CS449/549: Learning for Robotics
Bilkent University
Fall 2024

RAI Tutorial 1

1 Installation of RAI with Python virtual environment

- Create a new folder for your RAI.
- Create a virtual environment.
  ```
  python3 -m venv rai_venv
  ```
- Activate the virtual environment
  ```
  source rai_venv/bin/activate
  ```
- Install the required packages
  ```
  sudo apt install liblapack3 freeglut3 libglew-dev python3 python3-pip
  pip install numpy scipy
  pip install robotic
  ```
- Tests
  ```
  ry-info
  python3 -c 'import robotic as ry; ry.test.RndScene()'
  ```

You can also visit `https://github.com/MarcToussaint/robotic` for more details on the installation.

2 Introduction

In RAI, the **Configuration** is a central concept representing the complete state of the robotic environment, encompassing both the robot and any objects in the world. It serves as a container for all relevant simulation information, including the positions, orientations, and kinematic relationships of objects and robot links.
Another key term is **Frame**. A Frame represents a reference point or a rigid body in the environment. It defines the position, orientation, and kinematic relationships of objects or robot components. Frames can have shapes (such as boxes or spheres) attached to them and are connected by joints or constraints, forming a kinematic tree.

- Let's begin by building a configuration from scratch, adding some frames, and setting their properties. You can create a new .ipynb file, add the code segments in separate cells, and run them, or use the provided .ipynb file that contains the code.
- Add the necessary imports at the beginning, create our configuration, and add the first frame. We will define the frame as a `marker`, which is not a physical shape but displays the base vectors for visualization. Later, you can use this marker for sub-goals.

```
import robotic as ry

C = ry.Config()

f = C.addFrame(name="first")
f.setShape(type=ry.ST.marker, size=[.3])
f.setPosition([0.0 ,0.0 , 0.5])
f.setQuaternion([1.0, 0.3, 0.0, .0])
print("frame name:", f.name)
print("pos:", f.getPosition())
print("quat:", f.getQuaternion())
C.view()
```

C.view() opens a view window. You can right-click on the window bar and select "Always on Top."

- For more details about configurations, basic shapes, frames, and joint states, you can check `https://github.com/MarcToussaint/rai-tutorials/blob/main/config_1_intro.ipynb`

3 Adding a Box to the Environment

- Let's clear the configuration and add a new box object.

```
C.clear()
C.addFrame("box1") \
    .setPosition([0, 0, .25]) \
    .setShape(ry.ST.ssBox, size=[.5, .5, .5, .05]) \
    .setColor([1, .5, 0]) \
    .setMass(.1) \
    .setContact(True)

C.view()
```

4 Relative Positions

Let's examine the relative positions of the objects.

- Add a second box 1 meter above the first one.

```
box2 = C.addFrame(name="box2", parent="box1")
box2.setShape(ry.ST.ssBox, size=[.5, .5, .5, .05])
box2.setRelativePosition([0,0,1])
box2.setColor([0,0,1])
```

- Now, display the position and orientation of the second box.

```
f = C.getFrame("box2")
print("position:", f.getPosition())      # position: [0.   0.   1.25]
print("orientation:", f.getQuaternion()) # orientation: [1. 0. 0. 0.]
```

- Why is it not [0.0, 0.0, 1.0]?

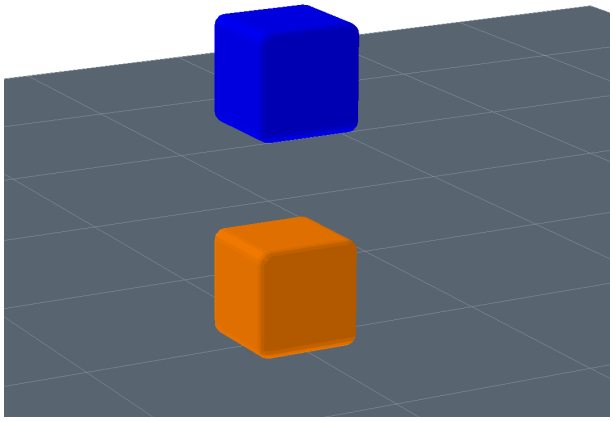
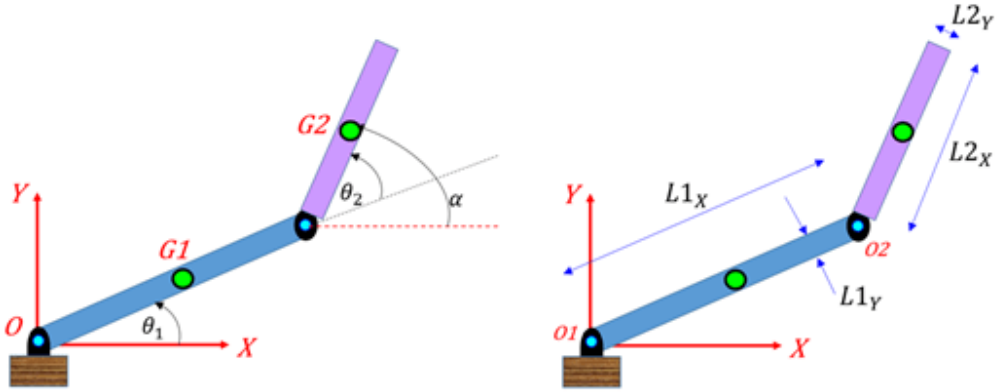


Figure 1: Relative Positions

- When we used **relative position**, we mean the position of a frame with respect to its reference (parent) frame in the environment. Instead of specifying the frame's position and orientation in absolute world coordinates, we define them relative to the parent frame. As a result, the position values will reflect this relationship, so you won't see fixed coordinates like [0, 0, 1].

5 Creating Two Link Manipulator

In this section, we will create a two-link planar manipulator system.



- One of the simplest ways to add frames is by using **.g** files. In this example, we will define our manipulator in a **.g** file, though you can also create it directly using Python commands.

- First, let's create the world and a sphere representing the hinge joint on the [0,0,0].

```
world: {}

zero (world): {
    shape: sphere, size: [0.05],
    mass: 1.0,
    X:"T t(0 0 0) d(180 0 0 1)"
}
```

- Next, create the first joint and first link. We defined the first joint as a child of the 'zero' frame. In the given code, hingeX is a type of joint that allows rotation around a single axis, similar to a hinge. It is used to connect two body parts and specify their relative motion. The "pre" matrix specifies the position and orientation of the joint axis in the local coordinate system of the first body part,

```
joint1(zero):{
    joint: hingeX,
    pre:"T t(0 0 0)"
}

link1 (joint1): {
    shape: capsule, size: [0.3, 0.05],
    color: [0.0, 0.0, 1.0],
    Q: [0.0, 0.0, 0.2]
}
```

- Next, create the second joint and link.

```
joint2(link1):{
   joint: hingeX,
   pre:"T t(0 0.0 .15)"
}

link2 (joint2): {
    shape: capsule, size: [0.2, 0.05],
    color: [1.0, 0.0, 0.0],
    Q: [0.0, 0.0, 0.15]
}
```

- Last, create the third joint and the end-effector.

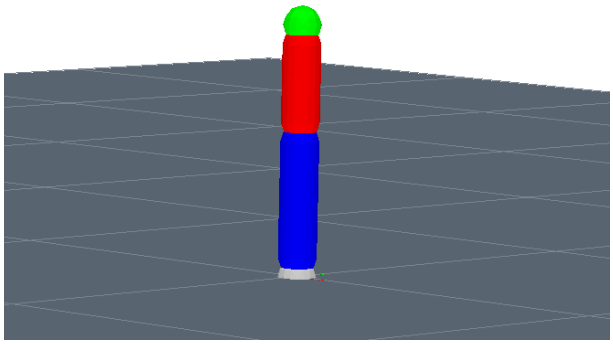
```
joint3 (link2): {
    joint: hingeX,
    pre:"T t(0 0.0 .12)",
}

end-effector (joint3): {
    shape: sphere, size: [0.05],
    color: [0.0, 1.0, 0.0],
    Q: [0.0, 0.0, 0.03]
}
```

- We simply need to use the **ry.addFile()** method to add the contents of the **.g** files to the configuration. Additionally, we will include the Franka Panda robot using these files.

```
K = ry.Config()
K.addFile("two_link_manipulator.g")
K.view()
```

- Finally, you should see the two-link manipulator in the Figure.



6 Features

Features represent differentiable properties of a configuration. They are fundamental in formulating optimization problems, which we will frequently use.

- First, let's explore how to compute them directly from the configuration. To do this, we will clear our previous configuration and add a Franka Panda robot. We can use the **.eval(ry.FS.<name>, frame_set)** method, where frame_set is a list of frame names.

```
C.clear()
C = ry.Config()
C.addFile(ry.raiPath("panda/panda.g"))
q = C.getJointState()

[y,J] = C.eval(ry.FS.position, ["gripper"])
print("feature value:", y, "\nJacobian:", J)

C.view()
```

6.1 List of Features

We are adding the list of features shown in the Fig. 2. However, you can refer to the documentation `https://github.com/MarcToussaint/rai-tutorials/blob/main/config_2_features.ipynb` for more information on features.

FS	frames	D	description
position	[A]	3	3D position of A in world coordinates
positionDiff	[A,B]	3	difference of 3D positions of A and B in world coordinates
positionRel	[A,B]	3	3D position of A in B-coordinates
quaternion	[A]	4	4D quaternion of A in world coordinates\footnote[There is ways to handle the invariance w.r.t.\ quaternion sign properly.]
quaternionDiff	[A,B]	4	...
quaternionRel	[A,B]	4	...
pose	[A]	7	7D pose of A in world coordinates
poseDiff	[A,B]	7	...
poseRel	[A,B]	7	...
vectorX	[A]	3	The x-basis-vector of frame A in world coordinates
vectorXDiff	[A,B]	3	The difference of the above for two frames A and B
vectorXRel	[A,B]	3	The x-basis-vector of frame A in B-coordinates
vectorY...			same as above
scalarProductXX	[A,B]	1	The scalar product of the x-basis-vector of frame A with the x-basis-vector of frame B
scalarProduct...	[A,B]		as above
angularVel	[A]	3	The angular velocity of frame A across two configurations (must be order=1!)
accumulatedCollisions	[]	1	The sum of collision penetrations; when negative/zero, nothing is colliding
jointLimits	[]	1	The sum of joint limit penetrations; when negative/zero, all joint limits are ok
negDistance	[A,B]	1	The NEGATIVE distance between convex meshes A and B, positive for penetration
qItself	[]	n	The configuration joint vector
aboveBox	[A,B]	4	when all negative, A is above (inside support of) the box B
insideBox	[A,B]	6	when all negative, A is inside the box B

Figure 2: The list of features

7 Inverse Kinematics

We know that Forward Kinematics (FK) is the process of finding the position of a robot's end-effector based on the joint angles and link lengths. However, in general robotic manipulation, the goal is typically to find the required joint angles to achieve a desired position, which is known as Inverse Kinematics (IK). However, there may be multiple joint configurations that fit the desired end-effector position; therefore, optimization helps us find the most efficient one based on the provided constraints and objectives.

- Let's begin with the necessary imports and add a Franka Panda robot and a table.

```
import robotic as ry
import numpy as np
import time

C = ry.Config()
C.addFile(ry.raiPath("../rai-robotModels/scenarios/pandaSingle.g"))
C.view()
```

- Next, we add the target frame

```
target = C.addFrame("target", "table")
target.setShape(ry.ST.marker, [.1])
target.setRelativePosition([0., .3, .3])
pos = target.getPosition()
cen = pos.copy()
C.view()
```

- You will learn KOMO later. For now, think of this method as a basic inverse kinematics approach.

```
def IK(C, target, qHome):
    q0 = C.getJointState()
    komo = ry.KOMO(C, 1, 1, 0, False)
    komo.addObjective([], ry.FS.accumulatedCollisions, [], ry.OT.eq)
    komo.addObjective([], ry.FS.jointState, [], ry.OT.sos, [1e-1], q0)
    komo.addObjective([], ry.FS.jointState, [], ry.OT.sos
                      , [1e-1], qHome)
    komo.addObjective([], ry.FS.positionDiff, ["l_gripper", target]
                      , ry.OT.eq, [1e1])

    ret = ry.NLP_Solver(komo.nlp(), verbose=0) .solve()

    return [komo.getPath()[0], ret]
```

- The IK() function in the given code is an implementation of inverse kinematics (IK) for a robot arm. Inverse kinematics is the process of determining the joint angles of a robot arm that will result in a desired end-effector position and orientation. The IK() function takes a configuration object C, a target frame, and qHome (the home configuration of the robot) as input and returns the joint configuration of the robot that achieves the target position.

- The IK() function first gets the current joint state of the robot using the getJointState() method. It then creates a KOMO object with the configuration object C and adds three optimization objectives to it using the addObjective() method. The first objective specifies that the robot should minimize the collisions over the course of the trajectory. The second objective specifies that the cost should be minimized for joint angles that are close to the current joint state. The third objective specifies that the cost should be minimized for joint angles that are close to the home joint state. The fourth objective specifies that the gripper position should be equal to the target position.

- The KOMO constructor takes several arguments, including configuration C, the number of phases, the number of time slices per phase, the order of the optimization problem, and a boolean flag that indicates whether to use the sparse solver.

- The NLP_Solver() method is then used to solve the optimization problem and return the joint configuration that achieves the target position. The getPath() method is used to extract the joint configuration from the KOMO object, and the joint configuration and the optimization status are returned as a list.

```
qHome = C.getJointState()
q_target, ret = IK(C, "target",  qHome)
print(ret)
C.setJointState(q_target)
C.view()
```

- Here, we first compute the joint configuration of the robot that achieves the target position using the IK() function and store it in q_target. The optimization status is stored in ret and printed using the print() function. The joint configuration of the robot is then set to q_target using the setJointState() method, and the scene is updated using the view() method to show the new joint configuration.

- Now let's do it in the loop for better understanding.

```
for t in range(20):
    time.sleep(.1)
    pos = cen + .98 * (pos-cen) + 0.02 * np.random.randn(3)
    target.setPosition(pos)
    q_target, ret = IK(C, "target", qHome)
    print(ret)
    C.setJointState(q_target)
    C.view()
```

- The main loop of the code updates the position of the target frame and calls the IK() function to compute the joint configuration of the robot that achieves the target position. The joint configuration is then set using the setJointState() method and the scene is updated using the view() method.

8 Theory of Inverse Kinematics

- So, we have seen how IK works in simulation. Now, we will solve IK numerically from scratch. Here, we will use the theory behind it, and we will help our robot reach its goal step by step. For this, we will return to our two-link manipulator.

```
import robotic as ry
import numpy as np

K = ry.Config()
K.addFile("two_link_manipulator.g")
K.view()
```

- And here is the basic implementation of theoretical inverse kinematics.

```
n = K.getJointDimension()
q = K.getJointState()
w = 1e-1
W = w * np.identity(n)

y_target = [0.0, 0.7, 0.7]
for i in range(10):

    y, J = K.eval(ry.FS.position, ["end-effector"])

    q += np.linalg.inv(J.T @ J + W) @ J.T @ (y_target - y)

    K.setJointState(q)
    K.view()
```

9 Additional Tutorials

- https://github.com/MarcToussaint/rai-tutorials/blob/main/config_1_intro.ipynb
- https://github.com/MarcToussaint/rai-tutorials/blob/main/config_2_features.ipynb
- https://github.com/MarcToussaint/rai-tutorials/blob/main/config_3_import_edit.ipynb
- https://github.com/MarcToussaint/rai-tutorials/blob/main/komo_1_intro.ipynb
- https://github.com/MarcToussaint/rai-tutorials/blob/main/botop_1_intro.ipynb