AUTONOMIZATION OF A MOBILE HEXAPEDAL ROBOT USING A GPS

Phillip Dupree (Mechanical Engineering)
- Columbia University
Advisors: Daniel E. Koditschek, Galen Clark Haynes
Summer 2009

An important step in the autonomization of robots specifically designed for mobility is autonomous navigation: the ability to navigate from a current position to a programmed point, without the manual control of a human user. The object of this summer research was the autonomization of the robot RHex, a highly mobile hexapodal robot built in the GRASP lab of the University of Pennsylvania, which was accomplished by the integration of a global positioning system (GPS) module into the robot. The GPS module gave the robot the ability to follow a "breadcrumb" path of GPS way-points. Once the GPS data was parsed, the coordinates of both the robot's location and the path of waypoints were converted into flat-earth approximate Cartesian coordinates, and then inputted into a linear control system. Once this was accomplished, the robot had the ability to "know" its current position and navigate from it to any programmed point, providing there were no obstacles in its path.

Introduction

Robots are becoming more and more visible in daily life, as technological advancements make them capable of performing a variety of tasks. In particular, mobile robots are being used more frequently in everything from domestic purposes, such as mowing the lawn, to military tasks, such as bomb detection. As mobile robots become more advanced, human control is neither convenient nor adequate. More sophisticated robots must move towards autonomy: the ability to perform their tasks without human operation. Mobile robot navigation refers to the robot's ability to safely move towards the goal using its knowledge and the sensorial information of the surrounding environment. [2]. This can be achieved by using advanced sensors that give a robot the ability to "know" where it needs to go, and collect data on its surrounding environment in order to safely move about and complete its task.

This paper documents the process of making a mobile robot autonomous by integrating a global positioning system (GPS) module. The former is used for global navigation: a higher level of navigation in which the robot both knows its own position and can follow a path of GPS way points to navigate from its current position to some other position. The latter is used for local navigation: the ability of a robot to "see" its surrounding environment and use this knowledge to avoid obstacles. The ultimate goal of this autonomization is to have the mobile robot follow a "breadcrumb" path of GPS way points, while avoiding immediate obstacles in its path.

Background

Section 1: Navigation

1.1 Odometry

Robots use data from sensors on their motor called encoders to calculate how far they have moved in some certain period of time, just as blindfolded humans could guess their approximate position by counting how many steps they have taken. Odometry is the process of estimating the position of a mobile robot using the wheel's rotational velocity and angular velocity, determined by the encoders[4].

1.2 Dead reckoning

Dead reckoning is a simple mathematical procedure for determining the present location of a vessel by advancing some previous position through known course and velocity information over a given length of time [6]. Though dead reckoning has a broad meaning, it is typically implemented using odometry data. Dead reckoning is used to compute the position of a robot during path tracking and other behaviors. However, due to cumulative errors in its motor control, a robot that navigates by dead reckoning alone will eventually lose track of its trueposition [7]. Inertial measurement units, which work by sensing changes in acceleration and calculating change in position over time, are also used for dead reckoning. However, accumulated errors in estimation of displacement over time eventually lead to significant errors. Overall, cumulative errors make dead reckoning a useful yet imprecise measurement of navigation.

1.3 Global Navigation: Global Positioning System

The Navstar Global Positioning System (GPS) developed as a Joint Services Program by the Department of Defense uses a constellation of 24 satellites (including three spares) orbiting the earth every 12 hours at a height of about 10,900 nautical miles. According to Borenstein and Feng in their essay "Where Am I?", the absolute three-dimensional location of any GPS receiver is determined through simple trilateration techniques based on time of flight for uniquely coded spread-spectrum radio signals transmitted by the satellites. Knowing the exact distance from the ground receiver to three satellites theoretically allows for calculation of receiver latitude, longitude, and altitude. [5]. GPS gives users their positions whenever and wherever they are outdoors all over the world. In addition, using GPS, robots can always locate their own position with only one coordinate system that is standardized as latitude/longitude in GPS. This is very helpful to avoid complicated management of coordinate systems [8].

The integration of a GPS into a robot can give the robot the ability to know its current position, and follow a "breadcrumb path" of points from its current position to any determined way point, given an adequate control system. However, the global positioning system has its own set of constraints. GPS cannot be used when users are indoors because the radio waves from satellites cannot penetrate walls [8].

GPS modules give raw data as constant streams of sentences according to protocol (a standard set of rules) defined by the National Marine Electronics Association. The data included in the sentence is defined in the first word, while the individual pieces of data (such as latitude and longitude) are separated by commas. Different sentences give different pieces of data, and modules rotate through outputting these sentences as they constantly update the data they calculate using the satellite signals. In order to utilize the data, a code must be written to *parse* the data: break down each sentence and organize the data into a form useable by whatever device or program means to use it.

Section 2: The Robot

2.1 The Body

The RHex robot is a highly mobile hexapodal robot built in the GRASP lab at University of Pennsylvania. The robot's design consists of a rigid body with six compliant legs, each possessing one independently actuated revolute degree of freedom. The attachment points of the legs as well as the joint orientations are all fixed relative to the body [2].

The legs utilize an alternating tripod phase, with the two "tripods" revolving in anti-phase relative to one another. Each tripod is set to a Buehler clock function with a slow and fast phase. To achieve this, the clock uses a piece-wise linear angle vs. time reference trajectory characterized by four parameters: the total stride or cycle period, the duty factor (the ratio of a single stance period over the cycle period), the leg angle swept during stance, and an angle offset to break symmetry in the gait [11]. Different "gaits" can be designed by altering the parameters of the gait.

2.2. Control

A control system, a function used to mathematically compute appropriate commands to move a robot to a desired position, must be put in place to take the GPS data and move the robot along its path of way points. I decided to utilize a basic linear control system, one in which the system being studied can be modeled by linear differential equations [12]. This linear control system assumes the robot is moving at a constant speed, and alters the orientation over time to steer the robot to its desired position. Such a control system is commonly referred to as a unicycle control system.

Control System Design

1. The Linear Controller

The linear control system (LCS) decided upon is one in which only the orientation of the robot is controlled, and the velocity is assumed constant. Basic trigonometry shows that if this is the case, then if the robot (modeled as a point) moved from point A to point B with a constant velocity, the equations for movement are:

$$x = s \cdot \cos(\theta) + x_o$$

$$y = s \cdot \sin(\theta) + y_o$$
$$s = s$$
$$\theta = \theta$$

Where x, y are displacement in the x and y direction, respectively; theta ($\theta = \arctan(y/x)$) is the orientation of the point relative to the origin of a defined coordinate system, and x_o,y_o are the initial displacements of the point relative to the origin of the defined coordinate system (both zero is the point is initially at the origin). The derivatives of these equations with respect to time are then:

$$x' = s \cdot \cos(\theta)$$
$$y' = s \cdot \sin(\theta)$$
$$s' = 0$$
$$\theta' = \mu$$

The change in speed over time, ds/dt, is zero because speed is constant. The change in theta, μ, is determined later, as it is the only input into the system.

The control system took the form of:

$$X' = [A][X] + [B]$$

Where X is a 4 x 1 matrix consisting of X coordinate displacement, Y coordinate displacement, speed, and orientation; A is the state space equation that describes the state of the system, and B is the input into the system. Using the all ready derived equations we can fill in the state space equation and complete the linear controller:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{s} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & 0 & \cos(\theta) & 0 \\ 0 & 0 & \sin(\theta) & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ s \\ \theta \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ \mu \end{bmatrix}$$

1.2. Closing the Loop – Determining $d\theta/dt$, the Input into the System

The change in theta with respect to time is the keystone of the controller, as this is what steers the robot from its current position to its destination. In order to come up with an adequate equation, it is necessary not only to know the current orientation of the robot,

theta, but also the angle from the robot's current position to the destination point, theta star ($\theta^* = \arctan(y^*/x^*)$). By subtracting these two terms, the difference is the necessary change to alter the orientation of the robot from its current orientation to the direction of the destination point. Therefore, the equation for the change in theta became:

$$\theta' = k_p \cdot [\theta^* - \theta]$$

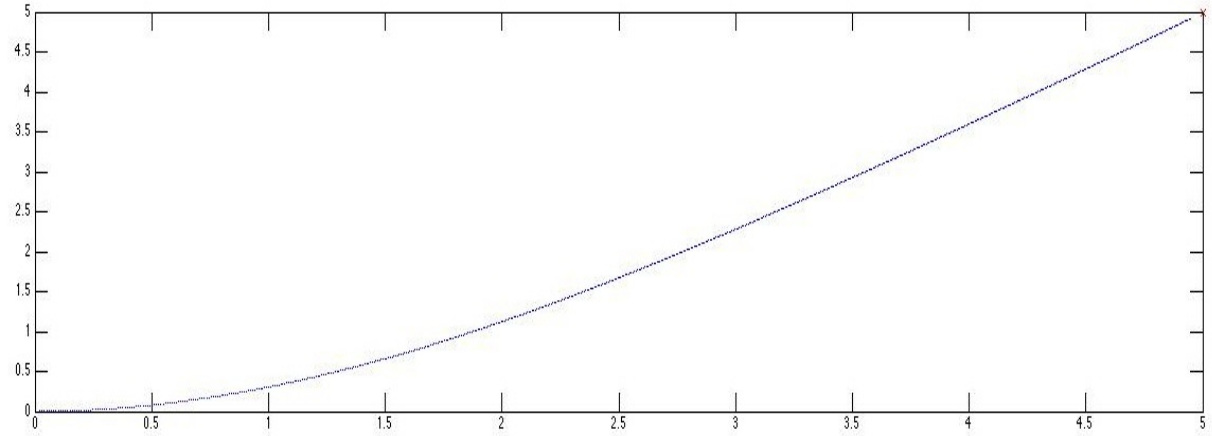
where k_p is the gain, a constant that determines how strongly the controller tells the robot to turn per iteration of time. If it is too high, the robot will try to change its orientation faster than it is able, causing errors. If it is too small, the robot will not be able to turn fast enough and will miss the point all together. θ' is recalculated each iteration of the controller, constantly updating the robot's orientation theta and recalculating how much the robot needs to turn to align itself with theta star.

With this equation determined, the controller s complete[1], as shown below.

$$\begin{aligned} x' &= s \cdot \cos(\theta) \\ y' &= s \cdot \sin(\theta) \\ s' &= 0 \\ \theta' &= kp \cdot [\theta^* - \theta] \end{aligned}$$

I then designed a Matlab simulation. The simulation utilized these equations to equate a giant matrix of the incremental changes of x and y of a point moving with some constant speed and changing its orientation according to the equation for θ'. After each iteration θ' was recalculated using the new values of x and y. The simulation looked like:

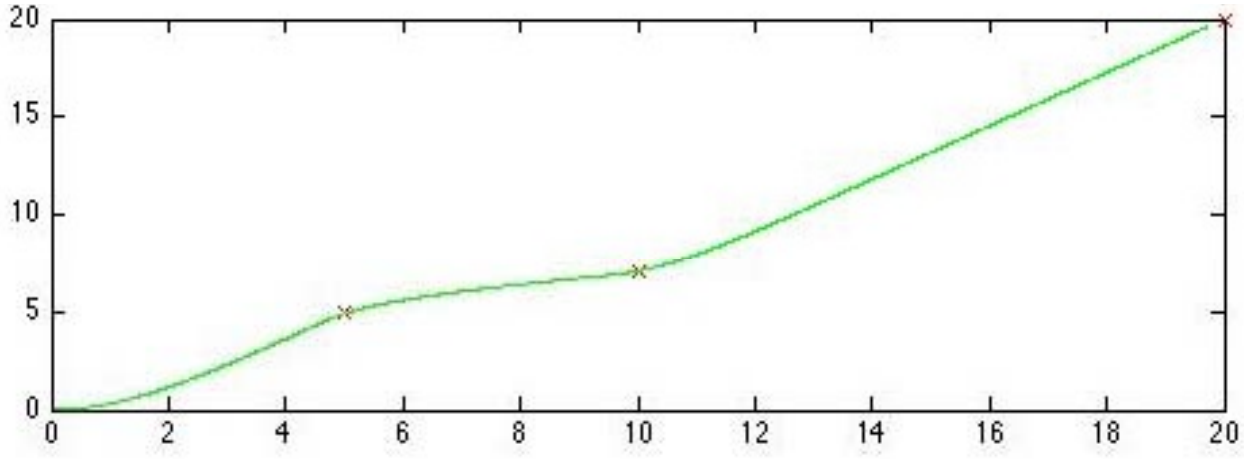
Figure 1: Robot utilizing Basic Controller to move from X = 0, Y = 0 to X = 5, Y = 5



[1] See Appendix A for Basic Linear Control System,Matlab simulation code.

The origin is defined as the start point, and coordinate (5,5) as the destination. The robot starts off moving and gradually alters its orientation, ending at the destination point. When modeled with multiple destination way points, the controller resembled this:

Figure 2: Robot utilizing Basic Controller to move to multiple Waypoints.



The controller reaches some certain distance from its destination point (depending on how accurate the controller needs to be; this value is set. In this case, the value is set at .1 units), and then moves on towards the next point. We call this term proximity.

However, as smooth as the graph may look, in reality such a controller has errors. The robot reaches its destination and then jerks towards the next point. A smoother, more advanced controller "rounds corners", beginning to orient itself towards the next point as it nears its first destination. I set about upgrading my control system to smoothly round corners. Considering that change in theta with respect to time, $d\theta/dt$, is the only variable controlled in this unicycle control system, the equation for θ' had to be altered. In order to have the robot start moving towards the $(n + 1)^{th}$ destination as it nearned the n^{th}, I needed to put in a θ^* term not just from the n^{th} term, but also the $(n + 1)^{th}$ term. This $(n + 1)^{th}$ θ^* became known as $\theta^{*2.}$ The equation for $d\theta/dt$ became:

$$\theta' = k_p \cdot [\theta^* - \theta] + \frac{k_{p2}}{\| X_1 - X \|} \cdot [\theta^{*2} - \theta]$$

Where $\| x - x_1 \|$ is a magnitude term, the displacement between the current position of the robot and the n^{th} way point. The second gain is divided by this term because this magnitude will get smaller as the robot nears the n^{th} way point. Therefore, the closer the robot moves towards the n^{th} way point, the stronger the influence of the second part of the θ' equation. If k_{p2} is too large and the influence of θ^{*2} is felt too strongly too early, the

robot will not reach the nth way point before it starts being dragged off course. The robot will loop around several times before eventually reaching proximity (Figure 3). In a worst case, the robot cannot reach proximity, cannot begin to move towards the (n+1)th point, and the controller will fail as the robot loops infinitely (Figure 4).

Figure 3: Robot has difficulty reaching Proximity due to a high Gain.

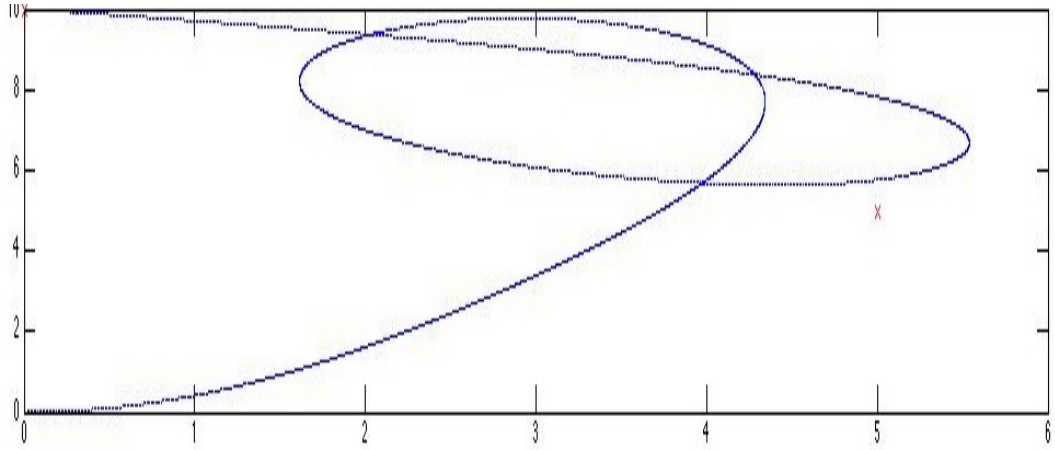
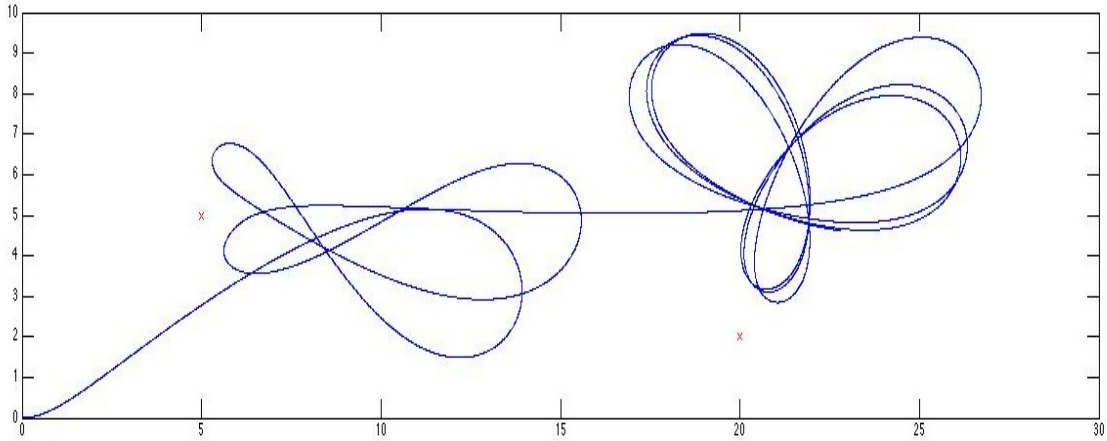


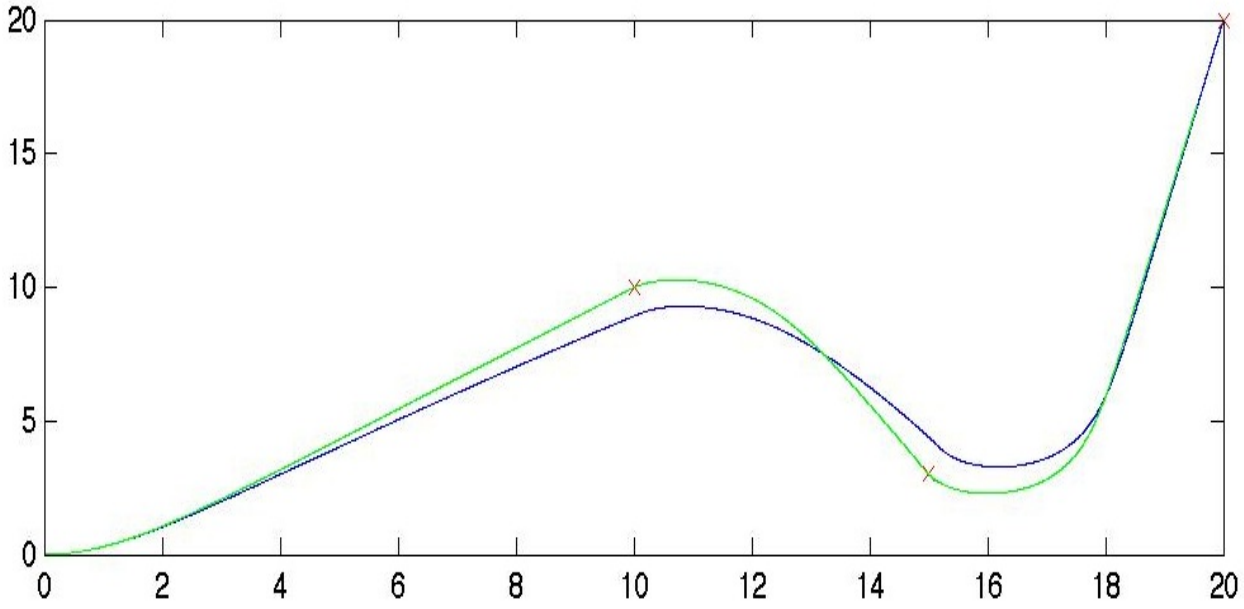
Figure 4: Robot unable to reach Proximity due to a high Gain.



However, working normally, the controller gently rounds corners, leading to a more efficient controller[2]. In Figure 5., the basic controller is in green while the advanced controller is in blue:

Figure 5: Basic Controller (Green) juxtaposed against the Advanced Controller (Blue).

[2] See Appendix B for Advanced Linear Control System, Matlab simulation code.



The final linear control system, in matrix form, is:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{s} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & 0 & \cos(\theta) & 0 \\ 0 & 0 & \sin(\theta) & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ s \\ \theta \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ k_p \cdot [\theta^* - \theta] + \dfrac{k_{p2}}{\| X_1 - X \|} \cdot [\theta^{*2} - \theta] \end{bmatrix}$$

Utilizing the GPS Data

GPS modules give position in terms of latitude and longitude. Latitude and longitude are angular displacements from the equator and prime meridian, respectively. They are angular coordinates that create a grid on a spherical object, and not the simple two dimensional Cartesian coordinates that the linear control system uses to operate. Therefore, directly plugging latitudes and longitudes into the LCS as x and y coordinates would not work. In order to use our control system, we must convert the latitude and longitude given by the GPS into Cartesian coordinates. These conversions are flat-earth approximations: the formulas draw the angular coordinates onto a flat, two dimensional plane that is, in reality, tangential to the earth's surface. However, due to the large circumference of the earth, flat earth approximations have minimal errors for areas under several hundred kilometers.

The latitude and longitude coordinates were put through these two functions:

$$x = \cos(\phi) \cdot \sqrt{\frac{1}{(\frac{\sin(\phi)}{a})^2 + (\frac{\cos(\phi)}{c})^2}} \cdot [(longitude2 - longitude1) * \frac{\pi}{180}]$$

$$y = \sqrt{\frac{1}{(\frac{\sin(\phi)}{a})^2 + (\frac{\cos(\phi)}{c})^2}} \cdot [(latitude2 - latitude1) \cdot \frac{\pi}{180}]$$

where:

$$\phi = \frac{\pi}{2} - \frac{(latitude1 + latitude2)}{2} \cdot \frac{\pi}{180},$$

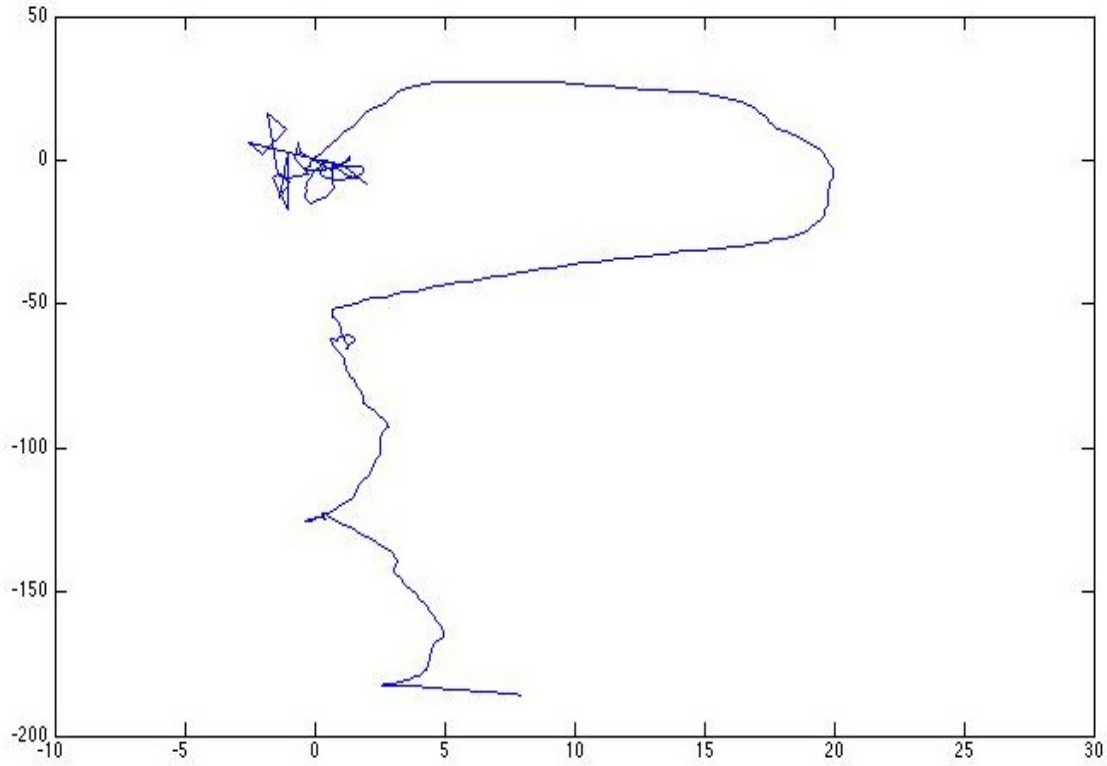
a = 6378136.6 meters, the equatorial radius of the earth, and c = 6356751.9 meters , the polar radius of the earth. These equations model the earth as a spheroid, instead of simple equations that model the earth as a sphere. This leads to more precise Cartesian approximations, due to a more precise radius of the earth (the radical term).

These equations converted a latitude/longitude path that very approximately resembles:

Figure 6: Approximate Path of Movement on Google Earth.

Into a Cartesian path of:

Figure 7: Cartesian Approximation of a path of Latitude/Longitude Points.



The path is rendered quite precisely in a Cartesian coordinate system[3]. The converted points can now be utilized in the linear control system.

Experimental Phase

The linear control system could not be used directly by the robot; in order to be tested, the controller had to be adapted into a Python code[4] that the robot could utilize. The Python code was only a rough approximation of the sophisticated controls detailed in the previous section. The Gaitrunner code could not tell the robot exactly the angle it needed to turn; instead, it told the robot to turn by a certain speed. The speed of the turn was determined on a -1 to 1 scale, in which -1 told the robot to turn as quickly as possible to the left, and 1 told it to turn as quickly as possible to the right. Giving the robot a value such as -0.5 to .5, or -.1 to .1, would have the robot move far more slowly and gently to the left or right.

The Python code that was eventually used to control the robot first calculated the robot's orientation, theta, by basic trigonometry. It took the tangent of the difference on the y axis and the x axis of the robot's last two coordinates. It then calculated the optimal

[3] See Appendix C for parsing/converting code in Python.
[4] See Appendix D for the Robot Controlling Python code.

orientation theta star, the angle from the robot's current position to the destination point, also utilizing basic trigonometry. If theta star was greater than theta the robot was commanded to turn by -1 (as quickly as possible) to the left, and vice versa. Also by simple trigonometry, the robot calculated the displacement from its current position to the destination point. Once some certain proximity to the destination point was reached, the robot would either start navigating toward the next way point, or sit down if the final way point had been reached. Though the path was inefficient as the robot weaved back and forth,, this basic code allowed the robot to successfully autonomously navigate to multiple GPS way points after it was set down in a field and the points programmed in. The necessary proximity was first set to five meters, then halved to two and a half meters. In both cases the robot reached this proximity and then continued on to the next way point.

Conclusion

There is a great deal of improvement that could be made to the Python code that controls the robot. Ultimately, its level of sophistication needs to rival that of the linear control system on which it was based. Gains on the turn speed must be tuned. Currently, the robot weaves so strongly to the right and left (resembling a sine wave) because even if the orientation theta is very close to the optimal orientation, the robot is still told to turn as quickly as possible to the left or right. A gain must be put on the turning speed so that the robot turns quickly when its angle is far from the optimal orientation, and by a very small amount when its angle is near. Once this is done, the code must again be adapted to resemble not just the basic controller, but also the enhanced controller that rounds corners by feeling input from the following point as the robot nears its initial destination.

Secondly, the controller itself could be improved. The enhanced θ' equation is nothing more than a poor approximation for a linear spline, one continuous mathematical function that smoothly links together multiple points. Utilizing linear splines in a controller would lead to a simpler, more efficient controller without the need for nearly as much gain tuning.

However, though there are many improvements to be made, the autonomization was ultimately a success. Using a GPS module, a linear control system, and basic trigonometry in the end, a code was designed that, given the absence of obstacles, could autonomously navigate a hexapodal robot from its current position to any set of multiple GPS way points.

References

[1] Riisgard, Søren and Blas, Morten Rufus. “SLAM for Dummies: A Tutorial Approach to Simultaneous Localization and Mapping”.

[2] U. Saranali, M. Buehler, and D.E. Koditschek. “Rhex: A Simple and Highly Mobile Hexapod Robot”. *The International Journal of Robotics Research*, Vol. 20, No.7. pp. 616-631, July 2001.

[3] Maria Isabel Ribeiro. “Obstacle Avoidance”. Instituto de Sistemas e Robótica, Instituto Superio Técnico, 2005.

[4] Seung-Hun Kim , Chi-Won Roh , Sung-Chul Kang and Min-Yong Park. “Outdoor Navigation of a Mobile Robot Using Differential GPS and Curb Detection”. 2007 IEEE International Conference on Robotics and Automation. Roma, 2007

[5] J. Borenstein, H. R. Everett, and L. Feng “Where am I?: Sensors and Methods for Mobile Robot Positioning”. University of Michigan, 1996.

[6] Dunlap, G.D. and Shufeldt, H.H., Dutton’s Navigation and Piloting, Naval Institute Press, pp. 557-579.

[7] Jane Yung-Jen Hsu, Der-Chiang, and Lo Shih-Chun Hsu. “Fuzzy Control for Behavior-Based Mobile Robots”. Department of Computer Science and Information Engineering National Taiwan University, 1994.

[8] Haruhiko Niwa, Kenri Kodaka, Yoshihiro Sakamoto, Masaumi Otake, Seiji Kawaguchi, Kenjirou Fujii, Yuki Kanemori, and Shigeki Sugano. “GPS-based Indoor Positioning system with Multi-Channel Pseudolite.” 2008 IEEE International Conference on Robotics and Automation Pasadena, 2008.

[9] Sanjiv Singh and Jay West. “Cyclone: A Laser Scanner For Mobile Robot Navigation”. The Robotics Insitute. Carnegie Mellon, 1991.

[10] Robert Ouellette, Kotaro Hirasawa. “A Comparison of SLAM Implementations for Indoor Mobile Robots”. 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems. San Diego, 2007.

[11] J. Seipel and P. Holmes. “A Simple Model for Clock-Actuated Legged Locomotion”. Regular and Chaotic Dynamics, 2007, Vol. 12, No. 5, pp. 502–520. c Pleiades Publishing, Ltd., 2007.

Appendix A

```
function [theta, X] = controller(start_x, start_y,
start_angle, end_x, end_y, speed, gain)
%X is the original state of the system. X = 0, Y = 0, S =
.1, theta = 0
    X = [start_x; start_y; speed; start_angle];
    X2 = [end_x; end_y; 0;0];


    xtilda = X2 - X;
    magxtilda = sqrt((X2(1) - X(1))^2 + (X2(2) - X(2))^2);
    i=1;

    while magxtilda > .001
        kp = gain; %gain
        theta = X(4);

        thetastar = atan2(X2(2) - X(2),X2(1) - X(1));
        thetadot = kp*sin(thetastar - theta);

        A =
[0,0,cos(theta),0;0,0,sin(theta),0;0,0,0,0;0,0,0,0];
%State equation.
        B = [0;0;0;thetadot]; %Input
        xdot = A*X + B;
        X = X + xdot*.1;
        magxtilda = sqrt((X2(1) - X(1))^2 + (X2(2) -
X(2))^2);

        subplot(2,1,1)
        plot(X(1),X(2), 'b')
        hold on
        plot(end_x,end_y, 'rx')
        hold on


        x(i) = X(1); % x axis data
        y(i) = X(2); % y axis data



    end

end
```

Appendix B

```
function [theta] = newcontroller(start_x, start_y,
```

```
start_angle, speed, gain)
    %newcontroller(initial x, initial y, initial
orientation, speed, gain)
    X = [start_x; start_y; speed; start_angle];
    i = true;
    j = 1;
    hold on
 while i == true
     endx(j) = input('Give the end point on the x axis: ');
     endy(j) = input('Give the end point on the y axis: ');
     i = input('Input another set of points? Y=1/N=0: ');
        j = j+1;
    end

  distance = sqrt((endx(1) - X(1))^2 + (endy(1) - X(2))^2);
    b = 1;

    g=1;
 while b < j
    while distance > g
       kp = gain; %gain
       kp2=.8*kp;
       theta = X(4);
       thetastar = atan2(endy(b) - X(2),endx(b) - X(1));
       thetastar2 = theta;

       if b+1<j
         thetastar2=atan2(endy(b+1)-X(2),endx(b+1)-X(1));
       end

       diff = thetastar - theta;
       diff2 = thetastar2-theta;
       thetadot1 = kp*(mod(diff + pi, 2*pi) - pi);
       thetadot2 = (kp2/distance)*(mod(diff2+pi, 2*pi)-pi);
       thetadot = kp*sin(diff) + (kp2/distance)*sin(diff2);

     A=[0,0,cos(theta),0;0,0,sin(theta),0;0,0,0,0;0,0,0,0];
%State equation.
         B = [0;0;0;thetadot]; %Input
         xdot = A*X + B;
         X = X + xdot*.1;
         distance=sqrt((endx(b)-X(1))^2+(endy(b)- X(2))^2);
         subplot(2,1,1)
         plot(X(1),X(2), 'b', endx(b), endy(b), 'rx')
          hold on
        end

     b = b+1;
     if b == j-1
         g = .1;
     end

     if b+1>j
        break;
```

```
      end
      distance = sqrt((endx(b)-X(1))^2+(endy(b) - X(2))^2);
    end
end
```

Appendix C

```
import math
# === We begin by creating the Objects ===
#Weird checksum function that no one understands.
def checksum(line):
    sum = 0
    for char in line:
        sum ^= ord(char)
        checksum_manual = "%02X" % sum
    if checksum != checksum_manual:
```

```
        print 'bad checksum at line %d' % lineno
    return [checksum_manual]

#Analyzes GPRMC lines of GPS data, gives out latitude, longitude, speed, orientation
def gprmc(line):

    #checksum(line)
    linedata = line.split(',')
    if linedata[3] == "": linedata[3] = float(0)
    lat = int(linedata[3][0:2]) + float(linedata[3][2:]) / 60.0
    lon = int(linedata[5][0:3]) + float(linedata[5][3:]) / 60.0
    speed = float(linedata[7])* 0.514444444 #meters/second
    angle = float(linedata[8])
    if linedata[4] == 'S':
        lat = -lat
    if linedata[6] == 'W':
        lon = -lon

    return [lat, lon, speed, angle]

#Analyzes gpgga lines of data
def gpgga(line):
    #checksum(line)
    linedata = line.split(',')
    #print linedata
    latgga = int(linedata[2][0:2]) + (float(linedata[2][2:]) * 60.0)/3600
    print latgga
    longga = int(linedata[4][0:3]) + (float(linedata[4][3:]) * 60.0)/3600
    if linedata[3] == 'S':
        latgga = -latgga
    if linedata[5] == 'W':
        longga = -longga
    quality = int(linedata[6])
    satnum = int(linedata[7])
    altitude = float(linedata[9])
    return [latgga, longga, quality, satnum, altitude]

#Analyzes gpgsa lines of data
def gpgsa(line):
    #checksum(line)
    linedata = line.split(',')
    mode = linedata[1]
    #if mode == "A": print "Automatic Mode"
    #elif mode == "M": print "Manual Mode"
    mode2 = int(linedata[2])
    #if mode2 == 1: print "Fix not available"
```

```
    #elif mode2 == 2: print "2D fix"
    #elif mode2 == 3: print "3D fix"
    j = 0
    for i in range(3,15):
        if linedata[i] != '': j=j+1
    print "There are " +str(j)+ " active satellites."
    pdop = linedata[-3]
    hdop = linedata[-2]
    vdop = linedata[-1]
    return [mode, mode2, j, pdop, hdop, vdop]

#Analyzes gpgsv lines of data
def gpgsv(line):
    #checksum(line)
    linedata = line.split(',')
    number = int(linedata[2])
    sentnum = int(linedata[3])
    satnumgsv = int(linedata[4])
    return [number, sentnum, satnumgsv]

def getCartesian(lat1, lon1, lat2, lon2):
        #equatorial radius
         a = 6378136.6
        #polar radius
         c = 6356751.9
         phi = (lat1+lat2)/2
         phi = math.pi/2 - (phi*math.pi/180)
         lon_diff = lon2-lon1
         lat_diff = lat2-lat1

         temp = (math.sin(phi)/a)**2 + (math.cos(phi)/c)**2
         r = (1/temp)**0.5

         lon_r = math.cos(phi)*r
         x = lon_r*lon_diff*math.pi/180
         y = r*lat_diff*math.pi/180
         return [x, y]


# ====  Time to Analyze ===
gpsfile = open('/Users/ryo/GPS2e.txt','r')

lat, lon, speed, angle = [],[],[],[]
latgga, longga, quality, satnum, altitude = [],[],[],[],[]
mode, mode2, j, pdop, hdop, vdop = [],[],[],[],[], []
number, sentum, satnumgsv = [],[],[]
```

```
coordinates,x,y,dx,dy=[],[],[],[],[]


lineno = -1
while 1:
    lineno = lineno + 1
    line = gpsfile.readline()
    linesplit = line.split(',')
    if line == '': break
    line = line.strip()
    id = line[0:6]
    (line,checksum) = (line[0:-3],line[-2:])
    if id == '$GPRMC':

        if len(linesplit)<6: break
        data = gprmc(line)
        lat.append(data[0])
        lon.append(data[1])
        speed.append(data[2])
        angle.append(data[3])
    elif id == '$GPGGA':
        if len(linesplit)<6: break

        data = gpgga(line)
        if data[0] != 0:
            latgga.append(data[0])
            longga.append(data[1])
            quality.append(data[2])
            satnum.append(data[3])
            altitude.append(data[4])
    elif id == '$GPGSA':
        if len(linesplit)<6: break
        data = gpgsa(line)
        mode.append(data[0])
        mode2.append(data[1])
        j.append(data[2])
        pdop.append(data[3])
        hdop.append(data[4])
        vdop.append(data[5])
    elif id == '$GPGSV':
        if len(linesplit)<6: break
        data = gpgsv(line)
        lat.append(data[0])
        lon.append(data[1])
        speed.append(data[2])
```

```
for i in range(0,len(longga)):
    coordinate = [latgga[i], longga[i]]
    coordinates.append(coordinate)
    re = 6371*1000
    y.append(re*(latgga[i]-latgga[0])*math.pi/180)
    x.append(re*(longga[i]-longga[0])*(math.pi/180)*math.cos(longga[0]*math.pi/180))
    ex, ey = getCartesian(latgga[0], longga[0], latgga[i], longga[i])
    if ex < -60:
        print i
    dx.append(ex)
    dy.append(ey)
gpsfile.close()
```

Appendix D:

```
import math
import dy
import time
dy.init()
dy.network.connect('penn4', 8650)
dy.signal.send('calibrate_start')
time.sleep(5)
dy.signal.send('gaitrunner_start')
time.sleep(5)
```

```
dy.signal.send('stand_start')
time.sleep(5)
ds = dy.data.path('penn6')
dyturn = dy.data.create_at(ds, dy.DY_FLOAT, 'gait.turn')
dyspeed = dy.data.create_at(ds, dy.DY_FLOAT, 'gait.speed')
dy.data.set_float(dyspeed, 0 ,0);
dy.network.push("penn6.gait.speed");
dy.data.set_float(dyturn,0,0)
dy.network.push("penn6.gait.turn");

# === We begin by creating the Objects ===

#Weird checksum function that no one understands.
def checksum(line):
    sum = 0
    for char in line:
        sum ^= ord(char)
        checksum_manual = "%02X" % sum
    if checksum != checksum_manual:
        print 'bad checksum at line %d' % lineno
    return [checksum_manual]

#Analyzes GPRMC lines of GPS data, gives out latitude, longitude, speed, orientation
def gprmc(line):

    #checksum(line)
    linedata = line.split(',')
    if linedata[3] == "": linedata[3] = float(0)
    lat = int(linedata[3][0:2]) + float(linedata[3][2:]) / 60.0
    lon = int(linedata[5][0:3]) + float(linedata[5][3:]) / 60.0
    #speed = float(linedata[7])* 0.514444444 #meters/second
    #angle = float(linedata[8])
    if linedata[4] == 'S':
        lat = -lat
    if linedata[6] == 'W':
        lon = -lon

    return [lat, lon]

#Analyzes gpgga lines of data
def gpgga(line):
    #checksum(line)
    linedata = line.split(',')
    latgga = int(linedata[2][0:2]) + (float(linedata[2][2:]) * 60.0)/3600
    longga = int(linedata[4][0:3]) + (float(linedata[4][3:]) * 60.0)/3600
    if linedata[3] == 'S':
```

```
        latgga = -latgga
    if linedata[5] == 'W':
        longga = -longga
    quality = int(linedata[6])
    satnum = int(linedata[7])
    altitude = float(linedata[9])
    return [latgga, longga, quality, satnum, altitude]

#Analyzes gpgsa lines of data
def gpgsa(line):
    #checksum(line)
    linedata = line.split(',')
    mode = linedata[1]
    mode2 = int(linedata[2])
    j = 0
    for i in range(3,15):
        if linedata[i] != '': j=j+1
    #print "There are " +str(j)+ " active satellites."
    pdop = linedata[-3]
    hdop = linedata[-2]
    vdop = linedata[-1]
    return [mode, mode2, j, pdop, hdop, vdop]

#Analyzes gpgsv lines of data
def gpgsv(line):
    #checksum(line)
    linedata = line.split(',')
    number = int(linedata[2])
    sentnum = int(linedata[3])
    satnumgsv = int(linedata[4])
    return [number, sentnum, satnumgsv]

def getCartesian(lat1, lon1, lat2, lon2):
        #equatorial radius
         a = 6378136.6
        #polar radius
         c = 6356751.9
         phi = (lat1+lat2)/2
         phi = math.pi/2 - (phi*math.pi/180)
         lon_diff = lon2-lon1
         lat_diff = lat2-lat1

         temp = (math.sin(phi)/a)**2 + (math.cos(phi)/c)**2
         r = (1/temp)**0.5

         lon_r = math.cos(phi)*r
```

```
        x = lon_r*lon_diff*math.pi/180
        y = r*lat_diff*math.pi/180
        return [x, y]


# ====  Time to Analyze ===
gpsfile = open('/dev/ttyS1');

lat, lon, speed, angle = [],[],[],[]
latgga, longga, quality, satnum, altitude = [],[],[],[],[]
mode, mode2, j, pdop, hdop, vdop = [],[],[],[],[], []
number, sentum, satnumgsv = [],[],[]
coordinates,x,y,dxx,dyx,endx,endy,pathx, pathy=[],[],[],[],[],[],[],[],[],[]
i=1
j=0;
b = 0
while i == 1:
   deg = raw_input('Please give the latitudinal coordinate,
degrees.minutes.seconds.hemisphere: ').split('.');
   degx = int(deg[0]) + (float(deg[1]) + (float(deg[2] + "." + deg[3])/60))/60

   if deg[4] == 'S':
      degx = -degx
   deg2 = raw_input('Please give the longitudinal coordinate, degrees.minutes.seconds:
').split('.');
   deg2x = int(deg2[0]) + (float(deg2[1]) + (float(deg2[2] + "." + deg2[3])/60))/60
   if deg2[4] == 'W':
      deg2x = -deg2x
   print degx, deg2x
   endx.append(degx)
   endy.append(deg2x)

   i = raw_input('Would you like to input another set of points? Y=1/N=0: ');
   i = int(i)
   j = j+1;

for i in range(0,j):
   path_x, path_y = getCartesian(39.9535, -75.1916388889, endx[i],endy[i])
   pathx.append(path_x)
   pathy.append(path_y)

lineno = -1
distance = 100

while b < j:
   while distance > 5:
```

```
        lineno = lineno + 1
        line = gpsfile.readline()
        linesplit = line.split(',')
        if line == '': break
        line = line.strip()
        id = line[0:6]
        (line,checksum) = (line[0:-3],line[-2:])
        if id == '$GPGGA' and linesplit[4] != "":

            dy.data.set_float(dyspeed, 1 ,0);
            dy.network.push("penn6.gait.speed" );

            data = gpgga(line)
            latgga.append(data[0])
            longga.append(data[1])
            quality.append(data[2])
            satnum.append(data[3])
            coordinate = [data[0], data[1]]
            coordinates.append(coordinate)
            ex, ey = getCartesian(39.9535, -75.1916388889, data[0], data[1])
            dxx.append(ex)
            dyx.append(ey)

            if len(dyx) > 1:
              delta_y = dyx[len(dyx)-1] - dyx[len(dyx) - 2]
              delta_x = dxx[len(dxx)-1] - dxx[len(dxx) - 2]
              theta = math.atan2(delta_y,delta_x)
              thetax = theta*180/math.pi
              thetastar = math.atan2(pathy[b] - dyx[len(dyx)-1],pathx[b] - dxx[len(dxx) - 1])
              thetastarx = thetastar*180/math.pi
              diff = (((thetastar - theta)+math.pi)%(2*math.pi)) - math.pi

              print thetax, thetastarx
              if diff > 0:
                 print "turn left!"
                 dy.data.set_float(dyturn,-1,0)
                 dy.network.push("penn6.gait.turn");

              elif diff < 0:
                 print "turn right!"
                 dy.data.set_float(dyturn,1,0)
                 dy.network.push("penn6.gait.turn");

              distance = math.sqrt((pathx[b] - dxx[len(dxx)-1])**2 + (pathy[b] -
dyx[len(dyx)-1])**2)
              print distance
```

```
   print "Target Reached!"

   b = b+1
   if b + 1 > j:
      dy.signal.send('sit_start')
      break

   distance = math.sqrt((pathx[b] - dxx[len(dxx)-1])**2 + (pathy[b] - dyx[len(dyx)-
1])**2)
   print distance
gpsfile.close();
```