An **inertial measurement unit** (**IMU**) is an electronic device that measures the inclination of the robot’s link.

The **Twist** structure contains two 3D vectors. First vector defines the **linear velocity** and the second vector defines **angular velocit**y. Our robot lives in the 2D world so we can move forward/backward (“x” axis) and rotate (around the “z” axis). To move the robot publish the data to the “cmd\_vel” topic (set the proper values for the linear and angular velocity).

**Rosbag** is a set tools for recording from and playing back to ROS topics. It is intended to be high performance and avoids deserialization and reserialization of the messages.

* has a way of recording the data from multiple components at once
* has a way of replaying the same data in the same way
* In the end it is all about the verification of the system’s performance

**Rosservice** is another way that nodes can communicate with each other. Services allow nodes to send a request and receive a response. If service is empty, this means when the service call is made it takes no arguments (i.e. it sends no data when making a request and receives no data when receiving a response). To spawn a new turtle we have to provide x, y, theta arguments:

$ rosservice type /spawn | rossrv show

float32 x

float32 y

float32 theta

string name

---

string name

$ rosservice call /spawn 2 2 0.2 ""

Where x = 2, y = 2, z = 0.2

**Rosparam** allows you to store and manipulate data on the ROS Parameter Server. The Parameter Server can store integers, floats, boolean, dictionaries, and lists. Rosparam uses the YAML markup language for syntax. In simple cases, YAML looks very natural: 1 is an integer, 1.0 is a float, one is a string, true is a boolean, [1, 2, 3] is a list of integers, and {a: b, c: d} is a dictionary.

1) "--clock" connects bag file to clock server. In our case, we use --clock to get a mapping process in realtime in RVIZ.

2) $ rosbag info

/scan

10376 msgs

=> 10376 laser scans

3) To display the current pose of the robot:

$ rostopic echo /<name\_of\_robot>/pose

4) Framerate of the pose information is equal to 30.

This data is gotten from rviz:

Displays => Global Options => Frame Rate

5) $ rosservice list

/pause\_mapping

(Y/N)

6) $ rosparam get /

hector\_mapping:

map\_size: 2048

In practice, we usually **have to create our own nodes**: either to perform simple data conversions or to do more sophisticated operations.

It is just worth noting that robotic systems are usually using many simultaneous threads and are often distributed making debugging much more complicated in practice.

It is important to remember that right now, the ROS does not know about the existence of the *talker.py* script. We have to manually inform that this is a script that can be executed by adding the following line to the *CMakeLists.txt*:

catkin\_install\_python(PROGRAMS scripts/talker.py

  DESTINATION ${CATKIN\_PACKAGE\_BIN\_DESTINATION}

)

**Attention!** We should modify the CMakeLists.txt INSIDE our begginer\_tutorials package.

Publishing information is one-way communication. In reality, in most cases, we need to interact with different nodes by also receiving information. Therefore, we will now create a simple node subscriber following further the same tutorial. Remember that the **division between publishing and subscribing is just for educational** purposes as usually, most nodes do both of these operations at the same time, i.e., receive some information, process it, and publish the results of the processing for further use.

**Tip 1**: The turtle is NOT spawned at position (0, 0) based on messages available at  `/turtle1/pose`. X is a horizontal axis, while Y is vertical.

**Tip 2**: The goal and pose are specified in the global coordinate system while cmd\_vel is provided in the local coordinate system of the turtle.

What type of ROS message is used to store images? - sensor\_msgs/CompressedImage

ROS tf2 can be used to store information about coordinate systems and the relative poses between them (transformations).

The **roslaunch** command is just a way of executing **roscore** and multiple **rosrun** commands, which is performed based on a XML launch file.

**tf\_echo**determines the relation between any two frames available in the ROS system. I underline *any* as **tf\_echo** can find the relation between any coordinate systems (frames) in the connected graph performing all of the necessary intermediate computation for you. Let’s check the relation between the turtles:

$ rosrun tf tf\_echo turtle1 turtle2

The execution of this command provides you with all of the information about the transformation: translation is represented in the form of the translation vector while rotation is represented in 3 ways utilizing quaternions, RPY (roll, pitch yaw) angles in radians, and RPY angles in degrees.

We can use **tf\_conversions**to help us handle between different representations of transformations.

buffer\_.transform(\*point\_ptr, point\_out, target\_frame\_);

In this case, the original point with a specified frame (*point\_ptr*) can be recomputed to the other coordinate system (*point\_out*) just by specifying which coordinate system we would like to transform to (*target\_frame*). It works for points, point clouds (multiple points), some maps, etc. but always check if it is implemented as far as I know it is still not implemented for velocities or uncertainties.

* ROS **tf2** can be used to find the transformation for any of the provided timestamps (we used it only for ‘now’),
* ROS **tf2** interpolated the transformations so the accuracy should be better when transformations are available with low resolution,
* ROS **tf2** can be used to transform different, standard data types.

A robot front corresponds to the X axis of the base coordinate system.

* What is the degree of freedom (**DoF**)?

A number of independent motions that the robot can execute in space. For example, the robot from our homework has two **DoF**, because it can execute motion using only two rotational joints. Probably the most popular type of robotic manipulator is a 6 **DoF** robotic arm.

* What is the difference between **links and joints**?

**Links** are rigid connections between joints and typically are defined as constants. **Joints** cause the relative motion between consecutive parts of robots.

* What **types of joints** do manipulators have?

In our classes, we consider two types of joints - **prismatic (translational) and rotational**. In the analysis of the kinematic chains, each joint represents a relative rigid transformation (e.g. joint that rotates around its Z-axis). We will consider two types of such **transformations - translations and rotations.**

* What is a **kinematic chain**?

It is a mathematical model of an actual robot that specifies the motion of a robot. Analyzing kinematic chains we can calculate the position and orientation of the end-effector concerning joint values (i. e. consecutive rotations and translations) or go in the opposite direction and calculate joint values given the end-effector’s pose.

• What is **Forward Kinematics**?

Simply speaking, FK is a task of calculating the robot’s end-effector pose given joint values. For example for the 6 DoF robotic arm with 6 rotational joints the FK can be depicted as:

Joint angles [α, β, γ, δ, ε, ζ] → position [x, y, z], orientation [rot\_x, rot\_y, rot\_z]

!! Each joint represents some transformation in the Euclidean space.

* How do we **calculate transformations between links**?

As was said, each joint represents some transformation in the Euclidean space. For each joint in the kinematic chain, we should specify the coordinate frame associated with it and then specify the transformation matrix (explained in the next point as the homogeneous matrix). In the end, we take the dot product of defined transformation matrices to calculate the transformation between the base link and the end-effector.

* What are **homogeneous matrices**?

To calculate the transformation between two frames we can use the homogeneous matrices (we referred to them as transformation matrices until now). It is defined as the 4x4 matrix that includes some rotation (3x3 R matrix) and translation (3x1 T vector).

def rotation\_x(theta):

h = np.array([[1, 0, 0, 0],

[0, np.cos(theta), -np.sin(theta), 0],

[0, np.sin(theta), np.cos(theta), 0],

[0, 0, 0, 1]])

def rotation\_y(theta):

h = np.array([[np.cos(theta), 0, np.sin(theta), 0],

[0, 1, 0, 0],

[-np.sin(theta), 0, np.cos(theta), 0],

[0, 0, 0, 1]])

def rotation\_z(theta):

h = np.array([[np.cos(theta), -np.sin(theta), 0, 0],

[np.sin(theta), np.cos(theta), 0, 0],

[0, 0, 1, 0],

[0, 0, 0, 1]])

def translation\_xyz(x, y, z):

h = np.array([[1, 0, 0, x],

[0, 1, 0, y],

[0, 0, 1, z],

[0, 0, 0, 1]])

If we want to **calculate the transformation between the 1st frame of a kinematic chain and the last one** we have to define homogeneous matrices for each joint and take the dot product in the correct order.

What is **Denavit-Hartenberg notation**?

Simply speaking, each joint’s transformation is described as a dot product of four consecutive homogeneous matrices representing:

1. rotation around Z-axis
2. translation along Z-axis
3. translation along a new X-axis
4. rotation around a new X-axis

Each transformation represents rotation or translation around/along only one axis.

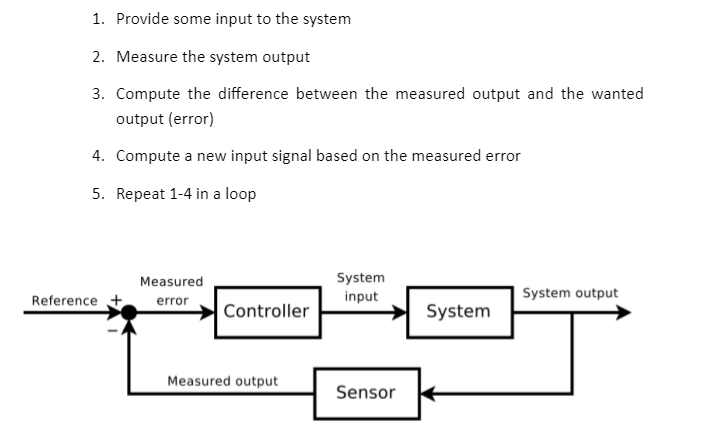
* What is inverse kinematics (IK)?

The IK task is something exactly the opposite of the FK. This is the function that gives us the joint values that we need in order to achieve the desired end-effector pose (position and orientation).

position [x, y, z], orientation [rot\_x, rot\_y, rot\_z] → Joint angles [α, β, γ, δ, ε, ζ]

Calculated joint values will be None if there is no solution for the specified pose, for example when the interaction marker is outside the workspace of the robot or the robot would self collide.

**PID**



In real-life scenarios, when we want to reach a certain value (i.e., position/orientation/speed) we have to properly steer the physical object (i.e., engine or actuator) in order to reach the desired state.

The PID takes the measured error and produces the new system’s input signal based on the equation:



Where:

* Kpe(t)is a proportional part (P in PID) that scales the controller output (system’s input) linearly to the last measured error,
* Ki0te(t) dtis a integral part (I in PID) that scales the controller output (system’s input) based on the integrated (in practice it is a sum) of historically measured errors,
* Kdde(t)dtis a derivative part (D in PID) that scales the controller output (system’s input) based on the derivative of historically measured errors.

Methods of **tuning PID** according to the dynamics of the controlled system:

* The proportional part is usually set first as it is responsible for fast response to the measured errors. We usually set it to the greatest value that does not result in oscillations (system constantly overshooting the wanted target, i.e., when asked to reach value 1 from 0, it reaches 1.5, then 0.75, then 1.25, etc. and slowly converges to the wanted value)
* The derivative part is used as a second one to (almost) remove overshoot (not going over 1 when going from 0 to 1)
* The integral part is used to remove any residual offset that might appear when using the PD controller (i.e., PD might get ‘stuck’ at 0.95 when going from 0 to 1 and I can be used to reach 1)

**Kalman filter** - “An algorithm that uses a series of measurements observed over time, containing statistical noise and other inaccuracies, and produces estimates of unknown variables that tend to be more accurate than those based on a single measurement alone”

To put it short, the **Kalman filter** joins two sources of estimates to provide the best feasible estimate. These sources are:

* Prediction from the previous timestamp using the so-called systems model
* Measurements from sensors

In **the Kalman filter**, each time moment is represented by the current estimate (**x**) that contains the most probable state of the system and the covariance matrix (**P**) that determines the distribution of the estimate (in other words - how believable is the current estimate).

The processing is divided into two steps connected with sources of information:

1. prediction - predicts the next state of the system by moving forward in time. Increases our uncertainty about the system’s state
2. update - correct the predicted state by including the external measurements. It reduces our uncertainty about the system’s state

In practice **you have to define**:

* the size and content of state vector (**x**)
* how state from consecutive time moments are related (**F, B**) but in many scenarios **B=0**
* which element is measured with an external sensor (**H**)
* the values of noise of the model (prediction) **Q** and the noise of the measurement (update) **R**. In practice it is not about absolute values but the relation of **Q** and **R**

**Odometry** is done by taking the original velocity and adding some Gaussian noise to it. (relative, velocity)

**GPS** (absolute position)

**AHRS** sensor provides orientation

**Calibration**

The image acquisition process is approximated by the chosen model of registration of such an image. The most popular is the pinhole camera model, where the registration location of a point in the image in pixels (u,v) depends on the position of the point in 3D space (X, Y, Z) and on the internal parameters of the camera consisting of optical center (cx, cy), focal lengths (fx, fy) and possible distortion parameters corresponding to radial and tangential distortions.

The goal of **calibration** is to find these parameters so that the image registration model matches the actual measurements, which is critical when projecting points onto an image or rectifying images for other computer vision algorithms.

And several plots informing about the **results of the calibration**. There are several indicators of a good calibration result:

* The number of finally used images exceeds ~30 images
* The +- values for the projection (focal lengths and image center) are rather small (below 1 px)
* The reprojection error is below 1 px