Aalto University
School of Electrical
Engineering

Introduction to Robotic Manipulation in ROS

Francesco Verdoja

Dept. of Electrical Engineering and Automation
francesco.verdoja@aalto.fi

ELEC-E8126 - Robotic manipulation

Introduction

Where to find course information?

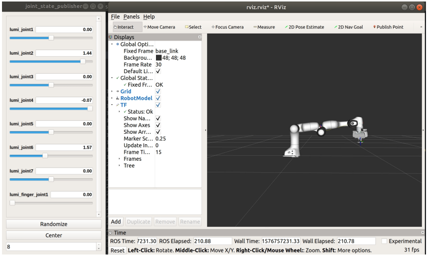
- Mycourses page: ELEC-E8126 - Robotic manipulation
- First lecture by Prof. Kyrki (Monday 13/1)
- If not found: contacting Prof. Kyrki or the TAs

Today

- Topics:
  - Coordinate frames and transforms
  - KDL representations
  - Forward and inverse kinematics
  - Velocity kinematics (Jacobian)
  - Dynamics
- Introduction to ROS:
  - nodes
  - publisher/subscriber
  - message types
  - some useful libraries

The Robot Operating System (ROS)

- Robotic middleware
- De-facto standard for robotic research
- Main features:
  - Open-source
  - Decentralized architecture
  - Asyncronous communication
  - Visualization and simulation tools



Publisher

```python
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def publisher():
    pub = rospy.Publisher('welcome', String, queue_size=10)
    rospy.init_node('publisher', anonymous=True)
    rate = rospy.Rate(10)
    while not rospy.is_shutdown():
        msg = "Welcome to Robotic Manipulation course %s" % rospy.get_time()
        pub.publish(msg)
        rate.sleep()

if __name__ == '__main__':
    try:
        publisher()
    except rospy.ROSInterruptException:
        pass
```

Subscriber

```python
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(data.data)

def listener():
    rospy.init_node('subscriber', anonymous=True)
    rospy.Subscriber("welcome", String, callback)
    rospy.spin()

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

Demo time

Coordinate frames and transforms

The Lie group $SE(3)$

- Three-dimensional Special Euclidean group:

$$SE(3) = \left\{ \mathbf{A} \mid \mathbf{A} = \left[\begin{array}{c:c} \mathbf{R} & \mathbf{r} \\ \hdashline \mathbf{0}^{1\times 3} & 1 \end{array} \right], \mathbf{R} \in SO(3), \mathbf{r} \in \mathbb{R}^3 \right\}$$

- **R** represents rotation/orientation
- **r** represents translation

Orientation representations

$SO(3)$ Three-dimensional Special Orthogonal group:

$$SO(3) = \left\{\mathbf{R} \mid \mathbf{R} \in \mathbb{R}^{3\times3}, \mathbf{R}^T\mathbf{R} = \mathbf{R}\mathbf{R}^T = \mathbf{I}, \det(\mathbf{R}) = 1\right\}$$

Euler angles Vector representanting rotation angle in each direction

Axis-angle An orientation vector $\vec{u} = (u_x, u_y, u_z)$ and an angle value θ

Quaternions 4-dimensional complex number $w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$ embedding a 3D orientation $\mathbf{q}$:

$$\mathbf{q} = \exp^{\frac{\theta}{2}(u_x\mathbf{i}+u_y\mathbf{j}+u_z\mathbf{k})} = \cos\frac{\theta}{2} + (u_x\mathbf{i} + u_y\mathbf{j} + u_z\mathbf{k})\sin\frac{\theta}{2}$$

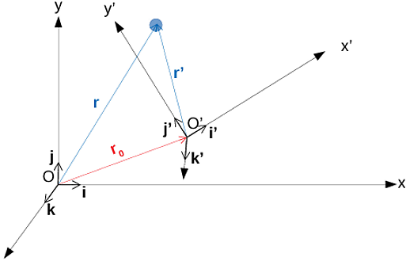
Coordinate transformation

Transformation from O' to O

$$\begin{bmatrix} \mathbf{r} \\ \hdashline 1 \end{bmatrix} = \left[\begin{array}{c:c} {}^{O}\mathbf{R}_{O'} & {}^{O}\mathbf{r}^{O'} \\ \hdashline \mathbf{0}^{1\times 3} & 1 \end{array}\right] \begin{bmatrix} \mathbf{r}' \\ \hdashline 1 \end{bmatrix} \quad \Rightarrow \quad \mathbf{r} = {}^{O}\mathbf{R}_{O'}\mathbf{r}' + {}^{O}\mathbf{r}^{O'}$$

Homogeneous transformation

$${}^{A}\mathbf{A}_{B} = \left[\begin{array}{c:c} {}^{A}\mathbf{R}_{B} & {}^{A}\mathbf{r}^{B} \\ \hdashline \mathbf{0}^{1\times 3} & 1 \end{array}\right]$$



ROS: imports

```
import tf2_ros
import geometry_msgs.msg
from tf.transformations import quaternion_from_euler
from tf.transformations import euler_from_quaternion
from tf.transformations import quaternion_matrix
```

First, we need to import a couple of ROS libraries:

- `tf2_ros` ROS bindings to transform library
- `geometry_msgs.mgs` messages for geometric primitives
- `tf.transformations` transformations between rotation representations

ROS: initialization

```python
if __name__ == '__main__':
    rospy.init_node('tf2_intro')

    tfBuffer = tf2_ros.Buffer()
    listener = tf2_ros.TransformListener(tfBuffer)
    chatter_pub = rospy.Publisher('/tf_pose', ↩
         geometry_msgs.msg.TransformStamped, queue_size=1)
    ee_to_base_pose = geometry_msgs.msg.TransformStamped()
```

- Similar initialization to the simple publisher example
- We will be publishing the end-effector pose to the `tf_pose` topic

ROS: loop

```python
    while not rospy.is_shutdown():
        try:
            ee_to_base_pose = tfBuffer.lookup_transform("base_link", "lumi_ee", rospy.Time())
        except (tf2_ros.LookupException, tf2_ros.ConnectivityException, tf2_ros.ExtrapolationException):
            rate.sleep()
            continue
        ee_to_base_translation = ee_to_base_pose.transform.translation
        ee_to_base_rotation = ee_to_base_pose.transform.rotation
```

- We ask for the transformation from robot base to end-effector at current time

ROS: loop (cont.)

```
        ee_to_base_rotation_quaternion = (ee_to_base_rotation.x, ee_to_base_rotation.y, ee_to_base_rotation.z, ee_to_base_rotation.w)
        ee_to_base_rotation_euler = euler_from_quaternion(ee_to_base_rotation_quaternion)
        ee_to_base_rotation_RotationMatrix = quaternion_matrix(ee_to_base_rotation_quaternion)
```

```
        print("\n *** 3. Orientation (Euler in radian) *** \n")
        print(ee_to_base_rotation_euler)
        print("\n *** 4. Orientation (Rotation Matrix) *** \n")
        print(ee_to_base_rotation_RotationMatrix)
        chatter_pub.publish(ee_to_base_pose)
        rate.sleep()
```

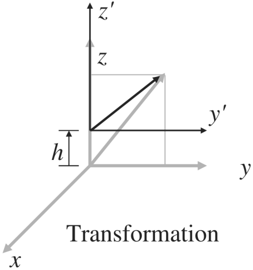
- We perform some conversions
- We print out the the end-effector rotation in different formats
- Finally, we publish the pose

Demo time

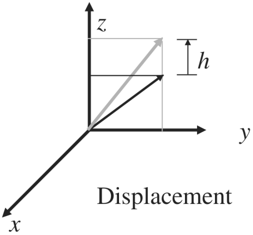
Translation

Translation along the z-axis through h

$$Trans(z,h) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & h \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & h \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}$$



Transformation

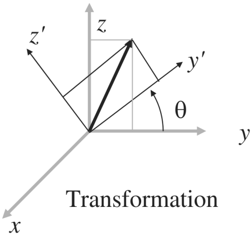


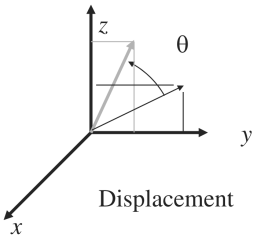
Displacement

Rotation

Rotation along the x-axis through θ

$$Rot(x,\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$





Rotation (cont.)

Rotation along the *y*-axis through θ

$$Rot(y,\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

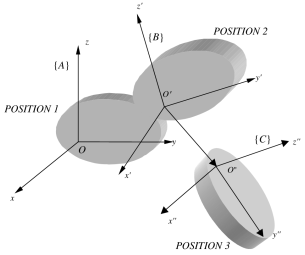
Rotation along the *z*-axis through θ

$$Rot(z,\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Composition

Displacement from $\{A\}$ to $\{C\}$

$$
\begin{aligned}
{}^{A}\mathbf{A}_{C} &= {}^{A}\mathbf{A}_{B}{}^{B}\mathbf{A}_{C} \\
&= \left[\begin{array}{c:c} {}^{A}\mathbf{R}_{B} & {}^{A}\mathbf{r}^{B} \\ \hdashline \mathbf{0}^{1\times 3} & 1 \end{array}\right] \times \left[\begin{array}{c:c} {}^{B}\mathbf{R}_{C} & {}^{B}\mathbf{r}^{C} \\ \hdashline \mathbf{0}^{1\times 3} & 1 \end{array}\right] \\
&= \left[\begin{array}{c:c} {}^{A}\mathbf{R}_{B} \times {}^{B}\mathbf{R}_{C} & {}^{A}\mathbf{R}_{B} \times {}^{B}\mathbf{r}^{C} + {}^{A}\mathbf{r}^{B} \\ \hdashline \mathbf{0}^{1\times 3} & 1 \end{array}\right]
\end{aligned}
$$



Inversion

$$\mathbf{A} = \begin{bmatrix} \mathbf{R} & \mathbf{r} \\ \mathbf{0}^{1\times 3} & 1 \end{bmatrix} \quad \text{where} \quad \mathbf{R}\mathbf{R}^T = \mathbf{R}^T\mathbf{R} = \mathbf{I}$$

Therefore:

$$\begin{aligned} \mathbf{p} &= \mathbf{A}\mathbf{p}' \\ &= \mathbf{R}\mathbf{p}' + \mathbf{r} \end{aligned}$$

$$\begin{aligned} \mathbf{p}' &= \mathbf{R}^T(\mathbf{p} - \mathbf{r}) \\ &= \mathbf{R}^T\mathbf{p} - \mathbf{R}^T\mathbf{r} \end{aligned}$$

From which:

$$\mathbf{A}^{-1} = \begin{bmatrix} \mathbf{R}^T & -\mathbf{R}^T\mathbf{r} \\ \mathbf{0}^{1\times 3} & 1 \end{bmatrix}$$

ROS: composition

```
    try:
        ee_to_base_pose = tfBuffer.lookup_transform("base_link", "lumi_ee", rospy.Time())
        ee_to_joint6_pose = tfBuffer.lookup_transform("base_link", "lumi_link6", rospy.Time())
        joint6_to_base_pose = tfBuffer.lookup_transform("lumi_link6", "lumi_ee", rospy.Time())
    except (tf2_ros.LookupException, tf2_ros.ConnectivityException, tf2_ros.ExtrapolationException):
        rate.sleep()
        continue

    ee_to_base_M = transform_to_matrix(ee_to_base_pose.transform)
    ee_to_joint6_M = transform_to_matrix(ee_to_joint6_pose.transform)
    joint6_to_base_M = transform_to_matrix(joint6_to_base_pose.transform)

    ee_to_base_comp_M = np.dot(ee_to_joint6_M, joint6_to_base_M)
```

Output

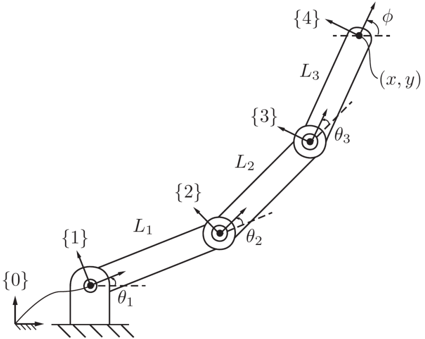
```
 *** ee_to_joint6 (tf) ***

[[ 0.43605542  0.02259849 -0.89963603 -0.29086978]
 [-0.62006018  0.73206128 -0.28215536  0.31070571]
 [ 0.65221242  0.68086385  0.33323171  0.74334375]
 [ 0.          0.          0.          1.        ]]

 *** joint6_to_base (tf) ***

[[ 9.36074581e-01  3.51801620e-01 -8.32667268e-17  8.80000000e-02]
 [-1.72270531e-12  4.58338922e-12 -1.00000000e+00 -2.23000000e-01]
 [-3.51801620e-01  9.36074581e-01  4.89644436e-12  1.09195986e-12]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  1.00000000e+00]]

 *** ee_to_base (tf) ***

[[ 0.72467381 -0.68872141 -0.02259849 -0.25753637]
 [-0.48115986 -0.48225664 -0.73206128  0.09289074]
 [ 0.49328801  0.54137912 -0.68086385  0.6489058 ]
 [ 0.          0.          0.          1.        ]]

 *** ee_to_base (compositon) ***

[[ 0.72467381 -0.68872141 -0.02259849 -0.25753637]
 [-0.48115986 -0.48225664 -0.73206128  0.09289074]
 [ 0.49328801  0.54137912 -0.68086385  0.6489058 ]
 [ 0.          0.          0.          1.        ]]
```

Kinematics

Forward kinematics

- Computing the transformation of the end-effector T from the joint parameters
- For a robot chain with n joints:

$$^0T_n = {}^0T_1 {}^1T_2 {}^2T_3 \dots {}^{n-1}T_n$$



Question time

1. Is there always a solution for forward kinematics?
2. Is the solution unique?

ROS: importing the robot model

```python
import kdl_parser_py.urdf as kdl_parser
import PyKDL as kdl
```

```python
(status, tree) = kdl_parser.treeFromFile("/path/to/models/panda.urdf")
print("\n *** Successfully parsed urdf file and constructed kdl tree *** \n" if status else "Failed to parse urdf file to kdl tree")
chain = tree.getChain("base_link", "lumi_ee")
num_joints = chain.getNrOfJoints()
print("\n*** This robot has %s joints  *** \n" % num_joints)
```

- The Panda robot URDF model is loaded by the `kdl_parser`
- From it, we get the full chain and the number of joints

ROS: forward kinematics

```python
fk_pos_solver = kdl.ChainFkSolverPos_recursive(chain)
ee_pose = kdl.Frame()
theta = create_joint_angles([0, 0, 0, -1.57, 0, 1.57, 0])
fk_pos_solver.JntToCart(theta, ee_pose)
print('\n*** End-effector Position FK: ***')
print(ee_pose.p)
print("\n*** Rotational Matrix FK: ***")
print(ee_pose.M)
```

- We load the chain in the solver
- And then ask for the end-effector pose, given a set of joint angles θ

Output

```
*** End-effector Position FK: ***
[    0.554434,-3.32573e-12,    0.508806]

*** Rotational Matrix FK: ***
[    0.707107,    0.707107,           0;
     0.707107,   -0.707107,-9.79318e-12;
 -6.92482e-12, 6.92482e-12,          -1]
```

Velocity kinematics

Forward kinematics:

$$x(t) = f(\theta(t))$$

where:

- $x \in \mathbb{R}^m$ is the end-effector cartesian configuration
- $\theta \in \mathbb{R}^n$ is a set of joint variables

Velocity $\dot{x} = dx/dt \in \mathbb{R}^m$ is given by:

$$\dot{x} = \frac{\partial f(\theta)}{\partial \theta}\dot{\theta} = J(\theta)\dot{\theta}$$

$J(\theta) \in \mathbb{R}^{m \times n}$ is called the Jacobian matrix

Question time

1. Is there always a solution for velocity kinematics?
2. Is the solution unique?

ROS: velocity kinematics

```python
# Velocity FK with KDL solver
fk_vel_solver = kdl.ChainFkSolverVel_recursive(chain)
twist = kdl.FrameVel()
theta_dot = create_joint_angles([0, 0, 0, 0.5, 0, 0, 0])
fk_vel_solver.JntToCart(kdl.JntArrayVel(theta, theta_dot), twist)
print("\n*** End-effector Velocity FK by KDL solver: ***")
print(twist.deriv())
# Velocity FK with Jacobian
jacobian_solver = kdl.ChainJntToJacSolver(chain)
J = kdl.Jacobian(num_joints)
jacobian_solver.JntToJac(theta,J)
J = kdl_to_mat(J)
theta_dot_array = np.array([0,0,0,0.5,0,0,0])
print("\n*** End-effector Velocity FK by mapping Jacbian: ***")
print(np.dot(J ,theta_dot_array))
```

- Similar structure, different solver
- Note we are providing also $\dot{\theta}$

Output

```
*** End-effector Velocity FK by KDL solver: ***
[   0.0700971, 1.15543e-12,    0.235967,           0,        -0.5, 2.44829e-12]

*** End-effector Velocity FK by mapping Jacbian: ***
[[ 7.00971184e-02  1.15543383e-12  2.35967091e-01  0.00000000e+00
  -5.00000000e-01  2.44829443e-12]]
```

Inverse kinematics

Forward kinematics: $x(t) = f(\theta(t))$

f^{-1}?

Question time

1. Is there always a solution for inverse kinematics?
2. Is the solution unique?

ROS: inverse kinematics

```
ik_vel_solver = kdl.ChainIkSolverVel_pinv(chain)
theta_min = kdl.JntArray(num_joints)
theta_max = kdl.JntArray(num_joints)
theta_init = kdl.JntArray(num_joints)
theta_out = kdl.JntArray(num_joints)
theta_init = theta
theta_min[0] = theta_min[2] = theta_min[4] = theta_min[6] = -2.9
theta_min[1] = -1.76
theta_min[3] = -3.07
theta_min[5] = -0.02
theta_max[0] = theta_max[2] = theta_max[4] = theta_max[6] = 2.9
theta_max[1] = 1.76
theta_max[3] = -0.07
theta_max[5] = 3.75
```

- Set joints angle limits

ROS: inverse kinematics (cont.)

```python
ik_pos_solver = kdl.ChainIkSolverPos_NR_JL(chain, theta_min, theta_max, fk_pos_solver, ik_vel_solver)
desired_position = kdl.Frame(ee_pose.M, ee_pose.p)
ik_pos_solver.CartToJnt(theta_init, desired_position, theta_out)
print("\n*** Calcualted Joint angles IK: ***")
print(theta_out)
```

- Use the solver to compute a joint configuration corresponding to the desired end-effector pose

Output

```
*** Calcualted Joint angles IK: ***
    0
    0
    0
-1.57
    0
 1.57
    0
```

Inverse velocity kinematics

$$J^{-1}?$$

Question time

1. Is there always a solution for inverse velocity kinematics?
2. Is the solution unique?

ROS: inverse velocity kinematics

```
theta_dot_out = kdl.JntArray(num_joints)
ik_vel_solver.CartToJnt(theta, twist.deriv(), theta_dot_out)
print("\n*** Calculated Joint Velocity IK by KDL solver: ***")
print(theta_dot_out)

desired_twist = np.array([0.0700971, 1.15543e-12, 0.235967, 0, -0.5, 2.44829e-12])
J_pinv = np.dot(np.linalg.inv(np.dot(J, J.transpose())), J)
print("\n*** Calculated Joint Velocity IK by mapping Jacobian: ***")
print(np.dot(desired_twist, J_pinv))
```

Output

```
*** Calculated Joint Velocity IK by KDL solver: ***
 1.57602e-12
-1.66533e-16
-1.57607e-12
        0.5
 1.30289e-16
 2.77556e-16
-5.29257e-18

*** Calculated Joint Velocity IK by mapping Jacobian: ***
[[ 1.57592614e-12 -1.27096013e-07 -1.57593100e-12  4.99999609e-01
   5.75310613e-28  2.63408470e-07 -1.67265687e-18]]
```

Dynamics

Equations of motion

$$\tau = M(\theta)\ddot{\theta} + c(\theta, \dot{\theta}) + g(\theta)$$

where:

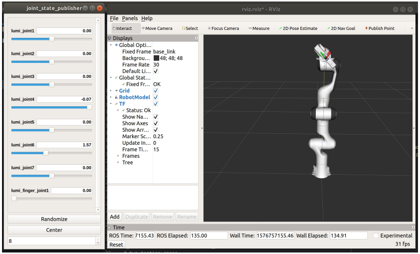
- $M(\theta)$: mass matrix (symmetric positive-definite)
- $c(\theta, \dot{\theta})$: Coriolis and centripetal torques
- $g(\theta)$: gravitational torques

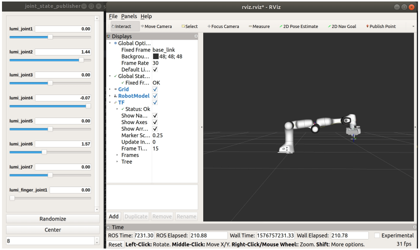
ROS: Gravity in different configurations

```python
dynamic_param_solver = kdl.ChainDynParam(chain, kdl.Vector(0, 0, -9.8))
G_vertical = kdl.JntArray(num_joints)
G_horizontal = kdl.JntArray(num_joints)
# two testing cases
theta_vertical = create_joint_angles([0, 0, 0, -0.07, 0, 1.57, 0])
theta_horizontal = create_joint_angles([0, 1.44, 0, -0.07, 0, 1.57, 0])

dynamic_param_solver.JntToGravity(theta_vertical, G_vertical)
print("\n*** Gravity when the robot is in vertical position: ***")
print(G_vertical)
dynamic_param_solver.JntToGravity(theta_horizontal, G_horizontal)
print("\n*** Gravity when the robot is in horizontal position: ***")
print(G_horizontal)
```

Question time

In which configuration is the gravity term bigger?





Output

```
*** Gravity when the robot is in vertical position: ***
-1.3531e-16
    -4.3625
-1.3531e-16
     -2.534
8.06885e-14
   0.780196
          0

*** Gravity when the robot is in horizontal position: ***
-6.43082e-19
    -58.0259
 1.32433e-10
      23.629
 4.07107e-12
     1.01552
           0
```

Dynamics

```
# Initialize dynamics parameters
M = kdl.JntSpaceInertiaMatrix(num_joints)
C = kdl.JntArray(num_joints)
G = kdl.JntArray(num_joints)
```

```
# Calculate dynamics parameter and convert them to matrix to manipulate
dynamic_param_solver.JntToMass(theta_init, M)
dynamic_param_solver.JntToCoriolis(theta_init, theta_dot_init, C)
dynamic_param_solver.JntToGravity(theta_init, G)
```

```
# Calculate the required torque from dynamics equation
tau = np.dot(M, theta_dotdot_goal) + C + G
print("\n*** Calculated torque: ***")
print(tau)
```

Output

```
*** Calculated torque: ***
[[  0.        ]
 [-29.8135194 ]
 [  0.        ]
 [ 23.33412024]
 [ -0.        ]
 [  0.76773177]
 [ -0.        ]]
```

Today’s takeaways

Questions?