

# Coordinate systems and transforms

Ivan Marković   Matko Orsag   Damjan Miklič

Automation and Robotics  
Robot Programming and Simulation

2021



UNIVERSITY OF ZAGREB

Faculty of Electrical  
Engineering and  
Computing

- ➊ Introduction to coordinate systems
- ➋ The ROS TF tree
- ➌ Transformation matrices
- ➍ rospy TF2 API

- Install ROS TF2 tools

```
$ sudo apt install ros-noetic-tf2-tools
```

- Install Python symbolic math library

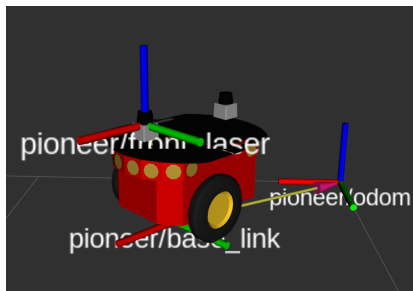
```
$ sudo apt install isympy3
```

- Clone the PSR Stage Worlds and build your workspace

```
$ roscd && cd ../src  
$ git clone \  
> https://github.com/pftros/rps_stage_worlds.git  
$ cd ..  
$ catkin_make  
$ source devel/setup.bash
```

# Coordinate systems

We **assign** coordinate systems (frames) to localize (parts of) objects **relative** to each other.

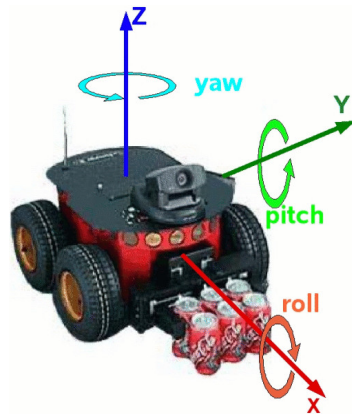


By convention, we use **right-handed** Cartesian coordinate systems with principal axes **x**, **y** and **z**.

# Representing rotations in 3D

A **minimal** description (parametrization) of rotation in 3D is given by Euler angles. In mobile robotics we use the **convention**:

- **roll** - rotation around **x-axis**
- **pitch** - rotation around **y-axis**
- **yaw** - rotation around **z-axis**
- Positive rotation is counterclockwise



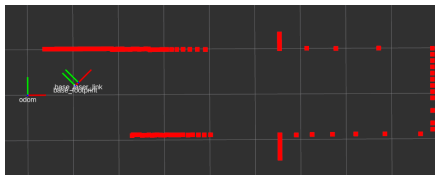
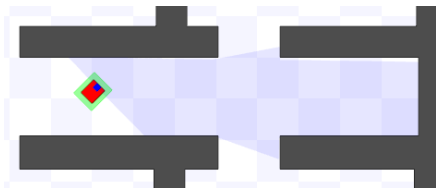
## Rotation conventions

There are **many** possible Euler angle definitions, e.g., for manipulators, rotation around the end-effector z-axis is typically called roll. When assigning angles, be careful to check the convention!

# Coordinate systems in 2D

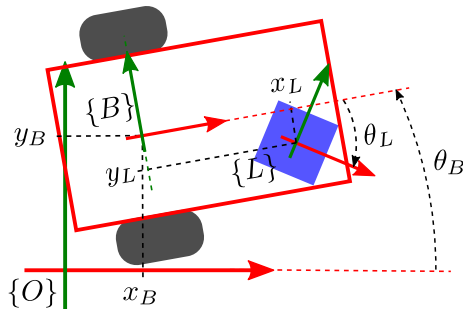
For simplicity, we will consider a 2D (planar) world, ignoring  $z$ , roll and pitch:

```
$ roscore&
$ rosparam set use_sim_time_true
$ rosrunc stage_ros stageros \
> `rospack find rps_stage_worlds`/worlds/simple_rps.world
$rviz -d \
> `rospack find rps_stage_worlds`/rviz/simple_rps.rviz
```



# Coordinates in 2D: a naive representation

- A 2D **pose** consists of position  $(x, y)$  and orientation (yaw)  $\theta$
- Every pose defines a **frame**, and every frame has a pose
- A **transform** links the poses of two frames
- A pose is **always** expressed relative to a frame of reference
- base\_footprint pose in the odom frame:  
 $p_{OB} = (x_{OB}, y_{OB}, \theta_{OB})$
- Laser pose in the base\_footprint frame:  
 $p_{BL} = (x_{BL}, y_{BL}, \theta_{BL})$



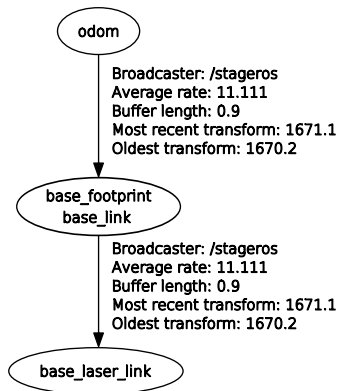
## Notation

For brevity and aesthetics, we will often simplify indexing by omitting the frame of reference when we are referring to coordinates in the **parent** frame, e.g.  $x_{OB} \rightarrow x_B$ .

# Frame relationships

- Parent-child relationships
- By ROS convention, represented by a **(poly)tree** (directed acyclic graph)
- A node may have only one parent!

```
$ rosrun rqt_tf_tree rqt_tf_tree
```



## Note

Because `base_footprint` and `base_link` frames coincide in 2D, we will treat them as one frame in this lecture. We will also refer to `base_laser_link` as `laser_link`.



# Querying frame relationships using tf2\_tools

The TF2 `echo.py` script can provide the relative pose of any frame in the TF tree:

```
$ rosrun tf2_tools echo.py odom base_laser_link
At time 9698.0, (current time 9698.0)
- Translation: [0.071, 0.071, 0.420]
- Rotation: in Quaternion [0.000, 0.000, 0.383, 0.924]
             in RPY (radian) [0.000, -0.000, 0.785]
             in RPY (degree) [0.000, -0.000, 45.000]
```

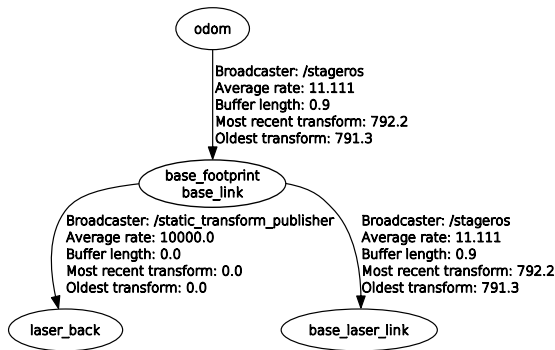
## TF and TF2

TF2 is a new implementation of the TF library. TF is still available for backwards compatibility, but it is deprecated. Always use TF2!

# Publishing transforms to the TF tree

The TF2 `static_transform_publisher` can publish **static** transforms to the TF tree:

```
$ rosrn tf2_ros static_transform_publisher \  
-0.1 0.0 0.0 3.14 0 0 base_link laser_back
```



```
$ rosrn tf2_tools echo.py base_laser_link laser_back
```

# Transforming coordinates by hand

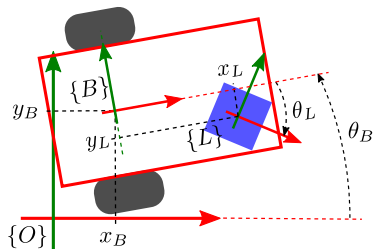
## Assignment: Transforming a pose

Assuming we can directly measure robot pose  $p_B$  in the odom frame and laser pose  $p_L$  in the base\_link frame, find the equations of the laser pose  $p_{OL}$  expressed in the odom frame.

$$x_{OL} = \dots$$

$$y_{OL} = \dots$$

$$\theta_{OL} = \dots$$



# Transforming coordinates by hand

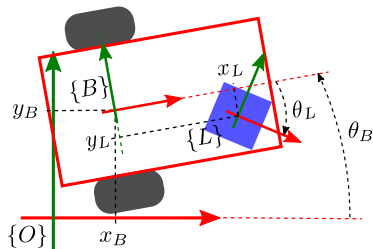
## Assignment: Transforming a pose

Assuming we can directly measure robot pose  $p_B$  in the odom frame and laser pose  $p_L$  in the base\_link frame, find the equations of the laser pose  $p_{OL}$  expressed in the odom frame.

$$x_{OL} = x_B + x_L \cos(\theta_B) - y_L \sin(\theta_B)$$

$$y_{OL} = y_B + x_L \sin(\theta_B) + y_L \cos(\theta_B)$$

$$\theta_{OL} = \theta_B + \theta_L$$



## Frame Transforms

We say that by the equations above, we have **transformed** the laser pose from the base\_link frame to the odom frame.

# Inverting the coordinate transform (by hand)

## Assignment: Inverting a transform

Find the equations for the pose of the `odom` frame expressed in the `laser_link` frame,  $p_{LO}$ . You can verify your solution for specific values in the Stage simulation using `echo.py` from `tf2_tools`.

**Warning:** Doing this by hand is tricky, it is ok if you are unable to come up with a solution, we will show a more elegant way of finding this transform.

# Inverting the coordinate transform (by hand)

## Assignment: Inverting a transform

Find the equations for the pose of the odom frame expressed in the laser\_link frame,  $p_{LO}$ . You can verify your solution for specific values in the Stage simulation using `echo.py` from `tf2_tools`.

**Warning:** Doing this by hand is tricky, it is ok if you are unable to come up with a solution, we will show a more elegant way of finding this transform.

$$x_{LO} = -x_B \cos(\theta_B + \theta_L) - y_B \sin(\theta_B + \theta_L) - x_L \cos(\theta_L) - y_L \sin(\theta_L)$$

$$y_{LO} = x_B \sin(\theta_B + \theta_L) - y_B \cos(\theta_B + \theta_L) + x_L \sin(\theta_L) - y_L \cos(\theta_L)$$

$$\theta_{LO} = -\theta_B - \theta_L$$

```
$ rosrun tf2_tools echo.py base_laser_link odom
```

# The problem

Problems with manual handling of coordinate transforms:

- Can be difficult to visualize
- Complex algebraic expressions
- Error prone
- Becomes a mental health hazard in 3D

## The solution

We need some more powerful mathematical tools for dealing with coordinate transforms. We would like to have well-defined mathematical objects that we can (efficiently) do algebra with.

# Describing transforms with matrices

## Rotation matrix

A 2D rotation by angle  $\theta$  can be written down in **matrix form** as

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$



# Describing transforms with matrices

## Rotation matrix

A 2D rotation by angle  $\theta$  can be written down in **matrix form** as

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

## Transformation matrix

A 2D coordinate transformation from child frame  $C$  to parent frame  $P$ , consisting of a rotation by angle  $\theta$  and translation by vector  $\mathbf{t} = [x, y]^T$  can be written as

$$\mathbf{T}_{PC} = \begin{bmatrix} \mathbf{R}(\theta) & \mathbf{t} \\ \mathbf{0}_{[1 \times 2]} & 1 \end{bmatrix}$$

# Transformation matrix examples

Robot pose in odom:

$$\mathbf{T}_{OB} = \begin{bmatrix} \cos(\theta_B) & -\sin(\theta_B) & x_B \\ \sin(\theta_B) & \cos(\theta_B) & y_B \\ 0 & 0 & 1 \end{bmatrix}$$

Laser pose in base link:

$$\mathbf{T}_{BL} = \begin{bmatrix} \cos(\theta_L) & -\sin(\theta_L) & x_L \\ \sin(\theta_L) & \cos(\theta_L) & y_L \\ 0 & 0 & 1 \end{bmatrix}$$

# Transforming coordinates by matrix multiplication

## Transforming coordinates

Written down in matrix form, transforms become matrix multiplications, e.g. laser pose in odom frame:

$$\begin{aligned}\mathbf{T}_{OL} &= \mathbf{T}_{OB} \cdot \mathbf{T}_{BL} \\ &= \begin{bmatrix} \cos(\theta_B + \theta_L) & -\sin(\theta_B + \theta_L) & x_B + x_L \cos(\theta_B) - y_L \sin(\theta_B) \\ \sin(\theta_B + \theta_L) & \cos(\theta_B + \theta_L) & y_B + x_L \sin(\theta_B) + y_L \cos(\theta_B) \\ 0 & 0 & 1 \end{bmatrix}\end{aligned}$$

# Transforming coordinates by matrix multiplication

## Transforming coordinates

Written down in matrix form, transforms become matrix multiplications, e.g. laser pose in odom frame:

$$\begin{aligned}\mathbf{T}_{OL} &= \mathbf{T}_{OB} \cdot \mathbf{T}_{BL} \\ &= \begin{bmatrix} \cos(\theta_B + \theta_L) & -\sin(\theta_B + \theta_L) & x_B + x_L \cos(\theta_B) - y_L \sin(\theta_B) \\ \sin(\theta_B + \theta_L) & \cos(\theta_B + \theta_L) & y_B + x_L \sin(\theta_B) + y_L \cos(\theta_B) \\ 0 & 0 & 1 \end{bmatrix}\end{aligned}$$

## Retrieving the angle from the rotation matrix

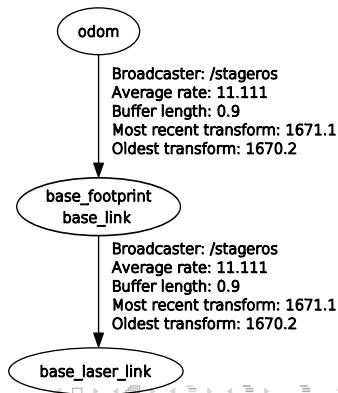
$$\theta_{OL} = \text{atan2}(r_{21}, r_{11}) = \text{atan2}(\sin(\theta_B + \theta_L), \cos(\theta_B + \theta_L)) = \theta_B + \theta_L$$

# Inverting transforms

## Transform inversion

Transforms are inverted by inverting matrices. By multiplying transform matrices and/or their inverses, we can "walk" up and down the transform tree, finding any transform that we need.

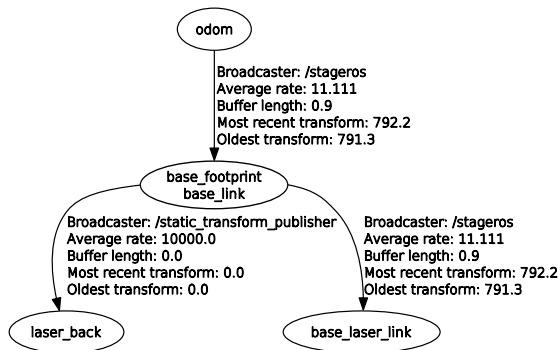
$$\begin{aligned}\mathbf{T}_{LO} &= (\mathbf{T}_{OL})^{-1} = (\mathbf{T}_{OB} \cdot \mathbf{T}_{BL})^{-1} \\ &= (\mathbf{T}_{BL})^{-1} \cdot (\mathbf{T}_{OB})^{-1} = \mathbf{T}_{LB} \cdot \mathbf{T}_{BO}\end{aligned}$$



# Walking the TF tree

## Assignment

Given the TF tree with front and back (rear) lasers, depicted in the image below, express the relative pose of the rear laser in the front laser frame  $T_{LR}$  in terms of parent-child transforms  $T_{OB}$ ,  $T_{BL}$  and  $T_{BR}$ .



# Symbolic computation in sympy

Sympy is a Python library for symbolic computation:

```
$ isympy3
```

```
In [1]: xB, yB, thetaB = symbols('x_B y_B theta_B')
```

```
In [2]: T_OB = Matrix([[cos(thetaB), -sin(thetaB), xB],\
...: [sin(thetaB), cos(thetaB), yB], [0,0,1]])
```

```
In [3]: xL, yL, thetaL = symbols('x_L y_L theta_L')
```

```
In [4]: T_BL = Matrix([[cos(thetaL), -sin(thetaL), xL],\
...: [sin(thetaL), cos(thetaL), yL], [0,0,1]])
```

```
In [5]: T_OL = T_OB*T_BL
```

```
In [6]: T_OL.simplify()
```

```
In [7]: T_OL
```

```
Out[7]: ...
```

```
In [8]: T_LO = T_OL.inv()
```

# Recap (theory)

- Right-handed Cartesian coordinate systems for spatial relationships
- Object pose consists of position and orientation
- Euler angles are a minimal description of orientation
- We use the **roll-pitch-yaw** convention
- Poses, coordinate frames and transforms are equivalent
- Poses are always relative
- Matrices are a mathematically convenient pose representation



# Recap (ROS tools)

ROS provides convenience tools for handling transforms:

- `rqt_tf_tree` for visualizing transforms
- `tf2_tools echo.py` for getting pose info
- `tf2_ros static_transform_publisher` for publishing

# The TF2 Python API: An example

## Task

Write a node that is going to compute the coordinates of the closest obstacle point detected by the laser scanner and broadcast its pose to the TF tree, in the odom frame.

Solution outline:

- 1 Write the program structure (the action will be taking place in the LaserScan callback)  

```
$ git clone \> https://github.com/pftros/rps_tf2_tutorial.git
```
- 2 Implement a TF2 **broadcaster**, sending a dummy pose in front of the laser
- 3 Compute the closest obstacle coordinates and broadcast them in the `laser_link` frame
- 4 Transform the obstacle coordinates and broadcast them in the odom frame

# Broadcasting a transform (initialization)

```
import tf2_ros
from geometry_msgs.msg import TransformStamped
from tf_conversions import transformations

def __init__(self):
    # ...
    self.tf_bcaster = tf2_ros.TransformBroadcaster()
    self.tf_laser_obst = TransformStamped()
    # Initialize the constant transform fields
    self.tf_laser_obst.header.frame_id = 'base_laser_link'
    self.tf_laser_obst.child_frame_id = 'obstacle'
    self.tf_laser_obst.transform.translation.z = 0.0
```

# Broadcasting a transform (broadcasting)

```
def scan_callback(self, scan):  
    """ This is where all the action happens! """  
    self.tf_laser_obst.header.stamp = rospy.Time.now()  
    self.tf_laser_obst.transform.translation.x = 1  
    self.tf_laser_obst.transform.translation.y = 0.5  
    q = transformations.quaternion_from_euler(0, 0, 0.707)  
    self.tf_laser_obst.transform.rotation.x = q[0]  
    self.tf_laser_obst.transform.rotation.y = q[1]  
    self.tf_laser_obst.transform.rotation.z = q[2]  
    self.tf_laser_obst.transform.rotation.w = q[3]  
    self.tf_bcaster.sendTransform(self.tf_laser_obst)
```

# A note on quaternions

Quaternions can be regarded as a 3D extension of complex numbers:

$$\mathbf{q} = x\mathbf{i} + y\mathbf{j} + z\mathbf{k} + w, \mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$$

Unit quaternions provide an alternative representation of 3D rotations, which is

- More powerful than Euler angles
- More compact than rotation matrices

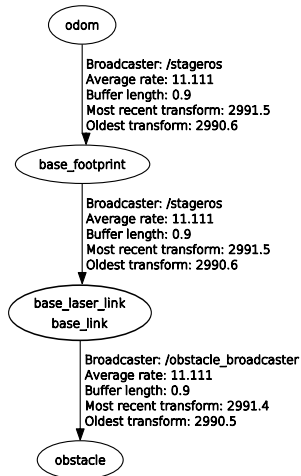
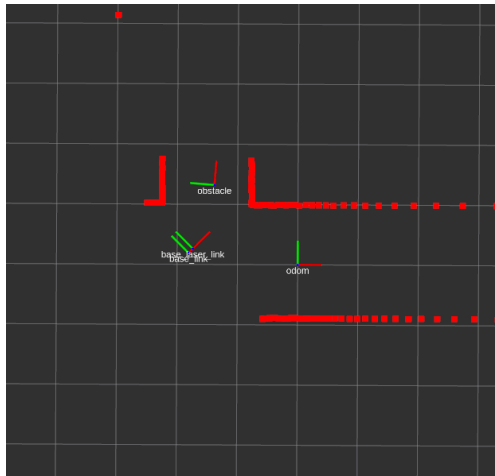
## Rotation in the plane as a quaternion

Rotation by  $\theta$  around the  $z$ -axis:

$$\mathbf{q}(\theta) = 0\mathbf{i} + 0\mathbf{j} + \sin\left(\frac{\theta}{2}\right)\mathbf{k} + \cos\left(\frac{\theta}{2}\right) \quad (1)$$

## Broadcasting a transform (RViz and TF tree)

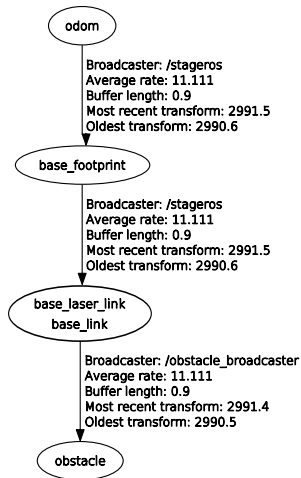
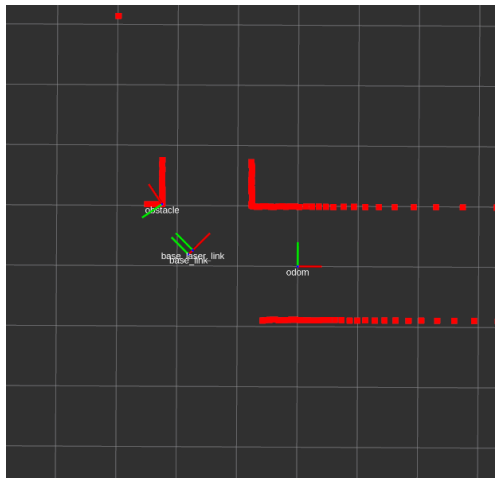
```
roslaunch rps_tf2_tutorial closest_obstacle.py scan:=base_scan
```



# Broadcasting the closest point in a laser scan

```
# ...  
from geometry_msgs.msg import Quaternion  
# ...  
def scan_callback(self, scan):  
    """ This is where all the action happens! """  
    self.tf_laser_obst.header.stamp = rospy.Time.now()  
    range_obs = min(scan.ranges)  
    idx_obs = scan.ranges.index(range_obs)  
    angle_obs = scan.angle_min + idx_obs*scan.angle_increment  
    trans = self.tf_laser_obst.transform.translation  
    trans.x = range_obs*cos(angle_obs)  
    trans.y = range_obs*sin(angle_obs)  
    q = transformations.quaternion_from_euler(0, 0, angle_obs)  
    self.tf_laser_obst.transform.rotation = Quaternion(*q)  
    self.tf_bcaster.sendTransform(self.tf_laser_obst)
```

# Broadcasting the closest point (RViz and TF tree)





# Listening to transforms (preparing the data)

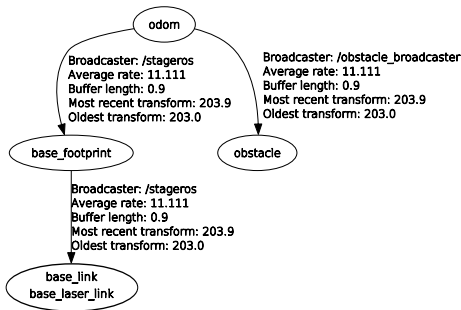
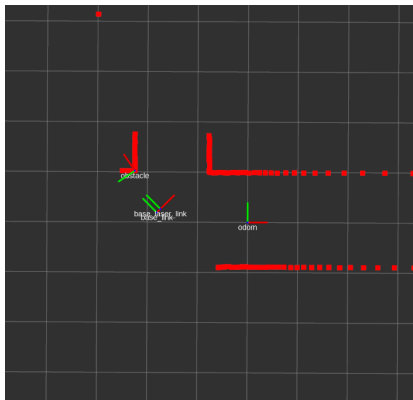
```
# ...  
from geometry_msgs.msg import PoseStamped  
# ...  
def __init__(self):  
    # ...  
    self.tf_buffer = tf2_ros.Buffer()  
    self.tf_listener = tf2_ros.TransformListener(self.tf_buffer)  
    # ...  
    self.tf_odom_obst.header.frame_id = 'odom'  
    # ...  
def scan_callback(self, scan):  
    pose_obs = PoseStamped()  
    pose_obs.pose.position.x = range_obs*cos(angle_obs)  
    pose_obs.pose.position.y = range_obs*sin(angle_obs)  
    q = transformations.quaternion_from_euler(0, 0, angle_obs)  
    pose_obs.pose.orientation = Quaternion(*q)
```

# Listening to transforms and transforming a pose

```
try:
    buf = self.tf_buffer
    tf_odom_laser = buf.lookup_transform('odom',
                                         'base_laser_link',
                                         rospy.Time())

    pose_odom_obs = do_transform_pose(pose_obs, tf_odom_laser)
    pose_odom_obs = pose_odom_obs.pose
    tf = self.tf_odom_obst.transform
    tf.translation = pose_odom_obs.position
    tf.rotation = pose_odom_obs.orientation
    self.tf_bcaster.sendTransform(self.tf_odom_obst)
except (tf2_ros.LookupException,
        tf2_ros.ConnectivityException,
        tf2_ros.ExtrapolationException) as ex:
    rospy.logwarn(ex)
```

# Broadcasting in the odom frame



The ROS package with the example code is available at the [pftros/rps\\_tf2\\_tutorial](#) GitHub repo.

How is TF2 actually implemented?

- The `/tf` and `/tf_static` topics
- The `geometry_msgs/TransformStamped` message
- The TF2 buffers in Python and C++ which perform the lookups and computations

# Recap: TF2 Python API

- TF2 provides tools for working with transforms in Python (and C++)
- Analogy with topics:
  - Publishing  $\leftrightarrow$  Broadcasting
  - Subscribing  $\leftrightarrow$  Listening
- The TF tree is **global**, frame names are not scoped

- Official TF2 package doc
- Official TF2 tutorials
- TF Quaternion tutorial (a bit outdated)
- A nice visualization of Euler angle shortcomings (Gimbal lock)
- A brief introduction to the PyKDL library (usful for coding coordinate transforms)