Fakultät Wirtschaftsinformatik und Angewandte Informatik

Otto-Friedrich-Universität Bamberg

Coding Guide

Jaime A. Delgado Granados
(Matr. No. 1720649)
Fabian Selig (Matr. No. 1588436)
Norman Steinmeier (Matr. No. 1581404)
Daniel Winterstein (Matr. No. 1580725)

Term paper for the seminar *Nao Robot*

WS 2012-13

February 25, 2013

Supervisor: Prof. Dr. Ute Schmid

Contents

Chapter 1

Overview

In the bachelor project, during winter semester, we use the humanoid robot Nao from Aldebaran. Our goal is to use Nao and a small red ball and make Nao kicking the ball into a recognized goal.

1.1 Why a document like this?

In order to get a good overview about how to programm Nao robot using Python language, we thought necessary to create this document for people who will work with this robot in the following semesters so they can improve the skills of Nao quickly having an idea about how to programm it and how our code was made for and how to use again the code if it is necessary.

For that purpose, we will introduce every piece of code we developed for the robot in this document. We will explain the use of each class so the functions include in the class.

This document is divided in four additional chapters. First one includes a brief description of every class and why the were created for. The second one contents a brief description of each method. The fourth chapter will try to explain the code for the most remarkable algorithms we created during the development of this project and the fifth explains where to find more information about coding python for Nao robot.

The organization of the document follows the next structure during the first chapter:

- Name of the Class
- Brief description of the Class and why it was necessary
- Name of each method
- Brife description of the method, what is it doing, parameters we need and information we returned

Chapter 2

Class Description

In this section we describe each class, what it does, why it was created and any other important information about it.

2.1 Class: bootloader.py

Bootloader is created to be started we we initialize Nao. It will load all the arguments we pass to the robot, so the ports, ip or any option we pass to the parser. It will start the execution.

2.2 Class: bps_config.py

Config will try to load all the proxys in the robot. This proxys are mainly: AlMotion, AlTextToSpeech, AlBehaviorManager, AlRedBallTracker, AlVideoDevice, AlRedBallDetection, AlMemory, AlRobotPosture, AlNavigation, AlSonar, AlSensor or AlLandMarkDetection. Once you load all of them, you can use some special characteristics they offer like controlling sensors, cameras, behaviors, etc.

2.3 Class: bps_controller.py

This class will control the main line of execution. It will control the motions, the sensors and mainly the actions that Nao does.

2.4 Class: bps_logger.py

Logger.py is created to recieve all the important information and save it into a file we can read and recognize if something happened during the execution or everything was great because this class will write down everything that is happening in every moment like normal information, fatal errors, warning alarms, etc.

2.5 Class: bps_main.py

Bootloader will initialize bps_main.py class and this one is used to start main line of execution in bp_controller.py class or stop it. That is the only target of this class.

2.6 Class: bps_motion.py

This class will provide us all the information about the motions in Nao. We will control every behavior already loaded in Nao so the movements, the velocity of Nao, we will be able to start a behavior or stop it, control the head or the sensors, etc.

2.7 Class: bps_sensor.py

Sensor wil start and stop all the sensors, it will provide us with the information about where is the ball in the space in relation with where Nao is, the angle of its head, detection of the goal and its position with Nao, etc.

Chapter 3

Coding Guide

In the following pages, there is a complete description of each class and every method we implement during the development of this project.

3.1 Class: bootloader.py

3.1.1 init(self, name)

- SUMARY
  it starts the load of all modules
- PARAMETERS:
  self, name of the module to initialize
- RETURN:

3.1.2 onTouched(self, args)

- SUMARY:
  called when head is touched
- PARAMETERS:
  self, args in case there are arguments
- RETURN:

3.1.3 onSecondTouch(self, args)

- SUMARY:
  called when head is touched again
- PARAMETERS:
  self, args in case there are arguments
- RETURN

3.1.4 main()

main execution method

3.2 Class: bps_config.py

3.2.1 init(self, logger)

- SUMARY:
  initialize logger atribute
- PARAMETERS:
  self, logger where to send all the messages we want to save during execution
- RETURN:

3.2.2 throw(self, logger, message)

- SUMARY:
  send the message passed by PARAMETERS to logger class in case of fatal error
- PARAMETERS:
  self, logger, message we want to write into logger
- RETURN:
  message in logger class

3.2.3 getProxy(self, proxy)

- SUMARY:
  get the information inside a proxy
- PARAMETERS:
  self and the proxy we want information about
- RETURN:
  information about the data inside the proxy or a fatal message into logger

3.2.4 getIP(self)

- SUMARY:
  get the IP
- PARAMETERS:
  self
- RETURN:
  IP

3.2.5 getPort(self)

- SUMARY:
  get the Port
- PARAMETERS:
  self

- RETURN:
  port

3.3 Class: bps_controller.py

3.3.1 init(self, logger, config)

- SUMARY:
  main method for setting up proxys and variables
- PARAMETERS:
  self, logger in which we will write all the important messages and config to activate all the configurations
- RETURN

3.3.2 start(self)

- SUMARY:
  start method with general setup calls
- PARAMETERS:
  self
- RETURN:
  1 if it is stoped

3.3.3 lookForBallCloseRange(self)

- SUMARY:
  used to look for the red ball in close range (0.2-1.2m)
- PARAMETERS:
  self
- RETURN:
  1 if it is stopped

3.3.4 ballFound(self)

- SUMARY:
  Called when the ball has been found
- PARAMETERS:
  self
- RETURN:
  1 if it is stopped

3.3.5 walkToBall(self)

- SUMARY:
  Used to walk to the red ball target
- PARAMETERS:
  self
- RETURN:
  1 if it is stopped

3.3.6 pickUpBall(self)

- SUMARY:
  Picking up the ball
- PARAMETERS:
  self
- RETURN:
  1 if it is stopped

3.3.7 findGoal(self)

- SUMARY:
  Method for finding the goal
- PARAMETERS:
  self
- RETURN

3.3.8 goalFound(self)

- SUMARY:
  Method which is called when the goal was found
- PARAMETERS:
  self
- RETURN:
  if goal was found or not

3.3.9 tooClose(self, colLeft, colRight)

- SUMARY:
  this method control robot in case of collisions
- PARAMETERS:
  self, colLeft and colRight controls collisions from left and right sides
- RETURN

3.3.10 end(self)

- SUMARY:
  method invokated in case of finishing
- PARAMETERS:
  self
- RETURN

3.4 Class: bps_logger.py

3.4.1 init(self)

- SUMARY:
  initialize logger class
- PARAMETERS:
  self
- RETURN

3.4.2 info(self, infoMessage)

- SUMARY:
  this method gives us general information
- PARAMETERS:
  self, infoMessage will be the message we want to write in the log file
- RETURN

3.4.3 warn(self, warnMessage)

- SUMARY: This method gives us a warning message so we know something is not going well
- PARAMETERS:
  self, warnMessage will be the message written into log file
- RETURN

3.4.4 fatal(self, fatalMessage)

- SUMARY:
  method called in case of fatal error
- PARAMETERS:
  self, fatalMessage will be the message written into log file
- RETURN

3.4.5 getTime(self)

- SUMARY:
  method to know the local time
- PARAMETERS:
  self
- RETURN:
  the local time in the following format (h:m:s)

3.4.6 setLogLevel(self, level)

- SUMARY:
  method to set the level of the errors
- PARAMETERS:
  self, level of the error
- RETURN

3.4.7 setLogToFile(self, boolean)

- SUMARY:
  this method confirm we can write log messages into a file
- PARAMETERS:
  self, boolean will set the value of variable logToFile
- RETURN

3.4.8 writeLog(self, message)

- SUMARY:
  this method will write the log message into a file
- PARAMETERS:
  sefl, message will be written into the log file
- RETURN

3.4.9 clearLogFile(self)

- SUMARY:
  this method will remove the log file
- PARAMETERS:
  self
- RETURN

3.5 Class: bps_main.py

3.5.1 start()

- SUMARY:
  it starts the logger, the config and the controller
- PARAMETERS:
  none
- RETURN

3.5.2 stop()

- SUMARY:
  it stops everything
- PARAMETERS:
  none
- RETURN:

3.6 Class: bps_motion.py

3.6.1 init(self, logger, config)

- SUMARY:
  initialize logger, config and load behaviors
- PARAMETERS:
  self, logger and config
- RETURN:

3.6.2 run(self, behaviorName)

- SUMARY:
  execute a behavior in case this is already installed, if not, give us a warn message saying behavior XXX is not installed
- PARAMETERS:
  self, behavior Name which is the name of the behavior we want to run
- RETURN

3.6.3 stop(self)

- SUMARY:
  stops the execution of all the motion behaviors
- PARAMETERS:
  self
- RETURN

3.6.4 gerBehaviors(self)

- SUMARY:
  write in a list all the behaviors already installed in the robot
- PARAMETERS:
  self
- RETURN

3.6.5 getRunningBehaviors(self)

- SUMARY:
  write in a list all the behaviors currently running in the robot
- PARAMETERS:
  self
- RETURN

3.6.6 changeStiffness(self, stiffness)

- SUMARY:
  change the stiffness in the robot, if it is on changes it to off and viceversa
- PARAMETERS:
  self, stiffness will change the stiffness to on or off
- RETURN

3.6.7 enableBalancer(self)

- SUMARY:
  turn on body balancer
- PARAMETERS:
  self
- RETURN

3.6.8 disableBalancer(self)

- SUMARY:
  turn off body balancer
- PARAMETERS:
  self
- RETURN

3.6.9 standUp(self)

- SUMARY:
  the robot will stand up
- PARAMETERS:
  self
- RETURN

3.6.10 rest(self)

- SUMARY:
  the robot will stay in resting position
- PARAMETERS:
  self
- RETURN

3.6.11 moveTo(self, x, y, theta)

- SUMARY:
  make the robot move to (x,y) with the angle theta
- PARAMETERS:
  self, x and y are the position in the space and theta is the angle till that position
- RETURN

3.6.12 turnHead(self, degree, time)

- SUMARY:
  Nao moves its head expressed in degree and time
- PARAMETERS:
  self, degree indicate how many degrees Nao will move its head and time indicates in how much time it will do that movement.
- RETURN

3.6.13 pitchHead(self, degree, time)

- SUMARY:
  Nao pitches its head expressed in degree and time
- PARAMETERS:
  self, degree indicate how many degrees Nao will move its head and time indicates in how much time it will do that movement.
- RETURN

3.6.14 getUp(self)

- SUMARY:
  load the behavior bps_GetUp and Nao will get up
- PARAMETERS:
  self
- RETURN

3.6.15 getDown(self)

- SUMARY:
  load the behavior bps_GetDown and Nao will get down
- PARAMETERS:
  self
- RETURN

3.6.16 getSensorValue(self, sensorName)

- SUMARY:
  get the value of the angle of a sensor in Nao
- PARAMETERS:
  self, sensorName indicate the name of a sensor from which we want to get information about the angles
- RETURN:
  angles in the sensor indicated in sensorName

3.6.17 getRobotPosition(self)

- SUMARY:
  it gives us the position of Nao
- PARAMETERS:
  self
- RETURN:
  position of Nao

3.6.18 getRobotVelocity(self)

- SUMARY:
  it gives us the velocity of Nao
- PARAMETERS:
  self
- RETURN:
  the velocity of Nao

3.6.19 setWalkTargetVelocity(self, x, y, theta, freq)

- SUMARY:
  it sets the velocity to the position indicated in (x,y) with the angle theta
- PARAMETERS:
  self, x and y are the position in the space, theta the angle till that position and freq is the velocity
- RETURN

3.6.20 stopEverything(self)

- SUMARY:
  Nao will stop moving
- PARAMETERS:
  self
- RETURN

3.6.21 turnAround(self, degree)

- SUMARY:
  Nao will turn around the degrees indicated in degree
- PARAMETERS:
  self, degree indicate how many degrees Nao will be moved
- RETURN

3.6.22 placeBall(self)

- SUMARY:
  it loads the behavior bps_PlaceBall which tries to place the ball on the floor
- PARAMETERS:
  self
- RETURN

3.6.23 openLeftHand(self)

- SUMARY:
  it will open left hand of Nao
- PARAMETERS:
  self
- RETURN

3.6.24 closeLeftHand(self)

- SUMARY:
  it will close left hand of Nao
- PARAMETERS:
  self
- RETURN

3.6.25 openRightHand(self)

- SUMARY:
  it will open right hand of Nao
- PARAMETERS:
  self
- RETURN

3.6.26 closeRightHand(self)

- SUMARY:
  it will close right hand of Nao
- PARAMETERS:
  self
- RETURN

3.6.27 closeBothHands(self)

- SUMARY:
  it will close both hands of Nao
- PARAMETERS
  self
- RETURN

3.6.28 kickBall(self)

- SUMARY:
  it will execute Kick behavior
- PARAMETERS
  self
- RETURN

3.6.29 rotateAroundBall(self, distance, angle)

- SUMARY:
  Nao will rotate around the ball
- PARAMETERS:
  self, distance to ball, angle of rotation
- RETURN

3.6.30 celebrateGoal(self)

- SUMARY:
  execute Victory behavior
- PARAMETERS
  self
- RETURN

3.6.31 grabBall(self)

- SUMARY:
  Nao grabs the ball according to where is the ball
- PARAMETERS:
  self, number indicate in which area is the ball. There are 3 established areas
- RETURN

3.7 Class: bps_sensor.py

3.7.1 init(self, logger, config)

- SUMARY:
  initialize Sensor class
- PARAMETERS:
  self, logger in which we will write all the important messages and config to activate all the configurations
- RETURN

3.7.2 getBallPosition(self)

- SUMARY:
  give us the position of the ball
- PARAMETERS:
  self
- RETURN:
  position of the ball

3.7.3 getBallData(self)

- SUMARY:
  this method will give us datas of the ball
- PARAMETERS:
  self
- RETURN:
  data about the detected red ball

3.7.4 getTimeBallData(self)

- SUMARY:
  this method will indicate if we have information in that moment about the ball
- PARAMETERS:
  self
- RETURN: data in case we have information or 0 in case we do not have

3.7.5 getHeadAngle(self)

- SUMARY:
  it will give us the angle of the head in that moment
- PARAMETERS:
  self
- RETURN:
  angles of the head

3.7.6 isBallInHand(self)

- SUMARY:
  This will indicate us if the ball is being grabbed by the hand
- PARAMETERS:
  self
- RETURN:
  return if ball is in hand or not

3.7.7 isNewBall(self)

- SUMARY:
  indicate if Nao found the ball again after missing it
- PARAMETERS:
  self
- RETURN:
  false if there is no data or it is old ball and true if it is new ball

3.7.8 removeBallData(self)

- SUMARY:
  delete datas of the ball from memory
- PARAMETERS:
  self
- RETURN

3.7.9 startHeadTracker(self)

- SUMARY:
  start head tracking of the ball
- PARAMETERS:
  self
- RETURN

3.7.10 stopHeadTracker(self)

- SUMARY:
  stop head tracking of the ball
- PARAMETERS:
  self
- RETURN

3.7.11 subscribeToRedBall(self)

- SUMARY:
  subscribe to the even (redBallDetection)
- PARAMETERS:
  self
- RETURN

3.7.12 unsubscribeToRedBall(self)

- SUMARY:
  unsubscribe from the eve (redBallDetection)
- PARAMETERS:
  self
- RETURN

3.7.13 setCamera(self, use)

- SUMARY:
  change the parameters of the camera
- PARAMETERS:
  self, use determine which camera will be used. 0top 1bottom
- RETURN

3.7.14 startSonar(self)

- SUMARY:
  initialize the use of the sonar
- PARAMETERS:
  self
- RETURN

3.7.15 stopSonar(self)

- SUMARY:
  stop using the sonar
- PARAMETERS:
  self
- RETURN

3.7.16 getSonarLeft(self)

- SUMARY:
  get information from the left sonar
- PARAMETERS:
  self
- RETURN

3.7.17 getSonarRight(self)

- SUMARY:
  get information from the right sonar
- PARAMETERS:
  self
- RETURN

3.7.18 subscribeToLandmarks(self)

- SUMARY:
  subscribe to the event (landmark)
- PARAMETERS:
  self
- RETURN

3.7.19 unsubscribeToLandmarks(self)

- SUMARY:
  unsubscribe to the event (landmarkevent)
- PARAMETERS:
  self
- RETURN

3.7.20 getLandmarkAngle(self)

- SUMARY:
  get the angle of the land mark
- PARAMETERS:
  self
- RETURN:
  angle in case there is information inside the LandMark or 10 in contrary case

3.7.21 getLandmarkDistance(self)

- SUMARY:
  get the distance from Nao to the land mark
- PARAMETERS:
  self
- RETURN:
  the distance till the landmark

3.7.22 getLandmarkPosition(self)

- SUMARY:
  get the information from memory about land mark detected
- PARAMETERS:
  self
- RETURN:
  data of landmark detected previously

Chapter 4

Remarkable Code Passages

In the following pages, there is a complete description of some of the most important algorithms of the code. These algorithms are splitted by class and only those we considered important are included in this guide. The rest of them are also correctly explained with comments inside the files of the code.

4.1 Class: bootloader.py

- METHOD:

  def main():

  # OptionParser is a powerful library for parsing command-line options and arguments in Python
  parser = OptionParser()
  parser.add_option("pip", help="Parent broker port. The IP address or your robot", dest="pip")
  parser.add_option("pport", help="Parent broker port. The port NAOqi is listening to", dest="pport", type="int")
  parser.set_defaults(pip=NAO$_IP, pport = 9559)(opts, args_) = parser.parse_args()
  $pip = opts.pip$
  $pport = opts.pport$
  # You should only need AlBroker if you want to write NAOqi modules in Python,
  $myBroker = ALBroker("myBroker", "0.0.0.0", 0, pip, pport)$
  $globalbootloadr$
  $bootloadr = Bootloader("bootloadr")$
  $try:$
  $whileTrue:$
  $time.sleep(1)$
  $exceptKeyboardInterrupt:$
  $print$
  $print"Interruptedbyuser, shuttingdown"$
  $myBroker.shutdown()$
  $sys.exit(0)$

  $if__name__ == "__main__":$
  $main()$

4.2 Class: bps_config.py

- METHOD:

def __init__(self, logger):

 # initialize logger atribute
self.logger = logger

 # send information to logger indicating config class is initializes and proxys are open
logger.info("Config-Class initialized")
logger.info("Opening Proxys at: " + str(self.IP) + ":" + str(self.PORT))
try:
 # try to start AlMotion proxy and send info to logger
self.motionProxy = ALProxy("ALMotion", self.IP, self.PORT)
logger.info("ALMotion loaded")
except RuntimeError:
 # in case of error, send information to logger
self.throw(logger, "Unable to connect to MotionProxy")
try:
 # try to start AlTextToSpeech proxy and send info to logger
self.speechProxy = ALProxy("ALTextToSpeech", self.IP, self.PORT)
logger.info("ALTextToSpeech loaded")
except RuntimeError:
 # in case of error, send information to logger
self.throw(logger, "Unable to connect to SpeechProxy")
try:
 # try to start AlBehaviorManager proxy and send info to logger
self.behaviorProxy = ALProxy("ALBehaviorManager", self.IP, self.PORT)
logger.info("ALBehaviorManager loaded")
except RuntimeError:
 # in case of error, send information to logger
self.throw(logger, "Unable to connect to BehaviorProxy")
try:
 # try to start AlRedBallTracker proxy and send info to logger
self.redBallProxy = ALProxy("ALRedBallTracker", self.IP, self.PORT)
logger.info("ALRedBallTracker loaded")
except RuntimeError:
 # in case of error, send information to logger
self.throw(logger, "Unable to connect to ALRedBallTracker")
try:
 # try to start AlVideoDevice proxy and send info to logger
self.videoProxy = ALProxy("ALVideoDevice", self.IP, self.PORT)
logger.info("ALVideoDevice loaded")
except RuntimeError:
 # in case of error, send information to logger
self.throw(logger, "Unable to connect to ALVideoDevice")
try:
 # try to start AlRedBallDetection proxy and send info to logger

```
self.redBallDetection = ALProxy("ALRedBallDetection", self.IP, self.PORT)
logger.info("ALRedBallDetection loaded")
except RuntimeError:
# in case of error, send information to logger
self.throw(logger, "Unable to connect to ALRedBallDetection")
try:
# try to start AlMemory proxy and send info to logger
self.memory = ALProxy("ALMemory", self.IP, self.PORT)
logger.info("ALMemory loaded")
except RuntimeError:
# in case of error, send information to logger
self.throw(logger, "Unable to connect to ALMemory")
try:
# try to start AlPosture proxy and send info to logger
self.posture = ALProxy("ALRobotPosture", self.IP, self.PORT)
logger.info("ALRobotPosture loaded")
except RuntimeError:
# in case of error, send information to logger
self.throw(logger, "Unable to connect to ALRobotPosture")
try:
# try to start AlNavigation proxy and send info to logger
self.navigation = ALProxy("ALNavigation", self.IP, self.PORT)
logger.info("ALNavigation loaded")
except RuntimeError:
# in case of error, send information to logger
self.throw(logger, "Unable to connect to ALNavigation")
try:
# try to start AlSonar proxy and send info to logger
self.sonar = ALProxy("ALSonar", self.IP, self.PORT)
logger.info("ALSonar loaded")
except RuntimeError:
# in case of error, send information to logger
self.throw(logger, "Unable to connect to ALSonar")
try:
# try to start AlSensors proxy and send info to logger
self.sensors = ALProxy("ALSensors", self.IP, self.PORT)
logger.info("ALSensors loaded")
except RuntimeError:
# in case of error, send information to logger
self.throw(logger, "Unable to connect to ALSensors")
try:
# try to start AlProxy proxy and send info to logger
self.naoMarkProxy = ALProxy("ALLandMarkDetection", self.IP, self.PORT)
logger.info("ALLandMarkDetection loaded")
except RuntimeError:
# in case of error, send information to logger
self.throw(logger, "Unable to connect to ALLandMarkDetection")
```

4.3 Class: bps_controller.py

- METHOD:

def walkToBall(self):

if (self.isStop):
self.end()
return 1

#"Looking at my ball" message
self.speech.say("Looking at my ball")

ballLost = 0
atBall = False

#Starting the sensors
self.sensor.startHeadTracker()
self.sensor.startSonar()

while(atBall == False):
if (self.isStop):
self.sensor.stopHeadTracker()
self.sensor.stopSonar()
self.end()
return 1

time.sleep(self.walkIterationTime)

headAngle = self.motion.getSensorValue("HeadYaw")[0]

 # Check whether or not we are looking at our own shoulders
if(headAngle ¡ - 0.75 or headAngle ¿ 0.75):
self.logger.info("HeadYaw: " + str(headAngle))
self.sensor.stopHeadTracker()
self.motion.turnHead(0.0, 0.5)
self.sensor.startHeadTracker()

 # get the ball position
x = self.sensor.getBallPosition()[0]
y = self.sensor.getBallPosition()[1]

self.distance = math.sqrt(math.pow(x,2)+math.pow(y,2))
angle = math.atan2(y, x)
angleRounded = int(angle/(5.0*motion.TO_RAD))*(5.0*motion.TO_RAD)

 # walking velocity angle must be between -1 and 1
if(angleRounded¿1):
angleRounded = 1
if(angleRounded¡-1):

```
angleRounded = -1

self.logger.info("Ball at: " + str(x) + "," + str(y) + " with " + str(angleRounded)
+ " in " + str(self.distance))

# Reducing the speed the closer we get to the ball
speed = self.walkingSpeed
if(self.distance ¡= self.walkingSpeed):
speed = self.distance

#Actual walking call (non blocking and iterated)
self.motion.setWalkTargetVelocity(1.0, 0.0, angleRounded, speed)

# Checking if the ball is still visible
if(self.sensor.isNewBall() == False):
ballLost = ballLost + 1
self.logger.info("Ball lost?")
else:
ballLost = 0

# Collision detection and reaction
self.colLeft = False
self.colRight = False

if(self.sensor.getSonarLeft() ¡= self.maxSonar):
self.colLeft = True

if(self.sensor.getSonarRight() ¡= self.maxSonar):
self.colRight = True

self.tooClose(self.colLeft, self.colRight)

# If we lost sight of the ball a certain amount
if(ballLost ¿= self.ballLostMax):
self.speech.say("I lost track of my ball")
atBall = True
self.motion.stopEverything()
self.sensor.stopHeadTracker()
self.sensor.stopSonar()
self.motion.standUp()
self.lookForBallCloseRange()
return 1

# If we reached our target distance
if(self.distance ¡= self.distanceToTarget):
self.logger.info("At my Target")
self.motion.stopEverything()
self.sensor.stopHeadTracker()
atBall = True
```

self.motion.standUp()

self.speech.say("Final correction")
self.motion.turnAround(math.degrees(angle))

self.sensor.stopSonar()
self.findGoal()
return 1

- METHOD:
def goalFound(self):

if (self.isStop):
self.end()
return 1

self.speech.say("Found the goal!")
self.motion.standUp()

 # set on the bottom camera. 0top 1bottom
self.sensor.setCamera(1)
self.sensor.subscribeToLandmarks()
 # moving head
self.motion.pitchHead(-30, 0.5)

self.sensor.subscribeToLandmarks()

time.sleep(self.retardSecond*2)

 # locating landmark
self.sensor.getLandmarkPosition()
 # getting landmark angle
landMarkAngle = self.sensor.getLandmarkAngle()
 # writing info about where landmark is
self.logger.info("Landmark at: "+str(self.sensor.getLandmarkPosition()))

if(landMarkAngle!=10):
 # Nao rotates around the ball according to distance to target and landmark angle
self.motion.rotateAroundBall(self.distanceToTarget, math.degrees(landMarkAngle))

 # moving head
self.motion.turnHead(0, 0.1)
self.motion.pitchHead(15, 0.4)

 # Nao says Aiming
self.speech.say("Aiming!")

```
# Nao starts head tracking
self.sensor.startHeadTracker()
time.sleep(1)

# get info about position of the ball
x = self.sensor.getBallPosition()[0]
y = self.sensor.getBallPosition()[1]

# stopping head tracking
self.sensor.stopHeadTracker()

# Nao moves to that position
self.motion.moveTo(x-0.13,y-0.05, 0)
# Nao kicks the ball
self.motion.kickBall()

self.sensor.unSubscribeFromLandMarks()
self.end()
else:
# Nao speaks
self.speech.say("Where did the goal go?")
self.sensor.unSubscribeFromLandMarks()
# Nao looks for the goal
self.findGoal()
```

4.4 Class: bps_motion.py

- METHOD:
  def rotateAroundBall(self, distance, angle):

  # write information about rotation into logger
  self.logger.info("Rotating around ball at distance "+str(distance)+" / "+str(angle)+" degrees")
  # calculate position and radians till the ball
  x = distance - math.cos(math.radians(angle))*0.25
  y = math.sin(math.radians(angle))*distance
  theta = math.radians(angle)
  # move to position indicated in (x,y) with the angle theta
  self.moveTo(x, -y, theta)

- METHOD:
  def run(self, behaviorName):

  # write into logger
  self.logger.info("Trying to run Behavior: " + behaviorName)
  # if behavior is already installed...
  if(self.behaviorManager.isBehaviorInstalled(behaviorName)):
  # change the stiffness in Nao
  self.changeStiffness(1.0)
  # write into logger
  self.logger.info("Now running Behavior: " + behaviorName)
  # execute the behavior
  self.behaviorManager.runBehavior(behaviorName)
  # write into logger
  self.logger.info("Finished running Behavior: " + behaviorName)
  else:
  # write a warning message into logger
  self.logger.warn("Behavior ", + behaviorName + " is not installed")

4.5 Class: bps_sensor.py

- METHOD:
  def isNewBall(self):
  # get moment of last time Nao got data of the ball
  data = self.getTimeBallData()
  # if there is no data
  if(data == 0):
  return False

  # get moment of last time Nao got data of the ball
  timeMillis = self.getTimeBallData()[1]
  # write it on logger
  self.logger.info("Time for last ball: "+str(timeMillis))

  # if data is not null
  if(data):
  if(timeMillis ¡= self.timeMillisOld):
  # change value of old time Nao saw the ball for new moment
  self.timeMillisOld = timeMillis
  # write into logger
  self.logger.info("Old Ball!")
  return False
  else:
  # change value of old time Nao saw the ball for new moment
  self.timeMillisOld = timeMillis
  self.logger.info("New Ball!")
  return True
  else:
  return False

- METHOD:
  def getLandmarkAngle(self):

  get position of landmark
  data = self.getLandmarkPosition()
  if there is information in data
  if(data):
  get the angle from the data
  angle = data[1][0][0][1]
  write the angle into logger
  self.logger.info("Angle: "+str(angle))
  return angle
  else:
  return 10

Chapter 5

Further Information

5.1 Main Information

Every part of the code is properly commented inside the original code files. Every class is described so every method and inside each method we can find inside comments about what the code is doing in every moment so the person who reads the code is perfectly able to understand what is code made for.

5.2 Read More

Aldebaran Robotics offers a complete guide for programming Nao with a complete list of examples, installing guide for the SDK, tips and tricks section, Python tutorial and many samples of code for Nao. Take a look into this site provided by Aldebaran Robotics in order to solve any doubt you could have.