XING WENYU 3035906071

1 Extended Kalman filter (60 points)

(a)
1
N(0,Rt)
N(0,Qt)
N(mu_t-1,sigma_t-1)
2
mu_t: predicted mean
sigma_t: predicted variance
g: motion function
G_t: jacobian matric
h: measurement function
H_t: jacobian matric
Q_t: noise
R_t: noise
K_t: kelman gain
differences: EKF can deal with no-linear problem, while KF can only be used on linear system.
(b)
1.
from slide we can get:
x_t=x_t-1+delta_t*cos(theta+delta_r1)
y_t=y_t-1+delta_t*sin(theta+delta_r1)
theta_t=theta_t-1+delta_r1+delta_r2
we do derivation on x,y,theta to each equation,then we can get jacobian matrix
G_t=[
[1,0, -delta_t*sin(theta+delta_r1)],
[0,1, delta_t*cos(theta+delta_r1)],
[0,0,1]
]
2.

```
G_t = np.array([[1, 0, -delta_trans * np.sin(theta + delta_rot1)],
              [0, 1, delta_trans * np.cos(theta + delta_rot1)],
              [0, 0, 1]])
k = 1.0
R_t = np.array([[0.2, 0, 0],
              [0, 0.2, 0],
              [0, 0, 0.02]]) * k
mu_t = np.array([x + delta_trans * np.cos(theta + delta_rot1),
               y + delta_trans * np.sin(theta + delta_rot1),
               theta + delta_rot1 + delta_rot2])
```

```python
sigma_t = G_t.dot(sigma).dot(G_t.T) + R_t

return mu_t, sigma_t
```

(c)

1.

h_t=sqrt((x-landmarks.x)^2+(y-landmarks.y)^2)

H_t=[

(x-lanmarks.x)/h_t,

(y-lanmarks.y)/h_t,

0]

2.

```python
h = np.zeros(len(ids))
H_t = np.zeros((len(ids), 3))
i = 0
for id in ids:
    h[i] = np.sqrt((x - landmarks[id][0]) ** 2 + (y -
landmarks[id][1]) ** 2)
    H_t[i] = [(x - landmarks[id][0]) / h[i], (y - landmarks[id][1]) /
h[i], 0]
    i += 1
k = 1.0
Q_t = np.eye(len(ids)) * 0.5 * k
K_t = sigma.dot(H_t.T).dot(np.linalg.inv(H_t.dot(sigma).dot(H_t.T) +
Q_t))
mu_t = mu + K_t.dot(ranges - h)
sigma_t = (np.eye(3) - K_t.dot(H_t)).dot(sigma)

return mu_t, sigma_t
```

(d)

```python
Q_t = np.eye(len(ids) * 2) * 0.5
H_t, Z_t, h = [], [], []

for i in range(len(ids)):
    lx = landmarks[ids[i]][0]
    ly = landmarks[ids[i]][1]
    q = (lx - x) ** 2 + (ly - y) ** 2
    hi = [np.arctan2([ly - y], [lx - x])[0] - theta, q ** 0.5]
    Hi = [[(ly - y) / q, (x - lx) / q, -1], [-(lx - x) / np.sqrt(q),
-(ly - y) / np.sqrt(q), 0]]
    H_t.append(Hi[0])
    H_t.append(Hi[1])
    Z_t.append(bearing[i])
    Z_t.append(ranges[i])
    h.append(hi[0])
```

```
    h.append(hi[1])
H_t, Z_t, h = np.array(H_t), np.array(Z_t), np.array(h)
inv = np.linalg.inv(np.dot(np.dot(H_t, sigma), np.transpose(H_t)) +
Q_t)
K_t = np.dot(np.dot(sigma, np.transpose(H_t)), inv)
mu_t = mu + np.dot(K_t, (Z_t - h))
sigma_t = np.dot(np.eye(len(K_t)) - np.dot(K_t, H_t), sigma)

return mu_t, sigma_t
```

(e)

R(range)	R(Bearing)	Q	Prior	Error
S	S	S	1	Almost zero
L	L	L	1	Very small
L	L	L	0	Small,but observable
L		L	1	Small,but observable
	L	L	1	Small,but observable
L		L	0	Large error in the beginning,smaller later
	L	L	0	Large error in the beginning,always significant

2 Estimate the location of an object on a circle (40 points)

(a)

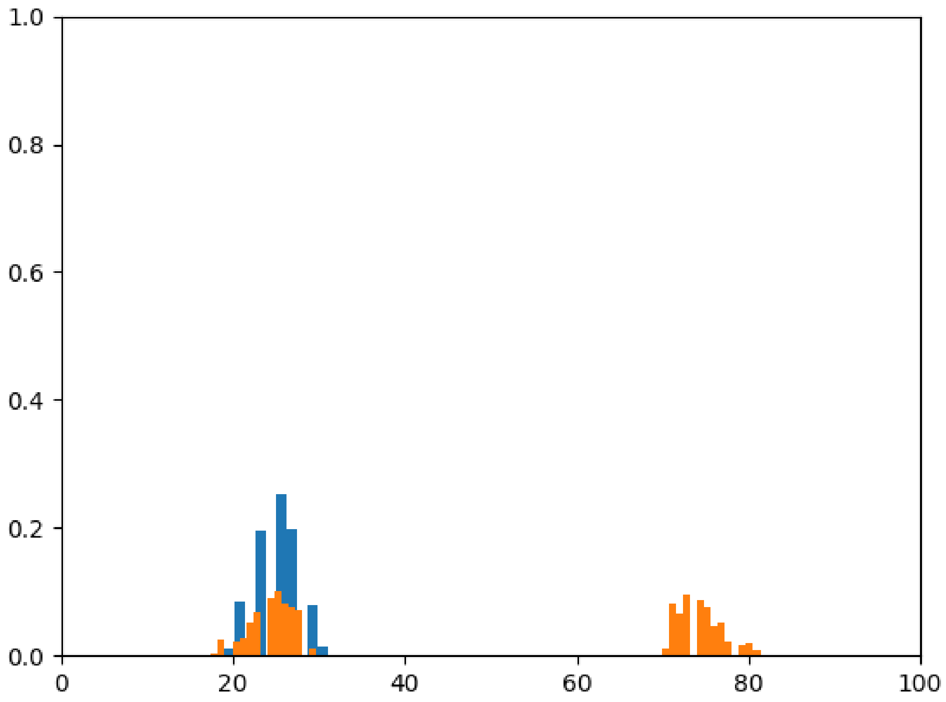
attachment:

(Install numpy,matplotlib,run Estimate.py)

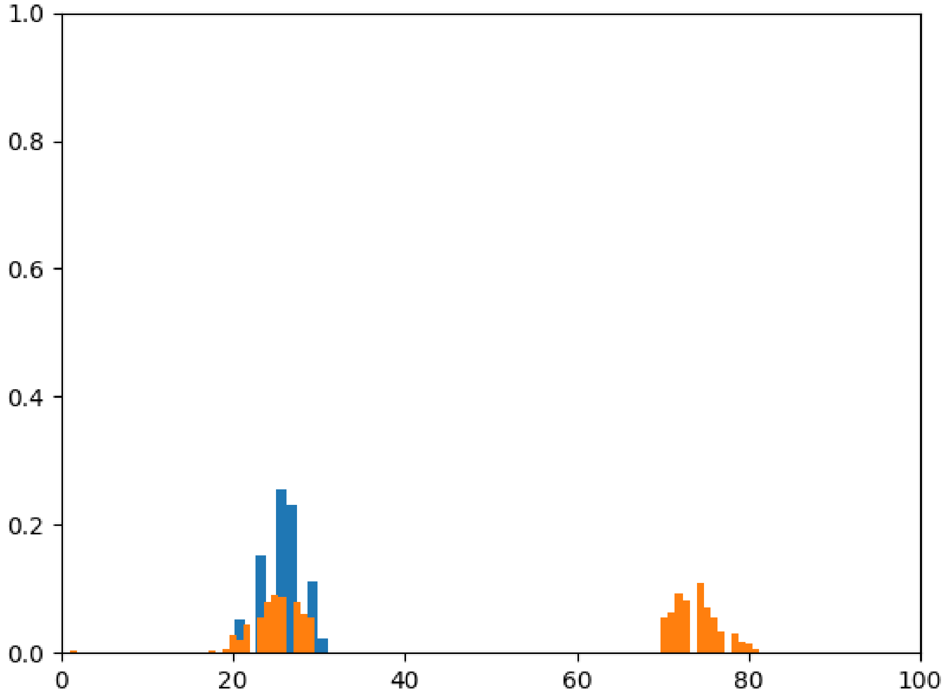
(b)

The following figures are the estimation and simulation in step 5.

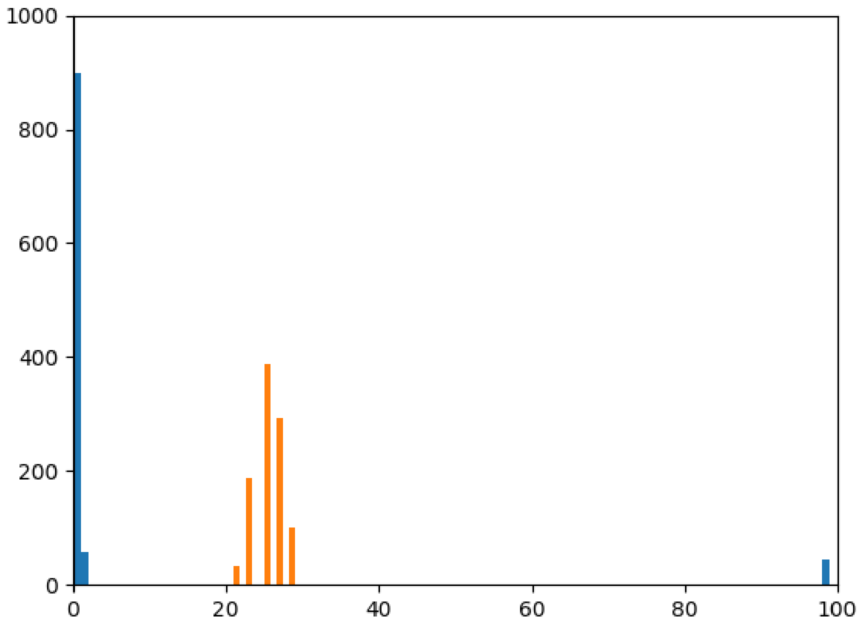
1



2



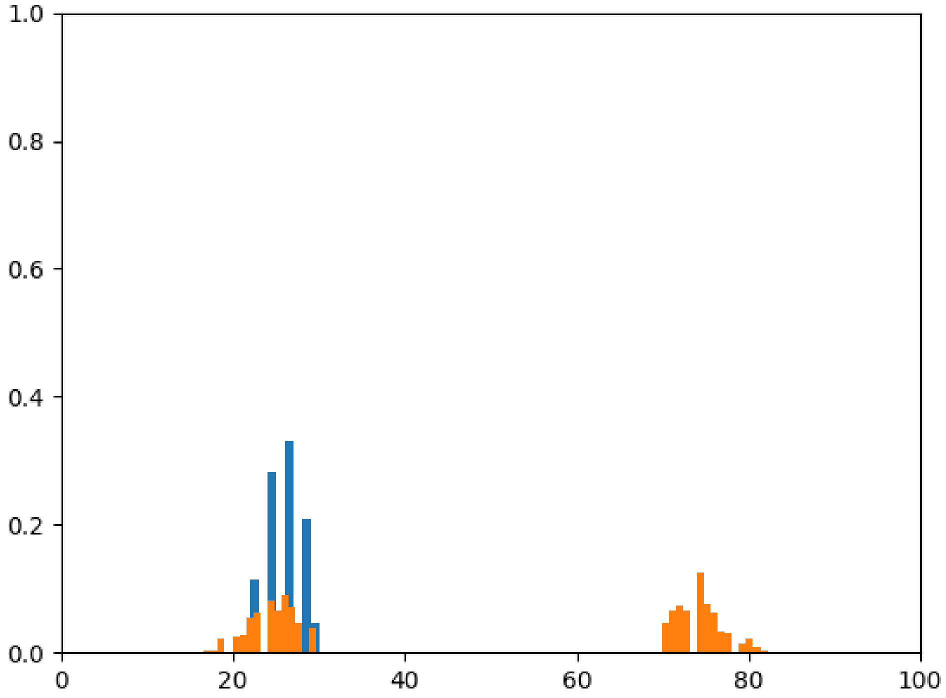
3\4



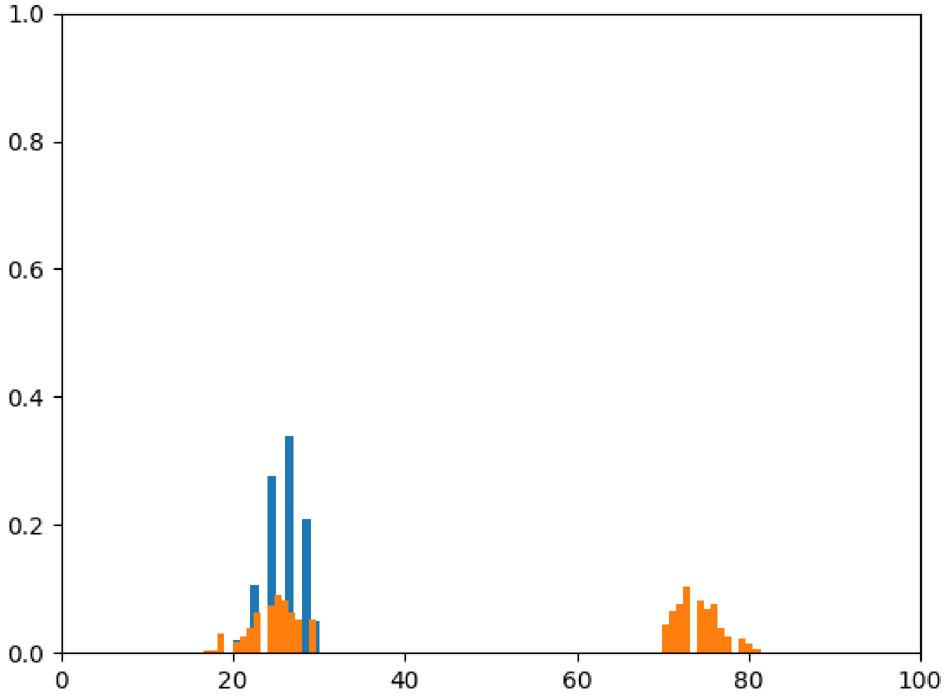
In case 3 and case 4,cause L is close to zero, the measurement values are ignored by my algorithm in the normalization progress(like using int() function, it will merge small values to zero), so we can see nothing.

(c)

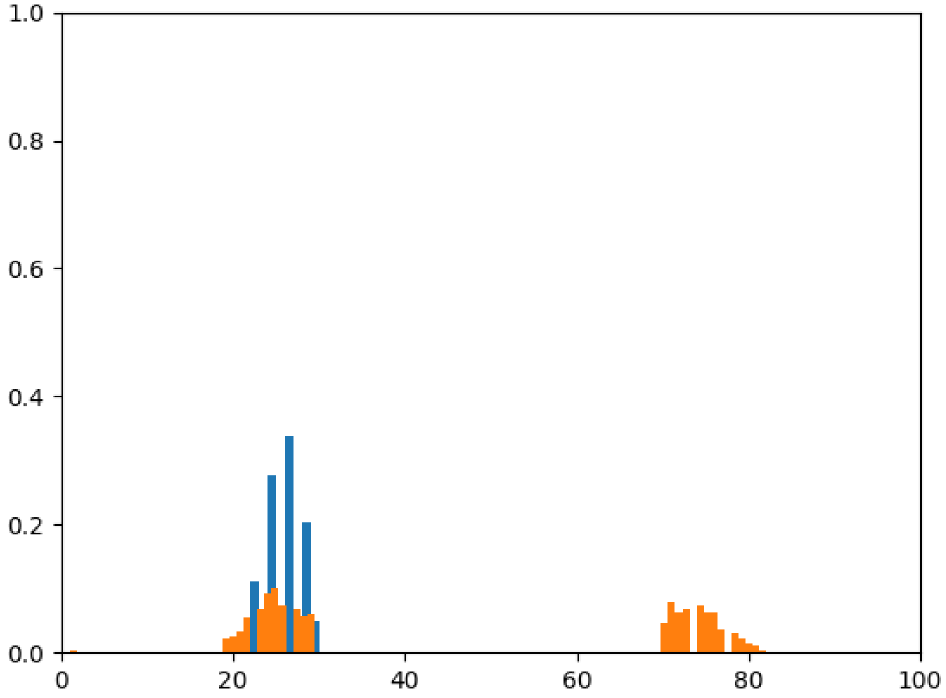
1.



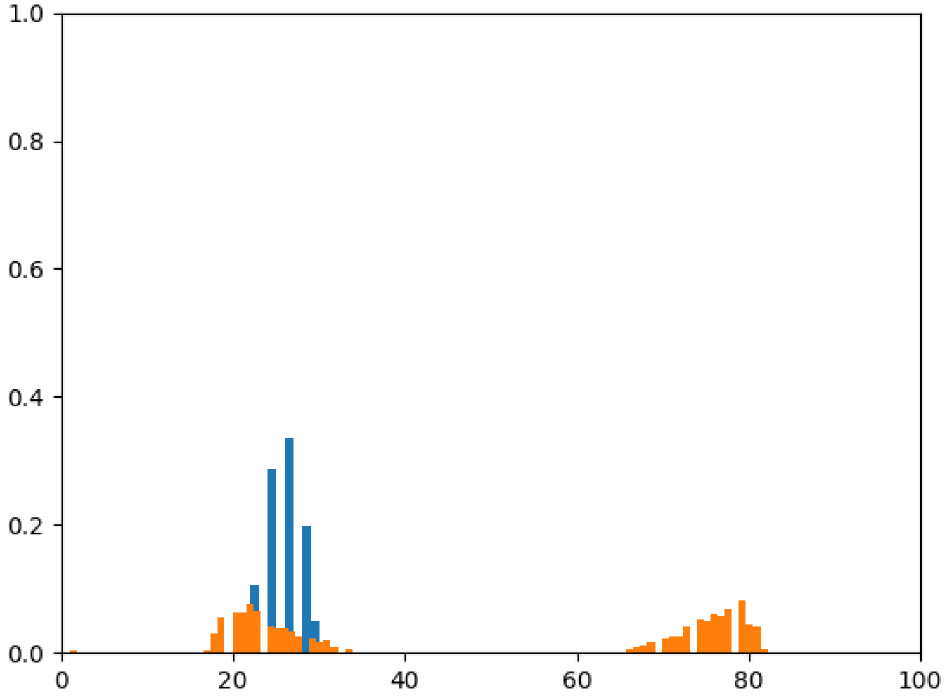
2.



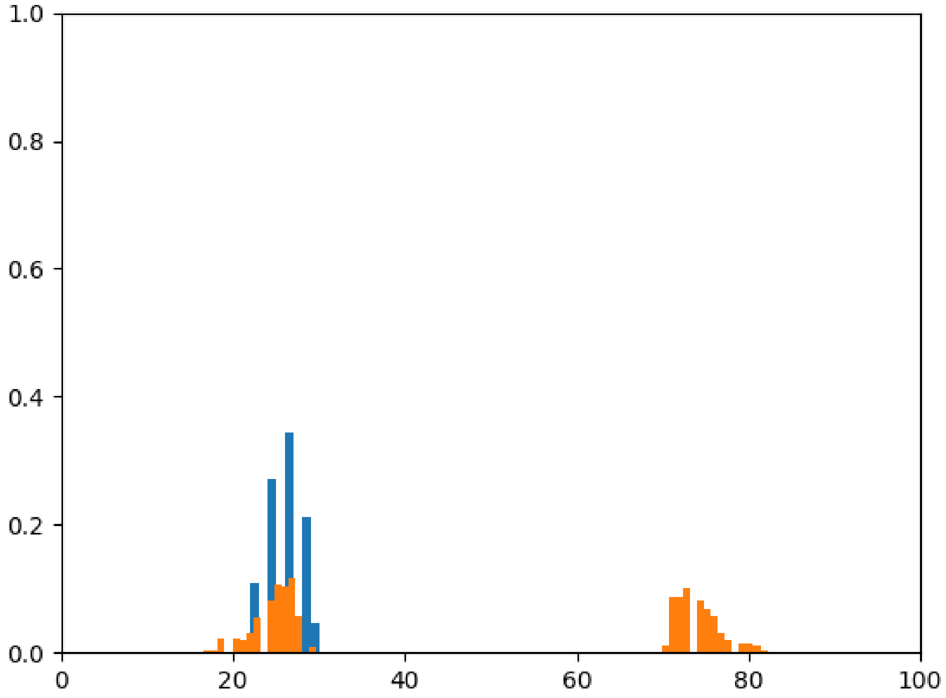
3.



4.



5.



As the figures above, if the difference between p and p' is large, the main estimation value will offset a little(like case 3). So the algorithm is robust if only changing p. But if the e' is much larger than e, the result will show a big offset(like case 4).

attachment-Estimate.py

```
import numpy as np
import matplotlib.pyplot as plt


##############################basic
functions##############################
def x_2_next_x(x, N, p):
    aa = np.random.binomial(n=1, p=p, size=1)
    if aa[0] == 1:
        v_k = 1
    else:
        v_k = -1
    x_k = (x + v_k) % N
    return x_k


def x_2_z(x_k, N, e, L):
    theta_k = 2 * np.pi * x_k / N
    w_k = np.random.uniform(-e, e)
    z_k = ((L - np.cos(theta_k)) ** 2 + (np.sin(theta_k)) ** 2) **
0.5 + w_k
    return z_k


def z_2_x(avg, z_k, L, N, e):
    # deal with noise
    if z_k > avg + e / 2: z_k += np.random.uniform(-e, 0)
    # if z_k<avg-e: z_k+=np.random.uniform(0, e)
    theta = np.arccos((L ** 2 - z_k ** 2 + 1) / (2 * L))
    if theta < 0:
        theta += 2 * np.pi
    if theta > 2 * np.pi:
        theta -= 2 * np.pi
    # we cant tell whether the obj is above zero or below zero
    x_1 = theta * N / (2 * np.pi)
    x_2 = (2 * np.pi - theta) * N / (2 * np.pi)
    return x_1, x_2


####################################params########################
############
sample_num = 10000
k = 10
```

```
N = 100
x0 = N / 4
e = 0.5
L = 2
p = 0.5
# parms in question(c)
ee = e
pp = p

##################################init###########################
###############
# init estimition distribution using uniform
x_m_distribute = np.empty([sample_num], dtype=int)
for i in range(sample_num):
    x_m_distribute[i] = np.random.uniform(0, N)

# init simulation distribution using x0
x_d_distribute = x0 * np.ones(sample_num, dtype=int)

# plt
plt.hist(x_d_distribute, density=1)
plt.hist(x_m_distribute, bins=N, density=1)
plt.xlim(0, N)
plt.ylim(0, 1)
plt.show(block=False)
plt.pause(1)
plt.clf()

####################start simulation and
estimation####################################
for step in range(k):
    # init 2 temp distribution
    x_distribute = np.ones([sample_num], dtype=int)
    z_distribute = np.ones([sample_num], dtype=float)
    for i in range(sample_num):
        # using motion model to calculate the simulation result
        x_d_distribute[i] = x_2_next_x(x_d_distribute[i], N, p)
        # using simulation result and measurement model to get temp z
distribution
        z_distribute[i] = x_2_z(x_d_distribute[i], N, e, L)
    # use a little trick to narrow the z distribution by reduce
noise(more close to the avg)
    avg = sum(z_distribute) / len(z_distribute)
    # use measurement model to calculate x distribution(using
```

```
estimation pp adn ee)
    for i in range(sample_num):
        x1, x2 = z_2_x(avg, z_distribute[i], L, N, ee)
        # there are some strange errors, ignore
        try:
            x_distribute[i] = int(x1 if np.random.binomial(n=1, p=0.5,
size=1) else x2)
        except:
            x_distribute[i] = 0
    # normalization and bayes
    # transform value distribution to possibility distribution
    p1 = np.zeros([N], dtype=int)
    for i in range(N):
        for j in range(sample_num):
            if x_distribute[j] == i:
                p1[i] += 1
    p2 = np.zeros([N], dtype=int)
    for i in range(N):
        for j in range(sample_num):
            if x_m_distribute[j] == i:
                p2[i] += 1
    for i in range(N):
        p1[i] = p1[i] * p2[i]
    # multiply x estimation distribution and x simulation
distribution
    p3 = np.zeros([N], dtype=int)
    for i in range(N):
        p3[i] = (p1[i] / p1.sum()) * sample_num
    # transform possibility distribution to value distribution(for
loop)
    x_m_distribute = np.ones([sample_num], dtype=int)
    count = 0
    for i in range(N):
        for j in range(int(p3[i])):
            if count == sample_num: break
            x_m_distribute[count] = i
            count += 1
    # using motion model to calculate next step
    for i in range(N):
        x_m_distribute[i] = x_2_next_x(x_m_distribute[i], N, pp)
    # plt
    plt.hist(x_d_distribute, density=1)
    plt.hist(x_m_distribute, bins=N, density=1)
    plt.xlim(0, N)
```

```
plt.ylim(0, 1)
plt.show(block=False)
plt.pause(1)
plt.clf()
```