**Simple kinematics for mobile robot**

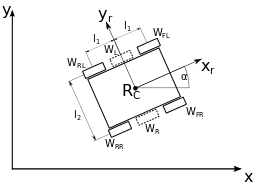
You can run this tutorial on:

* [ROSbot 2.0](https://store.husarion.com/products/rosbot)
* [ROSbot 2.0 PRO](https://store.husarion.com/collections/dev-kits/products/rosbot-pro)
* [ROSbot 2.0 simulation model (Gazebo)](https://github.com/husarion/rosbot_description)

**A little bit of theory**

**Introduction**

The purpose of forward kinematics in mobile robotics is to determine robot position and orientation based on wheels rotation measurements. To achieve that we'll create robot kinematic model. ROSbot is four wheeled mobile robot with separate drive for each wheel, but in order to simplify kinematic calculation we will treat it as two wheeled. Two virtual wheels (marked as WL and WR on the scheme) will have axis going through robot geometric center. This way we can use simpler kinematic model of differential wheeled robot. The name "differential" comes from the fact that robot can change its direction by varying the relative rate of rotation of its wheels and does not require additional steering motion. Robot scheme is presented below:



Description:

* Rc - robot geometric centre
* xc - robot geometric centre x position
* yc - robot geometric centre y position
* xr - robot local x axis that determines front of the robot
* α - robot angular position
* WFL - front left wheel
* WFR - front right wheel
* WRL - rear left wheel
* WRR - rear right wheel
* WL - virtual left wheel
* WR - virtual right wheel
* l1 - distance between robot centre and front/rear wheels
* l2 - distance between robot left and right wheels

Our mobile robot has constraints. It can only move in x-y plane and it has 3 DOF (degrees of freedom). However not all of DOFs are controllable which means robot cannot move in every direction of its local axes (e.g. it cannot move sideways). Such drive system is called **non-holonomic**. When amount of controllable DOFs is equal to total DOFs then a robot can be called **holonomic**. To achieve that some mobile robots are built using Omni or Mecanum wheels and thanks to vectoring movement they can change position without changing their heading (orientation).

**Forward Kinematics task**

The robot position is determined by a tuple (xc, yc, α). The forward kinematic task is to find new robot position (xc, yc, α)' after time *δt* for given control parameters:

* vR - linear speed of right virtual wheel
* vL - linear speed of left virtual wheel

In our case the angular speed ω and the angular position Φ of each virtual wheel will be an average of its real counterparts:









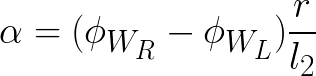
Linear speed of each virtual wheel:

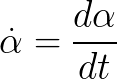
https://husarion.com/docs/assets/img/ros/man_3_formula_2_1.png

https://husarion.com/docs/assets/img/ros/man_3_formula_2_2.png

where *r* - the wheel radius.

We can determine robot angular position and speed with:



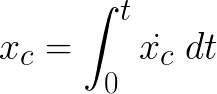


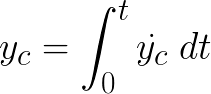
Then robot speed x and y component:

https://husarion.com/docs/assets/img/ros/man_3_formula_4_1.png

https://husarion.com/docs/assets/img/ros/man_3_formula_4_2.png

To get position:





We assume starting position as (0,0).

In order for code to work correctly wheels should be connected to ports in following manner:

* front left wheel (WFL) - hMot4
* front right wheel (WFR) - hMot1
* rear left wheel (WRL) - hMot3
* rear right wheel (WRR) - hMot2

The implementation of the equations above in hFramework can be found [here](https://github.com/husarion/hFramework/blob/master/src/rosbot/ROSbot.cpp#L126).

**Controlling the motor**

Most common way to send movement commands to the robot is with use of geometry\_msgs/Twist message type. Then motor driver node should use data stored in them to control the motor.

The geometry\_msgs/Twist message express velocity in free space and consists of two fields:

* Vector3 linear - represents linear part of velocity [m/s]
* Vector3 angular - represents angular part of velocity [rad/s]

You will control ROSbot in the x-y plane by manipulating the x component of linear speed vector and the z component of angular speed vector.

**Publishing the motion command for robot**

You will use keyboard to control the movement of your robot. For getting the key events and converting them to geometry\_msgs/Twist messages you can use teleop\_twist\_keyboard.py node from package[teleop\_twist\_keyboard](http://wiki.ros.org/teleop_twist_keyboard).

Alternatively you can use joystick to control your robot, then you will need joy\_node node from [joy](http://wiki.ros.org/joy) package and teleop\_node node from [teleop\_twist\_joy](http://wiki.ros.org/teleop_twist_joy) package.

**Converting motion command to motor drive signal**

In this section you will create a node for interfacing motors. Your node will subscribe to topic with geometry\_msgs/Twist messages, drive the motors, read encoders and publish their state to appropriate topic. To create this node you will use Husarion Cloud. Create new project and paste following:

#**include** "hFramework.h"

#**include** "hCloudClient.h"

#**include** "ros.h"

#**include** "geometry\_msgs/Twist.h"

#**include** "sensor\_msgs/BatteryState.h"

#**include** "std\_msgs/Bool.h"

#**include** "ROSbot.h"

**using** **namespace** hFramework;

// Uncomment one of these lines, accordingly to range sensor type of your ROSbot

// If you have version with infrared sensor:

// static const SensorType sensor\_type = SENSOR\_INFRARED;

// If you have version with laser sensor:

**static** **const** SensorType sensor\_type = SENSOR\_LASER;

// If you want to use your own sensor:

// static const SensorType sensor\_type = NO\_DISTANCE\_SENSOR;

// Uncomment one of these lines, accordingly to IMU sensor type of your device

// If you have version with MPU9250:

**static** **const** ImuType imu\_type = MPU9250;

// If you want to use your own sensor:

// static const ImuType imu\_type = NO\_IMU;

// Uncomment one of these lines, accordingly version of your device

**uint32\_t** baudrate = 500000; // for ROSbot 2.0

// uint32\_t baudrate = 230400; // for ROSbot 2.0 PRO

ros::NodeHandle nh;

sensor\_msgs::BatteryState battery;

ros::Publisher \*battery\_pub;

**int** publish\_counter = 0;

**void** **twistCallback**(**const** geometry\_msgs::Twist &twist)

{

rosbot.setSpeed(twist.linear.x, twist.angular.z);

}

**void** **initCmdVelSubscriber**()

{

ros::Subscriber<geometry\_msgs::Twist> \*cmd\_sub = **new** ros::Subscriber<geometry\_msgs::Twist>("/cmd\_vel", &twistCallback);

nh.subscribe(\*cmd\_sub);

}

**void** **resetCallback**(**const** std\_msgs::Bool &msg)

{

**if** (msg.data == true)

{

rosbot.reset\_odometry();

}

}

**void** **initResetOdomSubscriber**()

{

ros::Subscriber<std\_msgs::Bool> \*odom\_reset\_sub = **new** ros::Subscriber<std\_msgs::Bool>("/reset\_odom", &resetCallback);

nh.subscribe(\*odom\_reset\_sub);

}

**void** **initBatteryPublisher**()

{

battery\_pub = **new** ros::Publisher("/battery", &battery);

nh.advertise(\*battery\_pub);

}

**void** **hMain**()

{

rosbot.initROSbot(sensor\_type, imu\_type);

RPi.init(baudrate);

platform.begin(&RPi);

nh.getHardware()->initWithDevice(&platform.LocalSerial);

nh.initNode();

initBatteryPublisher();

initCmdVelSubscriber();

initResetOdomSubscriber();

**while** (true)

{

nh.spinOnce();

publish\_counter++;

**if** (publish\_counter > 10)

{

// get battery voltage

battery.voltage = rosbot.getBatteryLevel();

// publish battery voltage

battery\_pub->publish(&battery);

publish\_counter = 0;

}

sys.delay(10);

}

}

**Copy**

Below is explanation for code line by line.

Include required headers:

#**include** "hFramework.h"

#**include** "hCloudClient.h"

#**include** "ros.h"

#**include** "geometry\_msgs/Twist.h"

#**include** "sensor\_msgs/BatteryState.h"

#**include** "std\_msgs/Bool.h"

#**include** "ROSbot.h"

**Copy**

Load namespace for Husarion functions:

**using** **namespace** hFramework;

**Copy**

Define which type of distance sensor you are using in your robot:

// Uncomment one of these lines, accordingly to range sensor type of your ROSbot

// If you have version with infrared sensor:

// static const SensorType sensor\_type = SENSOR\_INFRARED;

// If you have version with laser sensor:

**static** **const** SensorType sensor\_type = SENSOR\_LASER;

// If you want to use your own sensor:

// static const SensorType sensor\_type = NO\_DISTANCE\_SENSOR;

**Copy**

Define which type of IMU you are using in your robot:

// Uncomment one of these lines, accordingly to IMU sensor type of your device

// If you have version with MPU9250:

**static** **const** ImuType imu\_type = MPU9250;

// If you want to use your own sensor:

// static const ImuType imu\_type = NO\_IMU;

**Copy**

Define baudrate for serial communication, ROSbot 2.0 and ROSbot 2.0 PRO needs different serial port speed:

// Uncomment one of these lines, accordingly version of your device

**uint32\_t** baudrate = 500000; // for ROSbot 2.0

// uint32\_t baudrate = 230400; // for ROSbot 2.0 PRO

**Copy**

Create handle for node:

ros::NodeHandle nh;

**Copy**

Define type of message and publisher for a battery:

sensor\_msgs::BatteryState battery;

ros::Publisher \*battery\_pub;

**Copy**

Function for handling incoming messages:

**void** **twistCallback**(**const** geometry\_msgs::Twist &twist)

{

rosbot.setSpeed(twist.linear.x, twist.angular.z);

}

**Copy**

Function for initialization of velocity command subscriber:

**void** **initCmdVelSubscriber**()

{

ros::Subscriber<geometry\_msgs::Twist> \*cmd\_sub = **new** ros::Subscriber<geometry\_msgs::Twist>("/cmd\_vel", &twistCallback);

nh.subscribe(\*cmd\_sub);

}

**Copy**

Function for initialization of battery state publisher:

**void** **initBatteryPublisher**()

{

battery\_pub = **new** ros::Publisher("/battery", &battery);

nh.advertise(\*battery\_pub);

}

**Copy**

Function for handling incoming requests of robot odometry reset:

**void** **resetCallback**(**const** std\_msgs::Bool &msg)

{

**if** (msg.data == true)

{

rosbot.reset\_odometry();

}

}

**Copy**

Function for initialization of odometry reset requests subscriber:

**void** **initResetOdomSubscriber**()

{

ros::Subscriber<std\_msgs::Bool> \*odom\_reset\_sub = **new** ros::Subscriber<std\_msgs::Bool>("/reset\_odom", &resetCallback);

nh.subscribe(\*odom\_reset\_sub);

}

**Copy**

Main function, device and messages initialization:

**void** **hMain**()

{

rosbot.initROSbot(sensor\_type, imu\_type);

RPi.init(baudrate);

platform.begin(&RPi);

nh.getHardware()->initWithDevice(&platform.LocalSerial);

nh.initNode();

initBatteryPublisher();

initCmdVelSubscriber();

initResetOdomSubscriber();

**Copy**

Infinite loop, waiting for incoming messages:

**while** (true)

{

nh.spinOnce();

publish\_counter++;

**if** (publish\_counter > 10)

{

// get battery voltage

battery.voltage = rosbot.getBatteryLevel();

// publish battery voltage

battery\_pub->publish(&battery);

publish\_counter = 0;

}

sys.delay(10);

}

**Copy**

Build your project and upload it to device.

**Running motor controller step by step**

In this section you will learn how to control your robot movement with keyboard. You will need teleop\_twist\_keyboard node from teleop\_twist\_keyboard package.

Log in to your CORE2 device through remote desktop and run terminal. In first terminal window run $ roscore, in second run:

/opt/husarion/tools/rpi-linux/ros-core2-client /dev/ttyCORE2

**Copy**

This program is responsible for bridging your CORE2 to ROS network. When you are working with simulator, then above bridge is not necessary. Gazebo will subscribe appropriate topics automatically. In third terminal window run:

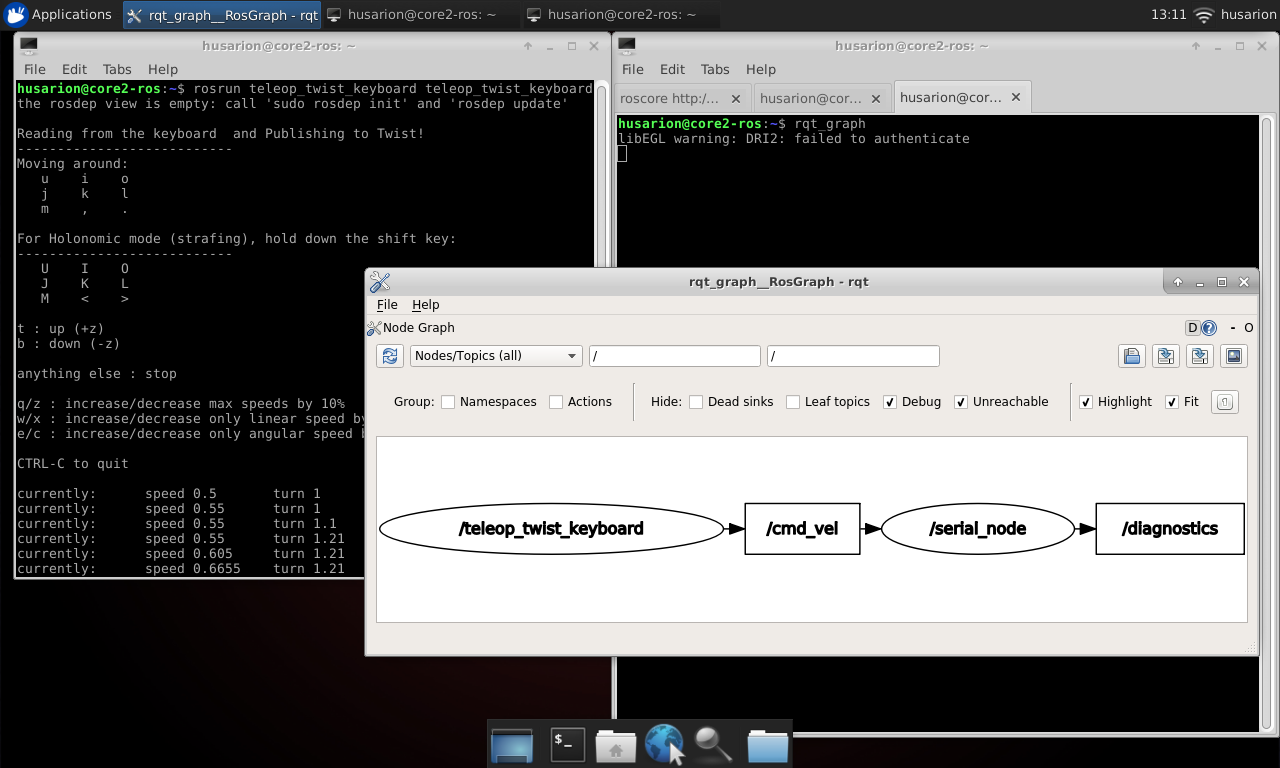
rosrun teleop\_twist\_keyboard teleop\_twist\_keyboard.py

**Copy**

Now you can control your robot with keyboard with following functions for buttons:

* ’i’ - move forward
* ’,’ - move backward
* ’j’ - turn left
* ’l’ - turn right
* ’k’ - stop
* ’q’ - increase speed
* ’z’ - decrease speed

You should get similar view in rqt\_graph:



**Running motor controller with roslaunch**

To enable control of ROSbot with single launch file we need to prepare script which calls ros-core2-client. In tutorial\_pkg direcotry create scripts sudirectory:

cd ~/ros\_workspace/src/tutorial\_pkg

mkdir scripts

**Copy**

Inside the scripts directory create file serial\_bridge.sh and set it to executable:

cd ~/ros\_workspace/src/tutorial\_pkg/scripts

touch serial\_bridge.sh

chmod a+x serial\_bridge.sh

**Copy**

Open the file and paste into it:

#!/bin/bash

/opt/husarion/tools/rpi-linux/ros-core2-client /dev/ttyCORE2

**Copy**

Inside the ~/ros\_workspace/src/tutorial\_pkg/launch create tutorial\_3.launch with below content:

<**launch**>

<**arg** name="use\_rosbot" default="true"/>

<**arg** name="use\_gazebo" default="false"/>

<**include** if="$(arg use\_gazebo)" file="$(find rosbot\_gazebo)/launch/rosbot.launch"/>

<**node** name="teleop\_twist\_keyboard" pkg="teleop\_twist\_keyboard" type="teleop\_twist\_keyboard.py" output="screen"/>

</**launch**>

**Copy**

We do not need to create separate launch files for serial\_bridge.sh or teleop\_twist\_keyboard as these nodes can be configured with only one line.

**Determining robot position**

This section is required only for ROSbot. Gazebo has already implemented it's own plugin to publish robot position. Now we will perform forward kinematics task- we will use encoders that are attached to every motor and process their measurements with equations shown in section **Forward kinematics task**.

Open Husarion WebIDE and open project that you created in section **Converting motion command to motor drive signal**.

Add header file:

#**include** "geometry\_msgs/PoseStamped.h"

#**include** "tf/tf.h"

#**include** "tf/transform\_broadcaster.h"

#**include** "sensor\_msgs/JointState.h"

**Copy**

Define message type and publisher for robot position:

geometry\_msgs::PoseStamped pose;

ros::Publisher \*pose\_pub;

**Copy**

Define message type and publisher for /tf frame:

geometry\_msgs::TransformStamped robot\_tf;

tf::TransformBroadcaster broadcaster;

**Copy**

Define message type and publisher for wheel angular position:

sensor\_msgs::JointState joint\_states;

ros::Publisher \*joint\_state\_pub;

**char** \*name[] = {"front\_left\_wheel\_hinge", "front\_right\_wheel\_hinge", "rear\_left\_wheel\_hinge", "rear\_right\_wheel\_hinge"};

**float** pos[] = {0, 0, 0, 0};

**float** vel[] = {0, 0, 0, 0};

**float** eff[] = {0, 0, 0, 0};

**Copy**

Create a data structure for robot pose:

std::vector<**float**> rosbot\_pose;

**Copy**

Create data structure for wheel positions:

wheelsState ws;

**Copy**

Initialization of robot position publisher as geometry\_msgs::PoseStamped message type:

**void** **initPosePublisher**()

{

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

pose.pose.orientation = tf::createQuaternionFromYaw(0);

pose\_pub = **new** ros::Publisher("/pose", &pose);

nh.advertise(\*pose\_pub);

}

**Copy**

Initialization of robot position publisher in /tf tree:

**void** **initTfPublisher**()

{

robot\_tf.header.frame\_id = "odom";

robot\_tf.child\_frame\_id = "base\_link";

robot\_tf.transform.translation.x = 0.0;

robot\_tf.transform.translation.y = 0.0;

robot\_tf.transform.translation.z = 0.0;

robot\_tf.transform.rotation.x = 0.0;

robot\_tf.transform.rotation.y = 0.0;

robot\_tf.transform.rotation.z = 0.0;

robot\_tf.transform.rotation.w = 1.0;

broadcaster.init(nh);

}

**Copy**

Initialization of wheel angular position publisher:

**void** **initJointStatePublisher**()

{

joint\_state\_pub = **new** ros::Publisher("/joint\_states", &joint\_states);

nh.advertise(\*joint\_state\_pub);

}

**Copy**

In main function, call initializers:

initPosePublisher();

initJointStatePublisher();

initTfPublisher();

**Copy**

Put values to messages and publish them:

// get ROSbot pose

rosbot\_pose = rosbot.getPose();

pose.pose.position.x = rosbot\_pose[0];

pose.pose.position.y = rosbot\_pose[1];

pose.pose.orientation = tf::createQuaternionFromYaw(rosbot\_pose[2]);

// publish pose

pose\_pub->publish(&pose);

// get ROSbot tf

robot\_tf.header.stamp = nh.now();

robot\_tf.transform.translation.x = pose.pose.position.x;

robot\_tf.transform.translation.y = pose.pose.position.y;

robot\_tf.transform.rotation.x = pose.pose.orientation.x;

robot\_tf.transform.rotation.y = pose.pose.orientation.y;

robot\_tf.transform.rotation.z = pose.pose.orientation.z;

robot\_tf.transform.rotation.w = pose.pose.orientation.w;

// publish tf

broadcaster.sendTransform(robot\_tf);

ws = rosbot.getWheelsState();

pos[0] = ws.FL;

pos[1] = ws.FR;

pos[2] = ws.RL;

pos[3] = ws.RR;

joint\_states.position = pos;

joint\_states.header.stamp = nh.now();

joint\_state\_pub->publish(&joint\_states);

**Copy**

Your final code should look like this:

#**include** "hFramework.h"

#**include** "hCloudClient.h"

#**include** "ros.h"

#**include** "geometry\_msgs/Twist.h"

#**include** "sensor\_msgs/BatteryState.h"

#**include** "std\_msgs/Bool.h"

#**include** "sensor\_msgs/JointState.h"

#**include** "geometry\_msgs/PoseStamped.h"

#**include** "tf/tf.h"

#**include** "tf/transform\_broadcaster.h"

#**include** "ROSbot.h"

**using** **namespace** hFramework;

// Uncomment one of these lines, accordingly to range sensor type of your ROSbot

// If you have version with infrared sensor:

// static const SensorType sensor\_type = SENSOR\_INFRARED;

// If you have version with laser sensor:

**static** **const** SensorType sensor\_type = SENSOR\_LASER;

// If you want to use your own sensor:

// static const SensorType sensor\_type = NO\_DISTANCE\_SENSOR;

// Uncomment one of these lines, accordingly to IMU sensor type of your device

// If you have version with MPU9250:

**static** **const** ImuType imu\_type = MPU9250;

// If you want to use your own sensor:

// static const ImuType imu\_type = NO\_IMU;

// Uncomment one of these lines, accordingly version of your device

**uint32\_t** baudrate = 500000; // for ROSbot 2.0

// uint32\_t baudrate = 230400; // for ROSbot 2.0 PRO

ros::NodeHandle nh;

sensor\_msgs::BatteryState battery;

ros::Publisher \*battery\_pub;

sensor\_msgs::JointState joint\_states;

ros::Publisher \*joint\_state\_pub;

//arrays for the message

**char** \*name[] = {"front\_left\_wheel\_hinge", "front\_right\_wheel\_hinge", "rear\_left\_wheel\_hinge", "rear\_right\_wheel\_hinge"};

**float** pos[] = {0, 0, 0, 0};

**float** vel[] = {0, 0, 0, 0};

**float** eff[] = {0, 0, 0, 0};

geometry\_msgs::PoseStamped pose;

ros::Publisher \*pose\_pub;

std::vector<**float**> rosbot\_pose;

wheelsState ws;

geometry\_msgs::TransformStamped robot\_tf;

tf::TransformBroadcaster broadcaster;

**int** publish\_counter = 0;

**void** **twistCallback**(**const** geometry\_msgs::Twist &twist)

{

rosbot.setSpeed(twist.linear.x, twist.angular.z);

}

**void** **initCmdVelSubscriber**()

{

ros::Subscriber<geometry\_msgs::Twist> \*cmd\_sub = **new** ros::Subscriber<geometry\_msgs::Twist>("/cmd\_vel", &twistCallback);

nh.subscribe(\*cmd\_sub);

}

**void** **resetCallback**(**const** std\_msgs::Bool &msg)

{

**if** (msg.data == true)

{

rosbot.reset\_odometry();

}

}

**void** **initResetOdomSubscriber**()

{

ros::Subscriber<std\_msgs::Bool> \*odom\_reset\_sub = **new** ros::Subscriber<std\_msgs::Bool>("/reset\_odom", &resetCallback);

nh.subscribe(\*odom\_reset\_sub);

}

**void** **initBatteryPublisher**()

{

battery\_pub = **new** ros::Publisher("/battery", &battery);

nh.advertise(\*battery\_pub);

}

**void** **initPosePublisher**()

{

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

pose.pose.orientation = tf::createQuaternionFromYaw(0);

pose\_pub = **new** ros::Publisher("/pose", &pose);

nh.advertise(\*pose\_pub);

}

**void** **initTfPublisher**()

{

robot\_tf.header.frame\_id = "odom";

robot\_tf.child\_frame\_id = "base\_link";

robot\_tf.transform.translation.x = 0.0;

robot\_tf.transform.translation.y = 0.0;

robot\_tf.transform.translation.z = 0.0;

robot\_tf.transform.rotation.x = 0.0;

robot\_tf.transform.rotation.y = 0.0;

robot\_tf.transform.rotation.z = 0.0;

robot\_tf.transform.rotation.w = 1.0;

broadcaster.init(nh);

}

**void** **initJointStatePublisher**()

{

joint\_state\_pub = **new** ros::Publisher("/joint\_states", &joint\_states);

nh.advertise(\*joint\_state\_pub);

joint\_states.header.frame\_id = "base\_link";

//assigning the arrays to the message

joint\_states.name = name;

joint\_states.position = pos;

joint\_states.velocity = vel;

joint\_states.effort = eff;

//setting the length

joint\_states.name\_length = 4;

joint\_states.position\_length = 4;

joint\_states.velocity\_length = 4;

joint\_states.effort\_length = 4;

}

**void** **hMain**()

{

rosbot.initROSbot(sensor\_type);

RPi.init(baudrate);

platform.begin(&RPi);

nh.getHardware()->initWithDevice(&platform.LocalSerial);

nh.initNode();

initBatteryPublisher();

initCmdVelSubscriber();

initResetOdomSubscriber();

initPosePublisher();

initJointStatePublisher();

initTfPublisher();

**while** (true)

{

nh.spinOnce();

publish\_counter++;

**if** (publish\_counter > 10)

{

// get ROSbot pose

rosbot\_pose = rosbot.getPose();

pose.pose.position.x = rosbot\_pose[0];

pose.pose.position.y = rosbot\_pose[1];

pose.pose.orientation = tf::createQuaternionFromYaw(rosbot\_pose[2]);

// publish pose

pose\_pub->publish(&pose);

// get ROSbot tf

robot\_tf.header.stamp = nh.now();

robot\_tf.transform.translation.x = pose.pose.position.x;

robot\_tf.transform.translation.y = pose.pose.position.y;

robot\_tf.transform.rotation.x = pose.pose.orientation.x;

robot\_tf.transform.rotation.y = pose.pose.orientation.y;

robot\_tf.transform.rotation.z = pose.pose.orientation.z;

robot\_tf.transform.rotation.w = pose.pose.orientation.w;

// publish tf

broadcaster.sendTransform(robot\_tf);

ws = rosbot.getWheelsState();

pos[0] = ws.FL;

pos[1] = ws.FR;

pos[2] = ws.RL;

pos[3] = ws.RR;

joint\_states.position = pos;

joint\_states.header.stamp = nh.now();

joint\_state\_pub->publish(&joint\_states);

// get battery voltage

battery.voltage = rosbot.getBatteryLevel();

// publish battery voltage

battery\_pub->publish(&battery);

publish\_counter = 0;

}

sys.delay(10);

}

}

**Copy**

Build your project and upload it to the device.

**Running motor controller with forward kinematics task**

In this section you will control your robot with keyboard and observe as it publishes its own position.

If you are working with ROSbot: Log in to your CORE2 device through remote desktop, run terminal and start your robot as previously. In another terminal window run:

rostopic echo /pose

**Copy**

If you are working with Gazebo: Start Gazebo as prevoiusly. In another terminal window run:

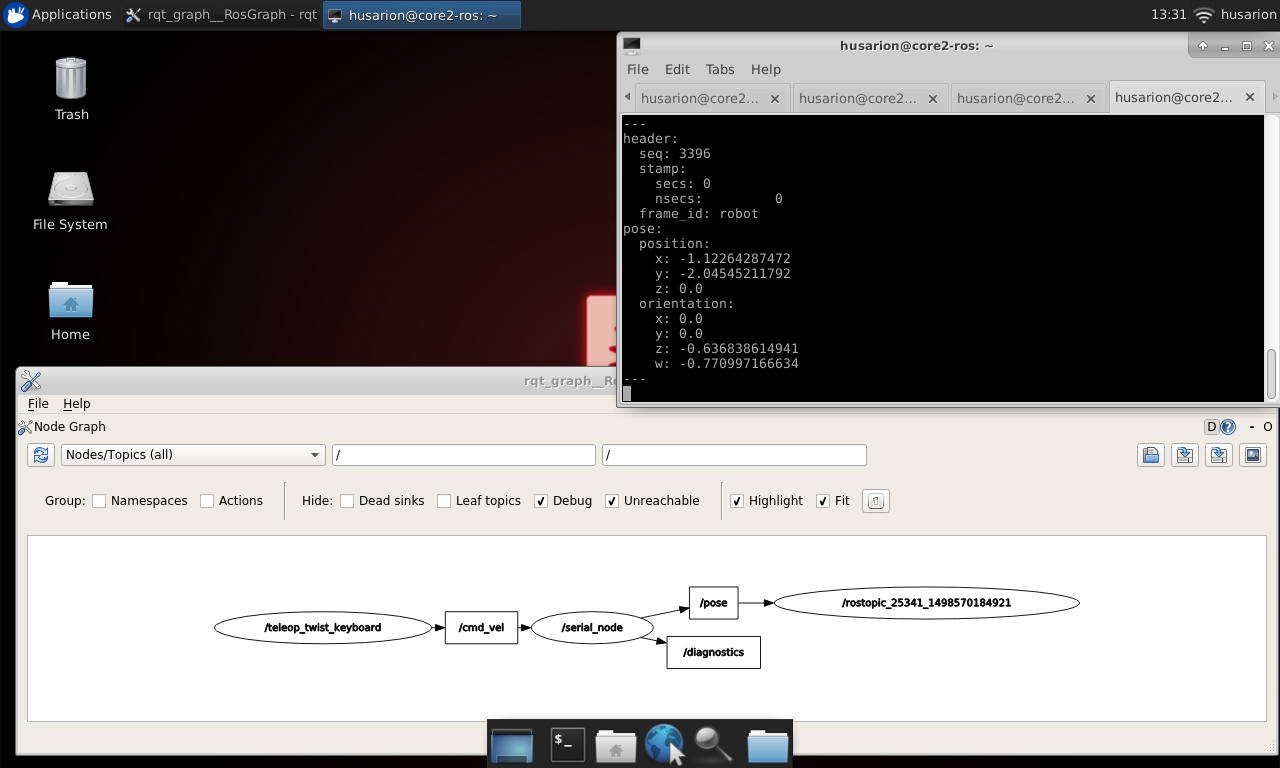
rostopic echo /odom

**Copy**

Above difference comes from the fact, that Gazebo and ROSbot are publishing its position in different ways.

Remember, that you need to have active window with teleop\_twist\_keyboard to control robot movement.

You should get something like this on your screen:



**Data visualization with PlotJuggler**

In this section you will learn how to visualize data from ros topics using PlotJuggler. It is a simple tool that allows you to plot logged data, in particular timeseries. You can learn more about the tool on its [official webpage](https://github.com/facontidavide/PlotJuggler).

**How to use**

Start PlotJuggler:

rosrun plotjuggler PlotJuggler

**Copy**

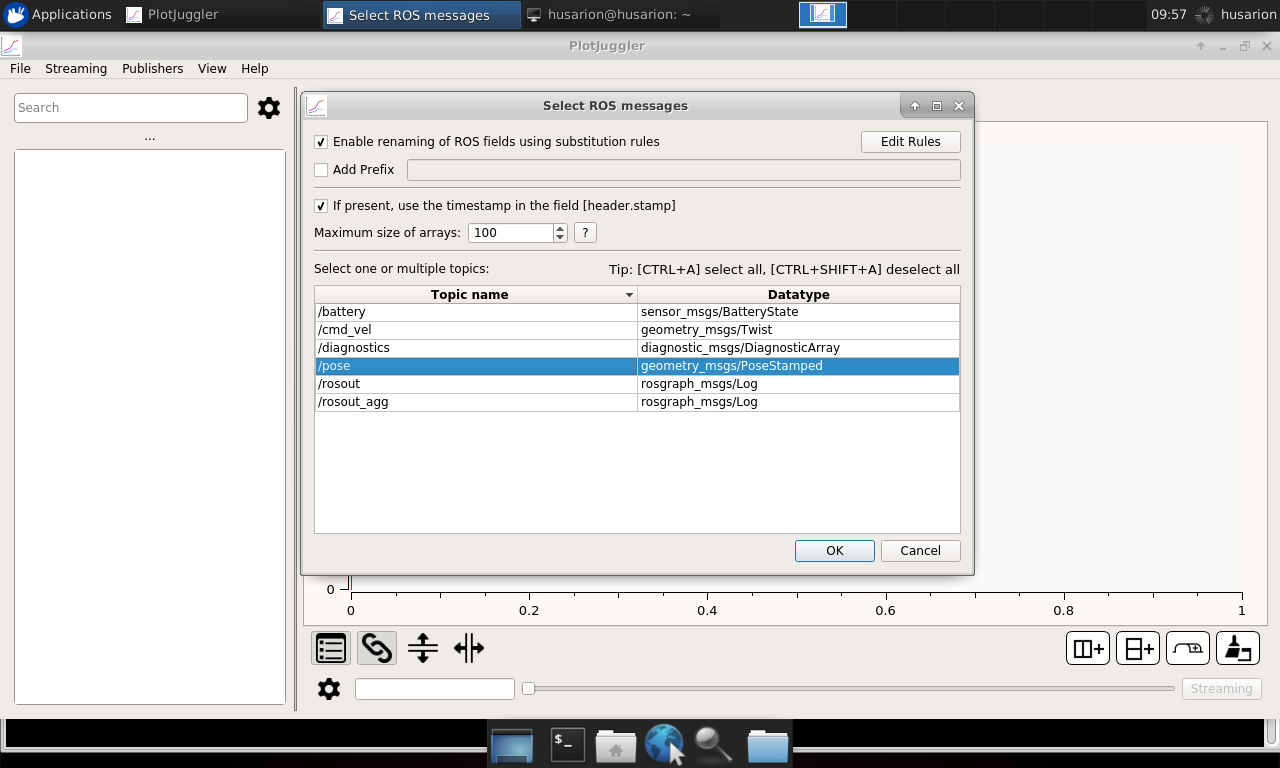
In case your image lacks this tool you can install it by typing:

sudo apt-get update

sudo apt-get install ros-kinetic-plotjuggler

**Copy**

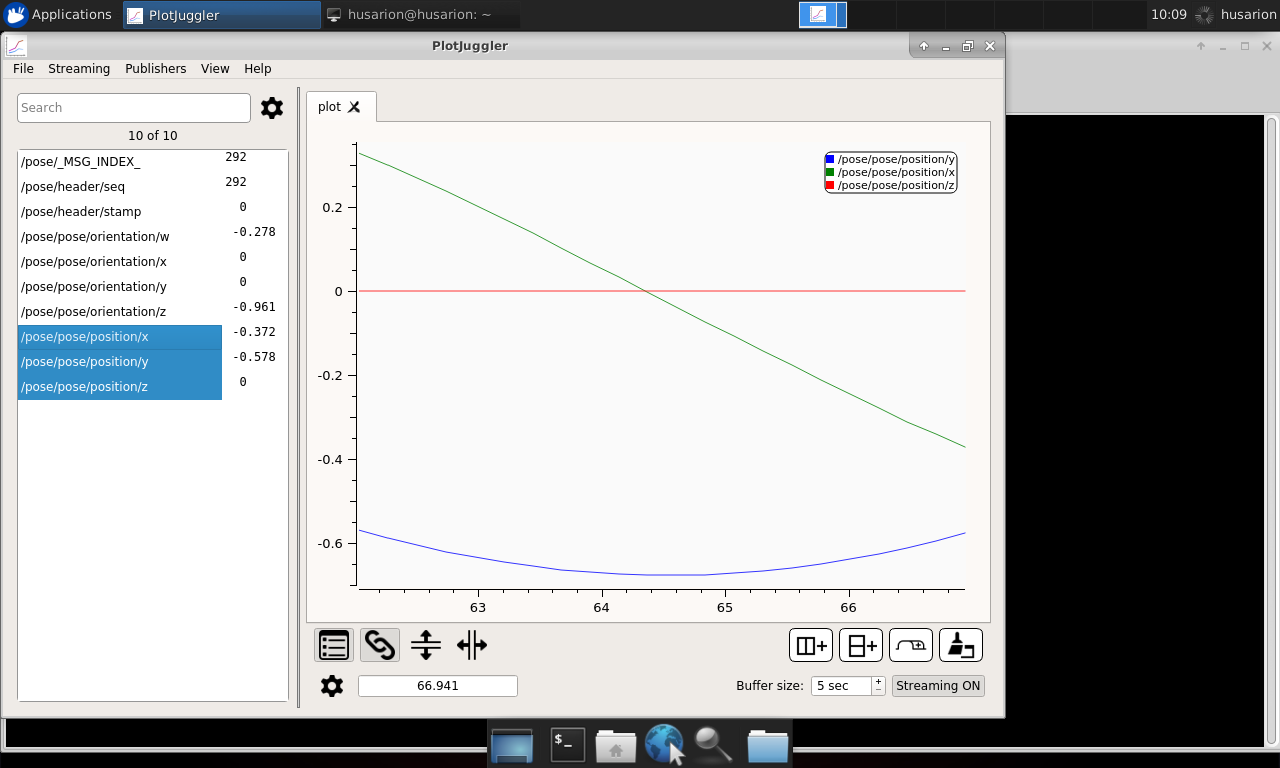
From menu bar select **Streaming > Start: ROS\_Topic\_Streamer**. In pop-up menu that will appear choose **/pose**from available topic names and press ok.



Pressing **CTRL** and **SHIFT** select positions:

* */pose/pose/position/x*
* */pose/pose/position/y*
* */pose/pose/position/z*

and then drag and drop them to the window area. This way you can comfortably observe changes in the odometry data during robot motion:



**Robot visualization with Rviz**

Rviz is tool which allows visualization of robot position, travelled path, planned trajectory, sensor state or obstacles surrounding robot.

To run it type in terminal:

rviz

**Copy**

You can also start all nodes with single .launch file:

<**launch**>

<**arg** name="use\_rosbot" default="true"/>

<**arg** name="use\_gazebo" default="false"/>

<**include** if="$(arg use\_gazebo)" file="$(find rosbot\_description)/launch/rosbot.launch"/>

<**node** name="rviz" pkg="rviz" type="rviz" args="-d $(find tutorial\_pkg)/rviz/tutorial\_3.rviz"/>

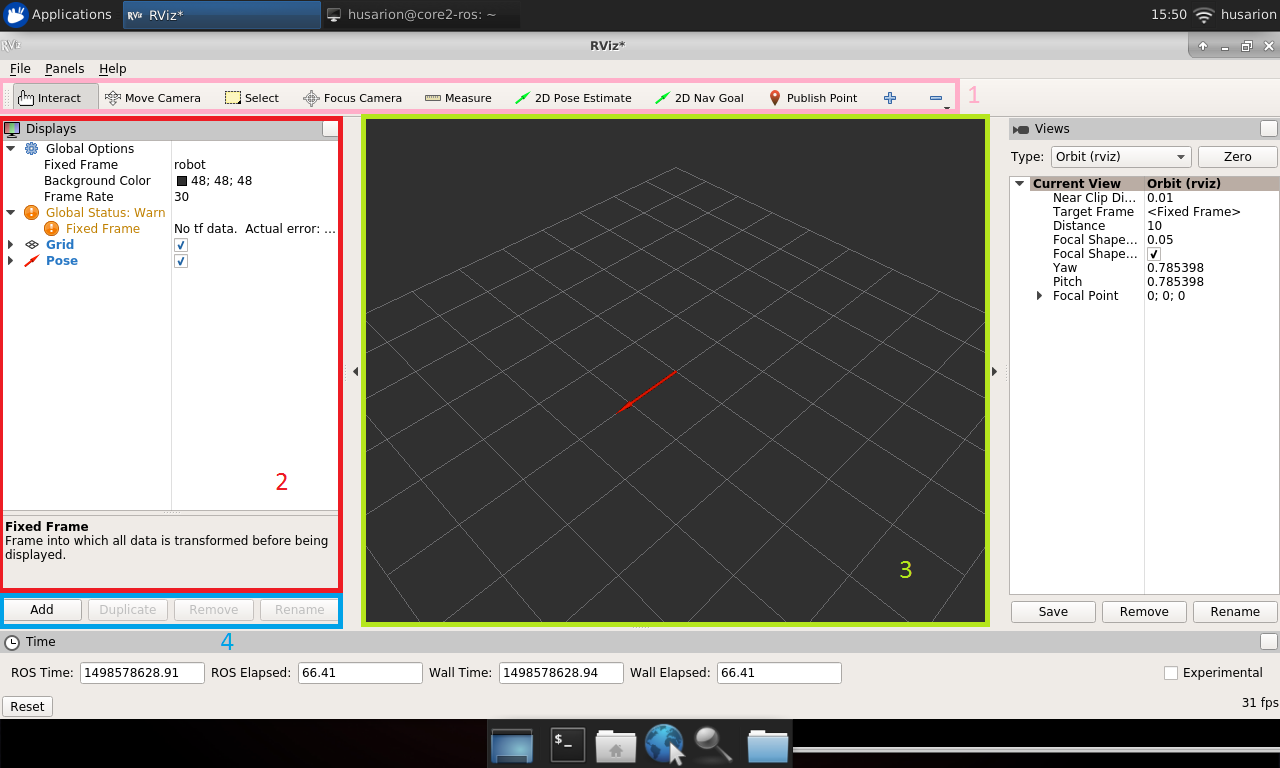
<**node** name="teleop\_twist\_keyboard" pkg="teleop\_twist\_keyboard" type="teleop\_twist\_keyboard.py" output="screen"/>

</**launch**>

**Copy**

Rviz can be launched with argument pointing to file .rviz, it will contain window configuration. When you are satisfied with visualization parameters, choose **File** -> **Save config**. At next rviz launch config will be restored.

New window will appear:



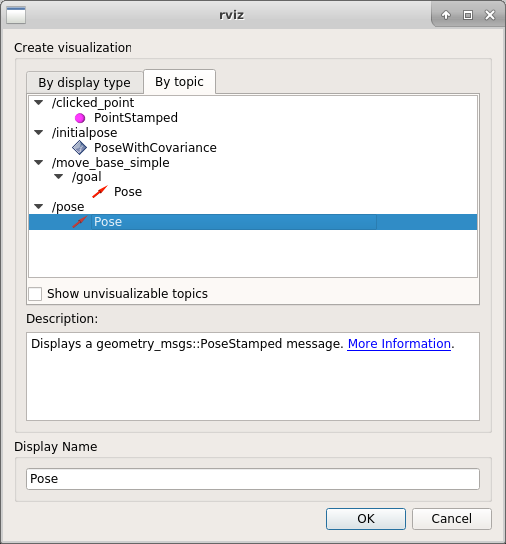
Application main view consists of:

1. Toolbar
2. Visualized items list
3. Visualization window
4. Object manipulation buttons

By default you will see only base frame, to add any other object push **Add** from object manipulation buttons. In new window, there are two tabs **By display type** and **By topic**. First one is for manual selection from all possible objects, second one contains only currently published topics.

After you choose object to display, click **OK** button and it will appear in visualization window.

Now we will visualize position published by your robot, run rviz, click **Add** and choose tab **By topic**.



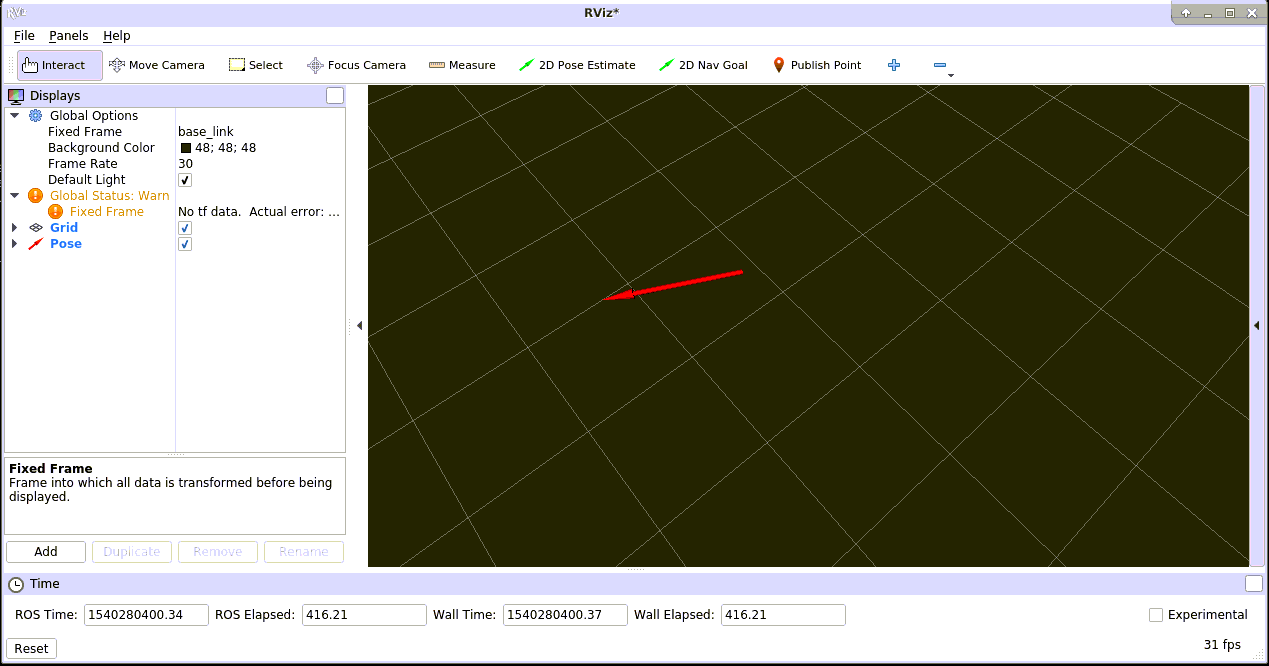
If you are working with ROSbot: Find topic /pose and choose Pose and click **OK**.

If you are working with Gazebo: Find topic /odom and choose Odometry and click **OK**.

Again click **Add**, choose tab **By display type**, **TF** and click **OK**.

Then in visualized items list find position Fixed Frame and from dropdown list choose odom.

After this is done, you should see an arrow representing position and orientation of your robot. You will see also representation of coordinate frames bounded with robot starting position and robot base. Move your robot and observe as arrow changes its position.



Visualization of any other item is performed in the same way. In further lessons, as you will produce more objects to visualize, you will add them to the same view.

**Summary**

After completing this tutorial you should be able to control motor attached to your CORE2 device, set desired velocity for robot with geometry\_msgs/Twist message, determine position of your robot using odometry, publish it into tf frames or as a PoseStamped message and visualize position of your robot using rviz.

*by Łukasz Mitka, Husarion*

*Do you need any support with completing this tutorial or have any difficulties with software or hardware? Feel free to describe your thoughts on our community forum:*[*https://community.husarion.com/*](https://community.husarion.com/)*or to contact with our support:*[*support@husarion.com*](mailto:support@husarion.com)

[← 2. CREATING NODES](https://husarion.com/tutorials/ros-tutorials/2-creating-nodes)[4. VISUAL OBJECT RECOGNITION →](https://husarion.com/tutorials/ros-tutorials/4-visual-object-recognition)