Cyclic Pursuit and Circular Path Following
UCSD Distributed ROS Project

Aaron Ma

March 22, 2015

University of California, San Diego

Abstract

Deployment of cyclic pursuit and circular path following algorithms have been implemented in the TurtleBot lab. Many kinds of cyclic pursuits have been studied. In our lab, we have focused on one cyclic pursuit scheme where TurtleBots are constrained to a circle and input is proportional to the next closest TurtleBot in the counterclockwise direction. In order to constrain the TurtleBots to a circle, it is necessary to have a circular path following algorithm that converges the TurtleBots unicycle kinematics to a prescribe radius and origin.

1 Big Picture

1.1 Circular Path Following

In order to deploy simpler cyclic pursuit algorithms, being able to constrain turtlebot motion to a circle is necessary. Being able to force converges to circles can be important in general for other use in algorithms where we simply need to control the TurtleBot.

We use the algorithm described in "Global Path Following for the Unicycle and Other Results" by Mohamed I, El-Hawwary and Manfredi Maggiore. In this paper they use Lyapunov stability to prove global convergence to a circle given a velocity and radius for unicycle kinematics. This paper also provides an algorithm for converging a vehicle with unicycle dynamics to a position with specific heading, which might be important for later lab work. It is also necessary to use this algorithm because of fluctuating TurtleBot dynamics. Two TurtleBots given the same input velocities will traverse a circle of different radius and speeds. This is also the best way to converge to a orbit around a desired origin.

1.2 Cyclic Pursuit

There are many different kinds of cyclic pursuit algorithms that have been studied. Many applications range from study of nature to distribution of satellites. In this paper, I will describe the implementation of one algorithm, and describe how to implement others.

The algorithm presented in this paper assumes a scenario where the TurtleBots are constrained to a prescribed radius $\bar{r}$. The algorithm needs information on the TurtleBots position to compute its relative angular position and will output a velocity that is proportional to the angular distance between the TurtleBot and the next closest TurtleBot in the counterclockwise direction.

$$v = k[\theta_{i+1} - \theta_i]$$
$$\theta_i(t+1) - \theta_i(t) = k[\theta_{i+1}(t) - \theta_i(t)]$$

With this algorithm, it can be seen that if $k \in (0, 1)$ (if the TurtleBots next position isn't greater than the angular distance), the TurtleBots will reach steady state angular velocity around the prescribed circle and be equally spaced apart.

2 Personal Contributions

2.1 Implementing Circular Path Following

Circular path following is implemented to force stability and convergence to the prescribed circle so that assumptions made about cyclic pursuit are true. I execute circular path following as a ROS node that runs on board the TurtleBots. A fair amount of debugging and tuning of the algorithm gains were necessary to insure quick but not oscillatory convergence to the circle.

2.2 Implementing Cyclic Pursuit

After circular path following was implemented on the TurtleBots, the cyclic pursuit algorithms could be implemented. Initially cyclic pursuit would only work with two TurtleBots of specific identification, but the algorithm has been modified to work with any amount of TurtleBots being activated. The TurtleBots converge to the circle, come to a steady state velocity (with noise), steady state angular position error (mostly fixed by integral control via Kalman filter), and cover the circle evenly.

3 Preliminaries

3.1 Circular Path Following

3.1.1 System Model

In order to employ circular path following, it is important to understand nonholonomic (underactuated) unicycle kinematics that the Turtlebots are subject to.

$$x_{1,k+1} = x_{1,k} + u_1 cos(x_3)$$
$$x_{2,k+1} = x_{2,k} + u_1 sin(x_3)$$
$$x_{3,k} = x_{3,k} + u_2$$

Where x_1, x_2 and x_3 are the x and y coordinates, and the heading angle of the TurtleBot.

We will generalize $u_1 = v$ and $u_2 = \omega$, that is the inputs are the linear and angular velocity, respectively, published to the TurtleBots. Our circular path following algorithm will act as a function that takes the desired radius, and linear velocity of the TurtleBots (determined by cyclic pursuit), and output an angular velocity, ω, necessary to converge to the desired circle.

3.1.2 Control Law and Lyapunov Stability

Mohamed I. El-Hawwary and Manfredi Maggiore introduce the control law for converging to a desired circle of radius $\bar{r}$, and origin $\bar{x}, \bar{y}$.

$$u_1 = v$$
$$u_2 = \frac{v}{\bar{r}} + u_c$$
$$u_2 = \frac{v}{\bar{r}} + \psi(\bar{r}x_1 cosx_3 + \bar{r}x_2 sinx_3)$$

Intuitively, the control to this algorithm drives the TurtleBot to the desired trajectory. The authors first prove that the system

$$u_1 = v$$
$$u_2 = \frac{v}{\bar{r}}$$

Is trivially stable

$$V = \frac{1}{2}[(x_1 - rsinx_3)^2 + (x_2 + rcosx_3)^2],\ \boldsymbol{V(x) \geq 0}$$

$$\dot{V} = x_1\dot{x_1} - rsinx_3\dot{x_1} - rx_1cosx_3\dot{x_3} + r^2cosx_3sinx_3\dot{x_3} + x_2\dot{x_2} + rcosx_3\dot{x_2} - rx_2sinx_3\dot{x_3} - r^2cosx_3sinx_3\dot{x_3}$$

$$\text{Where } \dot{x} = \begin{bmatrix} \dot{x_1} \\ \dot{x_2} \\ \dot{x_3} \end{bmatrix} = \begin{bmatrix} vcosx_3 \\ vsinx_3 \\ \frac{v}{r} \end{bmatrix}$$

$$\dot{V} = x_1vcosx_3 - rvcosx_3sinx_3 - rx_1cosx_3(\frac{v}{r}) + r^2cosx_3sinx_3(\frac{v}{r})$$

$$+x_2vsinx_3 + rvcosx_3sinx_3 - rx_2sinx_3(\frac{v}{r}) - r^2cosx_3sinx_3(\frac{v}{r})$$

$$\boldsymbol{\dot{V} = 0}$$

Which tells us that the first part of the control system is trivially globally stable through Lyapunov stability.

3.1.3 Passivity Based Feedback

The authors next prove that u_c is passive, which would make the the control law globally asymptotically stable.

$$\dot{V} = x_1\dot{x_1} - rsinx_3\dot{x_1} - rx_1cosx_3\dot{x_3} + r^2cosx_3sinx_3\dot{x_3} + x_2\dot{x_2} + rcosx_3\dot{x_2} - rx_2sinx_3\dot{x_3} - r^2cosx_3sinx_3\dot{x_3}$$

$$\text{Where } \dot{x} = \begin{bmatrix} \dot{x_1} \\ \dot{x_2} \\ \dot{x_3} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\dot{V} = x_1(0) - rsinx_3(0) - rx_1cosx_3 + r^2cosx_3sinx_3 + x_2(0) + rcosx_3(0) - rx_2sinx_3 - r^2cosx_3sinx_3$$

$$\dot{V} = -rx_1cosx_3 - rx_2sinx_3 = \frac{u_c}{\psi}$$

Hence, u_c is passive.

3.1.4 Extension of Stability Analysis (non-passive analysis)

It is also interesting to plug u_c into the Lie derivative and find that with the u_c is globally stable through LaSalle's stability.

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \\ \dot{x_3} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \psi(rx_1cosx_3 + rx_2sinx_3) \end{bmatrix}$$

$$\dot{V} = x_1(0) - rsinx_3(0) - rx_1cosx_3\dot{x_3} + r^2cosx_3sinx_3\dot{x_3} + x_2(0) + rcosx_3(0) - rx_2sinx_3\dot{x_3} - r^2cosx_3sinx_3\dot{x_3}$$

$$\dot{V} = -rx_1cosx_3(\psi(rx_1cosx_3 + rx_2sinx_3)) + r^2cosx_3sinx_3(\psi(rx_1cosx_3 + rx_2sinx_3)) - rx_2sinx_3(\psi(rx_1cosx_3 + rx_2sinx_3)) - r^2cosx_3sinx_3(\psi(rx_1cosx_3 + rx_2sinx_3))$$

$$\dot{V} = -\psi r^2[x_1^2cos^2x_3 + x_2^2sin^2x_3 + 2x_1x_2cosx_3sinx_3]$$

$$\text{Where } \boldsymbol{V(x) \geq 0},\ \boldsymbol{\dot{V}(x) \leq 0}\ |\psi \geq 0, r \geq 0.$$

Which is the same result as if you were to take the LaSalle's stability analysis of the whole control law (3.1.2 + 3.1.4), which shows that system will approach the set of equilibrium points (on the circle) as $t \to \infty$.

3.2 Cyclic Pursuit

The cyclic pursuit that I decided to implement follows a simple control law.

$$\theta_i(t+1) = \theta_k(t) + k[\theta_{i+1}(t) - \theta i(t)]$$

Assuming no noise, the cyclic pursuit algorithm will converge for any value $k \in (0, 1)$.

$$\theta_i(t+1) - theta_i(t) = \theta_i(t) - \theta_i(t-1) + k[\theta_{i+1}(t) - \theta i(t) - \theta_{i+1}(t-1) + \theta_i(t-1)]$$
$$v_i(t) - v_i(t-1) = k(D_i(t) - D_i(t-1))$$

Where $v_i(t) - v_i(t-1)$ monotonically decreases as $t \to \infty$ when $0 < k < 1$.

4 Methodology

4.1 Testing and Convergence to Circle Path

4.1.1 Simulations

To test the circular path following algorithm before implementing it in TurtleBot deployment I used matlab. I made a class with a given position and heading with a goal to converge to a prescribed circle with radius and origin. Using the circular path following algorithm, the simulation how the TurtleBot would converge to a circle with varying gains.

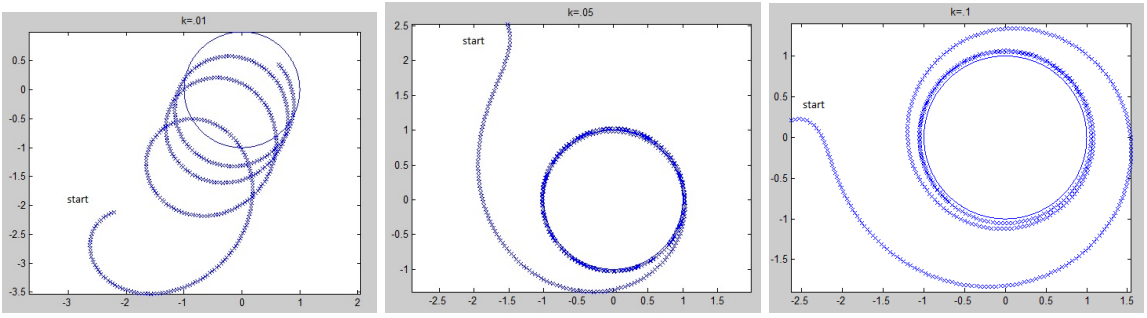

Figure 1: Left: This model shows a simulation where the proportional gain is too low. The trajectory converges very slowly due to overshoot.

Figure 2: Middle: The model shows very good convergence to the goal circle. This is the preferred trajectory for the TurtleBot to traverse.

Figure 3: Right: In this model, the proportional gain is too high, which forces the trajectory to converge slowly.

I found it important to compare the simulation results to the TurtleBot actual trajectories so that it is clear how to tune the gain. Finding the optimal gain has not been achieved yet. This can probably done using general optimal control methods (differential Riccati equation), but it the time varying model is undetermined so other methods might have be used to determine optimal gain online. For the time being, it is satisfactory to use a static "good" proportional gain that works for general cases.

4.1.2 Test Data

Test data is available to see the performance of how the TurtleBots follow the circular path once the algorithm is implemented. position data was published in order to track the TurtleBot as a function of time.

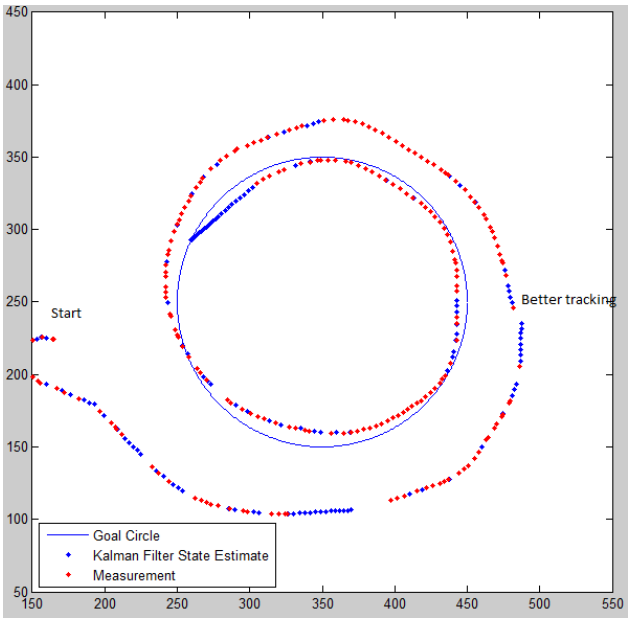

Figure 4: This is a representation of actual data collected from a TurtleBot that is following a circular path using the described algorithm. Notice that the actual robot is subject to measurement and state noise, yet still converges to the circle. It appears that the proportional gain lowered to decrease the convergence time.

4.2 Cyclic Pursuit Testing

4.2.1 Cyclic Pursuit Simulations

To roughly demonstrate the evolution of the TurtleBots constrained to a circle running cyclic pursuit here are a few simulations.

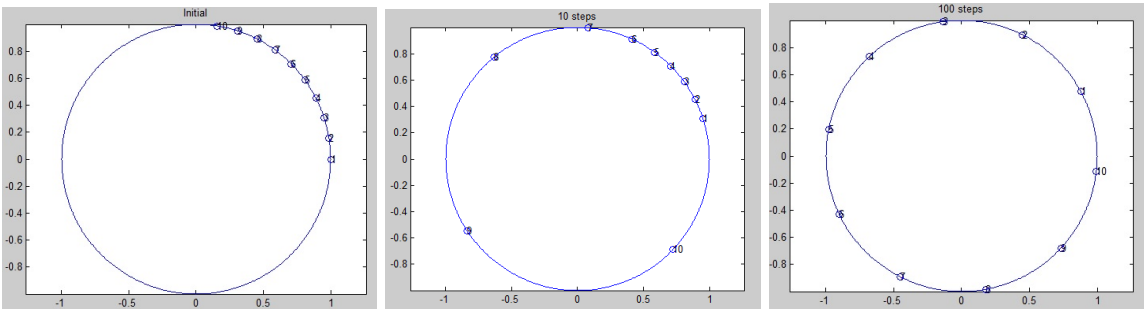

Figure 5: This is a mocked up initialized set up. In this model there are 10 TurtleBots on a circle who are grouped closely together

Figure 6: 10 steps in, the TurtleBots begin to spread, notice how 10 has traveled the furthest because the distance to the next TurtleBot in the counterclockwise direction is the greatest

Figure 7: 100 steps in, the TurtleBots have reached steady state equilibrium, where they are moving at roughly the same velocity and are equally spaced apart.

4.3 ROS Interface Design

The circular path following algorithm and the cyclic pursuit fits well into ROS network. The cyclic pursuit algorithm subscribes to "afterKalman" and "all_Positions" in order to get the positions of itself and the next closest TurtleBot in the counterclockwise direction. With the position data, the cyclic pursuit publishes the velocity to "velocity" based on the cyclic pursuit control law previously defined.

The circular pursuit algorithm makes sure that the TurtleBots converge to the circle for the cyclic pursuit. It subscribes to "velocity" to receive the velocity and calculates the angular velocity necessary to converge the TurtleBot to the circle using the circular path following control law. The circular path following node also subscribes to "cal10" and "cal1D" topics which are integral terms that allows the circular path following algorithm correct fluctuations in TurtleBot dynamics. The circular path following algorithm finally publishes the input linear and angular velocity to "mobile_" base/commands/velocity". A flowchart is represented in figure 5.

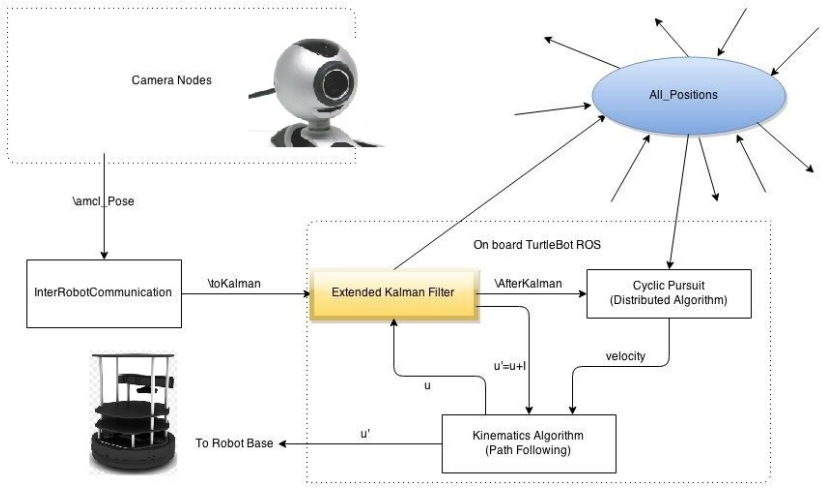

Figure 8: This flow chart shows the communication between cyclic pursuit, circular path following and other nodes.

5 Using the Cyclic Pursuit and Circular Path Following Algorithms

5.1 Using the Circular Path Following Algorithm

It might be necessary to use or modify the circular path following algorithm as a tool for other distributed deployment algorithms or maneuvers. The circular path following algorithm requires being subscribed to the topic "velocity" (otherwise a default value will be used). The radius and origin of the circle that it traces can be found in the launch file. It is also important that the circular path following algorithm (and any of our kinematic/dynamic positioning algorithms) to be subscribed to the extended Kalman filter's "cal10" and "cal1D" topics. Using these topics, fluctuations in state dynamics are taken care of.

5.2 Using the Cyclic Pursuit Algorithm

There is a lot to study with regards to cyclic pursuit. One of the potential projects would be to add an importance function to the map so that the TurtleBots would cover a certain part of the lab more densely, while executing cyclic pursuit. For reasons like this, it is important to learn how to use the cyclic pursuit algorithm.

To use the cyclic pursuit algorithm, the topic "all_Positions" must have published data. Cyclic pursuit subscribes to "all_Positions" in order to learn the other TurtleBots positions (important for a lot of our distributed algorithms). It is also important to understand the output of the simple cyclic pursuit algorithm that is implemented as well as the assumptions. Right now it is assumed that the TurtleBots are constrained to a circle, where the cyclic pursuit algorithm only needs to control the velocity to have the TurtleBots

converge to steady state. In order for the cyclic pursuit algorithm to work, it needs a supplementary kinematics algorithm (circular path following) so drive the TurtleBots.

6 Future Improvements

There are many improvements that can be made to the circular path following algorithm. The algorithm at hand is functional, but not optimal. Several things can be done to make the circular path following algorithm better. This includes finding optimal gain K, or using some positive definite function ψ which gives us quicker convergence to the desired circle.

The Cyclic pursuit algorithm is fine as it is, but many alterations can be made for interesting results. As mentioned before, the addition of an importance function on the circle it traverses could be a very interesting addition.

7 Conclusions

The circular path following algorithm is functional and the cyclic pursuit algorithm is complete.

8 Source Code

8.1 PathFollowing.cpp

```
/*
Path Following algorithm from
http://www.control.utoronto.ca/people/profs/maggiore/DATA/PAPERS/CONFERENCES/ACC08_2.pdf
*/

#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <nav_msgs/Odometry.h>
#include <std_msgs/Float64.h>
#include "geometry_msgs/PoseWithCovarianceStamped.h"
#include <turtlebot_deployment/PoseWithName.h>
#include <tf/tf.h>
#include <math.h>
#include <time.h>

//Declare Variables
double x2, x1, r;
double orientation;
double robVel_;
double OmegaC;
double OmegaD;
// Construct Node Class


class pathFollowing
{
public:
pathFollowing();
std::string this_agent_;
private:
// Methods

void poseCallback(const turtlebot_deployment::PoseWithName::ConstPtr&);
```

```
void velocityCallback(const geometry_msgs::Twist::ConstPtr&);
void cal0Callback(const std_msgs::Float64::ConstPtr&);
void calDCallback(const std_msgs::Float64::ConstPtr&);
// ROS stuff
ros::NodeHandle ph_, nh_;
ros::Subscriber pos_sub_;
ros::Subscriber vel_sub_;
ros::Subscriber cal0_sub_;
ros::Subscriber calD_sub_;

// Other member variables

geometry_msgs::Twist robVel;
turtlebot_deployment::PoseWithName Pose;

bool got_vel_;
};

pathFollowing::pathFollowing():
/*cmd_vel_(new geometry_msgs::Twist),
*/got_vel_(false),
ph_("~"),
this_agent_()
{
ph_.param("robot_name", this_agent_,this_agent_);
ph_.param("radius", r,r);
vel_sub_ = nh_.subscribe<geometry_msgs::Twist>("velocity",1, &pathFollowing::velocityCallback,
    this);
pos_sub_ = nh_.subscribe<turtlebot_deployment::PoseWithName>("afterKalman", 1,
    &pathFollowing::poseCallback, this);
cal0_sub_ = nh_.subscribe<std_msgs::Float64>("cal0",1, &pathFollowing::cal0Callback, this);
calD_sub_ = nh_.subscribe<std_msgs::Float64>("calD",1, &pathFollowing::calDCallback, this);
}

void pathFollowing::cal0Callback(const std_msgs::Float64::ConstPtr& OmegaC_){
OmegaC=OmegaC_->data;
}
void pathFollowing::calDCallback(const std_msgs::Float64::ConstPtr& OmegaD_){
OmegaD=OmegaD_->data;
}

void pathFollowing::velocityCallback(const geometry_msgs::Twist::ConstPtr& robVel){
robVel_= robVel->linear.x;
got_vel_ = true;
}

void pathFollowing::poseCallback(const turtlebot_deployment::PoseWithName::ConstPtr& Pose)
{
   orientation = tf::getYaw(Pose->pose.orientation);
   //orientation=-orientation;
   x1=Pose->pose.position.x;
   x1=x1-350;
   x2=Pose->pose.position.y; //centered
   x2=x2-250;

// got_vel_=false; *Delete

}
int main(int argc, char **argv)
```

```
{
ros::init(argc, argv, "PathFollowing");
r=100;
time_t timer,begin,end;
ros::NodeHandle ph_("~"), nh_;
ros::Publisher u_pub_;
geometry_msgs::Twist cmd_vel_;
u_pub_ = nh_.advertise<geometry_msgs::Twist>("mobile_base/commands/velocity", 1, true);
pathFollowing pathFollowingk;
robVel_=0;
time(&end);
double k=1;
ros::spinOnce();
double u1=robVel_;
double u2=robVel_/r;
OmegaC=2;
OmegaD=1;
while(1==1){
   //ph_.param("radius", r,r);
      //while ((time(&begin)-end)>.1){
         ros::spinOnce();
         u1=robVel_;
         //pathFollowingk.pathFollowing();
         u2=robVel_/r;
         //std::cout<<"initial angular velocity: \n"<<u2<<"\n\n";
         u2=u2-k*(r*x1*cos(orientation)+r*x2*sin(orientation))/167/167; //check orientation units
         u2=u2*OmegaC;
         u1=u1*OmegaD;
         if (u2>1){u2=1;}
         if (u2<-1){u2=-1;}

         //std::cout<<"final angular velocity: \n"<<u2<<"\n\n";
         cmd_vel_.linear.x=(u1/167);
         cmd_vel_.angular.z=(u2);
         u_pub_.publish(cmd_vel_);
      // time(&end);
   // }

         usleep(600000);
}

}
```

8.2 CyclicPursuit.cpp

```
/*
Cyclic Pursuit
*/

#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <nav_msgs/Odometry.h>
#include "PoseWithName.h"
#include "geometry_msgs/PoseWithCovarianceStamped.h"
#include <tf/tf.h>
#include <math.h>
#include <stdio.h>
```

```
#include <stdlib.h>

double rad1,x0, z;
double yc;
double rad2,x2,y2,radN;

std::string name_, name2_;


void selfCallback(const turtlebot_deployment::PoseWithName::ConstPtr& selfPtr)
{
name_=selfPtr->name;
x0=selfPtr->pose.position.x-350;
yc=selfPtr->pose.position.y-250;
z=yc/x0;
rad1=atan(z);
if (x0>=0)
{
    if (yc<0){
        rad1=rad1+2*3.14;
        }
    }
else
{
    if (yc<0){
        rad1=rad1+3.14;
    }
    else
    {
        rad1=rad1+3.14;
    }
}
}

void allPoseCallback(const turtlebot_deployment::PoseWithName::ConstPtr& posePtr)
{
    if (name_!=posePtr->name){
      x2=posePtr->pose.position.x-350;
      y2=posePtr->pose.position.y-250;
      rad2=atan(y2/x2);

if (x2>=0)
{
    if (y2<0){
        rad2=rad2+2*3.14;
        }
    }
else
{
    if (y2<0){
        rad2=rad2+3.14;
    }
    else
    {
        rad2=rad2+3.14;
    }
}

        if(rad2<rad1){
```

```
            rad2=3.14*2-rad1+rad2;
        }
        else{
          rad2=rad2-rad1;
        }

        /*if (rad2<radN){
          radN=rad2;
        }*/
        if (name2_=="temp"){
            name2_=posePtr->name;
        }
        if (name2_==posePtr->name){
        radN=rad2;
        }
        else {
            if (rad2<radN){
                name2_=posePtr->name;
                radN=rad2;
            }
        }
    }
}

int main(int argc, char **argv)
{
ros::init(argc, argv, "cyclic_pursuit");
ros::NodeHandle ph_, nh_;
ros::Publisher vel_pub_;
ros::Subscriber pos_sub_;
ros::Subscriber self_sub_;
geometry_msgs::Twist cmd_vel_;
ros::Rate loop_rate(.2);
vel_pub_ = nh_.advertise<geometry_msgs::Twist>("velocity", 5, true);
pos_sub_ = nh_.subscribe<turtlebot_deployment::PoseWithName>("/all_positions", 1,allPoseCallback);
self_sub_ = nh_.subscribe<turtlebot_deployment::PoseWithName>("afterKalman",1,selfCallback);
cmd_vel_.linear.x=25;
double k=1;
radN=1;
rad2=1;
rad1=1;
yc=0;
y2=0;
x0=0;
x2=0;
name2_="temp";
while (1==1){

    ros::spinOnce();

std::cout<<"RADIANS1: "<<rad1<<"\n";
std::cout<<"RADIANS2: "<<rad2<<"\n";
    cmd_vel_.linear.x=6*k*(radN);
    vel_pub_.publish(cmd_vel_);
    usleep(100000);

}
}
```