**Оглавление**

[ВВЕДЕНИЕ 3](#_Toc148124059)

[1. Задание 4](#_Toc148124060)

[1.1. Устройство simple\_controller 5](#_Toc148124061)

[1.2. Траектория движения 6](#_Toc148124062)

[1.3. Текущее устройство контроллера 7](#_Toc148124063)

[2. Ход работы 8](#_Toc148124064)

[ЗАКЛЮЧЕНИЕ 13](#_Toc148124065)

[ПРИЛОЖЕНИЯ 14](#_Toc148124066)

# введение

**Цель домашней работы:** разработать модуль управления движением мобильного робота (МР) по траектории.

**Задачи:**

1. Переделать модуль управления движением вдоль траектории.
2. Реализовать один из алгоритмов, описанных в лекции.
3. Изменить функцию таймера.
4. Реализовать устойчивое управление на скоростям до 10 м/сек.

**1. Задание**

Управляем движением МР в виртуальной среде Gazebo. В качестве МР выступает модель карта с кинематической схемой автомобиля (Ackerman steering – поворачивающиеся передние колеса).

МР оснащен сканирующим лазерным дальномером и другими датчиками (в Gazebo) (рисунок 1).

|  |
| --- |
|  |
| Рисунок 1 – Мобильный робот |

В этой задаче не используются данные датчиков, только одометрия и данные о положении робота, которые приходят из симулятора.

Управление роботом в этой задаче сводится к управлению поворотом переднего колеса велосипедной модели (топик /steering rуда можно отправить желаемый угол поворота руля в рад). Скорость движения по траектории задается отдельным сообщением (топик /velocity куда можно отправить желаемую скорость в м/сек).

В случае Gazebo за передачу команд управления в модель отвечает vehicle\_ros\_plugin. Моделируются ограничения на параметр управления и скорость его изменения (линейное ускорение и скорость вращения рулевого колеса соответственно). Если задать угол поворота руля, то он начнет меняться от текущего к заданному с фиксированной скоростью (задается в launch файле для модели stage).

Модуль simple\_controller был доработан и реализует управление вдоль заданной траектории. Запуск модуля управления с моделью был осуществлен с помощью launch файла controller.launch для gazebo, команда:

roslaunch simple\_controller controller.launch

В файле прописан запуск модели с нужными для работы модулями, запуск модуля simple\_controller и запуск панели rqt с нужными плагинами, среди которых: задание скорости движения через message\_publisher, графики ошибок управления, контроллеры для настройки регулятора (в текущей реализации).

* 1. **Устройство simple\_controller**

simple\_controller управляет движением МР по заданной траектории за счет управления поворотом рулевого колеса (топик/steering). Скорость движения задается извне, в данном случае из rqt (с помощью плагина publish massage) simple\_controller реализован в виде класса Controller, часть функций которого является колбеками, вызываемыми библиотекой ros при получении сообщений. В частности, он подписан на сообщения:

1. о текущем положении МР on\_pose;
2. с одометрией (получение текущей скорость) on\_odo;
3. обработка таймера on\_timer, где выполняется вся логика работы; модуля: вычисляется необходимый угол поворота рулевого колеса в зависимости от положения МР относительно траектории.

Модуль публикует:

1. управление МР в виде команды поворота переднего колеса с помощью паблишера steer\_pub;
2. текущую ошибку управления (которая отображается на графике в rqt) с помощью паблишера err\_pub;
3. траектория движения в виде облака точек для визуализации rviz в функции publisher\_trajectory.
   1. **Траектория движения**

В данной задаче траектория движения задается локально в виде набора сегментов с постоянной кривизной: прямых и дуг окружностей.

Программно каждый сегмент представлен в виде объектов класса [CircularSegment] или [LinearSegment], [TrajectorySegment], что позволяет работать с разными сегментами единым образом. Каждый сегмент задается некоторой начальной точкой, заданной кривизной, длиной и направлением (определяется ориентация в начальной точке).

В классах сегментах реализованы следующие функции:

– get\_lenght – возвращает длину сегмента;

– get\_curvature – возвращает кривизну текущего сегмента;

– get\_point(double point\_len) – возвращает точку (вектор x, y, z) сегмента на заданной длине от начала (т.е. get\_point(0.0) – вернет начальную точку сегмента);

– get\_point\_lenght(x,y) – вернет длину сегмента до точки, ближайшей к заданной;

– get\_point\_distance(x,y) – вернет расстояние от заданной точки до ближайшей точки сегмента с учетом направления.

Траектория задается как массив (std::list) trajectory из нескольких сегментов в конструкции контроллера.

Траектория образует замкнутый овал. В конструкторе траектория пересчитывается в сообщение [nav\_msgs::Path].

(<http://docs.ros.org/en/noetic/api/nav_msgs/html/msg/Path.html>) - стандартное сообщение ROS для задания траектории движения. Также траектория может быть задана за счет соответствующего сообщения, переданного в топик `path`.

* 1. **Текущее устройство контроллера**

Модуль управления движением вдоль траектории реализован в виде ПИД регулятора, входом которого (ошибкой) является расстояние до ближайшей точки траектории, а выходом желаемый угол управления. Начальное положение МР соответствует начальной точке первого сегмента. Далее в таймере контроллера -update\_robot\_pase – обновляется положение МР с учетом полученных данных (последнее положение + скорость \*dt)

– get\_narest\_path\_pose\_index – обновляем индекс ближайшей точки траектории к положению МР.

– Вычисляем ошибку – расстояние до траектории в координатах ближайшей точки. Учитываем знак.

– Считаем регулятор угловой скорости.

– Пересчитываем угловую скорость в кривизну.

– Отправляем кривизну в /stering.

– Публикуем траекторию.

– Публикуем текущую ошибку.

**2. Ход работы**

Описание программы файлов controller.cpp и controller.h.

Файл **controller.cpp:**

Метод Cjntroller::update\_robot\_pose(double dt) обновляет положение автомобиля на основе его текущей скорости и времени прошедшего с предыдущего обновления.

void Controller::update\_robot\_pose(double dt)

{

ROS\_DEBUG\_STREAM("update\_robot\_pose "<<dt<<" v = "<<current\_linear\_velocity );

robot\_x += current\_linear\_velocity \* dt \* sin(robot\_theta);

robot\_y += current\_linear\_velocity \* dt \* cos(robot\_theta);

robot\_theta = angles::normalize\_angle(robot\_theta + current\_angular\_velocity \* dt);

robot\_time += ros::Duration(dt);

}

Сохранение и инициализация переменных определения ближайшей точки и целевой точки на пути для автомобиля:

void Controller::on\_path(const nav\_msgs::Path& path) {

ROS\_INFO\_STREAM("Got path " << path.poses.size());

this->path = path;

nearest\_point\_index = 0;

target\_point\_index = 0;

}

Поиск ближайшей точки относительно текущего положения:

{

double nearest\_distance = 1e10;

std::size\_t index = start\_index;

std::size\_t nearest\_index;

geometry\_msgs::Pose nearest\_pose;

for (int index = start\_index; index < start\_index + static\_cast<int>(search\_len); ++index) {

std::size\_t real\_index;

if (index >= 0 && index < static\_cast<int>(path.poses.size())) {

real\_index = static\_cast<std::size\_t>(index);

}

if (index < 0) {

real\_index = (static\_cast<int>(path.poses.size()) + index);

}

if (index >= static\_cast<int>(path.poses.size())) {

real\_index = static\_cast<std::size\_t>(index) - path.poses.size();

}

const auto& path\_point = path.poses[real\_index].pose.position;

double dx = robot\_x - path\_point.x;

double dy = robot\_y - path\_point.y;

double distance\_sqr = dx \* dx + dy \* dy;

if (distance\_sqr < nearest\_distance) {

nearest\_distance = distance\_sqr;

nearest\_index = real\_index;

}

}

return nearest\_index;

}

Вычисление необходимых значений ПИД регулятора по входящей ошибке:

double Controller::get\_pid\_control(double error)

{

double diff\_err = error - last\_error;

last\_error = error;

if ( fabs(error) < max\_antiwindup\_error )

error\_integral += error;

else

error\_integral = 0.0;

//Desired angular velocity

double cmd = p\_factor \* error

+ d\_factor \* diff\_err

+ i\_factor \* error\_integral;

return cmd;

}

Поиск целевой точки относительно текущего положения:

std::size\_t Controller::get\_target\_path\_pose\_index(int old\_target\_index, double ld)

{

double distance\_sqr = 0;

std::size\_t real\_index;

for (int index = old\_target\_index - 10; distance\_sqr < ld\*ld; ++index) {

if (index < 0) {

real\_index = (static\_cast<int>(path.poses.size()) + index);

}

else if (index >= static\_cast<int>(path.poses.size())) {

real\_index = static\_cast<std::size\_t>(index) - path.poses.size();

}

else{

real\_index = static\_cast<std::size\_t>(index);

}

const auto& path\_point = path.poses[real\_index].pose.position;

double dx = robot\_x - path\_point.x;

double dy = robot\_y - path\_point.y;

distance\_sqr = dx \* dx + dy \* dy;

}

return real\_index;

}

Метод для подруливания автомобиля через каждые 0.1 секунду:

1. Обновляем положение автомобиля.
2. Вычисляем кривизну траектории.
3. Применяем ПИД регулятор для вычисления угла поворота руля автомобиля.
4. Публикуем все результаты, включая угол поворота руля и саму траекторию.
5. void Controller::on\_timer(const ros::TimerEvent& event)
6. {
7. if (std::abs(current\_linear\_velocity) < 0.01) {
8. return;
9. }
10. update\_robot\_pose((ros::Time::now() - robot\_time).toSec() );
11. double lookahead\_distance = 3.0;
12. target\_point\_index = get\_target\_path\_pose\_index(target\_point\_index, lookahead\_distance);
13. const auto& target\_pose = path.poses[target\_point\_index].pose;
14. double x = target\_pose.position.x - robot\_x;
15. double y = target\_pose.position.y - robot\_y;
16. double error = -x\*sin(robot\_theta) + y\*cos(robot\_theta);
17. double curvature = 2.0\*error/(lookahead\_distance\*lookahead\_distance);
18. std\_msgs::Float32 cmd;
19. cmd.data = clip<double>(curvature, max\_curvature);
20. steer\_pub.publish(cmd);
21. //send trajectory for velocity controller
22. publish\_trajectory();
23. //send error for debug proposes
24. publish\_error(error);
25. ROS\_DEBUG\_STREAM("steering cmd = " << curvature);
26. }

Обновляем текущие координаты автомобиля по данным одометрии:

void Controller::on\_pose(const nav\_msgs::OdometryConstPtr& odom)

{

robot\_x = odom->pose.pose.position.x;

robot\_y = odom->pose.pose.position.y;

robot\_theta = 2\*atan2(odom->pose.pose.orientation.z,

odom->pose.pose.orientation.w);

world\_frame\_id = odom->header.frame\_id;

robot\_time = odom->header.stamp;

}

Обновляем линейную и угловую скорость по данным одометрии:

void Controller::on\_odo(const nav\_msgs::OdometryConstPtr& odom)

{

current\_linear\_velocity = odom->twist.twist.linear.x;

current\_angular\_velocity = odom->twist.twist.angular.z;

//ROS\_DEBUG\_STREAM("odom vel = "<<current\_velocity);

}

Публикация ошибки пройденного пути:

void Controller::publish\_error(double error)

{

std\_msgs::Float32 err\_msg;

err\_msg.data = error;

err\_pub.publish(err\_msg);

}

Вычисление ошибки отклонения автомобиля от траектории:

double Controller::cross\_track\_error()

{

double error = 0.0;

if (robot\_y < radius)

{

double rx = robot\_x;

double ry = robot\_y - radius;

error = sqrt(rx\*rx + ry\*ry) - radius;

}

else if ( robot\_y > cy)

{

double rx = robot\_x;

double ry = robot\_y - cy;

error = sqrt(rx\*rx + ry\*ry) - radius;

}

else if ( robot\_x > 0 )

{

error = robot\_x - radius;

}

else if ( robot\_x < 0 )

{

error = -radius - robot\_x;

}

return error;

}

Публикация траектории:

void Controller::publish\_trajectory()

{

// ROS\_DEBUG\_STREAM("publish trajectory");

path\_pub.publish(path);

}

Сброс значений ошибок ПИД регулятора:

void Controller::reset()

{

error\_integral = 0.0;

last\_error = cross\_track\_error();

}

Перегрузка метода. Сбрасываются ошибки и задаются значения ПИД регулятора:

void Controller::reset(double p, double d, double i)

{

reset();

p\_factor = p;

d\_factor = d;

i\_factor = i;

}

Создание и публикация пути на основе текущей траектории автомобиля:

nav\_msgs::Path Controller::create\_path() const {

//prepare path message from trajectory

nav\_msgs::Path path;

path.header.frame\_id = "odom";

path.header.stamp = robot\_time;

auto segment\_it = trajectory.begin();

double previous\_segment\_left = 0.0;

std::size\_t points\_added = 0;

double point\_length = 0.0;

while (segment\_it != trajectory.end()) {

const auto segment = \*segment\_it;

double segment\_length = segment->get\_length();

//add points from the segment

while (point\_length <= segment\_length) {

const auto point = segment->get\_point(point\_length);

const auto angle = segment->get\_orientation(point\_length);

geometry\_msgs::PoseStamped pose;

pose.header.frame\_id = "odom";

pose.pose.position.x = point.x();

pose.pose.position.y = point.y();

pose.pose.orientation = tf::createQuaternionMsgFromYaw(angle);

path.poses.push\_back(pose);

point\_length += traj\_dl;

points\_added++;

}

point\_length -= segment\_length;

++segment\_it;

}

return path;

}

Полный код файла controller.cpp представлен в Приложении А.

Файл controller.h.

Включает в себя класс Controller, который необходим для управления автомобилем вдоль заданной траектории с использованием ПИД регулятра.

nh объект класса ros::NodeHandle, через него идет взаимодействие с ROS.

robot\_x, robot\_y, robot\_theta – текущие координаты и угол положения автомобиля.

robot\_time – время последнего обновления координат и угла автомобиля.

p\_factorv – коэффициент пропорциональности ПИД регулятора

d\_factor – коэффициент дифференцирования ПИД регулятора.

i\_factor – коэффициент интегрирования ПИД регулятора.

max\_antiwindup\_error – необходимо, чтобы задавать максимальное значение ошибки, при которой будет использоваться коэффициент интегрирования регулятора.

error\_integral – значение ошибки коэффициента интегрирования.

last\_error – значение, которое показывает предыдущее значение ошибки коэффициента дифференцирования.

radius – радиус окружности траектории.

cy – координата центра окружности траектории.

max\_curvature – максимальное значение кривизны траектории.

current\_linear\_velocity – текущая линейная скорость.

current\_angular\_velocity – текущая угловая скорость автомобиля

traj\_dl – расстояние между двумя точками траектории.

traj\_lenght – длина траектории.

trajectory – список с сегментами траектории автомобиля, сегменты могут быть прямыми и круглыми.

nav\_msgs::Path path – путь движения автомобиля.

near\_point\_index – ближайшая точка пути.

target\_point\_index – целевая точка пути.

сurrent\_segment – указатель на текущий сегмент.

current\_segment\_lenght – длина текущего сегмента траектории на текущей позиции автомобиля.

ros::Subscriber pose\_sub – субскрайбер на положение.

ros::Subscriber – субскрайбер на одометрию.

ros::Subscriber path\_sub – субскрайбер на путь.

ros::Timer timer – таймер, который вызывается через определенный промежуток времени (0,1 секунды) для подруливания автомобилем.

ros::Publisher err\_pub – паблишер для ошибки.

ros::Publisher steer\_pub – паблишер для угла поворота руля.

ros::Publisher path\_pub – паблишер для пути.

Полный код файла сontroller.h представлен в Приложении Б.

**ЗАКЛЮЧЕНИЕ**

В ходе выполнения домашнего задания цель и задачи были выполнены.

Был переделан модуль управления движением вдоль траектории, посредством реализации алгоритма (управление устойчво на скоростях 10 м/сек). Изменения в большей мере отразились на функции таймера.

**ПРИЛОЖЕНИЯ**

**ПРИЛОЖЕНИЕ А**

Код файла controller.cpp:

/\*

\* Controller.cpp

\*

\* Created on: 30 апр. 2017 г.

\* Author: aminin

\*/

#include "controller.h"

#include <sensor\_msgs/PointCloud.h>

#include <tf/transform\_datatypes.h>

namespace simple\_controller

{

template <class T>

T clip(T val, T max)

{

if ( val > max )

return max;

if ( val < -max)

return -max;

return val;

}

void Controller::update\_robot\_pose(double dt)

{

// Этот метод обновляет положение автомобиля на основе его текущей скорости

// и времени прошедшего с предыдущего обновления.

ROS\_DEBUG\_STREAM("update\_robot\_pose "<<dt<<" v = "<<current\_linear\_velocity );

robot\_x += current\_linear\_velocity \* dt \* sin(robot\_theta);

robot\_y += current\_linear\_velocity \* dt \* cos(robot\_theta);

robot\_theta = angles::normalize\_angle(robot\_theta + current\_angular\_velocity \* dt);

robot\_time += ros::Duration(dt);

}

void Controller::on\_path(const nav\_msgs::Path& path) {

// Здесь происходит сохранение и инициализация переменных определения ближайшей точки и целевой точки на пути для автомлбиля

ROS\_INFO\_STREAM("Got path " << path.poses.size());

this->path = path;

nearest\_point\_index = 0;

target\_point\_index = 0;

}

// search a pose in the pat nearest to the robot, assume path may be cyclic

std::size\_t Controller::get\_nearest\_path\_pose\_index(int start\_index,

std::size\_t search\_len)

{

// Здесь происходит поиск этой ближайшей точки относительно текущего положения

double nearest\_distance = 1e10;

std::size\_t index = start\_index;

std::size\_t nearest\_index;

geometry\_msgs::Pose nearest\_pose;

for (int index = start\_index; index < start\_index + static\_cast<int>(search\_len); ++index) {

std::size\_t real\_index;

if (index >= 0 && index < static\_cast<int>(path.poses.size())) {

real\_index = static\_cast<std::size\_t>(index);

}

if (index < 0) {

real\_index = (static\_cast<int>(path.poses.size()) + index);

}

if (index >= static\_cast<int>(path.poses.size())) {

real\_index = static\_cast<std::size\_t>(index) - path.poses.size();

}

const auto& path\_point = path.poses[real\_index].pose.position;

double dx = robot\_x - path\_point.x;

double dy = robot\_y - path\_point.y;

double distance\_sqr = dx \* dx + dy \* dy;

if (distance\_sqr < nearest\_distance) {

nearest\_distance = distance\_sqr;

nearest\_index = real\_index;

}

}

return nearest\_index;

}

double Controller::get\_pid\_control(double error)

{

// Здесь вычисляются необходимые значения ПИД-регулятора по входящей ошибке

double diff\_err = error - last\_error;

last\_error = error;

if ( fabs(error) < max\_antiwindup\_error )

error\_integral += error;

else

error\_integral = 0.0;

//Desired angular velocity

double cmd = p\_factor \* error

+ d\_factor \* diff\_err

+ i\_factor \* error\_integral;

return cmd;

}

std::size\_t Controller::get\_target\_path\_pose\_index(int old\_target\_index,

double ld)

{

// Здесь происходит поиск уже целевой точки относительно текущего положения

double distance\_sqr = 0;

std::size\_t real\_index;

for (int index = old\_target\_index - 10; distance\_sqr < ld\*ld; ++index) {

if (index < 0) {

real\_index = (static\_cast<int>(path.poses.size()) + index);

}

else if (index >= static\_cast<int>(path.poses.size())) {

real\_index = static\_cast<std::size\_t>(index) - path.poses.size();

}

else{

real\_index = static\_cast<std::size\_t>(index);

}

const auto& path\_point = path.poses[real\_index].pose.position;

double dx = robot\_x - path\_point.x;

double dy = robot\_y - path\_point.y;

distance\_sqr = dx \* dx + dy \* dy;

}

return real\_index;

}

void Controller::on\_timer(const ros::TimerEvent& event)

{

// Собственно метод для подруливания автомобиля через каждые 0.1 секунду

/\*1) Сначала мы обновляем положение автомобиля

2) вычисляем ошибку относительно целевой точки на пути

3) вычисляем кривизну траектории

4) применяем ПИД-регулятор для вычисления угла поворота руля автомобиля,

5) и публикуем все результаты, включая угол поворота руля и саму траекторию.\*/

if (std::abs(current\_linear\_velocity) < 0.01) {

return;

}

update\_robot\_pose((ros::Time::now() - robot\_time).toSec() );

double lookahead\_distance = 3.0;

target\_point\_index = get\_target\_path\_pose\_index(target\_point\_index, lookahead\_distance);

const auto& target\_pose = path.poses[target\_point\_index].pose;

double x = target\_pose.position.x - robot\_x;

double y = target\_pose.position.y - robot\_y;

double error = -x\*sin(robot\_theta) + y\*cos(robot\_theta);

double curvature = 2.0\*error/(lookahead\_distance\*lookahead\_distance);

std\_msgs::Float32 cmd;

cmd.data = clip<double>(curvature, max\_curvature);

steer\_pub.publish(cmd);

//send trajectory for velocity controller

publish\_trajectory();

//send error for debug proposes

publish\_error(error);

ROS\_DEBUG\_STREAM("steering cmd = " << curvature);

}

void Controller::on\_pose(const nav\_msgs::OdometryConstPtr& odom)

{

// здесь обновляются текущие координаты автомобиля по данным одометрии

robot\_x = odom->pose.pose.position.x;

robot\_y = odom->pose.pose.position.y;

robot\_theta = 2\*atan2(odom->pose.pose.orientation.z,

odom->pose.pose.orientation.w);

world\_frame\_id = odom->header.frame\_id;

robot\_time = odom->header.stamp;

}

void Controller::on\_odo(const nav\_msgs::OdometryConstPtr& odom)

{

// обновляет линейную и угловую скорость по данным одометрии

current\_linear\_velocity = odom->twist.twist.linear.x;

current\_angular\_velocity = odom->twist.twist.angular.z;

//ROS\_DEBUG\_STREAM("odom vel = "<<current\_velocity);

}

void Controller::publish\_error(double error)

{

// публикация ошибки пройденного пути

std\_msgs::Float32 err\_msg;

err\_msg.data = error;

err\_pub.publish(err\_msg);

}

double Controller::cross\_track\_error()

{

// Вычисление ошибки отклонения автомобиля от траектории

double error = 0.0;

if (robot\_y < radius)

{

double rx = robot\_x;

double ry = robot\_y - radius;

error = sqrt(rx\*rx + ry\*ry) - radius;

}

else if ( robot\_y > cy)

{

double rx = robot\_x;

double ry = robot\_y - cy;

error = sqrt(rx\*rx + ry\*ry) - radius;

}

else if ( robot\_x > 0 )

{

error = robot\_x - radius;

}

else if ( robot\_x < 0 )

{

error = -radius - robot\_x;

}

return error;

}

void Controller::publish\_trajectory()

{

// публикация траектоии

// ROS\_DEBUG\_STREAM("publish trajectory");

path\_pub.publish(path);

}

void Controller::reset()

{

// сброс значений ошибок ПИД-регулятора

error\_integral = 0.0;

last\_error = cross\_track\_error();

}

void Controller::reset(double p, double d, double i)

{

// перегрузка метода. Здесь, и сбрасываются ошибки, и уже непостредственно задаются значения ПИД-регулятора

reset();

p\_factor = p;

d\_factor = d;

i\_factor = i;

}

nav\_msgs::Path Controller::create\_path() const {

// Здесь происходит создание и публикация пути на основе текущей траектории автомобиля

//prepare path message from trajectory

nav\_msgs::Path path;

path.header.frame\_id = "odom";

path.header.stamp = robot\_time;

auto segment\_it = trajectory.begin();

double previous\_segment\_left = 0.0;

std::size\_t points\_added = 0;

double point\_length = 0.0;

while (segment\_it != trajectory.end()) {

const auto segment = \*segment\_it;

double segment\_length = segment->get\_length();

//add points from the segment

while (point\_length <= segment\_length) {

const auto point = segment->get\_point(point\_length);

const auto angle = segment->get\_orientation(point\_length);

geometry\_msgs::PoseStamped pose;

pose.header.frame\_id = "odom";

pose.pose.position.x = point.x();

pose.pose.position.y = point.y();

pose.pose.orientation = tf::createQuaternionMsgFromYaw(angle);

path.poses.push\_back(pose);

point\_length += traj\_dl;

points\_added++;

}

point\_length -= segment\_length;

++segment\_it;

}

return path;

}

/\*!

\* \brief constructor

\* loads parameters from ns

\* proportional, differential , integral - pid factors

\* max\_antiwindup\_error - max error for using integral component

\* trajectory consists of two circle segments connected with two lines

\* first circle center is (0, radius), second circle center is (0, cy)

\* radius - radius of circular parts

\* cy - center of second circle

\* traj\_dl - discrete of published trajectory

\* traj\_length - length of published trajectory

\* timer\_period - timer discrete

\*/

Controller::Controller(const std::string& ns):

/\* Это конструктор.

Сначала задаёся просранство имён (здесь simple\_controller), где контроллер работает

Инициализируются параметры ПИД-регулятора и его ошибки

Инициализируются все другие поля класса см. controller.h

Инициализируется начальное положение автомобиля

Создаётся траектория автомобиля из нескольких сегментов (прямых и круглых)

Инициализируется указатель на текущий сегмент траектории

Вызывается метод для создания сообщения о траектории

Вызывается метод для инициализации текущего пути

\*/

nh("~/" + ns),

p\_factor(nh.param("proportional", 1.0)),

d\_factor(nh.param("differential", 0.0)),

i\_factor(nh.param("integral", 0.0)),

max\_antiwindup\_error( nh.param("max\_antiwindup\_error", 0.5) ),

error\_integral(0.0),

last\_error(0.0),

radius(nh.param("radius", 10.0)),

cy(nh.param("cy", 2\*radius)), //default circle is in center (0,radius)

max\_curvature(nh.param("max\_curvature", 0.2 )),

traj\_dl(nh.param("traj\_dl", 0.2)),

traj\_length(nh.param("traj\_length", 5.0)),

pose\_sub(nh.subscribe("ground\_truth", 1, &Controller::on\_pose, this)),

odo\_sub(nh.subscribe("odom", 1, &Controller::on\_odo, this)),

path\_sub(nh.subscribe("path", 1, &Controller::on\_path, this)),

timer( nh.createTimer( ros::Duration(nh.param("timer\_period", 0.1)), &Controller::on\_timer, this ) ),

err\_pub(nh.advertise<std\_msgs::Float32>("error", 10) ),

steer\_pub(nh.advertise<std\_msgs::Float32>("/steering", 10)),

path\_pub(nh.advertise<nav\_msgs::Path>("controller\_path", 1))

{

//counter clock

trajectory.emplace\_back( std::make\_shared<trajectory::CircularSegment>( 1.0 / radius, 0, 0, 1.0, 0, M\_PI/2\*radius) );

trajectory.emplace\_back( std::make\_shared<trajectory::LinearSegment> ( radius, radius, 0.0, 1.0, cy - radius) );

trajectory.emplace\_back( std::make\_shared<trajectory::CircularSegment>( 1.0 / radius, radius, cy, 0.0, 1.0, M\_PI/2\*radius ) );

trajectory.emplace\_back( std::make\_shared<trajectory::CircularSegment>( 1.0 / radius, 0, radius + cy, -1.0, 0.0, M\_PI/2\*radius ) );

trajectory.emplace\_back( std::make\_shared<trajectory::LinearSegment> ( -radius, cy, 0.0, -1.0, cy - radius) );

trajectory.emplace\_back( std::make\_shared<trajectory::CircularSegment>( 1.0/ radius, -radius, radius, 0.0, -1.0, M\_PI/2\*radius) );

//clock wise track

// trajectory.emplace\_back( std::make\_shared<trajectory::CircularTrajectory>( -1.0 / radius, 0, 0, 1.0, 0, M\_PI/2\*radius) );

// trajectory.emplace\_back( std::make\_shared<trajectory::LinearTrajectory> ( radius, -radius, 0.0, -1.0, cy - radius) );

// trajectory.emplace\_back( std::make\_shared<trajectory::CircularTrajectory>( -1.0 / radius, radius, -cy, 0.0, -1.0, M\_PI/2\*radius ) );

// trajectory.emplace\_back( std::make\_shared<trajectory::CircularTrajectory>( -1.0 / radius, 0, -radius - cy, -1.0, 0.0, M\_PI/2\*radius ) );

// trajectory.emplace\_back( std::make\_shared<trajectory::LinearTrajectory> ( -radius, -cy, 0.0, 1.0, cy - radius) );

// trajectory.emplace\_back( std::make\_shared<trajectory::CircularTrajectory>( -1.0/ radius, -radius, -radius, 0.0, 1.0, M\_PI/2\*radius) );

current\_segment = trajectory.begin();

const auto trajectory\_path = create\_path();

on\_path(trajectory\_path);

}

Controller::~Controller()

{

// Это деструктор

// Здесь пусто...

// TODO Auto-generated destructor stub

}

} /\* namespace simple\_controller \*/

**ПРИЛОЖЕНИЕ Б**

Код файла сontroller.h.

/\*

\* Controller.h

\*

\* Created on: 30 апр. 2017 г.

\* Author: aminin

\*/

#ifndef SRC\_CONTROLLER\_H\_

#define SRC\_CONTROLLER\_H\_

#include <ros/ros.h>

#include <nav\_msgs/Odometry.h>

#include <nav\_msgs/Path.h>

#include <std\_msgs/Float32.h>

#include <list>

#include <memory>

#include "trajectory\_segment.h"

namespace simple\_controller

{

using TrajPtr = std::shared\_ptr<trajectory::TrajectorySegment>;

/\*!

\*\brief robot controller

\* controls following along defined trajectory via simple pid regulator

\* angular\_velocity = pid(error)

\* error is distance to trajectory

\* trajectory is list of angular and linear segments, saved as pointers to base class Trajectory

\* Trajectory is cycled

\* feedback from robot is received via ground\_truth callback (real position of robot)

\* during control future trajectory is published for velocity controller

\*/

class Controller

{

/\*класс Controller нужен для управления движением автомобиля вдоль

заданной траектории с использованием ПИД-регулятора\*/

protected:

ros::NodeHandle nh; // Это объект класса ros::NodeHandle. Через него идёт взаимодействие с ROS

double robot\_x = 0.0; //<---текущие координаты и угол положения автомобиля

double robot\_y = 0.0; //<-|

double robot\_theta = 0.0;//<-/

//time of robot coordinates update

ros::Time robot\_time; // Это судя по всему время последнего обновления координат и угла автомобиля

double p\_factor; //<---Это кооэффициенты ПИД-регулятора (Пропорциональности)

double d\_factor; //<-| (Дифференцирования)

double i\_factor; //<-/ (Интегрирования)

double max\_antiwindup\_error; /\* это нужно, чтобы задать максимальное значение ошибки, при котором будет

использоваться коэффициент интегрирования регулятора\*/

double error\_integral; // Это собственно само значение ошибки коэффициента интегрирования

double last\_error; // Это вроде значение, которое показывает предыдущее значение ошибки коэффициента дифференцирования

///\ circle params

double radius; // это радиус окружности траектории

///\ second circle center

double cy; // Это координата центра окружности траектории

double max\_curvature; // Максимальное значение кривизны траектории

double current\_linear\_velocity = 0.0; // Текущие линейная

double current\_angular\_velocity = 0.0; // и угловая скорости автомобиля

//discrete of publish trajectory

double traj\_dl; // Это расстояние между двумя точками траектории

//length of published trajectory

double traj\_length; // Длина траектории

using Trajectory = std::list<TrajPtr>;

/// \ container of trajectory segments

std::list<TrajPtr> trajectory; // Это список с сегментами траектории автомобиля. Эти сегменты могут быть прямыми и круглыми.

nav\_msgs::Path path; // Это сам путь движения автомобиля

std::size\_t nearest\_point\_index; // Ближайшая точка пути

std::size\_t target\_point\_index; // Целевая точка пути

/// \ current segment

std::list<TrajPtr>::iterator current\_segment; // Это указатель на текущий сегмент траектории

/// \ length of the current segment at the current point

double current\_segment\_length = 0.0; // Длина текущего сегмента траектории на текущей позиции автомобиля

ros::Subscriber pose\_sub; // Субскрайбер на положение

ros::Subscriber odo\_sub; // Субскрайбер на одометрию

ros::Subscriber path\_sub; // Субскрайбер на путь

ros::Timer timer; // Таймер который вызыватся через определённый промежуто времени (0.1 секунда здесь) для подруливания автомобиля

ros::Publisher err\_pub; // Паблишер для ошибки

ros::Publisher steer\_pub; // Паблишер для угла поворота руля

ros::Publisher path\_pub; // Паблишер для пути

/// \ frame\_id for coordinates of controller

std::string world\_frame\_id; // Идентификатор фрейма, используемый для... идентификации... фрейма...

// далее идут методы класса, расписанные в controller.cpp

double get\_pid\_control(double error);

std::size\_t get\_target\_path\_pose\_index(int old\_target\_index, double ld);

void on\_timer(const ros::TimerEvent& event);

void on\_pose(const nav\_msgs::OdometryConstPtr& odom);

void on\_path(const nav\_msgs::Path& path);

/\*

\*@brief calculates feedback error for trajectory

\*@return feedback error

\*/

double cross\_track\_error();

/// \ update robot pose to current time based on last pose and velocities

void update\_robot\_pose(double dt);

/\*

\* \brief publishes trajectory as pointcloud message

\*/

void publish\_trajectory();

void on\_odo(const nav\_msgs::OdometryConstPtr& odom);

void publish\_error(double error);

nav\_msgs::Path create\_path() const;

std::size\_t get\_nearest\_path\_pose\_index(int start\_index,

std::size\_t search\_len);

public:

double get\_p\_factor(){ return p\_factor; }

double get\_d\_factor(){ return d\_factor; }

double get\_i\_factor(){ return i\_factor; }

void reset();

void reset(double p, double d, double i );

Controller(const std::string& ns = "simple\_controller");

virtual ~Controller();

};

} /\* namespace simple\_controller \*/

#endif /\* SRC\_CONTROLLER\_H\_ \*/