Motion & Speech

MKI59: Robotlab practical
November 7, 2018

Before starting our Robotlab practicals, we expect you to have set up your python/naoqi environment. You can easily test if you have succeeded by running the following python script.

Code snippet 1: Test your environment

```
import naoqi
```

If you get no error messages, you should be ready to run python with NAOqi on your laptop and the Nao robots.

1 Proxies

Anything you do with NAOqi, you do through proxies. A proxy is an object that behaves as the module it represents. A module in this case is basically a class within a library with functions which you can use to program the Nao. We will dive deeper into what modules exactly are in a later practical, for now you only need to know how to use them. (If you cannot wait until then, here is some more information http://doc.aldebaran.com/2-1/dev/naoqi/index.html)
Lets take a look at the NAOqi `Hello World` example:

Code snippet 2: Hello world

```
import naoqi

ip   = "192.168.1.137"  # example
port = 9559

# to define a proxy, use ALProxy, passing the name of the proxy, ip and port
tts = naoqi.ALProxy("ALTextToSpeech", ip, port)
tts.say("Hello world")
```

So we start with defining the IP address and the port of the robot. **Note that the IP is a string!** You can get the IP by pressing the chest button of the Nao once. The port is always 9559. Next you create a `TextToSpeech` proxy by calling `naoqi.ALProxy()`. You pass the name of the module you wish to create a proxy of (that is in this case `ALTextToSpeech`), the IP and port. If you have created such a proxy you can start calling its functions. In this case you can call `tts.say("Hello world")` to have the robot say "Hello World".

2 Movement

In this section you will learn the basic steps of how to make the robot move using the `ALRobotPosture` and `ALMotion` modules. **WARNING:** be cautious when doing motion with the Nao, especially when performing movements on top of a table. Never let the robot walk on tables!

Your journey in the world of Nao movements starts with exploring Nao postures.

2.1 Postures

The Nao V5 Evolution (H25) has 25 joints that can be controlled to generate motion. Joints are distinguished in head, arm, pelvis, and leg joints. *Learn more about joints at* ***http://doc.aldebaran.com/2-1/family/robots/bodyparts.html#nao-effector***. By rotating joints in a particular manner, the Nao can change between postures. There are several predefined postures in NAOqi e.g. standing, sitting, lying down, etc. These postures are often useful as default starting positions for the robot and are very easy to use as is shown below.

Code snippet 3: Standing posture

```
# Create posture proxy
postureProxy = naoqi.ALProxy("ALRobotPosture", ip, port)
# pass name of posture as string and speed as float between 0.0 and 1.0
postureProxy.goToPosture("Stand", 0.6667)
```

This code snippet makes the robot stand up, again using a proxy but this time of the `ALRobotPosture` module. The `goToPosture` function needs the name of a posture and a speed which indicates how fast it should go to its posture. What makes this function especially useful is that is is 'intelligent' i.e. it can start from (almost) any position and will end in the given posture, avoiding collisions with its own body and maintaining balance. *Try out some different starting positions and postures*, you can get a list of postures from calling `postureProxy.getPostureList()` or from `http://doc.aldebaran.com/2-1/naoqi/motion/alrobotposture.html#term-predefined-postures`.

2.2 Head Motion

With the `ALMotion` module you can do both high level motion control (movement) and low level motion control (give specific commands to specific joints). High level commands are, e.g. to make the robot move at a certain speed in a certain direction. We will start with the lower level motion control and try to move the head. The head of the Nao robot has two degrees of freedom (DOF): *HeadYaw* and *HeadPitch*, as shown in Figure 1.

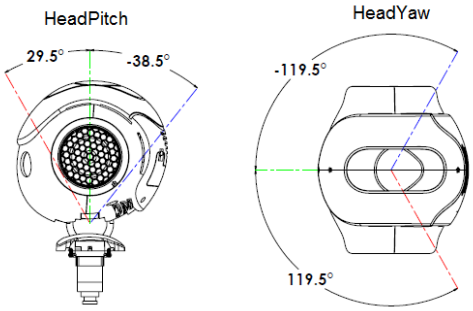

Figure 1: Head pitch and head yaw, with limits in degrees

You can move these two head joints as shown in code snippet 4.

Code snippet 4: "Set angles"

```
#Create the motion proxy
motionProxy = naoqi.ALProxy("ALMotion", ip, port)
# Stable position for the robot so it does not fall over
postureProxy.goToPosture("Sit", 0.8)
```

```
# Put the motors to rest to prevent overheating
motionProxy.rest()

# setAngles needs: a list of joint names (here: yaw and pitch),
#                  a list of corresponding angles and
#                  the speed of execution.
yaw   = 0.5                          #Arbitrary value within range
pitch = 0.5
speed = 0.1
joints = ["HeadYaw", "HeadPitch"]  # Names of the joints (string list)
angles = [yaw, pitch]              # list of the angles (in radians)

motionProxy.setStiffnesses(joints, 0.8)
motionProxy.setAngles(joints, angles, speed)
```

When executing this script the robot will move its head to the given position. You can pass any number of joints and corresponding angles to the `setAngles()` function and it will execute them all. Before calling the `setAngles()` function we call `setStiffnesses`. To understand what this function does, *try removing it and observe what happens*...

Exactly, nothing happens! The stiffness of a motor determines the torque on the motor:

- If set to 0.0 there is no torque at all on the motor and the joint does nothing but can be moved freely from outside.
- If set to 1.0 the motor uses full torque power to reach a given position and tries to maintain that position with all its strength.
- Between these two extremes, the joint is more or less compliant (tries to reach a position but if the torque needed to move is higher than the limitation of Stiffness, the joint does not reach the target position).

By calling motion.rest() we implicitly set the stiffness on all the joints to 0.0, so giving commands to the motors does nothing. By calling `setStiffnesses()` we can explicitly put some stiffness on the joints needed. It is good practice to only put stiffness on the joints you are using and to remove all stiffness if you are done with the entire program to prevent wear and overheating (be sure to put the robot in a stable position when removing stiffness).
When deciding how much stiffness you need to use you have to make a trade-off between precision and accuracy: a motion with low stiffness is generally smooth but inaccurate, while a motion with high stiffness is accurate but irregular. *Try this out with different stiffness values.*

Finally, *try writing a python script which makes the head move in different directions*, e.g. make the Nao nod "yes" or shake "no". Please consider that within a few weeks, you will have to be able to implement reactive behavior, which may consist of Nao's responses in terms of verbal responses (speech) and non-verbal gestures.

2.3 Blocking and non-blocking calls

Use the code from snippet 4, *but this time try calling the* `setAngles()| function multiple times immediately after each other (obviously with different angles) and observe what happens.`

It seems as if only the last motion is executed, but actually all the commands are executed but are immediately overwritten by the next command. That is because the `setAngles()` function is a so-called non-blocking call: it runs on a separate thread so you can give other commands

while it is executing, such as movement commands to other joints. But when calling the same joints again, it will simply override the previous thread and start a new one.
Blocking calls are the complete opposite of non-blocking calls: they run in the same thread as the rest of the program, so you have to wait until a blocking call is finished until you can start the next one. The `goToPosture()` function is actually an example of a blocking call: when calling the two postures in code snippet 3 they did not overwrite each other but were executed sequentially. Code snippet 5 shows an example of `angleInterpolation()`, the blocking equivalent of `setAngles()`:

Code snippet 5: Angle Interpolation

```
postureProxy.goToPosture("Sit", 1.0)
motionProxy.rest()

joints = "HeadYaw"
angles = 1.0
times = 1.0 #time in seconds
isAbsolute = True
# you can just pass "Head" to set all the joints in the head
motionProxy.setStiffnesses("Head", 0.8)

#Interpolates the head yaw to 1.0 radian in 1.0 seconds
motionProxy.angleInterpolation(joints, angles, times, isAbsolute)
```

Try putting multiple of these commands after each other (e.g., looking left and right) and you will notice that they will be executed sequentially. The `angleInterpolation()` function is slightly more complex than `setAngles()`: it also requires the duration of the movement and it uses the `isAbsolute` variable (more on this later). However, `setAngles()` also comes with some extra functionality. For instance, instead of putting multiple of these functions in sequence you could also pass a list of angles and execution times to execute the given angles in sequence:

Code snippet 6: Sequence of Angle Interpolation (single joint)

```
joints = "HeadYaw"
angles = [1.0, 0.0]
times = [1.0, 2.0]
isAbsolute = True
motionProxy.angleInterpolation(joints, angles, times, isAbsolute)
```

This moves the yaw at 1.0 seconds to 1.0 radian and at 2.0 seconds back to 0.0 radian (so the entire movement takes 2.0 seconds, but each part only takes 1.0 second). *Add some additional angles and corresponding times to make the head shake "no".*

Instead of moving the same joints in sequence you could also move two joints at the same time using:

Code snippet 7: Interpolate two joints

```
joints = ["HeadYaw", "HeadPitch"]
angles = [1.0, 0.5]
times = [1.0, 1.0]
isAbsolute = True
motionProxy.angleInterpolation(joints, angles, times, isAbsolute)
```

This moves first the head yaw to 1.0 radian at 1.0 seconds and moves the head pitch to 0.5 radian at 1.5 seconds. Not that the movement of both joints starts at the same time. Finally you can combine this all into one function which executes a sequence of movements for both joints at the same time:

Code snippet 8: Sequence of angle interpolation (two joints)

```
joints = ["HeadYaw", "HeadPitch"]
angles =[[1.0, 0],
         [-0.5, 0.5]]
times = [[1.0, 2.0],
         [0.75, 1.5]]
isAbsolute = True
motionProxy.angleInterpolation(joints, angles, times, isAbsolute)
```

So now it is getting a bit more complicated: in this snippet the head yaw moves at 1.0 second to 1.0 radian and back to 0.0 at 2.0 seconds, while (starting at the same time) the head pitch moves at 0.75 seconds to -0.5 radian and at 1.5 seconds to 0.5 radian. So both the pitch and yaw are starting at the same time, but are moving independently and also finish at a different moment. *Try this function out for a bit and see what you can and cannot do with it.*

One final remark about the `angleInterpolation()` function is the `isAbsolute` parameter. If True the angles are described in absolute angles, else the angles are relative to the current angle. So in code snippet 5 the head yaw is set to 1.0 radian. But if `isAbsolute` is set to False it just adds 1.0 radian to the current yaw and tries to reach the resulting position.

3 Your first arm gestures: Radians

Similar to our explorations in head movements, we will now move the arm. Code snippet 9 uses `setAngles` to move the left arm to specific angular joint positions. Try to run it!

Code snippet 9: Left arm movements (all five joints)

```
leftJoints = ["LShoulderRoll", "LShoulderPitch", "LElbowYaw",
              "LElbowRoll", "LWristYaw"]
angles     = [ -.15 ,  -.45 ,  -.3 ,  -.5 ,  -0.1]

times = [1.0, 1.0, 1.0, 1.0, 1.0]
isAbsolute = True
motionProxy.angleInterpolation(leftJoints, angles, times, isAbsolute)
```

As you see, the robot moves its left arm. Try to improve the karate defense a bit: is the arm high enough? Does the elbow bend enough? *Now, play a bit with these radians*. It seems that making these gestures requires some extensive empirical investigations. Instead, we may want the robot to learn these gestures. Indeed, a prominent example of cognitive robotics concerns the problem of how to learn the robot to move or to make movements.

Alternatively, we may want to move the hand to a certain position in Cartesian space. This is an important topic in robot control: the inverse kinematics problem for motion planning. The problem is as follows: assume the hand (or other effector) has to move from a position $p = (x, y, z)$ to a new position $p' = (x', y', z')$. Now, what sequence of joint rotations is required to make the hand move from p to p'? It may have been a long time since you followed you linear algebra classes?

Lucky for us, Aldebaran has provided several tools that will help us out a bit. NAOqi provides some predefined gestures and... we could also warp into Cartesian space. We will come back to these topics later.

Try to make the Nao perform waving gestures, or greeting gestures

4 Walking

Making the Nao walk is pretty easy, you can use predefined functions (the aforementioned higher level movement commands) so you do not have to worry about maintaining balance or on setting the exact angles for the leg joints. You only have to pass the direction of the movement, as shown in code snippet 10. Note: Always have the robot walk on an even floor (and not on the table!) and stay close to the Nao to catch it in case it falls. Even if it falls it should survive, but since the robotlab floor is very hard, *use one of the black 2x2 mats* (located near the Nao safe) as padding to further decrease the chance of permanent damage.

Code snippet 10: Make the Nao walk

```
import time

# First, make the Nao stand still before it can walk
# did you try the StandInit posture?
postureProxy.goToPosture("Stand", 0.8)

# X = speed forward:
#   1.0 is maximum forward speed,
#   0.0 is no speed,
#  -1.0 is maximum backward speed
X = 0.5

# Y = Sidewards speed:
#   1.0 is maximum speed to the left, -1.0 is maximum speed to the right
```

```
Y = 0.0

# Rotation speed:
#   1.0 is maximum speed counter-clockwise, -1.0 is maximum speed clockwise
Theta = 0.0

motionProxy.moveToward(X, Y, Theta)
time.sleep(10.0)

motionProxy.stopMove()   # Won't stop walking without.

# good practice to have the robot sit at the end
# and while it is safely sitting, let Nao take some rest

postureProxy.goToPosture("Sit", 0.5)
motionProxy.rest()
```

When executing this code snippet the Nao will walk forward for 10 seconds, stops, and sits down. The `moveToward` function needs a forward, sideways and rotational speed. `moveToward` is a non-blocking function, and the NAO will keep walking until `stopMove` is called. Try removing it. You can give new walking instructions, but it will ignore the `goToPosture` function. In code snippet 10 only the forward speeds is non-zero, so it will walk forward. *Try having the NAO walk sideways, rotate around its own axis, and combine those movements.* While exploring the Nao movement commands, consider the Nao reference frames, as explained in, e.g. http://doc.aldebaran.com/2-1/family/robots/joints_robot.html

There is more than one way of having the robot walk. There are two other functions you can use which behave slightly different:

- `moveTo(X,Y,Theta)`: you can pass X and Y coordinates to the robot with the robot as the center of the coordinate system. X and Y are in meters, Theta is the rotation of the robot in radians. You can also pass a list of multiple coordinates for the robot to move to. This is a blocking call and you do not have to call `stopMove` to end the movement: it will end when its position is reached.
- move(X,Y,Theta): in this case X = meters per second forward, Y = meters per seconds sideways, Theta = Rotation in radians per second. This is a non-blocking call and you have to call `stopMove` to end the movement.

Try to explore the walking routines of the Nao, using all three functions:

- *Can you make the Nao walk in circles?*
- *Try to implement "random walk"*