```python
# EKF SLAM - prediction, landmark assignment and correction.
#
# slam_09_c_slam_correction
# Claus Brenner, 20 JAN 13
from lego_robot import *
from math import sin, cos, pi, atan2, sqrt
from numpy import *
from slam_f_library import get_observations, write_cylinders,\
    write_error_ellipses


class ExtendedKalmanFilterSLAM:
    def __init__(self, specific_state, covariance,
                 robot_width, scanner_displacement,
                 control_motion_factor, control_turn_factor,
                 measurement_distance_stddev, measurement_angle_stddev):
        # The state. This is the core data of the Kalman filter.
        self.specific_state = specific_state
        self.covariance = covariance

        # Some constants.
        self.robot_width = robot_width
        self.scanner_displacement = scanner_displacement
        self.control_motion_factor = control_motion_factor
        self.control_turn_factor = control_turn_factor
        self.measurement_distance_stddev = measurement_distance_stddev
        self.measurement_angle_stddev = measurement_angle_stddev

        # Currently, the number of landmarks is zero.
        self.number_of_landmarks = 0

    @staticmethod
    def g(specific_state, control, w):
        x, y, theta = specific_state
        l, r = control
        if r != l:
```

```
            alpha = (r - l) / w
            rad = l/alpha
            g1 = x + (rad + w/2.)*(sin(theta+alpha) - sin(theta))
            g2 = y + (rad + w/2.)*(-cos(theta+alpha) + cos(theta))
            g3 = (theta + alpha + pi) % (2*pi) - pi
        else:
            g1 = x + l * cos(theta)
            g2 = y + l * sin(theta)
            g3 = theta

        return array([g1, g2, g3])

    @staticmethod
    def dg_dstate(specific_state, control, w):
        theta = specific_state[2]
        l, r = control
        if r != l:
            alpha = (r-l)/w
            theta_ = theta + alpha
            rpw2 = l/alpha + w/2.0
            m = array([[1.0, 0.0, rpw2*(cos(theta_) - cos(theta))],
                       [0.0, 1.0, rpw2*(sin(theta_) - sin(theta))],
                       [0.0, 0.0, 1.0]])
        else:
            m = array([[1.0, 0.0, -l*sin(theta)],
                       [0.0, 1.0,  l*cos(theta)],
                       [0.0, 0.0,  1.0]])
        return m

    @staticmethod
    def dg_dcontrol(specific_state, control, w):
        theta = specific_state[2]
        l, r = tuple(control)
        if r != l:
            rml = r - l
            rml2 = rml * rml
            theta_ = theta + rml/w
            dg1dl = w*r/rml2*(sin(theta_)-sin(theta))  - (r+l)/(2*rml)*cos(theta_)
```

```
            dg2dl = w*r/rml2*(-cos(theta_)+cos(theta)) - (r+l)/(2*rml)*sin(theta_)
            dg1dr = (-w*l)/rml2*(sin(theta_)-sin(theta)) + (r+l)/(2*rml)*cos(theta_)
            dg2dr = (-w*l)/rml2*(-cos(theta_)+cos(theta)) + (r+l)/(2*rml)*sin(theta_)

        else:
            dg1dl = 0.5*(cos(theta) + l/w*sin(theta))
            dg2dl = 0.5*(sin(theta) - l/w*cos(theta))
            dg1dr = 0.5*(-l/w*sin(theta) + cos(theta))
            dg2dr = 0.5*(l/w*cos(theta) + sin(theta))

        dg3dl = -1.0/w
        dg3dr = 1.0/w
        m = array([[dg1dl, dg1dr], [dg2dl, dg2dr], [dg3dl, dg3dr]])

        return m

    def predict(self, control):
        """The prediction step of the Kalman filter."""
        # covariance' = G * covariance * GT + R
        # where R = V * (covariance in control space) * VT.
        # Covariance in control space depends on move distance.

        # --->>> Put your code here.

        # Hints:
        # - The number of landmarks is self.number_of_landmarks.
        # - eye(n) is the numpy function which returns a n x n identity matrix.
        # - zeros((n,n)) returns a n x n matrix which is all zero.
        # - If M is a matrix, M[0:2,1:5] returns the submatrix which consists
        #   of the rows 0 and 1 (but not 2) and the columns 1, 2, 3, 4.
        #   This submatrix operator can be used on either side of an assignment.
```

```
        # - Similarly for vectors: v[1:3] returns the vector consisting of the
        #   elements 1 and 2, but not 3.
        # - All matrix and vector indices start at 0.

        G3 = self.dg_dstate(self.specific_state, control, self.robot_width)

        left, right = control
        left_var = (self.control_motion_factor * left)**2 +\
                   (self.control_turn_factor * (left-right))**2
        right_var = (self.control_motion_factor * right)**2 +\
                    (self.control_turn_factor * (left-right))**2
        control_covariance = diag([left_var, right_var])
        V = self.dg_dcontrol(self.specific_state, control, self.robot_width)
        R3 = dot(V, dot(control_covariance, V.T))

        # Now enlarge G3 and R3 to accomodate all landmarks. Then, compute the
        # new covariance matrix self.covariance.
        new_offset = 2*self.number_of_landmarks
        G = zeros((3 + new_offset, 3 + new_offset))
        G[0:3, 0:3] = G3
        G[3:, 3:] = eye(new_offset, new_offset)

        R = zeros((3 + new_offset, 3 + new_offset))
        R[0:3, 0:3] = R3

        self.covariance = dot(G, dot(self.covariance, G.T)) + R

        # specific_state' = g(specific_state, control)
        # In the prediction stage the landmarks' coordinates are not modified.
        # They are copied directly from specific_state t-1 to specific_state t.
        self.specific_state[0:3] = self.g(self.specific_state[0:3], control, self.robot_width)
```

```
            self.robot_width)

    def add_landmark_to_state(self, initial_coords):
        """Enlarge the current state and covariance matrix to include one more
           landmark, which is given by its initial_coords (an (x, y) tuple).
           Returns the index of the newly added landmark."""

        # --->>> Put here your new code to augment the robot's state and
        #        covariance matrix.
        #        Initialize the state with the given initial_coords and the
        #        covariance with 1e10 (as an approximation for "infinity".
        # Hints:
        # - If M is a matrix, use M[i:j,k:l] to obtain the submatrix of
        #   rows i to j-1 and colums k to l-1. This can be used on the left and
        #   right side of the assignment operator.
        # - zeros(n) gives a zero vector of length n, eye(n) an n x n identity
        #   matrix.
        # - Do not forget to increment self.number_of_landmarks.
        # - Do not forget to return the index of the newly added landmark. I.e.,
        #   the first call should return 0, the second should return 1.
        landmark_index = self.number_of_landmarks
        old_offset = 2*self.number_of_landmarks
        self.number_of_landmarks += 1
        new_offset = 2*self.number_of_landmarks

        covarianceN = zeros((3 + new_offset, 3 + new_offset))
        covarianceN[0:3+old_offset, 0:3+old_offset] = self.covariance
        covarianceN[-2,-2] = covarianceN[-1,-1] = 1e10
        self.covariance = covarianceN

```

```
        state_prime = zeros(3 + new_offset)
        state_prime[0:3+old_offset] = self.specific_state
        state_prime[-2] = initial_coords[0]
        state_prime[-1] = initial_coords[1]
        self.specific_state = state_prime

        return landmark_index

    @staticmethod
    def h(specific_state, landmark, scanner_displacement):
        """Takes a (x, y, theta) state and a (x, y) landmark, and returns the
           measurement (range, bearing)."""
        dx = landmark[0] - (specific_state[0] + scanner_displacement * cos(specific_state[2]))
        dy = landmark[1] - (specific_state[1] + scanner_displacement * sin(specific_state[2]))
        r = sqrt(dx * dx + dy * dy)
        # alpha in [-pi, pi]
        alpha = (atan2(dy, dx) - specific_state[2] + pi) % (2*pi) - pi

        return array([r, alpha])

    @staticmethod
    def dh_dstate(specific_state, landmark, scanner_displacement):
        theta = specific_state[2]
        cost, sint = cos(theta), sin(theta)
        dx = landmark[0] - (specific_state[0] + scanner_displacement * cost)
        dy = landmark[1] - (specific_state[1] + scanner_displacement * sint)
        q = dx * dx + dy * dy
        sqrtq = sqrt(q)
        drdx = -dx / sqrtq
        drdy = -dy / sqrtq
        drdtheta = (dx * sint - dy * cost) * scanner_displacement / sqrtq
        dalphadx =  dy / q
        dalphady = -dx / q
        dalphadtheta = -1 - scanner_displacement / q * (dx * cost + dy * sint)
```

```python

        return array([[drdx, drdy, drdtheta],
                      [dalphadx, dalphady, dalphadtheta]])

    def correct(self, measurement, landmark_index):
        """The correction step of the Kalman filter."""
        # Get (x_m, y_m) of the landmark from the state vector.
        index = 3+2*landmark_index
        landmark = self.specific_state[index : index+2]
        H3 = self.dh_dstate(self.specific_state, landmark, self.scanner_displacement)

        # --->>> Add your code here to set up the full H matrix.
        H = zeros((2, 3 + 2*self.number_of_landmarks))
        H[:, :3] = H3
        H[:, index:index+2] = -H3[:, 0:2]

        # This is the old code from the EKF - no modification necessary!
        Q = diag([self.measurement_distance_stddev**2,
                  self.measurement_angle_stddev**2])
        K = dot(self.covariance,
                dot(H.T, linalg.inv(dot(H, dot(self.covariance, H.T)) + Q)))
        innovation = array(measurement) -\
                     self.h(self.specific_state, landmark, self.scanner_displacement)
       # innovation[1] in [-pi, pi]
        innovation[1] = (innovation[1] + pi) % (2*pi) - pi
        self.specific_state = self.specific_state + dot(K, innovation)
        self.covariance = dot(eye(size(self.specific_state)) - dot(K, H),
                          self.covariance)

    def get_landmarks(self):
        """Returns a list of (x, y) tuples of all landmark positions."""
        return ([(self.specific_state[3+2*j],
                  self.specific_state[3+2*j+1])
```

```
        self.specific_state[3+2*j+1])
                    for j in xrange(self.number_of_landmarks)])

    def get_landmark_error_ellipses(self):
        """Returns a list of all error ellipses, one for each landmark."""
        ellipses = []
        for i in xrange(self.number_of_landmarks):
            j = 3 + 2 * i
            ellipses.append(self.get_error_ellipse(
                self.covariance[j:j+2, j:j+2]))
        return ellipses

    @staticmethod
    def get_error_ellipse(covariance):
        """Return the position covariance (which is the upper 2x2 submatrix)
           as a triple: (main_axis_angle, stddev_1, stddev_2), where
           main_axis_angle is the angle (pointing direction) of the main axis,
           along which the standard deviation is stddev_1, and stddev_2 is the
           standard deviation along the other (orthogonal) axis."""
        eigenvals, eigenvects = linalg.eig(covariance[0:2,0:2])
        angle = atan2(eigenvects[1,0], eigenvects[0,0])
        return (angle, sqrt(eigenvals[0]), sqrt(eigenvals[1]))


if __name__ == '__main__':
    # Robot constants.
    scanner_displacement = 30.0
    ticks_to_mm = 0.349
    robot_width = 155.0

    # Cylinder extraction and matching constants.
    minimum_valid_distance = 20.0
    depth_jump = 100.0
    cylinder_offset = 90.0
    max_cylinder_distance = 500.0
```

```
    # Filter constants.
    control_motion_factor = 0.35  # Error in motor control.
    control_turn_factor = 0.6  # Additional error due to slip when turning.
    measurement_distance_stddev = 600.0  # Distance measurement error of cylinders.
    measurement_angle_stddev = 45. / 180.0 * pi  # Angle measurement error.

    # Arbitrary start position.
    initial_state = array([500.0, 0.0, 45.0 / 180.0 * pi])

    # Covariance at start position.
    initial_covariance = zeros((3,3))

    # Setup filter.
    slam_ekf = ExtendedKalmanFilterSLAM(initial_state, initial_covariance,
                                        robot_width, scanner_displacement,
                                        control_motion_factor, control_turn_factor,
                                        measurement_distance_stddev,
                                        measurement_angle_stddev)

    # Read data.
    logfile = LegoLogfile()
    logfile.read("robot4_motors.txt")
    logfile.read("robot4_scan.txt")

    # Loop over all motor tick records and all measurements and generate
    # filtered positions and covariances.
    # This is the EKF SLAM loop.
    f = open("ekf_slam_correction.txt", "w")
    for i in xrange(len(logfile.motor_ticks)):
        # Prediction.
        control = array(logfile.motor_ticks[i]) * ticks_to_mm
```

```
        slam_ekf.predict(control)

        # Correction.
        observations = get_observations(logfile.scan_data[i], depth_jump, minimum_valid_distance, cylinder_offset, slam_ekf, max_cylinder_distance)
        for obs in observations:
            measurement, cylinder_world, cylinder_scanner, cylinder_index = obs
            if cylinder_index == -1:
                cylinder_index = slam_ekf.add_landmark_to_state(cylinder_world)
            slam_ekf.correct(measurement, cylinder_index)

        # End of EKF SLAM - from here on, data is written.

        # Output the center of the scanner, not the center of the robot.
        print >> f, "F %f %f %f" % \
            tuple(slam_ekf.specific_state[0:3] + [scanner_displacement * cos(slam_ekf.specific_state[2]),
                                                  scanner_displacement * sin(slam_ekf.specific_state[2]),
                                                  0.0])
        # Write covariance matrix in angle stddev1 stddev2 stddev-heading form.
        e = ExtendedKalmanFilterSLAM.get_error_ellipse(slam_ekf.covariance)
        print >> f, "E %f %f %f %f" % (e + (sqrt(slam_ekf.covariance[2,2]),))
        # Write estimates of landmarks.
        write_cylinders(f, "W C", slam_ekf.get_landmarks())
        # Write error ellipses of landmarks.
        write_error_ellipses(f, "W E", slam_ekf.get_landmark_error_ellipses())
```

```
        # Write cylinders detected by the scanner.
        write_cylinders(f, "D C", [(obs[2][0], obs[2][1])
                                   for obs in observations])

    f.close()

```