Coovi Meha

Mid Term Answer

a.) For this vehicle, the motion will be easier to plane in the workspace. Thus, the topology becomes an R^1S^2, that is controlling only the two angles of the agent and r the distance from the start point. The system is them reduced to a 3d coordinate (polar coordinate).

Explanation: The aerial vehicle shows in this problem as 3 coordinates (x, y, z) and 2 rotational angles. Thus, the workspace of the will have a topology od R^3S^2. It would be difficult and required a lot of computer resources to plane the motion this as we have about 5 parameters to tweak.

However, from the equations given (dynamics of the system), the controller inputs, the constrains, with careful examination; one can see that the system can be controlled using the two angles of the rotation and a radius. In fact, the input to the system is the rate of change of the angles and the acceleration. One can reverse calculate the coordinate of the agent at every sampled angle and check if it is obstacle free.

From this, the workspace will be spherical ball will radius the distance from the reference point to the goal. Every sample point will be a subset of the spherical ball. The motion will be planned from a smaller subset spherical ball starting at the start point to the toward goal.

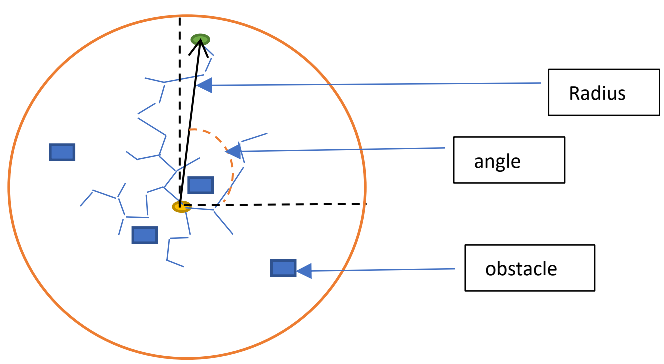

Figure: View of the 2d representation of the polar planning using RRT in R^1S^1 topology.

b.)

Planning Algorithms	Advantages	Disadvantages
Gradient descent planner with a potential function	- Can find the path to goal if no local minimum and one global minimum - Fast - No path smoothing needed	- Local minimum - Invalid points: the next point could be invalid due to constraints set. - Not easy to control controller input.
Wave-front planner	- Guaranteed to find path if it exists - Easy to implement - Can be implemented in workspace	- Little room to manipulate controller input because points are pre-sampled - Will need path smoothing
Probabilistic roadmap planner	- find path if it exists - Easy to implement	- Can become computationally intensive - Is probabilistic complete

	- Can be implemented in the chosen workspace	- Could lead to many invalid samples thus not path due to constraint if sample points too sparse - Will need path smoothing - Not easy to control controller input.
Randomized tree-based planner	- Ability to control or tweak controller inputs - Could be fast depending on the version chosen - Will find path if it exists (this is time dependent) - Might not need path smoothing small step is chosen	- Could take too long to complete depending on the planning flavor. - Can be computationally intensive. - Probabilistic complete

c.) Pseudocode RRT_Goal_Bias

```python
class mini_project:
    def __init__(self, args):
        self.start = args.start  # start point information
        self.destination = args.destination  # destination point information
        self.theta_range = args.theta_range  # range/contrain of the angle theta
        self.psi_range = args.psi_range  # range/contrain of the angle psi
        self.omega_range = args.omega_range  # range/contrain of the psi_dot
        self.v_range = v_range  # range/contrain of the velocity
        self.alpha_range = alpha_range  # range/contrain of the theta_dot
        self.acc_range = acc_range  # range of the acceleration.
        self.ball_radius = None

    def rrt_goal_bias(self):
        # get the ball_radius and the angle of the goal points
        self.ball_radius, theta_dest, psi_dest = compute_goal_point()
        # set nodes and edges variables
        nodes = []
        edges = []
        # hold the controller input
        ctrl_input = []
        # add the start point to the node list
        nodes.add(self.start)
        # set counter to  zero
        n = 0
        # compute thw path using rrt goal bias
        while not in goal_ragion:
            # sample a point in the workspace using a polar coodinate
            r, theta, psi = sample_point(n)
            # get a controller input from the sampled point
            theta_dot, psi_dot, a, closes_node = compute_controller_input(
                t_span, (r, theta, psi), nodes)
            # use the controller inputs and state space of the system to find x,y,z
            x, y, z = compute_cordinate((theta_dot, psi_dot, a))
            # make an edge with the new x,y z cordinate
            edge = [(nodes.point), (x, y, z)]
            # check for edge validity
            if is_obstacle_free(edge):
                # add the edge to the list of valide edge as well as the node
                new_node = [(r, theta, psi), (x, y, z)]
                nodes.add(new_node)
                edges.add(edge)
                ctrl_input.add((theta_dot, psi_dot, a))
            # increment the counter
            n += 1
        # find our path to goal based of the valide edges and nodes
        path = find_path(nodes, edges, ctrl_input)
        # return the path
        return path
```

```
    def sample_point(self, n):
        #n is used to bias our sampling toward goal
        #sample a point in the workspace
        if n%90:
            r=random(0, ball_radius)
# pick a randome pon along the max radius of the sphere that contains out goal
            theta=random(self.theta_range) # pick a ramdom in our theta range
            psi=random(self.psi_range) # pick a random point in our psi range
            return r, theta, psi  # return the values
        return sample_near_goal

    def compute_goal_point(self):
        # using the destination data we can calculate the radius of the ball in which
        # the destinantion point is located
        #the angles of the destination point from our reference point.
        ball_radius=distance(self.start, self.destination)
        theta_dest, psi_dest=calc_angles(self.destination)

        return ball_radius,theta_dest, psi_dest

    def compute_controller_input(self,t_span,data):
        # calculate the controller input based on the data and time_span
        # we can use the time span to control the velocities and accelation
        # return the differentiation of both the theta_dot and psi_dot calculated as
        # (x_1-x_0)/t_span
        #also return the acceleration and the point chosen for the calculation
        return theta_dot, psi_dot, a, closed_node

    def compute_cordinate(self,data):
        #calculate the x,y,z using the controller.
        #we know that v=rw or r_x*w for example

        q[i+1]=A*q[i] + Bu[i]
        y=C*q[i]
        return y


    def is_obstacle_free(self, edge):
        #check if our new edge is obstacle free
        #because the drone is not a point we have to consider the edge as a 3d box or tunnel
        if obstacle_free:
            return true
        return false


if __name__ == "__main__":
    mini=mini_project(args)
    path= mini.rrt_goal_bias()
```

d.) Using the chosen algorithm, RRT goal bias, the planner should find the path to goal if it existed given enough time. This is guaranteed because we will sample some data points near goal at x% of time. Thus, if the algorithm is given enough time I should converge to goal.

e.) With RRT Goal Bias, we sample a point in the space of the agent then we then choose a new P_new sample that is in a radius r that we define along the closes point in our graph's valid points. With this approach we can have control over the chosen P_new. Thus, for the point p_new, the team can optimize the controller input that allow to get the best point a.k.a best edge. The team can then construct the graph with points that are optimal. We can define each point with more than one optimal parameter.

For example, if each edge has the following structure

```
Struct edge {

        tuple p1;

        tuple p2;

        float time;

        float distance;

        float other_parm;

        }
```

The team can then find best path using for example time or distance of each edge.

f.) Dynamic of the drone in the environment:

- Before passing through rope plane:

Before passing through the rope plane, the dynamic of the drone should not be different. However, there would be a need to design a controller that control each element of the drone independently. That is, the pitch, roll and yaw would be controlled by independent controllers. This will allow adjusting input to the plant (drone) without affecting directly other dynamics of the drone.

The drone will also need to keep a high altitude with respect to the goal point altitude. In fact, if more than one motor is lost, the difficulty of flying the quadcopter should increase. The idea here is that if flying the quadcopter becomes (nearly) impossible, the system can calculate how much force is needed to truss itself as a projectile to the goal. Well, for this to work, one needs to flight above all known obstacle.

- After passing through rope plane:

As hinted above, there are few options to consider after passing through the ropes plane. Before we get there, it is necessary to point out that the yaw control of the quadcopter has to be deactivated after impact. Thus, the quadcopter will be allowed to spin toward goal horizontally but freely. This could work fine with three propellers with angle ψ out of the dynamics of the system.

With more than one propeller damaged, it would be almost impossible (to not say simply impossible) to fight the quadcopter, however, while flying above all know obstacles, projectile motion can be used to achieve the goal point requirements.

When the quadcopter hit the rope, system need to calculate the pitch, roll position acceleration needed to simulate a free fall projectile motion using the remaining propeller to get to goal. Normal flying cannot be achieved at this stage.

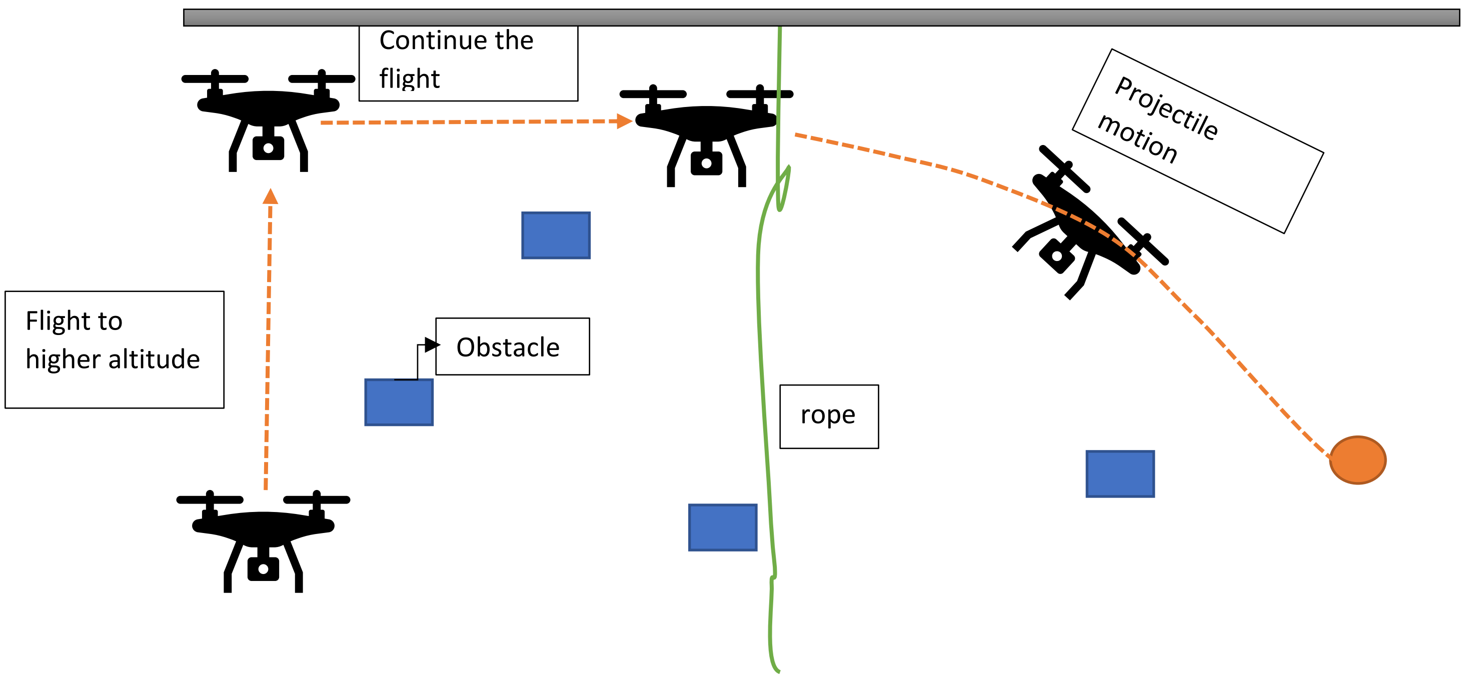

Figure: Representation of the flight scenario in 2d

g.) Pseudocode and description

Description:

From algorithm in the first part, the following improvement can be made according to the dynamics of the quadcopter.

- First the algorithm needs to find a path to the goal with the higher altitude that give a clear view with no obstacle on the way.
- The quadcopter flight to that maximum altitude first from the start point
- During flight, at each step or controller input, the system needs to calculate the force needed to flight as a projectile (almost free fall) to the goal using the requirements.
- Always check for damage at each step
- In case of damage, record the reference plane and get a new path to the goal by considering the plan as an obstacle.
- When a motor is loss, flight as projectile (free fall but truss toward goal with remaining propellers) with horizontal spinning toward goal
- If 3 motors loss, there is no possibility to reach goal, free (vertical) fall.

Coovi Meha

Pseudocode:

```
class mini_project:
    def __init__(self, args):
        self.start = args.start  # start point information
        self.destination = args.destination  # destination point information
        self.theta_range = args.theta_range  # range/contrain of the angle theta
        self.psi_range = args.psi_range  # range/contrain of the angle psi
        self.omega_range = args.omega_range  # range/contrain of the psi_dot
        self.v_range = v_range  # range/contrain of the velocity
        self.alpha_range = alpha_range  # range/contrain of the theta_dot
        self.acc_range = acc_range  # range of the acceleration.
        self.ball_radius = None  # the radius of the ball of contaning the goal
        self.avoid_plane = args.rope  # the vertical plane to avoid

    def path_check(self, path):
        # check the path for rope interaction
        # get the path the normal way without considering the rope
        path = rrt_goal_bias()
        new_path = []

        # check if the the rope is on the was of our path.
        while path:
            p = path.pop(0)
            if is_rope(p):
                # create a new path based on the description
                # given about the projectile motion, by turing
                # off the yaw control and act as projectile toward goal
                new_path = change_path(p)
                #return remaining of the path + the new path using this position as the starting point
                return path+new_path

    def rrt_goal_bias(self):
        # return the path to goal with the controller input associated
        # get the ball_radius and the angle of the goal points
        self.ball_radius, theta_dest, psi_dest = compute_goal_point()
        # set nodes and edges variables
        nodes = []
        edges = []
        # hold the controller input
        ctrl_input = []
        # add the start point to the node list
        nodes.add(self.start)
        # set counter to  zero
        n = 0

##########get#the#altitude#to#start#the#motion#from################################################
        r, theta, psi = get_clear_altitude()
        # get controller input to get there
        theta_dot, psi_dot, a, node = compute_controller_input(
            t_span, (r, theta, psi), nodes)

        # create an edge
        edge = [(node.point), (x, y, z)]

        # make a new node
        new_node = [(r, theta, psi), (x, y, z)]
        # add new node to all nodes
        nodes.add(new_node)
        # add edge to edges
        edges.add(edge)

        # add controle input
        ctrl_input.add((theta_dot, psi_dot, a))
        ##############################################################################################
        # compute thw path using rrt goal bias
        while not in goal_ragion:
            # sample a point in the workspace using a polar coodinate
            r, theta, psi = sample_point(n)
            # get a controller input from the sampled point
            theta_dot, psi_dot, a, node = compute_controller_input(
                t_span, (r, theta, psi), nodes)
            # use the controller inputs and state space of the system to find x,y,z
            x, y, z = compute_cordinate((theta_dot, psi_dot, a))
            # make an edge with the new x,y z cordinate
            edge = [node, (x, y, z)]
            # check for edge validity
            if is_obstacle_free(edge):
                # add the edge to the list of valide edge as well as the node
                new_node = [(r, theta, psi), (x, y, z)]
                nodes.add(new_node)
                edges.add(edge)
                ctrl_input.add((theta_dot, psi_dot, a))
            # increment the counter
            n += 1
        # find our path to goal based of the valide edges and nodes
        path = find_path(nodes, edges, ctrl_input)
        # return the path
        return check_path(path)
```

```python
    def get_clear_altitude(self):
        # sample the work space  in the start area in the z direction to get a
        # clear path for a projectile like motion with no static obstacles
        return r, theta, psi

    def sample_point(self, n):
        # n is used to bias our sampling toward goal
        # sample a point in the workspace
        if n % 90:
            # pick a randome pon along the max radius of the sphere that contains out goal
            r = random(0, ball_radius)
            # pick a ramdom in our theta range
            theta = random(self.theta_range)

            # pick a random point in our psi range
            psi = random(self.psi_range)
            return r, theta, psi  # return the values
        return sample_near_goal

    def compute_goal_point(self):
        # using the destination data we can calculate the radius of the ball in which
        # the destinantion point is located
        # the angles of the destination point from our reference point.
        ball_radius = distance(self.start, self.destination)
        theta_dest, psi_dest = calc_angles(self.destination)

        return ball_radius, theta_dest, psi_dest

    def compute_controller_input(self, t_span, data):
        # calculate the controller input based on the data and time_span
        # we can use the time span to control the velocities and acceletion
        # return the differentiation of both the theta_dot and psi_dot calculated as
        # (x_1-x_0)/t_span
        # also return the acceleration and the point chosen for the calculation
        return theta_dot, psi_dot, a, closed_node

    def compute_cordinate(self, data):
        # calculate the x,y,z using the controller.
        # we know that v=rw or r_x*w for example

        q[i+1] = A*q[i] + Bu[i]
        y = C*q[i]
        return y

    def is_obstacle_free(self, edge):
        # check if our new edge is obstacle free

# because# theodcheekisfnedga goesttheobghevertotcasipleanthe edge as a 3d box or tunnel
        if obstacle_free:
            return true
        return false


if __name__ == "__main__":
    mini = mini_project(args)
    path = mini.rrt_goal_bias()
```

h.) The planner in part g can find a path to goal if one exists but subject to how many motors are lost in the process. If 3 motors are lost, I will be impossible to flight. That is, as one cannot predict the location on the ropes, it is not possible to predict   how many motors we could be lost getting to the goal. So, the algorithm completion is related to the flying ability of the quadcopter. Thus, no I cannot guarantee that we can get to goal is a path exit