1. Requirements

The requirements of this lab were to develop a driver component capable of passing through an order list of GPS waypoints. The goal was simply to “touch” each waypoint in sequence. In this project, we need to refine your driver to keep with in the course boundaries.

2. Introduction

By making use of the onboard sensors in the ERTS vehicle, it can react to its environments and can be used for a variety of purpose ranging from path following. The labs involved writing CartFS components that monitor the sensors and reactively change their state depending on these input values at every clock step in the system.

A given list of waypoints defined a two-dimentional course, consisting a sequence number segments. Figure 1 is a screen shot of the ERTS visualizer tracing the path of a path-to-path follower.

![Figure 1](image_url)

This report describes the implementation of the basic components and functions to implement path planning and make sure the cart is running in the boundaries/corridor.
3. Implementation

Several places have been modified in the square.py code to meet the required functional objectives as demonstrated through the results. This section gives a brief overview of each of the functions.

3.1 Heading Controller

I used geopy to locate the cart’s current longitude and latitude. So the heading of the cart can be adjusted real timely. We use geopy to count the heading to target and distance between current longitude and latitude and next waypoint. Once the cart detect itself closing to a destination waypoint, it will start to detect the next waypoint.

Further, because of the cart's need some time to precess the command, I made the cart to start to detect the next waypoint 2.5 meter earlier.

Here are the code I did for implementing this part:

```python
    current_latlon = (self.gps_s['lat'], self.gps_s['lon'])
    newpoint_latlon = (new_wpts[self.a][1], new_wpts[self.a][2])

    self.meters_to_target = distance.distance(current_latlon, newpoint_latlon).kilometers * 1000

    heading_to_target = distance.distance(current_latlon, newpoint_latlon).forward_azimuth

    self.driver_s['direction'] = heading_to_target % 360

    if self.meters_to_target < new_wpts[self.a][4]+2.5:
        print 'next waypoint'
        self.a = (self.a + 1)% len(new_wpts)
```

3.2 Waypoint Controller

I wrote a loop condition to read in the original waypoints and make the program to generate a new waypoint list. Besides the original waypoints, I added some more waypoint in the middle of two connected original waypoints. So the new waypoint list have 18 waypoints to detect other than 9.

Here are the code I did for implementing this part:

```python
    k = 0
    new_wpts = []
    i = 0
    for i, point in enumerate(waypoints):
        new_wpts.append([k, point[1], point[2], 1.5, 3.0])
        k = (k+1)
        point[3] = (waypoints[(i+1)%len(waypoints)][1] + point[1])/2
```
point[4] = (waypoints[(i+1)%len(waypoints)][2] + point[2])/2
new_wpts.append([k,point[3],point[4],1.5,3.0])
k = (k+1)

3.3 Throttle Controller

For the throttle controller, I made some difference for the turning corner and the straight way. To make sure the Cart works well in the turning corner, I gave it a lower speed, but on the straight way, I set the throttle to 75% in the test field.

Here are the code I did for implementing this part:

```python
if self.meters_to_target <= self.driver_s['corner_radius']*3:
    self.jdriver_s['percent_throttle'] = self.jdriver_s['percent_throttle']*0.9
else:
    self.jdriver_s['percent_throttle'] = 75.0
```

3.4 Waypoints Adjustment

Besides, I also made some small adjustment for the waypoints the system read in.

4. Results

Running at the parking lot