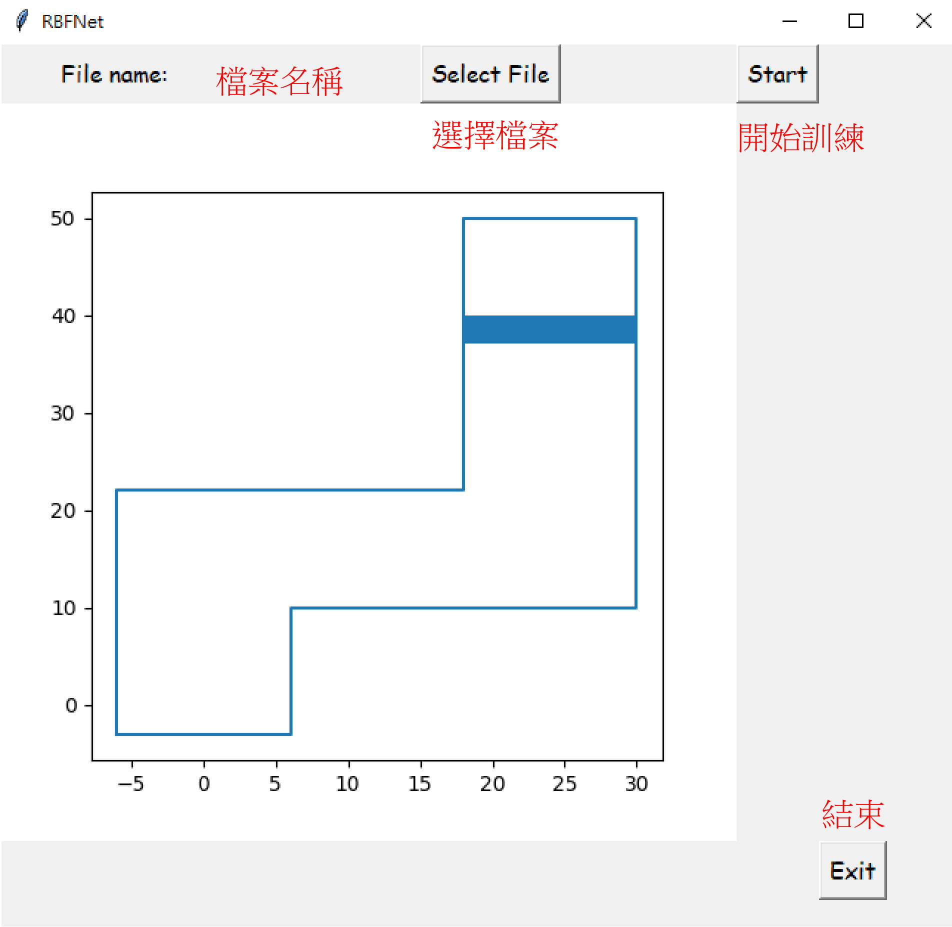
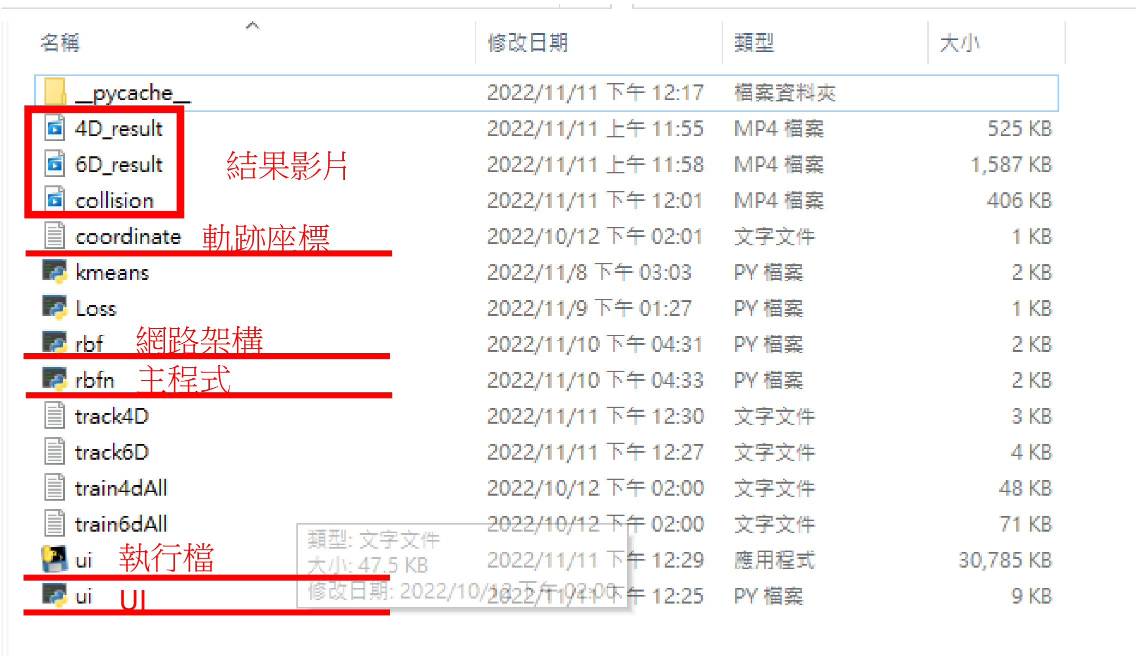
GUI



Brief introduction of code



✧ rbf.py (Network architecture)

```python
class rbf():
    def __init__(self, k, input, input_size, output_size):
        self.input = input
        self.output = None
        self.K = k
        # m: (k, in_dim, 1)
        self.m, self.std = kmeans(self.input, self.K)
        self.weights = np.random.randn(output_size, self.std.shape[1])
        self.bias = np.random.randn(output_size, 1)
        self.input = None

    def forward(self, input):                          輸出phi值
        self.input = input
        # euclidean.T: (k, 1) -> (1, k); same as std: (1, k)
        self.phi = np.exp(-1 * ((np.linalg.norm(self.input - self.m, axis=1).T)**2 / (2 * self.std**2)))
        # w: (1, k), phi: (1, k)
        return np.dot(self.weights, self.phi.T) + self.bias

    def backward(self, output_gradient, learning_rate):   更新參數
        weights_gradient = np.dot(output_gradient, self.phi)
        """ std """
        gradient = np.dot(output_gradient, self.weights)
        gradient = np.dot(gradient, self.phi.T)
        std_gradient = np.dot(gradient, (np.linalg.norm(self.input-self.m, axis=1).T)**2 / self.std**3)
        """ center """
        # gradient: (y-F) x w x phi = constant: (1, 1)
        m_gradient = np.dot((self.input - self.m), gradient)
        # squeeze (axis=(2,)): (k, in_dim, 1) -> (k, in_dim); std.T: (1, k) -> (k, 1)
        m_gradient = np.squeeze(m_gradient, axis=(2,)) / (self.std**2).T
        # reshape: (k, in_dim) -> (k, in_dim, 1)
        m_gradient = np.reshape(m_gradient, (m_gradient.shape[0], m_gradient.shape[1], 1))

        self.weights += learning_rate * weights_gradient
        self.bias += learning_rate * output_gradient    # phi_0 = 1
        self.std += learning_rate * std_gradient
        self.m += learning_rate * m_gradient
```

✧ rbfn.py (Main program; k=5)

```python
def data_process(file):
    with open(file, "r") as f:
        lines = f.readlines()
    data = []
    y = []
    for num, line in enumerate(lines):
        xdata = line.split()
        y.append(float(xdata[-1]))
        for i in range(len(xdata)):
            xdata[i] = float(xdata[i])
        data.append(xdata[0:-1])
        input_dim = len(xdata) - 1
    x_train = np.reshape(data, (len(data), input_dim, 1))
    y_train = np.reshape(y, (len(y), 1, 1))

    # normalizatin    期望輸出值由[-40, 40] normaliz至[0, 1]
    max_y = 40
    min_y = -40
    y_train = (y_train - min_y) / (max_y - min_y)
    return x_train, y_train, input_dim

def process(file, epochs, lr):
    x_train, y_train, input_dim = data_process(file)
    network = rbf(5, x_train, input_dim, 1)  K值設為5
    train(network, lms, lms_prime, x_train, y_train, epochs, lr)
    return network, input_dim
```

✧ kmeans.py (clustering; to determine center and std)

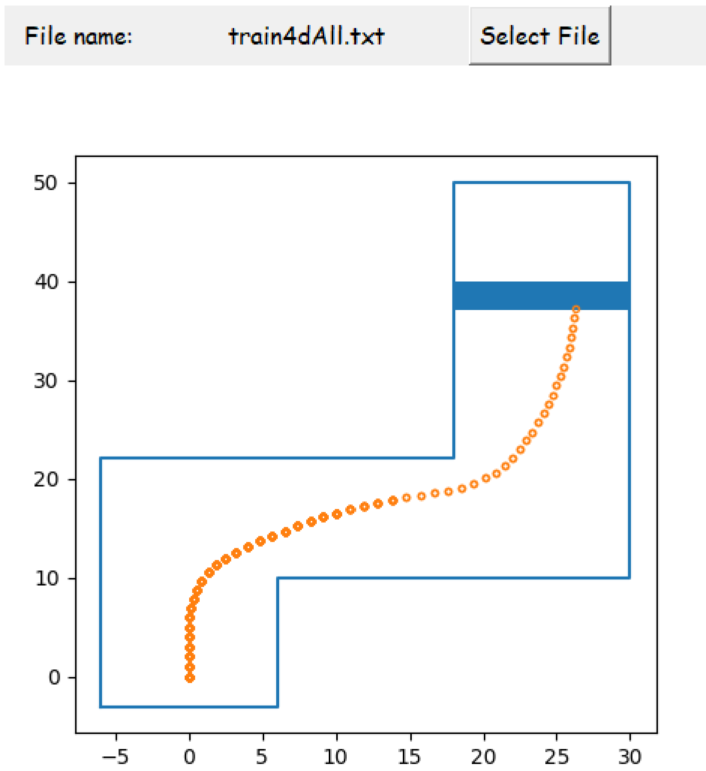
```
def get_distance(c, x):
    sum = 0
    for i in range(len(c)):
        sum += (c[i] - x[i]) ** 2
    return np.sqrt(sum)

def kmeans(X, k, max_iters=10000):
    centroids = X[np.random.choice(range(len(X)), k, replace=False)]
    converged = False
    current_iter = 0
    while (not converged) and (current_iter < max_iters):   最多做10000次
        cluster_list = [[] for i in range(len(centroids))]
        for x in X:  # Go through each data point
            distances_list = []
            for c in centroids:                             將所有點依距離分群
                distances_list.append(get_distance(c, x))
            cluster_list[int(np.argmin(distances_list))].append(x)

        cluster_list = list((filter(None, cluster_list)))
        prev_centroids = centroids.copy()
        centroids = []
        for j in range(len(cluster_list)):
            # use the mean of cluster to re-calculate the new centroid
            centroids.append(np.mean(cluster_list[j], axis=0))
        # pattern: test whether it's the same centroid
        pattern = np.abs(np.sum(prev_centroids) - np.sum(centroids))
        converged = (pattern == 0)  比對中心點有沒有移動
        current_iter += 1

    return np.array(centroids), np.array([[np.std(x) for x in cluster_list]])
```

✧ ui.py (learning rate=0.002, epoch=200)

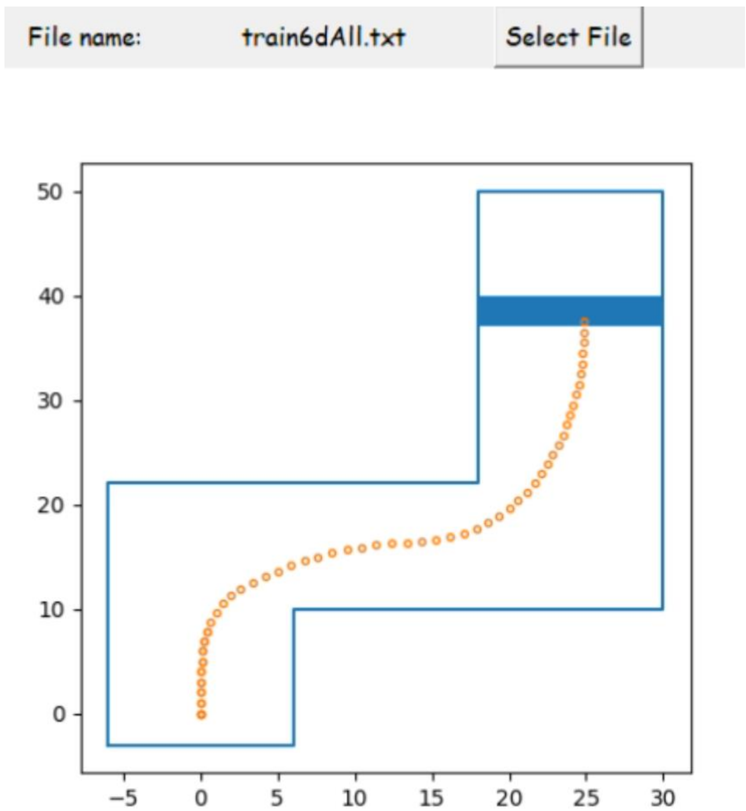
```
    def collision(self, distance):
        判斷有沒有碰撞
    def detect_distance(self, x, y, point='center'):
        以三個sensor的位置，偵測距離
    def point_on_circle(self, x, y, r, angle):
        取得車體長度1/2為半徑的圓上，sensor的位置
    def getLine(self, x1, y1, x2, y2):
        牆面或sensor的直線方程式
    def WallsIntersection(self, a, b, c):
        Sensor跟牆壁的交點
    def intersect_constraint(self, x, y):
        只取sensor跟牆壁方程式第一個交點當作sensor距離
    def wall_constrain(self, x1, y1, x2, y2, x_, y_):
        牆壁方程式只在點跟點的範圍內
    def cal_distance(self, x1, y1, x2, y2):

    def IntersectPoint(self, a, b, c, a1, b1, c1):
```

Result

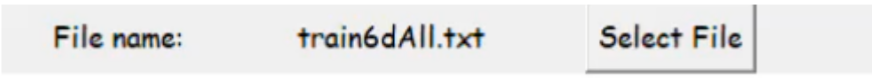
1. train4DAll.txt

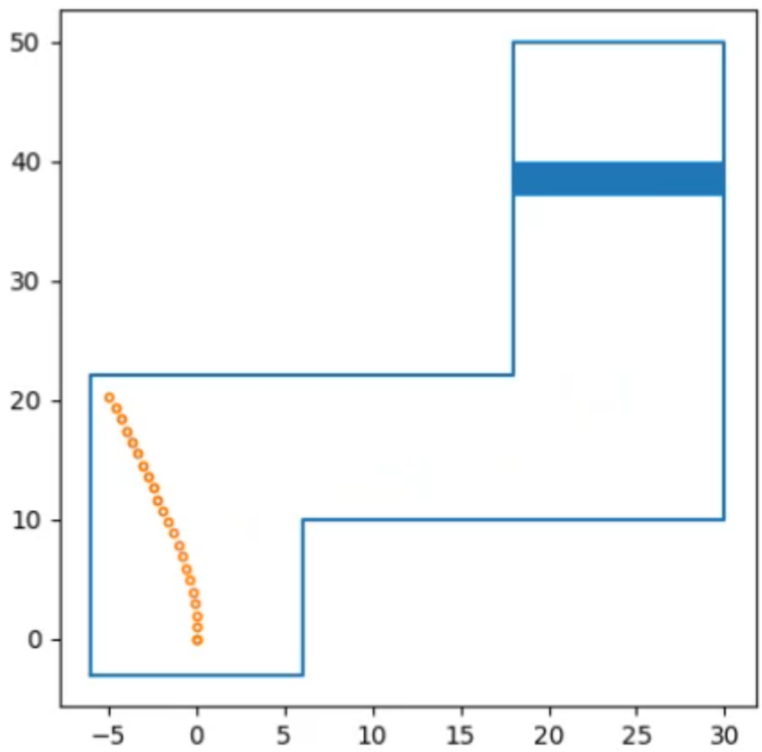


2. train6Dall.txt



3. collision example





Discussion

According to the result on both 4D and 6D, we could obviously observe that there are several parameters here that would make a huge effect on the experiment result, for instance, the k value and the initial centers play a role in the clustering result causing those samples' different clustering convergence, and the learning rate affects the step the model learned.

Because of that, collisions sometimes happened. In mine, there are around 2 to 3 out of 5 times that will arrive at the end successfully, which means, the car stopped when it rushes on the walls also have a high probability. I tried to adjust those parameters to fit the path, however, no matter doing the fine-tuning on these parameters, I got a result with only a little bit of improvement. And I noticed that the reason is, we change the angle of the car wheel could not directly change the angle of the car, it needs some buffer just like driving a real car. For example, when the car is 130 degrees according to the horizontal line, and the car is deviate from the path, we tried to turn/change the angle of the wheel to let the car back to the right way but the car was too closed to the walls, so it cannot turn back to the way even we turn a huge degree of the wheel, then the car will run into the walls. This is the reason why the car sometimes collision with the walls.

Furthermore, the training result of the model also plays an important role in the track of the car which means the bad training causes a bad result depending on the initialization of the “k” centers from the k-means algorithm. A different cluster centroid makes the different performance of the model. So that, sometimes it could fit well but sometimes the result crushes. It must go through spending time fine-tuning to get a better model for this self-driving car.

In addition, I met a problem when I use the animation function from the package matplotlib. It sometimes plots a part of the track in a confusing order, but it still completes the tracking path well. Till now, I cannot figure out what's mattering with it. I guess it’s because I am not familiar with this function. It's a small problem that does not influence the result which could perform the dynamic tracking well, just never mind it, please.