```
#!/usr/bin/env python
import rospy
import roslib
import math
from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan
from kobuki_msgs.msg import BumperEvent
from kobuki_msgs.msg import CliffEvent
from kobuki_msgs.msg import WheelDropEvent

class Move():
    STATE_STOPPED = 0   # longer pause for bumper stop, back off and
turn
    STATE_FORWARD = 1
    STATE_TURN_LEFT_SMALL = 2
    STATE_TURN_LEFT_MEDIUM = 3
    STATE_TURN_LEFT_LARGE = 4
    STATE_TURN_RIGHT_SMALL = 5
    STATE_TURN_RIGHT_MEDIUM = 6
    STATE_TURN_RIGHT_LARGE = 7
    STATE_TURN = 8              # this is for turning 180 degree
without backing off
    STATE_BACKWARD_BUMPER = 9
    STATE_PAUSED = 10   # this is for shorter pause used for turning
    STATE_WHEEL_DROP = 11
    STATE_CLIFF = 12

    def __init__(self):
        ############################
        # Setting up for LaserScane
        ############################
        # Binary values (obstacle exists or not exists)
        self.b1 = 0
        self.b2 = 0
        self.b3 = 0
        self.b4 = 0
        self.b5 = 0

        # Average value of each section
        self.sect_1 = 0.0
        self.sect_2 = 0.0
        self.sect_3 = 0.0
        self.sect_4 = 0.0
        self.sect_5 = 0.0

        # Tweaking settings
        self.threshold = 0.5

        self.LASER_FLAG = Move.STATE_FORWARD
        self.BUMPER_FLAG = Move.STATE_FORWARD
```

```
        self.WHEELDROP_FLAG = Move.STATE_FORWARD
        self.CLIFF_FLAG = Move.STATE_FORWARD

        self.states = {
                    0: Move.STATE_FORWARD,
                    1: Move.STATE_TURN_LEFT_SMALL,
                   10: Move.STATE_TURN_LEFT_MEDIUM,
                   11: Move.STATE_TURN_LEFT_MEDIUM,
                  100: Move.STATE_TURN_LEFT_LARGE,
                  101: Move.STATE_TURN_LEFT_LARGE,
                  110: Move.STATE_TURN_LEFT_LARGE,
                  111: Move.STATE_TURN_LEFT_LARGE,
                 1000: Move.STATE_TURN_RIGHT_MEDIUM,
                 1001: Move.STATE_TURN_LEFT_LARGE,
                 1010: Move.STATE_TURN_LEFT_LARGE,
                 1011: Move.STATE_TURN_LEFT_LARGE,
                 1100: Move.STATE_TURN_RIGHT_LARGE,
                 1101: Move.STATE_TURN_LEFT_LARGE,
                 1110: Move.STATE_TURN_LEFT_LARGE,
                 1111: Move.STATE_TURN_LEFT_LARGE,
                10000: Move.STATE_TURN_RIGHT_SMALL,
                10001: Move.STATE_TURN_RIGHT_LARGE,
                10010:  Move.STATE_TURN_RIGHT_LARGE,
                10011:  Move.STATE_TURN_LEFT_LARGE,
                10100:  Move.STATE_TURN_RIGHT_LARGE,
                10101:  Move.STATE_TURN_RIGHT_LARGE,
                10110:  Move.STATE_TURN_RIGHT_LARGE,
                10111:  Move.STATE_TURN_LEFT_LARGE,
                11000:  Move.STATE_TURN_RIGHT_MEDIUM,
                11001:  Move.STATE_TURN_RIGHT_LARGE,
                11010:  Move.STATE_TURN_RIGHT_LARGE,
                11011:  Move.STATE_TURN_RIGHT_LARGE,
                11100:  Move.STATE_TURN_RIGHT_LARGE,
                11101:  Move.STATE_TURN_RIGHT_LARGE,
                11110:  Move.STATE_TURN_RIGHT_LARGE,
                11111:  Move.STATE_TURN_RIGHT_LARGE}

        # Initialize LaserScan
        rospy.init_node('Move')
        rospy.Subscriber('/scan', LaserScan, self.call_back)


        ############################
        # Setting up for Move
        ############################
        # initiliaze Move Node
        rospy.Subscriber("/mobile_base/events/
bumper",BumperEvent,self.BumperEventCallback)
        rospy.Subscriber("/mobile_base/events/
cliff",CliffEvent,self.CliffEventCallback)
```

```
        rospy.Subscriber("/mobile_base/events/
wheel_drop",WheelDropEvent,self.WheelDropEventCallback)

        # tell user how to stop TurtleBot
        rospy.loginfo("To stop TurtleBot CTRL + C")

        # What function to call when you ctrl + c
        rospy.on_shutdown(self.shutdown)

        # Create a publisher which can "talk" to TurtleBot and tell it
to move
        # Tip: You may need to change cmd_vel_mux/input/navi to /
cmd_vel if you're not using TurtleBot2
        self.cmd_vel = rospy.Publisher('cmd_vel_mux/input/navi',
Twist, queue_size=10)

        #TurtleBot will stop if we don't keep telling it to move.  How
often should we tell it to move? 10 HZ
        r =rospy.Rate(10);

        # Twist is a datatype for velocity
        move_cmd = Twist()
        # let's go forward at 0.2 m/s
        move_cmd.linear.x = 0.2
        # let's turn at 0 radians/s
        move_cmd.angular.z = 0

        #stop_cmd: By default, new instance has velocity 0
        stop_cmd = Twist()

        #turn_cmd: let's turn at 45 deg/s
        turn_cmd = Twist()
        turn_cmd.linear.x = 0
        turn_cmd.angular.z = math.radians(45) #45 deg/s in radians/s
counterclockwise

        #turn_cmd_inv: let's turn at 45 deg/s
        turn_cmd_inv = Twist()
        turn_cmd_inv.linear.x = 0
        turn_cmd_inv.angular.z = math.radians(-45) #45 deg/s in
radians/s counterclockwise

        # backward_cmd
        backward_cmd = Twist()
        # backward speed at -0.2 m/s
        backward_cmd.linear.x = -0.2
        # let's turn at 0 radians/s
        backward_cmd.angular.z = 0
```

```
        # self. refers to an instance of a class
        # Move. refers to the class variable
        self.BUMPER_STATE = Move.STATE_FORWARD
        self.LASER_STATE = Move.STATE_FORWARD
        ############################
        # State Machine
        ############################
        while not rospy.is_shutdown():
                # Update states using different flags
                self.LASER_STATE = self.LASER_FLAG
                self.BUMPER_STATE = self.BUMPER_FLAG
                self.WHEELDROP_STATE = self.WHEELDROP_FLAG
                self.CLIFF_STATE = self.CLIFF_FLAG

                if (self.BUMPER_STATE == Move.STATE_FORWARD and
self.LASER_STATE == Move.STATE_FORWARD and self.WHEELDROP_STATE ==
Move.STATE_FORWARD and self.CLIFF_STATE == Move.STATE_FORWARD):
                        rospy.loginfo("In moving state")
                        self.cmd_vel.publish(move_cmd)
                        r.sleep()

                # This is where all the bumper checking should happen
                if (self.BUMPER_STATE != Move.STATE_FORWARD or
self.CLIFF_STATE != Move.STATE_FORWARD):
                        if (self.BUMPER_STATE == Move.STATE_STOPPED or
self.CLIFF_STATE == Move.STATE_STOPPED):
                                for x in range(0,20):
                                        self.cmd_vel.publish(stop_cmd)
                                        rospy.loginfo("Waiting ...")
                                        r.sleep()

                                for x in range(0,2):

self.cmd_vel.publish(backward_cmd)
                                        r.sleep()
                                for x in range(0,10):
                                        r.sleep()

                                rospy.loginfo("Turning 180 degree ")
                                for x in range(0,40):
                                        self.cmd_vel.publish(turn_cmd)
                                        r.sleep()
                                self.BUMPER_STATE = Move.STATE_FORWARD
                                self.CLIFF_STATE = Move.STATE_FORWARD
                                for x in range(0,10):
                                        r.sleep()


                if (self.WHEELDROP_STATE != Move.STATE_FORWARD):
```

```
                        self.cmd_vel.publish(stop_cmd)
                        r.sleep()

                # when laser detects obstacles
                if (self.LASER_STATE != Move.STATE_FORWARD):

                        while (self.LASER_STATE !=
Move.STATE_FORWARD):
                                # Pause for a bit
                                for x in range(0,10):
                                        self.cmd_vel.publish(stop_cmd)
                                        r.sleep()
                                # Turn Correspondingly
                                if (self.LASER_STATE ==
Move.STATE_TURN_LEFT_SMALL):
                                        for x in range(0,10):

self.cmd_vel.publish(turn_cmd)
                                                r.sleep()
                                        for x in range(0,10):
                                                r.sleep()

                                elif (self.LASER_STATE ==
Move.STATE_TURN_LEFT_MEDIUM):
                                        for x in range(0,20):

self.cmd_vel.publish(turn_cmd)
                                                r.sleep()
                                        for x in range(0,10):
                                                r.sleep()
                                elif (self.LASER_STATE ==
Move.STATE_TURN_LEFT_LARGE):
                                        for x in range(0,30):

self.cmd_vel.publish(turn_cmd)
                                                r.sleep()
                                        for x in range(0,10):
                                                r.sleep()
                                elif (self.LASER_STATE ==
Move.STATE_TURN_RIGHT_SMALL):
                                        for x in range(0,10):

self.cmd_vel.publish(turn_cmd_inv)
                                                r.sleep()
                                        for x in range(0,10):
                                                r.sleep()
                                elif (self.LASER_STATE ==
Move.STATE_TURN_RIGHT_MEDIUM):
                                        for x in range(0,20):
```

```
self.cmd_vel.publish(turn_cmd_inv)
                                                r.sleep()
                                        for x in range(0,10):
                                                r.sleep()

                                else:
                                        # elif (self.LASER_STATE ==
Move.STATE_TURN_RIGHT_LARGE):
                                        for x in range(0,30):

self.cmd_vel.publish(turn_cmd_inv)
                                                r.sleep()
                                        for x in range(0,10):
                                                r.sleep()

                                # Update flag again
                                self.LASER_STATE = self.LASER_FLAG
                r.sleep()

    ############################
    # Setting up for functions
    ############################
    def reset_sect(self):
        self.sect_1 = 0.0
        self.sect_2 = 0.0
        self.sect_3 = 0.0
        self.sect_4 = 0.0
        self.sect_5 = 0.0

    def sort(self, laserscan):
        entries = len(laserscan.ranges)
        entry_per_sect = entries / 7.0
        #for entry in range(0,entries):

        for i, entry in enumerate(laserscan.ranges):
                # Professor said that we should assume NAN as too far
away
                # Value ranges from 0.45 m to 10m
                # I use 5 out of 7 sections. Just cuz
                entry_i = entry if (not math.isnan(entry)) else 0.0
                self.sect_1 += entry_i if (entries/7 < i < 2*entries/
7) else 0
                self.sect_2 += entry_i if (2*entries/7 < i <
3*entries/7) else 0
                self.sect_3 += entry_i if (3*entries/7 < i <
4*entries/7) else 0
                self.sect_4 += entry_i if (4*entries/7 < i <
5*entries/7) else 0
                self.sect_5 += entry_i if (5*entries/7 < i <
6*entries/7) else 0
```

```
        self.sect_1 = float(self.sect_1/entry_per_sect)
        self.sect_2 = float(self.sect_2/entry_per_sect)
        self.sect_3 = float(self.sect_3/entry_per_sect)
        self.sect_4 = float(self.sect_4/entry_per_sect)
        self.sect_5 = float(self.sect_5/entry_per_sect)
        #rospy.loginfo("sort complete,sect_1: " +
'{0:.2f}'.format(self.sect_1) + " sect_2: " +
'{0:.2f}'.format(self.sect_2) + " sect_3: " +
'{0:.2f}'.format(self.sect_3) + " sect_4: " +
'{0:.2f}'.format(self.sect_4) + " sect_5:" +
'{0:.2f}'.format(self.sect_5))

    def laser_update(self):
        self.b1 = 1 if self.sect_1 < self.threshold else 0
        self.b2 = 1 if self.sect_2 < self.threshold else 0
        self.b3 = 1 if self.sect_3 < self.threshold else 0
        self.b4 = 1 if self.sect_4 < self.threshold else 0
        self.b5 = 1 if self.sect_5 < self.threshold else 0
        sect = int(str(self.b1) + str(self.b2) + str(self.b3) +
str(self.b4) + str(self.b5))
        sect_str = str(self.b1) + str(self.b2) + str(self.b3) +
str(self.b4) + str(self.b5)
        #rospy.loginfo("Sect = " + sect_str)
        #rospy.loginfo("LASER_FLAG is " + str(self.states[sect]))
        self.LASER_FLAG = self.states[sect]
        self.reset_sect()

    def call_back(self,laserscan):
        self.sort(laserscan)
        self.laser_update()


    def BumperEventCallback(self,data):
                if (data.state == BumperEvent.PRESSED):
                        # sleep for 2 seconds, if the bumper is
pressed
                        # publish the velocity
                        r =rospy.Rate(1)
                        rospy.loginfo("Bumper Pressed")
                        self.BUMPER_FLAG = Move.STATE_STOPPED

                if (data.state == BumperEvent.RELEASED):
                        r =rospy.Rate(1)
                        rospy.loginfo("Bumper Released")
                        self.BUMPER_FLAG = Move.STATE_FORWARD
    def CliffEventCallback(self,data):
                if (data.state == CliffEvent.CLIFF):
                        r =rospy.Rate(1)
                        rospy.loginfo("CliffEvent Triggered")
                        self.CLIFF_FLAG = Move.STATE_STOPPED
```

```
                else:
                        r =rospy.Rate(1)
                        self.CLIFF_FLAG = Move.STATE_FORWARD
    def WheelDropEventCallback(self,data):
                if (data.state == WheelDropEvent.RAISED):
                        r =rospy.Rate(1)
                        rospy.loginfo("WheelDropEvent Triggered")
                        self.WHEELDROP_FLAG = Move.STATE_STOPPED
                else:
                        r =rospy.Rate(1)
                        self.WHEELDROP_FLAG = Move.STATE_FORWARD


                self.CLIFF_FLAG = Move.STATE_FORWARD

    def shutdown(self):
        # stop turtlebot
        rospy.loginfo("Stop TurtleBot")
        # a default Twist has linear.x of 0 and angular.z of 0.  So
it'll stop TurtleBot
        self.cmd_vel.publish(Twist())
        # sleep just makes sure TurtleBot receives the stop command
prior to shutting down the script
        rospy.sleep(1)



if __name__ == '__main__':
        Move()
```