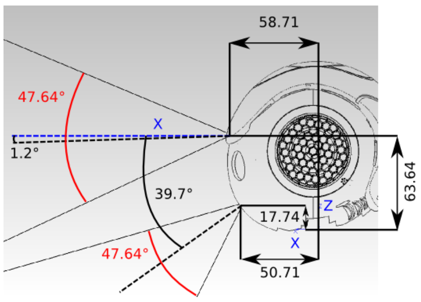
SENSES OF THE NAO

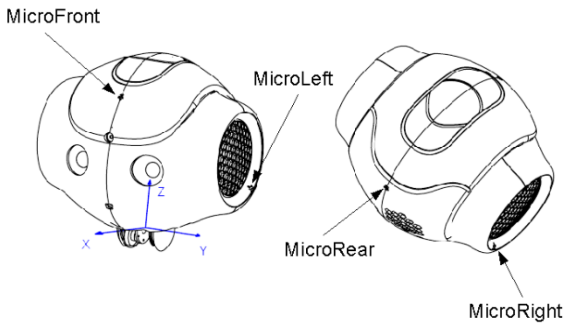
Humans have five senses - hearing, sight, touch, smell and taste. In addition, there are other lesser known human senses. They include the sense of balance, the sense of temperature, and kinesthetic sense. Kinesthetic sense is the ability to know where different parts of your body are without relying on other senses. It is this kinesthetic sense that enables you to close your eyes and touch parts of your body.

There are robots that are capable of performing similar functions to all the senses listed above. The physical devices in robots that sense the environment are called sensors.

The NAO has two cameras in its head. The cameras are vision sensors, and provide camera images for the NAO's computer to process. This is different from the human "sense of sight" as sight would include processing the image to understand what the robot sees.

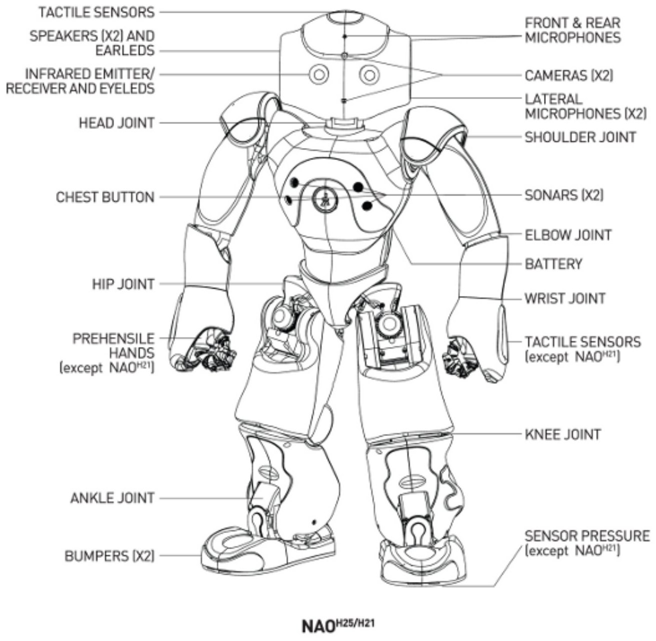
The NAO also has 4 microphones built into its head that it uses to listen to sounds around it. In





the earlier exercises, we used the microphones to allow the NAO to perform speech recognition.

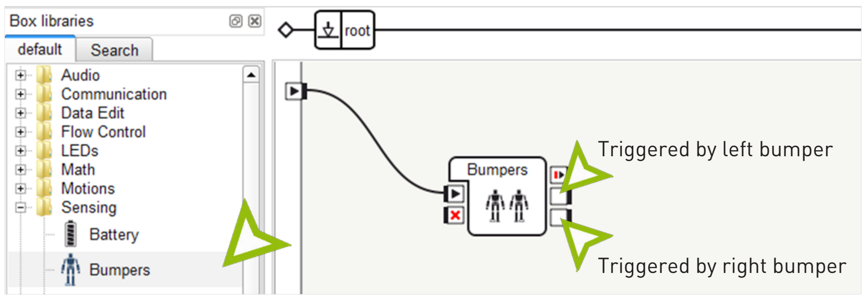
The NAO does not have sensors that allow it to smell or taste things. However, it does have touch sensors on its head and body. It has a tactile sensor on its head that is split into three parts. Each part of the head tactile sensor triggers when touched. It also has a chest button and two foot bumpers, which require an actual press to trigger.



BASIC TASK
LIGHT UP

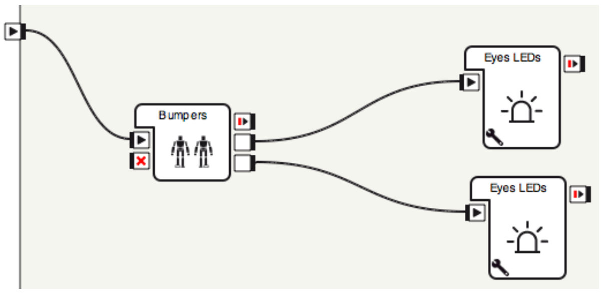
THE NAO HAS FOOT BUMPERS WHICH CAN BE PRESSED, AND VARIOUS LEDS THAT LIGHT UP.
IN THIS LESSON, WE WILL MAKE THE NAO'S EYE LEDS CHANGE COLOR WHEN A FOOT BUMPER IS PRESSED.

01/



Drag out a Bumpers box from the sensors category. The Bumpers box has two outputs that trigger when the foot bumpers are pressed. The top output triggers on the left foot bumper, and the bottom output triggers on the right foot bumper.

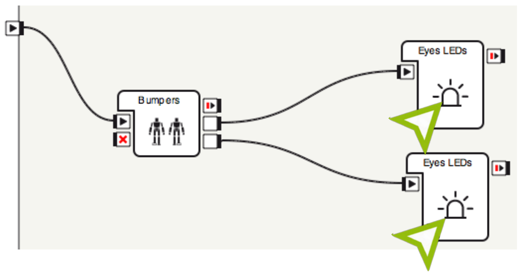
02/

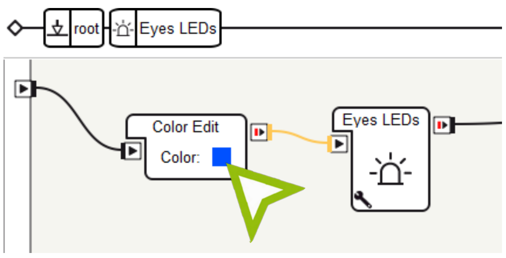


Drag two Eyes LEDs boxes (found in the LEDs category). Connect them as shown below. When the left bumper is triggered, the upper box will be executed. When the right bumper is triggered, the lower box will be executed.

03/

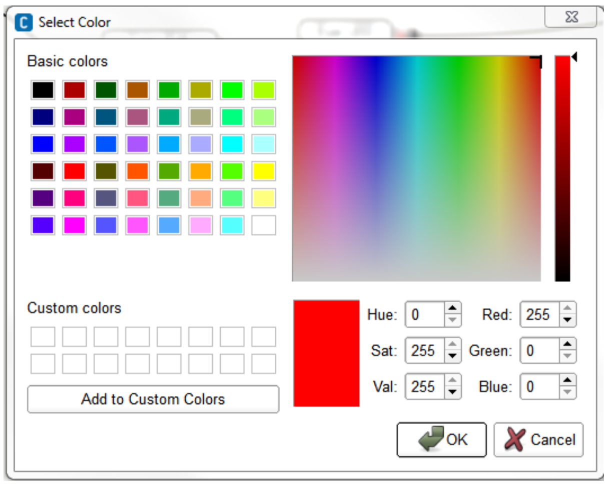
Now we will set the eye colors. Double click on one of the Eyes LEDs boxes, and then click on the color.





04/

Choose any color from the color chooser that appears, but use a different color for each box.
You can click anywhere on the color area and the LED will be set to that color.



Hue: 0 Red: 255
Sat: 255 Green: 0
Val: 255 Blue: 0

05/

Play the behavior. Pressing the foot bumpers should switch between the eye colors you selected.

FINITE STATE MACHINES

A finite state machine, or FSM, is an abstraction commonly used in computer science and robotics. A finite state machine includes *states* (a finite number of them) and *transitions*. We will use a running example to explain these concepts.

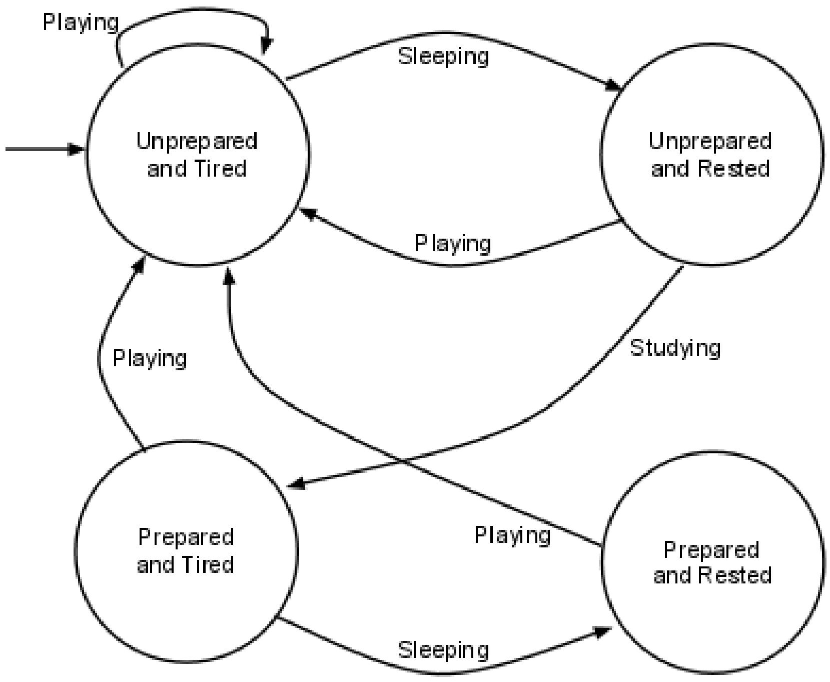
Suppose that an exam is coming up, and you are at home preparing for it over the weekend. Your behaviors (what you do during the weekend) can be described with a finite state machine. Your state can be described using two *features*: how prepared you are for the exam (prepared or unprepared), and how much energy you have (rested or tired). The goal is to ultimately be both prepared and rested at the end of the weekend.

An FSM state must completely describe the situation without any external information. So, one state would be *(prepared and rested)*. Notice that we combined both features into a single state. There will be four states in total - *(prepared and rested)*, *(prepared and tired)*, *(unprepared and rested)* and *(unprepared and tired)*. Each of the four states contains all the information about the situation. Within a FSM, only one state is active at a time.

So in our example, say that we start off in the *(unprepared and tired)* state, since a week of classes just ended (so we're tired), and we haven't had time to study for the exam yet. We want to end up in *(prepared and rested)*. To go from state to state, we have to define the transitions. Transitions move us from one state to another, and can be triggered through actions or events. Intuitively, actions are things that are performed by choice, and events are things that occur. We will use actions in the example below, and discuss events at the end of this section.

Some actions we can take in our example are *studying*, *playing*, and *sleeping*. If we're tired, then sleeping should make us rested. Thus, the transitions from *(prepared and tired)* to *(prepared and rested)*, and from *(unprepared and tired)* to *(unprepared and rested)*, are triggered by *sleeping*. Similarly, the action *studying* will make us prepared for the exam but will tire us out. However, we can only perform this action if we are rested. As such, there is a transition from *(unprepared and rested)* to *(prepared and tired)* that is triggered by *studying*. The last action is *playing*, which can be performed at any state. However, playing causes us to both become tired and forget what we've learned. So, there are transitions from all the other states to *(unprepared and tired)* that are triggered by *playing*.

A FSM can be illustrated with a diagram, as shown below. Circles indicate states, and arrows indicate transitions. The straight arrow that points to (unprepared and tired) indicates that it is the initial state, or the state that we start out in.



Besides triggering transitions with actions, events can also be used. Events are things that happen, that are typically caused by something external to the FSM. For example, an event could be the teacher reducing the scope of the exam. This event might transition us from *(unprepared and tired)* to *(prepared and tired)*, since we already know the areas covered in the revised exam even without studying.

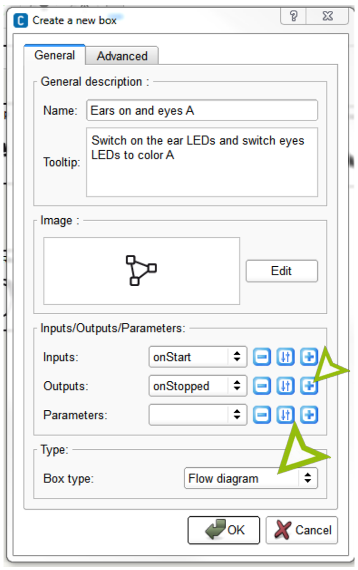
INTERMEDIATE TASK
SWITCHING STATES

IN THIS TASK, WE WILL MAKE THE ROBOT TURN ITS EAR LEDS ON AND OFF BY PRESSING ONEFOOT BUMPER, AND TOGGLE ITS EYE COLORS BY PRESSING THE OTHER FOOT BUMPER. WE WILL IMPLEMENT THIS USING A FINITE STATE MACHINE.

Similar to our finite state machine example in the section above, we have two features in the states. The features are: whether the ear LEDs are on (on and off), and what color the eye LEDs are set to (A or B). Thus, we have four states: *(ears off and eyes as color A)*, *(ears off and eyes as color B)*, *(ears on and eyes as color A)*, and *(ears on and eyes as color B)*. Each press of a foot bumper is an event that will trigger a transition to a different state.

01/

First, we'll create one of the states. Add a custom flow diagram box named BumperState, and add two outputs, "Left", and "Right" to the box.


Create a new box

General | Advanced

General description :

Name: Ears on and eyes A

Tooltip: Switch on the ear LEDs and switch eyes LEDs to color A

Image :

Inputs/Outputs/Parameters:

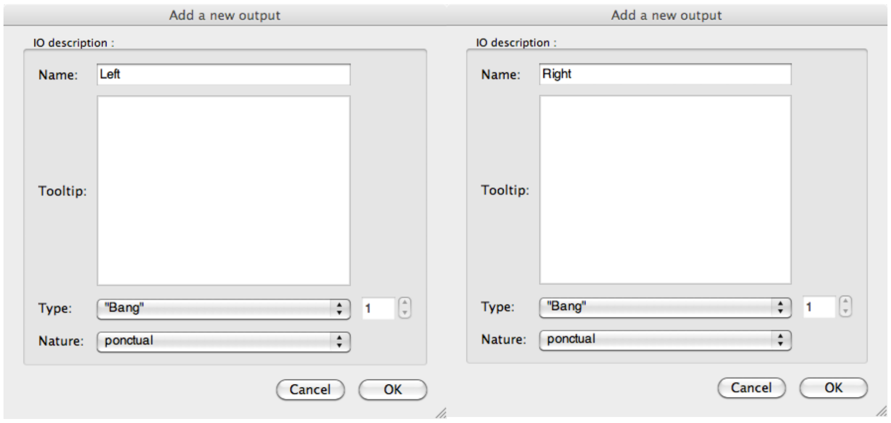
Inputs: onStart

Outputs: onStopped

Parameters:

Type:

Box type: Flow diagram


Add a new output

IO description :

Name: Left

Tooltip:

Type: "Bang" 1

Nature: ponctual

Add a new output

IO description :

Name: Right

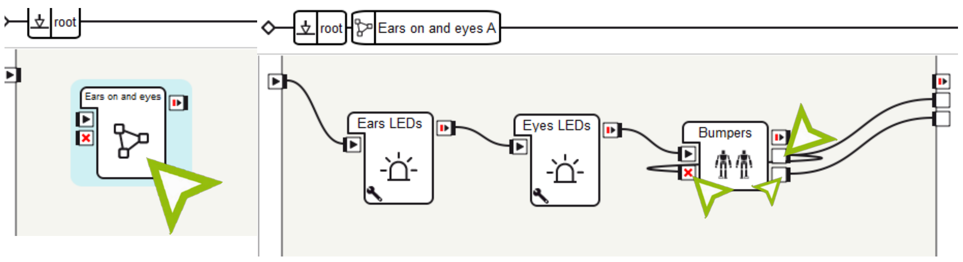
Tooltip:

Type: "Bang" 1

Nature: ponctual

02/

Double-click on the custom box, and a new flow diagram will show up. Add a Bumpers box, an Eyes LED box, and a Ears LED box, and connect them as shown below. Ensure that the outputs of the Bumpers box also link back to its X.

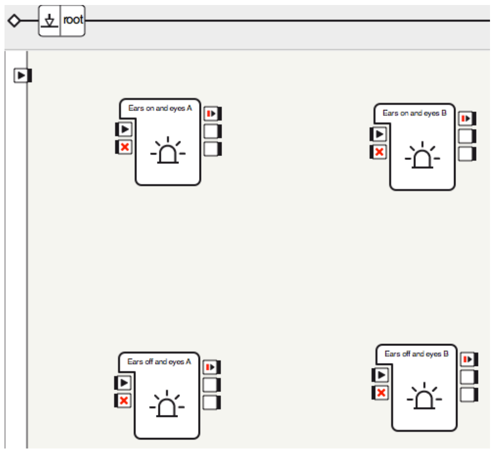


03/

Set the ears intensity to 100%, and set the eye color to a color of your choosing, which we will refer to as color A. The state (ears on and eyes as color A) is now defined with these three boxes. The two outputs, left and right, correspond to the events of the left foot bumper and right foot bumper being pressed.

04/

Click on root to go back to the main flow diagram, then copy and paste these three boxes to create four states, as shown below. Rename each box to correspond to the state it represents.



05/

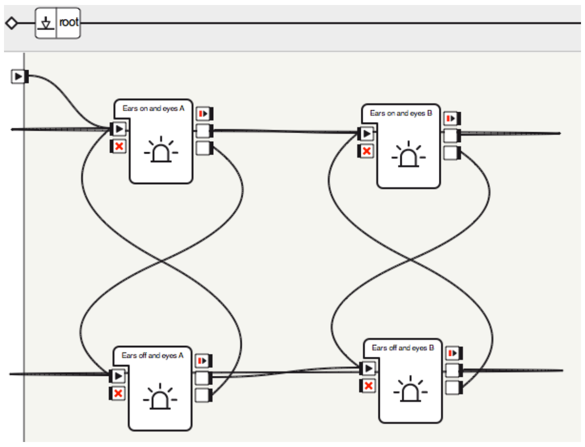
Double-click each of the three new states, and edit the Ears LEDs and Eyes LEDs boxes so that they match the state they are in. In the box "Ears off and eyes B", the ear LEDs should be switched to 0% and the eye LEDS should be set to color B, and so on for all of the states.

06/

You have created the four states. The states the boxes represent, clockwise from the top-left are: *(ears on and eyes as color A)*, *(ears on and eyes as color B)*, *(ears off and eyes as color B)* and *(ears off and eyes as color A)*.

07/

Now add the transitions between them, as shown below. The left foot bumper event should switch states such that the eyes change color, and the right foot bumper event should switch states so that the ear LEDs switch on and off. Also, connect the start event to the Ears on and eyes A box - this defines the initial state of our FSM.



Notice how the figure above corresponds to the diagram of a finite state machine in the earlier section.

08/

Run the behavior. Press the bumpers to ensure that all the states work as intended. Congratulations! You have implemented a finite state machine.

INTERMEDIATE TASK
SENSING WITH MONITOR

IN THIS LAB, WE WILL LEARN HOW TO OBSERVE THE RAW SENSOR VALUES OF THE ROBOT AND GRAPH THEM WITH MONITOR, A TOOL THAT COMES WITH CHOREGRAPHE.

01/

First, run Monitor. It should be in the same directory that you run Choregraphe from.

02/

Click on Memory. We wish to observe some of the sensors, and this information is stored in the NAO's memory.

03/

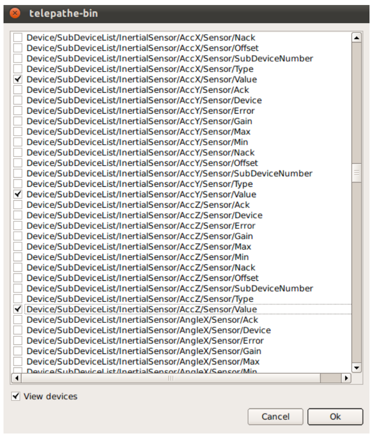
Select your robot from the list, the same way you do in Choregraphe, and connect to it.

04/

Now Telepathe asks you which memory values you want to read. Select "New Configuration."

05/

The dialog window below pops up. Check "View Devices" at the bottom, and scroll down until you find "Device/SubDeviceList/InertialSensor/AccX/Sensor/Value", and check this item. Also select ".../AccY/Sensor/Value" and ".../AccZ/Sensor/Value". These are the accelerometer (A sensor that detects acceleration and tilt) (see sensor section) readings along the x, y and z axes. Click OK.

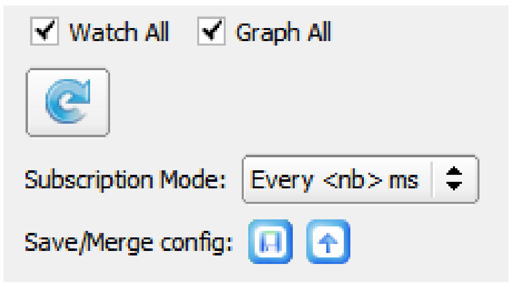


☐ Device/SubDeviceList/InertialSensor/AccX/Sensor/Nack
☐ Device/SubDeviceList/InertialSensor/AccX/Sensor/Offset
☐ Device/SubDeviceList/InertialSensor/AccX/Sensor/SubDeviceNumber
☐ Device/SubDeviceList/InertialSensor/AccX/Sensor/Type
✓ Device/SubDeviceList/InertialSensor/AccX/Sensor/Value
☐ Device/SubDeviceList/InertialSensor/AccY/Sensor/Ack
☐ Device/SubDeviceList/InertialSensor/AccY/Sensor/Device
☐ Device/SubDeviceList/InertialSensor/AccY/Sensor/Error
☐ Device/SubDeviceList/InertialSensor/AccY/Sensor/Gain
☐ Device/SubDeviceList/InertialSensor/AccY/Sensor/Max
☐ Device/SubDeviceList/InertialSensor/AccY/Sensor/Min
☐ Device/SubDeviceList/InertialSensor/AccY/Sensor/Nack
☐ Device/SubDeviceList/InertialSensor/AccY/Sensor/Offset
☐ Device/SubDeviceList/InertialSensor/AccY/Sensor/SubDeviceNumber
☐ Device/SubDeviceList/InertialSensor/AccY/Sensor/Type
✓ Device/SubDeviceList/InertialSensor/AccY/Sensor/Value
☐ Device/SubDeviceList/InertialSensor/AccZ/Sensor/Ack
☐ Device/SubDeviceList/InertialSensor/AccZ/Sensor/Device
☐ Device/SubDeviceList/InertialSensor/AccZ/Sensor/Error
☐ Device/SubDeviceList/InertialSensor/AccZ/Sensor/Gain
☐ Device/SubDeviceList/InertialSensor/AccZ/Sensor/Max
☐ Device/SubDeviceList/InertialSensor/AccZ/Sensor/Min
☐ Device/SubDeviceList/InertialSensor/AccZ/Sensor/Nack
☐ Device/SubDeviceList/InertialSensor/AccZ/Sensor/Offset
☐ Device/SubDeviceList/InertialSensor/AccZ/Sensor/SubDeviceNumber
☐ Device/SubDeviceList/InertialSensor/AccZ/Sensor/Type
✓ Device/SubDeviceList/InertialSensor/AccZ/Sensor/Value
☐ Device/SubDeviceList/InertialSensor/AngleX/Sensor/Ack
☐ Device/SubDeviceList/InertialSensor/AngleX/Sensor/Device
☐ Device/SubDeviceList/InertialSensor/AngleX/Sensor/Error
☐ Device/SubDeviceList/InertialSensor/AngleX/Sensor/Gain
☐ Device/SubDeviceList/InertialSensor/AngleX/Sensor/Max
☐ Device/SubDeviceList/InertialSensor/AngleX/Sensor/Min

✓ View devices

06/

In the bottom left corner of the new window, check the boxes next to “Watch All” and “Graph All”. This will make Telepathe observe and graph the variables we selected. Finally, change the Subscription Mode to “Every ←nb→ ms”. The default value in the dialog that pops up is fine. This is how often we refresh the sensor values in the graph.



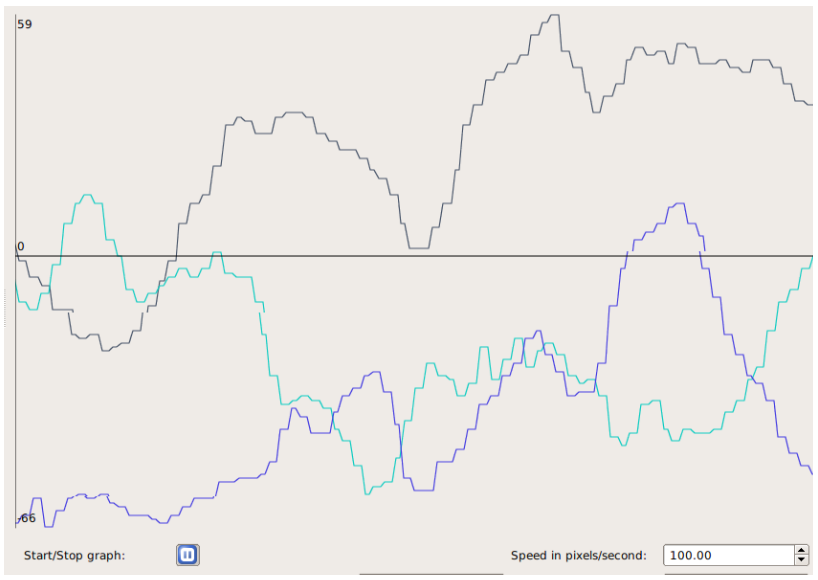
☑ Watch All ☑ Graph All

Subscription Mode: Every <nb> ms

Save/Merge config:

07/

Now the x, y, and z axis accelerometer readings will appear on the graph. Try turning the NAO in all directions. Determine what direction each of the three axes measure acceleration along.



08/

(Optional) Examine some of the other sensor readings. Some good ones to try are any of the Device/SubDeviceList/JOINT/Position/Sensor/Value, which returns the measured joint angle, or Device/SubDeviceList/InertialSensor/(AngleX, AngleY, AngleZ, GyrX, or GyrY)/Sensor/Value. Try to figure out what these measure on your own.

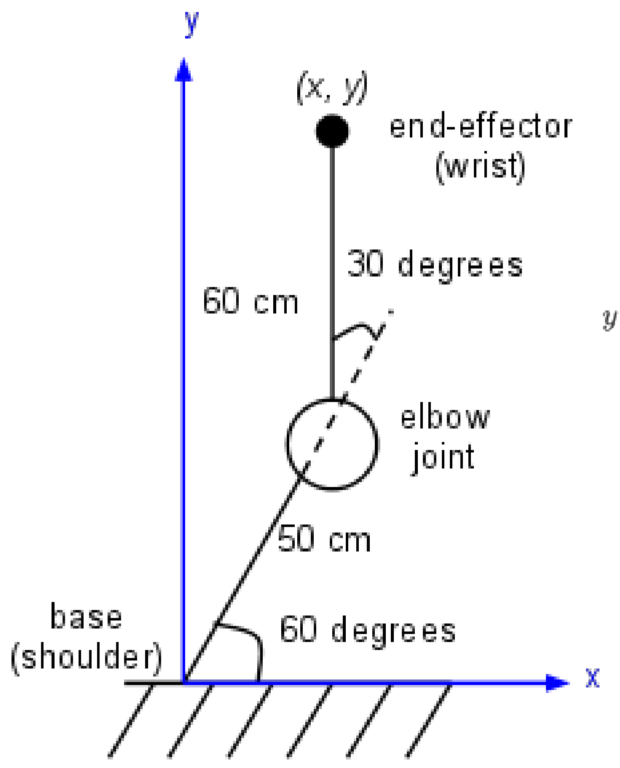
SENSORS AND ACTUATORS

Besides the sensors described earlier, the NAO also has an internal gyroscope and accelerometer inside its torso. The internal gyroscope and accelerometer function like the inner ear, which provides a sense of balance for humans. The gyroscope measures angular velocity (how fast the robot is turning). The accelerometer measures acceleration (which way gravity is pointing). Together, the gyroscope and accelerometer can tell the NAO if it is upright, lying on its back, or lying on its front. Additionally, the two sensors let the NAO know if it is falling down, so it can brace itself for a fall with its arms.

Actuators on a robot refer to its joint motors. The NAO has 21 different motors that can be controlled separately. There are two motors on its head/neck, two for each shoulder, two for each elbow, five for the hips, one for each knee, and two for each ankle. In Module 4, we moved many of these motors to make the NAO dance.

Each motor on the NAO is coupled with a sensor, called an *encoder*. This sensor measures how far each motor has turned. This is known as the motor's angle of rotation. For example, the sensor on the elbow joint is able to tell if the arm is straight or bent at an angle.

Using the angles of rotation of its joints, the NAO is aware of the pose of its entire body. For example, the NAO can calculate how far the hand is from the head. It does so using a *kinematic chain*, which essentially uses trigonometry to calculate relative positions of joints. In the figure below, we show a two-link arm with an elbow joint. Using the length of the arms and the angle of the elbow, we can calculate the position of the *end-effector* (the wrist) relative to the base (the shoulder).



$$y = 50\,\sin(60^o) + 60\,\sin(60^o + 30^o)$$

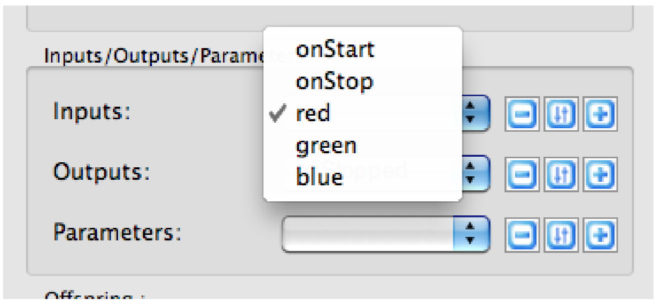
ADVANCED TASK
A BRIGHT IDEA

BEFORE, WE USED THE FOOT BUMPERS TO TOGGLE BETWEEN FOUR STATES. NOW WE'LL USE THE THREE BUTTONS ON THE NAO'S HEAD TO TOGGLE BETWEEN EIGHT STATES.

Things are becoming unwieldy using Choregraphe boxes, so we'll switch to using Python. Each of the three head buttons will control one of the three color components: red, green and blue. By combining these colors, we can make others. For example, red and blue together will make purple, and red and green make yellow.

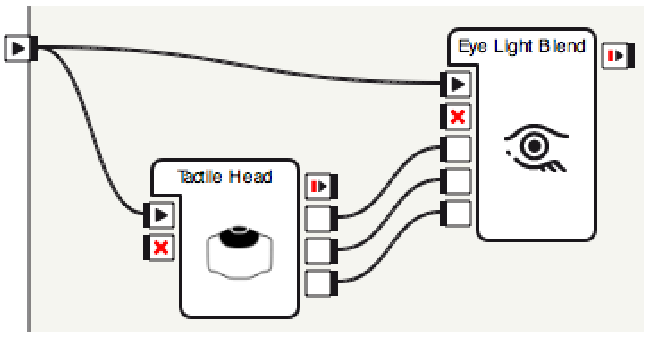
01/

First, add a new box to blend the lights. Add three "bang" inputs, named "red", "green", and "blue".



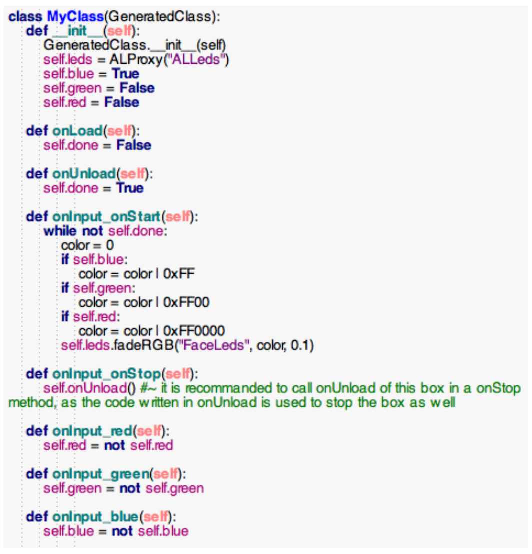
02/

Add a Tactile Head box (in the Sensors category), and connect the boxes as shown below. The three outputs of the Tactile Head box trigger when the front, middle and rear buttons on the head are touched.



03/

Double-click on your custom box to edit the source code. Add the code shown below.



```
class MyClass(GeneratedClass):
    def __init__(self):
        GeneratedClass.__init__(self)
        self.leds = ALProxy("ALLeds")
        self.blue = True
        self.green = False
        self.red = False

    def onLoad(self):
        self.done = False

    def onUnload(self):
        self.done = True

    def onInput_onStart(self):
        while not self.done:
            color = 0
            if self.blue:
                color = color | 0xFF
            if self.green:
                color = color | 0xFF00
            if self.red:
                color = color | 0xFF0000
            self.leds.fadeRGB("FaceLeds", color, 0.1)

    def onInput_onStop(self):
        self.onUnload() #~ it is recommanded to call onUnload of this box in a onStop
method, as the code written in onUnload is used to stop the box as well

    def onInput_red(self):
        self.red = not self.red

    def onInput_green(self):
        self.green = not self.green

    def onInput_blue(self):
        self.blue = not self.blue
```

This script maintains a state consisting of three *boolean* variables: red, green, and blue. Boolean variables can be either `True` or `False`. When a button is pressed, we toggle the corresponding variable using the `not` operator, which performs negation. So, `not True` is `False`, and `not False` is `True`.

We use the state variables to set the LED colors in the `onInput_onStart` method, which continually loops and sets the eye colors. The colors are set using binary 24-bit RGB values. The last eight bits are the blue component, the preceding eight bits are the green component, and the eight bits before that are the red component. So, written in hexadecimal, 0xFF0000 is red, 0x00FF00 is green, and 0x0000FF is blue.

The vertical bar (|) is the bitwise-or operator. This takes two binary numbers, and for each bit, if that bit is set in either number it is set in the output. For example, 100110 | 010100 = 11010. So the sequence of `if` statements sets the corresponding color component if the `red, green` and `blue` variables have been set to `true`.

04/

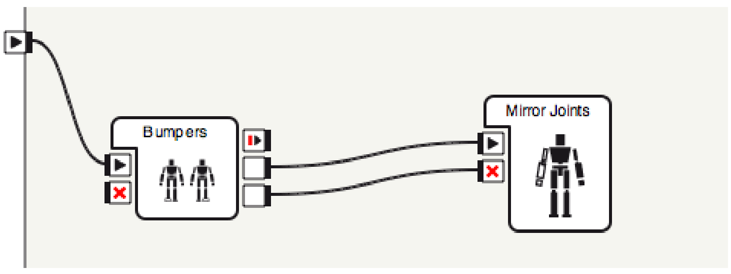
Now run the behavior. Experiment with all eight different color combinations.

05/

(Optional) Draw the finite state machine to describe the behavior we just created. How many states are there? What are the transitions between the states?

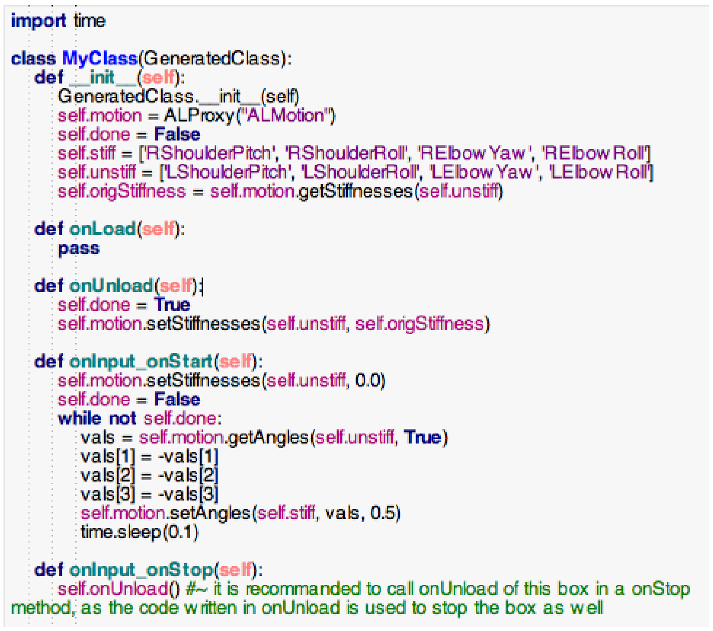
ADVANCED TASK MIRROR, MIRROR ON THE WALL

ANOTHER SENSOR OF THE NAO, OFTEN OVERLOOKED, IS ITS ENCODERS. THESE SENSORS MEASURE THE ANGLES OF ALL OF THE NAO'S JOINTS. IN THIS TASK, WE WILL DISABLE STIFFNESS ON ONE OF THE NAO'S ARMS SO THAT IT CAN BE MOVED FREELY. THEN, WE WILL USE THE ENCODERS TO READ THE POSITIONS OF THE ARM JOINTS, AND MIRROR THESE SAME POSITIONS ON THE NAO'S OTHER ARM. IN THIS WAY, BY MOVING ONE ARM, THE OTHER ARM WILL FOLLOW.



01/

Create a Bumpers box, and connect it to a custom Mirror Joints box as shown below. Hitting the left bumper will start the behavior and hitting the right bumper will stop it.



```
import time

class MyClass(GeneratedClass):
    def __init__(self):
        GeneratedClass.__init__(self)
        self.motion = ALProxy("ALMotion")
        self.done = False
        self.stiff = ['RShoulderPitch', 'RShoulderRoll', 'RElbowYaw', 'RElbowRoll']
        self.unstiff = ['LShoulderPitch', 'LShoulderRoll', 'LElbowYaw', 'LElbowRoll']
        self.origStiffness = self.motion.getStiffnesses(self.unstiff)

    def onLoad(self):
        pass

    def onUnload(self):
        self.done = True
        self.motion.setStiffnesses(self.unstiff, self.origStiffness)

    def onInput_onStart(self):
        self.motion.setStiffnesses(self.unstiff, 0.0)
        self.done = False
        while not self.done:
            vals = self.motion.getAngles(self.unstiff, True)
            vals[1] = -vals[1]
            vals[2] = -vals[2]
            vals[3] = -vals[3]
            self.motion.setAngles(self.stiff, vals, 0.5)
            time.sleep(0.1)

    def onInput_onStop(self):
        self.onUnload() #~ it is recommanded to call onUnload of this box in a onStop
method, as the code written in onUnload is used to stop the box as well
```

02/

Enter the Python code shown below into your custom box.

The variable `stiff` refers to the joints on the arm which will be doing the mirroring (the right arm). The variable `unstiff` refers to the joints on the arm which will not be stiffened (the left arm). The joints are relaxed when the box begins running, and returned to their initial stiffness when the X is pressed and the box finishes.

The `onInput_onStart` method is where the action happens. We have a loop that continuously measures the joint angles in the left arm and sets those same angles in the right arm. The `getAngles` method gets the angles in the arm, and the `setAngles` method sets the angles in the other arm.

All of the joint angles except the shoulder pitch are negated because they are mirrored on the opposite side of the body.

03/

Run the behavior. Press the foot bumper, and move the robot’s arm around. Check that the other arm mirrors the same position.