Reading Instructions

Please carefully study the papers on the pure pursuit path tracking algorithm [2] and exponential stabilization of a wheeled mobile robot via discontinuous control [3] prior to the lab. A general overview on motion control approaches for wheeled mobile robots is provided in [1].

Technical Prerequisites:

This lab is concerned with homing and trajectory tracking and does not consider obstacles, therefore all assignments simulate the robot with the empty world map.

```
roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
```

As an empty map lacks any environmental objects that a range sensor may perceive a proper localization with the *Adaptive Monte Carlo Localization* (AMCL) is infeasible. Therefore, the pose estimation of the `/odom` relies on odometry only. In a second terminal, start RViz using

```
roslaunch turtlebot3_gazebo turtlebot3_gazebo_rviz.launch
```

The goal pose is specified by the user in RViz with the `2D Nav Goal` button and is published on the topic `/move_base_simple/goal`. Notice that the goal pose in `/move_base_simple/goal` is published w.r.t. the `odom` rather than the `map` frame. Thus the representation of goal pose is consistent with the robots pose from odometry.

Motion Control

Motion control of a mobile robot is concerned with tracking a feasible reference state trajectory. The mobile robot regulates its planar pose via the control of its motors in terms of velocity commands. The approaches rest upon feedback control of the actual robots pose either w.r.t. a global fixed frame or w.r.t. the reference trajectory. A non-holonomic robot possesses fewer local degrees of freedom than global degrees of freedom. The prototypical mobile robot is the unicycle (see figure 1) with controls v, ω and global pose x_r, y_r, θ_r governed by the kinematics

$$\begin{aligned} \dot{x}_r &= v \cos\theta_r \\ \dot{y}_r &= v \sin\theta_r \\ \dot{\theta} &= \omega \end{aligned} \tag{1}$$

A differential drive robot is composed of two independent actuated wheels on a common axle whose direction is rigidly connected to the robot body. The purpose of the passive, orientable caster wheel is to support the robot, it plays no role in motion control. The kinematic equations of the unicycle apply to the Turtlebot differential drive robot as well if one considers the robots frame origin to be located on the center of the virtual axle

connecting the two driven wheels. The differential drive robot kinematics are compliant with the unicycle model in Eq. (2) if left and right wheel velocities $[v_l,\ v_r]$ are mapped onto the vector of linear and angular velocity $[v, \omega]$.

$$v = \frac{v_r + v_l}{2} \tag{2}$$

$$\omega = \frac{v_r - v_l}{b} \tag{3}$$

in which b denotes the wheel base of the differential drive robot.

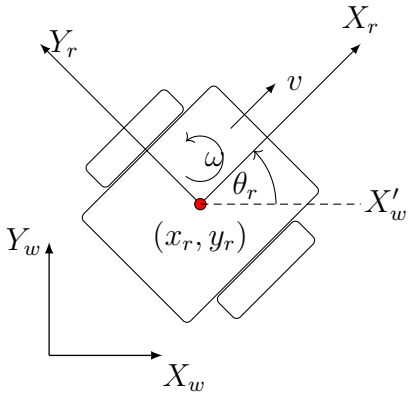

Figure 1: Kinematics of a unicycle robot with global frame X_w, Y_w and robo-centric frame X_r, Y_r.[2]

Feedback Robot Control with Publisher and Subscriber

The figures illustrates the subscriber and publisher architecture for interaction between the robot in the ROS Gazebo simulation and Matlab.

The `/odom` topic provides the robot pose (x_r, y_r, θ_r) w.r.t. the world frame X_w, Y_w . Notice that the `nav_msgs/Odometry` describes the orientation of the robot frame in terms of a quaternion $\mathbf{q} = [\eta\ \epsilon_x\ \epsilon_y\ \epsilon_z]$.

As the mobile robot resides in a planar world, all rotations of the robot frame are along the z-axis only. The yaw angle θ_r that denotes the rotation between the world x-axis X_w and the robots x-axis X_r is extracted from the quaternion with `quat2eul`. The helper function `OdometryMsg2Pose` provided in the Moodle workspace maps the message to a pose `[x, y, theta]`.

The topic `/cmd_vel` publishes velocity commands to the robot in the Gazebo simulator from Matlab via messages of type `geometry_msgs/Twist` which describes translational and angular velocity.

[2]source: RST

Your code calculates the general pose error in robocentric polar coordinates $[\rho, \alpha, \phi]$ (see figure 4) between the robots pose (x_r, y_r) and goal pose (x_g, y_g) and maps the error onto a motion command v, ω. ρ denotes the Euclidean distance between the current robot position and the goal position, α denotes the orientation error between the robot current heading in direction X_r and the heading towards the goal pose (x_g, y_g). The angle ϕ denotes the error between current robot heading and the desired orientation at the goal pose. The angle ϕ becomes relevant later in the context of homing with goal poses (x_g, y_g, θ_g) rather than merely goal points (x_g, y_g).

There are three options to specify the current goal pose (navigation goal):

- with the 2D navigation goal button in RVIZ, that pose is published by the topic `/move_base_simple/goal` in terms of message of type `geometry_msgs/PoseStamped`.
  Your Matlab code monitors the navigation with a corresponding subscriber.

- with a sequence of reference poses (waypoints) specified in terms of $[{x_g}^i, y_g^i, \phi_g^i]$. The immediate i-th waypoint determines the current goal pose and upon reaching the waypoint the goal pose transits to the i+1-th waypoint in the sequence. Your code provides its own publisher that publishes the imminent waypoint as the current navigation goal.

- with a sequence of reference poses (waypoints) in which the current navigation goal is determined by a look-ahead-point located on the straight line connecting the i-th and i+1-the waypoint some distance ahead of the current robot pose. This look-ahead-point calculation is used in conjunction with a so called pure-pursuit path tracking controller. Your code provides its own publisher that publishes this look-ahead-point as the current navigation goal. As the look-ahead-point depends on the current robot pose, it is best to realize the look-ahead-point calculation and publishing with a callback subscriber on the `odom` topic.

Your code employs a subscriber `navGoalSub` for the current navigation goal which subscribes to either of the three topics by the three types of publishers. That way you encapsulate the particular navigation goal method from the underlying homing or tracking controller.

The pose error calculation and feedback control are either integrated into the while loop of the Matlab main program ...

- ... explicitly receiving a `navGoalMsg` by the subscriber with `LatestMessage`, from which one calculates the pose using the ROS transformation tree and sends the velocity command within the while loop:

```
tftree = rostf;
rateObj=robotics.Rate(rate);

rho = inf;
goalRadius = 0.2;

...

disp('Specify 2D Nav Goal...');
navGoalMsg = receive(navGoalSub, 30);

rateObj.reset;  % reset time at the beginning of inner loop
while (rho > goalRadius)
   % calculate pose error from odom and navgoal
   [rho, alpha] = ...


   % calculate velocity command from pose error
   [v, omega] = ...

   velMsg = rosmessage(velPub);
   velMsg.Linear.X = v;
   velMsg.Angular.Z = omega;
   send(velPub,velMsg);

   ...

   waitfor(rateObj);
end
disp('Goal reached!');
velMsg.Linear.X = 0;
velMsg.Angular.Z = 0;
send(velPub,velMsg);
```

The `robotics.Rate` object achieves the execution of the loop at a fixed cycle rate. The command `waitfor(rateObj)` synchronizes the loop execution time with the ROS time.

- ... or the feedback control is handled by a callback subscriber for `odom` in which the callback function

```
function [] = odomHomingCallback( ~, ~, navGoalSub, velPub, poseErrorObject,
    tfTree)
```

determines the pose error, computes the controls and publishes the corresponding motion command. The pose error is communicated to the main loop via the handle object `poseErrorObject` of class `poseErrorHandle` with properties `poseErrorObject.rho` and `poseErrorObject.alpha` . After reaching the goal the callback subscriber is cleared in order to stop publishing further motion commands.

```
tftree = rostf;
rateObj = robotics.Rate(rate);
```

```
...

poseErrorObject = PoseErrorHandle;
poseErrorObject.rho = inf;
goalRadius = 0.2;

disp('Specify 2D Nav Goal...');
navGoalMsg = receive(navGoalSub, 30);

odomSubCallback = rossubscriber('/odom',{@odomHomingCallback,navGoalSub,...
  velPub, poseErrorObject, tftree});

rateObj.reset;  % reset time at the beginning of inner loop
while (poseErrorObject.rho > goalRadius)
  ...
  waitfor(rateObj);
end
disp('Goal reached!');
velMsg.Linear.X = 0;
velMsg.Angular.Z = 0;
send(velPub,velMsg);
clear odomSubCallback;
```

Homing

Mobile robot homing is a simpler case of the general task of trajectory tracking as the goal pose is static. One distinguishes between mere position based homing, in which the robot navigates towards a point x_g, y_g irrespective of the heading at which it approaches the target (figure 3 left). In homing with orientation the robot is expected to approach the target from a particular direction, for example to enter a docking station for recharging. The robot is expected to acquire pose defined in terms of x_g, y_g, θ_g (figure 3 right).

Homing with Goal Point

Homing considers the current position and heading of the robot relative to a desired position and calculates a motion command based on their difference. The turn rate ω regulates the heading error towards the goal point α to zero, the forward velocity is proportional to the Euclidean distance ρ to the goal.

The objective of the first part of the lab is to implement a simple homing control law that guides the robot to a fixed goal position x_g, y_g. Either implement the solution with a while loop that explicitly receive messages or more elegantly with a callback subscriber that automatically responds to messages. The helper functions `OdometryMsg2Pose`, `PoseStampedMsg2Pose` and `Pose2PoseStampedMsg` map the message to a pose `[x, y, theta]` and vice versa.

[4]source: RST
[6]source: RST
[8]source: RST

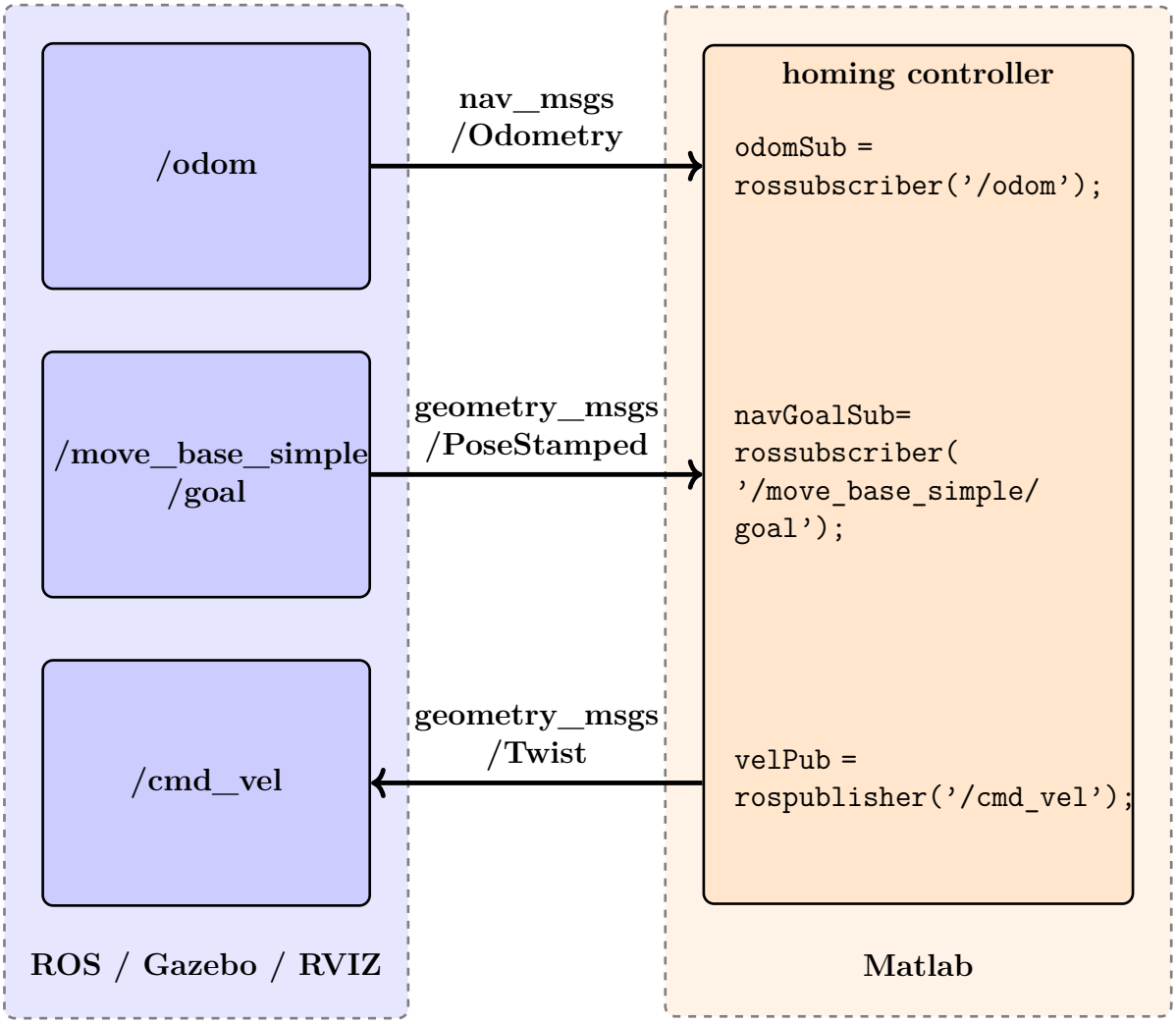

Figure 2: Publisher and subscriber communication between ROS / Gazebo and Matlab [4]

For the following tasks, you have to instantiate a variable with the ROS transformation tree in the beginning of your main script:

```
2 tftree = rostf;
```

1) **subscriber and publisher**

Define a subscriber `odomSub` that subscribes to the `/odom` topic to receive the robot pose. Define a subscriber `navGoalSub` that subscribes to the `/move_base_simple/goal` topic in order to receive the user selected navigation goal from RViz. Define a publisher `velPub` for publishing velocity commands on the topic `/cmd_vel`.

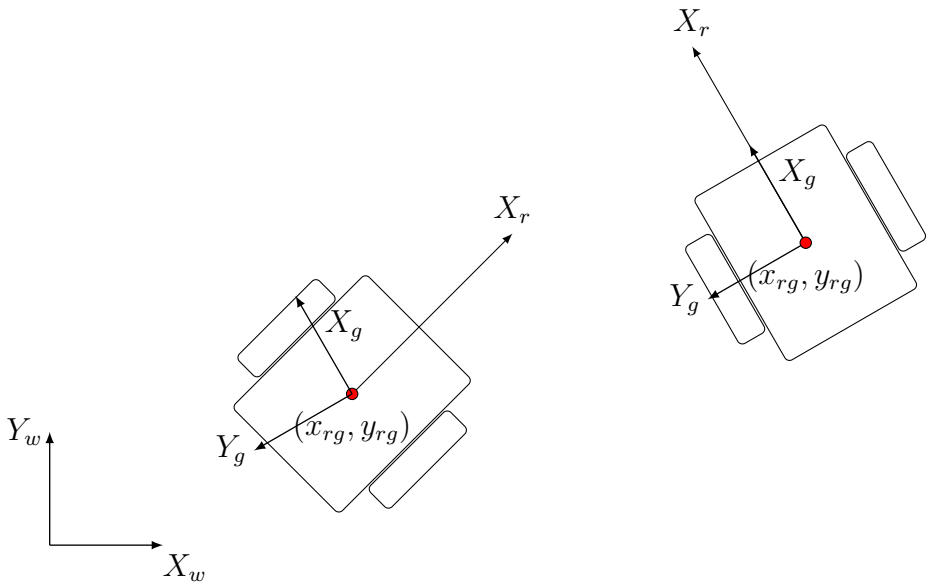

Figure 3: Homing with position x_g, y_g only (left) and homing with full pose x_g, y_g, θ_g (right) [6]

2) **pose error calculation with transforms**

Implement a function

```
function [rho, alpha] = poseErrorTf( navGoalMsg, tftree )
```

that calculates the pose error in polar coordinates ρ, α from the pose messages `navGoalMsg`. Determine the goal pose in robocentric coordinates by transforming the entity `navGoalMsg` specified w.r.t. `odom` frame to the `'base_link'` with the function `transform`. Obtain the resulting goal pose $x_g^{(r)}, y_g^{(r)}, \theta_g^{(r)}$ using the helper function `PoseStampedMsg2Pose`. The goal position error becomes

$$\rho = \sqrt{{x_g^{(r)}}^2 + {y_g^{(r)}}^2} \tag{4}$$

$$\alpha = \text{atan2}(y_g^{(r)}, x_g^{(r)}) \tag{5}$$

This transformation can be realized using the Matlab built-in function `[alpha, rho] = cart2pol(x, y)`.

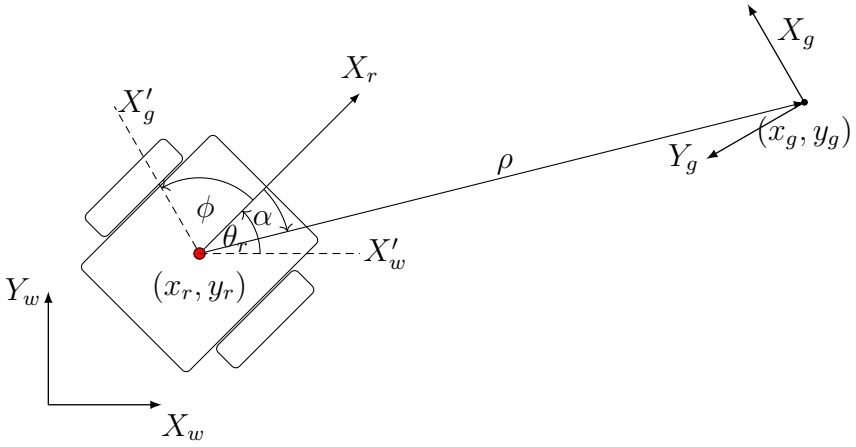

Figure 4: Global frame X_w, Y_w, robo-centric frame X_r, Y_r and goal frame X_g, Y_g for homing for $\alpha(0) \in I_1 = (-\pi/2, \pi/2]$[8]

3) **homing controller**

Implement a function

```
function [ v, omega ] = homingControl( rho, alpha )
```

that realizes a simple proportional controller to regulate the heading error α and the translational error ρ to zero:

$$v = k_\rho \rho \tag{6}$$
$$\omega = k_\alpha \alpha \tag{7}$$

Include a control signal saturation that limits the commanded forward velocity to v_{max} and turn rate to $\omega \in [\omega_{min}, \omega_{max}]$. Reasonable values for k_ρ and k_α are 0.5 and 1 respectively.

4) **component integration**

Integrate the two functions into the while loop of the main program code. Retrieve the `navGoalMsg` from the corresponding (callback) subscriber. Determine the pose error from the messages with `poseErrorTf` and map the pose error onto the motion command with `homingControl`.

Publish the commanded turn rate and forward velocity on the `/cmd_vel` topic and message type `geometry_msg/Twist`.

5) **testing and analysis**

Test your code by commanding the robot to navigation goals in RViz. Record and plot the evolution of the pose error $[\rho, \alpha]$ over time as shown in figure 5. Record and plot the robot poses $[x, y, \theta]$ using

```
quiver(posesX, posesY, cos(posesTheta), sin(posesTheta));
```

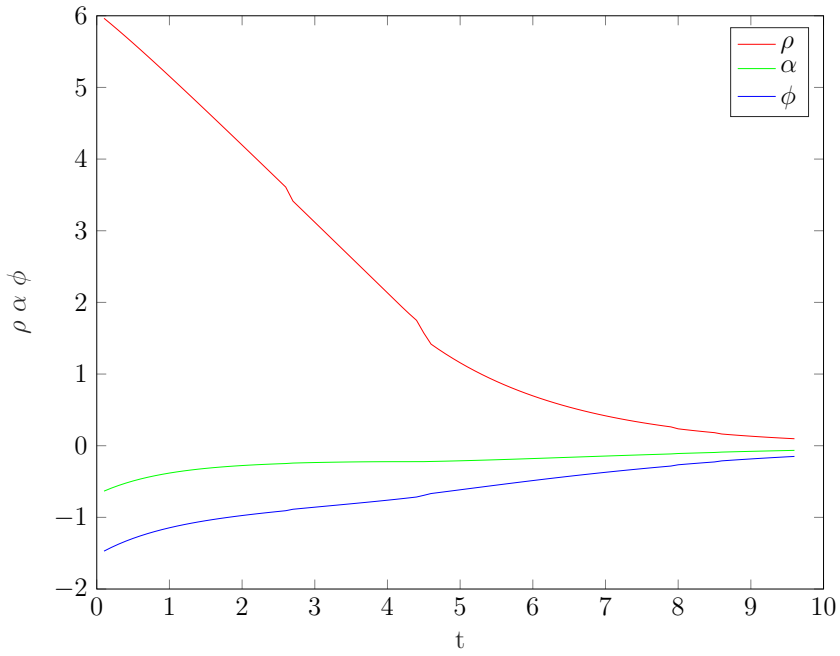

Figure 5: Robot Pose Error

6) **callback subscriber [optional]**

Implement the same homing controller with a callback subscriber on the `/odom` topic.

```
odomSubCallback = rossubscriber('/odom',{@odomHomingCallback,navGoalSub,velPub,
    poseErrorObject, tftree});
```

The callback function

```
function [] = odomHomingCallback( ~, ~, navGoalSub, velPub, poseErrorObject,
    tftree)
```

obtains references to the publisher of the navigation goal and velocity command. The handle object `poseErrorObject` of class `PoseErrorHandle` (m-file provided in Moodle) has the properties `rho`, `alpha`, `phi`, `deltay`.

```
classdef PoseErrorHandle < handle
  properties
    rho = 0
    alpha = 0
    phi = 0
    deltay = 0
  end
  methods
  end
end
```

The properties `poseErrorObject.rho` and `poseErrorObject.alpha` are assigned within the callback function. The two remaining properties `phi`, `deltay` will become relevant later in the context of homing with goal orientation and pure pursuit path tracking.

The handle object `poseErrorObject` enables it to share the pose error data between the callback function and the main program. Reception of pose messages, calculation of pose error and velocity commands and publishing of velocity commands is delegated to the callback function. The main program merely monitors the translational error `poseErrorObject.rho` to determine whether the goal point has been reached upon which it clears the callback subscriber.

Trajectory Tracking and Homing with Goal Pose

The reference trajectory $x_g(t), y_g(t)$ is a geometric path augmented with a timing law. The locus x, y is parameterized with the time variable t. The mapping $t \to (x_g(t), y_g(t))$ is defined w.r.t. a global fixed reference frame. Motion control is supposed to regulate the position error $\mathbf{e_p} = [x_g(t) - x_r(t), y_g(t) - y_r(t)]$ to zero.

Perfect tracking is only achieved if the reference trajectory is feasible for the robot and compliant with the mobile robot kinematics. The turning radius of a car like robot is bounded from below, which imposes an upper limit on the reference path curvature. Kinodynamic constraints impose bounds on the forward velocity v_{max} and turn rate $\omega \in [\omega_{min}, \omega_{max}]$. In general case the robots orientation $\theta(t)$ is supposed to follow a reference orientation $\theta_g(t)$ as well. For the unicycle robot the reference orientation is tangential to the path in case of a feasible reference trajectory. A general reference trajectory $(x_g(t), y_g(t), \theta_g(t))$ is feasible if it is generated by a reference vehicle with the same kinematics.

A control law for exponential stabilization of a robot along a reference trajectory is detailed in [3]. Figure 4 shows the geometry of the robot frame and the goal frame. Consider the coordinate transformation from equation (5). Here, α denotes the angle between the x-axis of the robot frame and the heading vector that points from the robot center to the goal position. Additionally, $\phi = \theta_g^{(r)}$ denotes the angle between the x-axis X_R of the robocentric frame and the x-axis X_G associated with the goal pose. v and ω denote the linear and angular velocity, respectively. With the new coordinates $\mathbf{e}(t) = [\rho(t), \alpha(t), \phi(t)]$ the kinematics in equation (2) are described by

$$\begin{aligned} \dot{\rho} &= -v\cos\alpha \\ \dot{\alpha} &= \frac{v\sin\alpha}{\rho} - \omega \\ \dot{\phi} &= -\omega \end{aligned} \tag{8}$$

The transformation applies for $\alpha \in I_1 = (-\pi/2, \pi/2]$. In case $\alpha \in I_2 = (-\pi, -\pi/2] \cup (\pi/2, \pi]$ (see figure 6) redefining the forward direction of the robot, by setting $v = -v$,

and applying a similar transformation yields

$$\begin{aligned} \dot{\rho} &= v\cos\alpha \\ \dot{\alpha} &= \frac{-v\sin\alpha}{\rho} - \omega \\ \dot{\phi} &= -\omega \end{aligned} \tag{9}$$

with again $\alpha(0) \in I_1$. If $\alpha \in I_1$, the forward direction of the robot points toward the goal, whereas if $\alpha \in I_2$, the backward direction of the robot points toward the goal. In the first case the robot moves with a positive linear velocity v, whereas in the second case the translational motion is reversed.

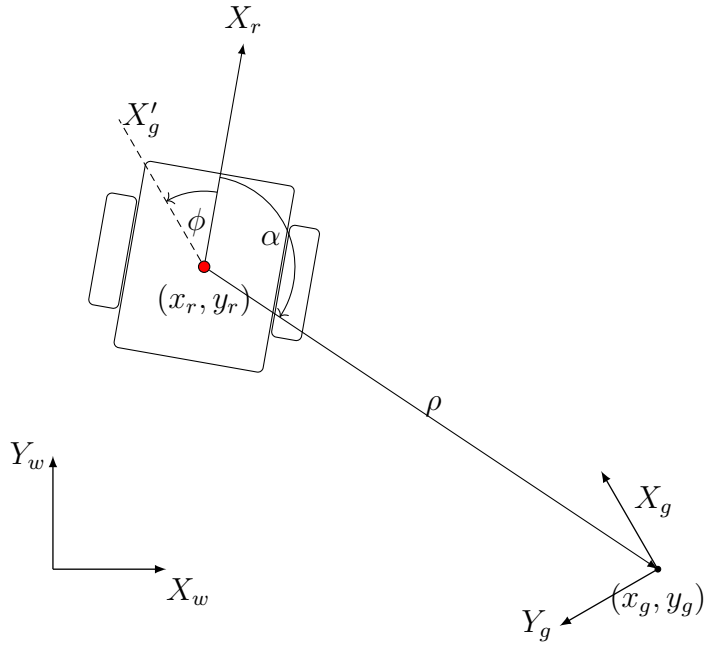

Figure 6: Global frame, robo-centric frame and goal frame for homing with exponential stabilization for $\alpha(0) \in I_2 = (-\pi, -\pi/2] \cup (\pi/2, \pi]$ [10]

The linear control law

$$\begin{aligned} v &= k_\rho \rho \\ \omega &= k_\alpha \alpha + k_\phi \phi \end{aligned} \tag{10}$$

achieves an exponential stabilization of the system at the equilibrium $(\rho, \alpha, \phi) = (0, 0, 0)$

[10] source: RST

under the following conditions on the gains

$$\begin{aligned} k_\rho &> 0 \\ k_\phi &< 0 \\ k_\alpha + k_\phi - k_\rho &> 0 \end{aligned} \tag{11}$$

A further condition applies to the initial pose. Assume that $\alpha(0) \in I_1$ and $\phi(t) \in (-n\pi, n\pi]$ for all t. Then if

$$k_\alpha + 2nk_\phi - \frac{2}{\pi}k_\rho > 0 \tag{12}$$

one has $\alpha(t) \in I_1$ for all t.

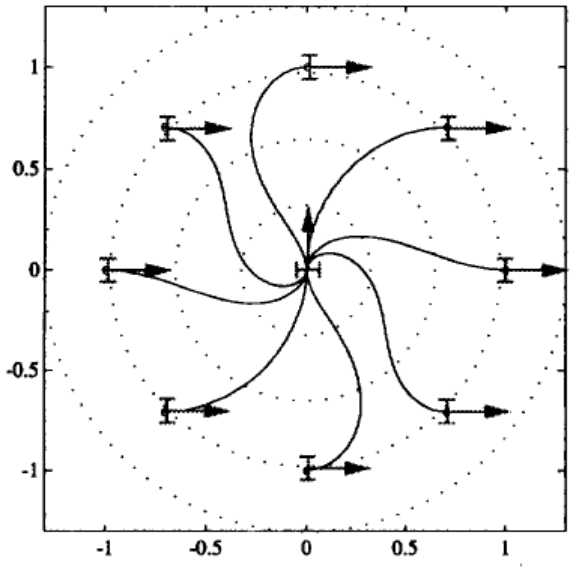

Figure 7: Paths towards the origin when starting on the unit circle. [11]

[11] source: Astolfi, A., Exponential Stabilization of a Wheeled Mobile Robot Via Discontinuous Control [3]

7) **pose error with heading:**

Augment the function

```
function [rho, alpha, phi] = poseErrorTf( navGoalMsg, tftree )
```

such that it additionally outputs the angle ϕ between the x-axis X_R of the robocentric frame and the x-axis X_G associated with the goal pose.

8) **homing controller:**

Implement the control law for exponential stabilization Eq. (10) w.r.t. the new coordinates ρ, ϕ, α. For that purpose augment the original controller function by an additional input parameter `phi`

```
function [ v, omega ] = homingControl( rho, alpha, phi )
```

Make sure to determine the correct sign of the linear velocity v according to whether $\alpha \in I_1$ or $\alpha \in I_2$. For $\alpha \in I_2$ substitute v with $-v$ and remap α to I_1

9) **controller gain selection and homing:**
Select proper settings for the controller gains k_ρ, k_ϕ, k_α according to (11). Change the while loop condition to

```
  while (rho > goalRadius || abs(phi) > goalOrientation)
```

where `goalOrientation = deg2rad(5)`. Test the controller for reference poses commanded with the RViz navigation button.

Record and plot the evolution of the pose error $[\rho, \alpha, \phi]$ over time as shown in figure 5. Record and plot the robot poses $[x, y, \theta]$ as shown in 8.

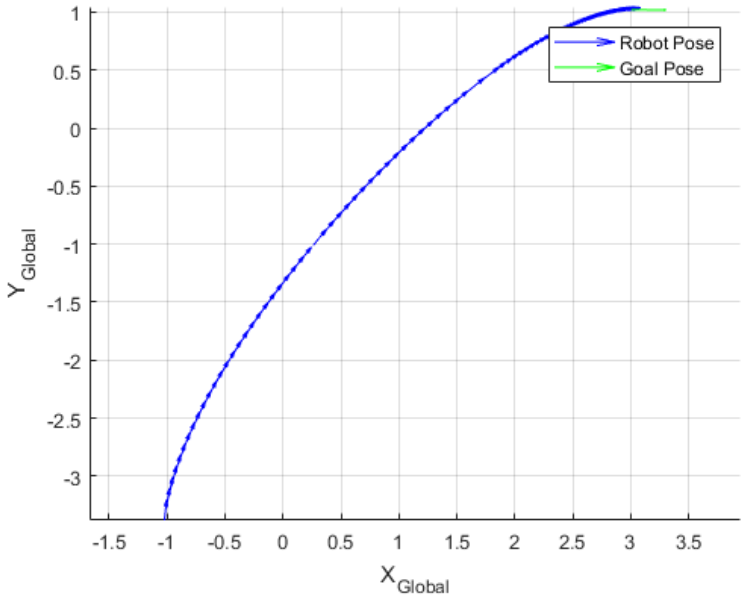

Figure 8: Robot Path

10) **navigation goal publisher:**

Replace the RVIZ navigation goal publisher by your own publisher. Figure 9 illustrates the subscriber and publisher structure. The `'/waypoint'` node is added to the ROS topic list and operates in Matlab.

```
wayPointPub = rospublisher('/waypoint','geometry_msgs/PoseStamped');
```

The navigation goal subscriber `navGoalSub` subscribes to the topic `'/waypoint'` rather than the RVIZ navigation goal.

```
navGoalSub = rossubscriber('/waypoint');
```

In that way the communication structure encapsulates the generation and maintenance of navigation goals from the navigation controller.

Define a n-by-3 array `waypoints` as a **global** variable that will contain a sequence of reference poses to be traversed in the order 1 to n.

Instantiate a callback subscriber that monitors the robot pose

```
odomWayPointSubCallback = rossubscriber('/odom',{@odomWayPointCallback,
    wayPointPub});
```

The callback function

```
function [] = odomWayPointCallback( ~, odomMsg, wayPointPub)
```

is provided as a helper function. It merely publishes the first waypoint in the list as `'geometry_msgs/PoseStamped'` message on the topic `'/waypoint'`. It removes the current waypoint if the distance between the current robot pose and the waypoint

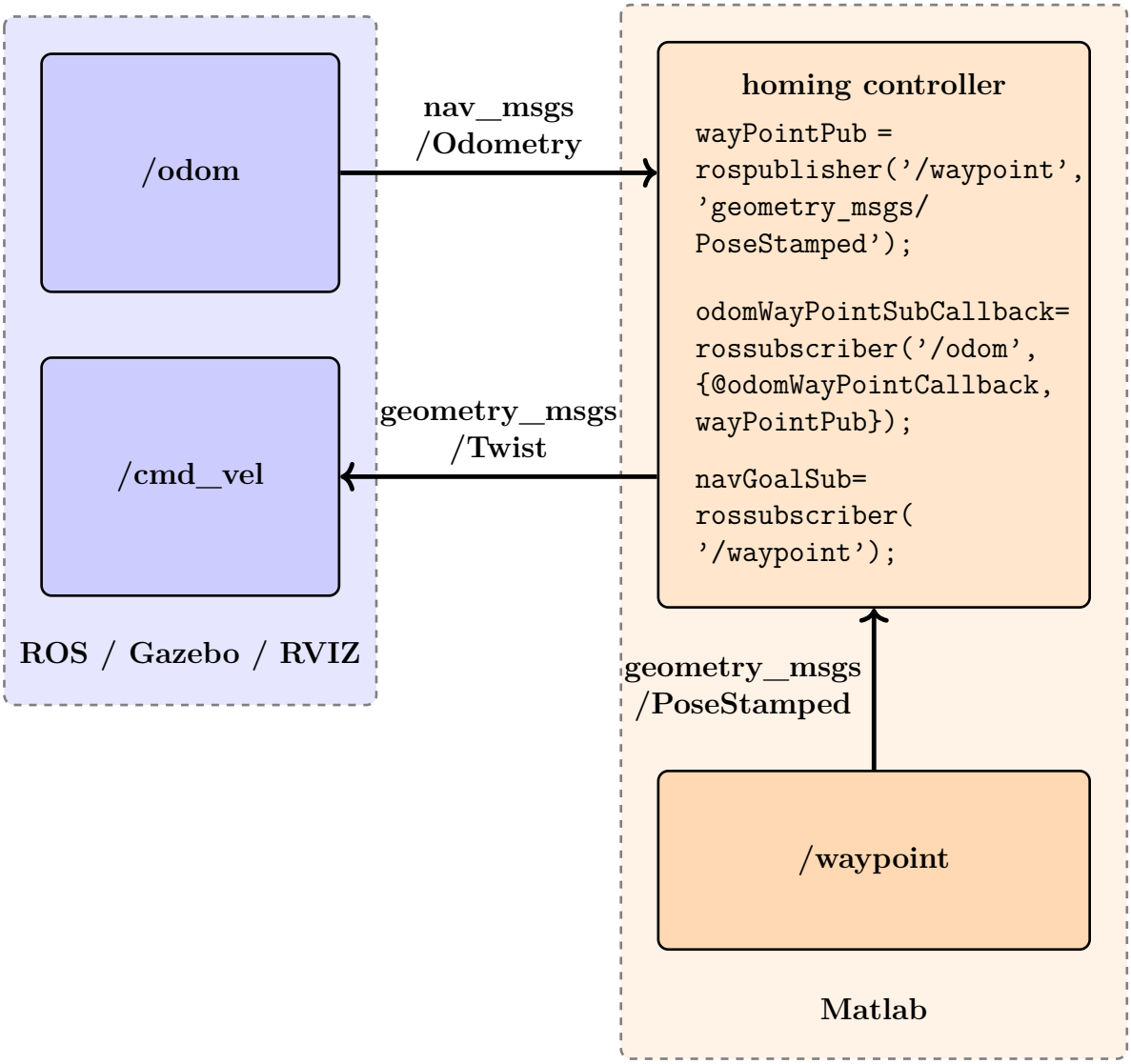

Figure 9: Publisher and subscriber structure for waypoint navigation [12]

is less than the goal radius. It stops publishing navigation goals once the list of waypoints is empty.

11) **modifications to while loop:**

The while loop termination condition in the main program changes. The while loop terminates as the list of waypoints is empty.

```
rateObj=robotics.Rate(rate);
...
rateObj.reset;  % reset time at the beginning of inner loop
while ~isempty(waypoints)
   ...
     waitfor(rateObj);
end
```

12) **testing and analysis:**

Test your controller in reaching the central goal pose $[x_r, y_r, \theta_r] = [0, 0, \pi/2]$ starting from a set of initial poses
$[x, y, \theta] = \{[1, 0, 0], [1/\sqrt{2}, 1/\sqrt{2}, 0], [0, 1, 0], \ldots, [1/\sqrt{2}, -1/\sqrt{2}, 0]\}$ located on the unit circle as shown in figure 7. For that purpose define a sequence of waypoints that alternates between the start poses on the unit circle and the central goal pose.

Pure Pursuit Tracking Algorithm

The pure pursuit algorithm has the objective to guide a robot along a reference path. Pure pursuit is a path tracking scheme that determines the curvature of the robot path that guides the robot from its current pose to the goal pose. For that purpose the scheme considers a dynamic goal position on the path located some distance ahead of the robots current position [2]. The robot is supposed to chase the moving goal point (look ahead point) on the path. This strategy is similar to human drivers that steer a vehicle towards a dynamic lookahead point on the road, which distance depends on the vehicle speed, road curvature and visibility.

The figure 10 illustrates the geometry of the algorithm. In compliance with ROS transforms the x-axis of coordinate frame coincides with robots current heading and the y-axis with the axle connecting the two wheels. The vector $(x_r - x, y_r - y)$ denotes the location of the lookahead (goal) point in the robocentric frame, l denotes the lookahead distance. In the following we consider all vectors to be expressed w.r.t. robot base frame. The algorithm calculates the radius of the arc that connects the origin of the robocentric frame with the goal point.

The following relationships hold:

$$\rho^2 = (x_g - x_r)^2 + (y_g - y_r)^2 = {x_g^{(r)}}^2 + {y_g^{(r)}}^2 \tag{13}$$

$$r = y_g - y_r + d = y_g^{(r)} + d \tag{14}$$

[13] source: RST

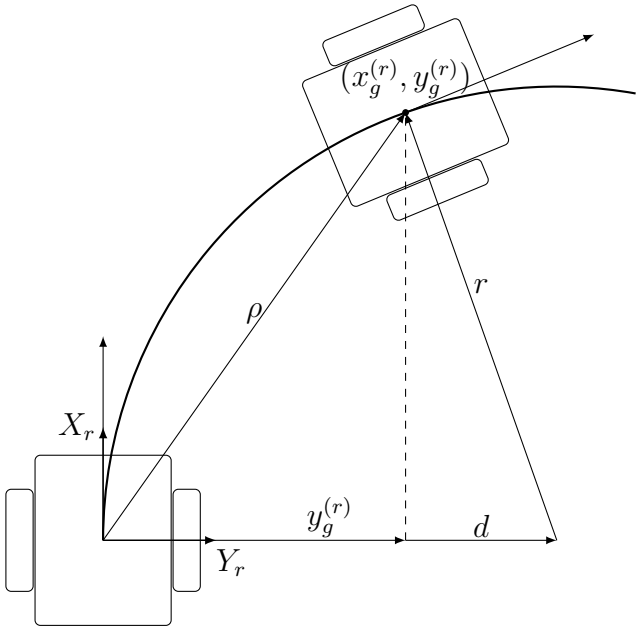

Figure 10: Geometry of the pure pursuit algorithm [13]

the first one is the distance to the lookahead point, the second relates the radius of the arc r and the lateral offset of the robot $y_g - y_r$ from the goal point. Combining the equations and solving for the searched radius r yields

$$r = \frac{\rho^2}{2y_g^{(r)}} \tag{15}$$

or in terms of curvature

$$\gamma = \frac{2y_g^{(r)}}{\rho^2} \tag{16}$$

The curvature provides the relationship between the robots translational velocity v and angular velocity ω

$$\omega = \gamma v \tag{17}$$

The remaining issue is to determine the lateral offset of the robot w.r.t. to the reference path. The implementation involves the following steps

- find the point (x_p, y_p) on the path closest to the current robot position
- determine the lookahead point (x_r, y_r) on the path a distance l from (x_p, y_p)
- transform the lookahead point into robocentric coordinates.

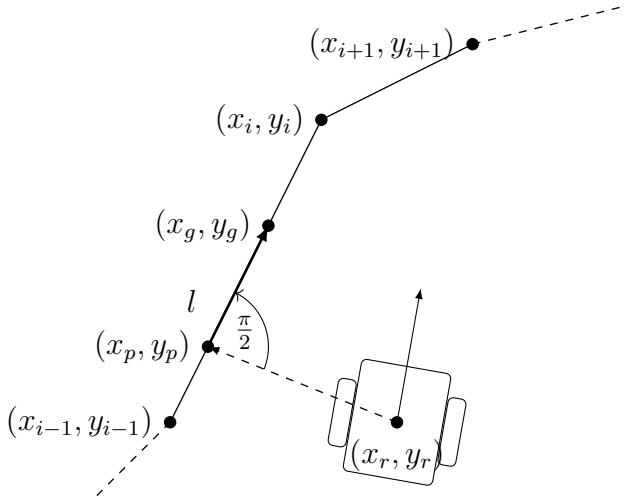

Figure 11: Look ahead point for a path composed of straight line segments [14]

- select a constant velocity v and compute turn rate ω from curvature γ.

Assume that the robot reference path is described by a sequence of waypoints $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ as shown in figure 11 The set of waypoints is either defined explicitly or generated by a path planner. The path is assumed to originate from straight line segments connecting the waypoints. The closest point on the path (x_p, y_p) is determined by projection of the robots current pose x_r, y_r onto the current line segment $(x_{i-1}, y_{i-1}) \rightarrow (x_i, y_i)$. The lookahead point (x_g, y_g) is located a lookahead distance l from (x_p, y_p) along the straight line segment.

If the remaining distance to (x_i, y_i) is less than l the lookahead point (x_g, y_g) is shifted towards the next segment $(x_i, y_i) \rightarrow (x_{i+1}, y_{i+1})$ as shown in figure 12,

Figures 13 and 14 illustrate the effects of small and large look ahead distance on path tracking. For small look ahead distances the robot trajectory overshoots the planned path or pure pursuit might even become unstable. For large look ahead distances the robot cuts the corner at waypoints.

The pure pursuit strategy is utilized in the Robotics toolbox example Path Following for a Differential Drive Robot.

13) **publisher lookahead point for navigation goal:**

[14] source: RST
[16] source: RST
[17] source: RST
[18] source: RST

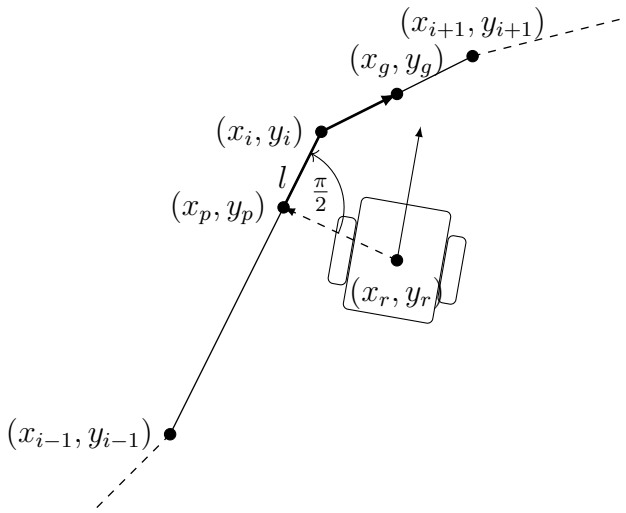

Figure 12: Look ahead point for a path composed of straight line segments with look ahead point on the next segment [16]

Following the same strategy as for the `'waypoint'` publisher shown in figure 9, implement and instantiate a publisher `lookAheadPointPub` that publishes the current lookahead point as a navigation goal topic `'/lookaheadpoint'` to which `navGoalSub` subscribes.

```
lookAheadPointPub=rospublisher('/lookaheadpoint','geometry_msgs/PoseStamped
');
navGoalSub=rossubscriber('/lookaheadpoint');
```

Instantiate a callback subscriber that monitors the robot pose

```
odomLookAheadPointSubCallback = rossubscriber('/odom',{
@odomLookAheadPointCallback,lookAheadPointPub, lookAheadDistance});
```

A callback function

```
function [] = odomLookAheadPointCallback( ~, odomMsg, lookAheadPointPub,
lookAheadDistance)
```

is provided as a helper function. It publishes the current look ahead point as `'geometry_msgs/PoseStamped'` message on the topic `'/lookaheadpoint'`. It automatically transitions to the next segment and removes waypoints that are located on segments that the robot already completed. It stops publishing navigation goals once the list of waypoints is empty.

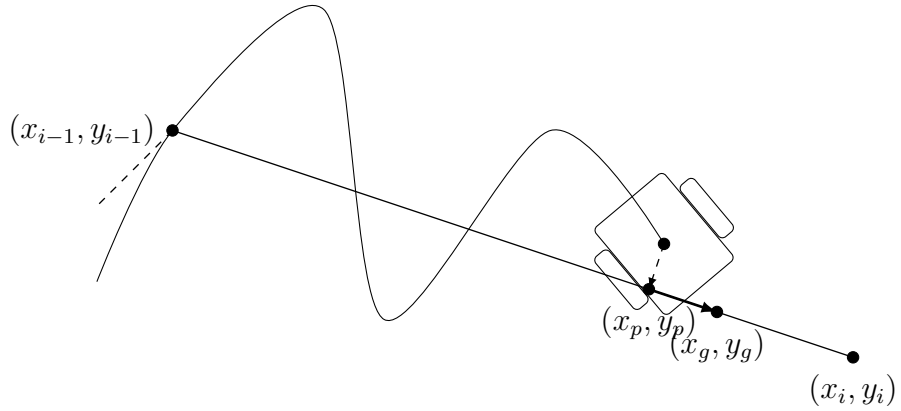

Figure 13: Effect of small look ahead distance on the smoothness of the path [17]

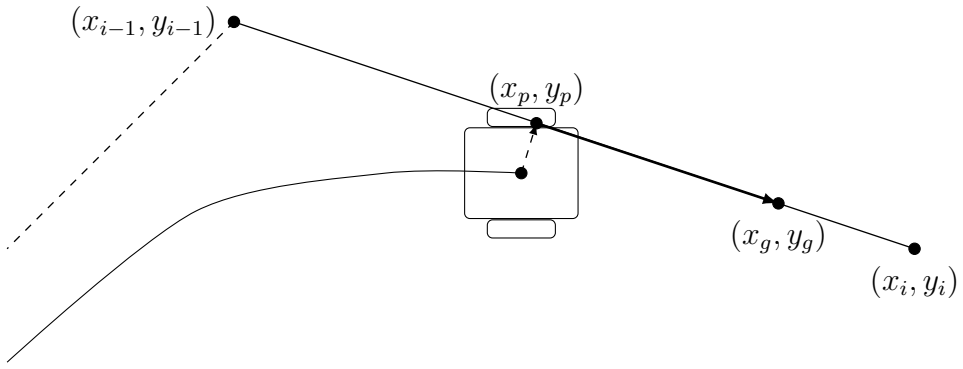

Figure 14: Effect of large look ahead distance on the path [18]

14) **pose error calculation:**

Augment the function `poseErrorTf` such that it provides the lateral displacement $y_g^{(r)}$ in robocentric coordinates (see figure 10) as an additional output parameter.

```
function [rho, alpha, phi, deltay] = poseErrorTf( navGoalMsg, tfTree )
```

15) **pure pursuit control linear velocity:**

Implement the pure pursuit controller according to equations (16) and (17) for a constant forward velocity $v = 0.5$. Modify the function

```
function [ v, omega ] = homingControl( rho, alpha, phi, deltay )
```

to implement the curvature based control law.

16) **testing and visualisation:**

Test your pure pursuit controller for the sequence of waypoints.

```
    waypoints=[0 0 0; 4 4 0; 8 0 0; 12 4 0; 16 0 0];
2
```

17) **analysis of the effect of lookahead distance (optional):**

Investigate the effect of the lookahead distance on the tracking performance. Obtain a statistics of the average lateral offset of the robot from the straight line path as a function of the lookahead distance.

References

[1] Morin P., Samson C., Motion Control of Wheeled Robots, Springer Handbook of Robotics

[2] Coulter, R. , Implementation of the Pure Pursuit Path Tracking Algorithm, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1990.

[3] Astolfi, A., Exponential Stabilization of a Wheeled Mobile Robot Via Discontinuous Control, Journal of Dynamic Systems, Measurement, and Control, vol. 121, 1999