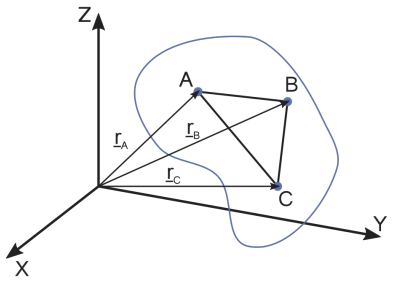
# 04. Principles of robotics. Programming a da Vinci surgical robot in simulated environment.

## Lecture

**Warning**

**Test 1** (ROS principles, publisher, subscriber. Python principles. Principles of robotics.) **October 27.**

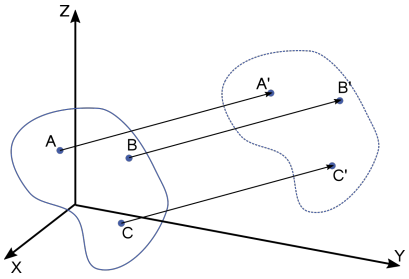
### Rigid body motion



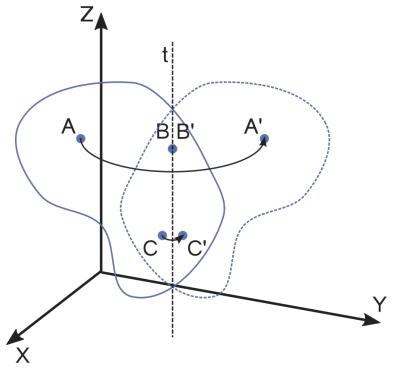

**Def. Rigid body**

A rigid body is defined as a body on which the distance between two points remains constant in time regardless of the force applied on it.

- Shape and the volume of the rigid bodies are also constant.
- The **pose** of a rigid body can be given by the three coordinates of three of its points that do not lie on the same straight line.

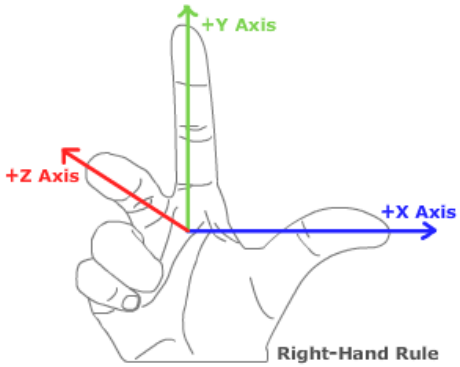



- The **pose** of a rigid body can be described in a more expressive way by the three coordinates of one of its points chosen arbitrarily **position** and the body's **orientation**.
- The **motion of rigid bodies** is composed by two elemental motions: **translation** and **rotation**.
- During **translation**, all points of the body move along straight, parallel lines.




- During **rotation**, the position of the points of the rotational axis are constant, and the other points of the body move along circles in planes perpendicular to the axis of rotation.
- The **free motion** of rigid bodies can always be expressed as the superposition of a translational motion and a rotation around a single axis.

# 3D transformations

- 


  **Position:** 3D offset vector
- **Orientation:** 3 x 3 rotation matrix
  - further orientation representations: Euler-angles, RPY, angle axis, quaternion
- **Pose**: 4 × 4 (homogenous) transformation matrix
- **Frame**: origin, 3 axes, 3 base vectors, right hand rule
- **Homogenous transformation:** rotation and translation in one transfromation
  - e.g., for the rotation $\mathbf{R}$ and translation $\mathbf{v}$:

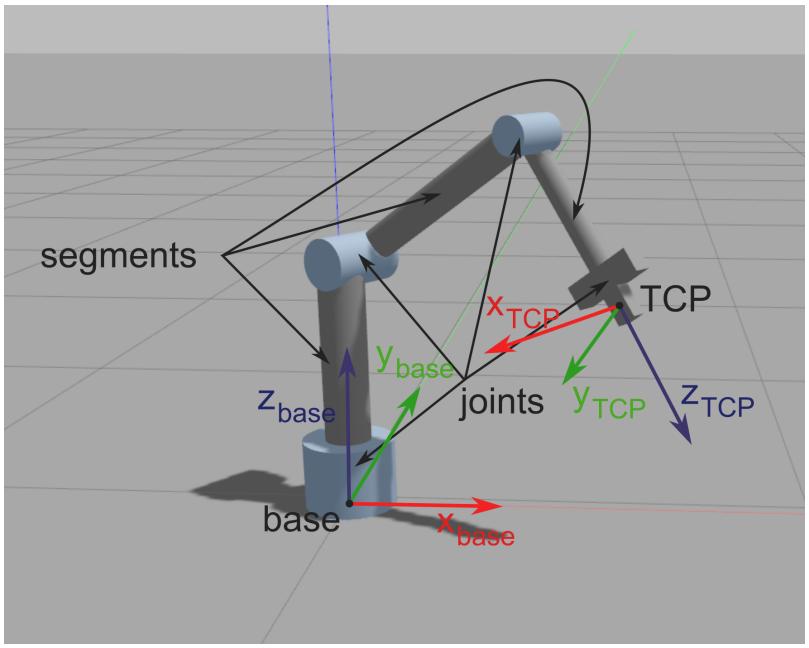
$$ \mathbf{T} = \left[\matrix{\mathbf{R} & \mathbf{v}\\\mathbf{0} & 1 }\right] = \left[\matrix{r_{1,1} & r_{1,2} & r_{1,3} & v_x\\r_{2,1} & r_{2,2} & r_{2,3} & v_y\\r_{3,1} & r_{3,2} & r_{3,3} & v_z\\\ 0 & 0 & 0 & 1 }\right] $$

- **Homogenous coordinates:**
  - **Vector:** extended with 0, $\mathbf{a_H}=\left[\matrix{\mathbf{a} \\ 0}\right]=\left[\matrix{a_x \\ a_y \\ a_z \\ 0}\right]$
  - **Point:** extended by 1, $\mathbf{p_H}=\left[\matrix{\mathbf{p} \\ 1}\right]=\left[\matrix{p_x \\ p_y \\ p_z \\ 1}\right]$
  - Applying transfomrations is much easier:

$$ \mathbf{q} = \mathbf{R}\mathbf{p} + \mathbf{v} \to \left[\matrix{\mathbf{q} \\ 1}\right] = \left[\matrix{\mathbf{R} & \mathbf{v}\\\mathbf{0} & 1 }\right] \left[\matrix{\mathbf{p} \\ 1}\right] $$

- **Degrees of Freedom** (DoF): the number of independent parameters.

## Principles of robotics




- Robots are built of: **segments** (or links) és **joints**
- **Task space** (or cartesian space):
  - 3D space around us, where the task, endpoint trajectories, obstacles are defined.
  - **TCP** (Tool Center Point): Frame fixed to the end effector of the robot.
  - **Base frame**, **world frame**
- **Joint space**:
  - Properties or values regarding the joints.
  - Low-level controller.
  - Joint angles, joint velocities, accelerations, torques....

## Python libraries

### Numpy

- Python library
- High dimension arrays and matrices

- Mathematical functions

```
import numpy as np

# Creating ndarrays
a = np.zeros(3)
a.shape
a.shape=(3,1)
a = np.ones(5)
a = np.empty(10)
l = np.linspace(5, 10, 6)
r = np.array([1,2])    # ndarray from python list
r = np.array([[1,2],[3,4]])
type(r)

# Indexing
l[0]
l[0:2]
l[-1]
r[:,0]

# Operations on ndarrays
r_sin = np.sin(r)
np.max(r)
np.min(r)
np.sum(r)
np.mean(r)
np.std(r)

l < 7
l[l < 7]
np.where(l < 7)

p = np.linspace(1, 5, 6)
q = np.linspace(10, 14, 6)

s = p + q
s = p * q
s = p * 10
s = p + 10
s = p @ q    # dot product
s = r.T
```

If not installed:

```
pip3 install numpy
```

**Matplotlib**

- Visualization in python
- Syntax similar to Matlab

```
import numpy as np
from matplotlib import pyplot as plt

X = np.linspace(-np.pi, np.pi, 256)
C, S = np.cos(X), np.sin(X)

plt.plot(X, C)
plt.plot(X, S)

plt.show()
```

If not installed:

```
pip3 install matplotlib
```

# Practice

## 1. dVRK ROS 2 install

The da Vinci Surgical System is used to perform minimally invasive surgeries by teleoperation. The da Vinci Research Kit (DVRK) is an open-source hardware and software platform, offers, amongst others, reading and writing all the joints of the da Vinci, and also simulators for each arm. The DVRK software can be built as follows.

1. Clone the dVRK (da Vinci Reserach Kit) using `vcs` into a new workspace, then build it:

   ```
   mkdir -p ~/dvrk2_ws/src
   cd ~/dvrk2_ws/src
   vcs import --input https://raw.githubusercontent.com/jhu-dvrk/dvrk_robot_ros2/main/dvrk.vcs --recursive
   cd ~/dvrk2_ws
   colcon build --symlink-install --cmake-args -DCMAKE_BUILD_TYPE=Release
   source ~/dvrk2_ws/install/setup.bash
   ```

2. Add the following line to the end of the `.bashrc` file:

   ```
   source ~/dvrk2_ws/install/setup.bash
   ```

3. Run these commands in separate terminals to launch the simulation. Do not forget to push the Home button in the DVRK console.

```
# dVRK main console
ros2 run dvrk_robot dvrk_console_json -j ~/dvrk2_ws/install/sawIntuitiveResearchKitAll/
share/sawIntuitiveResearchKit/share/console/console-PSM1_KIN_SIMULATED.json
```

```
# ROS 2 joint and robot state publishers
ros2 launch dvrk_model dvrk_state_publisher.launch.py arm:=PSM1
```

```
# RViz
ros2 run rviz2 rviz2 -d ~/dvrk2_ws/install/dvrk_model/share/dvrk_model/rviz/PSM1.rviz
```

```
# rqt_gui
ros2 run rqt_gui rqt_gui
```

## 2. PSM subscriber

1. Create a new file named `psm_grasp.py` in the `~/ros2_ws/src/ros2_course/ros2_course` folder. Add the new entry point to the `setup.py`, as usually.

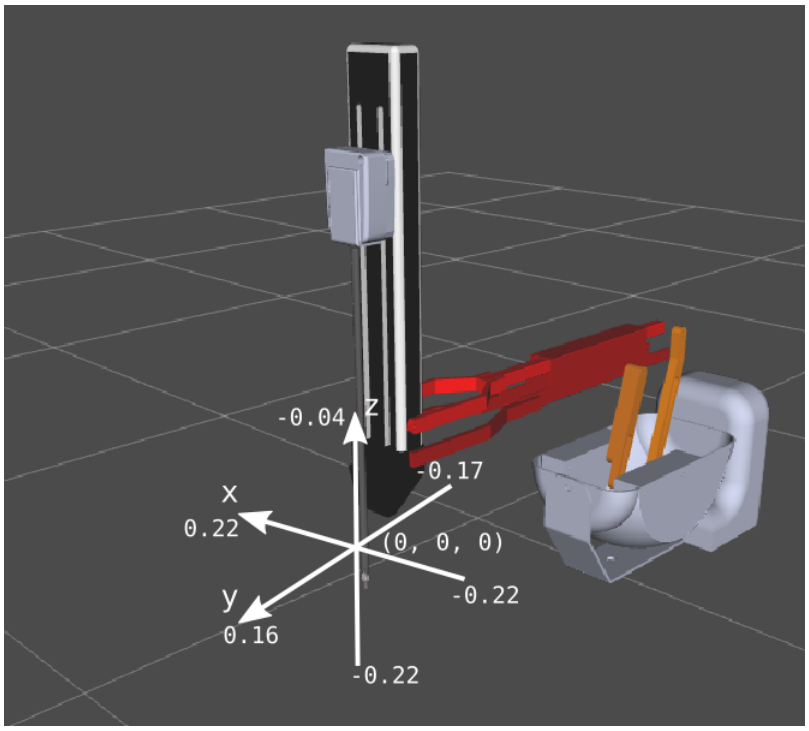
2. Check the topics and nodes of the simulator using the commands learned earlier (`rostopic list`, `rosrun rqt_graph rqt_graph`, etc.). PSM1 publishes the pose of the TCP and the angle of the jaws into the topics below. Subscribe to these topic in `psm_grasp.py` and store the current values into variables.

```
/PSM1/measured_cp
/PSM1/jaw/measured_js
```

3. Build and run the node:

```
cd ~/ros2_ws
colcon build --symlink-install
ros2 run ros2_course psm_grasp
```

## 3. Move the TCP along a linear trajectory




1. PSM1 expects commands regarding the pose of the TCP and the angle of the jaws from the topics below. Create publishers to these topic in `psm_grasp.py`.

```
/PSM1/servo_cp
/PSM1/jaw/servo_jp
```

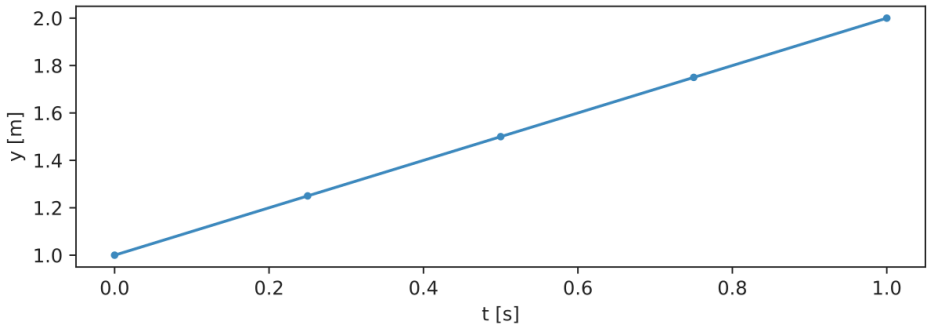
2. Implement a method that moves the TCP to the desired position along a linear trajectory. Send the gripper to the position (0.0, 0.05, -0.12), leave the orientation as it is. Let the sampling time **dt** be 0.01s.

```
def move_tcp_to(self, target, v, dt):
```

> **Tip**
>
> Use the function `np.linspace(start, stop, num)` to create the array of **t** values (T). This function can also be used to create the linear trajectory along the axes **x, y, z** in separate arrays X, Y and Z.

3. Write a method that can open and close the gripper jaws, also along a linear trajectory.

```
def move_jaw_to(self, target, omega, dt):
```




## 4. Dummy marker

1. Write a node that creates a virtual marker that can be grasped publishing `visualization_msgs/Marker` messages. Create a new file named `dummy_marker.py` in the `~/ros2_ws/src/ros2_course/ros2_course` folder. Add it to the `setup.py`, as usually. Copy the following code into the file `dummy_marker.py`:

```
import rclpy
from rclpy.node import Node
from visualization_msgs.msg import Marker

class DummyMarker(Node):
    def __init__(self, position):
        super().__init__('minimal_publisher')
        self.position = position
        self.publisher_ = self.create_publisher(Marker, 'dummy_target_marker', 10)
        timer_period = 0.1  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.i = 0
        i = 0

    def timer_callback(self):
        marker = Marker()
        marker.header.frame_id = 'PSM1_psm_base_link'
        marker.header.stamp = self.get_clock().now().to_msg()
        marker.ns = "dvrk_viz"
        marker.id = self.i
        marker.type = Marker.SPHERE
        marker.action = Marker.MODIFY
```

```
        marker.pose.position.x = self.position[0]
        marker.pose.position.y = self.position[1]
        marker.pose.position.z = self.position[2]
        marker.pose.orientation.x = 0.0
        marker.pose.orientation.y = 0.0
        marker.pose.orientation.z = 0.0
        marker.pose.orientation.w = 1.0
        marker.scale.x = 0.008
        marker.scale.y = 0.008
        marker.scale.z = 0.008
        marker.color.a = 1.0 # Don't forget to set the alpha!
        marker.color.r = 0.0
        marker.color.g = 1.0
        marker.color.b = 0.0;

        self.publisher_.publish(marker)
        self.i += 1


def main(args=None):
    rclpy.init(args=args)
    marker_publisher = DummyMarker([-0.05, 0.08, -0.12])
    rclpy.spin(marker_publisher)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    marker_publisher.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

2. Build and run the node. Visualize the marker in RViz.

## 5. Grasp the marker

1. Subscribe to the topic with the marker position `dummy_target_publisher` the file `psm_grasp.py` .

2. Implement a method in `psm_grasp.py` to grasp the generated marker with PSM1.

**Note**

Some values tends to stuck in the simulator. Thus, at the beginning of the program, it is a good idea to reset the arm:

```
#Reset the arm
psm.move_tcp_to([0.0, 0.0, -0.12], 0.01, 0.01)
psm.move_jaw_to(0.0, 0.1, 0.01)
```

## Links

- Download and compile dVRK 2
- Marker examples
- Numpy vector magnitude
- Numpy linspace