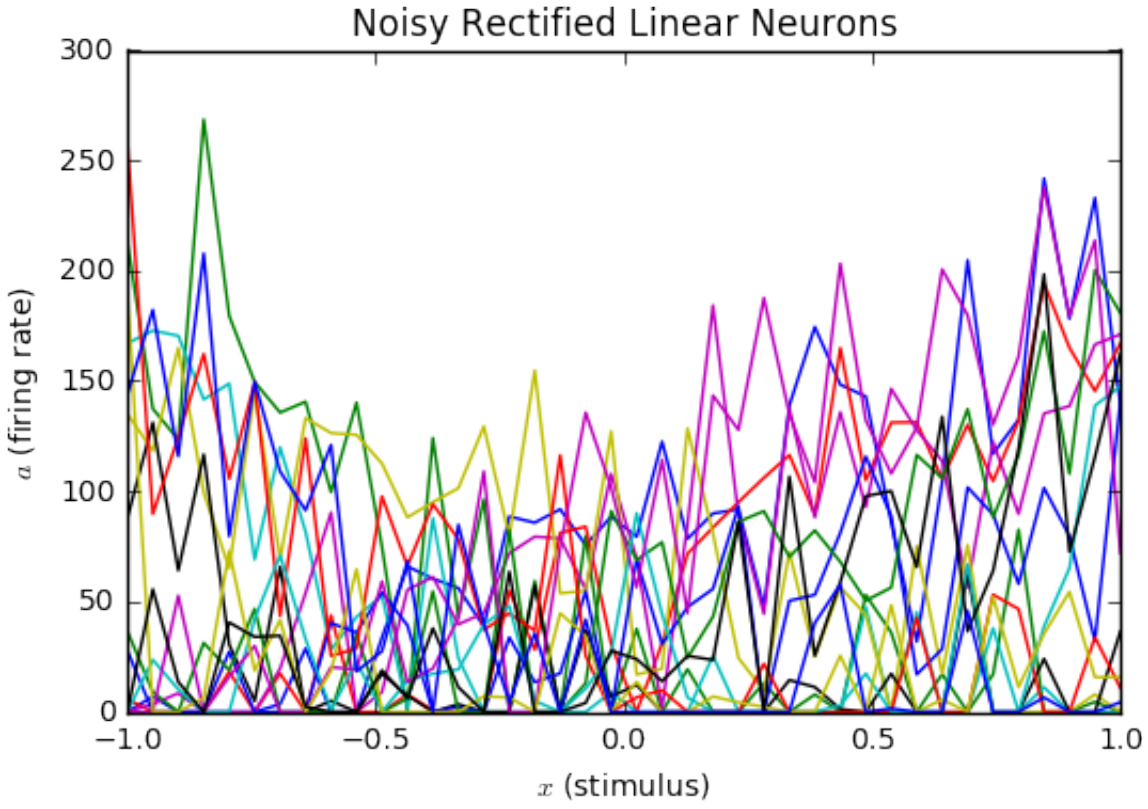


In [393]:
```
plot(X, Diff)
xlabel('$x$ (stimulus)')
ylabel('$\epsilon$')
title('Error in Decoded Stimulus');
```

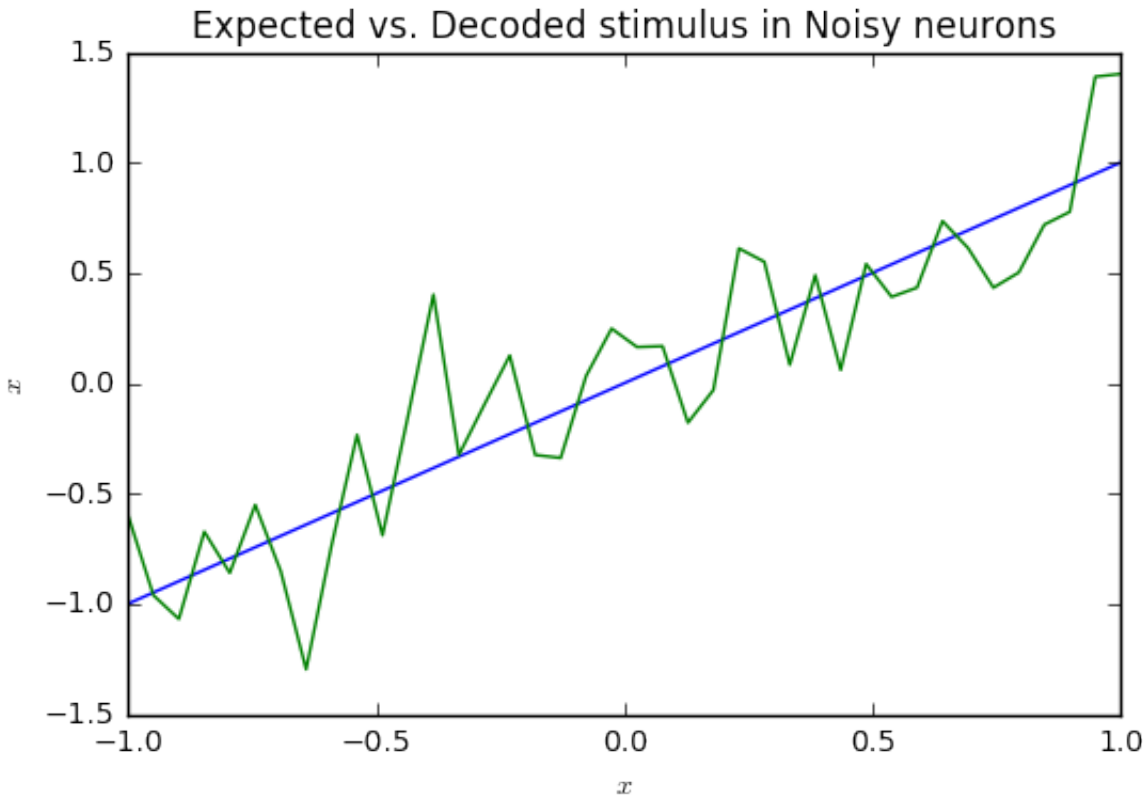


d. [1 mark] Now try decoding under noise. Add random normally distributed noise to a and decode again. The noise is a random variable with mean 0 and standard deviation of 0.2 times the maximum firing rate of all the neurons. Resample this variable for every different x value for every different neuron. Create all the same plots as in part c). Report the Root Mean Squared Error value.

In [394]:
```
def generateNoise(A,std_pct):
#     Anoise = A
    Anoise = A + np.random.normal(loc = 0, scale = 0.2 * np.max(A), size
    for i in range(0,A.shape[0] - 1): #for each neuron
        for j,x in enumerate(X): # for each step in x
            Anoise[i][j] = max(Anoise[i][j], 0)
    return Anoise

A_noise = generateNoise(A,0.2)
for i in range(0,A.shape[0] - 1): #for each neuron
    plot(X, A_noise[i])
xlabel('$x$ (stimulus)')
ylabel('$a$ (firing rate)')
title('Noisy Rectified Linear Neurons');
```

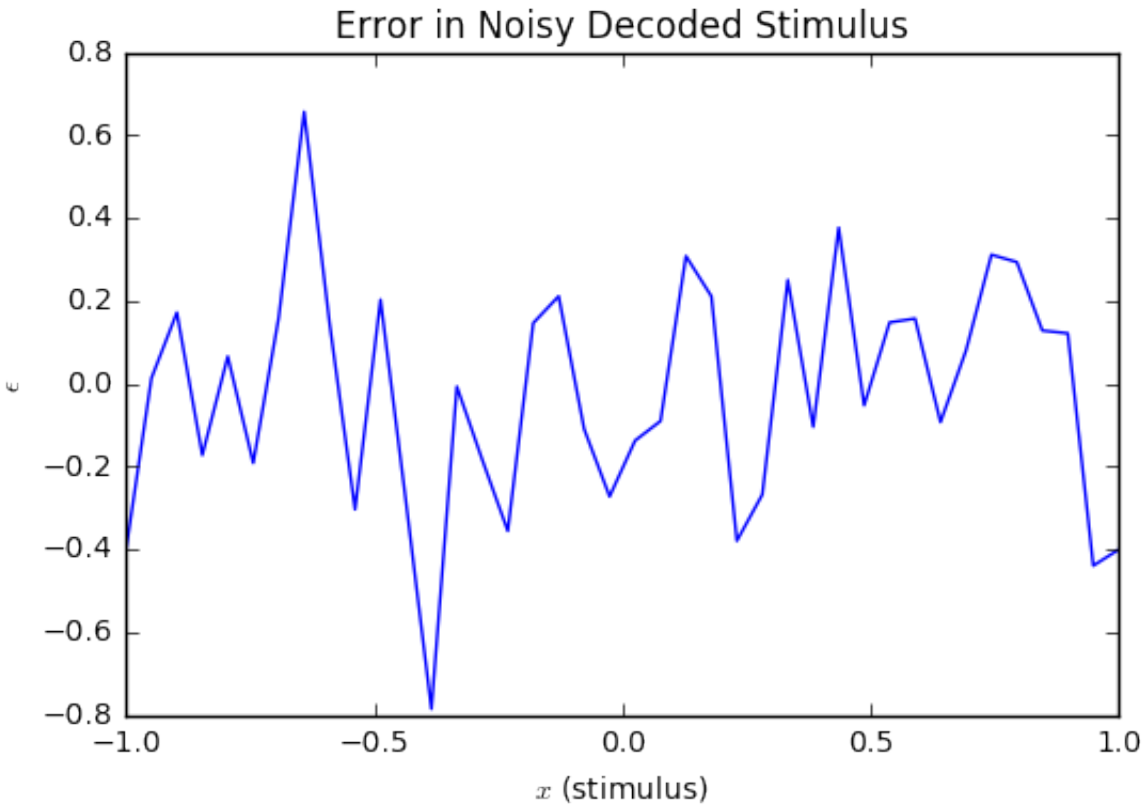


In [395]:
```
X_hat_noise = np.matmul(A_noise.transpose(),d)

plot(X, X)
plot(X, X_hat_noise)
xlabel('$x$')
ylabel('$x$')
title('Expected vs. Decoded stimulus in Noisy neurons');
```



In [396]:
```
RMS2, Diff_noise = getRMS(X, X_hat_noise)
print "RMS error = " + str(float(RMS2))

plot(X, Diff_noise)
xlabel('$x$ (stimulus)')
ylabel('$\epsilon$')
title('Error in Noisy Decoded Stimulus');
```

```
RMS error = 0.279040811169
```

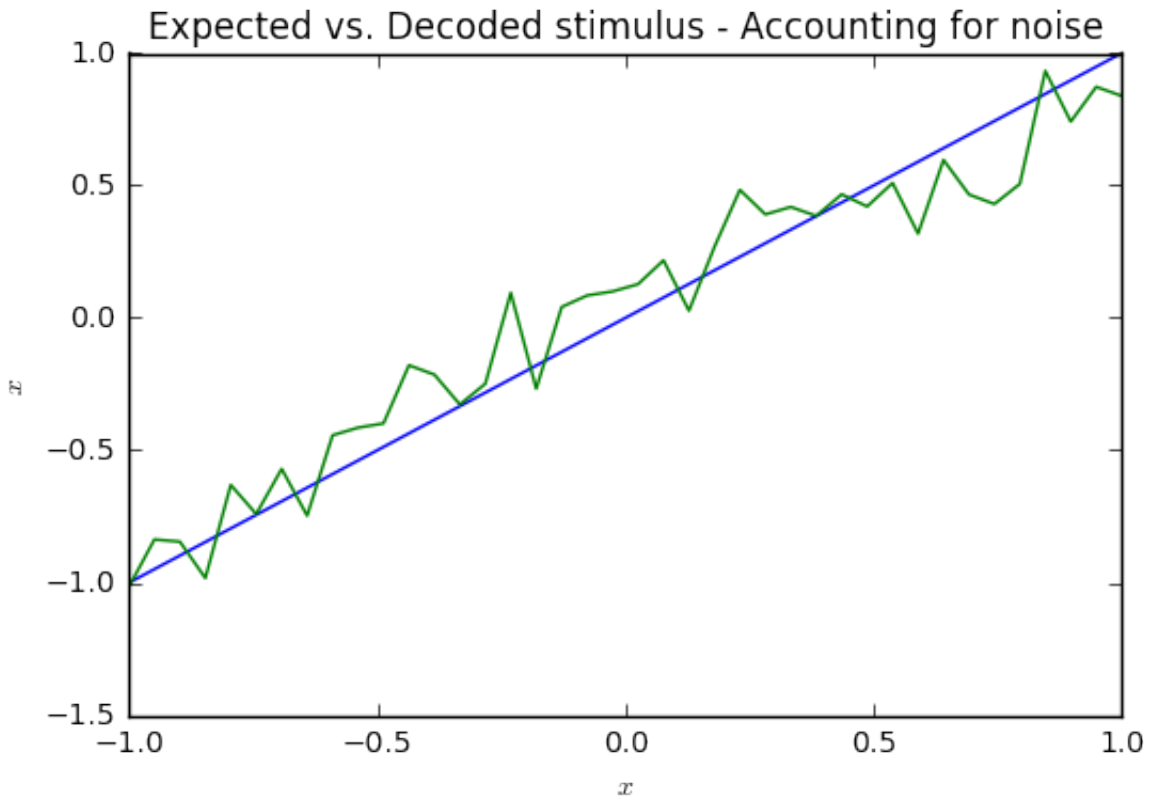


e. [1 mark] Recompute the decoders d_i taking noise into account (as shown in class). Show how these decoders behave when decoding both with and without noise added to a by making the same plots as in c) and d). Report the RMSE for both cases.

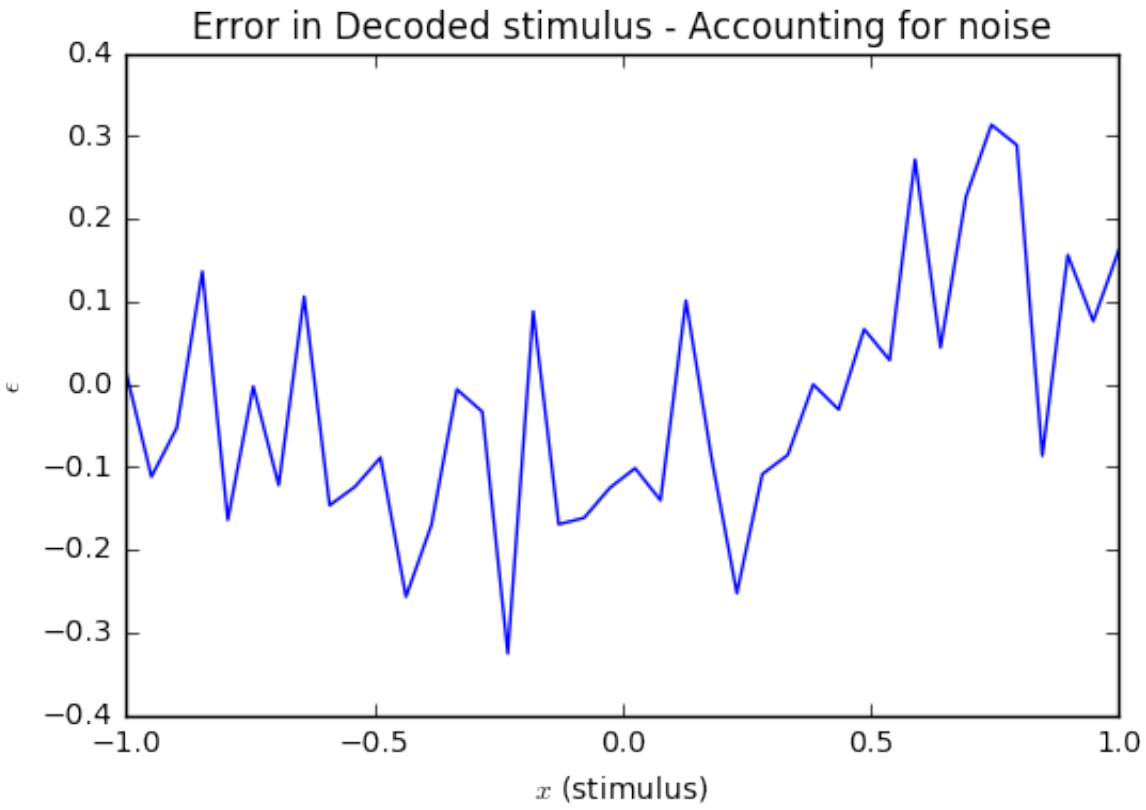
- As in the previous question, σ is 0.2 times the maximum firing rate of all the neurons.

In [397]:
```
def decodeWithNoise(A):
    Ypsilon = stepSize * np.matmul(A,X)
    Gamma = stepSize * np.matmul(A, np.transpose(A))
    Var = np.power(0.2 * np.max(A),2) * np.identity(A.shape[0])
    Gamma += Var
    Gamma_inv = np.linalg.inv(Gamma)
    d = np.dot(Gamma_inv, Ypsilon)
    return d

d_noise = decodeWithNoise(A_noise)

X_hat_noise = np.matmul(A_noise.transpose(),d_noise)

RMS3, Diff_noise = getRMS(X, X_hat_noise)
print "RMS error = " + str(float(RMS3))

plot(X, X)
plot(X, X_hat_noise)
xlabel('$x$')
ylabel('$x$')
title('Expected vs. Decoded stimulus - Accounting for noise');
```

```
RMS error = 0.151855652949
```

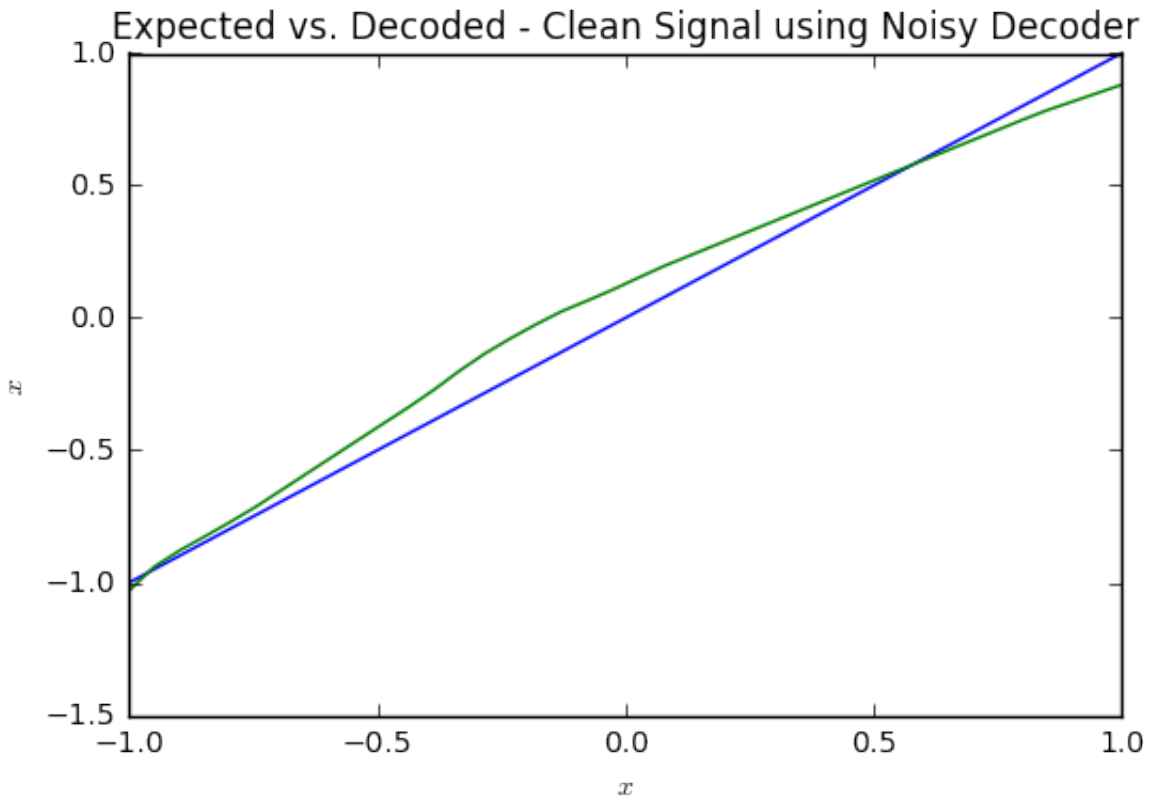


In [398]:
```
plot(X, Diff_noise)
xlabel('$x$ (stimulus)')
ylabel('$\epsilon$')
title('Error in Decoded stimulus - Accounting for noise');
```

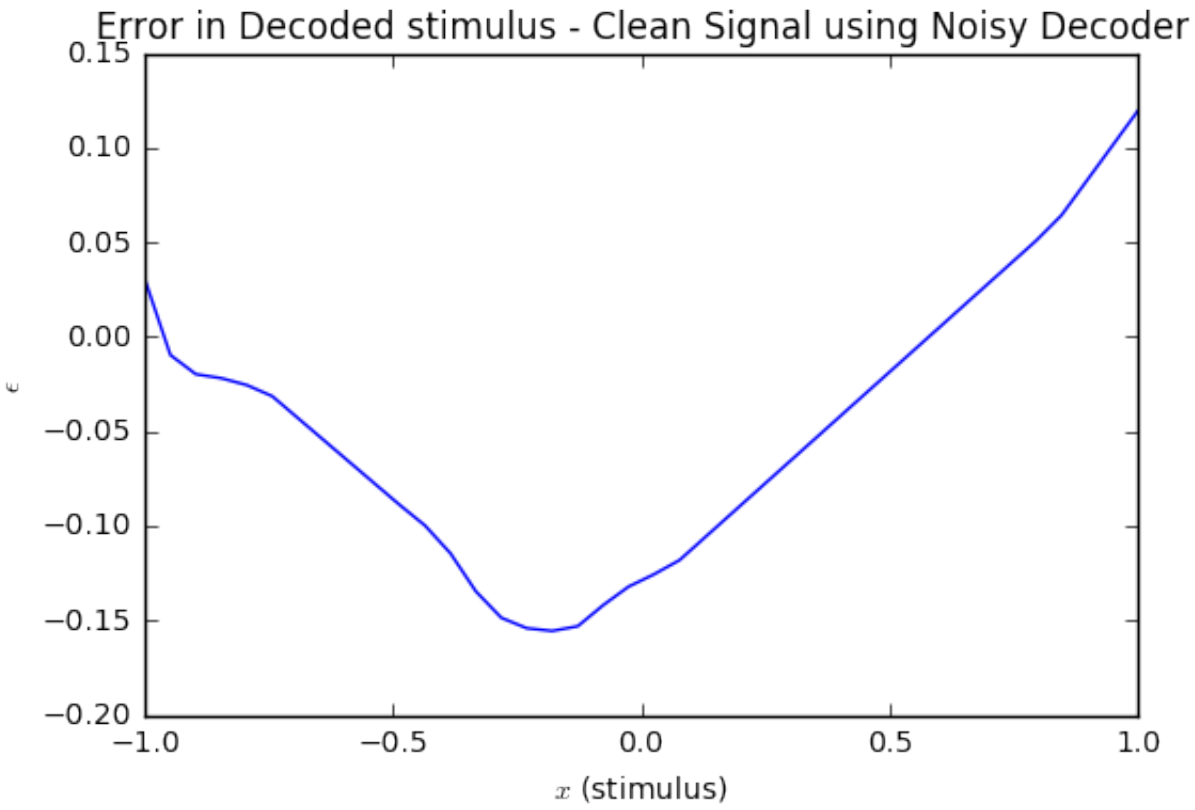


In [399]:
```
X_hat_clean = np.matmul(A.transpose(),d_noise)

RMS4, Diff_noise = getRMS(X, X_hat_clean)
print "RMS error = " + str(float(RMS4))

plot(X, X)
plot(X, X_hat_clean)
xlabel('$x$')
ylabel('$x$')
title('Expected vs. Decoded - Clean Signal using Noisy Decoder');
```

```
RMS error = 0.0877282297659
```



In [400]:
```
plot(X, Diff_noise)
xlabel('$x$ (stimulus)')
ylabel('$\epsilon$')
title('Error in Decoded stimulus - Clean Signal using Noisy Decoder');
```



f. [1 mark] Show a 2x2 table of the four RMSE values reported in parts c), d), and e). This should show the effects of adding noise and whether or not the decoders d are computed taking noise into account. Write a few sentences commenting on what the table shows.

In [401]:
```
print "Clean Neurons, Original Decoder \nRMS = " + str(RMS1) + "\n"
print "Noisy Neurons, Original Decoder \nRMS = " + str(RMS2) + "\n"
print "Noisy Neurons, Noisy Decoder \nRMS = " + str(RMS3) + "\n"
print "Clean Neurons, Noisy Decoder \nRMS = " + str(RMS4) + "\n"
```

```
Clean Neurons, Original Decoder
RMS = 0.000755688588185

Noisy Neurons, Original Decoder
RMS = 0.279040811169

Noisy Neurons, Noisy Decoder
RMS = 0.151855652949

Clean Neurons, Noisy Decoder
RMS = 0.0877282297659
```

This table shows that the lowest RMS error always comes from the noiseless set of neurons. When decoding a noisy signal though, the best RMSE comes from the decoder that takes that noise into account. It also shows that when trying to decode the noiseless neurons using the noisy decoder, we get a worse result than using the original decoder.

1.2) Exploring sources of error

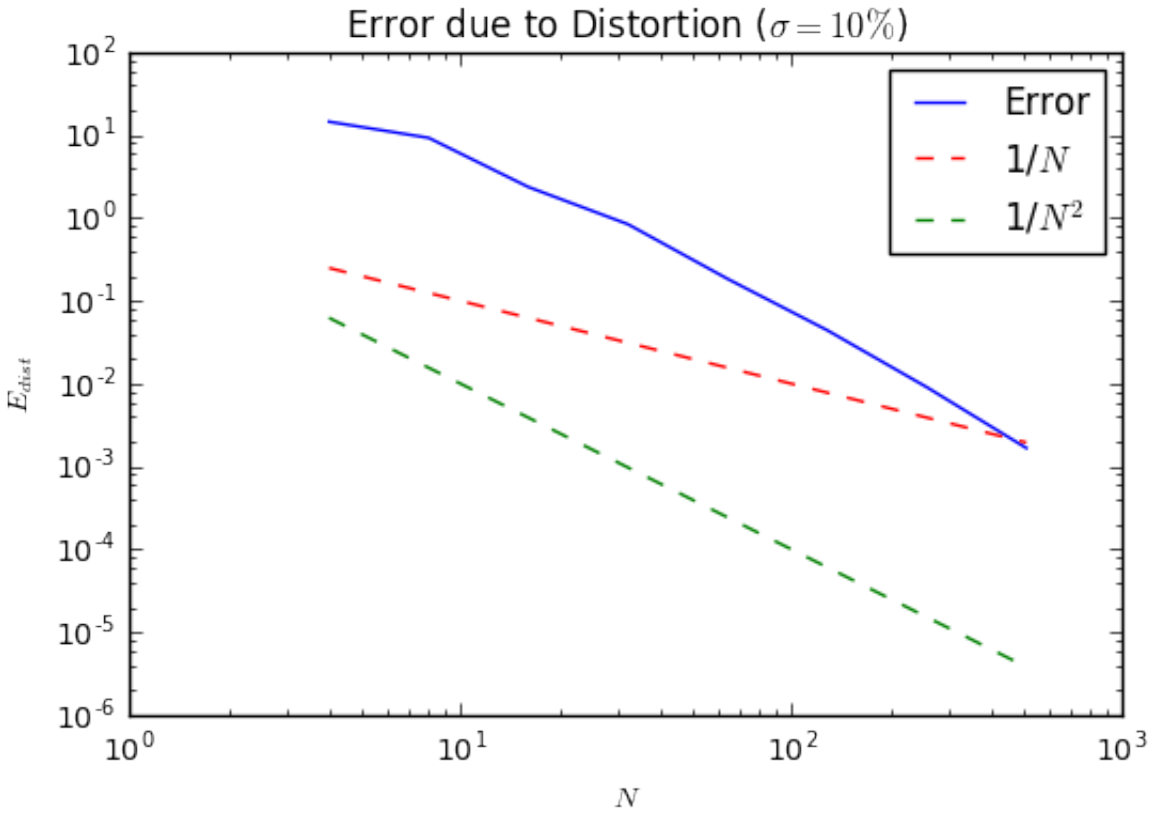
Use the program you wrote in 1.1 to examine the sources of error in the representation.

a. [2 marks] Plot the error due to distortion E_{dist} and the error due to noise E_{noise} as a function of N, the number of neurons. Use the equation with those two parts as your method (2.9 in the book). Generate two different loglog plots (one for each type of error) with N values of [4, 8, 16, 32, 64, 128, 256, 512] (and more, if you would like). For each N value, do at least 5 runs and average the results. For each run, different α, J^{bias}, and e values should be generated for each neuron. Compute d under noise, with σ equal to 0.1 times the maximum firing rate. Show visually that the errors are proportional to $1/N$ or $1/N^2$ (see figure 2.6 in the book).

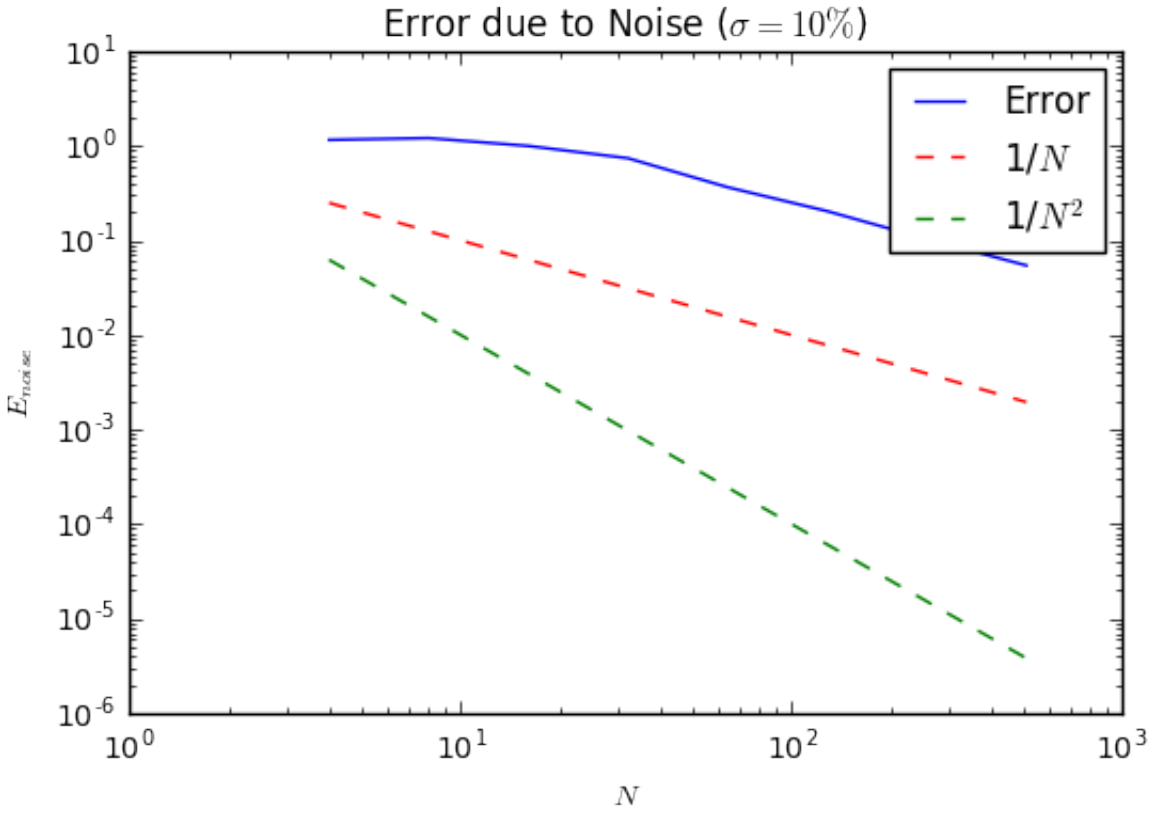
In [402]:
```
def getErrorSources(A):
    d = decodeWithNoise(A)
    X_hat = np.matmul(A.transpose(),d)
    Edist = 0.5 * np.sum((X - X_hat)**2)
    var = 0.2 * np.max(A)**2
    Enoise = var * np.sum(d**2)
    return Edist, Enoise

N = [4, 8, 16, 32, 64, 128, 256, 512]
E_dist = np.zeros(len(N))
E_noise = np.zeros(len(N))

for i,n in enumerate(N):
    print "Running n = " + str(n)
    Edist_i = [0] * 5
    Enoise_i = [0] * 5
    for j in range(0,5):
        A_ = initLinearNeurons(n,X)
        Anoise = generateNoise(A_,0.1)
        Ed, En = getErrorSources(Anoise)
        Edist_i += Ed
        Enoise_i += En
    E_dist[i] = np.mean(Edist_i)
    E_noise[i] = np.mean(Enoise_i)
print "OK"
```

```
Running n = 4
Running n = 8
Running n = 16
Running n = 32
Running n = 64
Running n = 128
Running n = 256
Running n = 512
OK
```

In [403]:
```
N_inv = [0] * len(N)
N_inv2 = [0] * len(N)
for i,m in enumerate(N):
    N_inv[i] = (m**(-1))
    N_inv2[i] = (m**(-2))

loglog(N, E_dist, label = "Error")
loglog(N,N_inv, 'r--', label="1/$N$")
loglog(N,N_inv2, 'g--', label="1/$N^2$")
plt.legend()
xlabel('$N$')
ylabel('$E_{dist}$')
title('Error due to Distortion ($\sigma = 10\%$)');
```



In [404]:
```
loglog(N, E_noise,'b', label="Error")
loglog(N,N_inv, 'r--', label="1/$N$")
loglog(N,N_inv2, 'g--', label="1/$N^2$")
plt.legend()
xlabel('$N$')
ylabel('$E_{noise}$')
title('Error due to Noise ($\sigma = 10\%$)');
```

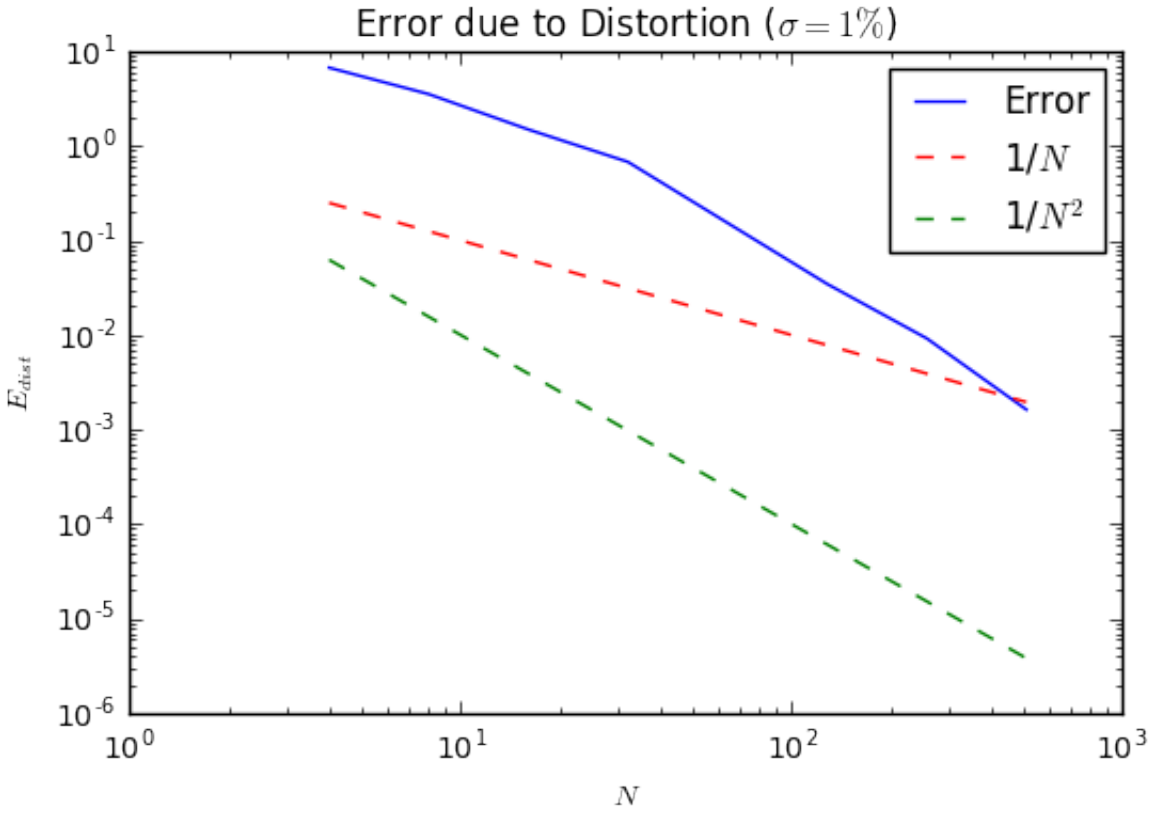


b. [1 mark] Repeat part a) with σ equal to 0.01 times the maximum firing rate.

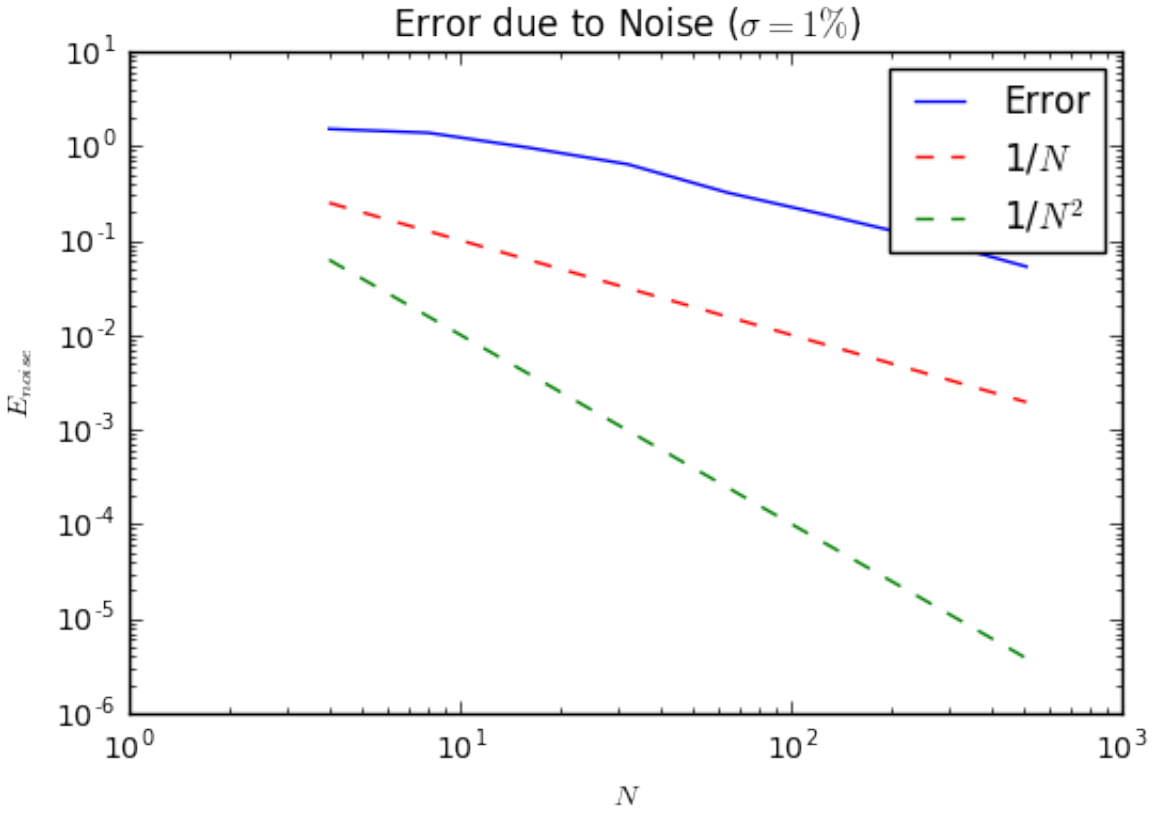
In [405]:
```
E_dist2 = np.zeros(len(N))
E_noise2 = np.zeros(len(N))

for i,n in enumerate(N):
    print "Running n = " + str(n)
    Edist_i = [0] * 5
    Enoise_i = [0] * 5
    for j in range(0,5):
        A_ = initLinearNeurons(n,X)
        Anoise = generateNoise(A_,0.01)
        Ed, En = getErrorSources(Anoise)
        Edist_i += Ed
        Enoise_i += En
    E_dist2[i] = np.mean(Edist_i)
    E_noise2[i] = np.mean(Enoise_i)
print "OK"
```

```
Running n = 4
Running n = 8
Running n = 16
Running n = 32
Running n = 64
Running n = 128
Running n = 256
Running n = 512
OK
```

In [406]:
```
loglog(N, E_dist2, label = "Error")
loglog(N,N_inv, 'r--', label="1/$N$")
loglog(N,N_inv2, 'g--', label="1/$N^2$")
plt.legend()
xlabel('$N$')
ylabel('$E_{dist}$')
title('Error due to Distortion ($\sigma = 1\%$)');
```



In [407]:
```
loglog(N, E_noise2,'b', label = "Error")
loglog(N,N_inv, 'r--', label="1/$N$")
loglog(N,N_inv2, 'g--', label="1/$N^2$")
plt.legend()
xlabel('$N$')
ylabel('$E_{noise}$')
title('Error due to Noise ($\sigma = 1\%$)');
```



c. [1 mark] What does the difference between the graphs in a) and b) tell us about the sources of error in neural populations?

> The difference between the graphs in a) and b) tells us that as we increase the number of neurons in a population, the error due to noise supercedes the error due to distortion.

1.3) Leaky Integrate-and-Fire neurons

Change the code to use the LIF neuron model:

$$a_i = \begin{cases} \frac{1}{\tau_{ref} - \tau_{RC} ln(1 - \frac{1}{J})} & \text{if } J > 1 \\ 0 & \text{otherwise} \end{cases}$$

Finding α and J_{bias}

$$a_i = \frac{1}{\tau_{ref} - \tau_{RC} ln(1 - \frac{1}{J(x)})}$$

where

$$J(x) = \alpha x + J_{bias}$$

we know that the maximum firing rate occurs at

$$a_{max} = a(x = 1)$$

so setting

$$J(1) = \alpha + J_{bias}$$

we can solve for $\alpha + J_{bias}$

$$a = \frac{1}{\tau_{ref} - \tau_{RC} ln(1 - \frac{1}{\alpha + J_{bias}})}$$

$$\alpha + J_{bias} = \frac{1}{1 - e^{\frac{a_{max}\tau_{ref} - 1}{a_{max}\tau_{RC}}}}$$

We also know that the x-intercept occurs on the $J(X)$ curve where $J = 0$, and J_{bias} will occur where $x = 0$. Using these values, we can calculate

$$\alpha = \frac{J_{bias}}{x_{int}}$$

or

$$J_{bias} = \alpha x_{int}$$

Substituting this into our equation for $\alpha + J_{bias}$ gives

$$\alpha + \alpha x_{int} = \frac{1}{1 - e^{\frac{a_{max}\tau_{ref} - 1}{a_{max}\tau_{RC}}}}$$

$$\alpha = \frac{1}{1 - e^{\frac{a_{max}\tau_{ref} - 1}{a_{max}\tau_{RC}}}} \frac{1}{1 + x_{int}}$$

and we calculate J_{bias} using the equation we found before:

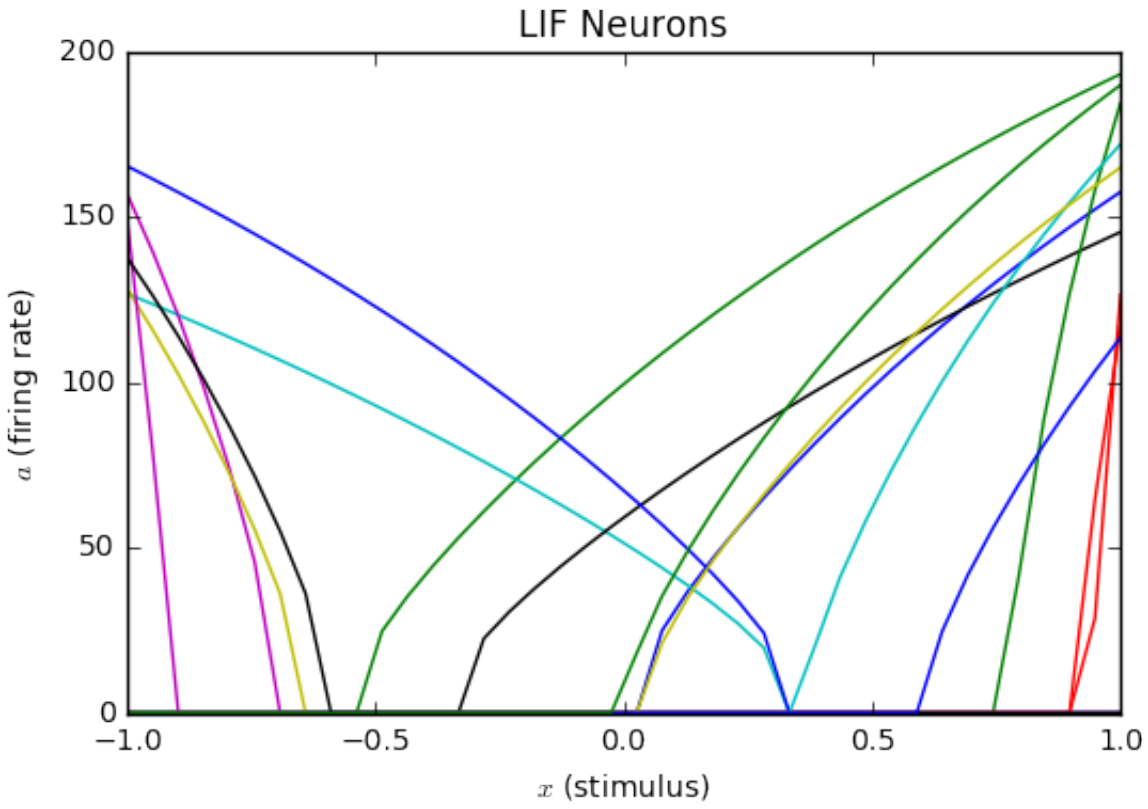
$$J_{bias} = \alpha x_{int}$$

In [408]:
```
class LIFNeuron(object):
    alpha = 0
    jbias = 0
    encodextr = 0

    def __init__(self, tau_ref, tau_rc):
        m = np.random.uniform(low = 100, high = 200) # a @ x = 1
        xint = np.random.uniform(low = -0.95,high = 0.95) # x @ a = 0
        self.encoder = np.random.uniform(low = -1, high = 1)
        if self.encoder > 0: self.encoder = 1
        else: self.encoder = -1

        # a @ x=1 == max
        self.alpha = calcAlpha(tau_ref, tau_rc, xint, m)
        self.jbias = self.alpha * xint


def initLIFNeurons(n,X, tau_ref, tau_rc):
    A = np.zeros([n,len(X)])
    for i in range(0,n):
        neu = LIFNeuron(tau_ref, tau_rc)
        for k,x in enumerate(X): # for each step dx
            j = (neu.encoder * neu.alpha * x + neu.jbias)
            a = LIFcurve(tau_ref, tau_rc, j)
            A[i][k] = a
    return A

def LIFcurve(tau_ref, tau_rc, j):
    return (tau_ref - (tau_rc*np.log(1-j**(-1))))**(-1) if (j > 1) else 0

def calcAlpha(tau_ref, tau_rc, xint, m):
    return (1/(1+xint) * 1/(1-np.exp((m*tau_ref - 1)/(m*tau_rc))))
```

a. [1 mark] Generate the same plot as 1.1a). Use $\tau_{ref} = 0.002$s and $\tau_{RC} = 0.02$s.

- Note that you will need to compute new α and J^{bias} values that will achieve the desired tuning curves (uniform distribution of x-intercepts between -1 and 1, and maximum firing rates between 100Hz and 200Hz). Since you know two points on the tuning curve (the x-intercept and the point where it hits maximum firing), this gives you 2 equations and 2 unknowns, so you can find α and J^{bias} by substituting and rearranging.

In [409]:
```
n = 16
stepSize = 0.05
X = np.linspace(-1,1,2/stepSize)
A_lif = initLIFNeurons(n,X, tau_ref = 0.002, tau_rc = 0.02)

for i in range(0,n): #for each neuron
    plot(X, A_lif[i])
xlabel('$x$ (stimulus)')
ylabel('$a$ (firing rate)')
title('LIF Neurons');
```



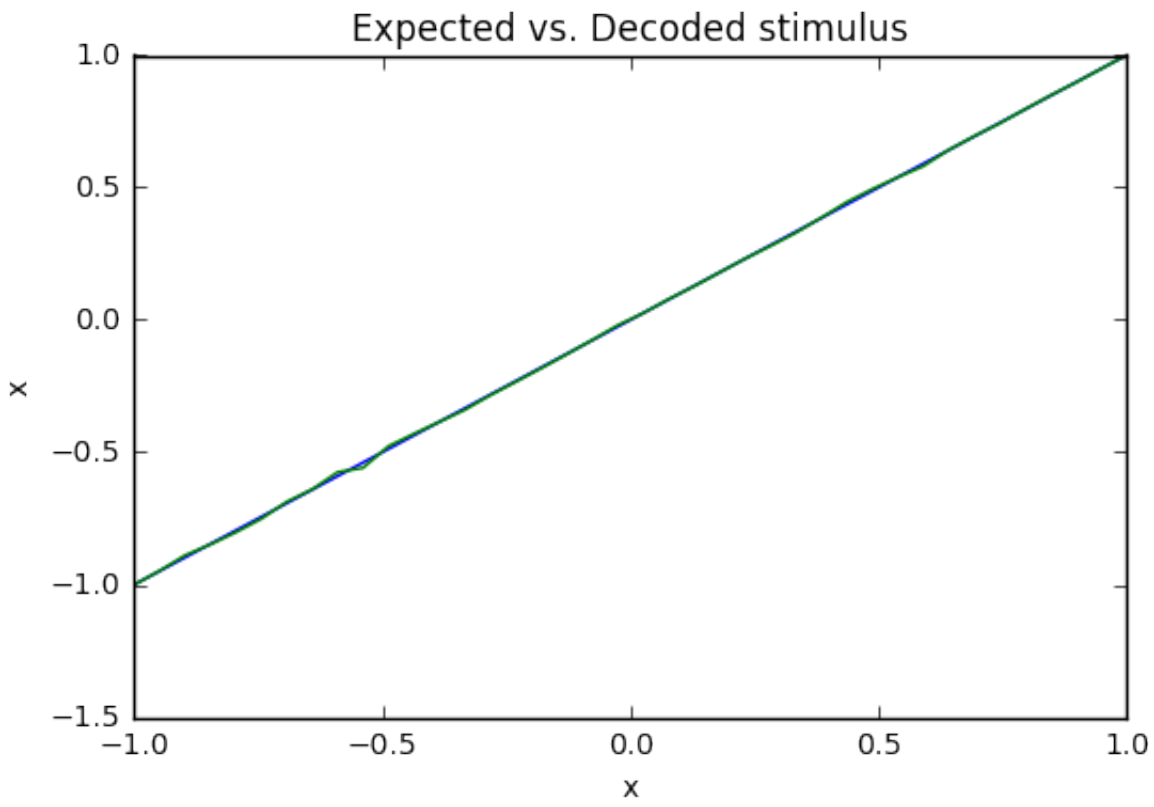
b. [2 marks] Generate the same plots as 1.1e), and report the RMSE for both.

In [410]:
```
d_lif = decodeNeurons(A_lif)
print "Decoders: \n\n d = " + str(d_lif)
```

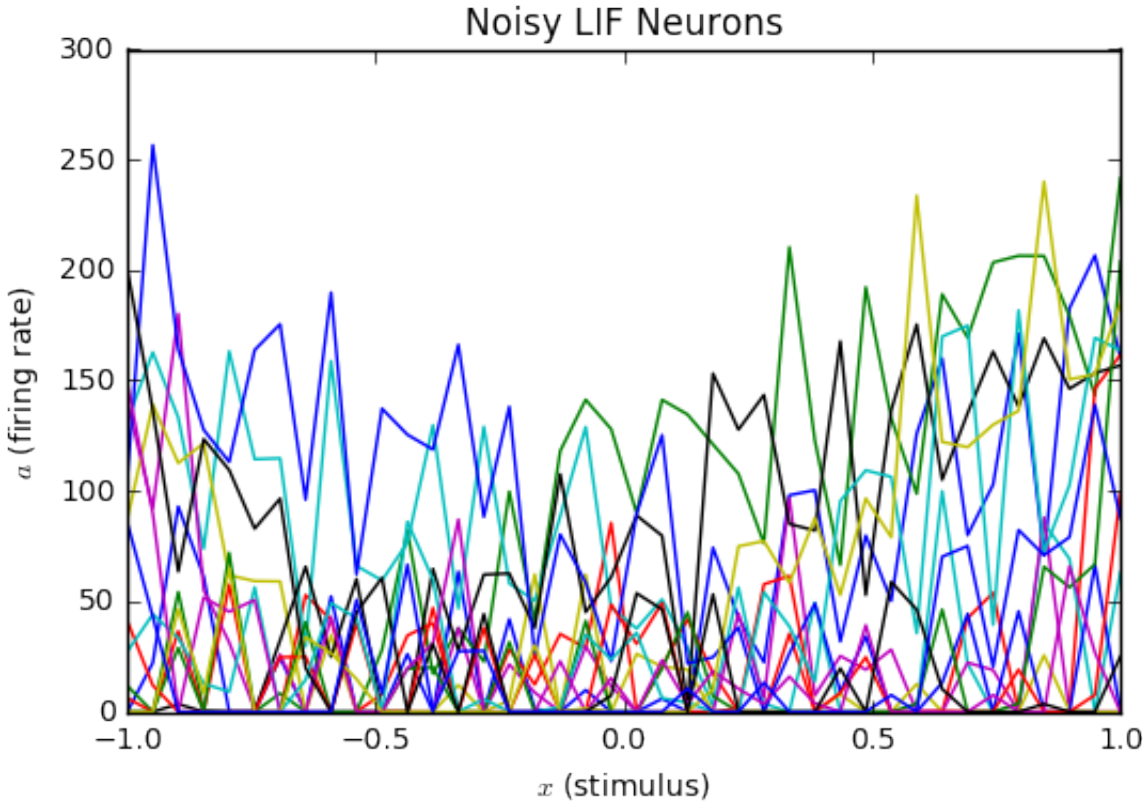
```
Decoders:

 d = [  1.47761911e-03   2.59857151e-03   9.83265170e-05   4.66290401e
-02
  -5.56050843e-04  -2.04992075e-04   4.19221057e-04  -3.97265974e-02
   2.05071102e-04   3.84289288e-05   2.44005224e-03  -2.66462812e-04
  -1.49964699e-03  -1.39452470e-03   8.94328351e-04  -6.47828852e-04]
```

In [411]:
```
X_hat_lif = np.matmul(A_lif.transpose(), d_lif)

RMS_lif, diff = getRMS(X, X_hat_lif)
print "RMS error = " + str(RMS_lif)
plot(X, X)
plot(X, X_hat_lif)
xlabel('x')
ylabel('x')
title('Expected vs. Decoded stimulus');
```

```
RMS error = 0.00580424056215
```

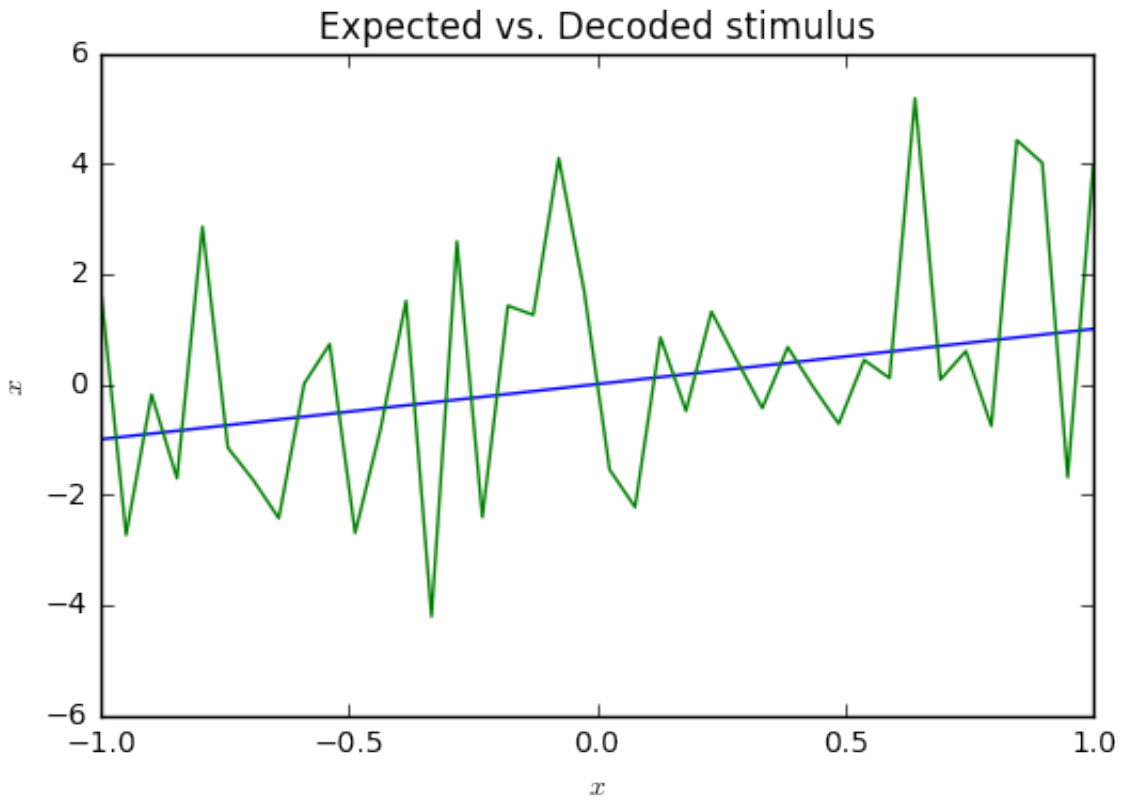


In [412]:
```
A_noise_lif = generateNoise(A_lif,0.2)
for i in range(0,A_lif.shape[0] - 1): #for each neuron
    plot(X, A_noise_lif[i])
xlabel('$x$ (stimulus)')
ylabel('$a$ (firing rate)')
title('Noisy LIF Neurons');
```



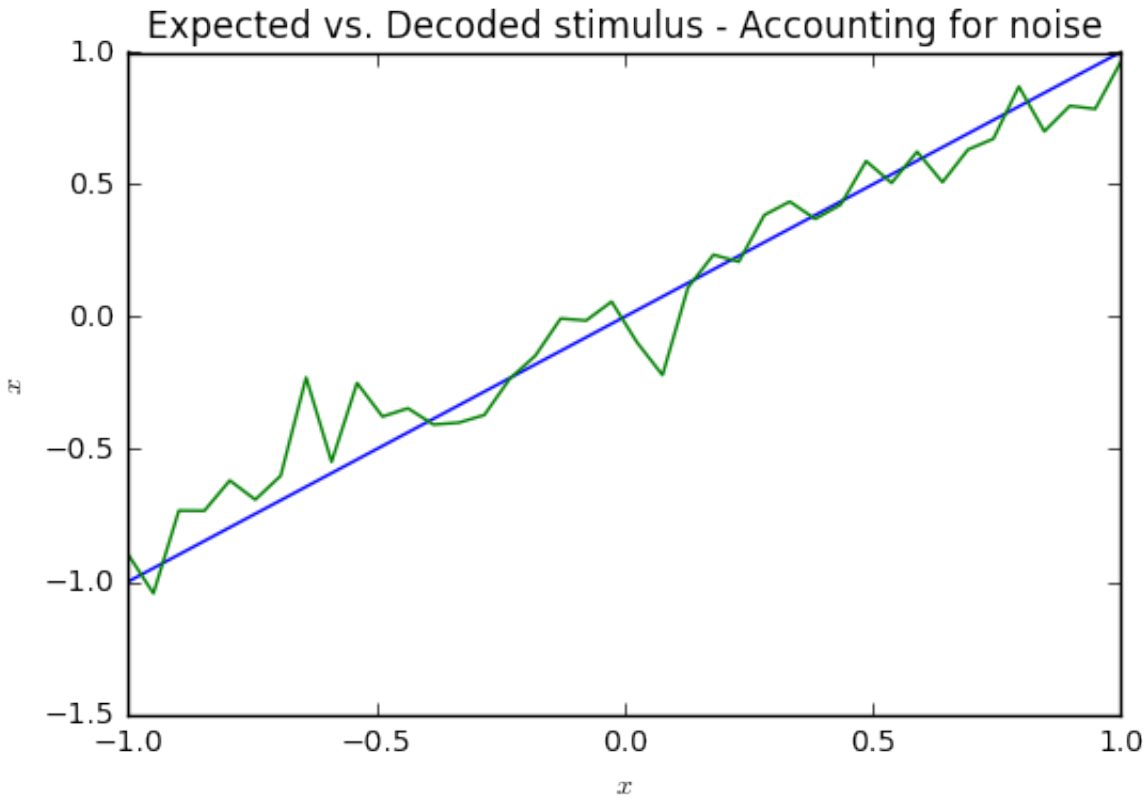
In [413]:
```
X_hat_noise_lif = np.matmul(A_noise_lif.transpose(),d_lif)

RMS_noise_lif, diff = getRMS(X, X_hat_noise_lif)
print "RMS error = " + str(RMS_noise_lif)
plot(X, X)
plot(X, X_hat_noise_lif)
xlabel('$x$')
ylabel('$x$')
title('Expected vs. Decoded stimulus');
```

```
RMS error = 2.04425525671
```



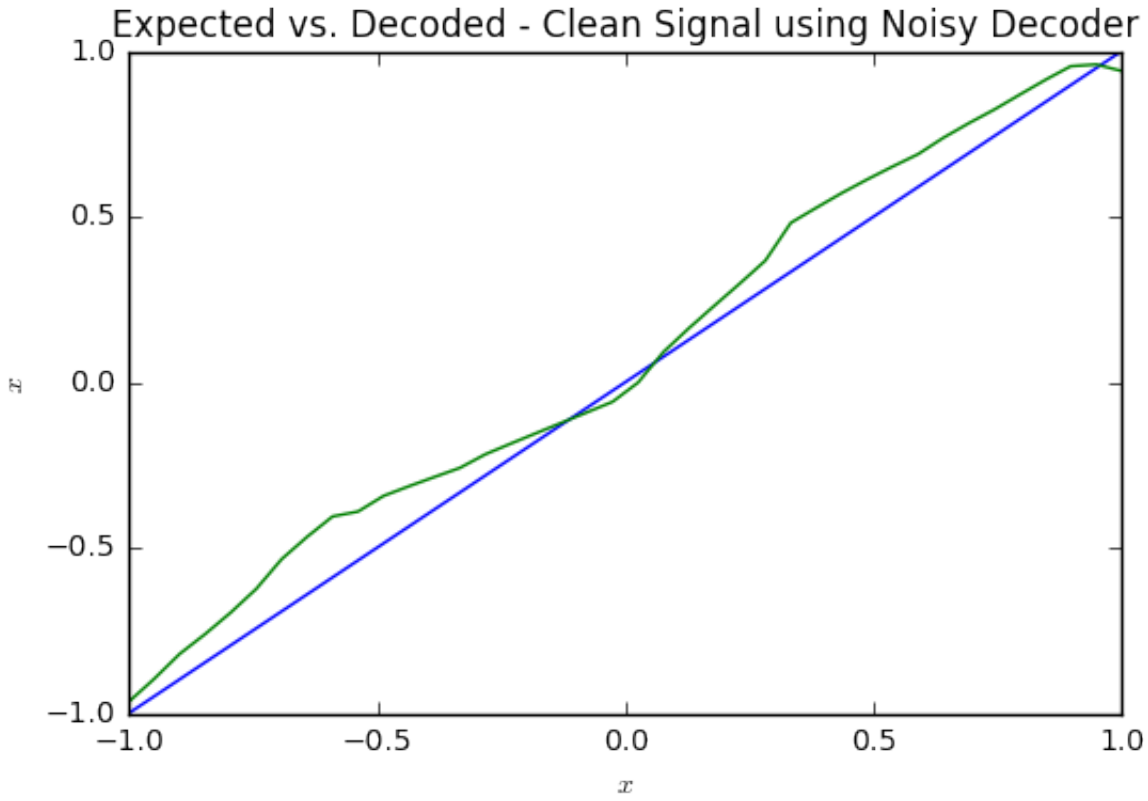
In [414]:
```
d_noise_lif = decodeWithNoise(A_noise_lif)

X_hat_noise_lif2 = np.matmul(A_noise_lif.transpose(),d_noise_lif)

RMS_noise_lif2, Diff_noise = getRMS(X, X_hat_noise_lif2)
print "RMS error = " + str(float(RMS_noise_lif2))
plot(X, X)
plot(X, X_hat_noise_lif2)
xlabel('$x$')
ylabel('$x$')
title('Expected vs. Decoded stimulus - Accounting for noise');
```

```
RMS error = 0.127501921334
```



In [415]:
```
X_hat_clean_lif = np.matmul(A_lif.transpose(),d_noise_lif)

RMS_clean_lif, Diff_noise = getRMS(X, X_hat_clean_lif)
print "RMS error = " + str(float(RMS_clean_lif))

plot(X, X)
plot(X, X_hat_clean_lif)
xlabel('$x$')
ylabel('$x$')
title('Expected vs. Decoded - Clean Signal using Noisy Decoder');
```

```
RMS error = 0.0957241406419
```



2) Representation of Vectors

2.1) Vector tuning curves

a. [1 mark] Plot the tuning curve of an LIF neuron whose 2D preferred direction vector is at an angle of $\theta = -\pi/4$, has an x-intercept at the origin (0,0), and has a maximum firing rate of 100Hz.

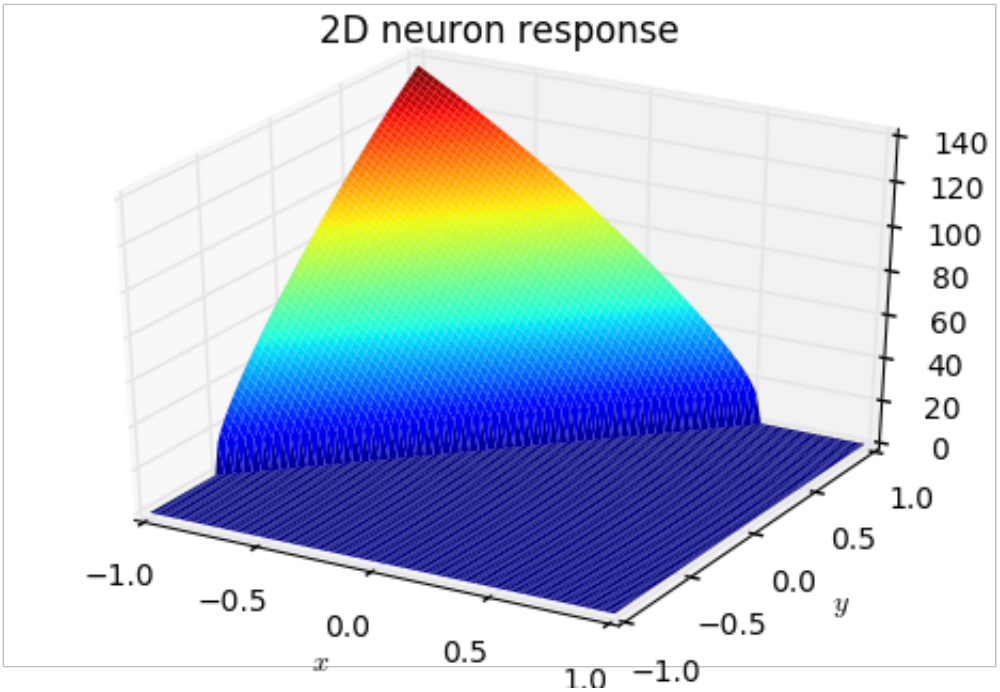
- Remember that $J = \alpha e \cdot x + J^{bias}$, and both x and e are 2D vectors.
- This is a 3D plot similar to figure 2.8a in the book.
- In the scalar case (that you did in question 1.1a), the maximum firing rate occurred when $x = 1$ for neurons with $e = 1$ and at $x = -1$ for neurons with

$e = -1$. Of course, if the graph in 1.1a was extended to $x > 1$ (or $x < -1$), neurons would start firing faster than their maximum firing rate. Similarly, here the "maximum firing rate" means the firing rate when $x = e$. This should allow you to reuse your code from 1.3a) to compute α and J^{bias} for a desired maximum firing rate and x-intercept.

- To generate 3D plots in MATLAB, see here (http://www.mathworks.com/help/matlab/learn_matlab/creating-mesh-and-surface-plots.html)
- To generate 3D plots in Python, see here (http://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html)

In [416]:
```
class TwoD_LIFNeuron(object):
    alpha = np.zeros(2)
    jbias = np.zeros(2)
    encoder = np.zeros(2)

    def __init__(self, tau_ref, tau_rc):
        m = 100  #np.random.uniform(low = 100, high = 200) # a @ x = 1
        xint = np.matrix([0,0])  #np.random.uniform(low = -0.95,high = 0.
        self.encoder = np.matrix([1.0, -1.0])
        self.encoder = self.encoder/np.linalg.norm(self.encoder)

        self.alpha = calcAlpha(tau_ref, tau_rc, np.linalg.norm(xint), m)
        self.jbias = self.alpha * np.linalg.norm(xint)

    def setEncoder(self, E):
#         self.__init__(self, tau_ref, tau_rc)
        self.encoder = E


def init2dLIFNeurons(n,X, tau_ref, tau_rc):
    Y = X
    A = np.zeros([n,len(X),len(Y)])
    J = np.zeros([n,len(X),len(Y)])
    for i in range(0,n):
        neu = TwoD_LIFNeuron(tau_ref, tau_rc)
        for xi,x in enumerate(X): # for each step dx
            for yi, y in enumerate(Y): #for each step dy
                v = np.matrix([x,y])
                j = neu.alpha * matmul(v, neu.encoder.transpose()) + neu.
                j = float(j)
                a = LIFcurve(tau_ref, tau_rc, j)
                A[i][xi][yi] = a
                J[i][xi][yi] = j
    return A,J
```

In [417]:
```
n = 1
dim = 2
stepSize = 0.05
X = np.linspace(-1,1,4/stepSize)
Y = np.linspace(-1,1,4/stepSize)
A_2d,J = init2dLIFNeurons(n,X, tau_ref = 0.002, tau_rc = 0.02)

X,Y = numpy.meshgrid(X, Y)

from mpl_toolkits.mplot3d.axes3d import Axes3D
fig = figure()
ax = fig.add_subplot(1, 1, 1, projection='3d')
p = ax.plot_surface(X, Y, A_2d[0],
                        linewidth=0, cstride=1, rstride=1, cmap=pylab.cm.
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")
ax.set_zlabel("$a$")
ax.set_title("2D neuron response");
```



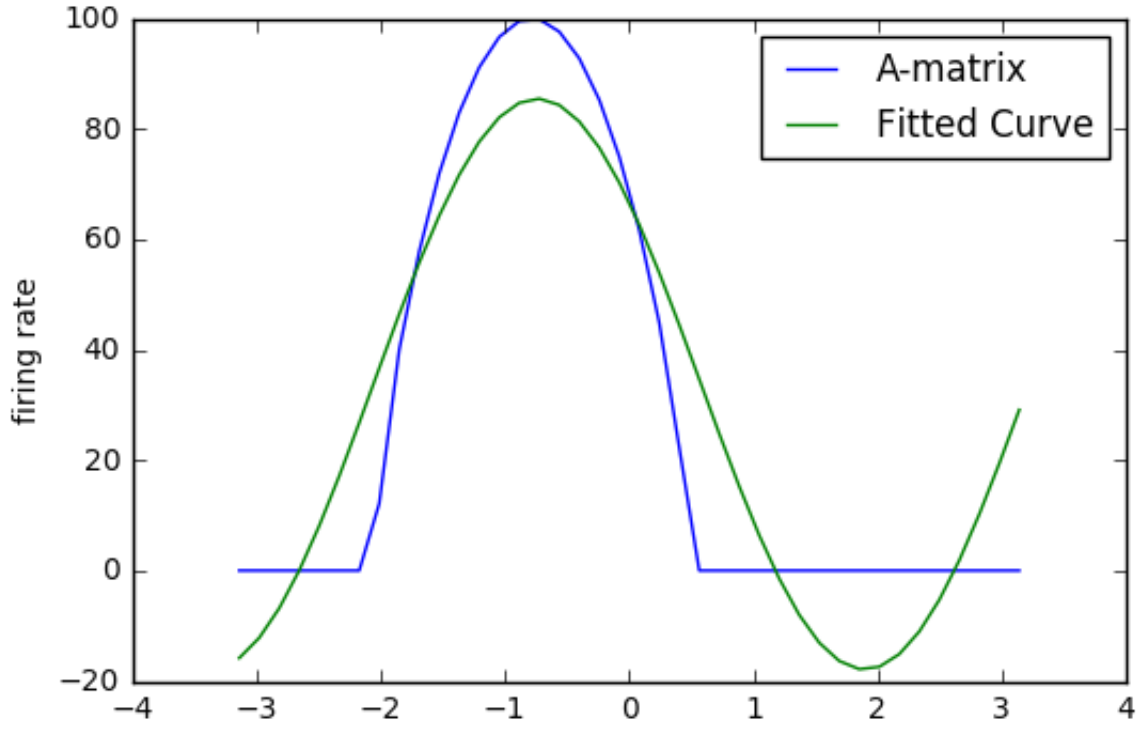
b. [1 mark] Plot the tuning curve for the same neuron as in a), but only considering the points around the unit circle. This will be similar to Figure 2.8b in the book. Fit a curve of the form $Acos(B\theta + C) + D$ to the tuning curve and plot it as well. What makes a cosine a good choice for this? Why does it differ from the ideal curve?

- To do curve fitting in MATLAB, see here (http://www.mathworks.com/help/optim/ug/lsqcurvefit.html).
- To do curve fitting in Python, see here (http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html).

In [418]:
```
X = np.linspace(-1,1,2/stepSize)
Y = np.linspace(-1,1,2/stepSize)
```

```
Theta = np.linspace(-np.pi,np.pi,2/stepSize)

def init2dLIFNeurons_angular(n,Theta,tau_ref,tau_rc):
    A = np.zeros([len(X)])
    neu = TwoD_LIFNeuron(tau_ref, tau_rc)
    for oi,o in enumerate(Theta): # for each step dx
        x = cos(o)
        y = sin(o)
        v = np.matrix([x,y])
        j = neu.alpha * matmul(v, neu.encoder.transpose()) + neu.jbias
        j = float(j)
        a = LIFcurve(tau_ref, tau_rc, j)
        A[oi] = a
    return A

def curveFit(Theta, A, B, C, D):
    return A * cos(B * Theta + C) + D

from scipy.optimize import curve_fit

A_theta = init2dLIFNeurons_angular(n,Theta,0.002,0.02)

popt, pcov = curve_fit(curveFit, Theta, A_theta)
print "Curve arguments: " + str(popt)

plot(Theta, A_theta, label = "A-matrix")
plot(Theta, curveFit(Theta, *popt), label = "Fitted Curve")
plt.legend()
xlabel("$theta$")
ylabel('firing rate');
```

```
Curve arguments: [ 51.68597073   1.19037048  -5.4007416   33.76410301]
```

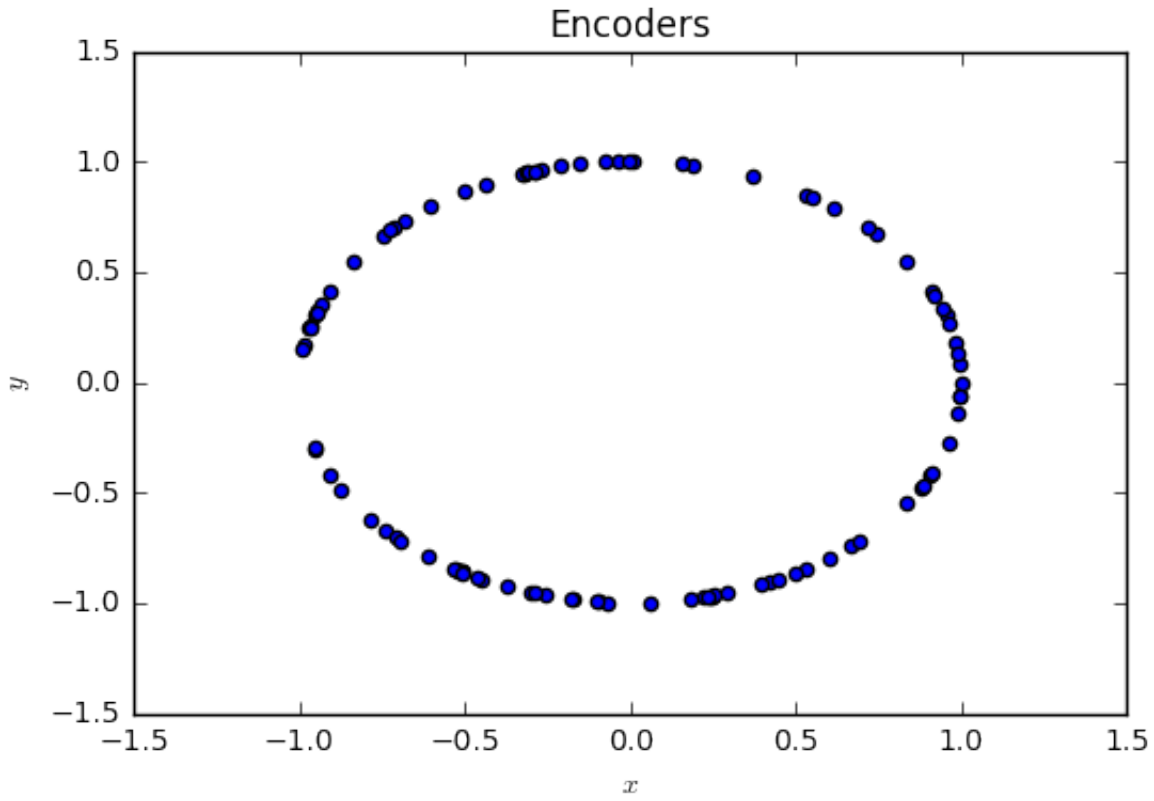


$theta$

2.2 Vector representation

a. [1 mark] Generate a set of 100 random unit vectors uniformly distributed around the unit circle. These will be the encoders e for 100 neurons. Plot these vectors.

In [419]:
```
def generateN_Encoders(N):
    Encoders = np.zeros([N,2])
    for e in range(0,len(Encoders)):
        th = np.random.uniform(0, 2*np.pi)
        x = cos(th)
        y = sin(th)
        Encoders[e][0] = x
        Encoders[e][1] = y
    return Encoders

n = 100
E = generateN_Encoders(n)
for e in range(0,len(E)):
    scatter(E[e][0],E[e][1])
title("Encoders")
xlabel("$x$")
ylabel("$y$");
```



b. [1 mark] Compute the optimal decoders. Use LIF neurons with the same properties as in question 1.3. When computing the decoders, take into account noise with σ as 0.2 times the maximum firing rate. Plot the decoders. How do these decoding vectors compare to the encoding vectors?

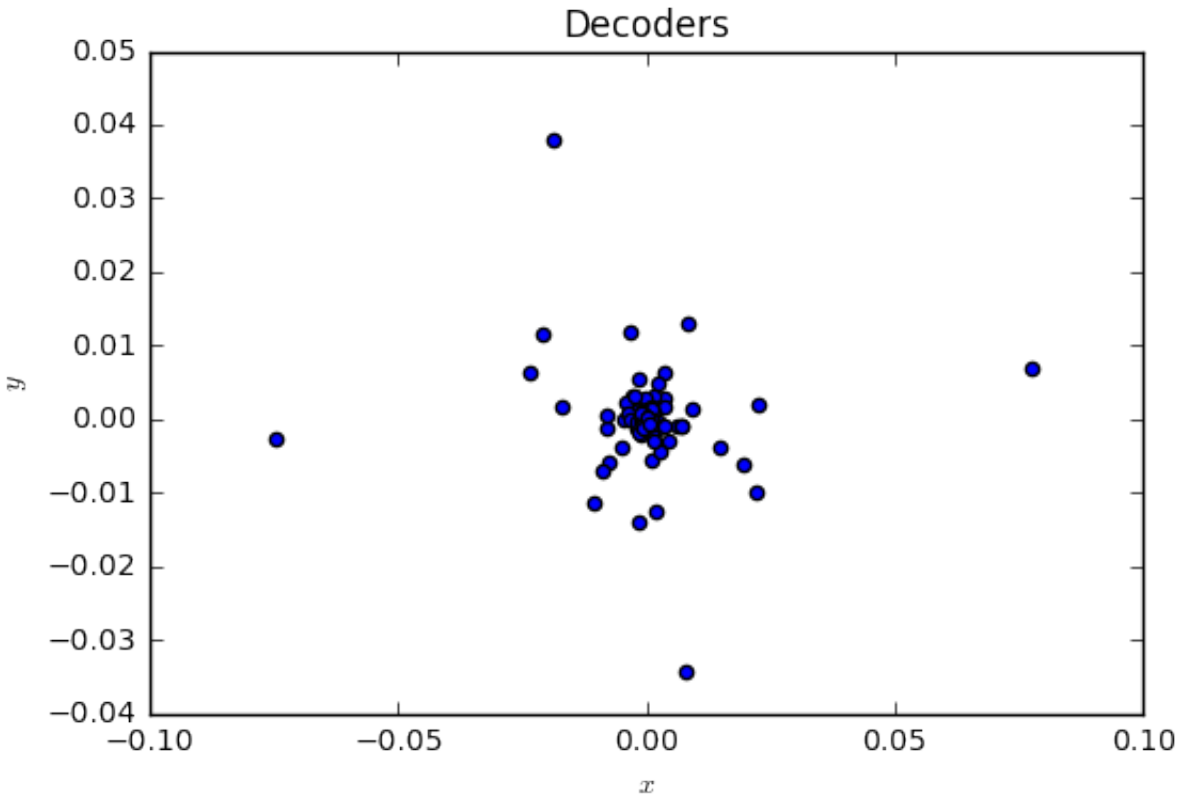
- Note that the decoders will also be 2D vectors.
- In the scalar case, you used x values between -1 and 1, with $dx = 0.05$. In this case, you can regularly tile the 2D x values ([1, 1], [1, 0.95], ... [-1, -0.95], [-1, 1]). Alternatively, you can just randomly choose 1600 different x values to sample.

In [420]:
```
X = np.linspace(-1,1,2/stepSize)
Y = np.linspace(-1,1,2/stepSize)
m = int(2/stepSize)

TwoD_Neurons = [None] * n
A_xy = np.zeros([n,m*m])

for i in range(0,n):
    neu = TwoD_LIFNeuron(0.002, 0.02)
    neu.setEncoder(E[i])
    TwoD_Neurons[i] = neu
    for xi,x in enumerate(X): # for each step dx
        for yi, y in enumerate(Y): #for each step dy
            idx = m*xi + yi
            v = np.matrix([x,y])
            j = neu.alpha * matmul(v, neu.encoder.transpose()) + neu.jbia
            j = float(j)
            a = LIFcurve(0.002, 0.02, j)
            A_xy[i][idx] = a
```

In [421]:
```
X = np.linspace(-1,1,2/stepSize)
Y = np.linspace(-1,1,2/stepSize)

X1 = np.zeros([m*m,1])
Y2 = np.zeros([m*m,1])

for xi,x in enumerate(X): # for each step dx
    for yi, y in enumerate(Y): #for each step dy
        idx = m*xi + yi
        X1[idx] = x
        Y2[idx] = y

def decode2DNeurons(A):
    YpsilonX = stepSize * np.matmul(A,X1)
    YpsilonY = stepSize * np.matmul(A,Y2)

    Gamma = stepSize * np.matmul(A, np.transpose(A))
    Gamma_inv = np.linalg.inv(Gamma)
    d_X = np.dot(Gamma_inv, YpsilonX)
    d_Y = np.dot(Gamma_inv, YpsilonY)
    d = np.append(d_X, d_Y, axis=1)
```

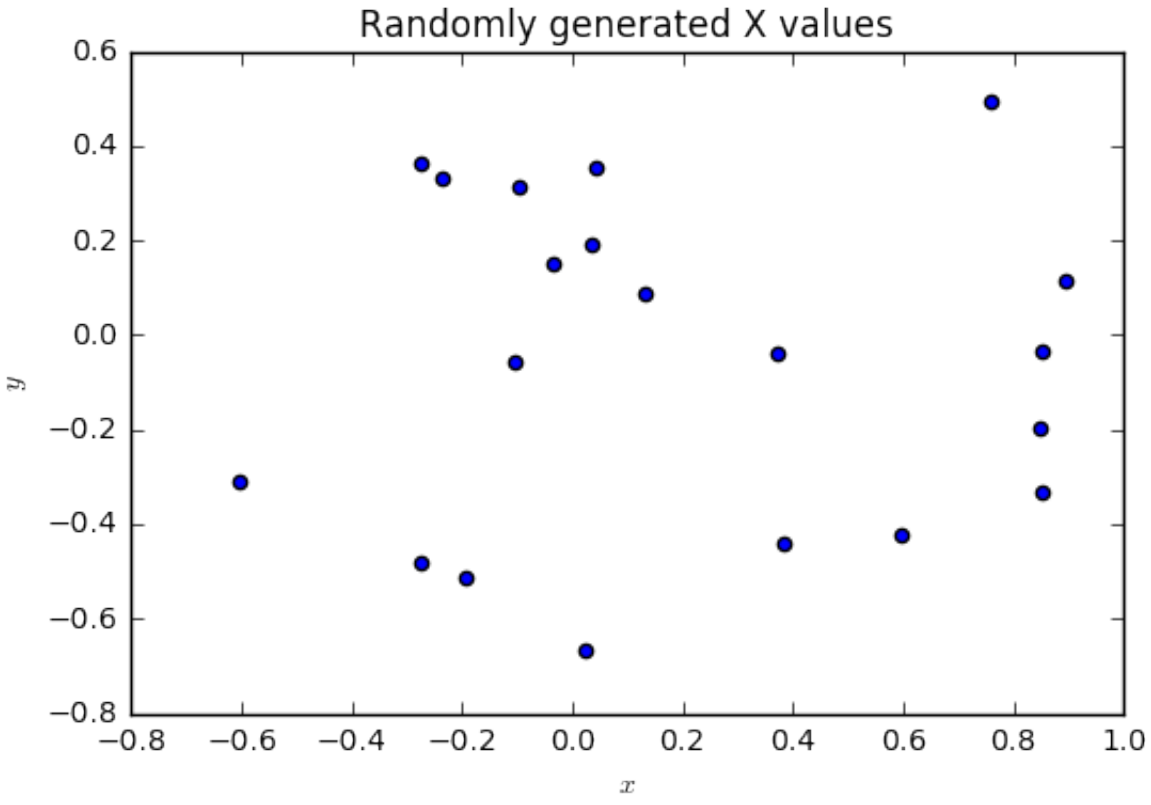
```
    return d

D = decode2DNeurons(A_xy)

for d in range(0,n):
    scatter(D[d][0],D[d][1])
title("Decoders")
xlabel("$x$")
ylabel("$y$");
print D.shape
```

```
(100, 2)
```



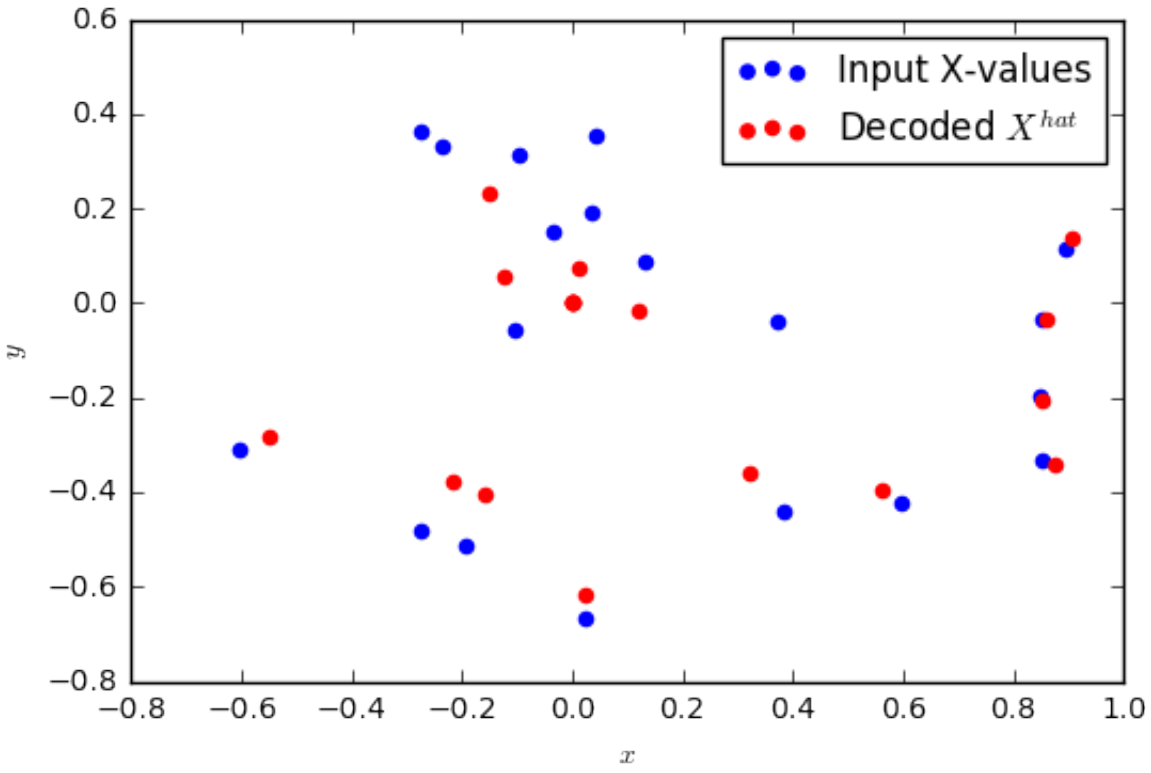
c. [1 mark] Generate 20 random x values over the unit circle (i.e. with different directions and radiuses). For each x value, determine the neural activity a for each of the 100 neurons. Now decode these values (i.e. compute $\hat{x}$) using the decoders from part b). Plot the original and decoded values on the same graph in different colours, and compute the RMSE.

In [422]:
```
Xrand = np.zeros([20,2])
Arand = np.zeros([n,20])
line = 0
figure()
for z in range(0,20):
    th = np.random.uniform(0,2*np.pi)
    r = np.random.uniform(0,1)
    xx = r*cos(th)
    yy = r*sin(th)
    Xrand[z][0] = xx
    Xrand[z][1] = yy
    scatter(xx,yy);
    for i in range(0,n):
        v = np.matrix([xx,yy])
        j = TwoD_Neurons[i].alpha * matmul(v, TwoD_Neurons[i].encoder.tra
        j = float(j)
        a = LIFcurve(0.002, 0.02, j)
        Arand[i][z] = a
title("Randomly generated X values")
xlabel("$x$")
ylabel("$y$");
```



In [423]:
```
X_hatrand = np.dot(Arand.transpose(), D)

figure()
scatter(Xrand[:,0],Xrand[:,1], color='blue', label = "Input X-values");
scatter(X_hatrand[:,0],X_hatrand[:,1], color='red', label = "Decoded $X^{
xlabel("$x$")
ylabel("$y$")
plt.legend()

RMS = 0
Diff = Xrand - X_hatrand
for x in range(0,20):
    dd = np.linalg.norm(Diff[x])
    RMS += np.power(dd,2)
RMS = np.sqrt(RMS/20)
print "RMS: " + str(RMS)
```

```
RMS: 0.161218315606
```



d. [2 marks] Repeat part c) but use the *encoders* as decoders. This is what Georgopoulos used in his original approach to decoding information from populations of neurons. Plot the decoded values this way and compute the RMSE. In addition, recompute the RMSE in both cases, but ignoring the magnitude of the decoded vector. What are the relative merits of these two approaches to decoding?

- To ignore the magnitude of the vectors, normalize the length of the decoded vectors before computing the RMSE.