In [2]:

```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(1977)
# fixed seed is useful for debugging
```

In [3]:

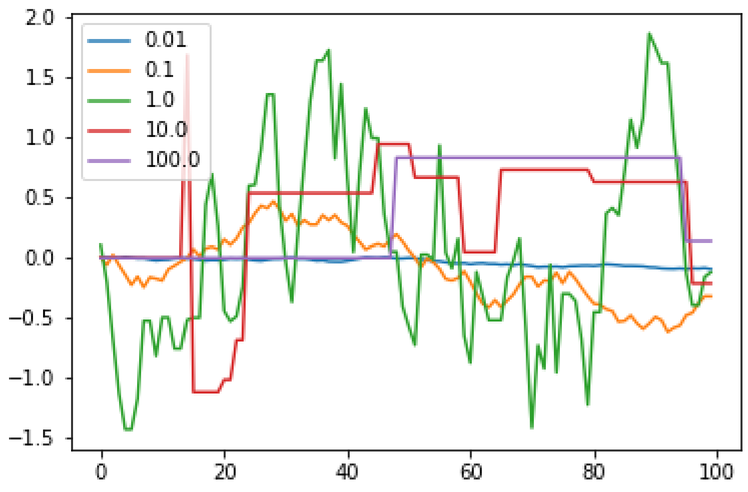
```
def MC(q0=0,delta=0.1,nsteps=1000,seed=None):
    if seed is not None:
        np.random.seed(seed)
    q=q0
    kappa=1.0
    energy=0.5*kappa*q**2
    kBT=1.0
    traj=[]
    for istep in range(nsteps):
        qtry=q+(2*np.random.rand()-1)*delta
        energy_try=0.5*kappa*qtry**2
        deltae=energy_try-energy
    # speeds the algorithm because draws random numbers
    # only if delta energy > 0
        if deltae<=0.0:
            acceptance=1.0
        else:
            acceptance=np.exp(-deltae/kBT)
        if acceptance >=1.0 or acceptance>np.random.rand():
            q=qtry
            energy=energy_try
        traj.append(q)
    return np.array(traj)

# tipycally the first points are discarded
# see autocorrelation time
```

In [4]:

```
# plot of the trajecotry fot different values of delta
# notice that the straight horizontal lines are caused by non accepted moves
# as I decrease delta the lines become smoother (acceptance increases)

for delta in [0.01,0.1,1.0,10.0,100.0]:
    traj=MC(delta=delta,nsteps=100)
    plt.plot(traj,label=str(delta))
plt.legend()
```

Out[4]:

```
<matplotlib.legend.Legend at 0x7fcfad539b50>
```

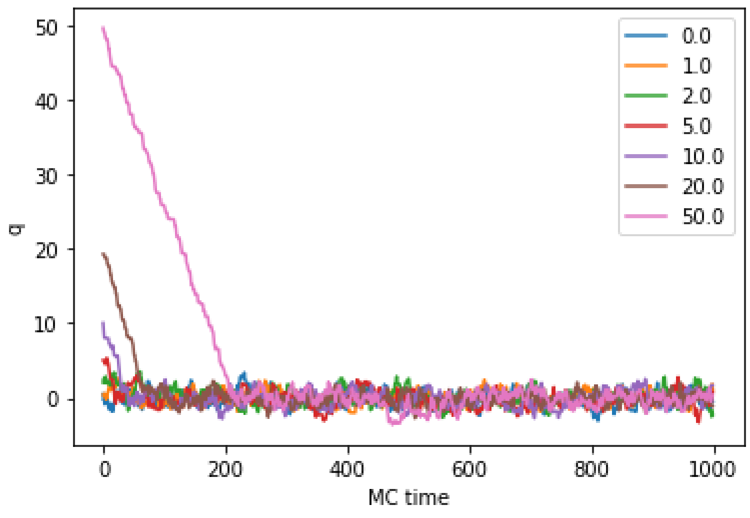


In [5]:

```
# Plot for different values of initial conditions at delta fixed
# after a time large enough the dependence on the initial condition vanishes
# initially there is a linear slope due to the form of the energy

for q0 in [0.0,1.0,2.0,5.0,10.0,20.0,50.0]:
    traj=MC(delta=1.0,nsteps=1000,q0=q0)
    plt.plot(traj,label=str(q0))
plt.xlabel("MC time")
plt.ylabel("q")
plt.legend()
```

Out[5]:

```
<matplotlib.legend.Legend at 0x7fcfad43a370>
```

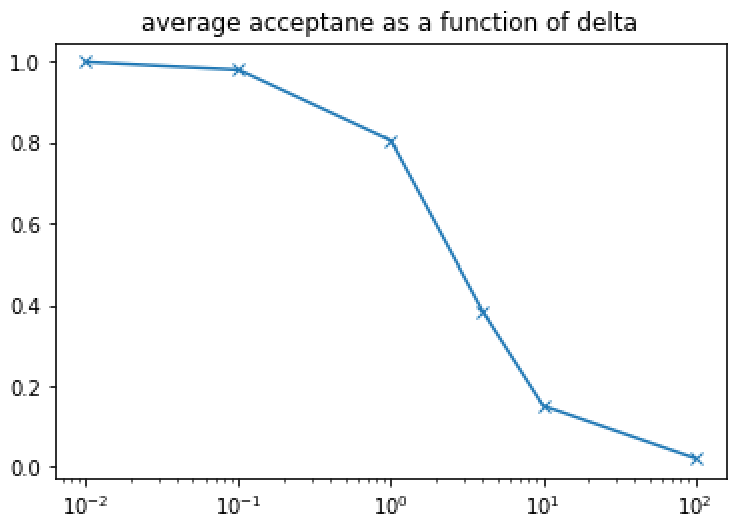


In [6]:

```
# returns the average acceptance
# notice that EVERY step must be counted
# possible mistake: never let acceptance be higher than 1. Prevent it with an if statem
ent

def MC_acc(q0=0,delta=0.1,nsteps=1000,seed=None):
    if seed is not None:
        np.random.seed(seed)
    q=q0
    kappa=1.0
    energy=0.5*kappa*q**2
    kBT=1.0
    acc=[]
    for istep in range(nsteps):
        qtry=q+(2*np.random.rand()-1)*delta
        energy_try=0.5*kappa*qtry**2
        acceptance=np.exp(-(energy_try-energy)/kBT)
        if acceptance > 1.0:
            acceptance=1.0
        if acceptance>np.random.rand():
            q=qtry
            energy=energy_try
        acc.append(acceptance)
    return np.average(acc)
```

In [7]:

```
acc=[]
deltas=[0.01,0.1,1.0,4.0,10.0,100.0]
for delta in deltas:
    acc.append(MC_acc(delta=delta))
plt.plot(deltas,acc,"x-")
plt.title("average acceptane as a function of delta")
plt.xscale("log")
```

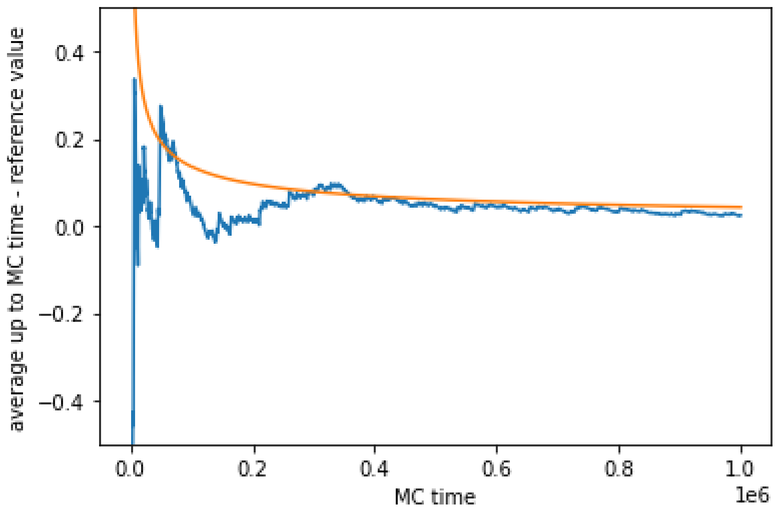


In [8]:

```
# plot of average value of q^2 up to MC time

traj=MC(nsteps=1000000)

# this is a moving average of q^2
# the -1 in the end is meant to normalize the average to zero
# so basically I am plotting an error
# as we can see, the longer the simulation, the more correct the result is
# statistically the error decays as (N)**(-0.5) where N is the number of points

plt.plot((np.cumsum(traj**2)/(np.array(range(len(traj)))+1))-1)
plt.xlabel("MC time")
plt.ylabel("average up to MC time - reference value")
plt.plot(30*np.std(traj**2) / np.sqrt(np.array(range(len(traj)))+1))

# though the error is not just the  s.dev. of q**2 divided by (N)**(-0.5) because the
# values are not independend, therefore we need to add the scaling factor 30, which
# accounts for that

plt.ylim((-0.5,0.5))
```

Out[8]:

```
(-0.5, 0.5)
```

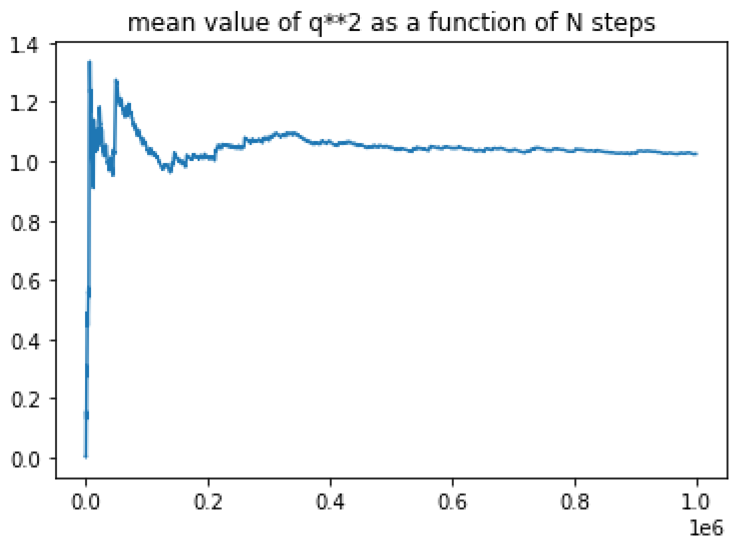


In [9]:

```
plt.plot((np.cumsum(traj**2)/(np.array(range(len(traj)))+1)))
plt.title('mean value of q**2 as a function of N steps')

# as one can notice it tends to approach 1
```

Out[9]:

```
Text(0.5, 1.0, 'mean value of q**2 as a function of N steps')
```

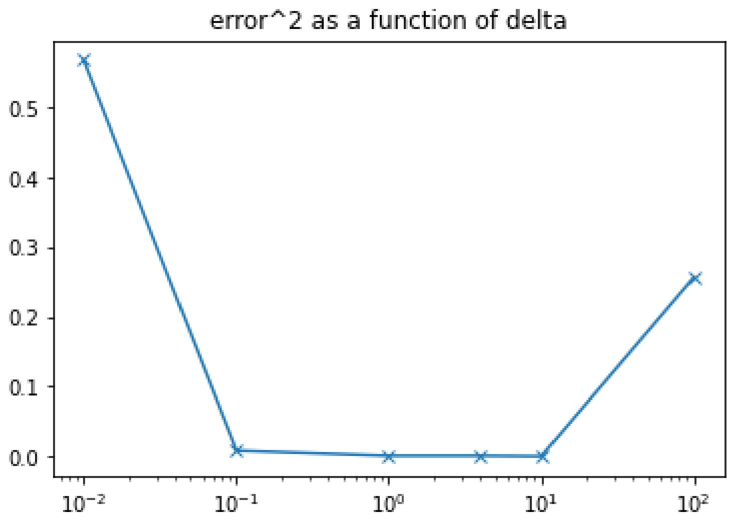


In [10]:

```
# plotting the error squared as a function of delta

err2=[]
for delta in deltas:
    err2.append((np.average(MC(nsteps=10000,delta=delta)**2)-1)**2)
plt.plot(deltas,err2,"x-")
plt.xscale("log")
plt.title('error^2 as a function of delta')

# As we can see there is a critical value of delta (order of units) for which the error
is
# minimal (for a fixed number of steps and initial condition)
```

Out[10]:

```
Text(0.5, 1.0, 'error^2 as a function of delta')
```

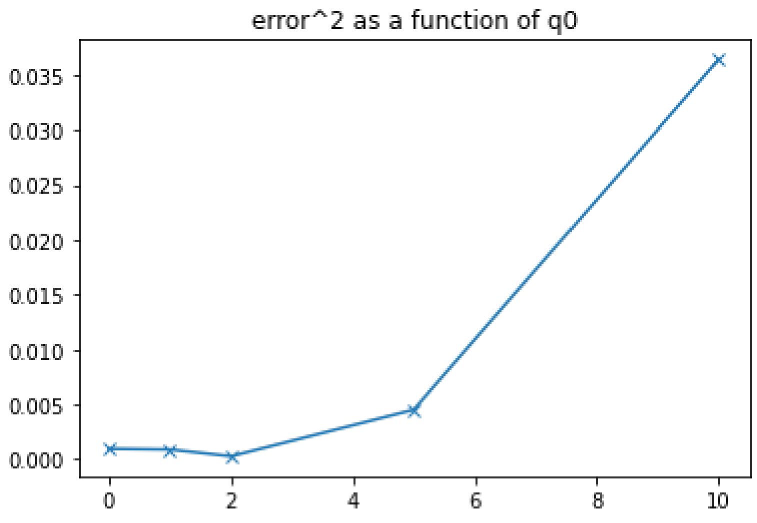


In [11]:

```
# Plotting the error squared as a function of q_0

err2=[]
q0s=[0.0,1.0,2.0,5.0,10.0]
for q0 in q0s:
    err2.append((np.average(MC(nsteps=10000,delta=1.0,q0=q0)**2)-1)**2)
plt.plot(q0s,err2,"x-")
plt.title('error^2 as a function of q0')

# As we can see, for fixed delta and N, there is a critical value of q0 (2 approximatel
y)
# for which the error is minimal
```

Out[11]:

```
Text(0.5, 1.0, 'error^2 as a function of q0')
```



In [12]:

```
# Part 4
# hint only put tuples as default arguments of the function

def MC3D(q0=None,delta=5,nsteps=100000,seed=None):
    if seed is not None:
        np.random.seed(seed)
    if q0 is None:
        q0=np.zeros((2,3))
    L=4

    # important hint: always use =+ to copy an array, otherwise it becomes a reference
    # copy of the original array and if I change one, also the other changes

    q=+q0
    kappa=1.0
    energy=0.5*kappa*(np.sqrt(np.sum((q[1]-q[0])**2))-L)**2
    kBT=1.0
    traj=[]
    for istep in range(nsteps):
        qtry=+q
        i=np.random.randint(6)
        if i<3:
            qtry[0,i]+=(2*np.random.rand()-1)*delta
        else:
            qtry[1,i-3]+=(2*np.random.rand()-1)*delta
        energy_try=0.5*kappa*(np.sqrt(np.sum((qtry[1]-qtry[0])**2))-L)**2
        deltae=energy_try-energy
        if deltae<=0.0:
            acceptance=1.0
        else:
            acceptance=np.exp(-deltae/kBT)
        if acceptance >=1.0 or acceptance>np.random.rand():
            q=+qtry
            energy=energy_try
        traj.append(q)
    return np.array(traj)
```

In [13]:

```
traj=MC3D()
```

In [14]:

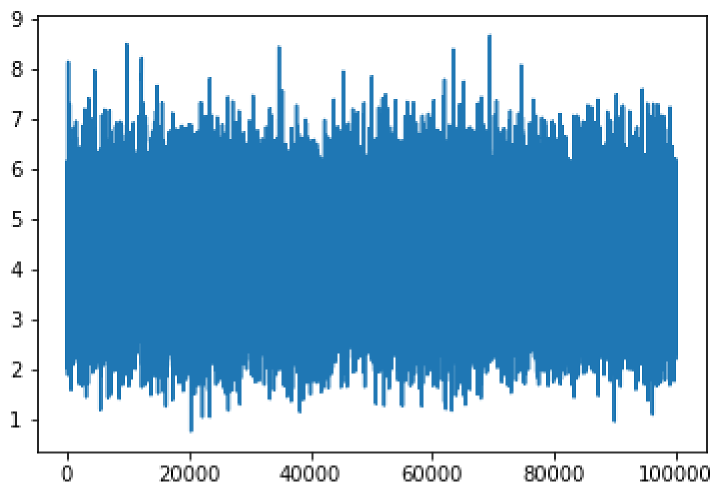
```
# writing down the trajectories in an xyz file

def write_xyz(traj):
    with open("traj.xyz","w") as f:
        for i in range(len(traj)):
            print(2,file=f)
            print("",file=f)
            print("Ar ",traj[i,0,0],traj[i,0,1],traj[i,0,2],file=f)
            print("Ar ",traj[i,1,0],traj[i,1,1],traj[i,1,2],file=f)
```

In [15]:

```
# plotting the time series of the distances

dist=np.sqrt(np.sum((traj[:,1]-traj[:,0])**2,axis=1))
plt.plot(dist)
```

Out[15]:

```
[<matplotlib.lines.Line2D at 0x7fcf7f13c220>]
```



In [177]:

```
# average distance
# if one computes the average distance analitically the result is the same
# it is not exactly 4 because of an additional term coming from the jacobian
np.average(dist)
```

Out[177]:

```
4.482734347303989
```

In [18]:

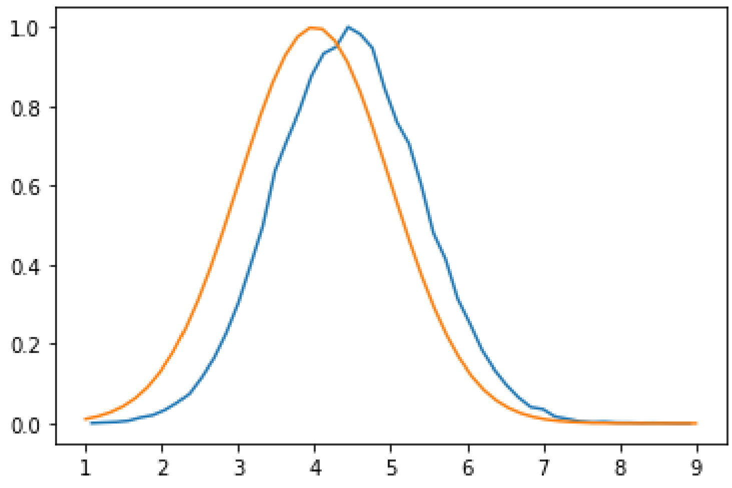
```
hist=np.histogram(dist,range=(1,9),bins=50)
```

In [20]:

```
# plotting the normalized histogram of the distances
# one can see that it is clearly different than a gaussian with average 4 (orange)
# that is because of a jacobian term that comes out when we compute the actual distribu
tion
# remember that we are plotting a distance (1-D object) coming from a 3-D problem


plt.plot(0.5*(hist[1][:-1]+hist[1][1:]),hist[0]/np.max(hist[0]))
x=np.linspace(1,9)
plt.plot(x,np.exp(-0.5*(x-4)**2))
```

Out[20]:

```
[<matplotlib.lines.Line2D at 0x7fcf7f019b80>]
```


1.0
0.8
0.6
0.4
0.2
0.0
1
2
3
4
5
6
7
8
9