Семинар 5

Вассуф Язан

Москва 2024

Скачать

- Yazan-pyth/MobileRobots (github.com)

Возможные ошибка

- Acado: https://acado.github.io/install_linux.html // sudo make install
- *.perspective:

"repr(QByteArray.hex)":"QtCore.QByteArray('**********')",

на

"repr(QByteArray.hex)": "b'**********'",

- C++11->C++17

Задача 1

(x_{ld}, y_{ld}) - точка предварительного прицеливания;

(x_r, y_r) - точка текущего положения;

l_d - длина предварительного прицеливания ;

к - кривизна рулевого управления;

R - радиус рулевого управления;

δ - Угол поворота колеса;

α - угол между точкой предварительного прицеливания и текущего направлением.

По теореме синуса:

$$\frac{l_d}{sin(2*\alpha)} = \frac{R}{sin(\frac{\pi}{2}-\alpha)}$$

$$R = \frac{l_d}{2sin(\alpha)}$$

Получаем кривизну рулевого управления:

$$\kappa = \frac{2sin(\alpha)}{l_d} = \frac{2*e_y}{{l_d}^2}$$

Где ey - погрешность точки предварительного прицеливания в горизонтальном направлении тележки.

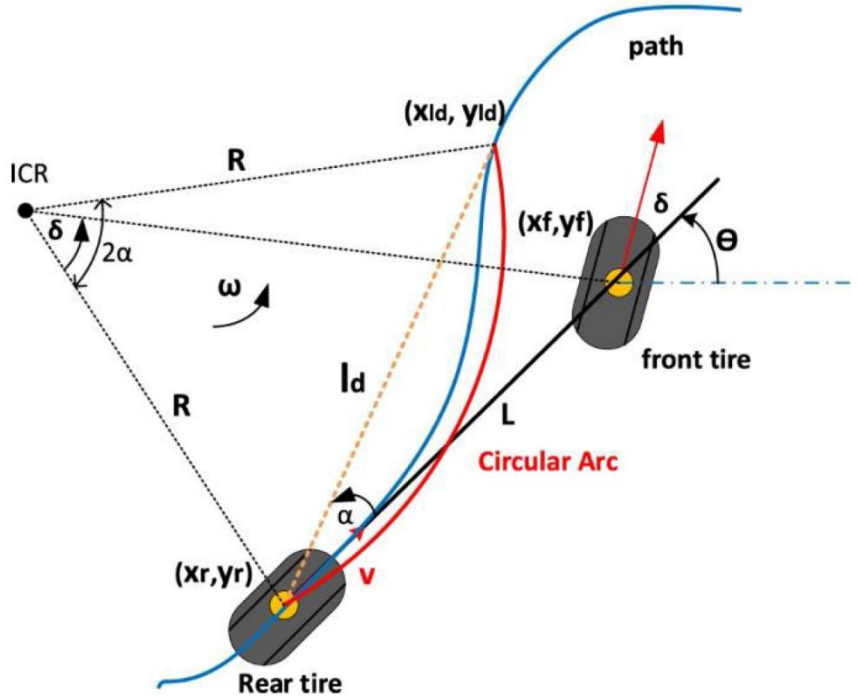


Рис. 3. Управление Pure Pursuite

ОУ

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).

Данные управления

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

Управление

- Управление роботом в этой задаче сводится к управлению поворотом переднего колеса велосипедной модели (топик /steering куда 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).

Основные функции в коде

- 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).

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.

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`.

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 /steering.

– Публикуем траекторию.

– Публикуем текущую ошибку.

Метод Controller::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 робота:

```
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.

```
void Controller::on_timer(const ros::TimerEvent& event)
{
  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);
```

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;
}
```

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 робота:

```
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;
}
```

Решение 1

- Для решения задачи в функции on_timer прописываем поворот рулевого колеса через
- выражение для алгоритма следования за маяком:
  - double lookahead_distance = 3.0;
  - double curvature = 2.0*error/(lookahead_distance*lookahead_distance);
  - Далее полученное значение публикуется в топик /steering.

Решение 2

1. Добавить переменные-члены в файл controller.h:

```
double lam = 0.1;
 double c = 1;
```

где lam - коэффициент дальности прогноза, а c - дальность прогноза.

2. Добавить функции-члены в файл controller.h:

```
std::size_t cal_target_index();
```

- Вычислить перспективную целевую точку для текущего положения автомобиля на опорной трассе.

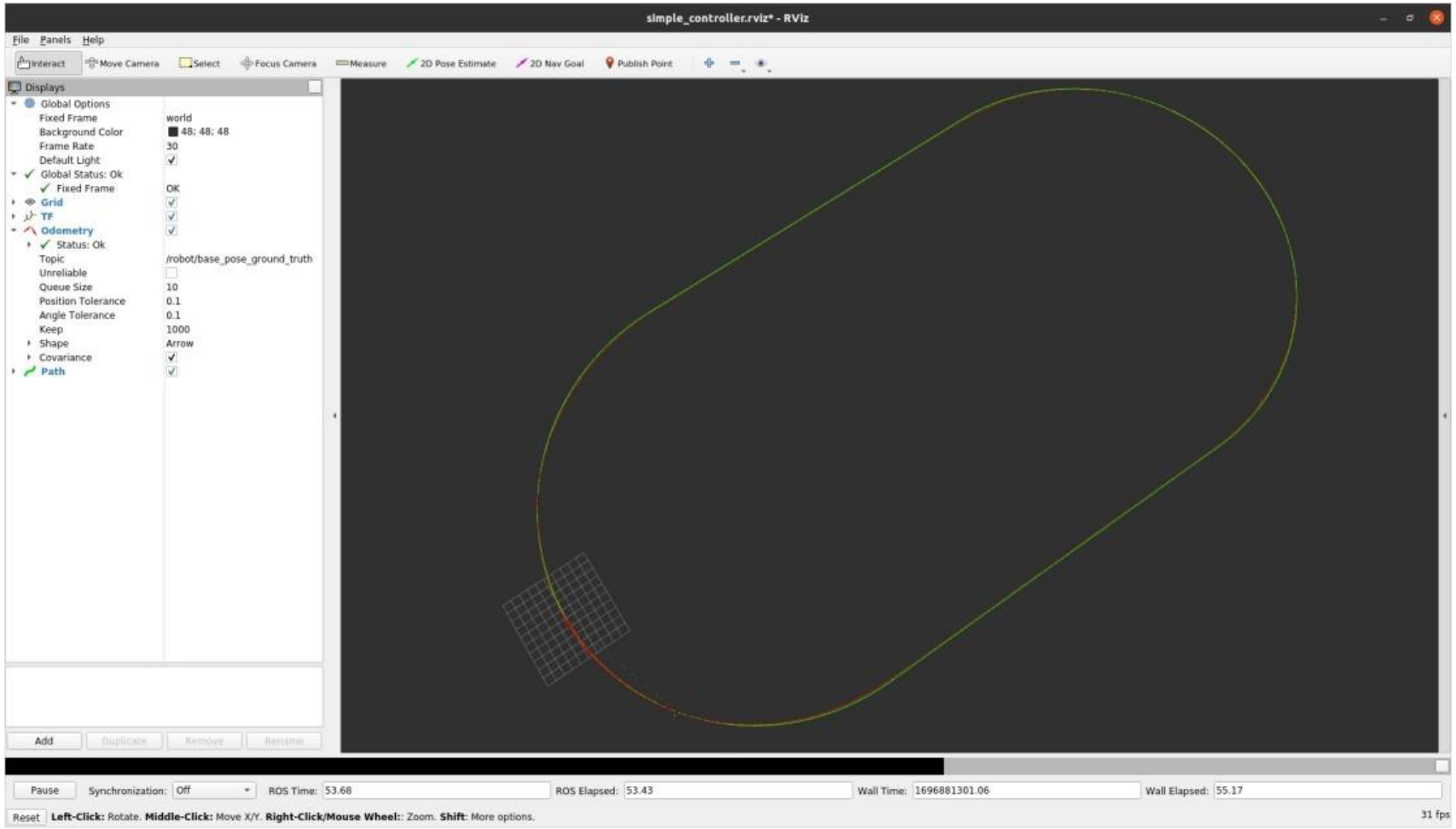
cal_target_index()

```
// Определение опорных точек траектории
std::size_t Controller::cal_target_index()
{
  update_robot_pose((ros::Time::now() - robot_time).toSec() );
  nearest_point_index = get_nearest_path_pose_index(nearest_point_index - 10, 20);

  const auto& nearest_pose = path.poses[nearest_point_index].pose;
  const auto& nearest_pose_angle = tf::getYaw(nearest_pose.orientation);

  double dx = robot_x - nearest_pose.position.x;
  double dy = robot_y - nearest_pose.position.y;
  // error is negative difference by y axe in the axis of the nereset pose
  double error = -(-dx * sin(nearest_pose_angle) + dy * cos(nearest_pose_angle));
  ROS_INFO("error %lf", error);

  // error
  double m_error = abs(error);
  // Printing results
  ROS_INFO("current_linear_velocity %lf", current_linear_velocity);

  double l_d = lam*current_linear_velocity + c;
  while (l_d > m_error && (nearest_point_index + 1) < static_cast<int>(path.poses.size()))
  {
    const auto& nearest_pose_plus1 = path.poses[nearest_point_index + 1].pose;
    const auto& nearest_pose_angle_plus1 = tf::getYaw(nearest_pose_plus1.orientation);
    double dx_plus1 = robot_x - nearest_pose_plus1.position.x;
    double dy_plus1 = robot_y - nearest_pose_plus1.position.y;
    double m_error_plus1 = abs(-dx_plus1 * sin(nearest_pose_angle_plus1) + dy_plus1 * cos(nearest_pose_angle_plus1));
    m_error = m_error_plus1;
    nearest_point_index += 1;
  }
  return nearest_point_index;
}
```

on_timer

```
void Controller::on_timer(const ros::TimerEvent& event)
{
  if (std::abs(current_linear_velocity) < 0.01) {
    return;
  }
  update_robot_pose((ros::Time::now() - robot_time).toSec() );

  // nearest_point_index = get_nearest_path_pose_index(nearest_point_index - 10, 20);
  nearest_point_index = cal_target_index();
  const auto& nearest_pose = path.poses[nearest_point_index].pose;
  const auto& nearest_pose_angle = tf::getYaw(nearest_pose.orientation);

  double dx = robot_x - nearest_pose.position.x;
  double dy = robot_y - nearest_pose.position.y;
  // error is negative difference by y axe in the axis of the nereset pose
  double error = -(-dx * sin(nearest_pose_angle) + dy * cos(nearest_pose_angle));
  ROS_INFO("r_error_plus %lf", error);

  // Ld
  double l_d = lam*current_linear_velocity + c;

  // Calculating  alpha
  double base2nearest_pose_angle = (nearest_pose_angle - robot_theta);

  // Control
  double angular_m_cmd = atan2(2*1.88*sin(base2nearest_pose_angle), l_d);

  ROS_INFO("Control angle %lf", angular_m_cmd);

  double curvature = angular_m_cmd / current_linear_velocity;

  //send curvature as command to drives
  std_msgs::Float32 cmd;
  cmd.data = clip<double>(curvature, max_curvature);
  // cmd.data = angular_m_cmd;
  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);
}
```

Results


simple_controller.rviz* - RViz
File Panels Help
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Global Options
Fixed Frame world
Background Color 48; 48; 48
Frame Rate 30
Default Light
Global Status: Ok
Fixed Frame OK
Grid
TF
Odometry
Status: Ok
Topic /robot/base_pose_ground_truth
Unreliable
Queue Size 10
Position Tolerance 0.1
Angle Tolerance 0.1
Keep 1000
Shape Arrow
Covariance
Path
Add
Duplicate
Remove
Rename
Pause
Synchronization: Off
ROS Time: 53.68
ROS Elapsed: 53.43
Wall Time: 1696881301.06
Wall Elapsed: 55.17
Reset
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.
31 fps

Задача

- Переделать задачу под ROS2. Срок до 05.11 в 17:25.