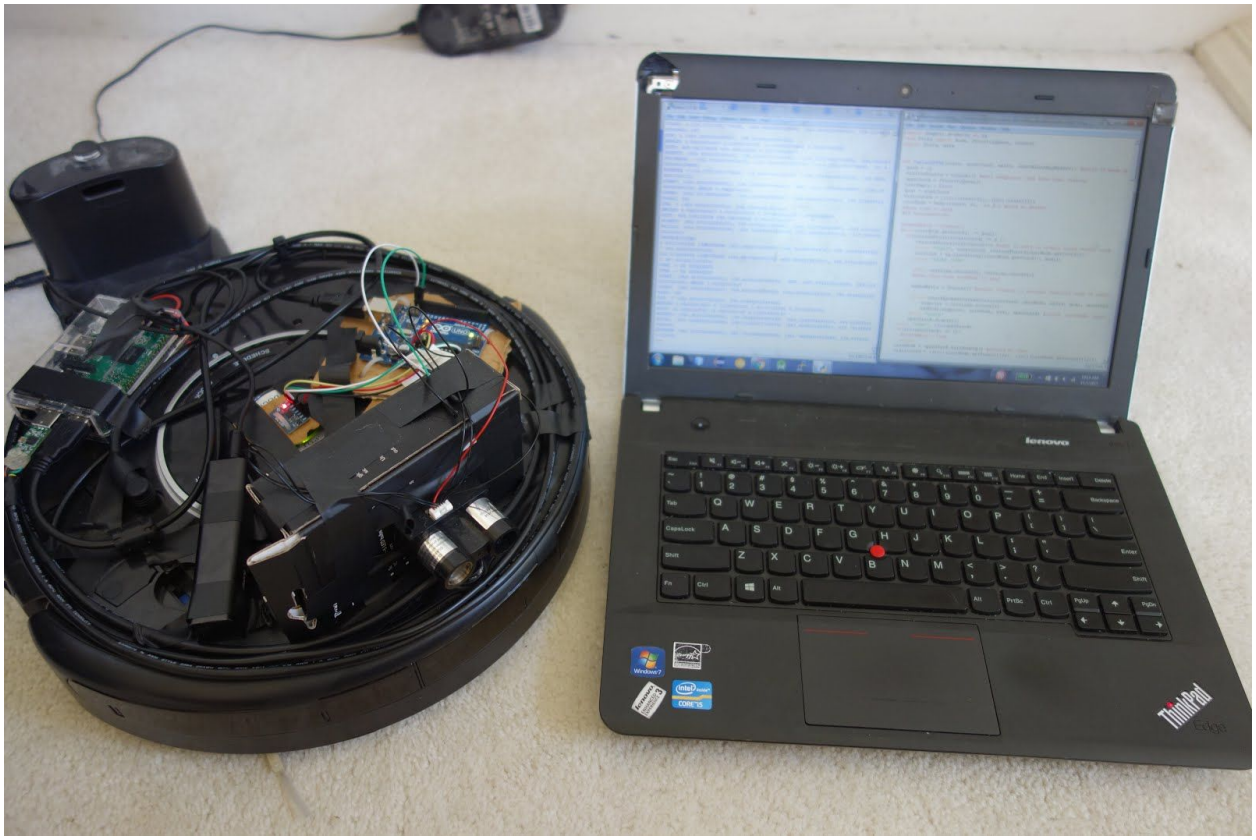


Maze Solving Robot

Justin Lin

Components:

For my maze solving robot, I used:

- iRobot Roomba 595
    - The main base for my robot, and what is being controlled
    - Features 2 individually controllable motors, both with wheel encoders
- Arduino Uno
    - For taking input from my sensors
- MPU-6050 Gyro and Accelerometer
    - To track the orientation of my robot
- PulsedLight LIDAR-Lite Rangefinder
    - To measure the proximity to objects
- Raspberry Pi 2 - Model B (with Wi-Fi adapter)
    - For processing sensor readings from the Arduino, carrying out the planning and mapping algorithm, and sending signal to the Roomba
- Jackery External Power Bank
    - To provide power to the Roomba, Arduino, and Raspberry Pi
- Lenovo Thinkpad
    - To remotely control the robot and display visual output of mapping

Communication:

The Roomba is controlled over a serial connection to the Raspberry Pi, which allows the Roomba to both send encoder data and receive commands. The Arduino is running a script to read and process raw values from my LIDAR rangefinder and gyro, and sends data to the Raspberry Pi over serial as well. The Raspberry Pi runs the main Python script controlling the robot. It parses the sensory data, and sends controls the Roomba. To execute the script on the Raspberry Pi remotely, I connected over SSH using Putty with my laptop, and also used X-11 forwarding from the Pi to allow the Raspberry Pi to draw output on my laptop screen, so I could see the planning and mapping.

Moving the Robot:

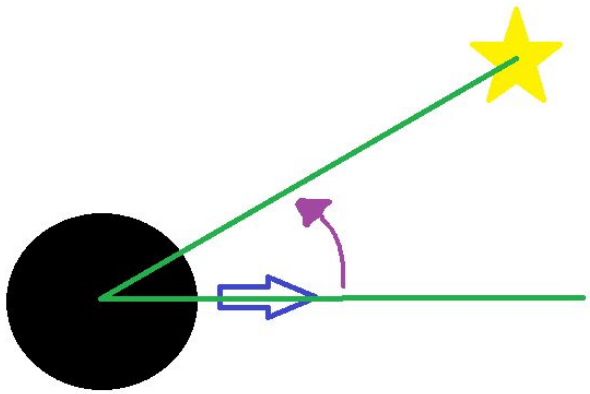
To control the Roomba, I made use of iRobot's Roomba Open Interface API, which allows the user to send and receive packets of data. I made use of the tank drive command for the Roomba, which would allow me to power each side of the robot individually. The Roomba drives at a constant speed, but I can turn and steer the Roomba by telling one side to move faster than the other.

During my program, I give a coordinate for the robot to travel to, and the direction and trajectory of the robot is based off of the difference between the current heading and the angle necessary to travel to the next point.

Code to allow each side of Roomba to move at different speeds

```
def tankDrive(self, power, powerDiff): #speed to move (from -1 to 1), and difference in power between left and right (from -2 to 2)
    speed = power*500 #max power is 500 mm/s
    dataArray =[145] #array of bytes to send Roomba; first byte signals a drive command

    #modify speeds based on powerDiff; 2 is full power left, -2 full power right, 0 full power both
    rightSpeed = speed
    leftSpeed = speed
    if(powerDiff > 0): #wants to have more power to the right (turns left)
        leftSpeed *= (1-powerDiff) #limit right motor
    else: #more power to right (or equal)
        rightSpeed *= (1+powerDiff) #limit left motor
    dataArray += intToHexBytes((int)(rightSpeed)) #convert right speed to bytes to send to robot
    dataArray += intToHexBytes( (int)(leftSpeed)) #convert left speed to bytes to send to robot
    self.writeArray(dataArray) #write the array of data to the Roomba


def writeArray(self, arr):
    for x in range(len(arr)):
        self.serial.write(chr(int(arr[x])))
```

Robot would turn based on difference between current orientation (blue) and angle to goal (green)



Code to determine how to move robot. Note that while the option for PID control is there, in my tests I used solely P (difference in angle)

```
def PID(path, params, elapsed): #time elapsed in seconds
    global cte, prev_cte, accum_error, diff_error, pidIndex
    prev_cte = cte
    nextPoint = path[pidIndex+1] #get current target point

    goalAngle = math.atan2( (nextPoint[1] - roomba.getY()),(nextPoint[0] - roomba.getX())) % Utils.tau #get target angle
    angleDiff = Utils.findAngleDiff(goalAngle, roomba.getAngle()) #get angle difference

    if(roomba.getPlanning() and abs(angleDiff) < .2):#basically reached target heading; this is to prevent the
                                                     #Roomba from immediately sensing a wall after replanning
        roomba.setPlanning(False) #allow the robot to check for collisions again

    cte = -angleDiff #store current error
    if(Utils.reachedPoint(roomba.getX(), roomba.getY(), nextPoint[0], nextPoint[1])): #if close enough to target point
        pidIndex  += 1 #set destination to next point in path


    diff_error = float(cte - prev_cte)/elapsed #find how error is changing over time
    accum_error += cte*elapsed #find total error accumulated

    steeringDir = -params[0] * cte - params[1] * diff_error - params[2] * accum_error #multiply errors by P, I, andD

    #constrain steering angle to [-2, 2]
    if(steeringDir > 2):
        steeringDir = 2
    elif(steeringDir < -2):
        steeringDir = -2
    roomba.tankDrive(.45, steeringDir) #move robot at .45 speed, at calculated angle
```

Sensing the Environment:

To gather data, I used several sensors. First of all I used the encoders that the Roomba has on each wheel to determine distance moved. This data was transferred to the Raspberry Pi over serial.

In addition, I used an MPU-6050 to get the current orientation of the robot, and a LIDAR sensor to check proximity to other objects, both attached to the Arduino. I used the I2C Device Library to read raw sensor values from the sensors plugged into the digital pins on the board. The Arduino continually writes data through the serial communication, which the Raspberry Pi receives and processes.

One challenge I faced was compensating for the gyro drifting over time - depending on conditions, the gyro drifted at varying rates. To solve this issue, at the beginning of my main script, I take several readings when the robot is at rest, and measure the rate at which it is changing. Then, when I need to calculate the robot's orientation, I'm able to modify the returned amount based on how much time has elapsed since the program has started.

Method for getting values from Roomba's encoders

```
def updateSensors(self, elapsed): #updates encoderVals array - contains [leftVal (raw), leftDelta (in mm), rightVal, rightDelta]
    #print("Update Sens")
    dataSize = 4 #bytes being sent
    dataArray = [149, (dataSize - 2), 43, 44] #requests 2 sensor readings from Roomba - left and right encoder values
    self.writeArray(dataArray) #sends request
    time.sleep(.03) #wait for sensor values to return

    index = 0 #index for which part of data array is being modified
    val = -1
    valid = True
    while(self.serial.inWaiting() > 0 and index < len(self.encoderVals)): #gets both raw encoder values  -
                                                                          #600 is about 300 mm (2 units is 1 mm)
        #readings returned as pairs of high and low bytes
        high = self.serial.read()
        low = self.serial.read()
        reading = bytesToLong(high, low) #converts bytes to int

        val = reading - self.encoderVals[index] #finds difference between current reading and previous reading

        if(val < -1000): #if encoder is less than previous val, it has rolled around
            val += 65535 #encoder jumped so add max val for encoder to compensate

        if(abs(val) > elapsed*2000): #compares distance moved with time elapsed to see if reading is reasonable
            valid = False #marks reading as bad
            print("BAD READING")
        else: #otherwise, valid reading
            self.encoderVals[index] = reading #stores raw value
            index+= 1
            self.encoderVals[index] = round(val/2) #stores delta (raw val converted to mm)
            index += 1
    return valid #returns whether reading was true or not
```

C++ code run on Arduino to send sensor readings

```
accelgyro.getRotation(&gx, &gy, &gz); //get gyro readings
t2 = millis() - t1;
t1 = millis();
zAngle += (float)gz * scaleConstant * (float)t2/1000; //integrate rate by multiplying by time elapsed, then multiply by constant to convert raw vals to degrees

String(zAngle).toCharArray(reading, 8); //convert to int to char
Serial.write(reading); //write data
Serial.write('!');//send end code

updateLIDAR(); //updates LIDAR array
int distance = (distanceArray[0] << 8) + distanceArray[1];  // Shift high byte [0] 8 to the left and add low byte [1] to create 16-bit int; gets distance in cm

String(distance).toCharArray(reading, 8);  //convert int to char
Serial.write(reading);
Serial.write('@');

delay(10); //delay 10 ms before sending data again
```

Code to get values from the Arduino

```
 def getReadings(self):
     fullReading = []
     curr = ""
     #check for end of previous message, to ensure full reading (nothing cut off)
     while(curr != '@'): #end of previous data ends with @
         curr = self.serial.read()     #discard until @ reached
     fullReading.append(self.readValue('!')) #gyro reading ends with !
     fullReading.append(self.readValue('@')) #lidar reading ends with @

     self.serial.flushInput() #need to flush to get newest serial data, since arduino writes faster than code reads
     return fullReading #return gyro and lidar readings
```

Code to get gyro drift over time

```
#CALCULATE DRIFT - find slope at 5 points, calc average
drift = []
t1= time.time() #mark starting time
for x in range(5):
    reading = arduino.getReadings() #get current angle
    angle = float(reading[0]) - initAngle #gets change in angle since absolute beginning
    t2=time.time()-t1 #find time that's elapsed
    t1=time.time()
    diff = (angle - prevAngle)/t2 #gets change in angle/sec
    drift.append(diff) #add val to array
    prevAngle = angle #save curr val to compare next value with
    time.sleep(1) #wait 1 sec
kDrift = sum(drift)/len(drift) #find average rate
```

Localization:

The robot moves and maps with Cartesian coordinates, relative to its starting position (which is the origin). Initially, I attempted to use odometry to estimate the robot's position, using values from the encoders, basing my code off of the algorithm for robots with differential drive. (http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-186-mobile-autonomous-systems-laboratory-january-iap-2005/study-materials/odomtutorial.pdf) However, I soon found that the estimation quickly became inaccurate, likely because the carpet the robot was moving across allowed for some slippage.

To find a better estimate of position, that's where the gyroscope that I was using comes into play. Using the current angle of trajectory, and the total distance moved, I can easily calculate the change in X and Y using trigonometry. After accounting for gyro drift, I found that this provided a satisfactory estimate of robot position.

Initial attempt to localize using odometry

```
def updateOdomEst(self, valid, elapsed):
    dist_between_wheels = 230.0  #in mm
    angle = self.odomEst[2] #gets last angle

    if(valid):#if a valid reading was received from encoders
        #update position with odometry
        if( abs( self.encoderVals[3] - self.encoderVals[1]) < 10): #tolerance robot moving straight
            self.odomEst[3] = ( self.encoderVals[1]+ self.encoderVals[3])/2 * math.cos(angle) #gets dx
                                                                    #(averages data from both wheels to get speed)
            self.odomEst[4] = ( self.encoderVals[1]+ self.encoderVals[3])/2 *math.sin(angle) #gets dy (using average dist moved)
            self.odomEst[5] = 0 #if going straight, no change in angle

        else: #robot travelling in arc

            dCenter = ( self.encoderVals[1] + self.encoderVals[3]) / 2 #distance center of robot moved
            phi = ( self.encoderVals[3] - self.encoderVals[1] ) / dist_between_wheels #angle that robot moved around center
            rCenter = dCenter/phi #solve for center r*theta = arcLength

            self.odomEst[3] = rCenter*( math.sin(phi + angle) -math.sin(angle) )  #calculate dx based on geometry
            self.odomEst[4] = rCenter*( math.cos(angle) - math.cos(phi + angle) ) #calculate dy based on geometry
            self.odomEst[5] = phi #dAngle = phi


    else: #if not valid, pretend robot moved same as it did the last time, keep things the same
        print "LAST VALS PRESERVED"
    #add rates to last vals to get new vals
    self.odomEst[0] += self.odomEst[3]
    self.odomEst[1] += self.odomEst[4]
    self.odomEst[2] += self.odomEst[5]
    self.odomEst[2] %= Utils.tau
```

Code which unites readings from both Arduino and Roomba to update estimate

```
def updateVals(timeStep, updatePos): #uses readings from Arduino to estimate robot position
    global start, kDrift, initAngle
    reading = arduino.getReadings() #gets values from Arduino
    elapsed = time.time() - start
    angle = float(reading[0]) - initAngle + elapsed*-kDrift #finds change in angle from initial
                                            #position based on time elapsed and drift constant
    radAngle = (math.radians(angle)) % Utils.tau #keeps angle within [0, 2pi]
    lidarDist = int(reading[1]) #gets LIDAR val
    roomba.update(t2, radAngle, lidarDist, updatePos) #updates Roomba's estimate of position
```

Method to update estimate of robot position, along with a way to handle bad data

```
def updatePosition(self, valid, angle):
    if(valid): #if given a valid reading
        self.vals[3] = ( self.encoderVals[1]+ self.encoderVals[3])/2 * math.cos(angle) #finds dX based on avg. dist and angle
        self.vals[4] = ( self.encoderVals[1]+ self.encoderVals[3])/2 *math.sin(angle) #finds dY based on avg. dist and angle
        self.vals[5] = angle - self.vals[2] #finds how rate of angle change
    else: #do not update dX or dY; assume robot continues to travel with same direction and speed
        print "USE LAST VALS"
    self.vals[0] += self.vals[3] #update X based on dX
    self.vals[1] += self.vals[4] #update Y based on dY
```

The Algorithm:

Now that I knew where the robot was, and could measure the distance to obstacles, I was ready to navigate through and map a maze. A summary of the algorithm is as follows:

1. A goal destination (coordinate) or path (set of coordinates) is supplied
2. Establish communication with all the devices, initialize starting values
3. Head towards target point (preferably in a straight line)
    a. While moving, update robot sensors and position
4. Once it has reached target point, go on to next point in path
5. However, if the robot detects an object that is too close:
    a. Spin 360 degrees, polling LIDAR readings periodically to get distance to nearby objects
    b. Parse these individual data points, and form lines (walls) using data
    c. Then, replan the path knowing the location of the walls
    d. Resume navigation

Mapping and Parsing Data:

To estimate the location of obstacles, I needed to make sense of the individual points of data I received. As the robot spun, using its current angle and the LIDAR reading, I could estimate the location of the nearest obstacle in that direction. After spinning 360 degrees and scanning its entire surroundings, I needed the robot to identify which groups of points formed lines. To do that, I chose a point, and then looked at adjacent points. By comparing their proximity, as well as the slope between the points, I determined whether they formed a line or not. To reduce the chance of error, I only considered points that were within a certain threshold of the robot. When creating the line, I also made sure to extend the lines a bit, to account for the fact that I wasn't taking an uninterrupted scan of the room, but instead periodic points, and as a result there would be some gaps in the coverage. One issue that I needed to account for was the fact that the robot would double-cover some spots, because it spun slightly more than a full circle. For points that were scanned twice (at the start and end of the spin), I averaged their location.

Code to group data set of points into lines

```
while(len(fixedData) > 0):
    length = 1 #length of line (# of points in it)
    startPoint = fixedData[index] #get starting point
    currPoint = startPoint
    if(len(fixedData) > 1):
        distMet = True #marks if point is within sight threshold (too far and it's not considered when drawing lines)

        #get point to the right of current point in the list
        rightInner = currPoint
        rightPoint = fixedData[(index+1)%len(fixedData)]

        #find angle (slope) between 2 points
        rDx = rightPoint[0] - currPoint[0]
        rDy = rightPoint[1] - currPoint[1]
        rightAngle = math.atan2(rDy, rDx)%Utils.tau

        angles = [rightAngle]
        currAngle = sum(angles)/len(angles) #average slope/angle of all the points in the line

        #while points are adjacent, and have similar slope
        while(distMet and pointsAdjacent(currPoint, rightPoint, minDist) and
              slopeSimilar(currPoint, rightPoint, currAngle, angleTolerance) and len(fixedData) > 0):

            angles.append(getAngleBetween(currPoint, rightPoint))
            currAngle = sum(angles)/len(angles) #calculate new avg slope

            currPoint = fixedData.pop((index+1)%len(fixedData)) #remove the adjacent point (and make it the current point)
            length += 1
            if(len(fixedData) > 0):
                rightPoint = fixedData[(index+1)%len(fixedData)]
            rightInner = currPoint
            if(rightBound[2] > sightThreshold): #over sightThreshold (min lidar reading for wall)
                distMet = False




        #find left bound
        currPoint = startPoint
        leftInner = currPoint
        currAngle = sum(angles)/len(angles)
        if(len(fixedData) > 0):
            leftPoint = fixedData[(index-1)%len(fixedData)]
            distMet = True
            while(distMet and pointsAdjacent(leftPoint, currPoint, minDist) and
                  slopeSimilar(leftPoint, currPoint, currAngle, angleTolerance) and len(fixedData) > 0):
                angles.append(getAngleBetween(leftPoint, currPoint))
                currAngle = sum(angles)/len(angles)
                currPoint = fixedData.pop((index-1)%len(fixedData)) #gets rid of current point
                length +=1
                if(len(fixedData) > 0):
                    leftPoint = fixedData[(index-1)%len(fixedData)]

                leftInner = currPoint
                if(leftBound[2] > sightThreshold):#over sightThreshold (min lidar reading for wall)
                    distMet = False
        if(length > 3): #3 points make a line
            newWall = extendLine(leftInner, rightInner, robotWidthAndBuffer) #extend the line a bit to give some buffer
            walls.append(newWall)
    else: #no more points to consider (avoids index out of bounds)
        print "LAST POINT"
        #drawPoint(pointsDraw, currPoint)
    if(len(fixedData) > 0):
        fixedData.pop(index) #get rid of point
return (walls, copyData) #return walls, data points
```

Planning:

After determining where the walls were, the next step was to figure out how to navigate around them to reach the goal. I knew that I wanted to use ASTAR, but I needed to decide what I wanted to use as the nodes. In the end, I decided on the endpoints of walls. I drew a line from the robot to the target point, and if it intersected one of the walls, I added the endpoints of that wall to the queue. Moving down the queue, I continued to draw lines, check for intersections, and add more endpoints until I reached my goal, or discovered there was no way out. One caveat to the planning was that since my robot had a certain width, I needed to make sure to have some buffer around the wall so I wouldn't collide with it, but be able to drive around it.

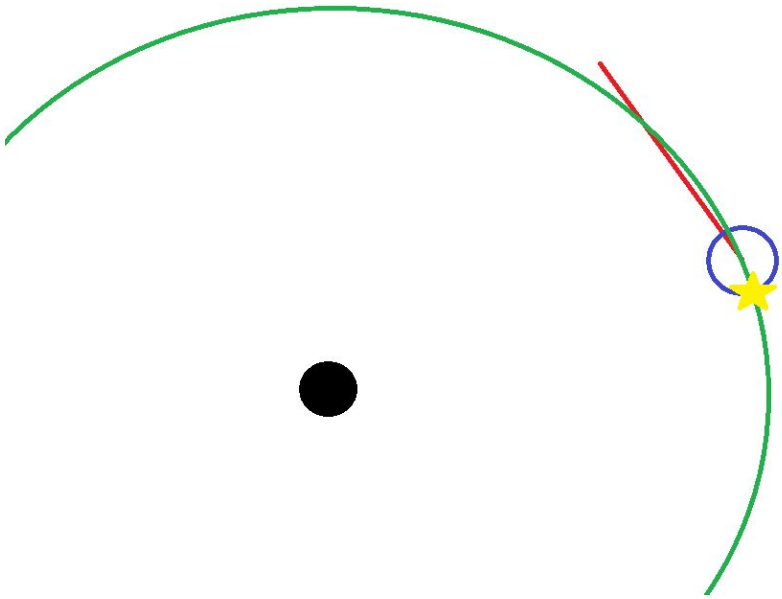
ASTAR algorithm which operates by checking walls for intersections

```
#ASTAR using line segments in a nondiscrete space
def replanASTAR(start, goalCoord, walls, robotWidthAndBuffer): #walls made up of LineStrings
  solution = [] #list of coords to travel to
  visitedPoints = Counter() #wall endpoints that have been visited
  openStack = PriorityQueue() #list of possible points to explore (and costs)
  listEmpty = False #whether search is over or not
  goal = goalCoord #ending point of the path
  visitCoord = ((int)(start[0]), (int)(start[1])) #current coord being visited to the closest int; gives some tolerance for math
  currNode = Node((start, 0), -1) #-1 means no parent

  while(currNode.getCoord() != goal): #continue until goal reached (or all points exhausted)
    if(visitedPoints.getCount(visitCoord) == 0 ):
      visitedPoints.addItem(visitCoord)
      currLine = sg.LineString([currNode.getCoord(), goal]) #predict path from current coord to goal
      addedWalls = Counter() #create Counter to prevent infinite loop of adding
      if not(checkAndAddIntersection(currLine, currNode, walls, goal, openStack, addedWalls, robotWidthAndBuffer)):
                                                              #if predicted path doesn't intersect any lines
          endpoint = currLine.coords[1]
          addNode(endpoint, currNode, goal, openStack) #create new node using end of path, add to queue

    if(len(openStack) == 0): #list is empty, no solution
      #listEmpty = True
      break #exit from search
    currNode = openStack.sortAndPop() #get next coord from queue (the one with lowest cost)
    visitCoord = ((int)(currNode.getCoord()[0]), (int)(currNode.getCoord()[1])) #round coord to int
                                                                  #(to check if this point has been visited before)
  if(currNode.getCoord() == goal): #if goal was found
    while(currNode.getCoord() != start): #backtrack to start
          solution.append(currNode.getCoord()) #get coord of node
          currNode = currNode.getParent() #get previous node
    solution.append(currNode.getCoord()) #finally, add the start coord, but don't try to get parent
    solution.reverse() #reverse path to have it from start to goal
  else: #no point was found
    print "NO PATH"
  return solution #return solution (if none, empty list)
```

Code to create buffer around wall

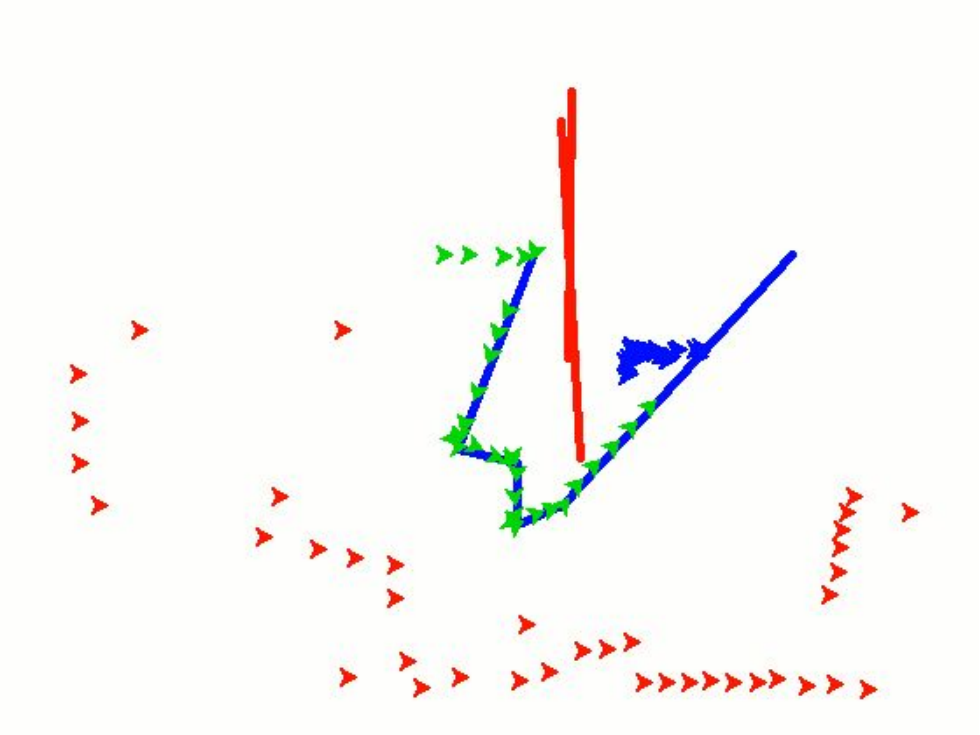
```
def addRoundNode(currWall, nextCoord, direction, currNode, walls, goal, queue,  addedWalls, robotWidthAndBuffer):
  currCoord = currNode.getCoord()
  minRadius = robotWidthAndBuffer  #convert distance to mm + robot radius
  distance = Utils.getDist(currCoord[0], currCoord[1], nextCoord[0], nextCoord[1]) #distance from robot to wall

  #create circles; radius = buffer
  c1 = sg.Point(currCoord).buffer(distance) #circle that tracks distance to wall
  c2 = sg.Point(nextCoord).buffer(minRadius) #circle to get enough area around obstacle
  intersection = c1.intersection(c2)

  #endpoint = point at which robot is far enough from wall (in given direction)
  endpoint = findBounds(currWall, currNode, intersection.exterior.coords, direction)#0 is below, 1 is right, 2 is above, 3 is left

  #gets intermediary angle between end of wall and
  a1 = math.atan2(nextCoord[1]-currCoord[1], nextCoord[0]-currCoord[0]) % Utils.tau
  a2 = math.atan2(endpoint[1]-currCoord[1], endpoint[0]-currCoord[0]) % Utils.tau
  addAngle = Utils.findAngleDiff(a2, a1)/2.0 #angle difference
  targetAngle = (a1 + addAngle) % Utils.tau

  #given avg angle of endPoint and currPoint; extend path - want robot to be in middle of gap
  targetRadius = abs(distance) #radii should always be positive value
  targetPoint = (currCoord[0] + targetRadius * math.cos(targetAngle), currCoord[1] + targetRadius * math.sin(targetAngle))
                    #get point at end of the line
```

Visual Representation of rounding code - the blue circle is the desired buffer around the wall, green circle is the distance to the wall, and yellow star is the point to move the robot to



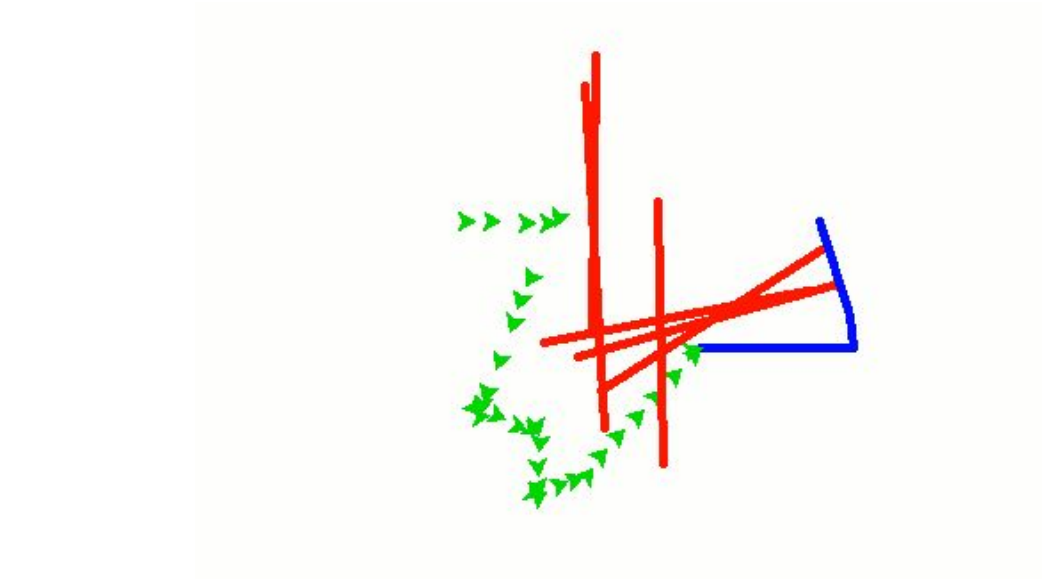
Displaying Data:

Finally, I wanted to visually represent the data, so I could better understand what my robot was doing (rather than simply printing out data points), I used Python's turtle module to display the robot's estimation of its environment. In addition, I also created a script that simulated a set of data, so I could test and debug my mapping and planning algorithms without having to worry about other variables. I could ensure that the set of data entered was consistent, and see what the problems were with my methods.



Real image of data being parsed live - the blue and red arrows show where obstacles have been detected

---



New walls and path after processing the data - the green dots are the path the robot has taken, the red lines where obstacles are, and the blue lines the target path

Code to simulate and test mapping and planning functions

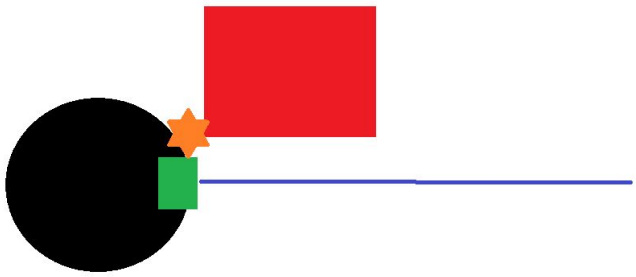
```
def drawLines(pen, p1, p2): #method to draw line from point to point
    global scalingFactor
    p1X = p1[0]
    p1Y = p1[1]
    p2X = p2[0]
    p2Y = p2[1]
    pen.penup()
    pen.goto(p1X*scalingFactor, p1Y*scalingFactor)
    pen.pendown()
    pen.goto(p2X*scalingFactor, p2Y*scalingFactor)

def drawPoint(pen, data): #stamps point at correct point
  global scalingFactor, sightThreshold
  pen.goto(data[0]*scalingFactor, data[1]*scalingFactor)
  pen.stamp()
  pen.goto(0,0)

for x in range(len(dataSet)): #get all data points, draw them
    drawPoint(pointsDraw, dataSet[x])

walls = [] #walls initially empty
#parse data to get walls
walls = RoombaFunctParseData.manageData(dataSet, walls, currPos, minDist, sightThreshold,
                                        angleTolerance, robotWidthAndBuffer, samePointDist, False)[0]
for x in range(len(walls)): #draw the walls
    drawLines(linesDraw, walls[x].coords[0], walls[x].coords[1])

path = RoombaFunctASTAR.replanASTAR(currPos, (1000,0), walls, robotWidthAndBuffer) #plan path using walls
for x in range(len(path)-1): #draw planned path
    drawLines(pathDraw, path[x], path[x+1])
```

Problems:

Throughout the project, I faced many challenges and difficulties. One major issue was trying to keep the control system on my robot portable and compact. While I tried to use the Arduino at first, it lacked the processing power necessary to carry out my planning and mapping algorithms. My laptop wouldn't work either, as the cables got twisted up when the robot was spinning, as well as the fact it was quite bulky. Eventually I decided to purchase a Raspberry Pi to solve this issue, which I remotely controlled from my laptop.

Another issue was the limited field of vision of the LIDAR sensor. Unlike other autonomous vehicles, my robot's field of vision was restricted to a single point in front of the robot. In order to create an accurate estimation of its surroundings, I needed to spin the robot 360 degrees. Furthermore, all obstacles needed to be at a certain height, and the obstacle needed to be centered in the robot's field of vision. As a result, I needed to develop ways to work around this limitation, such as adding plenty of buffer around the obstacles.

If an obstacle was off to the side of the Roomba, the LIDAR sensor might not detect it, and a collision could occur.



Reflection:

If I were to continue to improve on this project, I would definitely attempt to increase the field of vision of the robot, perhaps by adding more sensors. I might also find not build the buffer around walls into the planning, because it occasionally results in wonky paths. I would explore PID as well, and have the robot travel in a smooth path, rather than straight lines. Nevertheless, I'm proud of what I have managed to create, and through the process of building, programming, and testing my own robot I have learned much about the challenges that autonomous navigation presents.

To make your own self-driving Roomba you will need:

- Roomba 570 iRobot series?
- Arduino Uno
- 7-pin serial to USB cable
- Raspberry Pi 2 w/ Internet connection
- Power supply
- **LIDAR-Lite Laser Rangefinder (PulsedLight)**
- 3 - Axis Gyro (MPU 6050)
- Control system (laptop)
- Lots of tape

Challenges:

1. Communication
    a. Connect control system to raspberry pi wirelessly using SSH with Putty
    b. Running Xming on control system to allow for X-11 forwarding in order to allow Python to draw output on control system
    c. Raspberry Pi connected to Roomba and Arduino Uno
        i. Arduino Uno sends gyro and lidar readings
        ii. Communicates w/ Roomba to get encoder values and send signal to the motor
    d. Run python script on Raspberry Pi to execute
        i. Use pyserial module to get and send data to Arduino / Roomba
2. Moving Roomba
    a. Use iRobot's Roomba Open Interface
        i. Send bytes over serial to start, move motors
        ii. Can request and receive sensor info
        iii. Sends high byte, then low byte
3. Getting Data
    a. Soldered wires onto gyro / lidar; connected to digital I/O of arduino
    b. Run short script that gets values, writes them to Pi
    c. Using library for MPU6050
4. Localization
    a. Read encoder vals to see how much changing for each
    b. Initially tried odometry and dead reckoning
        i. http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-186-mobile-autonomous-systems-laboratory-january-iap-2005/study-materials/odomtutorial.pdf
    c. Found it was too inaccurate; switched to gyro
        i. Had to compensate for gyro drift; average readings over time (avg. slope)

    ii. Dist * cos(angle) = dX, avgDist * sin(angle) = y
    iii. Now, just need to ensure angle is correct
5. Sensing
    a. Move until threshold is reached
    b. Take LIDAR readings
    c. Spin in circle to get map of points
    d. Then, attempt to combine points to form lines
    e. Extend lines based on robot width & buffer

6. Planning
    a. Use ASTAR
        i. Nodes are endpoints of line segments
        ii. Check to see if path intersects, if they do, circumvent node, add new point
        iii. Try to create rounded point based on intersection of circles
7. Executing plan
    a. Turn to make roomba face correct angle to get from curr pos to next point
    b. Once close enough to point, moves on to next point
    c. Follow path until sensor is tripped
        i. then replan
8. Basic utils
    a. Finding angle difference: basically, if diff > 180, try going the other way
    b. Avg angle?
    c. Counter

Next Time:

- Plan while moving
- Different state space
- PID, curved path, smoothing rather than