In [608]:
```
import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np
import math
```

Problem 1 :

N=2

In [609]:
```
eta = 1
N=2
num=1000

np.random.seed(18)
A = np.array([[np.random.normal(0,eta/N) for i in range(N)]for j in rang
e(num)])
y = np.random.normal(0,1)*np.identity(N)
X = np.matmul(A,y)



eigenvalues, eigenvectors = np.linalg.eig(np.cov(A.T))

W_Oja = np.random.normal(0,1, size=(N, 1))
W_Oja_prime = np.random.normal(0,1, size=(N, 1))

c = 0.001  ## learning rate
tol = 1e-10
iter=0
norm_W_Oja =[]
log_T = []
data_slice = []
while np.linalg.norm(W_Oja_prime - W_Oja) > tol:
    W_Oja_prime = W_Oja.copy()

    Y = np.dot(X, W_Oja)
    W_Oja += c * np.sum(Y*X - np.square(Y)*W_Oja.T, axis=0).reshape((N,
1))  ## with normalization
    data_slice .append(np.dot(X[0],W_Oja))
    norm_W_Oja.append(np.linalg.norm(W_Oja))
    iter+=1
    log_T.append(math.log(iter,10))
```

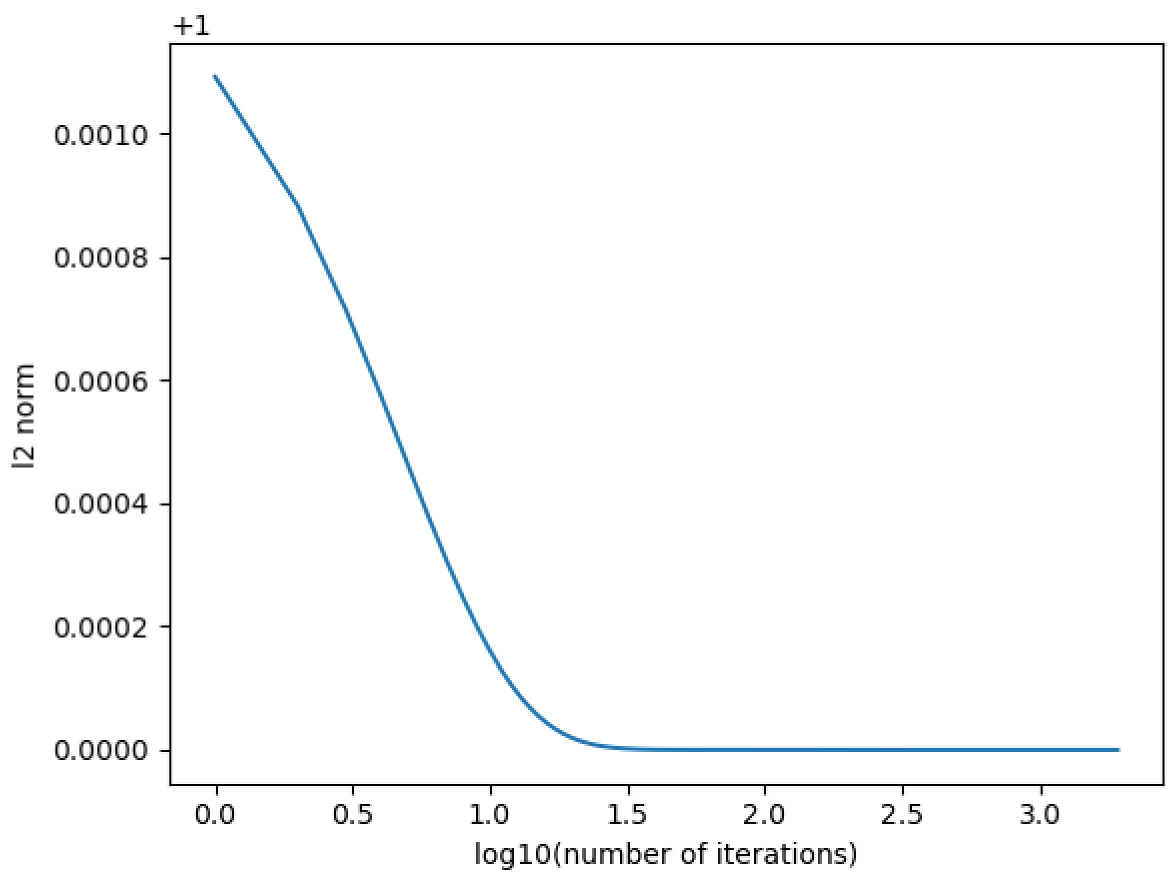
Check eigenvector corresponding to maximal eigenvalue :

In [610]:
```
print("eigenvalues : ",eigenvalues)
index = np.argmax(eigenvalues)
print()
print("eigenvector corresponding to maximal eigenvalue : ")
print(eigenvectors[:,index])
print()
print("Oja's weights :")
print(W_Oja[:,0])
```

```
eigenvalues :  [0.2695631  0.24824078]

eigenvector corresponding to maximal eigenvalue :
[0.91023197 0.41409874]

Oja's weights :
[-0.91084484 -0.41274893]
```

Indeed, the weights ended up aligning with the first principal component, i.e. the first eigenvector of Σ.

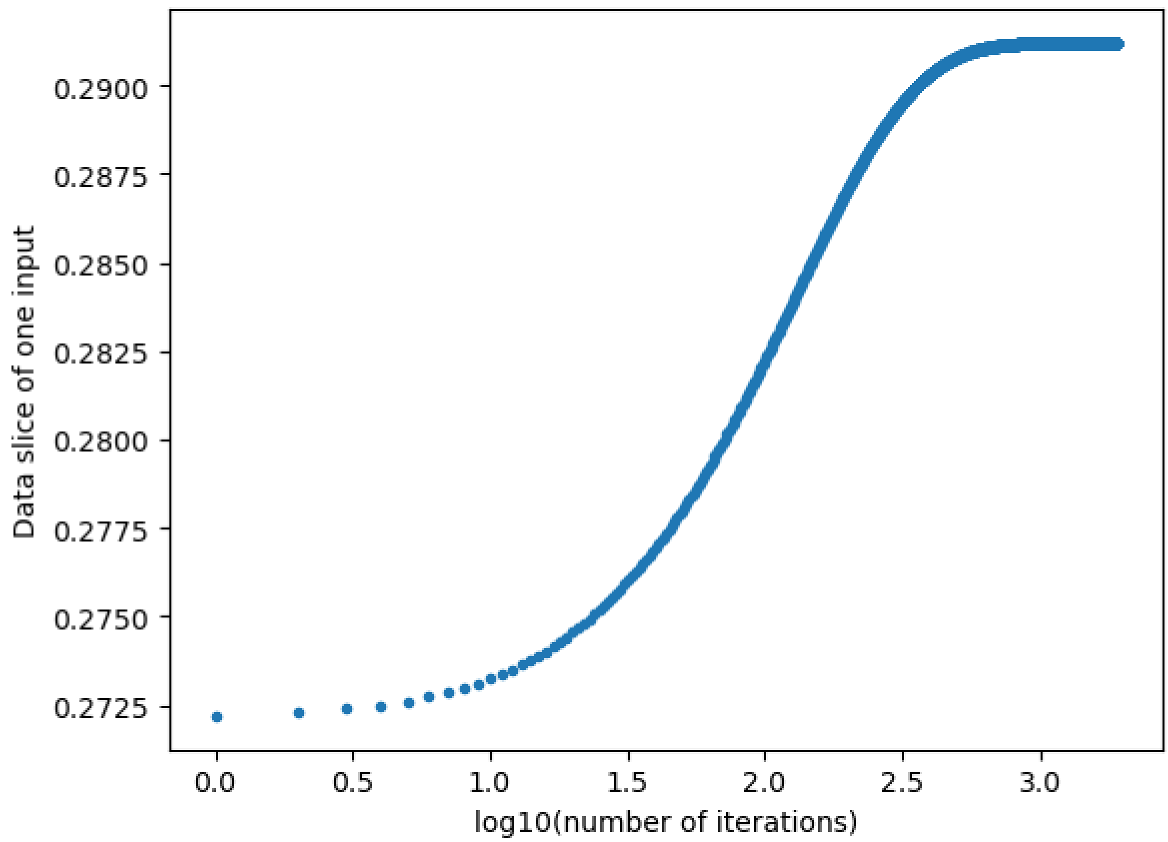
In [611]:
```
plt.plot(log_T,norm_W_Oja)
plt.xlabel("log10(number of iterations)")
plt.ylabel("l2 norm")
plt.show()
```



The normalization has the effect of shrinking the weight values

Data slice :

In [612]:
```
plt.plot(log_T,data_slice,linestyle='None',marker='.')
plt.xlabel("log10(number of iterations)")
plt.ylabel("Data slice of one input")
plt.show()
```



N=10

In [613]:
```
eta = 1
N=10
num=1000
np.random.seed(18)
A = np.array([[np.random.normal(0,eta/N) for i in range(N)]for j in rang
e(num)])
y = np.random.normal(0,1)*np.identity(N)
X = np.matmul(A,y)



eigenvalues, eigenvectors = np.linalg.eig(np.cov(A.T))

W_Oja = np.random.normal(0,1, size=(N, 1))
W_Oja_prime = np.random.normal(0,1, size=(N, 1))

c = 0.001
tol = 1e-10
iter=0
norm_W_Oja =[]
log_T = []
data_slice = []
while np.linalg.norm(W_Oja_prime - W_Oja) > tol:
    W_Oja_prime = W_Oja.copy()

    Y = np.dot(X, W_Oja)
    W_Oja += c * np.sum(Y*X - np.square(Y)*W_Oja.T, axis=0).reshape((N,
1))
    data_slice .append(np.dot(X[0],W_Oja))
    norm_W_Oja.append(np.linalg.norm(W_Oja))
    iter+=1
    log_T.append(math.log(iter,10))
```

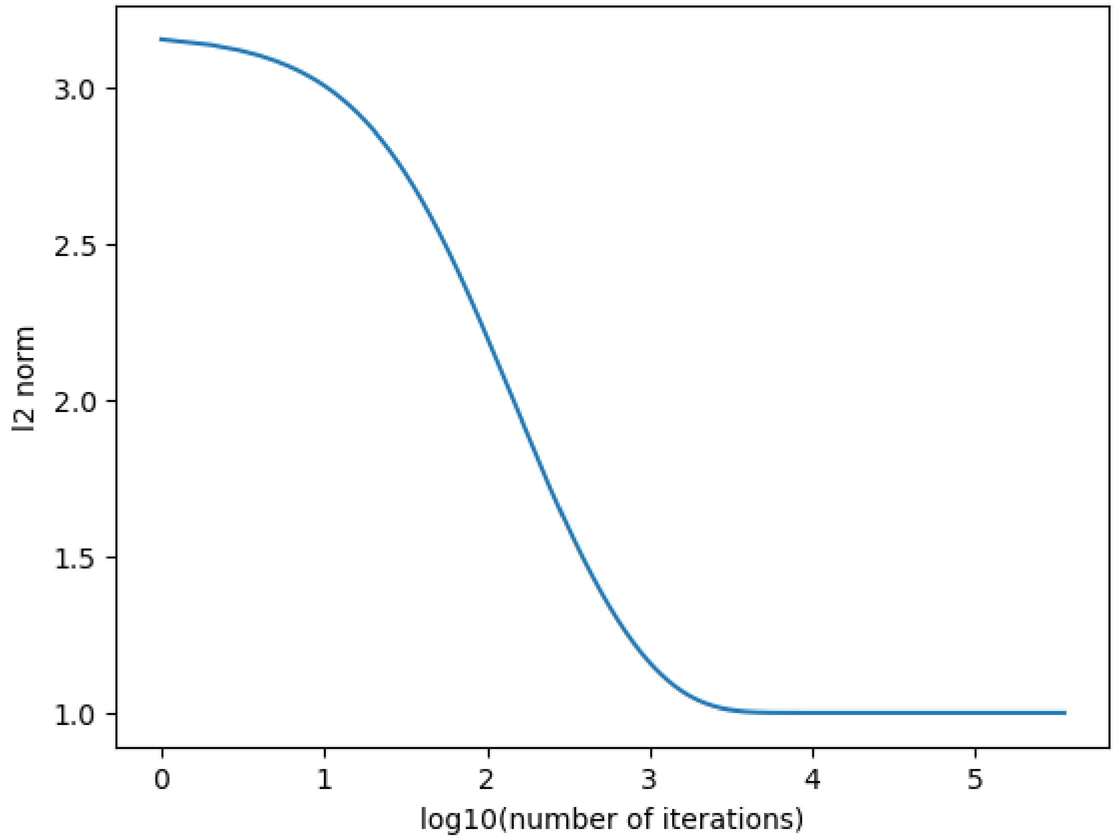
Check eigenvector corresponding to maximal eigenvalue :

In [614]:
```
print("eigenvalues : ",eigenvalues)
index = np.argmax(eigenvalues)
print()
print("eigenvector corresponding to maximal eigenvalue : ")
print(eigenvectors[:,index])
print()
print("Oja's weights :")
print(W_Oja[:,0])
```

```
eigenvalues :  [0.0084321  0.01169387 0.01113717 0.00891398 0.00915128
0.00956429
 0.00994603 0.01075398 0.01062643 0.01051229]

eigenvector corresponding to maximal eigenvalue :
[-0.07996659 -0.11903136  0.20394118  0.16414595 -0.21389625  0.3754132
5
  0.22794325 -0.80854231  0.12402269  0.05598078]

Oja's weights :
[ 0.08251594  0.12660158 -0.20461552 -0.16104483  0.20775896 -0.3675991
 -0.22036285  0.81271379 -0.13148548 -0.06866877]
```

Indeed, the weights ended up aligning with the first principal component, i.e. the first eigenvector of (they are in the same direction).

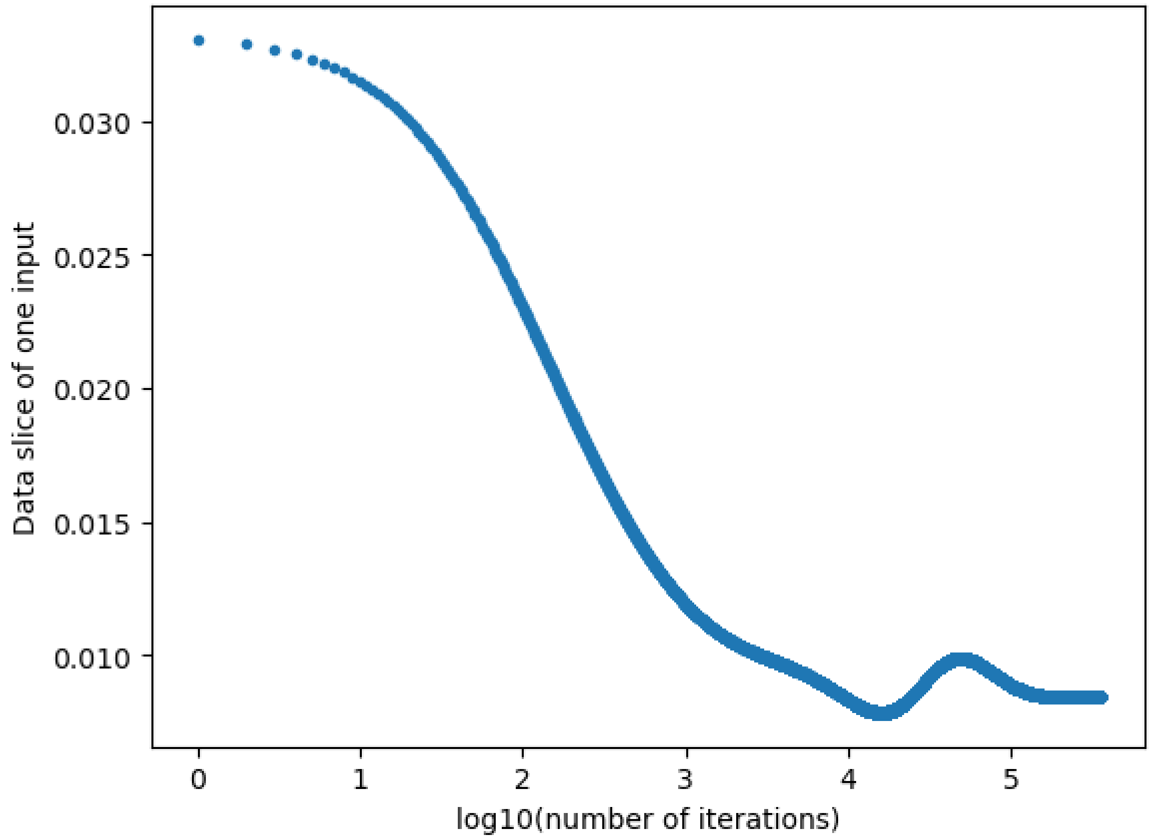
In [615]:
```
plt.plot(log_T,norm_W_Oja)
plt.xlabel("log10(number of iterations)")
plt.ylabel("l2 norm")
plt.show()
```



The normalization has the effect of shrinking the weight values

Data slice :

In [616]:
```
plt.plot(log_T,data_slice,linestyle='None',marker='.')
plt.xlabel("log10(number of iterations)")
plt.ylabel("Data slice of one input")
plt.show()
```

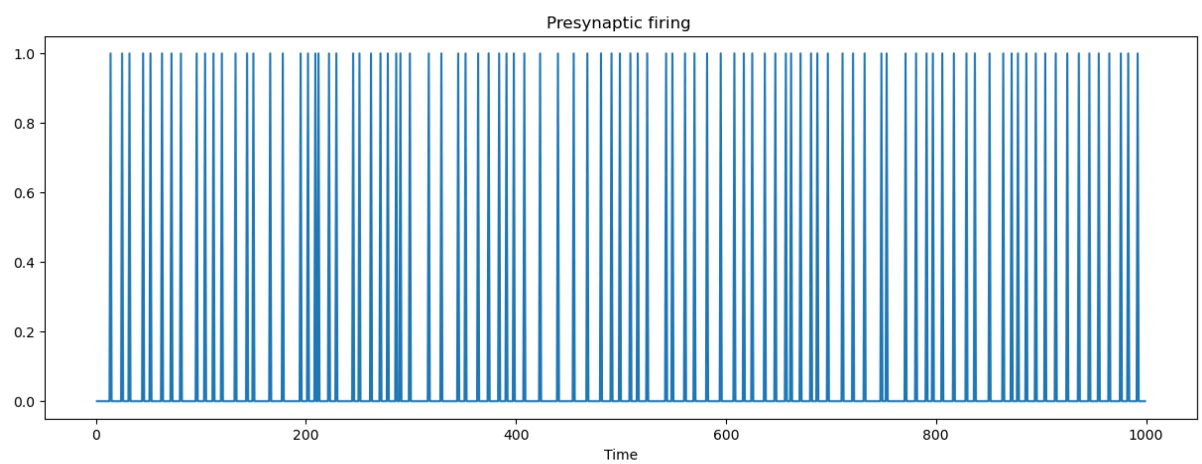


In [ ]:

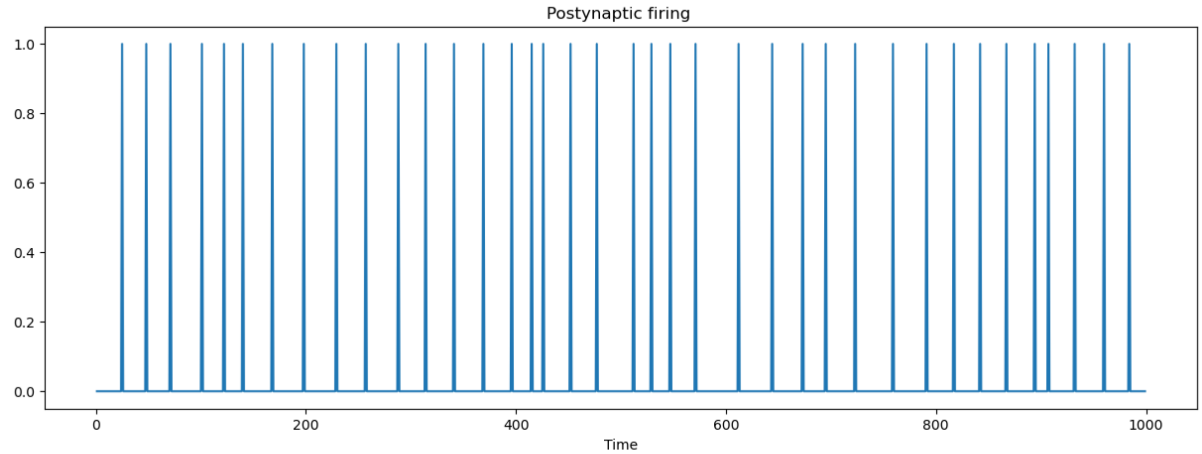
Problem 2:

In [635]:
```
x_pre = 10

c = 1.03
tau_plus = 14
A_minus = -0.51
tau_minus = 34

w_j=np.random.normal(0,1)
d_t=0.01
T=1000
time = np.arange(1,T,1)

x_j_pre =[0 for t in time]

time_spikes_pre =[]
t_spike_pre=np.random.poisson(10)
x_j_pre[t_spike_pre] = 1

while(t_spike_pre<T):
    t_spike_pre+=np.random.poisson(10)
    if(t_spike_pre<T):
        x_j_pre[t_spike_pre] = 1
        time_spikes_pre.append(t_spike_pre)

y_post = [0 for t in time]
time_spikes_post =[]
t_spike_post=np.random.poisson(25)
y_post[t_spike_post] = 1

while(t_spike_post<T):
    t_spike_post+=np.random.poisson(25)
    if(t_spike_post<T):
        y_post[t_spike_post] = 1
        time_spikes_post.append(t_spike_post)
```

In [636]:
```
from pylab import rcParams
rcParams['figure.figsize'] = 15, 5
plt.plot(time,x_j_pre)
plt.xlabel("Time")
plt.title("Presynaptic firing ")
plt.show()
```



In [639]:
```
plt.plot(time,y_post)
plt.xlabel("Time")
plt.title("Postynaptic firing ")
plt.show()
```



Implementation of Nearest-neighbor STDP

$\Delta_w =$ **average of potentiation + average of depression**
$= \int_0^\infty A_+ exp(\frac{-t}{\tau_+}) xexp(-xt)dt + \int_{-\infty}^0 A_- exp(\frac{t}{\tau-}) xexp(xt)dt$

$$\boxed{\Delta_w = x(\frac{A_+}{\tau_+^{-1} + x} + \frac{A_-}{\tau_-^{-1} + x})}$$

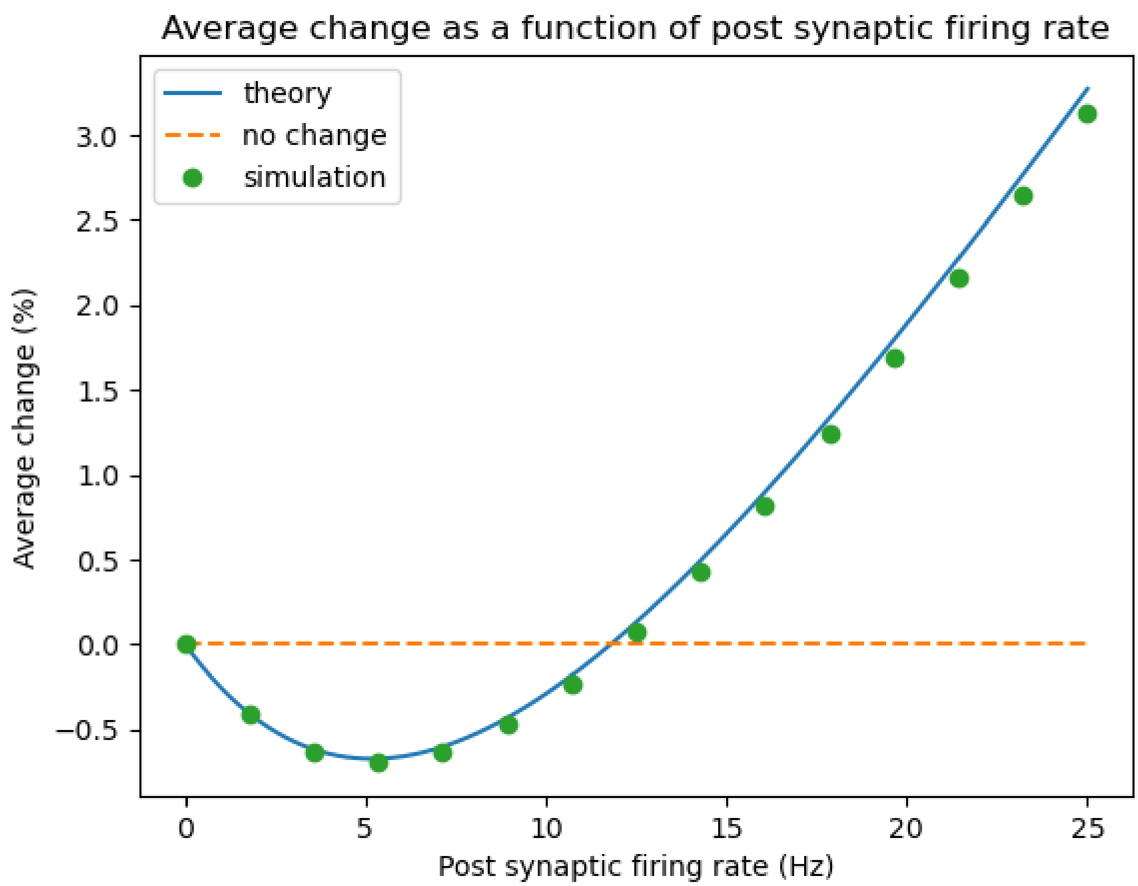
Numerical Simulation

In [640]:
```
x_pre = 10

A_plus = 1.03
tau_plus = 14
A_minus = -0.51
tau_minus = 34

def trapezoidal(f, a, b, n):
    h = float(b-a)/n
    result = 0.5*f(a) + 0.5*f(b)
    for i in range(1, n):
        result += f(a + i*h)
    result *= h
    return result

val=[]
X_sim = np.linspace(0,25,15)
for p in X_sim:
    post_syn = p*1e-3

    v = lambda t: A_plus*math.exp(-t/tau_plus)*post_syn*math.exp(-post_s
yn*t)

    n = 20000
    numerical1 = trapezoidal(v, 0, 10000, n)

    v = lambda t: A_minus*math.exp(t/tau_minus)*post_syn*math.exp(post_s
yn*t)

    n =20000
    numerical2 = trapezoidal(v, -100000, 0, n)
    val.append(numerical1+numerical2)
```

Theoretical formula

In [641]:
```
x_pre = 10

A_plus = 1.03
tau_plus = 14
A_minus = -0.51
tau_minus = 34

def average_change(x):
    return(100*x*((A_plus/((1/tau_plus)+x))+(A_minus/((1/tau_minus)+x
))))
```

In [642]:
```
mpl.rcParams.update(mpl.rcParamsDefault)
X = np.linspace(0,25,1000)
plt.plot(X,[average_change(x*1e-3) for x in X],label="theory")
plt.plot(X,[0 for x in X],'--',label="no change")
plt.plot(X_sim,np.multiply(val,100),'o',label="simulation",linestyle="None")
plt.xlabel("Post synaptic firing rate (Hz)")
plt.ylabel("Average change (%)")
plt.title("Average change as a function of post synaptic firing rate")
plt.legend()
plt.show()
```



The results are consistent with what we have seen in the lecture with a curve form that is very similar to the BCM rule.

In [ ]: