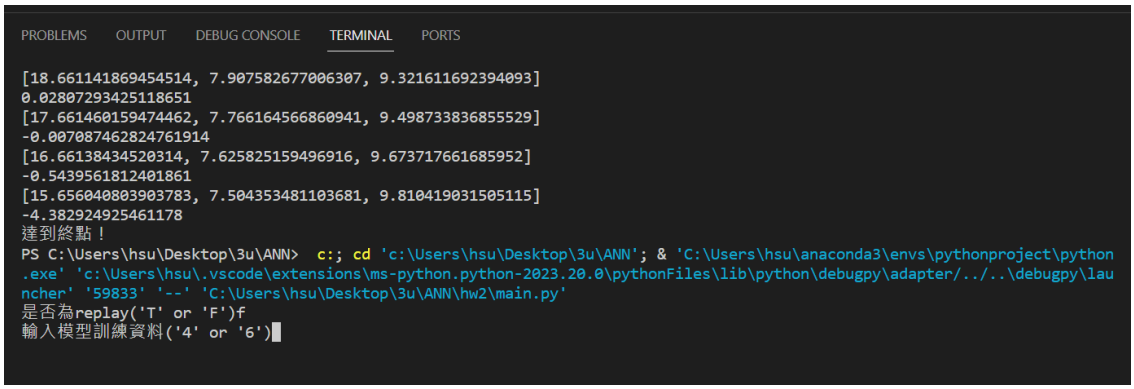
# 類神經網路作業二

110502516 許尚軒

## 一、程式介面說明

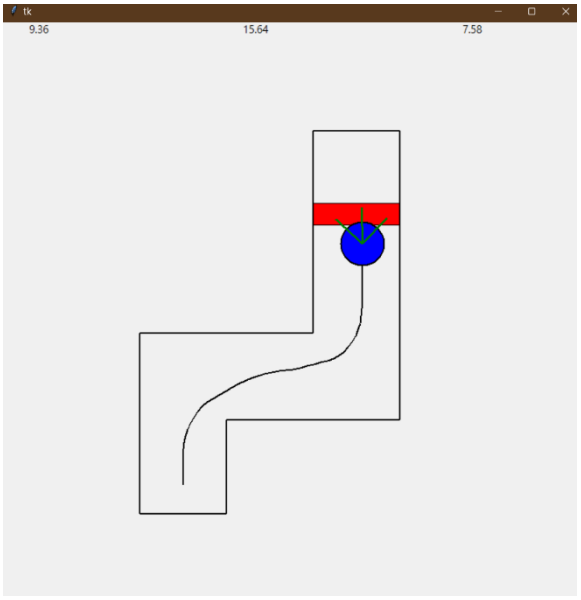
會先問是要重看之前的軌跡還是新訓練一筆資料

再問是要 4d 還 6d




之後會訓練直到 epoch=7000

然後就產生下方自走車模擬圖




## 二、程式碼說明

程式分為 2 個部分，MLP 模型和自走車模擬

**MLP 模型為四層的模型，當輸入為 4d 時各層的神經元數量為 3, 6, 10, 1，當輸入為 6d 時各層的神經元數量為 5, 10, 8, 1**

訓練資料會**打亂**送入模型訓練，**epoch 為 7000 次，學習率為 0.7(1-t/epoch)，慣性項係數為 0.9**

```
hw2 > main.py > ...
389     if is4or6:  #4
390         input_size = 3
391         hidden_size = 6
392         hidden_size1 = 10
393     else:   #6
394         input_size = 5
395         hidden_size = 10
396         hidden_size1 = 8
397     output_size = 1
398     model = MLP(input_size, hidden_size, hidden_size1, output_size)
399         # 訓練數據集
400
401     if getattr(sys, 'frozen', False):
402         current_dir = os.path.dirname(sys.executable)
403     else:
404         current_dir = os.path.dirname(os.path.abspath(__file__))
405
406     if not replay:
407         if is4or6:
408             data_4d = np.loadtxt(os.path.join(current_dir, 'train4DAll.txt'), delimiter=' ')
409             random.shuffle(data_4d)
410             X_train = data_4d[:, :-1]
411             xmax = X_train.max(axis=0)
412             xmin = X_train.min(axis=0)
413             X_train = (X_train - xmin) / (xmax - xmin)  #正規化在0-1之間
414             y_train = data_4d[:, -1] / 80 + 0.5
415         else:
416             data_6d = np.loadtxt(os.path.join(current_dir, 'train6DAll.txt'), delimiter=' ')
417             random.shuffle(data_6d)
418             X_train = data_6d[:, :-1]
419             xmax = X_train.max(axis=0)
420             xmin = X_train.min(axis=0)
421             X_train = (X_train - xmin) / (xmax - xmin)  #正規化在0-1之間
422             y_train = data_6d[:, -1] / 80 + 0.5
423         # 訓練模型
424         model.fit(X_train, y_train, epochs=7000, learning_rate=0.7, momentem_rate=0.9)
```

接下來看一下模型在做甚麼

先以**標準差為 sqrt((前層神經元數目+後層神經元數目) / 6)，平均值為 0** 來隨機生成在小範圍內**均勻分布**之鍵節值，閥值及慣性項皆設為 0

```
15  class MLP:
16      def __init__(self, input_size, hidden_size, hidden_size1, output_size):
17          # 初始化權重和偏差
18          std_dev_W1 = np.sqrt(6 / (input_size + hidden_size))
19          self.weights_input_hidden = np.random.normal(0, std_dev_W1, size=(input_size, hidden_size))
20
21          self.bias_hidden = np.zeros((1, hidden_size))
22
23          std_dev_W12 = np.sqrt(6 / (hidden_size1 + hidden_size))
24          self.weights_hidden_hidden1 = np.random.normal(0, std_dev_W12, size=(hidden_size, hidden_size1))
25
26          self.bias_hidden1 = np.zeros((1, hidden_size1))
27
28          std_dev_W2 = np.sqrt(6 / (output_size + hidden_size1))
29          self.weights_hidden_output = np.random.normal(0, std_dev_W2, size=(hidden_size1, output_size))
30
31          self.bias_output = np.zeros((1, output_size))
32          self.momentum_hiden = np.zeros((input_size, hidden_size))
33          self.momentum_hiden1 = np.zeros((hidden_size, hidden_size1))
34          self.momentum_output = np.zeros((hidden_size1, output_size))
35
```

**前向傳播**中得出實際輸出

```python
    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        return x * (1 - x)

    def forward(self, inputs):
        # 前向傳播
        self.hidden_layer_activation = np.dot(inputs, self.weights_input_hidden) + self.bias_hidden
        self.hidden_layer_output = self.sigmoid(self.hidden_layer_activation)

        self.hidden1_layer_activation = np.dot(self.hidden_layer_output, self.weights_hidden_hidden1) + self.bias_hidden1
        self.hidden1_layer_output = self.sigmoid(self.hidden1_layer_activation)

        self.output_layer_activation = np.dot(self.hidden1_layer_output, self.weights_hidden_output) + self.bias_output
        self.predicted_output = self.sigmoid(self.output_layer_activation)

        return self.predicted_output

```

**倒傳遞**根據此實際輸出與預期輸出的差與其他項目改變鍵節值和閥值

公式為: $$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n)$$

```python
def backward(self, inputs, target, learning_rate, momentem_rate, epoch_rate):
    # 反向傳播
    output_error = target - self.predicted_output
    output_delta = output_error * self.sigmoid_derivative(self.predicted_output)

    hidden1_layer_error = np.dot(output_delta, self.weights_hidden_output.T)
    hidden1_layer_delta = hidden1_layer_error * self.sigmoid_derivative(self.hidden1_layer_output)

    hidden_layer_error = np.dot(hidden1_layer_delta, self.weights_hidden_hidden1.T)
    hidden_layer_delta = hidden_layer_error * self.sigmoid_derivative(self.hidden_layer_output)

    learning_rate = learning_rate * (1 - epoch_rate)
    # 更新權重和偏差
    self.weights_hidden_output += self.momentum_output * momentem_rate + np.dot(self.hidden1_layer_output.T, output_delta) * learning_rate
    self.bias_output += np.sum(output_delta, axis=0, keepdims=True) * learning_rate
    self.weights_hidden_hidden1 += self.momentum_hiden1 * momentem_rate + np.dot(self.hidden_layer_output.T, hidden1_layer_delta) * learni
    self.bias_hidden1 += np.sum(hidden1_layer_delta, axis=0, keepdims=True) * learning_rate
    self.weights_input_hidden += self.momentum_hiden * momentem_rate + np.dot(inputs.T, hidden_layer_delta) * learning_rate
    self.bias_hidden += np.sum(hidden_layer_delta, axis=0, keepdims=True) * learning_rate
    self.momentum_output = self.momentum_output * momentem_rate + np.dot(self.hidden1_layer_output.T, output_delta) * learning_rate
    self.momentum_hiden1 = self.momentum_hiden1 * momentem_rate + np.dot(self.hidden_layer_output.T, hidden1_layer_delta) * learning_rate
    self.momentum_hiden = self.momentum_hiden * momentem_rate + np.dot(inputs.T, hidden_layer_delta) * learning_rate
```

訓練會重複 epoch 次前向傳播和倒傳遞並計算出均方誤差及輸出相似度

$(|d - y| / y)_{av}$

```python
def fit(self, X, y, epochs, learning_rate, momentem_rate):
    for epoch in range(epochs):
        a = 0
        b = 0
        for inputs, target in zip(X, y):
            inputs = np.array(inputs).reshape(1, -1)
            target = np.array(target).reshape(1, -1)

            # 前向傳播
            yi = self.forward(inputs)
            dif = abs(yi[0][0] - target[0][0])
            if target[0][0] != 0:
                rate = (target[0][0] - dif) / target[0][0]
            else:
                rate = 0
            if rate < 0 or rate == math.inf:
                rate = 0
            b += rate
            a += dif ** 2
            # 反向傳播
            self.backward(inputs, target, learning_rate, momentem_rate, epoch/epochs)
        a /= len(y)
        b /= len(y)
        print(str(epoch), str(a), str(b))

def predict(self, X):
    inputs = X.reshape(1, -1)
    return self.forward(inputs)
```

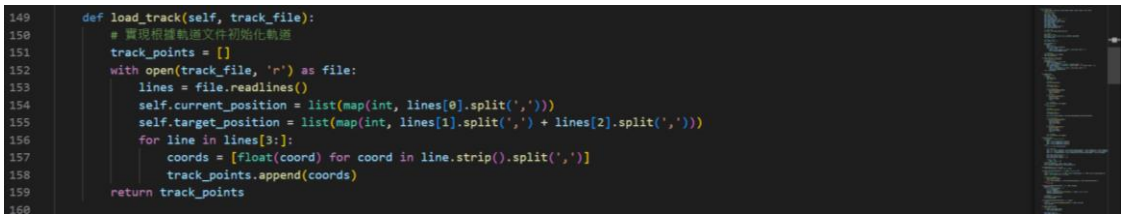
訓練完畢後就可以根據模型模擬自走車(因為 replay 只是根據 txt 檔去重現動畫，因此接下來接不會提到 replay 的作法)

```
    # 設置感測器角度
    sensor_angles = [0, -45, 45]
    if replay:
        if is4or6:
            with open(os.path.join(current_dir, 'track4D.txt'), 'r') as file:
                    with open(os.path.join(current_dir, 'track4.txt'), 'r') as file2:
                        simulation = CarSimulation(os.path.join(current_dir, "軌道座標點.txt"), sensor_angles, model,
        else:
            with open(os.path.join(current_dir, 'track6D.txt'), 'r') as file:
                    with open(os.path.join(current_dir, 'track6.txt'), 'r') as file2:
                        simulation = CarSimulation(os.path.join(current_dir, "軌道座標點.txt"), sensor_angles, model,

    else:
        # 創建模擬
        if is4or6:
            with open(os.path.join(current_dir, 'track4D.txt'), 'w') as file:
                with open(os.path.join(current_dir, 'track4.txt'), 'w') as file2:
                    simulation = CarSimulation(os.path.join(current_dir, "軌道座標點.txt"), sensor_angles, model, is4o
        else:
            with open(os.path.join(current_dir, 'track6D.txt'), 'w') as file:
                with open(os.path.join(current_dir, 'track6.txt'), 'w') as file2:
                    simulation = CarSimulation(os.path.join(current_dir, "軌道座標點.txt"), sensor_angles, model, is4o
```

simulation 一開始會先**初始化**許多變數並載入軌道

```
        self.sensor_distances = []

        # 讀取軌道檔案
        self.map = self.load_track(track_file)

        # 初始化GUI
        self.root = tk.Tk()
        self.canvas = tk.Canvas(self.root, width=800, height=800)
        self.canvas.pack()

        # 初始化感測器顯示
        self.sensor_lines = []

        self.trackdata = []
        if replay:
            self.count = 0
            lines = self.file2.readlines()
            for line in lines:
                coords = [float(coord) for coord in line.strip().split(' ')]
                self.trackdata.append(coords)

        # 設置動畫更新
        self.root.after(0, self.update)
        self.root.mainloop()
```

**載入軌道**如下



```
    def load_track(self, track_file):
        # 實現根據軌道文件初始化軌道
        track_points = []
        with open(track_file, 'r') as file:
            lines = file.readlines()
            self.current_position = list(map(int, lines[0].split(',')))
            self.target_position = list(map(int, lines[1].split(',') + lines[2].split(',')))
            for line in lines[3:]:
                coords = [float(coord) for coord in line.strip().split(',')]
                track_points.append(coords)
        return track_points
```

之後會在每個時間軸**更新**畫面，每次更新會先計算自走車位置再畫圖並計算偵測器偵測距離再根據此距離(以及 x,y)計算方向盤角度，最後偵測是否撞牆或抵達終點

```python
def update(self):
    # 更新模型車位置
    if not replay:
        self.move_car()

        # 繪製模擬
        self.draw_simulation()

        # 更新感測器顯示
        self.update_sensors()

        self.current_steering_angle = self.get_steering_angle_from_neural_network()

        # 檢查是否達到終點
        if self.check_reached_target():
            print("達到終點！")
            self.file.close()
            self.file2.close()
            return
        # 檢查是否撞牆
        if self.check_reached_wall():
            print("撞牆！")
            self.file.close()
            self.file2.close()
            return

        # 繼續更新動畫
        self.root.after(50, self.update)
    else:
```

move_car 會根據公式**計算出新的位置和方向**

```python
def move_car(self):
    # 根據運動方程式更新模型車的位置
    if replay:
        new_x = self.trackdata[self.count][0]
        new_y = self.trackdata[self.count][1]
        new_a = self.trackdata[self.count][2]
    else:
        x, y, a = self.current_position

        # 計算新的位置
        new_x = x + cos(np.radians(a + self.current_steering_angle)) + sin(np.radians(a)) * sin(np.radians(self.current_steering_angle))
        new_y = y + sin(np.radians(a + self.current_steering_angle)) - cos(np.radians(a)) * sin(np.radians(self.current_steering_angle))
        new_a = a - np.degrees(asin(2 * sin(np.radians(self.current_steering_angle)) / self.car_length))

        self.file2.write(str(new_x) + ' ')
        self.file2.write(str(new_y) + ' ')
        self.file2.write(str(new_a) + '\n')

        if new_a > 270:
            new_a =  new_a - 360
    # 更新模型車的位置
    self.current_position = [new_x, new_y, new_a]
    self.track_points.append(self.current_position[:2])
```

draw_simulation 會先清除畫布後畫出終點、軌跡、軌道、自走車，

因為此次作業的 xy 區間都太小，因此*10 放大 10 倍

```python
def draw_simulation(self):
    # 繪製模擬，包括軌道、自走車、感測器等
    self.canvas.delete("all")
    self.canvas.create_rectangle(250+self.target_position[0] * 10, 650-self.target_position[1] * 10,
                                 250+self.target_position[2] * 10, 650-self.target_position[3] * 10, fill='red')  # 繪製終點位置

    for i in range(len(self.track_points) - 1):  # 繪製軌跡
        x1, y1 = self.track_points[i]
        x2, y2 = self.track_points[i + 1]
        self.canvas.create_line(250+x1 * 10, 650-y1 * 10, 250+x2 * 10, 650-y2 * 10, width=2, fill="black")

    for i in range(len(self.map) - 1):   # 繪製軌道
        x1, y1 = self.map[i]
        x2, y2 = self.map[i + 1]
        self.canvas.create_line(250+x1 * 10, 650-y1 * 10, 250+x2 * 10, 650-y2 * 10, width=2, fill="black")

    # 繪製模型車
    x, y, a = self.current_position
    car_x = x + self.car_length / 2
    car_y = y - self.car_length / 2
    x -= self.car_length / 2
    y += self.car_length / 2
    self.canvas.create_oval(250+x * 10, 650-y * 10, 250+car_x * 10, 650-car_y * 10, width=2, fill='blue')
```

update_sensor 會計算出**偵測器測距並顯示**此距離及偵測方向

```python
def update_sensors(self):
    # 實現更新感測器顯示
    for line in self.sensor_lines:
        self.canvas.delete(line)

    self.sensor_lines = []
    self.sensor_distances = []
    xtmp = 0
    # 更新感測器顯示
    for angle in self.sensor_angles:
        # 計算感測器位置
        sensor_x = self.current_position[0] + 5 * cos(np.radians(self.current_position[2] + angle))
        sensor_y = self.current_position[1] + 5 * sin(np.radians(self.current_position[2] + angle))

        # 計算感測到的距離（這裡的邏輯需要根據實際情況進行調整）
        detected_distance = self.calculate_detected_distance(self.current_position[0], self.current_position[1], self.current_position[
        self.sensor_distances.append(detected_distance)
        # 顯示感測器線條
        line = self.canvas.create_line(250+self.current_position[0] * 10, 650-self.current_position[1] * 10,
                                       250+sensor_x * 10, 650-sensor_y * 10, width=3, fill='green')
        self.sensor_lines.append(line)

        # 在GUI中顯示感測到的距離
        self.canvas.create_text((350 + xtmp) % 900, 10, text=f"{detected_distance:.2f}", fill='black')
        xtmp += 300
```

下列四個函式是用來計算偵測器測距，由下而上功用分別為回傳、計算測器偵測每條軌道的距離、偵測器射線與軌道之交點、焦點與自走車距離

```python
hw2 > main.py > CarSimulation > calculate_detected_distance
def distance(self, x1, y1, x2, y2):
    return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)

def point_to_line_distance(self, x, y, angle, x1, y1, x2, y2):
    # 定義兩條線
    line1 = LineString([(x, y), (x + 1000 * cos(np.radians(angle)), y + 1000 * sin(np.radians(angle)))])
    line2 = LineString([(x1, y1), (x2, y2)])

    # 檢查兩條線是否相交
    if line1.intersects(line2):
        # 如果相交，找出相交點
        return self.distance(x, y, line1.intersection(line2).x, line1.intersection(line2).y)
    else:
        return 1e9

def point_to_polygon_distance(self, x, y, angle, polygon):
    distances = []
    for i in range(len(polygon) - 1):
        x1, y1 = polygon[i]
        x2, y2 = polygon[i + 1]
        distance = self.point_to_line_distance(x, y, angle, x1, y1, x2, y2)
        distances.append(distance)

    # 找到最小距離
    min_distance = min(distances)
    return min_distance

def calculate_detected_distance(self, x, y, angle):
    # 計算距離
    distance = self.point_to_polygon_distance(x, y, angle, self.map)
    return distance
```

根據上述測距及 x, y 丟入模型計算出方向盤角度

```python
def get_steering_angle_from_neural_network(self):
    # 使用神經網路模型預測方向盤轉角
    print(self.sensor_distances)

    if self.is4or6:
        steering_angle = self.model.predict((np.array(self.sensor_distances) - xmin) / (xmax - xmin))[0][0]
    else:
        steering_angle = self.model.predict((np.array(self.current_position[:2] + self.sensor_distances) - xmin) / (xmax - xmin))[0][0]
        self.file.write(str(self.current_position[0]) + " ")
        self.file.write(str(self.current_position[1]) + " ")

    self.file.write(str(self.sensor_distances[0]) + " ")
    self.file.write(str(self.sensor_distances[1]) + " ")
    self.file.write(str(self.sensor_distances[2]) + " ")

    steering_angle = (steering_angle - 0.5) * 80

    self.file.write(str(steering_angle) + "\n")
    print(steering_angle)
    return steering_angle
```

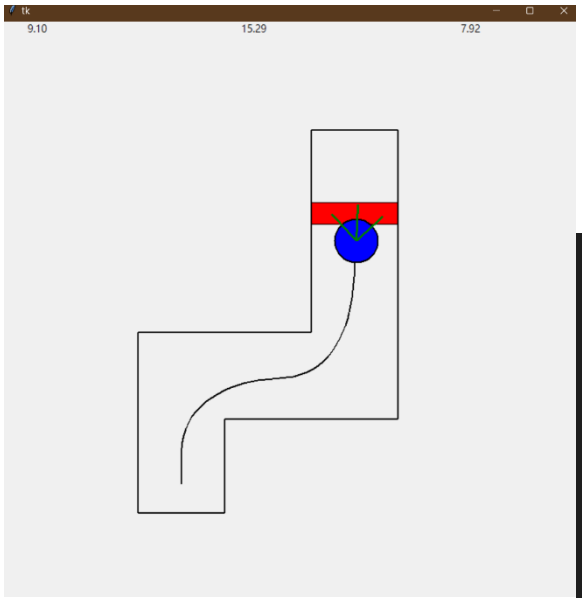
下方為偵測是否抵達終點或是撞牆，方法為判斷**車中心是否進入偵測對象**之 x+-車身半徑, y+-車身半徑之範圍內

```python
def check_reached_target(self):
    # 比較模型車當前位置與終點位置之間的距離，判斷是否達到終點
    if self.current_position[0] <= self.target_position[2] and self.current_position[0] >= self.target_position[0]:
        if self.current_position[1] >= self.target_position[3] - self.car_length / 2 and self.current_position[1] <= self.target_position[1
            return True
    elif self.current_position[1] <= self.target_position[1] and self.current_position[1] >= self.target_position[3]:
        if self.current_position[0] >= self.target_position[0] - self.car_length / 2 and self.current_position[0] <= self.target_position[2
            return True
    return False

def check_reached_wall(self):
    # 檢查是否達到牆
    for i in range(len(self.map) - 1):
        x1, y1 = self.map[i]
        x2, y2 = self.map[i + 1]
        x1, x2 = min(x1, x2), max(x1, x2)
        y1, y2 = max(y1, y2), min(y1, y2)

        if self.current_position[0] <= x2 and self.current_position[0] >= x1:
            if self.current_position[1] >= y2 - self.car_length / 2 and self.current_position[1] <= y1 + self.car_length / 2:
                return True
        elif self.current_position[1] <= y1 and self.current_position[1] >= y2:
            if self.current_position[0] >= x1 - self.car_length / 2 and self.current_position[0] <= x2 + self.car_length / 2:
                return True
    return False
```

## 三、實驗結果

### 4d:

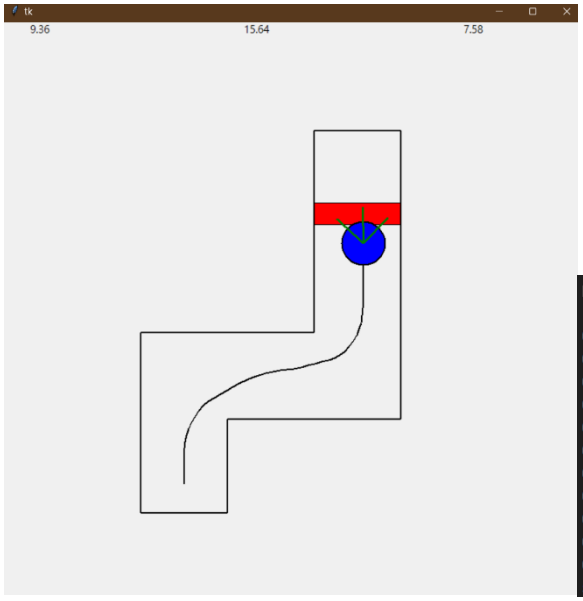
下方圖二每一行代表的是訓練次數、均方誤差、輸出的相似度$(|d - y| / y)_{av}$

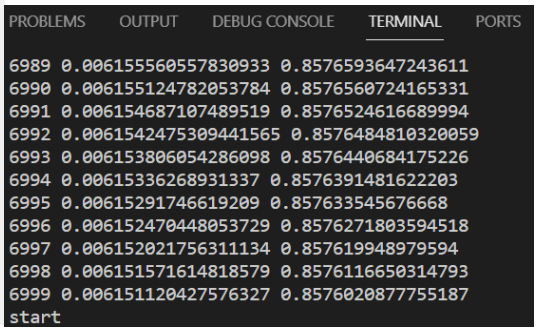



PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
6987 0.011153089895371657 0.8143924129501537
6988 0.011152588093511545 0.8143922764182275
6989 0.011152084024382428 0.8143917666870658
6990 0.011151577685405043 0.8143908260001561
6991 0.011151069077362645 0.8143893858130348
6992 0.011150558206647596 0.8143873643673661
6993 0.011150045088715236 0.8143846636308764
6994 0.011149529753387438 0.8143811654151407
6995 0.011149012253003323 0.8143767264211391
6996 0.011148492674984663 0.8143711718777497
6997 0.011147971161300925 0.8143641168505001
6998 0.011147447938812142 0.814355261511206
6999 0.011146923366933063 0.8143444384857981
start
```

**6d:**




**重現方法:**

執行程式後輸入 t 下一步再輸入要重現軌跡之維度就可以了(track4.txt 及 track6.txt 即為軌跡紀錄(track4D.txt 及 track6D.txt 為作業要求的檔案)

## 四、分析

可以看出來 6d 轉彎角度比較犀利，我認為是因為除了測距外還多了 x, y 可以判斷現在應該轉幾度，因此比起 4d 可以較晚再轉大角度，像是在上述結果中 4d 方向盤角度只在[23, -24]之間，6d 卻在[24, -35]之間，比 4d 大上不少。

而且 6d 的均方誤差也比 4d 少，我認為原因也是相同，當出現測距只相差一點但轉角差很多時比較不會互相影響到。