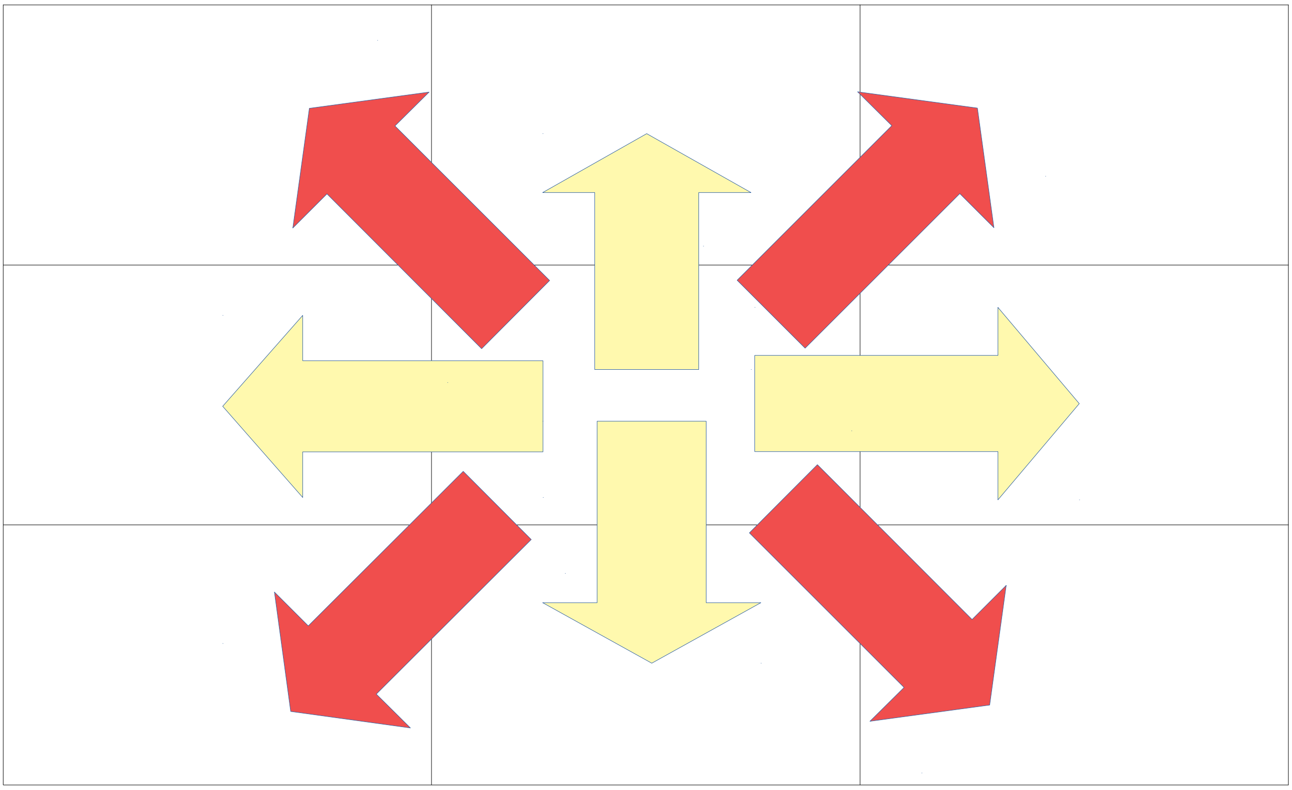
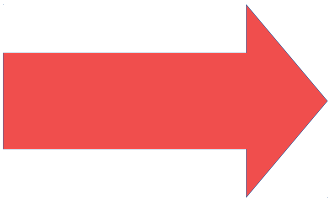
Q1.

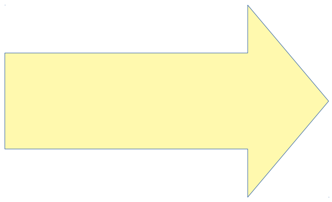
```
>>> f0
31.798989873223331
>>> f1
31.798989873223334
```

Q2.

a)



 = Cardinal

 = Ordinal

b)
It compares both the number of ordinals and cardinal moves of the current cell with the given distance.

c)
```
__add__(self, distance)
```

d)
```
from_separation(idx0, idx1)
```

Q3)

a)
g_cost = f_cost = inf , h_cost = 0

b)
When the cell is unknown, free and not inflated.

Q4)

a)
(1,4) is an idx while (1.23, 3.921) is a pos. Both tuple can be pos.

b)
occ_grid.idx2cell((1,2)).is_occupied()

c)
`update_at_idx(self, idx, occupied)`

d)
`mask_idx`

Q5)

```python
def get_path(self, start_idx, goal_idx):
    """ Returns a list of indices that represents the octile-optimal
        path between the starting index and the goal index
    Parameters:
        start_idx (tuple of int64): the starting index
        goal_idx (tuple of int64): the goal index
    Returns:
        list of tuple of int64: contains the indices in the optimal path
    """
    occ_grid = self.occ_grid
    open_list = self.open_list
    # get number of rows ni (x) and number of columns nj (y)
    ni, nj = occ_grid.num_idx
    path = []

    # resets h-cost, g-cost, update and occ for all cells
    for i in xrange(ni):
        for j in xrange(nj):
            # ! use occ_grid.idx2cell() and the cell's reset_for_planner()
            # pass
            if occ_grid.idx2cell((i,j)) != None:
                occ_grid.idx2cell((i,j)).reset_for_planner(goal_idx)

    # put start cell into open list
    # ! get the start cell from start_idx
    # ! set the start cell distance using set_g_cost and Distance(0, 0)
    # ! add the cell to open_list

    # ! get the start cell from start_idx
    start_cell = occ_grid.idx2cell(start_idx)
    # ! set the start cell distance using set_g_cost and Distance(0, 0)
    start_cell.set_g_cost(Distance(0, 0))
    # ! add the cell to open_list
    open_list.add(start_cell)

    # now we non-recursively search the map
    while open_list.not_empty():
        cell = open_list.remove()
        # skip if already visited, bcos a cheaper path was already found
        if cell.visited:
            continue

        # ! set the cell as visited
        cell.visited = True
```

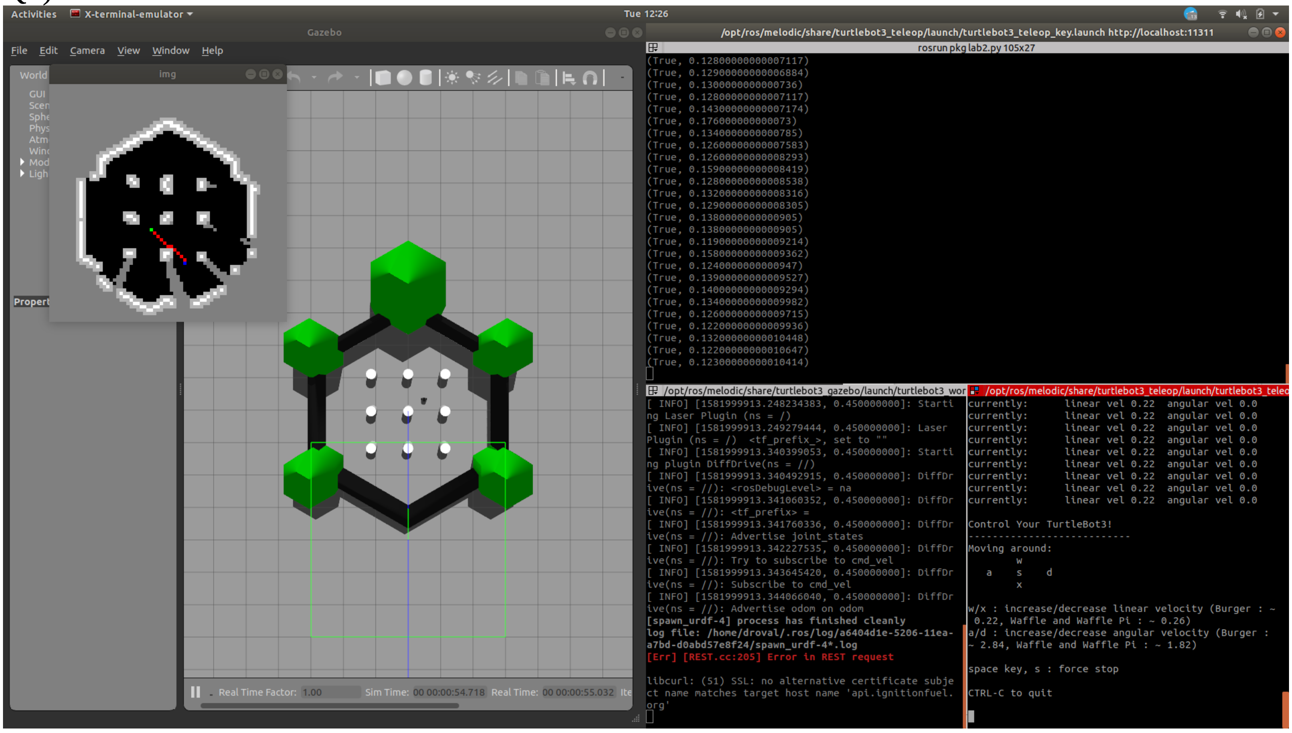
```
        # goal
        if cell.idx == goal_idx:
            # while cell.parent is not None:
            #     # ! append the cell.idx onto path
            #     # ! let cell = cell's parent
            #     # ! if cell is None, break out of the while loop
            #     # pass
            #     path.append(cell.idx)
            #     cell = cell.parent
            # path.append(cell.idx)
            while True:
                path.append(cell.idx)
                cell = cell.parent
                if cell == None:
                    break

            break # breaks out of the loop: while open_list.not_empty()

        # if not goal or not visited, we try to add free neighbour cells into the open list
        for nb_cell in self.get_free_neighbors(cell):
            # ! calculate the tentative g cost of getting from current cell (cell) to neighbouring cell (nb_cell)...
            # !     use cell.g_cost and Distance.from_separation()
            # ! if the tentative g cost is less than the nb_cell.g_cost, ...
            # !     1. assign the tentative g cost to nb_cell's g cost using set_g_cost
            # !     2. set the nb_cell parent as cell
            # !     3. add the nb_cell to the open list using open_list.add()
            # pass
            tent_g_cost = Distance.from_separation(nb_cell.idx, cell.idx) + cell.g_cost
            if tent_g_cost < nb_cell.g_cost:
                nb_cell.set_g_cost(tent_g_cost)
                nb_cell.parent = cell
                open_list.add(nb_cell)

    return path
```

Q6)

```python
def add(self, cell):
    """ Adds the cell and sorts it based on its f-cost followed by the h-cost
    Parameters:
        cell (Cell): the Cell to be sorted, updated with f-cost and h-cost information
    """
    # set l as the open list
    l = self.l

    # if l is empty, just append and return
    if not l:
        l.append(cell)
        return

    # now we sort and add
    i = 0; nl = len(l)
    # we start searching from index (i) 0, where the cells should be cheapest
    while i < nl:
        # ! get the cell (list_cell) in the index (i) of the open list (l)
        # ! now if the cell's f_cost is less than the list_cell's f_cost, ...
        # !     or if the cell's f_cost = list_cell's f_cost but ...
        # !     cell's h_cost is less than the list_cell's h_cost...
        # !     we break the loop (while i < nl)
        if cell.f_cost < l[i].f_cost :
            break
        elif cell.f_cost== l[i].f_cost and cell.h_cost < l[i].h_cost :
            break
        # increment the index
        i += 1

    # insert the cell into position i of l
    l.insert(i, cell)
```

```python
def remove(self):
    """ Removes and return the cheapest cost cell in the open list
    Returns:
        Cell: the cell with the cheapest f-cost followed by h-cost
    """
    # return the first element in self.l
    # pass
    return self.l.pop(0)
```

Q7)


Activities
X-terminal-emulator
Tue 12:26
Gazebo
File Edit Camera View Window Help
/opt/ros/melodic/share/turtlebot3_teleop/launch/turtlebot3_teleop_key.launch http://localhost:11311
rosrun pkg lab2.py 105x27
Control Your TurtleBot3!
Moving around:
space key, s : force stop
CTRL-C to quit


rosrun pkg lab2.py
rosrun pkg lab2.py 105x27
(True, 0.12500000000004619)
(True, 0.12400000000004496)
(True, 0.11600000000004584)
(True, 0.1190000000000424)
(True, 0.13900000000004553)
(True, 0.1370000000000502)
(True, 0.14200000000005275)
(True, 0.1160000000000494)
(True, 0.1230000000000544)
(True, 0.13100000000005707)
(True, 0.12100000000005906)
(True, 0.1470000000000553)
(True, 0.12400000000005917)
(True, 0.12800000000005696)
(True, 0.13500000000006196)
(True, 0.12400000000006628)
(True, 0.13100000000006418)
(True, 0.12100000000006617)
(True, 0.1300000000000665)
(True, 0.1370000000000715)
(True, 0.1270000000000735)
(True, 0.1370000000000715)
(True, 0.13200000000007606)
(True, 0.1270000000000735)
(True, 0.13700000000007861)
(True, 0.12800000000007827)
img

Q8)

Cell.set_occupancy:

```python
def set_occupancy(self, occupied=True):
    """ Updates the cell's observed occupancy state using the log-odds
        Binary Bayes model
    Parameters:
        occupied (bool):    If True, the cell was observed to be occupied.
                            False otherwise. Default is True
    """
    if occupied == True:
        self.occ += 1
    else:
        self.occ -= 1
    # lab2_aux.set_occupancy(self, occupied)
```

subscribe_wheels:

```python
def subscribe_wheels(msg):
    global rbt_wheels
    # rbt_wheels = lab2_aux.get_wheels(msg)
    right_wheel_angle = msg.position[0] # examine topic /joint_states
    left_wheel_angle = msg.position[1] # examine topic /joint_states
    rbt_wheels = (left_wheel_angle, right_wheel_angle)
```

gen_mask:

```python
def gen_mask(cell_size, radius):
    """ Generates the list of relative indices of neighboring cells which lie within
        the specified radius from a center cell
    Parameters:
        radius (float64): the radius
    """
    nCells_radius = int64(numpy.round(float64(radius) / float64(cell_size)))

    mask = []
    i = 0
    j = 0
    for i in xrange(0, nCells_radius):
        for j in xrange(0, nCells_radius):
            relative_distance_small = sqrt(pow((((cell_size/2)*nCells_radius) - ((cell_size/2)*i)),2)
                                           + pow((((cell_size/2)*nCells_radius) - ((cell_size/2)*j)),2))
            if relative_distance_small < radius:
                mask.append((i ,j))
                mask.append((-i ,j))
                mask.append((i ,-j))
                mask.append((-i ,-j))

    return mask
    # return lab2_aux.gen_mask(cell_size, radius)
```

OdometryMM:

```python
class OdometryMM:
    def __init__(self, initial_pose, initial_wheels, axle_track, wheel_dia):
        self.x = initial_pose[0] # m, robot's x position in world
        self.y = initial_pose[1] # m, robot's y position in world
        self.o = initial_pose[2] # rad, robot's bearing in world
        self.wl = initial_wheels[0] # rad, robot's left angle
        self.wr = initial_wheels[1] # rad, robot's right angle
        self.L = axle_track # m, robot's axle track
        self.WR = wheel_dia/2.0 # m, robot's wheel RADIUS, not DIAMETER
        self.t = rospy.get_time() # s, time last calculated
    def calculate(self, wheels):
        # calculates the robot's new pose based on wheel encoder angles
        # INPUT: wheels: (left_wheel_angle, right_wheel_angle)
        # OUTPUT: a new pose (x, y, theta)

        # previous wheel angles stored in self.wl and self.wr, respectively. Remember to overwrite them
        # previous pose stored in self.x, self.y, self.o, respectively. Remember to overwrite them
        # previous time stored in self.t. Remember to overwrite it
        # axle track stored in self.L. Should not be overwritten.
        # wheel radius, NOT DIAMETER, stored in self.WR. Should not be overwritten.
        dt = rospy.get_time() - self.t # current time minus previous time
        dwl = wheels[0] - self.wl
        dwr = wheels[1] - self.wr

        # Own Implementation of Odometry Motion Model

        # vt = (((self.WR * 2) / (4 * dt)) * (dwl + dwr))
        # dPhi = (((self.WR * 2) / (2 * self.L)) * (dwr - dwl))
        # dwt = (((self.WR * 2) / (2 * self.L * dt)) * (dwr - dwl))

        vt = ((self.WR*2)/4)*((dwr/dt)+(dwl/dt))
        dPhi = ((self.WR*2)/(2*self.L))*(dwr-dwl)
        dwt = ((self.WR*2)/(2*self.L))*(((dwr/dt) - (dwl/dt)))

        if abs(dwt) < 1e-10:
        #   MM for move straight
            self.x = self.x + (vt * dt * math.cos(self.o)) #???
            self.y = self.y + (vt * dt * math.sin(self.o)) #???
            self.o = self.o #???

        else:
        #   MM for curve turns
            rt = ((self.L * (dwr + dwl)) / (2 * (dwr - dwl)))

            self.x = self.x + ((rt * math.sin(self.o + dPhi)) - (rt * math.sin(self.o))) #???
            self.y = self.y + ((rt * math.cos(self.o)) - (rt * math.cos(self.o + dPhi)))  #???
            self.o = self.o + dPhi #???

        # End of Own Implementation

        # self.x = 0 #???
        # self.y = 0 #???
        # self.o = 0 #???
        # if abs(w) < 1e-10:
        #   MM for move straight
        # else:
        #   MM for curve turns
        self.wl = wheels[0]
        self.wr = wheels[1]
        self.t = self.t + dt # update the current time. There's a reason why resampling the time is discouraged
        return (self.x, self.y, self.o)
```

LOS:

```python
class LOS:
    def __init__(self, map):
        self.pos2idx = map.pos2idx # based on the map (occ_grid) it return's the map index represent
        # use self.pos2idx(pos, False) to return the exact index representation, including values th
        # use self.pos2idx(pos) to return the integer index representation, which is the rounded ver
    def calculate(self, start_pos, end_pos):
        # sets up the LOS object to prepare return a list of indices on the map starting from start_
        # start_pos is the robot position.
        # end_pos is the maximum range of the LIDAR, or an obstacle.
        # every index returned in the indices will be the index of a FREE cell
        # you can return the indices, or update the cells in here
        start_idx = self.pos2idx(start_pos)
        end_idx = self.pos2idx(end_pos)
        indices = [] # init an empty list

        # Start of General Line Algo

        # Getting Difference between start and final pose
        xi = float64(start_idx[0])
        yi = float64(start_idx[1])

        xf = float64(end_idx[0])
        yf = float64(end_idx[1])

        deltaX = xf - xi
        deltaY = yf - yi

        if (abs(deltaX) > abs(deltaY)):
            deltaLong = deltaX
            deltaShort = deltaY
            get_idx = lambda (intLineLongIni, intLineShortIni) : (intLineLongIni, intLineShortIni)
        else:
            deltaLong = deltaY
            deltaShort = deltaX
            get_idx = lambda (intLineLongIni, intLineShortIni) : (intLineShortIni, intLineLongIni)

        (lineLongIni, lineShortIni) = get_idx((xi, yi))
        (lineLongEnd, lineShortEnd) = get_idx((xf, yf))

        # Get integer (index) representation for accessing map
        intLineLongIni = round(lineLongIni)
        intLineShortIni = round(lineShortIni)
        intLineLongEnd = round(lineLongEnd)
        intLineShortEnd = round(lineShortEnd)

        # Get signs and increments
        delS = sign(deltaShort)
        delL = sign(deltaLong)
        psiS = deltaShort / abs(deltaLong)

        # Get Error
        errorS = lineShortIni - intLineShortIni

        # Get Lamda
        lam = abs(deltaShort / deltaLong) * (0.5 + (lineLongIni - intLineLongIni) * delL) - 0.5

        # Get Error Checker
        if (deltaShort >= 0):
            has_big_error = lambda e_k_s : e_k_s >= 0.5
        else:
            has_big_error = lambda e_k_s : e_k_s < -0.5

        # Initialize Counter
        i = 0
```

```python
        # Propagate
        while (intLineLongIni, intLineShortIni) != (intLineLongEnd, intLineShortEnd):
            # print((intLineLongIni, intLineShortIni), (intLineLongEnd, intLineShortEnd), i)
            i += 1
            if i == 100:
                raise Exception('e')
            intLineLongIni += delL
            errorS += psiS

            if has_big_error(errorS):
                errorS -= delS
                intLineShortIni += delS
                # Previous Cell(s)
                lam_bar = errorS * delS
                if (lam_bar < lam): # Short Direction
                    idx = get_idx((int64(intLineLongIni), int64(intLineShortIni) - int64(delS)))
                    indices.append(idx)
                elif (lam_bar > lam): # Long Direction
                    idx = get_idx((int64(intLineLongIni) - int64(delL), int64(intLineShortIni)))
                    indices.append(idx)
                else: # Both Directions
                    idx = get_idx((int64(intLineLongIni) - int64(delL), int64(intLineShortIni)))
                    indices.append(idx)
                    idx = get_idx((int64(intLineLongIni), int64(intLineShortIni) - int64(delS)))
                    indices.append(idx)
            # Current Cell
            idx = get_idx((int64(intLineLongIni), int64(intLineShortIni)))
            indices.append(idx)

        # End of General Line Algo


        # indices.append(start_idx) # append the starting index into the cell
        indices.pop()
        return indices
```

inverse_sensor_model:

```python
def inverse_sensor_model(rng, deg, pose):
    # degree is the bearing in degrees # convert to radians
    # range is the current range data at degree
    # pose is the robot 3DOF pose, in tuple form, (x, y, o)
    x, y, o = pose
    xk = x + rng * math.cos(o + radians(deg)) #???
    yk = y + rng * math.sin(o + radians(deg)) #???
    return (xk, yk)
```

MAX_RNG:

```python
MAX_RNG = 3.5
```