简单的二维平面粒子滤波定位

一、粒子滤波

粒子滤波的思想基于蒙特卡罗方法，是通过寻找一组在状态空间中传播的随机样本（粒子）来近似表示概率密度函数，用样本均值替代积分运算，从而获得系统状态的最小方差估计的过程。其核心思想是通过从后验概率中抽取随机状态粒子来表示其分布。

粒子滤波方法的基本步骤如下：

1. 随机生成一组粒子群`create_uniform_particles`(或使用`create_gaussian_particles`)

```
import numpy as np

def create_uniform_particles(x_range, y_range, hdg_range, N):
    '''
    参数说明:
        1.x_range 生成粒子的x坐标取值范围
        2.y_range 生成粒子的y坐标取值范围
        3.hdg_range 生成粒子的朝向 (heading_degree)取值范围
        4.N 粒子个数
    '''

    #生成N×3矩阵，存放N个粒子的二维坐标+朝向(服从均匀分布)
    particles = np.empty((N,3))
    particles[:, 0] = uniform(x_range[0], x_range[1], size = N)
    particles[:, 1] = uniform(y_range[0], y_range[1], size = N)
    particles[:, 2] = uniform(hdg_range[0], hdg_range[1], size = N)
    #将朝向映射到[0,1]的区间
    particles[:, 2] %= 2 * np.pi
    return particles

def create_gaussian_particles(init_state, noise, N):
    '''
    参数说明:
        1.init_state 机器人初始位置
        2.noise 高斯噪声
        3.N 粒子个数
    '''

    #生成N×3矩阵，存放N个粒子的二维坐标+朝向(服从高斯分布)
    particles = np.empty((N, 3))
    particles[:, 0] = init_state[0] + (randn(N) * noise[0])
    particles[:, 1] = init_state[1] + (randn(N) * noise[1])
    particles[:, 2] = init_state[2] + (randn(N) * noise[2])
    #将朝向映射到[0,1]的区间
    particles[:, 2] %= 2 * np.pi
    return particles
```

2. 预测粒子的下一状态 `predict_ptc_state`

```
def predict_ptc_state(particles, step_displacement, noise,  N):
    '''
    参数说明:
        1.particles 已生成的粒子群
        2.step_displacement 每步位移
        3.noise 噪声
        4.N 粒子个数
    '''

    N = len(particles)
    # 设置粒子移动方向（加高斯噪声）
    particles[:, 2] += step_displacement[0] + (randn(N) * noise[0])
    particles[:, 2] %= 2 * np.pi


    # 设置粒子移动距离（加高斯噪声）
    dist = step_displacement[1] + (randn(N) * noise[1])
    particles[:, 0] += np.cos(particles[:, 2]) * dist
    particles[:, 1] += np.sin(particles[:, 2]) * dist
```

3. 更新粒子权值 `update_ptc_weight`

```
def update_ptc_weight(particles, weights, rd, err, landmarks):
    '''
    参数说明:
        1.particles 已生成的粒子群
        2.weights 权值
        3 rd 机器人与路标真实距离（含高斯噪声）
        4.err 噪声参数
        5.landmarks 路标
    '''

    weights.fill(1.) #权值默认为1

    #计算与路标之间距离，并以此分配权值
    for i, landmark in enumerate(landmarks):
        #[0:2]取particles的x，y坐标
        #axis = 1按行求范数
        particles_distance = np.linalg.norm(particles[:, 0:2] - landmark, axis=1)
        #scipy.stats.norm(particles_distance,err)
        #生成期望值distance，标准差R的正态分布
        #pdf(rd[i])获得该处的概率密度作为权值
        weights *= scipy.stats.norm(particles_distance, err).pdf(rd[i])

    weights += 1.e-300      # 避免权重过小被舍为0
    weights /= sum(weights) # 将权值单位化
```

4. 重取样

```
def num_of_effected_particles(weights):
    '''
    参数说明:
        1.weights 权值
    '''

    #判断是否需要重取样
    return 1. / np.sum(np.square(weights))

def simple_resample(particles, weights):
    '''
    参数说明:
        1.particles 已生成的粒子群
        2.weights 权值
    '''
    N = len(particles) #现有粒子个数
    cumulative_sum = np.cumsum(weights) #权值排序
    cumulative_sum[-1] = 1.     # 避免计算求和时舍入误差
    indexes = np.searchsorted(cumulative_sum, random(N)) #计算权值满足条件的粒子索引值

    # 通过索引值进行重取样
    particles[:] = particles[indexes]
    weights[:] = weights[indexes]
    weights /= np.sum(weights)  # 单位化
```

5. 计算估计值

```
def estimate(particles, weights):
    '''
    参数说明:
        1.particles 已生成的粒子群
        2.weights 权值
    '''

    #返回均值，方差
    pos = particles[:, 0:2]
    mean = np.average(pos, weights=weights, axis=0)
    var  = np.average((pos - mean)**2, weights=weights, axis=0)
    return mean, var
```

二、定位

定位函数主要实现设置参数，给出符合题目情景的条件等功能，通过调用上述粒子滤波算法的函数来完成定位任务。

```
def run_pf(N, moveSteps=18, sensor_noise_err=0.05, xlim=(0, 20), ylim=(0, 20)):
    # 设置路标（此处设置六个）
    landmarks = np.array([[-1, 2], [3, 9], [5, 15] ,[9, 13], [12, 18], [18,21]])
    number_of_landmarks = len(landmarks)
```

```
    # 设置粒子及权值
    particles = create_uniform_particles((0,20), (0,20), (0, 2*np.pi), N)
    weights = np.zeros(N)   #create the weight of the particles(initialized with 0)

    predict_particles = []   # estimated values
    robot_pos = np.array([0., 0.])  # create positon array as [x, y]

    #设置地图[20,20]，步数18，移动方向速度[1,1]
    for x in range(moveSteps):
        robot_pos += (1, 1)

        # distance from robot to each landmark
        real_distance = np.linalg.norm(landmarks - robot_pos, axis=1) # real distance
        real_distance += randn(number_of_landmarks) * sensor_noise_err # add gaus noise

        # move particles forward to (x+1, x+1)
        predict_ptc_state(particles, step_displacement=(0.00, 1.414), noise=(.2, .05),
N=N)

        # incorporate measurements
        update_ptc_weight(particles, weights, rd=real_distance, err=sensor_noise_err,
landmarks=landmarks)

        # resample if too few effective particles
        noep = num_of_effected_particles(weights)
        print 'noep = %d'%noep
        print 'weight4='
        print weights
        if num_of_effected_particles(weights) < N/2:
            simple_resample(particles, weights)

        # Computing the State Estimate
        mu, var = estimate(particles, weights)
        print 'estimated position and variance:\n\t', mu, var
        predict_particles.append(mu)
```

三、可视化

为便于看出定位效果，采用`python`库`matplotlib`进行二维可视化，代码及效果图如下：

```
def all_plot(predict_particles, moveSteps, landmarks):
    predict_particles = np.array(predict_particles)
    #plot real path
    plt.plot(np.arange(moveSteps+1),'k+')
    #plot predicted path
    plt.plot(predict_particles[:, 0], predict_particles[:, 1],'r.')
    # plot landmarks
    plt.scatter(landmarks[:,0],landmarks[:,1],alpha=0.4,marker='o',c=randn(6),s=100)
    #plot legend
    plt.legend( ['Actual','PF'], loc=6, numpoints=1)
    #plot map
    plt.xlim([-2,22])
    plt.ylim([-2,22])
```

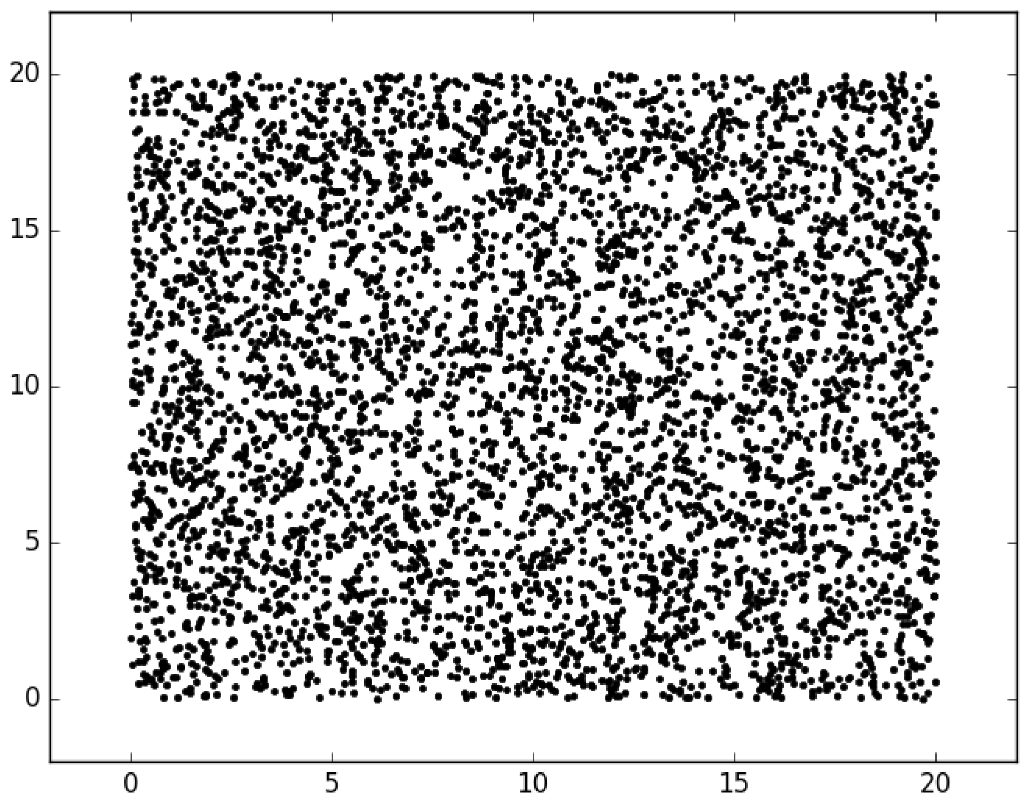
```
    #show all
    plt.show()

def particles_plot(particles):
    plt.xlim([-2,22])
    plt.ylim([-2,22])
    plt.plot(particles[0],particles[1],'r.')
    #r. - red , b. - blue , k. - black
    plt.show()
```

为直观观察粒子变化，将`particles_plot`函数嵌入（一）中各个函数，在每个函数更新粒子后输出图像，得到分步结果如下（N=5000，采用均匀分布）：

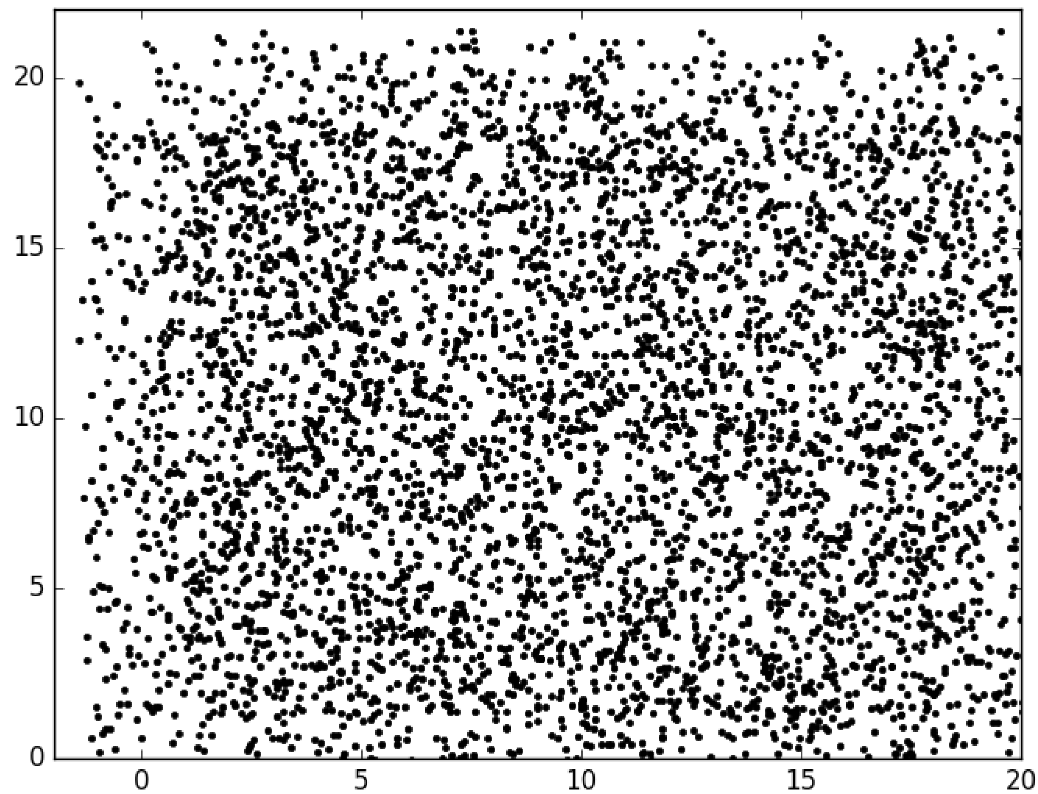
1. create_uniform_particles

   在[0,20],[0,20]范围内生成均匀分布的粒子，每个粒子都代表机器人一个可能的位置（粒子朝向在图中未显示）



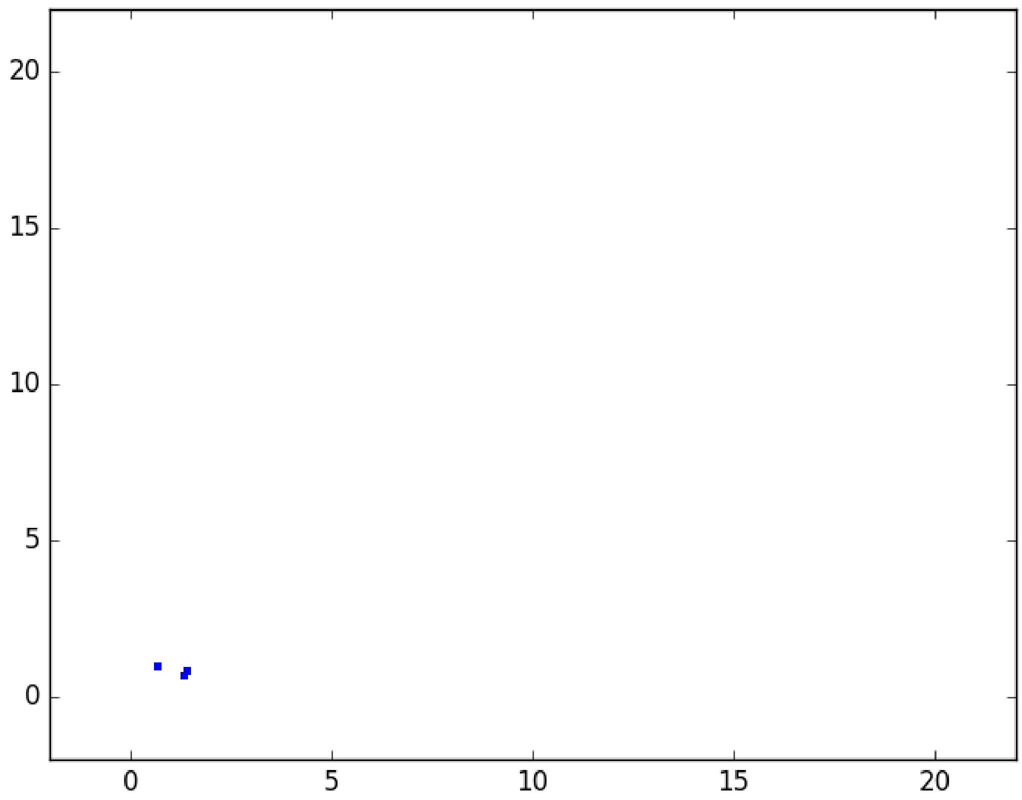
2. predict_ptc_state

   根据输入的转向，速度预测机器人下一时刻位置（由于实际输入存在误差，故将粒子的转向及速度均加入来高斯噪声，以模拟真实情况）



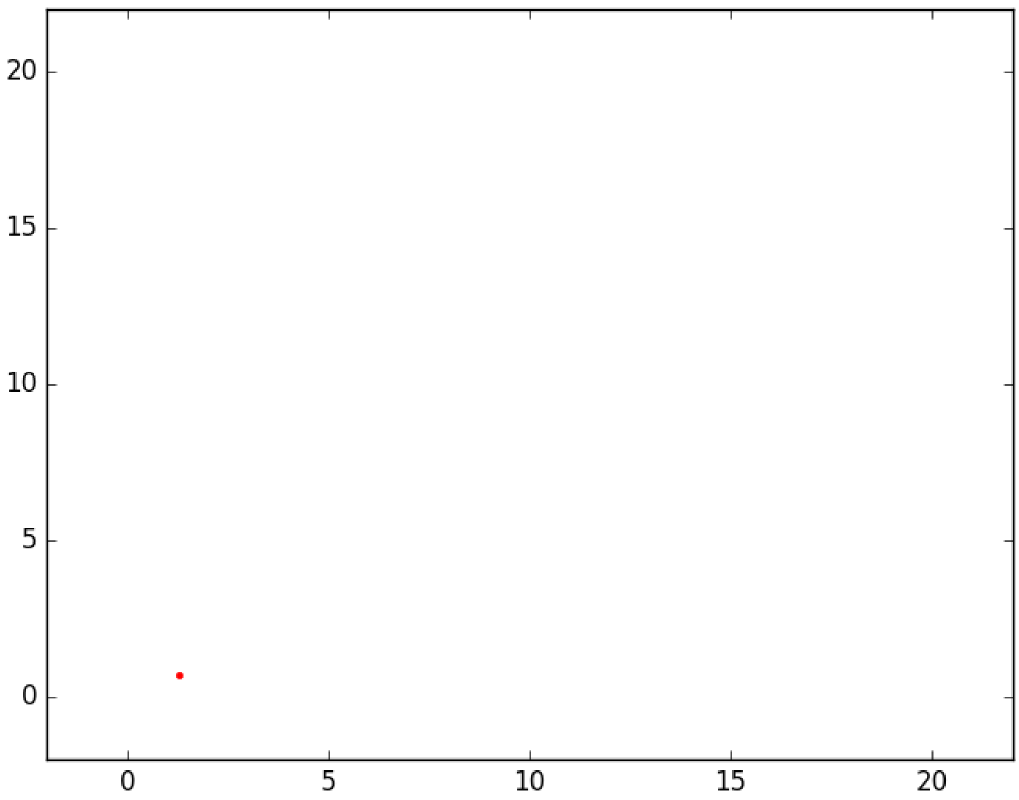
3. simple_resample

在重取样中计算了满足条件的索引值，它们代表的是生成的粒子中较为准确的一部分（可能是一个），此时舍去了大部分不准确点，代之以准确点。图像中显示的每个点都代表多个粒子，粒子总数不变（5000）



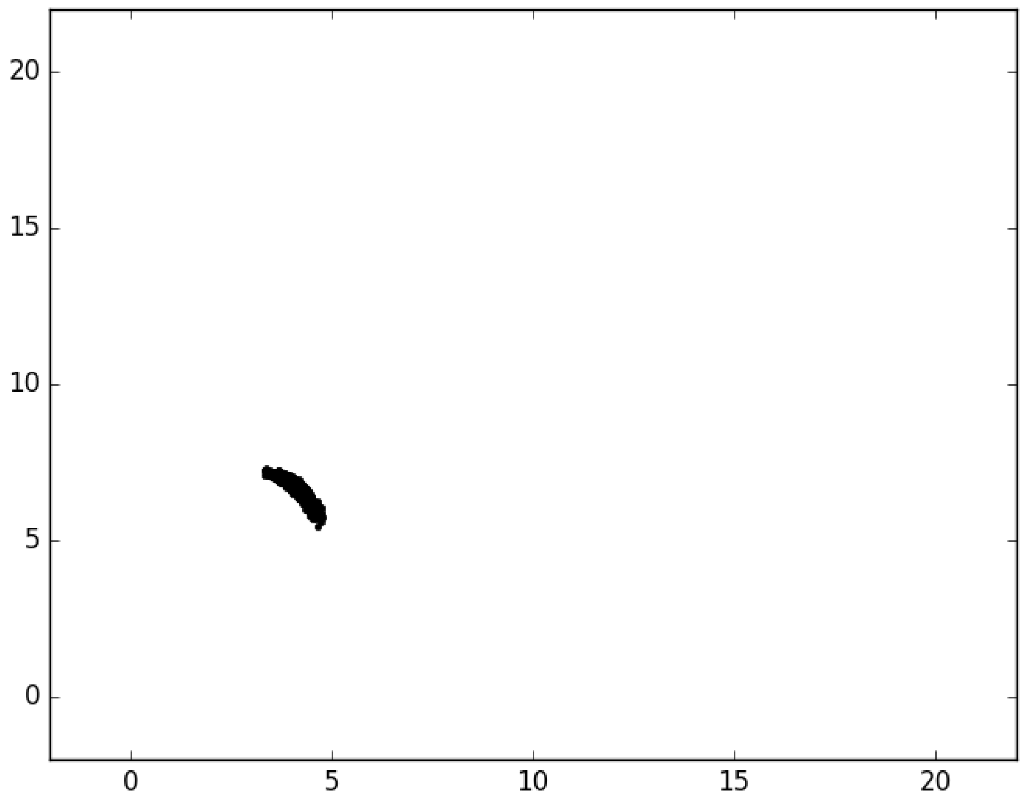
4. estimate

将重取样所得粒子的坐标平均值作为最终预测值，同时计算并返回粒子坐标及方差，以显示预测准确度。



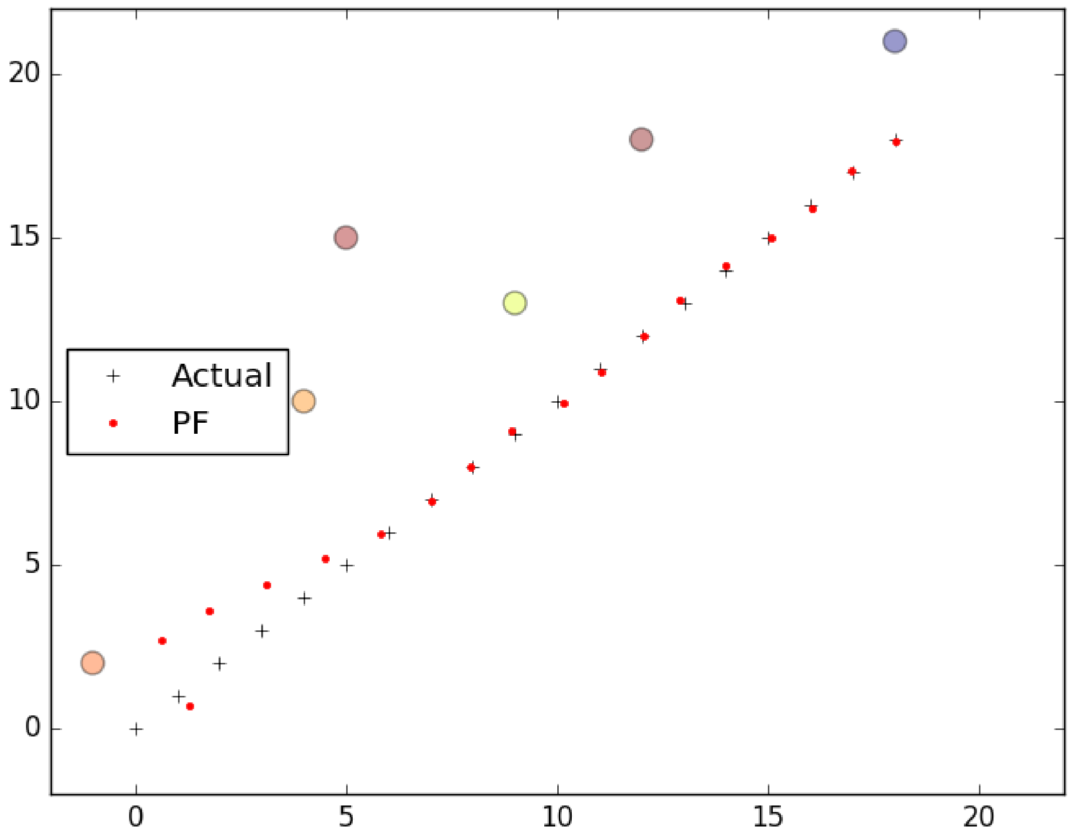
5. 重复2-4中步骤

截取其中一段如动图所示（黑色predict，蓝色：



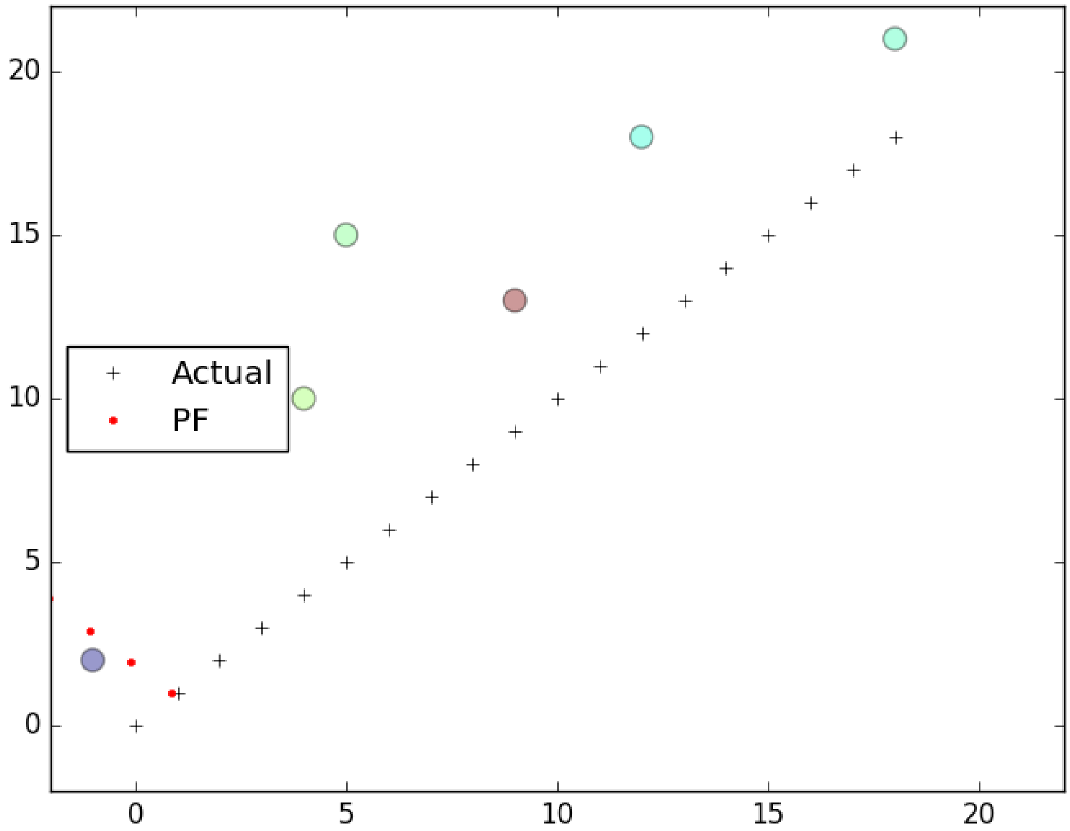
6. 预测结果

其中红点为预测机器人所在位置，黑色十字为理论实际位置，彩色圆为路标。可以看出随着迭代次数的增加，准确度逐渐提升并稳定在误差允许范围内。



四、修正

在测试代码时发现，并不是每次运行都能得到准确的预测值，一些情况下预测值会发生较大的偏移，如下图所示：



分析原因发现，第一步生成粒子时采用的是`均匀分布`，故而生成的粒子位置，朝向皆为均匀分布；而机器人能够测量的只有其与路标间距（仅位置维度存在反馈），输入转角与速度时又加入了噪声，故在迭代前期可能会发生预测偏移。

解决方法一是采用`高斯分布`生成粒子，即使用`create_gaussian_particles`函数生成粒子；测试结果表明，仅仅使用`高斯分布`生成粒子的朝向即可。使用该方法的前提是机器人的初始状态（或初始朝向）已知。

解决方法二是在`update_ptc_weight`函数中加入朝向维度的反馈（体现为朝向影响权值）的语句如：

```
weights *= scipy.stats.norm(particles[:,2], err*5).pdf(0.125) #45o映射到[0,1]为0.125
```

五、完整代码

```
import numpy as np
import scipy.stats
from numpy.random import uniform,randn,random
import matplotlib.pyplot as plt


def create_uniform_particles(x_range, y_range, hdg_range, N):
    #set params of particles:
    #x_range = [0,20]
    #y_range = [0,20]
    #heading_degree_range = [0,2*np.pi]
```

```
    particles = np.empty((N, 3))    #create an empty matrix for N particles and each
particle has 3 dimensions
    particles[:, 0] = uniform(x_range[0], x_range[1], size=N)   #dimension 1 is the x
coordinate of the particles
    particles[:, 1] = uniform(y_range[0], y_range[1], size=N)   #dimension 2 is the y
coordinate of the particles
    particles[:, 2] = uniform(hdg_range[0], hdg_range[1], size=N)   #dimension 3 is the
heading degree of the particles
    particles[:, 2] %= 2 * np.pi    #mapping the heading degree to [0, 1]
    return particles    #return the created particles matrix

 def create_gaussian_particles(init_state, noise, N):

   #create an empty matrix for N particles and each particle has 3 dimensions
    particles = np.empty((N, 3))
    particles[:, 0] = init_state[0] + (randn(N) * noise[0])
    particles[:, 1] = init_state[1] + (randn(N) * noise[1])
    particles[:, 2] = init_state[2] + (randn(N) * noise[2])
    #mapping the heading degree to [0, 1]
    particles[:, 2] %= 2 * np.pi
    return particles

def predict_ptc_state(particles, step_displacement, noise, N):
    """ move according to control input step_displacement (palstance, velocity)
    with noise Q (noise heading change, noise velocity)"""

    # update_ptc_weight heading
    particles[:, 2] += step_displacement[0] + (randn(N) * noise[0])
    particles[:, 2] %= 2 * np.pi

    # move in the (noisy) commanded direction
    dist = step_displacement[1] + (randn(N) * noise[1])
    particles[:, 0] += np.cos(particles[:, 2]) * dist
    particles[:, 1] += np.sin(particles[:, 2]) * dist


def update_ptc_weight(particles, weights, rd, err, landmarks):
    weights.fill(1.)
    for i, landmark in enumerate(landmarks): #i for the index and landmark for the
coordinate
        particles_distance = np.linalg.norm(particles[:, 0:2] - landmark, axis=1)
        weights *= scipy.stats.norm(particles_distance, err).pdf(rd[i])
        # weights *= scipy.stats.norm(particles[:,2], err*5).pdf(0.125)

    weights += 1.e-300      # avoid round-off to zero
    weights /= sum(weights) # normalize


def estimate(particles, weights):
    """returns mean and variance of the weighted particles"""

    pos = particles[:, 0:2]
    mean = np.average(pos, weights=weights, axis=0)
```

```
    var  = np.average((pos - mean)**2, weights=weights, axis=0)
    return mean, var


def num_of_effected_particles(weights):
    return 1. / np.sum(np.square(weights))


def simple_resample(particles, weights):
    N = len(particles)
    cumulative_sum = np.cumsum(weights)
    # print cumulative_sum
    cumulative_sum[-1] = 1.     # avoid round-off error
    indexes = np.searchsorted(cumulative_sum, random(N))
    # resample according to indexes
    particles[:] = particles[indexes]
    weights[:] = weights[indexes]
    weights /= np.sum(weights)  # normalize

def all_plot(predict_particles, moveSteps, landmarks):
    predict_particles = np.array(predict_particles)
    plt.plot(np.arange(moveSteps+1),'k+')
    plt.plot(predict_particles[:, 0], predict_particles[:, 1],'r.')
    plt.scatter(landmarks[:,0],landmarks[:,1],alpha=0.4,marker='o',c=randn(6),s=100) #
plot landmarks
    plt.legend( ['Actual','PF'], loc=6, numpoints=1)
    plt.xlim([-2,20])
    plt.ylim([0,22])
    plt.show()

def particles_plot(particles):
    plt.xlim([-2,22])
    plt.ylim([-2,22])
    plt.plot(particles[0],particles[1],'r.')
    #r. - red , b. - blue , k. - black
    plt.show()

def run_pf(N, moveSteps=18, sensor_noise_err=0.05, xlim=(0, 20), ylim=(0, 20)):
    landmarks = np.array([[-1, 2], [3, 9], [5, 15] ,[9, 13], [12, 18], [18,21]])
#set 6 landmarks
    number_of_landmarks = len(landmarks)

    # create particles and weights
    particles = create_uniform_particles((0,20), (0,20), (0, 2*np.pi), N)
    weights = np.zeros(N)   #create the weight of the particles(initialized with 0)

    predict_particles = []   # estimated values
    robot_pos = np.array([0., 0.])  # create positon array as [x, y]

    #a map of [20, 20] and step by [1, 1] for 18 steps
    for x in range(moveSteps):
        robot_pos += (1, 1)
```

```
        # distance from robot to each landmark
        real_distance = np.linalg.norm(landmarks - robot_pos, axis=1) #the real distance
        real_distance += randn(number_of_landmarks) * sensor_noise_err #add gaussian noise

        # move particles forward to (x+1, x+1)
        predict_ptc_state(particles, step_displacement=(0.00, 1.414), noise=(.2, .05), N=N)

        # incorporate measurements
        update_ptc_weight(particles, weights, rd=real_distance, err=sensor_noise_err, landmarks=landmarks)

        # resample if too few effective particles
        noep = num_of_effected_particles(weights)
        if num_of_effected_particles(weights) < N/2:
            simple_resample(particles, weights)

        # Computing the State Estimate
        mu, var = estimate(particles, weights)
        print 'estimated position and variance:\n\t', mu, var
        predict_particles.append(mu)

    all_plot(predict_particles, moveSteps, landmarks)



if __name__ == '__main__':
    run_pf(N=5000) #create 5000 particles
```