RAI Tutorial-1

Ubuntu 20 should be installed

Installation of RAI with Python virtual environment

- First create new folder for RAI
- Create a virtual environment

```
python3 -m venv rai_venv
```

- Activate the Virtual Environment

```
source rai_venv/bin/activate
```

- Install Required Packages

```
pip install numpy scipy
```

```
python3 -m pip install robotic
```

- Test

```
python3 -c 'from robotic import ry; ry.test.RndScene()'
```

- If everything okay, continue with next step

Part 1) Frames

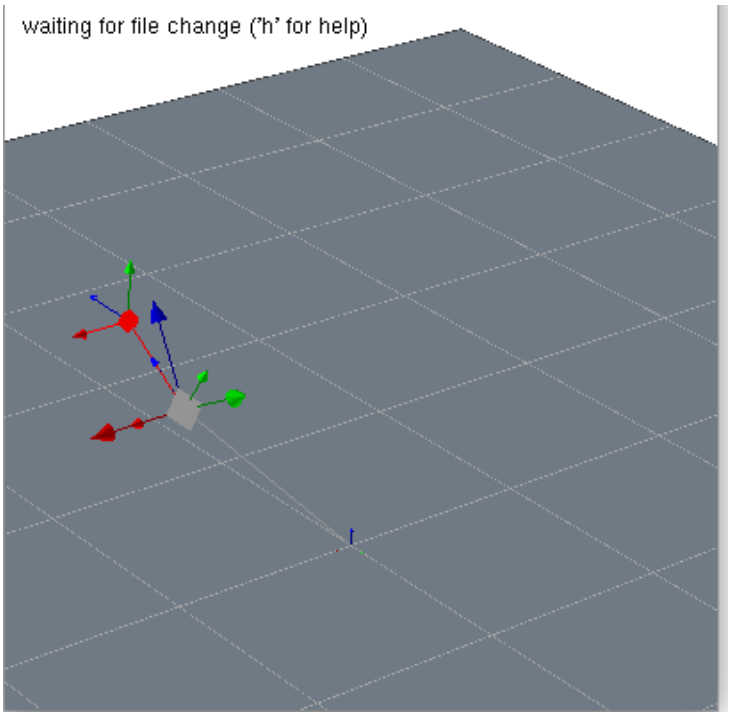
- Create new file: “part1.py”

- Download “mini.g” file from here:
  https://github.com/MarcToussaint/robotics-course/tree/master/course4-Panda

- Inside of the “part1.py” add these lines:

```
from robotic import ry

C = ry.Config()
C.addFile('mini.g')
C.view()

C.watchFile('mini.g')
```

- Now you should see this screen:



- For detailed information, follow here: https://marctoussaint.github.io/robotics-course/script.html#introduction

Part 2) Adding box to the environment

- Create new file: “part2.py”

- Now we will add the box step by step without creating another “.g” file

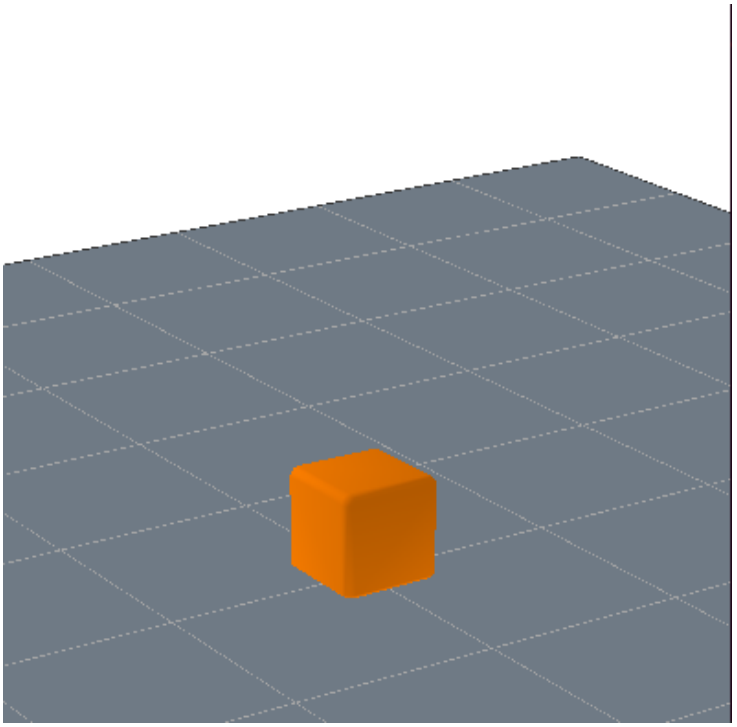
```
from robotic import ry
import time

C = ry.Config()
```

After necessary import and calling configuration, we will add our box:

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

Now visualize:

```
C.view()
time.sleep(5)
```



Here, we can play with the features of the box.

Size, color, position e.t.c

- We can also add the box with “.g” file.

- Create new file: “part2_gfile.py”
- Create new file “box.g”

```
box1: { X: "t(0 0 0.25) d(0 0 0 1)", shape: ssBox, size: [.5 .5 .5
.05], color: [1 .5 0], mass: .1, contact: true }
```

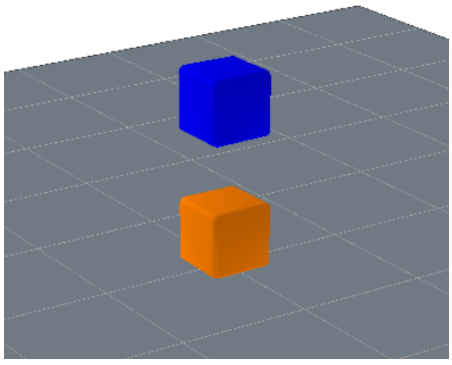
Difference between “ssBox” and “box” is smooth edges and corners.
*sphere-swept convex meshes *

- Now edit the “part2_gfile.py” with calling “watchFile”
- Change the position of the box to “(1 1 0.25)” while ConfigurationViewer is open. see what is happening!
- Then you can understand one of the advantage of “.g” file

Part 3) Relative Positions

- Add second box to 1m above the first box:

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

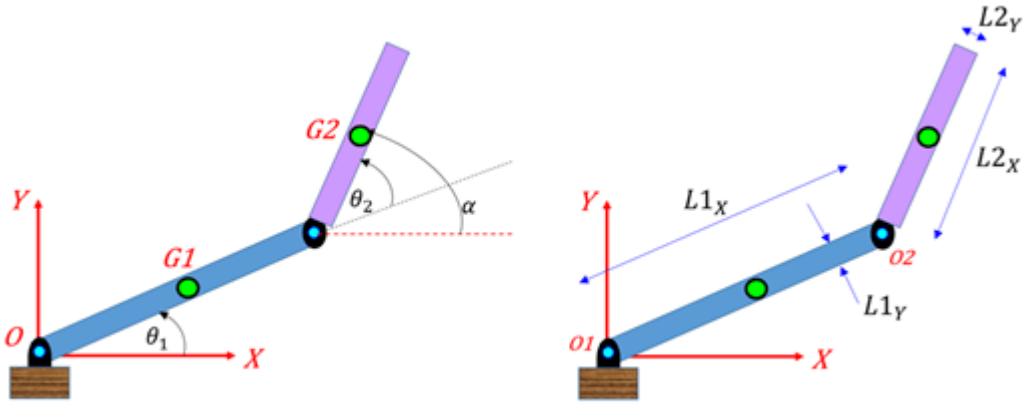


- Display the position and the orientation of the second box: (Why it is not [0,0,1]?)

```
f = C.frame("box2")
print("position:", f.getPosition())
print("orientation:", f.getQuaternion())
```

Part 4) Creating Two link Manipulator

- We are going to create a two link planar manipulator system in this tutorial.



- First create a sphere representing the hinge joint on the (0,0,0,)

```
zero { X:"T t(0 0 0) d(180 0 0 1)"  shape:sphere mass=1 size=[0. 0. .2 .05]}
```

- Then create the first link

```
link1      { shape:capsule mass=1 size=[0. 0. .3 .05] color: [0 0 1]}
```

The `size` parameter is a list of four values that specifies the size of the shape in the x, y, and z directions, as well as the radius of the shape.

- Connecting Link1 to world frame with joint

```
(zero link1) { joint:hingeX, pre:"T t(0 0 .00)" B:"T t(0 0 .2)" }
```

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` and `B` variables in the code specify the transformation matrices of the joint before and after the rotation, respectively. The `pre` matrix specifies the position and orientation of the joint axis in the local coordinate system of the first body part, while

the B matrix specifies the position and orientation of the joint axis in the local coordinate system of the second body part

- Create the second link

```
link2      { shape:capsule mass=1 size=[0. 0. .3 .05] color: [1 0 0]}
```

- Connecting Link2 to Link1 by Hinge joint in X frame

```
(link1 link2) { joint:hingeX, pre:"T t(0 0.0 .15)" B:"T t(0 0.0 .15)" }
```

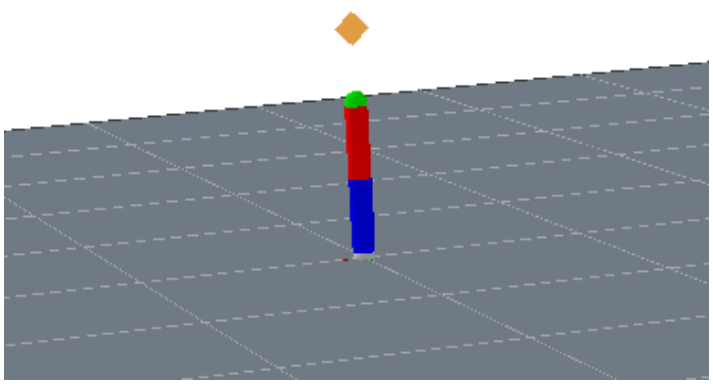
- Create an end effector

```
endeffector      { shape:sphere mass=1 size=[0. 0. .2 .05] color: [0 1 0]}
```

- Add joint to connect end effector to link2

```
(link2 endeffector) { joint:hingeX, pre:"T t(0 0.0 .18)" B:"T t(0 0.0 .0)" }
```

Finally, you should see that:



Part 5) Features

https://github.com/MarcToussaint/rai-python/blob/master/notebooks/1a-configurations.ipynb

Features: computing geometric properties

For every frame we can query its pose:

In [ ]:
```
f = C.getFrame('r_gripper')
print('gripper pos:', f.getPosition())
print('gripper quat:', f.getQuaternion())
print('gripper rot:', f.getRotationMatrix())
```

The above provides basic forward kinematics: After `setJointState` you can query the pose of any configuration frame. However, there is a more general way to query *features*, i.e. properties of the configuration in a differentiable manner. You might not use this often; but it is important to understand as these differentiable features are the foundation of how optimization problems are formulated, which you'll need a lot.

Here are some example features to evaluate:

In [ ]:
```
[y,J] = C.eval(ry.FS.position, ['gripper'])
print('position of gripper:', y, '\nJacobian:', J)
```

In [ ]:
```
# negative(!) distance between two convex shapes (or origin of marker)
C.eval(ry.FS.negDistance, ['panda_coll7', 'r_panda_coll7'])
```

In [ ]:
```
# the x-axis of the given frame in world coordinates
C.eval(ry.FS.vectorX, ['gripper'])
```

https://github.com/MarcToussaint/rai-maintenance/blob/master/help/features.md

FS	frames	D	k	description
position	{o1}	3		3D position of o1 in world coordinates
positionDiff	{o1,o2}	3		difference of 3D positions of o1 and o2 in world coordinates
positionRel	{o1,o2}	3		3D position of o1 in o2 coordinates
quaternion	{o1}	4		4D quaternion of o1 in world coordinates\footnote{There is ways to handle the invariance w.r.t.\ quaternion sign properly.}
quaternionDiff	{o1,o2}	4		...
quaternionRel	{o1,o2}	4		...
pose	{o1}	7		7D pose of o1 in world coordinates
poseDiff	{o1,o2}	7		...
poseRel	{o1,o2}	7		...
vectorX	{o1}	3		The x-axis of frame o1 rotated back to world coordinates
vectorXDiff	{o1,o2}	3		The difference of the above for two frames o1 and o2
vectorXRel	{o1,o2}	3		The x-axis of frame o1 rotated as to be seend from the frame o2
vectorY...				same as above
scalarProductXX	{o1,o2}	1		The scalar product of the x-axis fo frame o1 with the x-axis of frame o2
scalarProduct...	{o1,o2}			as above
gazeAt	{o1,o2}	2		The 2D projection of the origin of frame o2 onto the xy-plane of frame o1
angularVel	{o1}	3	1	The angular velocity of frame o1 across two configurations
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
distance	{o1,o1}	1		The NEGATIVE distance between convex meshes o1 and o2, positive for penetration
qItself	{}	n		The configuration joint vector
aboveBox	{o1,o2}	4		when all negative, o1 is above (inside support of) the box o2
insideBox	{o1,o2}	6		when all negative, o1 is inside the box o2
standingAbove				?

https://github.com/MarcToussaint/rai-python/blob/master/notebooks/retired/2-features.ipynb

Part 6) Inverse kinematics

So we know that Forward Kinematics is finding positions with knowing joint angles. Besides, IK is finding appropriate joint angles to achieve desired position. However, there could be many joint configurations which are fit to the desired end effector position. Therefore, optimization helps us to find the most efficient one according to provided constraints and objectives. Although, we will see optimization topics with more details, here we have an IK example with optimization.

```
from robotic import ry
import numpy as np
import time
C = ry.Config()
C.addFile(ry.raiPath('../rai-robotModels/scenarios/pandaSingle.g'))
C.view()
```

First we make the imports and add the necessary environment.

```
#this reference frame only appears in your workspace C - not the simulation!
target = C.addFrame('target', 'table')
target.setShape(ry.ST.marker, [.1])
target.setRelativePosition([0., .3, .3])
pos = target.getPosition()
cen = pos.copy()
C.view()
```

Now we add the target frame.

```
qHome=[ 0.,  -0.5,  0.,  -2.,   0.,   2.,  -0.5]
q0 = qHome.copy()
q1 = q0.copy()
q1[1] = q1[1] + .2
print(q0, q1)

qHome is the home configuration of the robot. For now, the current
configuration of the robot equals to home configuration.

# you'll learn about KOMO later - this defines a basic Inverse Kinematics
method
def IK(C, pos):
   q0 = C.getJointState()
   komo = ry.KOMO(C, 1, 1, 0, False) #one phase one time slice problem, with
'delta_t=1', order=0
```

```
   komo.addObjective([], ry.FS.jointState, [], ry.OT.sos, [1e-1], q0) #cost:
close to 'current state'
   komo.addObjective([], ry.FS.jointState, [], ry.OT.sos, [1e-1], qHome)
#cost: close to qHome
   komo.addObjective([], ry.FS.positionDiff, ['l_gripper', 'target'],
ry.OT.eq, [1e1]) #constraint: gripper position

   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 and a target position pos 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 cost should be minimized for joint angles that are close to the current joint state. The second objective specifies that the cost should be minimized for joint angles that are close to the home joint state. The third objective specifies that the gripper position should be equal to the target position.

The `KOMO` constructor takes several arguments, including the 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.

```
#pos= [0., .0, .0]
#target.setPosition(pos)
q_target, ret = IK(C, pos)
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, pos)
   print(ret)
   C.setJointState(q_target)
   C.view()
```

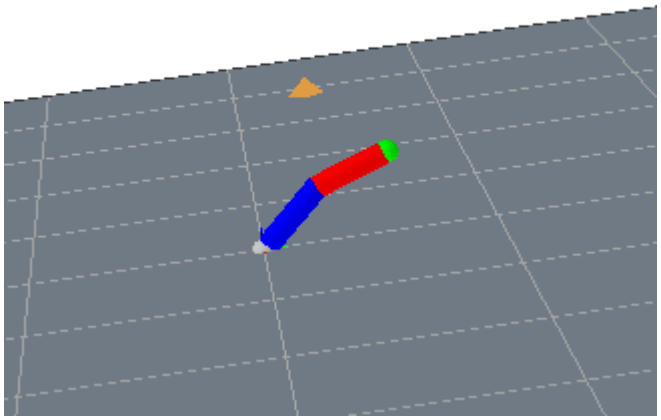
The main loop of the code updates the position of the target frame and calls the IK() function to compute the joint configuration of the robot that achieves the target position. The joint configuration is then set using the setJointState() method and the scene is updated using the view() method.

Part 7) Theory of IK

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 help our robot to reach the goal step by step.

```
n = K.getJointDimension()
q = K.getJointState()
w = 1e-4
W = w * np.identity(n)  # W is equal the ID_n matrix times scalar w

input("initial posture, press Enter to continue...")

y_target = [0.0, 0.7, 0]

for i in range(10):
   # 1st task
   F = K.feature(ry.FS.position, ["endeffector"])
   y, J = F.eval(K)

   # compute joint updates
   q += inv(J.T @ J + W) @ J.T @ (y_target - y)
   # NOTATION: J.T is the transpose of J; @ is matrix multiplication (dot
product)
```

```
# sets joint angles AND computes all frames AND updates display
K.setJointState(q)

# optional: pause and watch OpenGL
K.view()
input("Press Enter to continue...")
```



For more detail you can watch:
https://www.youtube.com/watch?v=7XeDkjekmy0