Team Nautilus: Wall Following With Explicit Instructions

This project can be organized into four parts: junction detection, feedback error computation, pid control, and check turn completion. Each part is explained with pseudo code and a short paragraph to explain the algorithm.

Part 1. Junction Detection

```
# load gap data from Lab 2
Class gaps = subscriber('LidarGaps')   # from Lab 2

# read csv instruction
list[] csvData(String dir, float vel) = csvRead('instructions.csv')

# check whether car can get into the gap
(bool flagC, bool flagR, bool flagL) = false
for n in gaps
    if gaps.angle() <= RightScanThreshold
        flagR = true
    if gaps.angle() > RightScanThreshold & gaps.angle() < LeftScanThreshold
        flagC = true
    if gaps.angle() >= LeftScanThreshold
        flagL = true

# find junction type (cross, T, etc)
if (flagR == true) & (flagC == true) & (flagL == true)
    juncType = 'cross'
elseif (flagR == true) & (flagL == true)
    juncType = 'T'
elseif (flagR == true) & (flagT == true)
    juncType = 'T'
elseif (flagL == true) & (flagT == true)
    juncType = 'T'
else
    juncType = 'nil'

# follow csv instruction
if juncType == 'cross' or 'T'
    vel = csvData[].vel
    dir = csvData[].dir

    setParameter('turn_flag', ''ACTIVE')  # update ros parameters
    setParameter('velocity', vel)
    setParameter('direction', dir)
```

In the junction detection part, the algorithm finds junctions using the gap finding algorithm, checks if the junction type is cross, T, or nil. The algorithm starts by loading the gap data. It contains the data of gaps, such as number of gaps, and angles. Once the data is loaded, it checks whether the entrance to the gap is within the scan threshold (i.e., to distinguish each of the three entrances). If it is in the range, that the gap is considered as a valid path, and the boolean value of each path turns true. Once the available paths are found, it categorizes the type of the

junction. Depending on what directions of paths are available, the intersection is categorized into cross, T, and nill. By using the junction type and the instruction.csv file, the car determines the velocity and the direction angle by changing the ros parameter internally, which is used by the other the pid_error and control node to make changes accordingly.

Part 2. Computation of Error for Feedback

```
# load lidar data
lidarData = subscriber('laserScan')

# compute error for following left, right, and center
errorLeft   = DistToLeftWall(laserScan)  - desireDistToLeftWall
errorRight  = DistToRightWall(laserScan) - desireDistToRightWall
errorCenter = DistToLeftWall(laserScan)  - DistToRightWall(laserScan)

# change error mode, depending on datafile
String direction = getParameter('direction')    # get from ros parameter server
If direction = 'left'
    error = errorLeft
If direction = 'right'
    error = errorRight
If direction = 'center'
    error = errorCenter

# update and publish data
publisher(error)
```

The algorithm computes feedback error, which is the difference between the measured distance and desired distance to both walls on the sides. It uses lidar laserscan data and trigonometry method to measure the distance between the car and right-side wall and the distance between the car and the left-side wall. Once the measured distance to both walls are computed, errors for three cases are computed: errors for following the left-side wall, errors for following the right-side wall, and the errors for following the centerline of the path. While it is following the wall, the ros parameter *direction's* value can change in the middle of execution, causing the change in which side wall to follow. These error values are utilized by the control node to calculate the required steering angle and velocity.

Part 3. PID Control

```
# load error data for PID control
error = Subscriber('pid_error')

# compute PID output (tuned by test)
KP = 0.001
KD = 0.001
KI = 0.001

pidOutput = KP*error * KD*(error-prevError) * KI*cumulSum(error)
```

```
# compute steering angle
steerAngle = pidOutput
steerAngle = clip(-0.4189, 0.4189)    # limit steering angle to 24 degree

Vel = getParameter('velocity') # from ros param server

# control the velocity
If abs(steeringAngle) < 10   # in degree
    Vel = vel
elseif abs(steeringAngle) < 20   # in degree
    Vel = 1.0
else
    Vel = 0.5

# update and publish parameters
drive_parameter.vel   = vel
drive_parameter.angle = angle
publisher(drive_parameters)
```

This algorithm computes the PID control. The algorithm starts by setting up the PID controller's feedback constants KP, KD, and KI. Those three constants can be determined by running multiple simulations and picking the one with the best result. Once the pid constants are determined, the pid_error value can be computed. This pid_error value is used to determine the value of steering angle: the steering angle depends on the pid_errors. Once the steering angle is determined, the velocity of the car is set up, so that the car does not move too fast when the car is having a high steering angle. This steering angle and velocity is the result of the PID control. The value of angle and velocity is published to the system.

Part 4. Checking Turn Completion

```
# load coordinate transform data
 Listener = tf.TransformListener()

# compute the yaw angle of the tf from /map → /base_link frame
Translation, quaternion = lookupTransform(/base_link, /map)
(float roll,float pitch,float yaw) = quaternionToEulerAngle(float[] quaternion)
float angleForward = rad2deg(yaw)

# check whether a turn is completed
string turnMsg = 'no turn complete'
float angleStartTurn = 0.0
if angleForward - angleStartTurn >= 50
    Turn = 'left turn complete'
    angleStartTurn = angleForward
elseif angleForward - angleStartTurn <= -50
    Turn = 'right turn complete'
    angleStartTurn = angleForward

# update the turn_flag parameter once the turn is completed
Turn_flag = setParameter('INACTIVE')
```

Once the car detects a junction (T/cross), the *turn_flag* parameter will be set to 'ACTIVE', until the turn is completed. The turn completion code continuously monitors the change in yaw angle of the car (w.r.t /map reference frame) between before initiating the turn, and during executing the turn. Once the turn angle crosses a particular threshold (set as 50 deg), we assume to turn to be complete, and change the *turn_flag* parameter back to 'INACTIVE'.

Parameter tuning

All the parameters listed here are tuned for the levine track, since it was the only track that was not so complex, and at the same time had multiple junctions and intersections to test our wall-following algorithm.

The look-ahead distance L was tuned by testing a range of possible values, starting with 0.1. The look-ahead distance was then increased by 0.1 for each trial. The error and overall performance of the car was observed to determine the proper lookahead value given different situations. The car was tested in straight-aways, corners, and t-junctions to confirm which lookahead distance value performed best over each scenario. For overly low or overly high values of L, the car would not be looking directly enough at relevant areas of the wall, so it would crash or go out of bounds. The lookahead value is also important for determining the next gaps within the wall, so there had to be enough distance so the car was prepared. After tuning, the desired L look-ahead value was determined to be **1.5m.**

To tune the PID gains Kp, Ki and Kd, the strategy was to test each value at 0 and slowly increase one at a time to see the changes in the car oscillation and overall performance. At first, all gains were set to 0 so there was no influence. Then, we incremented the weight value of Kp over multiple trials to see how the car oscillated around the set distance. Once we achieved better performance from increasing Kp, the proportion weight, we then started tuning Ki, the integral weight, and Kd, the derivative weight. After testing Ki, we found that the value that gave the least interference in behavior was 0. We slowly increased the value for Kd over trials to see what values made the car closer to converging closely around the desired distance. With each of these trials, we tested with the car following the left wall, right wall, and center to ensure good performance over all situations. The values for Kp and Kd were determined to be **0.5 and 0.01,** respectively.

The target distance from the wall was also tuned in a similar manner, by starting at 0 and slowly being incremented. The target distance from the wall can determine how closely the car navigates around different environments, so it was important to get right. The distance was tested

for the left wall, right wall, and center wall following methods to ensure clear maneuverability. The target distance was tuned to be **1.3m.**

General parameter tuning procedure:

1. Start with all parameters to be tested set to 0 (or as low as possible).
2. Slowly increase value (usually by 0.1) and observe effects on car movement.
3. Once the car is performing smoothly (reaches set point), or the expected stable behavior occurs, select a different parameter to tune.
4. Slowly increase the value of the new parameter, like done in previous steps, until expected performance is achieved.
5. Based on your understanding of the parameters being tuned, minorly adjust certain parameters if you see it encouraging better performance.
6. Repeat the process with all parameters within different environments and situations to confirm the proper behavior.