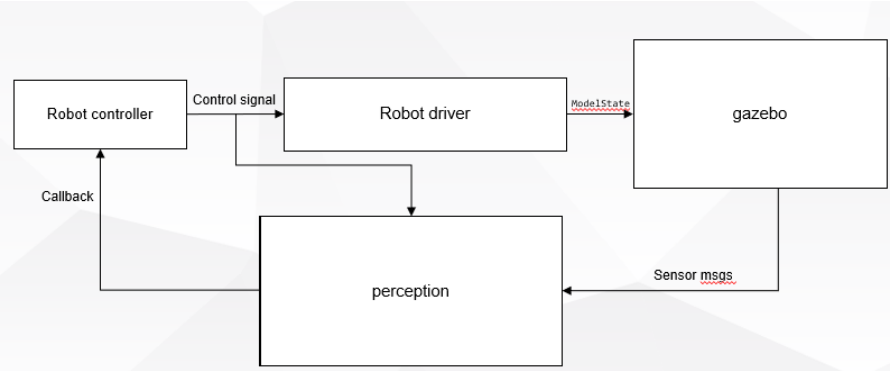
工程实践与科技创新 IV-E
课程报告

姓名：李厚霖　　姓名：

学号：520020910007　　学号：

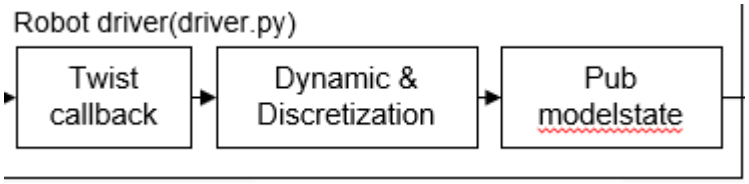
一、系统框架与说明

该实验的系统整体框架如下图所示（采用的是课件 ppt 中的图）。主要分为以下几个模块：用于控制模型小车运动的控制器（Robot controller）；用于获取控制器控制信号，并给模型小车输入不同方向力的执行器（Robot driver）；用于预测和观察的反馈器（perception）;以及构建仿真物理环境的平台 gazebo。



接下来具体说明一下各部分的功能：

- 控制器（Robot controller）：作为控制信号的发出者以及反馈信号的接收者。首先从 perception 中接收到反馈的信息，通过 pid 控制程序给出控制信号实现 pid，从而实现机器人模型的闭环运动控制。
- 执行器（Robot driver）：对应于课程中的质点动力学部分，这里主要建立了机器人的状态空间模型，同时将接收到的控制信号转为 gazebo 中控制机器人移动的力。（具体结构如下图所示）。



- 反馈器（perception）：主要是采用了卡尔曼滤波（Kalman filtering），负责融合预测值与测量值，获得更接近真实值的滤波值。之后再将这些值反馈给控制器进行下一步的 PID 控制。
- 仿真环境平台（gazebo）:作为仿真环境，给出机器人移动与环境的交互并提供传感器的值。

二、算法说明与实现

1. KF 算法及实现

卡尔曼滤波是一种用于估计系统状态的递归滤波器，通过观测数据和先验知识来优化状态估计的精确性。它最初由 R.E. Kalman 在 20 世纪 60 年代提出，并被广泛应用于控制系统、导航、信号处理等领域。

卡尔曼滤波算法基于两个关键假设：线性系统和高斯噪声。它通过将系统的状态建模为一个线性动态模型，并假设系统的观测值和过程噪声都是高斯分布的，从而实现对状态的最优估计。

接下来，介绍一下卡尔曼滤波的基本原理（机器人的状态空间模型已经由老师给出，故不再详细计算分析）：

对于隐马尔科夫模型：

$$x_{t+1} = A_t x_t + B_t u_t + w_t$$

$$y_t = C_t x_t + v_t$$

卡尔曼滤波器算法主要可以预测步骤和更新步骤估计模型的状态分布。对于预测步骤（预测状态和计算协方差）有以下公式：

$$\hat{x}_{t+1|t} = A_t \hat{x}_{t|t} + B_t u_t$$

$$P_{t+1|t} = A_t P_{t|t} A_t^T + \Sigma_w$$

其对应的代码如下图所示：

```python
# predict step
def _predict_Step(self, ctrl_time):
    dt = ctrl_time - self.t # calculate the time interval
    self.t = ctrl_time # update the current time

    At, Bt, Sigma = self._discretization_Func(dt) # get the discrete system matrices

    self.x = At.dot(self.x) + Bt.dot(self.u) # predict the state using the system equation
    self.P = At.dot(self.P).dot(At.T) + Sigma # predict the covariance using the error equation
```

更新步骤（测量更新）的公式为：

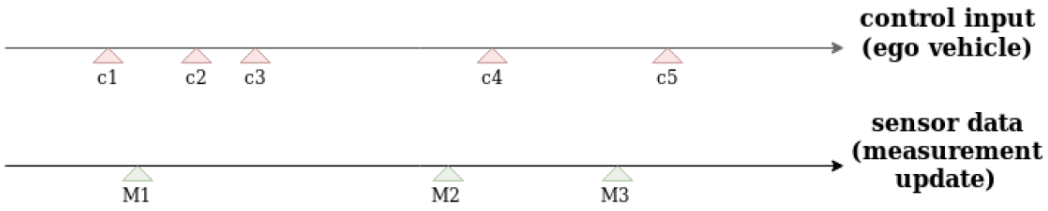
$$\hat{x}_{t+1|t+1} = \hat{x}_{t+1|t} + P_{t+1|t}C^T\left(CP_{t+1|t}C^T + \Sigma_v\right)^{-1}\left(y_{t+1} - C\hat{x}_{t+1|t}\right)$$

$$P_{t+1|t+1} = P_{t+1|t} - P_{t+1|t}C^T(CP_{t+1}C^T + \Sigma_v)^{-1}CP_{t+1|t}$$

其对应的代码如下：

```python
# correction step
def _correction_Step(self, y):
    innovation = self.P.dot(self.C.T).dot(
        np.linalg.inv(self.C.dot(self.P).dot(self.C.T) + self.Sigma_v))  # calculate the Kalman gain
    self.x =  self.x + innovation.dot(y - self.C.dot(self.x)) # correct the state using the measurement residual
    self.P = (np.eye(4) - innovation.dot(self.C)).dot(self.P) # correct the covariance using the Kalman gain
```

理想情况下，采样时间恒定，控制与观测在同一时刻完成，则 Kalman 滤波器只需要在每个采样点执行预测步骤、更新步骤即可。然而在实际机器人实现中，我们不可能做到将控制周期与传感器观测完全同步，如下图所示：



这种情况下，Kalman 滤波器按如下策略执行：

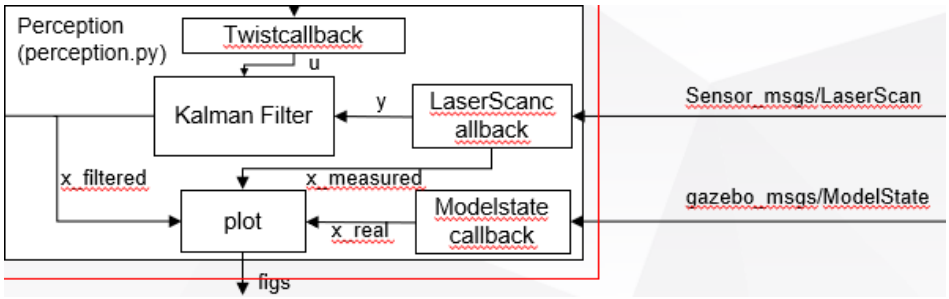
- 在 C2 时刻，$\Delta t = C_2 - M_1$，对 $x_{M1|M1}$ 执行 prediction step，得到 $x_{C2|M1}$。
- 在 C3 时刻，$\Delta t = C_3 - C_2$，对 $x_{C2|M1}$ 执行 prediction step，得到 $x_{C3|M1}$
- 在 M2 时刻，$\Delta t = M_2 - C_3$，先对 $x_{C3|M1}$执行 prediction step，得到 $x_{M2|M1}$。

  再依据 y_{M2} 执行 update step，得到 $x_{M2|M2}$

  其对应的代码如下图所示：

```
# when getting the control signal, execution the predict step, update the control signal
def control_moment(self, u_new, time_now):                          # Control moment
    self._predict_Step(time_now)  # execute the predict step with current time
    self.u = u_new # update the control signal

# when getting the observe info, execution the predict step, and then execution the correction step
def observe_moment(self, y_new, time_now):                          # The observation time
    self._predict_Step(time_now)  # execute the predict step with current time
    self._correction_Step(y_new)  # execute the correction step with new observation
```

因此，卡尔曼滤波器这部分的最终框图为：



其中，Modelstate、Twistcallback 以及 plot 画图部分已经由老师和助教提供，我并没有修改。因此，在报告中我不加赘述。LaserScancallback 部分，我将在下一章节（实验结果和分析中详细阐述）。

2. PID 闭环算法及实现

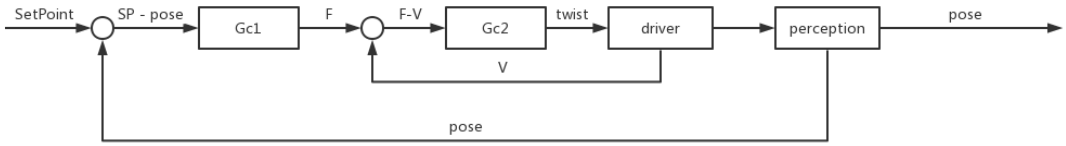
由于在具体的仿真实验当中，我采用的是以固定的采样时刻获取当前的位姿以及速度等信息，其是离散的。所以对于移动机器人，我采用的是位置式数字 PID 算法进行闭环控制。

位置式数字 PID 的计算公式如下：

$$u(k) = K_p e(k) + K_I \sum_{i=0} e(i) + K_D[e(k) - e(k-1)]$$

在实验过程中，我发现控制移动机器人运动的控制信号是 x 轴和 y 轴方向上的力，其对应的参数分别为 twist.linear.x 以及 twist.linear.y。当使用单回路 PID 控制时，我们知道力 F 与位置 x 的二阶导数线性相关，由于采样时间之间有一定的间隔，因此存在一定的滞后现象：即移动机器人无法提前减速导致有惯性会超过目标点。

为了解决这一问题，我采用了串级 PID 控制，内环为速度环，输出信号用于控制移动机器人的速度，外环为距离环，输出信号用于控制移动机器人移动至目标点。控制框图如下图所示：



此时，我在仿真平台上运行的效果较好，但是还需要调节 PID 参数。我调节 PID 参数的思路是：对外环控制（即 F）的 Kp 参数需要大一些，因为拥有足够大的力才能够比较迅速的对机器人的速度进行控制。PID 控制器及串级控制的最终代码如下图所示（只展示了一个 state 下的代码，剩余 state 的代码类似）：

PID 控制器模块代码（具体注释已写）：

```
class PID_Controller:
    #对PID参数进行初始化
    def __init__(self,kp,ki,kd,output_min,output_max):
        self.kp = kp
        self.ki = ki
        self.kd = kd
        self.error = 0
        self.last_error = 0
        self.error_sum = 0
        self.error_diff = 0
        self.output_min = output_min
        self.output_max = output_max
        self.output = 0
    #设置上下限
    def constrain(self, output):
        if output > self.output_max:
            output = self.output_max
        elif output < self.output_min:
            output = self.output_min
        else:
            output = output
        return output
    #得到PID的最终输出
    def get_output(self, error):
        self.error = error
        self.error_sum += self.error
        self.error_diff = self.error - self.last_error
        self.last_error = self.error

        output = self.kp * self.error + self.ki * self.error_sum + self.kd * self.error_diff
        self.output = self.constrain(output)
        return self.output
```

具体控制模块代码：

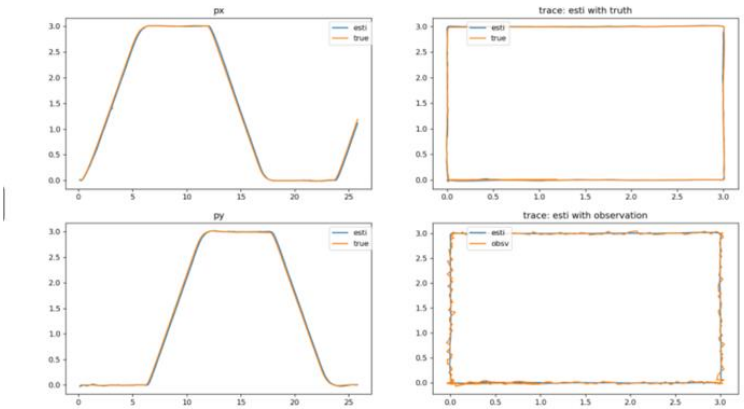
```
def velocity_publisher(self):
    #Set global variables
    global x, y, vx, vy,last_x, last_y
    global time_now,last_time
    rate = rospy.Rate(10)
    state = 0
    twist = Twist()
    #Set the error of target position and acceptable distance from the target
    L, H =3, 3
    move_er_threshold = 0.02
    rotate_er_threshold = 1e-3
    #Set the parameters of the PID controller (four in total, corresponding to
    #the x-direction and obtaining the controller move_controller_x for speed from the position,
    #controller move for force F obtained from velocity error_ Controller_ Vx and
    #two controllers of the same type corresponding to the y direction)
    move_controller_x = PID_Controller(1.3, 0, 0.4, -0.6, 0.6)
    move_controller_vx = PID_Controller(30,0, 0.01, -18, 18)
    move_controller_y = PID_Controller(1.3, 0, 0.4, -0.6, 0.6)
    move_controller_vy = PID_Controller(30,0, 0.01, -18, 18)
    rate.sleep()
    while not rospy.is_shutdown():
        if state == 0:
            print("x,y",x,y)
            #set the error of x and y
            error_x = L - x
            error_y = -y

        #When reaching the target position, all errors are cleared to zero and transfer State
        if abs(error_x) < move_er_threshold :
            state = state + 1
            twist.linear.x = 0
            twist.linear.y = 0
            twist.angular.z = 0
            move_controller_x.error = 0
            move_controller_x.last_error = 0
            move_controller_x.error_sum = 0

            move_controller_y.error = 0
            move_controller_y.last_error = 0
            move_controller_y.error_sum = 0

            move_controller_vx.error = 0
            move_controller_vx.last_error = 0
            move_controller_vx.error_sum = 0

            move_controller_vy.error = 0
            move_controller_vy.last_error = 0
            move_controller_vy.error_sum = 0

        else:
            #Calculate target speed from position error
            temp_x  = move_controller_x.get_output(error_x)
            temp_y = move_controller_y.get_output(error_y)
            #Calculate the speed error
            error_vx = temp_x-vx
            error_vy = temp_y - vy
            #Calculate target force from speed error
            twist.linear.x = move_controller_vx.get_output(error_vx)
            twist.linear.y = move_controller_vy.get_output(error_vy)
            twist.angular.z = 0
```

三、实验结果与分析

1.基础实验

首先，我先用 PID 加串级控制在原有的基础 gazebo 环境下对机器人进行简单的控制，我设计了一个简单的正方形回路来测试方案的可行性。其 gif 动图如下在实验结果文件夹栈示（由于一开始打开 gazebo 时没办法马上点截图，故在第一段没有录到，后续意识到方法后，之前代码以及被覆盖了）：

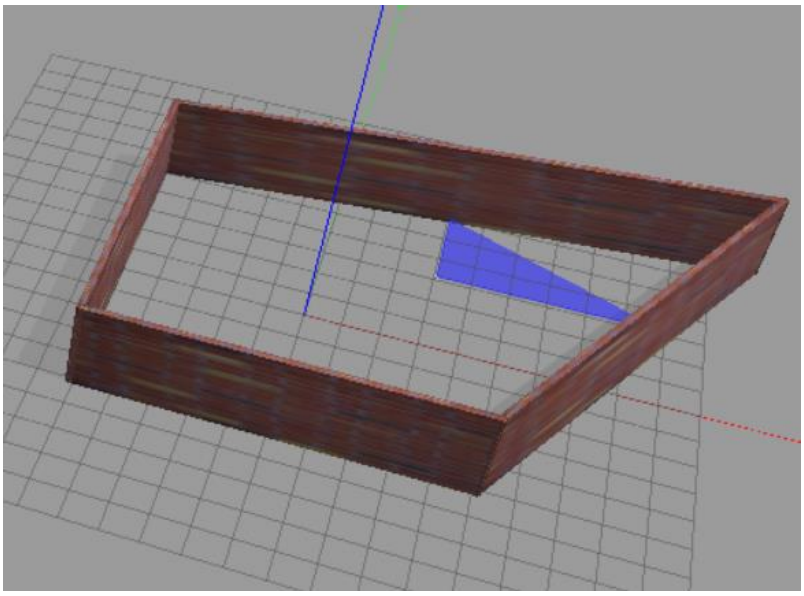
但是在输出的卡尔曼滤波器的滤波结果与读取 gazebo 获取到的真实值的对比图中可以看到具体的所有线路。如下图所示：



可以发现，其整体的效果还是非常不错的，因此可以采取此方法对后续的新 gazebo 环境下的机器人进行控制。

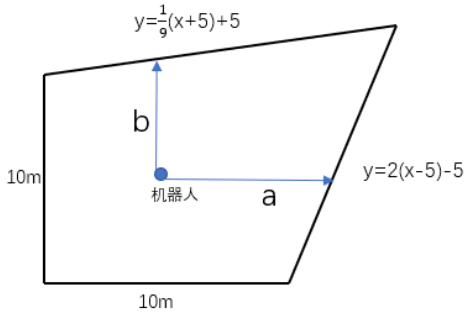
2.gazebo 重建及设计

首先，我在 gazebo 环境下对仿真环境进行了重建（主要将两面直的围墙变成卸的，增加了一定的定位难度），如下图所示：



由于围墙变成斜的，因此我需要改写 perception 中 Localization 类 callback_observe 函数，我们知道这个函数是用于测量机器人在地图中的实际位置的。它通过机器人所带的激光，可以得到机器人距离围墙的距离，而我们又可以知道围墙所处的位置，因此可以计算得到机器人的具体位置。

由于我对地图进行了重建，因此围墙的位置发生了改变，因此需要对该函数进行重写。现有的地图可以用下面的图来简单表示：



两个斜面围墙的解析式如上图所示。可知机器人右侧激光检测到机器人到围墙的距离为 a，上侧激光检测到机器人到围墙的距离为 b。因此，对于机器人的位置，我们可以得到以下方程：

$$\begin{cases} y = \frac{1}{9}(x+5) + 5 - b \\ y = 2 * (x - 5 + a) - 5 \end{cases}$$

联立这两个方程，我们可得：

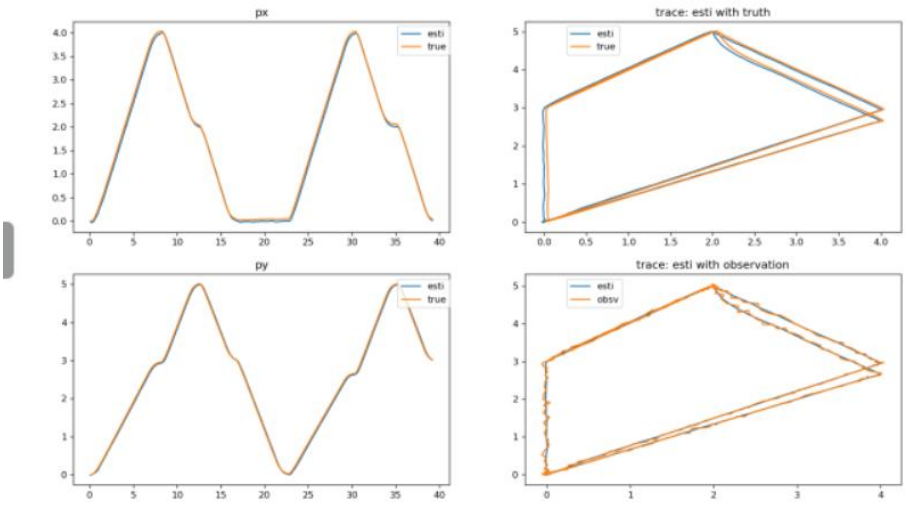
$$\begin{cases} x = \frac{185}{17} - \frac{9}{17}(2a + b) \\ y = 2 * (x - 5 + a) - 5 \end{cases}$$

由此得到对应的代码如下图所示：

```
def callback_observe(self, laserscan):
    # extract observe signal from message
    y = np.zeros([2,1])
    # y[0,0] = 5-laserscan.ranges[0]
    # y[1,0] = 5-laserscan.ranges[1]
    #Laser measurement of the distance between the robot and the wall in two directions
    x_init = laserscan.ranges[0]
    y_init = laserscan.ranges[1]
    #print("x: ", x_init,"y:", y_init)
    #Calculate the current position of the robot
    y[0,0] = 185/17.0-9.0/17*(2*x_init + y_init)
    y[1,0] = 2*( y[0,0] -5 + x_init) -5
    #print("x_true: ", y[0,0],"y_true:", y[1,0])

    current_time = rospy.get_time()
    # call observe moment function in Kalman filter
    self.kf.observe_moment(y,current_time)
    # save data for visualzation
    self.x_esti_save.append(self.kf.x)
    self.x_esti_time.append(current_time)
    self.p_obsv_save.append(y)
    self.p_obsv_time.append(current_time)
    # send estimated x to controller
    self.sendStateMsg()
```

最终给定一个具体的路线，得到的运行结果在实验结果文件夹中展示：

由 perception 函数中得到的卡尔曼滤波器的滤波结果与读取 gazebo 获取到的真实值的对比图如下：



可以看出，整体的效果还是非常好的预测值与实际值基本吻合。

此外，由于在走斜线时如（$\Delta y \neq \Delta x$）时，机器人走的路线通常不是最优路径（即先走一段斜率为 1 的线，在不断调整到指定位置）。因此我对控制函数又进行了优化调整，以下为优化的代码：

```
                                        #set the error of x and y
                                        error_x = 4- x
                                        error_y = 3-y

        #Calculate target force from speed error
        twist.linear.x = move_controller_vx.get_output(error_vx)
        twist.linear.y = move_controller_vy.get_output(error_vy)
        twist.linear.y = min(twist.linear.y, 3.0*twist.linear.x/4)
```

我们可以知道，当从（0，0）到（4，3）时，若要使机器人走最优路径，即要控制 y 方向的力为 x 方向上的 3/4，上述代码即实现了此优化。

四、小组分工与贡献度说明

单人完成 100%。其余代码由压缩包一起上传。