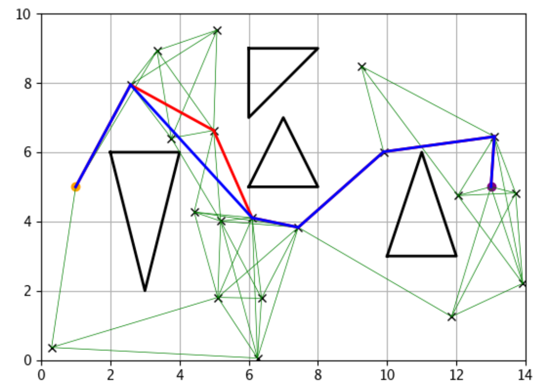
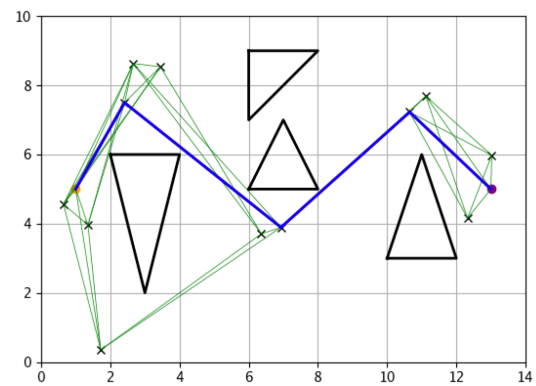
CS 133b **January 23, 2024**

Problem Set 2

Problem 1 (2D Point Planner) - 27 points:

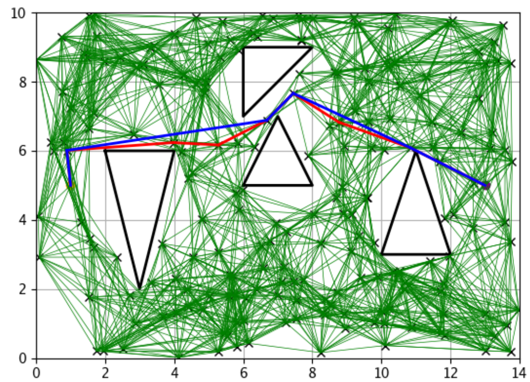


part (a)
For K = 5, 20 nodes were needed to find a path consistently (about 82% of the time). The code is shown on the next few pages. An example of output is shown above.



part (b)
For K = 10, 12 nodes were needed to find a path consistently (about 85% of the time). The code is shown on

the next few pages. I would argue that K=10 with N=12 is better than K=5, and N=20, since there are significantly less nodes and the time to perform each is not much different. An example of output is shown above.



part (c)
With K = 20, at least 200 nodes were needed to find a path consistently (about 83% of the time). The code is shown on the next few pages. An example of output is shown above.
Code for problem 1a, 1b, 1c :

```
    def distance(self, other):
        #FIXME: compute and return the distance.
        dist = sqrt((self.x - other.x)**2 + (self.y - other.y)**2)
        return dist
```

```
# Create the list of nodes.
def createNodes(N):
    # Add nodes sampled uniformly across the space.
    nodes = []
    #FIXME: create the list of valid nodes sampling uniformly in x and y.
    while len(nodes) < N:
        x_coord = random.uniform(xmin, xmax)
        y_coord = random.uniform(ymin, ymax)
        node = Node(x_coord, y_coord)
        if node.inFreespace():
            nodes.append(node)
    return nodes
```

```
# Post Process the Path
def PostProcess(path):
    #FIXME: Remove nodes in the path than can be skipped without collisions
    ref_node = path[0]
    skipped_nodes = []
    if len(path) > 2:
        for i in range(1, len(path)-1):
            if ref_node.connectsTo(path[i+1]):
                skipped_nodes.append(path[i])
            else:
                ref_node = path[i]
        for node in skipped_nodes:
            path.remove(node)
```

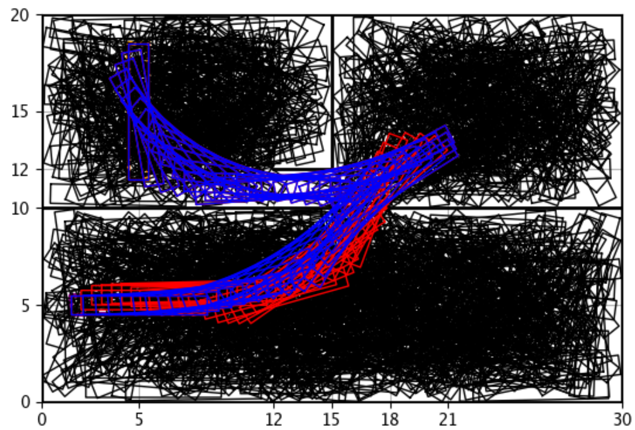
Code to get success rate for problem 1a and 1b

```python
def main():
    success_rate = 0.0
    num_samples = 10000
    for i in range(num_samples):
        # Report the parameters.
        print('Running with', N, 'nodes and', K, 'neighbors.')

        # Create the start/goal nodes.
        startnode = Node(xstart, ystart)
        goalnode  = Node(xgoal,  ygoal)

        # Create the list of nodes.
        print("Sampling the nodes...")
        tic = time.time()
        nodes = createNodes(N)
        toc = time.time()
        print("Sampled the nodes in %fsec." % (toc-tic))

        # Add the start/goal nodes.
        nodes.append(startnode)
        nodes.append(goalnode)

        # Connect to the nearest neighbors.
        print("Connecting the nodes...")
        tic = time.time()
        connectNearestNeighbors(nodes, K)
        toc = time.time()
        print("Connected the nodes in %fsec." % (toc-tic))

        # Run the A* planner.
        print("Running A*...")
        tic = time.time()
        path = astar(nodes, startnode, goalnode)
        toc = time.time()
        print("Ran A* in %fsec." % (toc-tic))

        # If unable to connect, show the part explored.
        if not path:
            #print("UNABLE TO FIND A PATH")
            pass
        else:
            success_rate += 1.0

    success_rate = success_rate/num_samples
    print("Success rate: {}".format(success_rate))
```

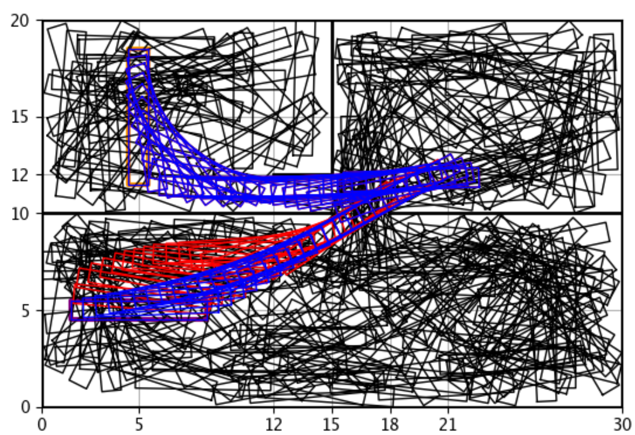
Code to get success rate for problem 1c

```
def main():
    success_rate = 0.0
    num_samples = 10000
    for i in range(num_samples):
        # Report the parameters.
        print('Running with', N, 'nodes and', K, 'neighbors.')

        # Create the start/goal nodes.
        startnode = Node(xstart, ystart)
        goalnode  = Node(xgoal,  ygoal)

        # Create the list of nodes.
        print("Sampling the nodes...")
        tic = time.time()
        nodes = createNodes(N)
        toc = time.time()
        print("Sampled the nodes in %fsec." % (toc-tic))

        # Add the start/goal nodes.
        nodes.append(startnode)
        nodes.append(goalnode)

        # Connect to the nearest neighbors.
        print("Connecting the nodes...")
        tic = time.time()
        connectNearestNeighbors(nodes, K)
        toc = time.time()
        print("Connected the nodes in %fsec." % (toc-tic))

        # Run the A* planner.
        print("Running A*...")
        tic = time.time()
        path = astar(nodes, startnode, goalnode)
        toc = time.time()
        print("Ran A* in %fsec." % (toc-tic))

        # If unable to connect, show the part explored.
        if not path:
            #print("UNABLE TO FIND A PATH")
            pass
        else:
            path_narrow = True
            for node in path:
                if node.y < 5 or node.y >= 9:
                    path_narrow = False
                    break
            if path_narrow:
                success_rate += 1.0

    success_rate = success_rate/num_samples
    print("Success rate: {}".format(success_rate))
```

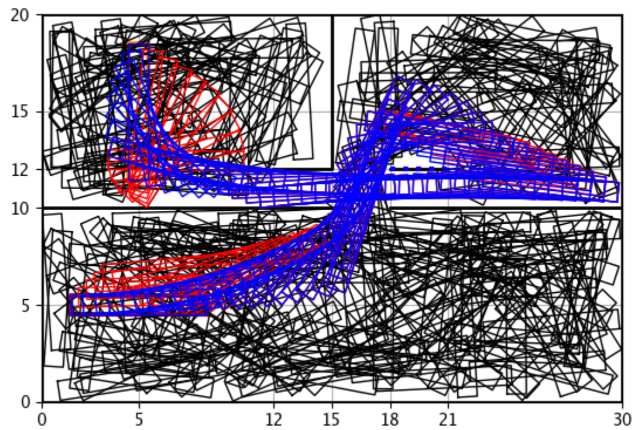
Problem 2 (3D Mattress-Movers Planner) - 32 points:



part (a)
With K = 15, 1100 nodes were needed to find a path consistently (about 82% of the time). The code is shown on the next few pages. An example of output is shown above.

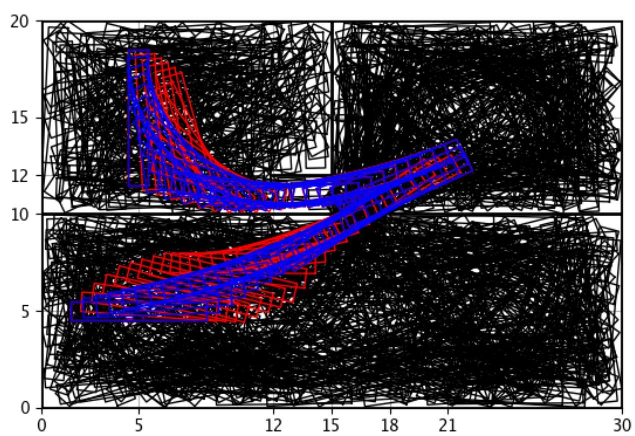


part (b)
With K = 15, 325 nodes were needed to find a path consistently (about 83% of the time). The code is shown on the next few pages. An example of output is shown above.



part (c)

Repeating the non uniform sampling approach of b, both N and K did not need to be increased to keep consistently creating solutions when the small wall is added. In fact the success rate increased by about (1.5%) when N and K were kept the same (N=325, K=15). An example of output is shown above. Code shown on the next few pages



part (d)

To consistently create solutions, K was increased to 25 and N was increased to 1000. The code is shown on the next few pages. An example of output is shown above

Code for problem 2a, 2b, 2c, 2d

Note that for createNodes(), the code used for part a is commented out, the code used for b-d is not commented out in createNodes().

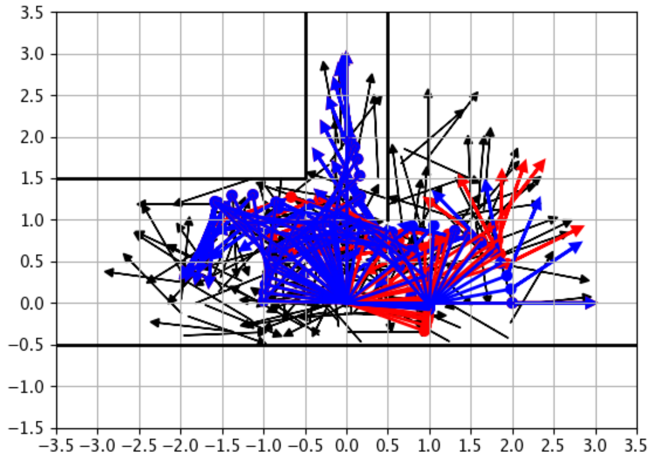
```python
def createNodes(N):
    #FIXME: create the list via (a) uniform sampling and (b) near edges.

    nodes = []

    #part a
    '''while len(nodes) < N:
        x_coord = random.uniform(xmin, xmax)
        y_coord = random.uniform(ymin, ymax)
        theta_coord = random.uniform(0, 2.0*pi)
        node = Node(x_coord, y_coord, theta_coord)
        if node.inFreespace():
            nodes.append(node)'''


    #part b
    q1 = None
    r = 2
    while len(nodes) < N:
        # get q1 (node that is in collision with object)
        while q1 is None:
            x_coord = random.uniform(xmin, xmax)
            y_coord = random.uniform(ymin, ymax)
            theta_coord = random.uniform(0, 2.0*pi)
            node = Node(x_coord, y_coord, theta_coord)
            if not node.inFreespace():
                q1 = node

        #sample about q1
        d = r * sqrt(random.uniform(0,1))
        phi = random.uniform(0, 2.0*pi)
        x_coord = q1.x + d * cos(phi)
        y_coord = q1.y + d * sin(phi)
        theta_coord = random.uniform(0, 2.0*pi)
        node = Node(x_coord, y_coord, theta_coord)
        if node.inFreespace():
            q2 = node
            nodes.append(q2)

        q1 = None


    return nodes
```

```python
# Post Process the Path
def PostProcess(path):
    #FIXME: Remove nodes in the path than can be skipped without collisions
    ref_node = path[0]
    skipped_nodes = []
    if len(path) > 2:
        for i in range(1, len(path)-1):
            if ref_node.connectsTo(path[i+1]):
                skipped_nodes.append(path[i])
            else:
                ref_node = path[i]
        for node in skipped_nodes:
            path.remove(node)
```

Note that the method of connecting neighbors was chosen as determined by the problem (for the code below)

```python
def success_rate():
    success_rate = 0
    num_samples = 1000

    for i in range(num_samples):
        # Report the parameters.
        print("Run number {}".format(i))
        print('Running with', N, 'nodes and', K, 'neighbors.')

        # Create the start/goal nodes.
        startnode = Node(xstart, ystart, tstart)
        goalnode  = Node(xgoal,  ygoal,  tgoal)

        # Create the list of sample points.
        print("Sampling the nodes...")
        tic = time.time()
        nodes = createNodes(N)
        toc = time.time()
        print("Sampled the nodes in %fsec." % (toc-tic))


        # Add the start/goal nodes.
        nodes.append(startnode)
        nodes.append(goalnode)


        # Connect to the nearest neighbors.  FIXME: Switch methods for (d).
        print("Connecting the nodes...")
        tic = time.time()
        #connectNearestNeighbors(nodes, K)
        connectKNeighbors(nodes, K)
        toc = time.time()
        print("Connected the nodes in %fsec." % (toc-tic))

        # Run the A* planner.
        print("Running A*...")
        tic = time.time()
        path = astar(nodes, startnode, goalnode)
        toc = time.time()
        print("Ran A* in %fsec." % (toc-tic))

        # If unable to connect, show the part explored.
        if not path:
            pass
        else:
            success_rate += 1.0

    success_rate = success_rate/num_samples
    print("Success rate: {}".format(success_rate))
```

Problem 3 (3DOF Robot Planner) - 37 points:




```
<Joints    0.0deg,   0.0deg,   0.0deg>
<Joints   -4.6deg,  76.4deg, 162.8deg>
<Joints   -1.8deg, 133.2deg,  71.5deg>
<Joints  100.8deg,  80.3deg,  39.3deg>
<Joints  150.2deg, -14.8deg, 130.0deg>
<Joints  178.5deg,-130.9deg, 123.0deg>
<Joints  101.9deg,-134.6deg, 155.5deg>
<Joints  140.5deg,-118.9deg, 101.8deg>
<Joints   90.0deg,   0.0deg,   0.0deg>
```

part a)
The following constants were used (uniform sampling was used):
Dx = 0.1 (distance to remain from wall in meters)
Dq = Dx/3.0 (joint step size)
To consistently create solutions (about 83% of the time) N was chosen to be 150 and K was chosen to be 6. An example of the output is shown above. The code is shown below and in the rest of the next pages.

```
class Node(AStarNode):
    def __init__(self, q1, q2, q3):
        # Setup the basic A* node.
        super().__init__()

        # FIXME: Finish the initialization.
        #FIXME: Save any states/coordinates you need.
        # joint angles
        self.q1 = q1
        self.q2 = q2
        self.q3 = q3

        # Pre-compute the link positions.
        (self.xA, self.yA) = (          cos(q1)      ,           sin(q1)      )
        (self.xB, self.yB) = (self.xA + cos(q1+q2)    , self.yA + sin(q1+q2)    )
        (self.xC, self.yC) = (self.xB + cos(q1+q2+q3), self.yB + sin(q1+q2+q3))
        self.links = LineString([[0,0], [self.xA,self.yA],
                                 [self.xB,self.yB], [self.xC, self.yC]])
```

Code for 3a (continued):

```python
    def inFreespace(self):
        #FIXME: return True if you are know the arm is not hitting any wall.
        return walls.disjoint(self.links)

    # Check the local planner - whether this connects to another node.
    def connectsTo(self, other):
        #FIXME: return True if you can move without collision.
        #case a)
        if walls.distance(self.links) < Dx or walls.distance(other.links) < Dx:
            return False

        for delta in vandercorput.sequence(Dq / self.distance(other)):
            intermediate_node = self.intermediate(other, delta)
            if not intermediate_node.inFreespace():
                return False
            if walls.distance(intermediate_node.links) < Dx:
                return False

        return True
```

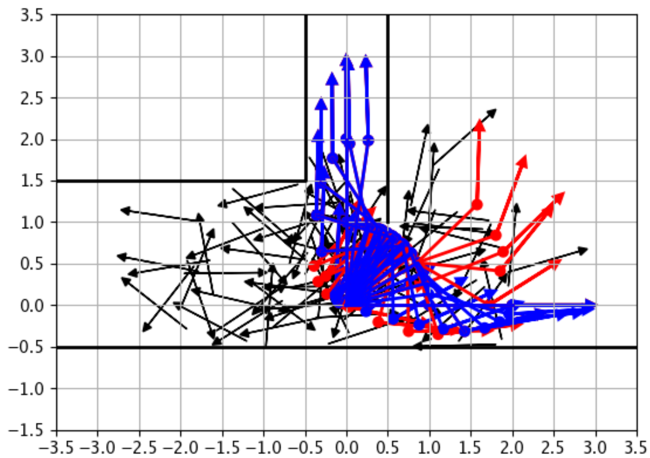
```python
    def intermediate(self, other, alpha):
        #FIXME: Please implement
        #case a)
        return Node(self.q1 + alpha *  (other.q1 - self.q1),
                    self.q2 + alpha *  (other.q2 - self.q2),
                    self.q3 + alpha *  (other.q3 - self.q3))

    # Return a tuple of coordinates, used to compute Euclidean distance.
    def coordinates(self):
        #FIXME: Please implement
        #case a)
        return (self.q1, self.q2, self.q3)

    # Compute the relative distance to another node.  See above.
    def distance(self, other):
        #FIXME: Please implement
        #case a)
        return sqrt((self.q1 - other.q1)**2 + (self.q2 - other.q2)**2 +
                    (self.q3 - other.q3)**2)
```

Code for 3a (continued):

```
def createNodes(N):
    #FIXME: return a list of valid nodes.
    #pass
    nodes = []
    while len(nodes) < N:
        q1_coord = random.uniform(-pi, pi)
        q2_coord = random.uniform(-pi, pi)
        q3_coord = random.uniform(-pi, pi)
        node = Node(q1_coord, q2_coord, q3_coord)
        if node.inFreespace():
            nodes.append(node)

    return nodes

# Connect to the nearest neighbors.
def connectNeighbors(nodes, K):
    #FIXME: determine/set the node.neighbors for all nodes.  Make sure
    #       you create an undirected graph: the neighbors should be
    #       symmetric.  So, if node B becomes a neighbor to node A,
    #       then also add node A to the neighbors of node B.

    # Clear any existing neighbors.  Use a set to add below.
    for node in nodes:
        node.neighbors = set()

    # Report all other nodes, sorted by distance, computed as the
    # Euclidean distance of the coordinates.  This includes the node
    # itself, so ignore the first element below.
    X = np.array([node.coordinates() for node in nodes])
    [dist, idx] = KDTree(X).query(X, k=len(nodes))

    # Check all until we have K neighbors:
    for i, nbrs in enumerate(idx):
        for n in nbrs[1:]:
            if len(nodes[i].neighbors) >= K:
                break
            if nodes[n] not in nodes[i].neighbors:
                if nodes[i].connectsTo(nodes[n]):
                    nodes[i].neighbors.add(nodes[n])
                    nodes[n].neighbors.add(nodes[i])

# Post Process the Path
def PostProcess(path):
    #FIXME: remove unnecessary nodes from the path, if the predecessor and
    #       successor can connect directly.  I.e. minimize the steps.
    ref_node = path[0]
    skipped_nodes = []
    if len(path) > 2:
        for i in range(1, len(path)-1):
            if ref_node.connectsTo(path[i+1]):
                skipped_nodes.append(path[i])
            else:
                ref_node = path[i]
        for node in skipped_nodes:
            path.remove(node)
```

Code for 3a (continued):

```python
def success_rate():
    success_rate = 0
    num_samples = 1000
    for i in range(num_samples):
        # Report the parameters.
        print("Run number {}".format(i))
        print('Running with', N, 'nodes and', K, 'neighbors.')

        # Create the start/goal nodes.
        startnode = Node(startq1, startq2, startq3)
        goalnode  = Node(goalq1,  goalq2,  goalq3)

        # Create the list of sample points.
        print("Sampling the nodes...")
        tic = time.time()
        nodes = createNodes(N)
        toc = time.time()
        print("Sampled the nodes in %fsec." % (toc-tic))

        # Add the start/goal nodes.
        nodes.append(startnode)
        nodes.append(goalnode)

        # Connect to the nearest neighbors.
        print("Connecting the nodes...")
        tic = time.time()
        connectNeighbors(nodes, K)
        toc = time.time()
        print("Connected the nodes in %fsec." % (toc-tic))

        # Run the A* planner.
        print("Running A*...")
        tic = time.time()
        path = astar(nodes, startnode, goalnode)
        toc = time.time()
        print("Ran A* in %fsec." % (toc-tic))

        # If unable to connect, show the part explored.
        if not path:
            pass
        else:
            success_rate += 1

    success_rate = success_rate/num_samples
    print("Success rate: {}".format(success_rate))
```




```
<Joints    0.0deg,    0.0deg,    0.0deg>
<Joints   30.6deg, -86.2deg,   69.0deg>
<Joints   73.7deg,-185.5deg, 151.7deg>
<Joints   38.2deg,-193.7deg, 243.2deg>
<Joints   83.0deg,-361.1deg, 369.9deg>
<Joints   90.0deg,-360.0deg, 360.0deg>
```

part b)
The following constants were used (uniform sampling was used):
Dx = 0.1 (distance to remain from wall in meters)
Dq = Dx/3.0 (joint step size)
To consistently create solutions (about 84% of the time) N was chosen to be 95 and K was chosen to be 6. An example of the output is shown above. The planner does find a solution that wraps up the arm, placing the tip correctly but ending with a joint at $\pm 360°$. The code that is different from part a is shown below and in the rest of the next page.

```
class Node(AStarNode):
    def __init__(self, q1, q2, q3):
        # Setup the basic A* node.
        super().__init__()

        # FIXME: Finish the initialization.
        #FIXME: Save any states/coordinates you need.
        # joint angles
        self.q1 = q1
        self.q2 = q2
        self.q3 = q3

        self.q1_s, self.q1_c = sin(q1), cos(q1)
        self.q2_s, self.q2_c = sin(q2), cos(q2)
        self.q3_s, self.q3_c = sin(q3), cos(q3)

        # Pre-compute the link positions.
        (self.xA, self.yA) = (          cos(q1)      ,           sin(q1)      )
        (self.xB, self.yB) = (self.xA + cos(q1+q2)   , self.yA + sin(q1+q2)   )
        (self.xC, self.yC) = (self.xB + cos(q1+q2+q3), self.yB + sin(q1+q2+q3))
        self.links = LineString([[0,0], [self.xA,self.yA],
                                [self.xB,self.yB], [self.xC, self.yC]])
```

Code for 3b(continued):

```
    def intermediate(self, other, alpha):
        #FIXME: Please implement
        return Node(self.q1 + alpha * wrap(other.q1 - self.q1, 2*pi),
                    self.q2 + alpha * wrap(other.q2 - self.q2, 2*pi),
                    self.q3 + alpha * wrap(other.q3 - self.q3, 2*pi))

    # Return a tuple of coordinates, used to compute Euclidean distance.
    def coordinates(self):
        #FIXME: Please implement
        return (self.q1_c, self.q1_s, self.q2_c, self.q2_s, self.q3_c, self.q3_s)

    # Compute the relative distance to another node.  See above.
    def distance(self, other):
        #FIXME: Please implement
        return sqrt( (self.q1_c - other.q1_c)**2 + (self.q1_s - other.q1_s)**2 +
                     (self.q2_c - other.q2_c)**2 + (self.q2_s - other.q2_s)**2 +
                     (self.q3_c - other.q3_c)**2 + (self.q3_s - other.q3_s)**2 )
```

Problem 4 (Time Spent) - 4 points

I spent about 6 hours on this set.