CODE SNIPPETS:

Control.py

```
def exploreEnvironment():
    global isObstacle
    global isBug
    global isGoalset
    global isNewTrajectoryReady

    publish_motion = rospy.Publisher('/cmd_vel_mux/input/teleop', Twist, queue_size=5)
    TBot = TurtlebotActions(publish_motion, ANGLE_ERR, POS_ERR)
    visitedPrev = 0
    visited_threshold = 5
    flag = 0
    while not rospy.is_shutdown() and not isExit:
        if isBug:
            isGoalset = False
            isNewTrajectoryReady = False
            isObstacle = False
            isBug = False

        print "---New Traversal---"
        print "1) Search Around."
        TBot.rotateSelf()

            if flag == 0:
            print "2) Determining the destination point using Random Walk Algorithm."
            else:
                print "2) Determining the destination point using biggest cluster Centroid Algorithm."
        try:
            curPos = Point(x=cur_x, y=cur_y, z=0)
            #g = Int8(data=10)
            Resp = getDestination(map, curPos, flag)

            diff = Resp.vislen
            #print "*******%i******" %diff
                diff = diff - visitedPrev
            if diff > visited_threshold:
                        flag = 1
            else :
                flag = 0
                visitedPrev = Resp.vislen
        except rospy.ServiceException, e:
            print "getDestination() call failed: %s" % e
            isBug = True
            isNewTrajectoryReady = True
```

```
        continue

    if not Resp.foundDest.data:
        break

    print "3) Finding Path to destination"

    global TempGoal
    TempGoal = False
    Thread(target=requestTrajectory, name="requestTrajectory() Thread", args=[Resp.dest]).start()
    executeTrajectory(TBot)

    print "----Completed the Traversal---"
```

Thus, the control.py is responsible for navigating the TurtleBot from the currentPosition to the goal position based on the received input. It is also responsible with any functionality that relates to controlling the TurtleBot, for instance rotating the TurtleBot in step 1 for exploring the map surrounding the TurtleBot.

This code is also responsible for keeping track of the number of visited nodes in the previous iteration. And it also holds a threshold value for (visited_threshold as discussed in the project report) for checking the number of new nodes that are being visited using the exploration in the current algorithm is either lesser or greater than the threshold. This is one of the factor based on which the algorithm switched back and forth the Probabilistic random algorithm or the centroid based frontier exploration. The execution of the current algorithm is determined based on the flag value, meaning flag = 0 denotes probabilistic random exploration and flag =1 denotes the Centroid based frontier exploration.

Mapping.py

```
def getDestination(req):
    global grid
    print "Querying destination."

    grid = processOccupancyGrid(req.map, 4, True)

    foundDest = False

    visited = set()
    clusters = []

    print "Started to look for clusters..."
    flag = 0
    threshold = fitting()*3/4
    prob_dist = 0

    for cell in grid.empty:
        cluster = []


        expandCluster(cell, cluster, visited)

        if len(cluster) != 0:
            clusters.append(cluster)

    print "DONE"

    print "Finding destination..."

    destPos = Point()
    s = len(clusters)
    j = random.randint(1, s)
    i = 0
    if req.flag == 0 and len(clusters) != 0:
            dest = calculateDest(clusterCell)
            destValue = grid.getCellValue(dest)
                i = i+1

                if j == i:
                        path = TurtleBotPF.findRoute(grid, dest, [GridPat.Open], (req.curPos.x, req.curPos.y))
                        dest = path.pop(len(path) - 1)
                        destPos = convertCellToPoint(dest, grid.cellOrigin, grid.resolution)
                        getWaypoints(req.curPos, destPos, grid)
                        prob_dist = distance(req.curPos.x,req.curPos.y,destPos.x,destPos.y)
```

```
                foundDest = True
                break
else:
      if len(clusters) != 0:
      clusters.sort(key = lambda tup: len(tup), reverse=True)

         for curCluster in clusters:
            dest = calculateDest(curCluster)
            destValue = grid.getCellValue(dest)

            try:
               if destValue == GridPat.Obstacle or destValue == GridPat.Unknown:
                  path = TurtleBotPF.findRoute (grid, dest,  [GridPat .Open], (req.curPos.x,
req.curPos.y))
                  dest = path.pop(len(path) - 1)

               dest = TurtleBotPF.findRouteNearest(grid, dest, GridPat.Open, [GridPat .Unknown])

               destPos = convertCellToPoint(dest, grid.cellOrigin, grid.resolution)

               getWaypoints(req.curPos, destPos, grid)
            except Exception, e:
               print "Exception thrown on evaluating destination: ", str(e)
               print "Skipping to the next cluster..."
               continue
               else:
                  foundDest = True
                  break


if len(clusters) != 0:
         clusterCells = []
         for cluster in clusters:
               clusterCells += cluster

         publishGridCells(cluster_cell_pub, clusterCells, grid.resolution, grid.cellOrigin)
         publishGridCells(destination_cell_pub, [dest], grid.resolution, grid.cellOrigin)
         publishGridCells(empty_cell_pub, grid.empty, grid.resolution, grid.cellOrigin)
         publishGridCells(obstacle_cell_pub, grid.obstacles, grid.resolution, grid.cellOrigin)

print "DONE"

print "Done with the destination request processing!"

gc.collect()

vislen = len(visited)

if prob_dist > threshold:
```

```
        vislen = MAX
    return DestinationResponse(Bool(data=foundDest), destPos, vislen)
```

NOTE: The lines highlighted in yellow are single line code.

The mapping.py script is responsible for operating on the occupancy grids of the explored portion of the map to identify the type of cells (frontier, explored, unexplored). Based on the type of the cell, the occupancy grid map can evaluate the next frontier position to be chosen.

Further, for executing the probabilistic technique I implemented a random function that randomly chooses one of the frontier points. For the centroid based technique, first all the frontier cells were found and its corresponding unexplored neighbor cells using the *expandCluster*() function. And then the found cluster sets are sorted based on length of each cluster. Then the centroid for the cluster with maximum number of unexplored neighboring cells is chosen and passed to the *exploreEnvironment*() (in control.py) as the goal position. Also, the number of visited nodes in the current iteration is passed to the *exploreEnvironment*() (in control.py).

Further, the *fitting*() function identifies all the visited nodes and tries to fit into a rectangle and the corresponding length of the rectangle is passed back to the function. Thus this threshold and the distance between the current position and the goal position is compared to decide whether the algorithm should switch from the random walk algorithm to frontier based exploration algorithm.