Lab 5: RRT Hardware Implementation

In this lab, you will work in teams to use your Rapidly-Exploring Random Tree (RRT) implementation from Lab 4 in practice. We have set up obstacles in the drone cage in G105 and ACEE012 (note that the obstacles are in different positions for each of the 3 netted areas). You will measure the positions of the obstacles, field, and starting/end locations, and use the RRT code to find a trajectory through the space. There are measuring tapes in the labs that you can use to make the measurements. We provide you the code to run the drone through a sequence of setpoints (in x,y) provided to it. You will be graded on your ability to autonomously navigate the drone from the starting position to the end goal location. Make sure that your video includes your drone landing within the end position.

As in the previous lab, you will work as a team. **Only one** team member should submit your team's results to the gradescope.

Part 1: Recycling the RRT code from Lab 4. (10 Pts)

This lab is a little more open-ended than previous labs, with the objective of encouraging you to put together concepts you learned in class.

Please use the cells below to fill with your RRT code. Since you are working as a group, take the opportunity to compare your different solutions to the previous homework. Some implementations were cleaner or more efficient than others, so use the version you like the best!

You will use the RRT code to find the trajectory from the starting position to the end goal in the drone cage. We recommend keeping the format of the code consistent with your homework for debugging ease.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from typing import List, Tuple

%matplotlib notebook

## TODO: Fill in the RRT code here, with corresponding helper functions.
def conf_free(q: np.ndarray, obstacles: List[Tuple[np.ndarray, float]]) -> bool
    """
    Check if a configuration is in the free space.

    This function checks if the configuration q lies outside of all the obstacl

    @param q: An np.ndarray of shape (2,) representing a robot configuration.
    @param obstacles: A list of obstacles. Each obstacle is a tuple of the forr
    @return: True if the configuration is in the free space, i.e. it lies outsi
             Otherwise return False.
    """
```

```
    for pos, r in obstacles:
        if np.linalg.norm(q - pos) < r:
            return False

    return True

def edge_free(edge: Tuple[np.ndarray, np.ndarray], obstacles: List[Tuple[np.nda
    """
    Check if a graph edge is in the free space.

    This function checks if a graph edge, i.e. a line segment specified as two
    every obstacle in the configuration space.

    @param edge: A tuple containing the two segment endpoints.
    @param obstacles: A list of obstacles as described in `config_free`.
    @return: True if the edge is in the free space, i.e. it lies entirely outs
             Otherwise return False.
    """

    for pos, r in obstacles:
        # find closest point on edge (extended to infinity)
        proj = np.dot(pos - edge[0], edge[1] - edge[0]) / np.linalg.norm(edge[
        closest = edge[0] + proj * (edge[1] - edge[0])
        # check whether closest point is in obstacle
        # and check whether edge (not extended to infinity) actually intersect
        if (np.linalg.norm(closest - pos) < r and
            not (np.linalg.norm(edge[1] - pos) > r and
            np.sign(np.cross(edge[0] - pos, closest - pos)) == np.sign(np.cross
            return False

    return True
```

In [2]:
```
def random_conf(width: float, height: float) -> np.ndarray:
    """
    Sample a random configuration from the configuration space.

    This function draws a uniformly random configuration from the configuratio
    does not necessarily have to reside in the free space.

    @param width: The configuration space width.
    @param height: The configuration space height.
    @return: A random configuration uniformily distributed across the configura
    """
    conf = np.random.rand(2)
    conf[0] *= width
    conf[1] *= height
    return conf
```

In [3]:
```
def nearest_vertex(conf: np.ndarray, vertices: np.ndarray) -> int:
    """
    Finds the nearest vertex to conf in the set of vertices.

    This function searches through the set of vertices and finds the one that
    conf using the L2 norm (Euclidean distance).

    @param conf: The configuration we are trying to find the closest vertex to
    @param vertices: The set of vertices represented as an np.array with shape
                     a vertex.
```

```
    @return: The index (i.e. row of `vertices`) of the vertex that is closest t
    """
    return np.argmin(np.linalg.norm(vertices - conf, axis=1))

def extend(origin: np.ndarray, target: np.ndarray, step_size: float=0.2) -> np
    """
    Extends the RRT at most a fixed distance toward the target configuration.

    Given a configuration in the RRT graph `origin`, this function returns a ne
    step of at most `step_size` towards the `target` configuration. That is, if
    and `target` is less than `step_size`, return `target`. Otherwise, return t
    segment between `origin` and `target` that is `step_size` distance away fro

    @param origin: A vertex in the RRT graph to be extended.
    @param target: The vertex that is being extended towards.
    @param step_size: The maximum allowed distance the returned vertex can be t

    @return: A new configuration that is as close to `target` as possible witho
            `step_size` away from `origin`.
    """
    if np.linalg.norm(target - origin) < step_size:
        return target
    else:
        return origin + step_size * (target - origin) / np.linalg.norm(target -
```

In [4]:

```
def rrt(origin: np.ndarray, width: float, height: float, obstacles: List[Tuple
        trials: int=1000, step_size: float=0.2) -> (np.ndarray, np.ndarray):
    """
    Explore a workspace using the RRT algorithm.

    This function builds an RRT using `trials` samples from the free space.

    @param origin: The starting configuration of the robot.
    @param width: The width of the configuration space.
    @param height: The height of the configuration space.
    @param obstacles: A list of circular obstacles.
    @param trials: The number of configurations to sample from the free space.
    @param step_size: The step_size to pass to `extend`.

    @return: A tuple (`vertices`, `parents`), where `vertices` is an (n, 2) `np
             and `parents` is an array identifying the parent, i.e. `parents[i]
             the `i`th row of `vertices.
    """
    num_verts = 1

    vertices = np.zeros((trials + 1, len(origin)))
    vertices[0, :] = origin

    parents = np.zeros(trials + 1, dtype=int)
    parents[0] = -1

    for trial in range(trials):
        #TODO: Fill this loop out for your assignment.
        conf = random_conf(width, height)
        nearest = nearest_vertex(conf, vertices[:num_verts])
        step = extend(vertices[nearest], conf, step_size)

        if conf_free(step, obstacles) and edge_free((vertices[nearest], step),
            vertices[num_verts] = step
```

```
            parents[num_verts] = nearest
            num_verts += 1

    return vertices[:num_verts, :], parents[:num_verts]

def backtrack(index: int, parents: np.ndarray) -> List[int]:
    """
    Find the sequence of nodes from the origin of the graph to an index.

    This function returns a List of vertex indices going from the origin verte

    @param index: The vertex to find the path through the tree to.
    @param parents: The array of vertex parents as specified in the `rrt` funct

    @return: The list of vertex indicies such that specifies a path through th
    """

    path = []
    while index != -1:
        path.append(index)
        index = parents[index]

    path.reverse()
    return path
```

Part 2: Defining the Configuration Space. (40 Pts)

Now that you have your RRT code, you can start setting up the code to navigate the drone through the PVC pipe forest.

The drone cage has been set up such that a series of PVC pipes are suspended from the ceiling. Your goal is to start your drone in the starting position, marked with an "X" on the ground in masking tape. You will be graded on your ability to land the drone within the end goal location, marked with a box and the word "END" in masking tape. And, of course, you will need to autonomously navigate from start to end while avoiding the obstacles in the drone's path.

Now: define your configuration space. You will need to measure the obstacles and their positions, plus whatever other information about the drone cage you think is necessary (relative positions of start/end locations, the width and height of the field, etc.). For the boundary of the space, please make sure to use the rectangular region marked by the orange tape; your drone should not leave this rectangular region as it goes from start to end. Once these measurements are collected, you will simulate your trajectory through the cage below. Sample code from the previous homework has been provided to help you plot the trajectory. Think: How will you handle absolute positions vs. relative positions to the starting location?

It's also important to remember that for the Crazyflie, the positive X direction is forward (where the nub is) and the positive Y direction is to the left when you stand behind the drone.

There is one additional caveat, however: **Remember to "inflate" your obstacles**, so that there is a buffer zone around each one. In practice, this will be necessary to help give the drone

adequate space from each PVC pipe. We leave it up to you to assess how much to inflate the obstacles.

Once you have taken these measurements, fill out the code below:

In [206…
```
## Sample code to define the space/obstacles and run the RRT:

## TODO: Take measurements in G105/ACEE and edit this block of code correspond
r_drone = 7
r_obstacle = 4
r_goal = 10.25
r_err = 4
r_buffer = r_drone + r_obstacle + r_err

# Width and height of the rectangular domain:
width = 360.5 + 30  # total width of the cage
height = 153.5 # total height of the cage

# Obstacles are represented as tuples, where the first element is an np.ndarra
# and the second element is the radius of the obstacle. For example (np.array(
# This variable is a list of such tuples.
obstacles = [(np.array([89.5, 36]), r_buffer),
             (np.array([126, 121]), r_buffer),
             (np.array([196, 81]), r_buffer),
             (np.array([273, 34]), r_buffer),
             (np.array([265.5, 131.5]), r_buffer)]

# The goal is represented in the same way as an obstacle.
goal = (np.array([339.5 + 35, 78]), r_goal - r_drone)

# The starting position of the robot.
origin = np.array([19, 64.5])

# Run the RRT to find the trajectory in this space:
trials = 4000
step_size = 10
vertices, parents = rrt(origin, width, height, obstacles, trials=trials, step_s

# Check if path was found:
index = nearest_vertex(goal[0], vertices)
if np.linalg.norm(vertices[index, :] - goal[0]) < goal[1]:
    print('Path found!')
    path_verts = backtrack(index, parents)
else:
    print('No path found!')
    path_verts = []
```

```
Path found!
```

In [213…
```
## Sample code to plot the trajectory:

fig, ax = plt.subplots()

ax.set_xlim([0, width])
ax.set_ylim([0, height])
ax.set_aspect('equal')

for i in range(len(parents)):
    if parents[i] < 0:
```

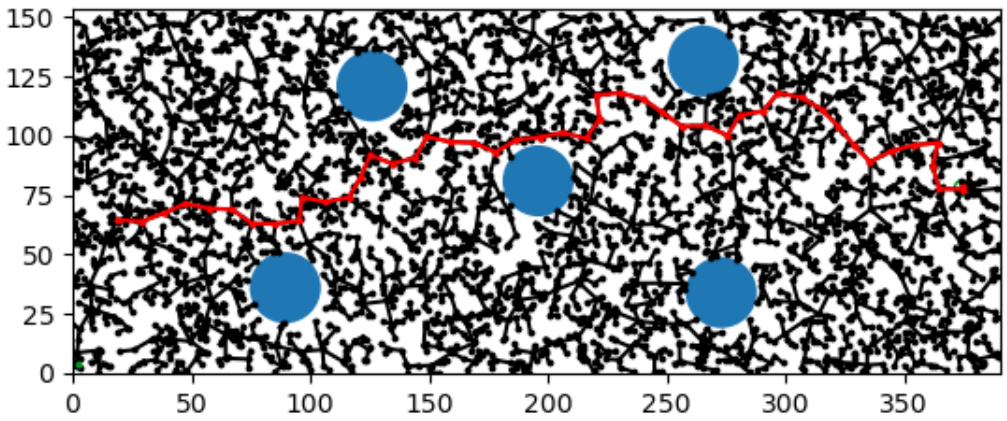
```
        continue
    plt.plot([vertices[i, 0], vertices[parents[i], 0]],
             [vertices[i, 1], vertices[parents[i], 1]], c='k')

for i in path_verts:
    if parents[i] < 0:
        continue
    plt.plot([vertices[i, 0], vertices[parents[i], 0]],
             [vertices[i, 1], vertices[parents[i], 1]], c='r')

for o in obstacles:
    ax.add_artist(plt.Circle(tuple(o[0]), o[1]))

ax.add_artist(plt.Circle(tuple(goal[0]), goal[1], ec=(0.004, 0.596, 0.105), fc=

plt.scatter([2.5], [3.5], zorder=3, c=np.array([[0.004, 0.596, 0.105]]), s=3)
plt.scatter(vertices[path_verts, 0], vertices[path_verts, 1], c=np.array([[1, (
plt.scatter(vertices[1:, 0], vertices[1:, 1], c=np.array([[0, 0, 0]]), s=3)
```



Out[213]: `<matplotlib.collections.PathCollection at 0x7fad5b99f7d0>`

Part 3: Hardware Implementation. (50 Pts)

Now that you have a trajectory, you can begin testing with the drone.

First, define your group number (as in Lab 2):

In [208…

```
group_number = 4
```

Here, we are providing code which takes your RRT trajectory (assuming the format of the output is consistent with that in Lab 4) and turns it into a sequence of setpoints which the drone can follow.

In [209…
```
def seg_to_setpoints(start_conf: np.ndarray, end_conf: np.ndarray) -> np.ndarra
    # This function takes in the RRT trajectory and outputs a sequence of setp

    dist = np.linalg.norm(start_conf - end_conf)
    num_samples = int(100 * dist)

    return end_conf.reshape((1, 2))

traj = origin.reshape(1, 2)
for i in range(len(path_verts) - 1):
    traj = np.concatenate((traj, seg_to_setpoints(vertices[path_verts[i]], vert
```

Here, we provide the code which actually tells the drone how to follow a sequence of setpoints:

In [210…
```
# Code adapted from: https://github.com/bitcraze/crazyflie-lib-python/blob/mas

import time
# CrazyFlie imports:
import cflib.crtp
from cflib.crazyflie import Crazyflie
from cflib.crazyflie.log import LogConfig
from cflib.crazyflie.syncCrazyflie import SyncCrazyflie
from cflib.crazyflie.syncLogger import SyncLogger

## Some helper functions:
## ------------------------------------------------------------------------

# Determine initial position:
def wait_for_position_estimator(scf):
    print('Waiting for estimator to find position...')

    log_config = LogConfig(name='Kalman Variance', period_in_ms=500)
    log_config.add_variable('kalman.varPX', 'float')
    log_config.add_variable('kalman.varPY', 'float')
    log_config.add_variable('kalman.varPZ', 'float')

    var_y_history = [1000] * 10
    var_x_history = [1000] * 10
    var_z_history = [1000] * 10

    threshold = 0.001
    with SyncLogger(scf, log_config) as logger:
        for log_entry in logger:
            data = log_entry[1]

            var_x_history.append(data['kalman.varPX'])
            var_x_history.pop(0)
            var_y_history.append(data['kalman.varPY'])
            var_y_history.pop(0)
            var_z_history.append(data['kalman.varPZ'])
            var_z_history.pop(0)

            min_x = min(var_x_history)
```

```
            max_x = max(var_x_history)
            min_y = min(var_y_history)
            max_y = max(var_y_history)
            min_z = min(var_z_history)
            max_z = max(var_z_history)

            print("{} {} {}".
                format(max_x - min_x, max_y - min_y, max_z - min_z))

            if (max_x - min_x) < threshold and (
                    max_y - min_y) < threshold and (
                    max_z - min_z) < threshold:
                break

# Initialize controller:
def set_PID_controller(cf):
    # Set the PID Controller:
    print('Initializing PID Controller')
    cf.param.set_value('stabilizer.controller', '1')
    cf.param.set_value('kalman.resetEstimation', '1')
    time.sleep(0.1)
    cf.param.set_value('kalman.resetEstimation', '0')

    wait_for_position_estimator(cf)
    time.sleep(0.1)
    return

# Ascend and hover:
def ascend_and_hover(cf):
    # Ascend:
    for y in range(20):
        cf.commander.send_hover_setpoint(0, 0, 0, y / 50)
        time.sleep(0.1)
    # Hover at 0.5 meters:
    for _ in range(30):
        cf.commander.send_hover_setpoint(0, 0, 0, 0.5)
        time.sleep(0.1)
    return

# Follow the setpoint sequence trajectory:
def run_sequence(scf, sequence, setpoint_delay):
    cf = scf.cf
    for position in sequence:
        print(f'Setting position {(position[0], (position[1]))}')
        for i in range(setpoint_delay):
            cf.commander.send_position_setpoint(position[0],
                                                (position[1]),
                                                0.5,
                                                0.0)
            time.sleep(0.1)

# Hover, descend, and stop all motion:
def hover_and_descend(cf):
    # Hover at 0.5 meters:
    for _ in range(30):
        cf.commander.send_hover_setpoint(0, 0, 0, 0.5)
        time.sleep(0.1)
    # Descend:
    for y in range(10):
        cf.commander.send_hover_setpoint(0, 0, 0, (10 - y) / 25)
```

```
        time.sleep(0.1)
    # Stop all motion:
    for i in range(10):
        cf.commander.send_stop_setpoint()
        time.sleep(0.1)
    return
## ------------------------------------------------------------------------

def run_setpoint_trajectory(group_number, sequence):
    # This is the main function to enable the drone to follow the trajectory.

    # User inputs:
    #
    # - group_number: (int) the number corresponding to the drone radio settin
    #
    # - sequence: a series of point locations (float) defined as a numpy array
    #     [x(meters), y(meters)]
    #   Note: the input should be given in drone coordinates (where positive x
    # Example:
    # sequence = [
    #     [[ 0.          0.        ]
    #      [0.18134891  0.08433607]]
    #

    # Outputs:
    # None.

    setpoint_delay = 3 # (int) Time to give the controller to reach next setpo

    # Set the URI the Crazyflie will connect to
    uri = f'radio://0/{group_number}/2M'

    # Initialize all the CrazyFlie drivers:
    cflib.crtp.init_drivers(enable_debug_driver=False)

    # Sync to the CrazyFlie:
    with SyncCrazyflie(uri, cf=Crazyflie(rw_cache='./cache')) as scf:
        # Get the Crazyflie class instance:
        cf = scf.cf

        # Initialize and ascend:
        set_PID_controller(cf)
        ascend_and_hover(cf)
        # Run the waypoint sequence:
        run_sequence(scf, sequence, setpoint_delay)
        # Descend and stop all motion:
        hover_and_descend(cf)

    print('Done!')
    return
```

Putting it all together.

The objective: Get from the starting position to the end goal location without collisions.

Use your RRT trajectory, the code provided to turn the trajectory into a sequence of setpoints, and the code provided to run the sequence of setpoints on the drone.

The caveat: You may need to modify some of your trajectory coordinates in order to get this to work. Think about how you defined your space initially, and then put this in terms of coordinates that the drone can understand. Again, from the perspective of the drone, positive x is **forward**, and positive y is to the **left**. The drone defines its origin at the starting point.

The success of your run may depend strongly on how cleanly the drone ascends from its starting location. If your drone always ascends wildly, it will be difficult for the drone to hit the setpoints you defined. Make sure your hardware is in good conditions, as a single failing motor or damaged propeller may impact ascension consistently.

In [211…
```
# TODO: Implement any code necessary to convert your setpoint sequence to rela
traj = np.array(traj) - origin
traj /= 100
```

In [212…
```
# Run the setpoint sequence on the drone:
run_setpoint_trajectory(group_number, traj)
```

```
Initializing PID Controller
Waiting for estimator to find position...
999.9999891590624 999.9999891933521 999.9997603953234
999.9999904146061 999.9999904120732 999.9997698396619
999.9999904146061 999.9999904120732 999.9997751540359
999.9999904146061 999.9999904120732 999.9997751540359
999.9999904323622 999.9999904194119 999.9997751540359
999.9999904323622 999.9999904194119 999.9997751540359
999.9999904323622 999.9999904194119 999.9997751540359
999.9999904323622 999.999990427069 999.9997751540359
999.9999904323622 999.999990427069 999.9997751540359
1.2732998584397137e-06 1.2337168300291523e-06 1.4758712495677173e-05
Setting position (0.0, 0.0)
Setting position (0.09983052482072725, -0.005819477125838546)
Setting position (0.19371380249597478, 0.028617859776894933)
Setting position (0.2847844521753068, 0.06992326822402674)
Setting position (0.38268820296805545, 0.049555264377027584)
Setting position (0.4825401028034392, 0.044114850857404235)
Setting position (0.5636997669563212, -0.014306967965629625)
Setting position (0.6636891412520198, -0.01576471552768922)
Setting position (0.7628588585325651, -0.002905198643422864)
Setting position (0.776328622924059, 0.0961834760234514)
Setting position (0.8743172949522782, 0.07622802177790433)
Setting position (0.972437051226984, 0.09552862872671071)
Setting position (1.0225372452679573, 0.18207324471786762)
Setting position (1.0633493440466504, 0.27336602971908336)
Setting position (1.156607870010356, 0.23727105312847158)
Setting position (1.2531316064252065, 0.2634085416940637)
Setting position (1.3016218588072983, 0.3508653620337094)
Setting position (1.3993348906070284, 0.32960122752263854)
Setting position (1.4992648563991053, 0.32585931132978874)
Setting position (1.5911429567259279, 0.28638255006022234)
Setting position (1.6770498741412379, 0.3375685006736387)
Setting position (1.7763700459158356, 0.34920909682093537)
Setting position (1.8745545371376644, 0.3681776414016504)
Setting position (1.9715379959730035, 0.343801244778398)
Setting position (2.0272269955641207, 0.4268598706070293)
Setting position (2.015132806103608, 0.5261258294312567)
Setting position (2.114595874001881, 0.5364746517924437)
Setting position (2.210956198814613, 0.5097410062874846)
Setting position (2.293092859234222, 0.4527009412882768)
Setting position (2.374949146650481, 0.39525924728098916)
Setting position (2.474916764181462, 0.39780393688476096)
Setting position (2.566150894029294, 0.3568608843494536)
Setting position (2.61855126750498, 0.4420324809037545)
Setting position (2.717169784431661, 0.45859714568098114)
Setting position (2.7796549878232444, 0.5366714651279826)
Setting position (2.877463236219742, 0.5158496845698974)
Setting position (2.963692038015645, 0.4652078664662491)
Setting position (3.0339487608273403, 0.39404612885564405)
Setting position (3.0973814263065855, 0.3167396158687156)
Setting position (3.1672253095542335, 0.2451726404338902)
Setting position (3.255966447686681, 0.29127047555613683)
Setting position (3.3527518058805934, 0.3164219057197688)
Setting position (3.452392748678614, 0.324888458638257)
Setting position (3.429386173315268, 0.227570949987189)
Setting position (3.4584865100065456, 0.13189874710435434)
Setting position (3.5584522903691806, 0.1292828808062677)
Setting position (3.5580799554816593, 0.13761107763994246)
Done!
```

In [ ]:

Submission Instructions

One member from each group should submit this notebook along with a video of a successful flight to "HW5: Coding" on gradescope.

In [ ]: