CTR D-STOP

Technical Report 105

Semi-Autonomous Parking for Enhanced Safety and Efficiency

Sriram Vishwanath
WNCG

June 2017

Data-Supported Transportation Operations & Planning Center (D-STOP)

A Tier 1 USDOT University Transportation Center at The University of Texas at Austin

CENTER FOR TRANSPORTATION RESEARCH

Wireless Networking & Communications Group

D-STOP is a collaborative initiative by researchers at the Center for Transportation Research and the Wireless Networking and Communications Group at The University of Texas at Austin.

Technical Report Documentation Page

1. Report No. D-STOP/2017/105	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Self-Parking for Semi-Autonomous Vehicles		5. Report Date June 2017 6. Performing Organization Code
7. Author(s) Sriram Vishwanath		8. Performing Organization Report No. Report 105
9. Performing Organization Name and Address Data-Supported Transportation Operations & Planning Center (D-STOP) The University of Texas at Austin 1616 Guadalupe Street, Suite 4.202 Austin, Texas 78701		10. Work Unit No. (TRAIS) 11. Contract or Grant No. DTRT13-G-UTC58
12. Sponsoring Agency Name and Address Data-Supported Transportation Operations & Planning Center (D-STOP) The University of Texas at Austin 1616 Guadalupe Street, Suite 4.202 Austin, Texas 78701		13. Type of Report and Period Covered 14. Sponsoring Agency Code
15. Supplementary Notes Supported by a grant from the U.S. Department of Transportation, University Transportation Centers Program. Project title: Semi-Autonomous Parking for Enhanced Safety and Efficiency		
16. Abstract This project focuses on the use of tools from a combination of computer vision and localization based navigation schemes to aid the process of efficient and safe parking of vehicles in high density parking spaces. The principles of collision avoidance, simultaneous localization and mapping together with vision based actuation in robotics will be used to enable this functionality.		
17. Key Words	18. Distribution Statement No restrictions. This document is available to the public through NTIS (http://www.ntis.gov): National Technical Information Service 5285 Port Royal Road Springfield, Virginia 22161	
19. Security Classif.(of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages / 22. Price

Form DOT F 1700.7 (8-72) **Reproduction of completed page authorized**

Disclaimer

Acknowledgements

The authors recognize that support for this research was provided by a grant from the U.S. Department of Transportation, University Transportation Centers.

Self-Parking for Semi-Autonomous Vehicles

Objective

The objective of the self-parking project was to provide a testbed on which individual self-parking techniques and algorithms for coordination could be developed and tested. The overall project consisted of three main stages:

1. Self-parking capabilities of Proteus robots
2. Communication framework between robots for algorithm execution
3. Algorithm deployment for multiple robots in simulated parking lot environment

Background: Pharos Robotic Platform

The Pharos testbed is composed of robots called Proteus III. Proteus III is a robot designed for research that brings vehicular mobility/control and communications together. What sets Proteus apart from most other robotic research platforms is its modular architecture and development-friendly design. Proteus provides a powerful base platform while allowing for the essential customization and expansion of the robot for whatever specific research application it must serve. In order to maintain simplicity and intuition with such diverse and dynamic capabilities, great care has been taken in designing the robot.

A key feature of the Pharos testbed is the mobility capabilities of the Proteus III robots. Mobility is exceedingly hard to model accurately, and thus testbeds are invaluable in mobile network communications research. Theoretical bounds and numerical simulations can provide insight into real-world behavior, but each of these has their respective weaknesses. Particularly, when communication and vehicular behavior become dependent on one another, it is near-impossible to model every aspect of this interaction. Thus, a testbed is needed, both as a mechanism to test new techniques and to provide feedback to the design process of the impact of mobility on communication performance metrics. Pharos has previously been used in several communication network scenarios including network coding, delay tolerant networks, multi-robot patrol, and autonomous intersections.

Proteus Control Interface
The Proteus III robots have a computational plane that consists of an x86 Computer (Via EPIA N800-10E) with a WiFi Network interface. The computational plane runs ROS (Robot Operating System) which provides a platform for control of each individual robot and coordination between multiple robots. We briefly highlight the capabilities of the Proteus Control Interface relevant to this work.

Mobility: The Proteus robots have mobility capabilities via a Traxxas Stampede chassis controlled by an Arduino micro-controller. The Traxxas chassis gives the Proteus robot robustness in outdoor environments and the motor provides a range of speeds for different mobility scenarios.

Outdoor Navigation: One of the modalities of the Proteus robots allows us to attach a GPS and compass module that provide the ability to navigate to predetermined GPS locations. This capability provides the opportunity to create interesting mobility scenarios with reliable repeatability and also generate a wide range of spatial topologies.

Coordination: The ROS framework running on the robots provides the infrastructure for wireless communication using the publication and subscription of network-wide messages. Using the WiFi interface, the robots in a connected network can exchange information about their current states and trigger actions on other robots that are part of the same ROS core.

Self-Parking Capabilities of Proteus Robots

We worked on developing and adding a number of capabilities to the Proteus robots to facilitate two components of self-parking: 1) getting *to* the spot and 2) getting *into* the spot. The self-parking capabilities of the Proteus robots were all developed as nodes on the ROS framework.

In order to get into the spot we worked on two different approaches

1. A visual approach using a webcam

2. A range approach using ultrasound range finders

Visual Approach

The visual approach uses a USB webcam to scan for a parking space. Once a parking lot is detected, a path finding algorithm is used to select an efficient path to navigate into a spot. The path finding algorithm developed within the lab is described below:

The core of the pathfinder algorithm is A* modified to use a 3D map based on the position and rotation of the vehicle combined with a window of valid transitions between coordinates in the map.

The parameters for this algorithm include:

1 Resolution of the vehicles rotation.

2. Size of a single pixel in the specified map.

3. Turning radius of the vehicle.

4. Array containing dimensions of the vehicle. There is nothing limiting the vehicle to a rectangular shape other than how this parameter is defined.

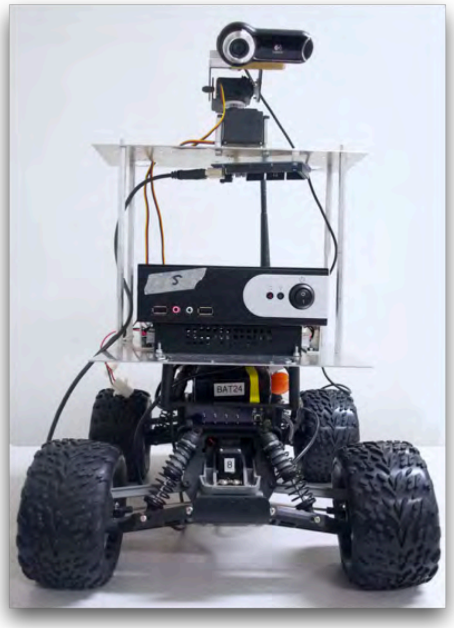


Figure 1

5. Size of window that is used to check valid coordinate transitions.

A major problem within pathfinding is boundary checking. To solve the problem of boundary checking, a slight modification was made to the traditional representation of coordinates. An additional bit was added in the highest position of the x and y values that remains zero, along with the constraint that both dimensions are bound to powers of 2. At first this seems like a significant waste of space, however, images were already being padded to fit to power of 2 boundaries to speed up the Fourier transforms during map generation, and Linux may lazily allocate pages, preventing the wasted bits from using any memory at all.

Boundary checking can then be performed trivially. When aligned to a power of 2, going out of bounds will cause the relevant value to overflow and set the extra bit to a 1. This allows all four map boundaries to be checked with a single bitmask. It should be noted that the overflow may corrupt other fields in the coordinate, but because the coordinate now represents an out of bounds value, it should already be considered invalid.

Creating a heuristic that does not require extraction of fields required a more complex arithmetic solution that takes advantage of the integer representation, overflow bits, and an additional constraint that both dimensions are equivalent.

This algorithmic framework was developed by Chris Haster, a student in the Lab. Overall, the appropriate path is then executed by publishing the appropriate angle and speed commands while monitoring distance to the desired spot.

Further Details of the Visual Approach:

The visual approach uses a camera to scan for a parking space. Once a parking lot is detected, a path finding algorithm is used to select an efficient path to navigate into a spot. Finally, the pathfinding algorithm is executed by publishing the appropriate angle and speed commands while monitoring distance.

Searching for a spot: The first step using the visual approach is to look for a parking space. In our case, we used a colored perimeter and a marker on the floor to denote a parking space. Next, the robot finds a path using the current location and the target location. We use a variation of the A* algorithm that is able to find paths in a couple of seconds. Finally, after a path to follow has been found, the node sends information to the semi-autonomous vehicle to execute the desired movements.

NOTE: Further details of the visual approach can be found in the presentation attached.

Figure 1 shows the configuration of the Proteus robot when using the Visual Approach.

Range Approach

The Range Approach uses ultrasound range finders to address situations where more precision is required. In particular it prevents physical collisions with other objects around the desired spot. The module developed for the range approach identifies obstacles, edges, and empty spaces that are large enough to park using the ultrasound range finder. Once space is identified, the robot turns into the space at low speed while monitoring distance to obstacles around it. The robot situates itself equidistant from obstacle at either side and stops 30 cm away from front obstacle. The algorithmic framework here identifies obstacles, edges, and empty spaces that are large enough to park using the ultrasound range finder Once space is identified, the robot turns into the space at low speed while monitoring distance to obstacles around it. Overall, the robot situates itself equidistant from obstacle at either side and stops 30 cm away from front obstacle.

The first step in the range approach is to find edges and the shape of the space In this beginning phase, the robot is assumed to be perpendicular to the parking spot, and so it must turn into the spot. Once the robot has entered the spot, it slowly parks itself while avoiding collisions with neighboring obstacles
The robot parks by aligning itself equidistant to both sides, straightening, and stopping 30 cm from the front obstacle.

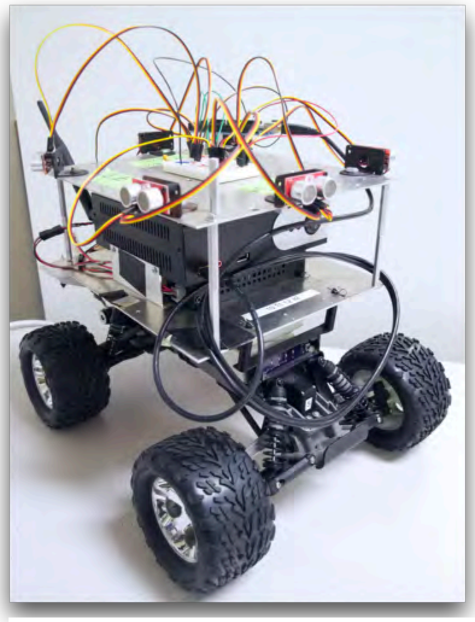


Figure 2

NOTE: More details of the range approach can be found in the presentation attached.

Figure 2 shows the configuration of the Proteus robot when using the Range Approach.

GPS/Compass Navigation

Both the Visual and Range Approaches described above concentrated in getting the robot *into* the parking spot. In order to get *to* the parking spot, we equipped the Proteus robots with GPS capabilities and a Compass module to determine its directions. These two ROS nodes, along with a third navigation node, allowed robots to know their location in outdoor environments and navigate to a specific set of GPS coordinates. The GPS node received updated GPS location every second and had capabilities to publish its current location for use by other nodes. The compass nodes constantly updated the nodes direction. The navigation nodes used both the GPS and Compass nodes to create feedback loop of correction in order to guide the robot to a desired location. We tested the GPS/Compass navigation system in the top floor of UTA by directing the robots to different GPS coordinates corresponding to different parking spots.

The code for these three ROS nodes can be found in:

- GPS: https://github.com/pesantacruz/utexas-ros-pkg/tree/experimental/stacks/pharos/proteus3_gps_hydro
- Compass: https://github.com/pesantacruz/utexas-ros-pkg/tree/experimental/stacks/pharos/proteus3_compass_hydro
- Navigation: https://github.com/pesantacruz/utexas-ros-pkg/tree/experimental/stacks/pharos/proteus3_navigation

Communication Framework and Algorithm Deployment

The self-parking project concluded on the testing phase of the self-parking capabilities. We were able to create communication between different robots by using the Publish/Subscribe framework in ROS. In other works, when connected in a WiFi ad hoc network, each robot had access to all published messages, by the robot itself or by another robot. In particular in our case, we were able to publish the GPS location and the Compass direction of each robots and that information was available to all other robots in the network. See the messages:

- ***CompassMsg.msg*** defined in https://github.com/pesantacruz/utexas-ros-pkg/blob/experimental/stacks/pharos/proteus3_compass_hydro/msg/CompassMsg.msg
- ***GPSMsg.msg*** defined in https://github.com/pesantacruz/utexas-ros-pkg/blob/experimental/stacks/pharos/proteus3_gps_hydro/msg/GPSMsg.msg

Self-Parking Presentation

Self-parking Project

Pharos Lab

01/16/2015

Project overview

- Large amounts of space is underutilized in parking lots due to the need for navigation space
- Can we develop algorithms with expected arrival and departure information to minimize the amount of underutilized space?
- How does the problem change if we add the extra constraint of putting electric cars in a restricted number of charging parking spaces?

Pharos Lab's Role

- Provide testbed support for implementation of developed algorithms

- Stages of involvement and requirements:

  1. Self-parking capabilities for Proteus robots

  2. Communication framework between robots for algorithm execution

  3. Algorithm deployment on network of robots in simulated parking lot environment

Pharos Lab's Role

- Provide testbed support for implementation of developed algorithms

- Stages of involvement and requirements:

1. **Self-parking capabilities for Proteus robots**

2. Communication framework between robots for algorithm execution

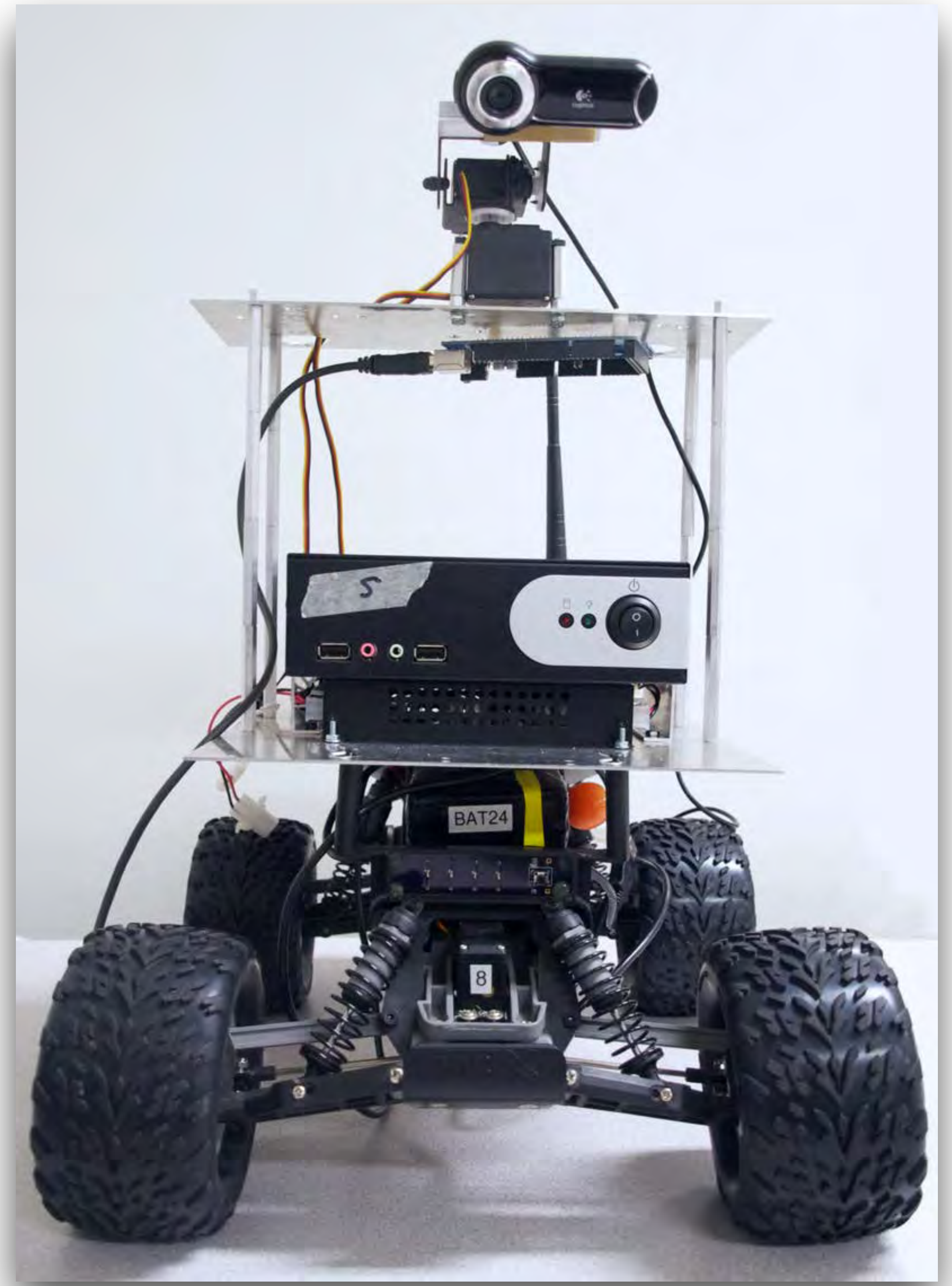
3. Algorithm deployment on network of robots in simulated parking lot environment

Self-parking

- We are taking 2 different approaches
  - Visual approach using webcam
  - Rage approach using ultrasound range finders

Visual approach

- The visual approach uses a USB webcam to scan for a parking space
- Once a parking lot is detected, a path finding algorithm is used to select an efficient path to navigate into a spot
- Path is executed by publishing the appropriate angle and speed commands while monitoring distance



Visual approach

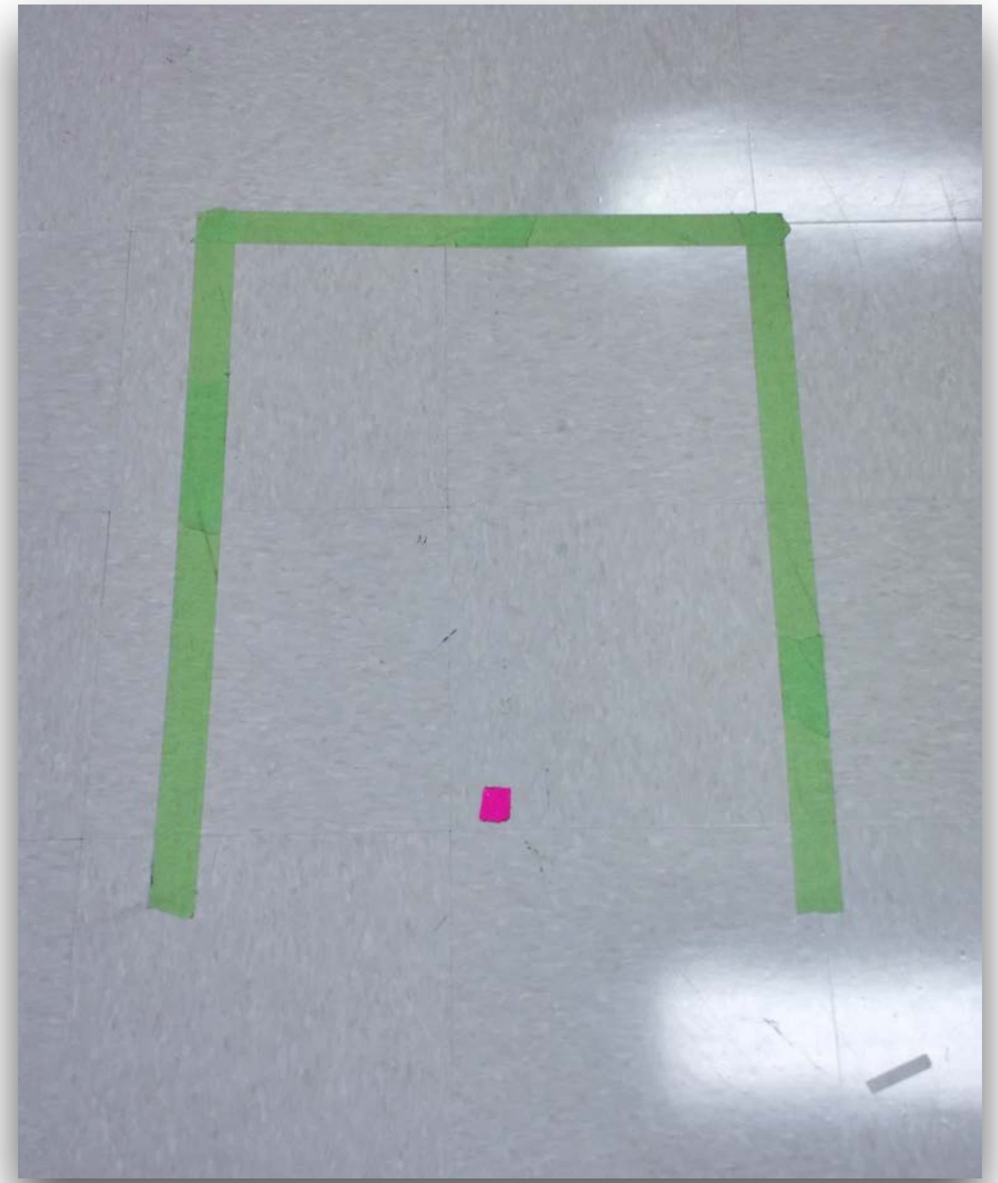
- The visual approach uses a USB webcam to scan for a parking space
- Once a parking lot is detected, a path finding algorithm is used to select an efficient path to navigate into a spot
- Path is executed by publishing the appropriate angle and speed commands while monitoring distance

Main Sensor - Camera

Logitech Webcam Pro 9000 720p

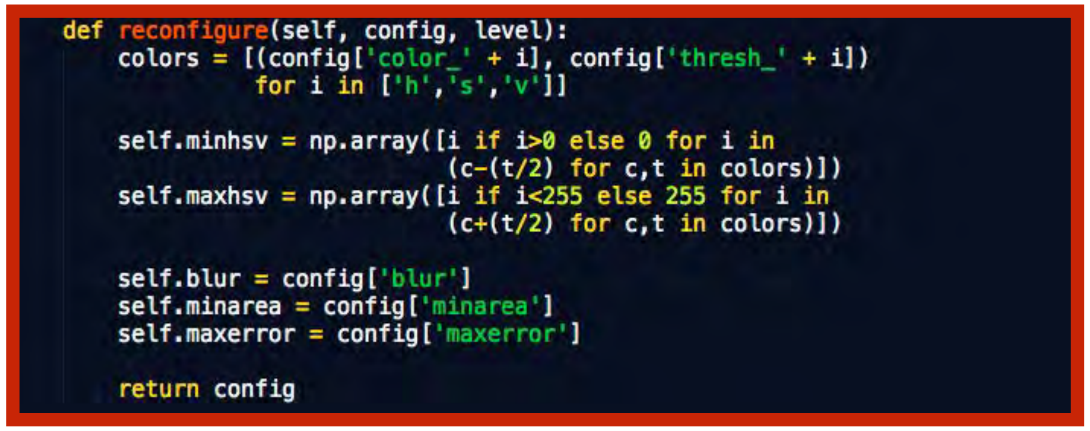
Visual Approach

- Searching for a spot
  - The first step using the visual approach is to look for a parking space
    - In our case, we used a green perimeter and a pink marker on the floor to denote a parking space



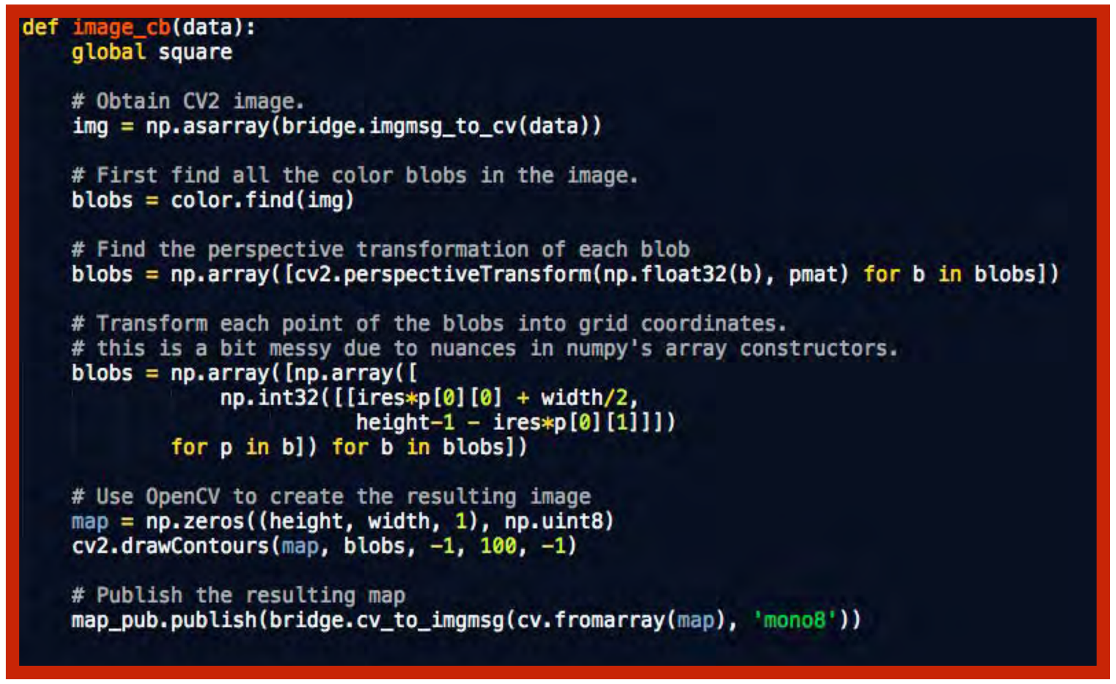
Visual Approach

- Searching for a spot
  - The first step using the visual approach is to look for a parking space
    - In our case, we used a green perimeter and a pink marker on the floor to denote a parking space



```python
def reconfigure(self, config, level):
    colors = [(config['color_' + i], config['thresh_' + i])
              for i in ['h','s','v']]

    self.minhsv = np.array([i if i>0 else 0 for i in
                              (c-(t/2) for c,t in colors)])
    self.maxhsv = np.array([i if i<255 else 255 for i in
                              (c+(t/2) for c,t in colors)])

    self.blur = config['blur']
    self.minarea = config['minarea']
    self.maxerror = config['maxerror']

    return config
```

Identify colors to find

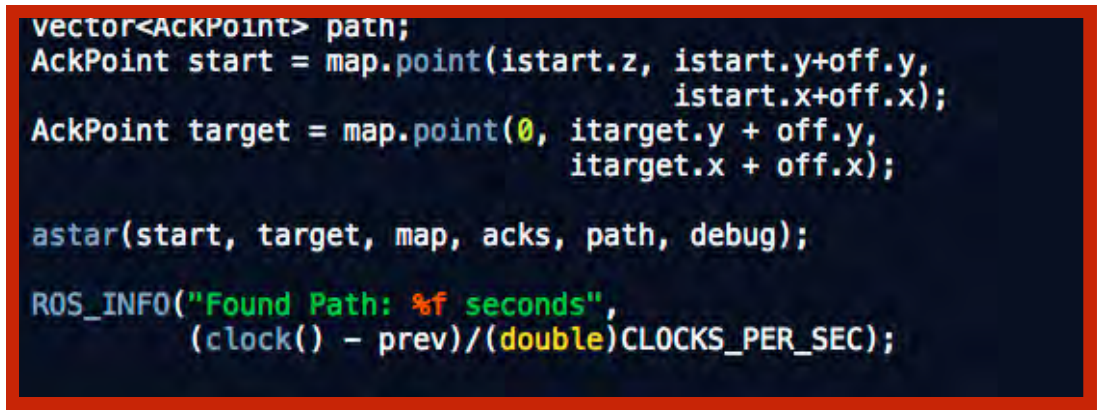


```python
def image_cb(data):
    global square

    # Obtain CV2 image.
    img = np.asarray(bridge.imgmsg_to_cv(data))

    # First find all the color blobs in the image.
    blobs = color.find(img)

    # Find the perspective transformation of each blob
    blobs = np.array([cv2.perspectiveTransform(np.float32(b), pmat) for b in blobs])

    # Transform each point of the blobs into grid coordinates.
    # this is a bit messy due to nuances in numpy's array constructors.
    blobs = np.array([np.array([
            np.int32([[ires*p[0][0] + width/2,
                       height-1 - ires*p[0][1]]])
        for p in b]) for b in blobs])

    # Use OpenCV to create the resulting image
    map = np.zeros((height, width, 1), np.uint8)
    cv2.drawContours(map, blobs, -1, 100, -1)

    # Publish the resulting map
    map_pub.publish(bridge.cv_to_imgmsg(cv.fromarray(map), 'mono8'))
```

Create map of spot location

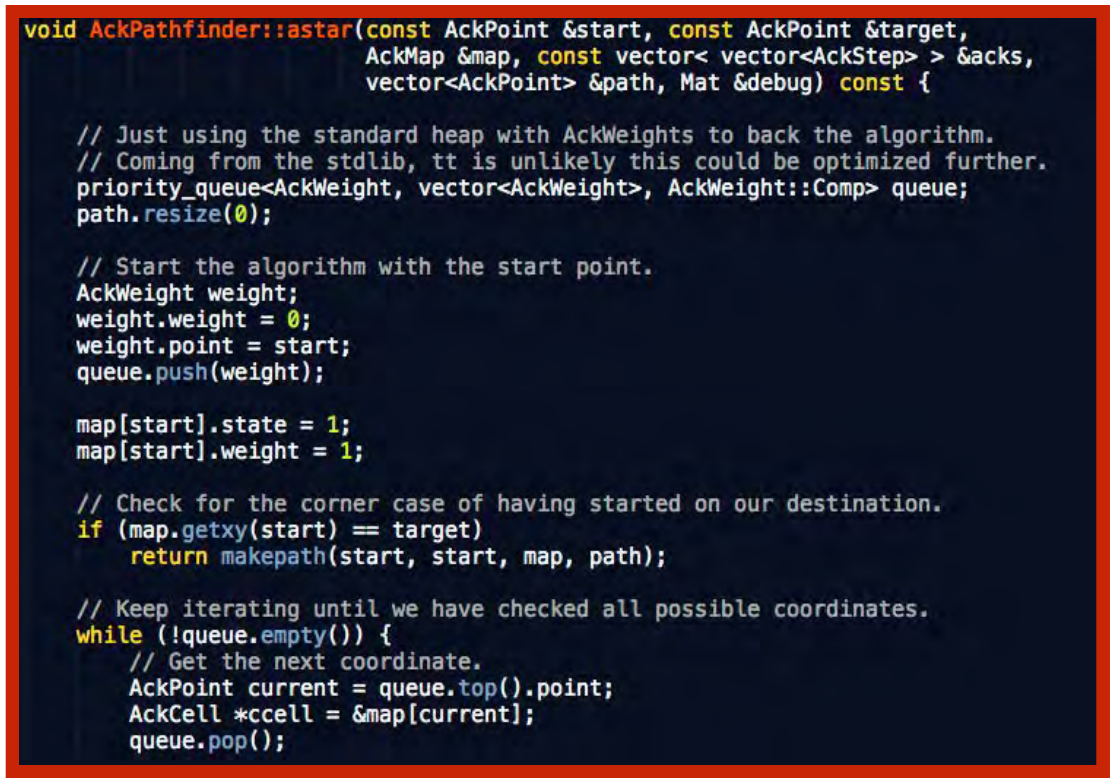
Visual Approach

- Find a path
  - Next, the robot finds a path using the current location and the target location
    - We use a variation of the A* algorithm that is able to find paths in a couple of seconds



```
vector<AckPoint> path;
AckPoint start = map.point(istart.z, istart.y+off.y,
                           istart.x+off.x);
AckPoint target = map.point(0, itarget.y + off.y,
                            itarget.x + off.x);

astar(start, target, map, acks, path, debug);

ROS_INFO("Found Path: %f seconds",
         (clock() - prev)/(double)CLOCKS_PER_SEC);
```

Use modified A* algorithm to find a path

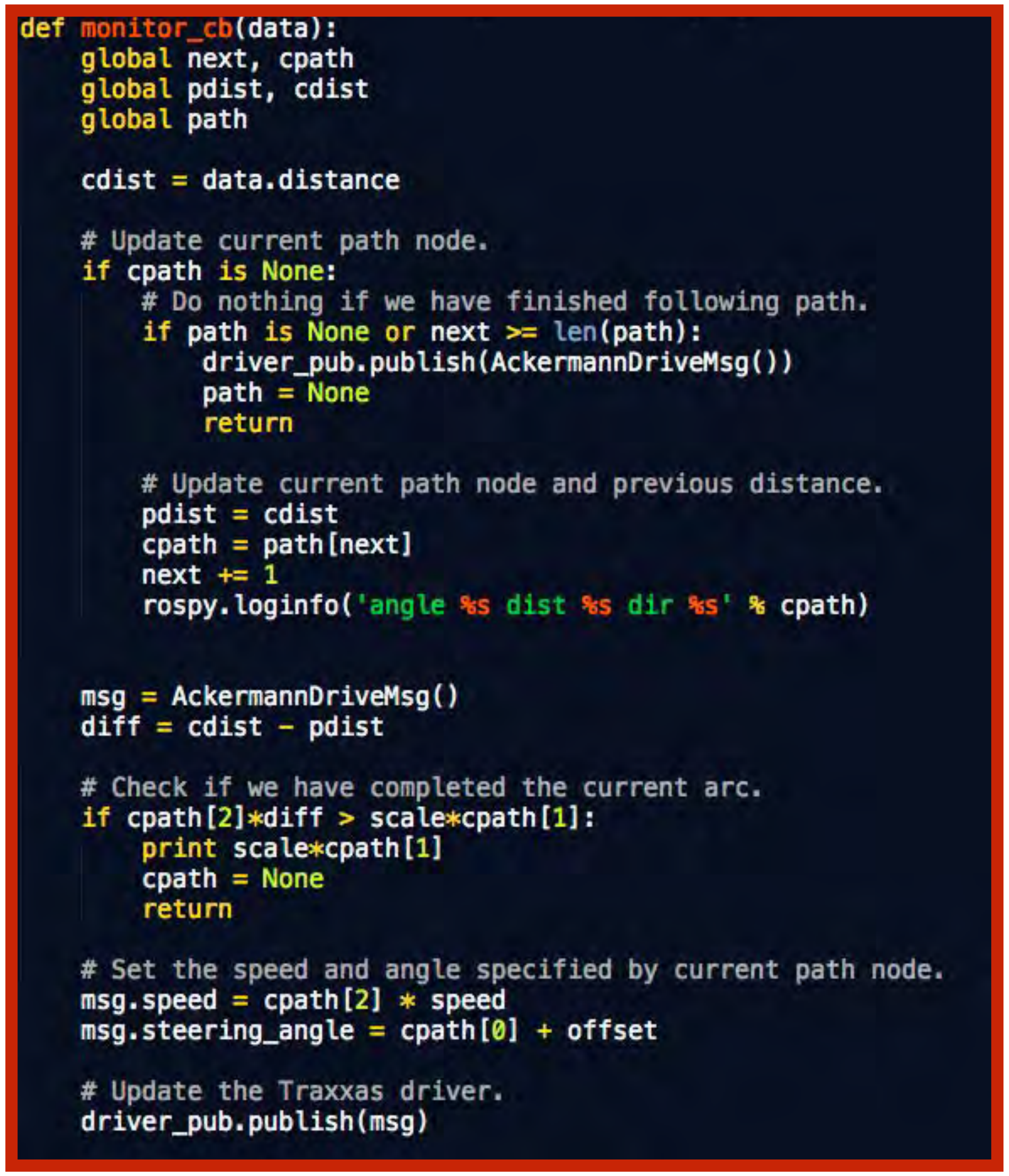


```
void AckPathfinder::astar(const AckPoint &start, const AckPoint &target,
                          AckMap &map, const vector< vector<AckStep> > &acks,
                          vector<AckPoint> &path, Mat &debug) const {

    // Just using the standard heap with AckWeights to back the algorithm.
    // Coming from the stdlib, tt is unlikely this could be optimized further.
    priority_queue<AckWeight, vector<AckWeight>, AckWeight::Comp> queue;
    path.resize(0);

    // Start the algorithm with the start point.
    AckWeight weight;
    weight.weight = 0;
    weight.point = start;
    queue.push(weight);

    map[start].state = 1;
    map[start].weight = 1;

    // Check for the corner case of having started on our destination.
    if (map.getxy(start) == target)
        return makepath(start, start, map, path);

    // Keep iterating until we have checked all possible coordinates.
    while (!queue.empty()) {
        // Get the next coordinate.
        AckPoint current = queue.top().point;
        AckCell *ccell = &map[current];
        queue.pop();
```

A* modification

Visual Approach

- Execute path
    - Finally, after a path to follow has been found, the node sends information to the Traxxas node to execute the movements
        - The node sends a series of speed and steering commands

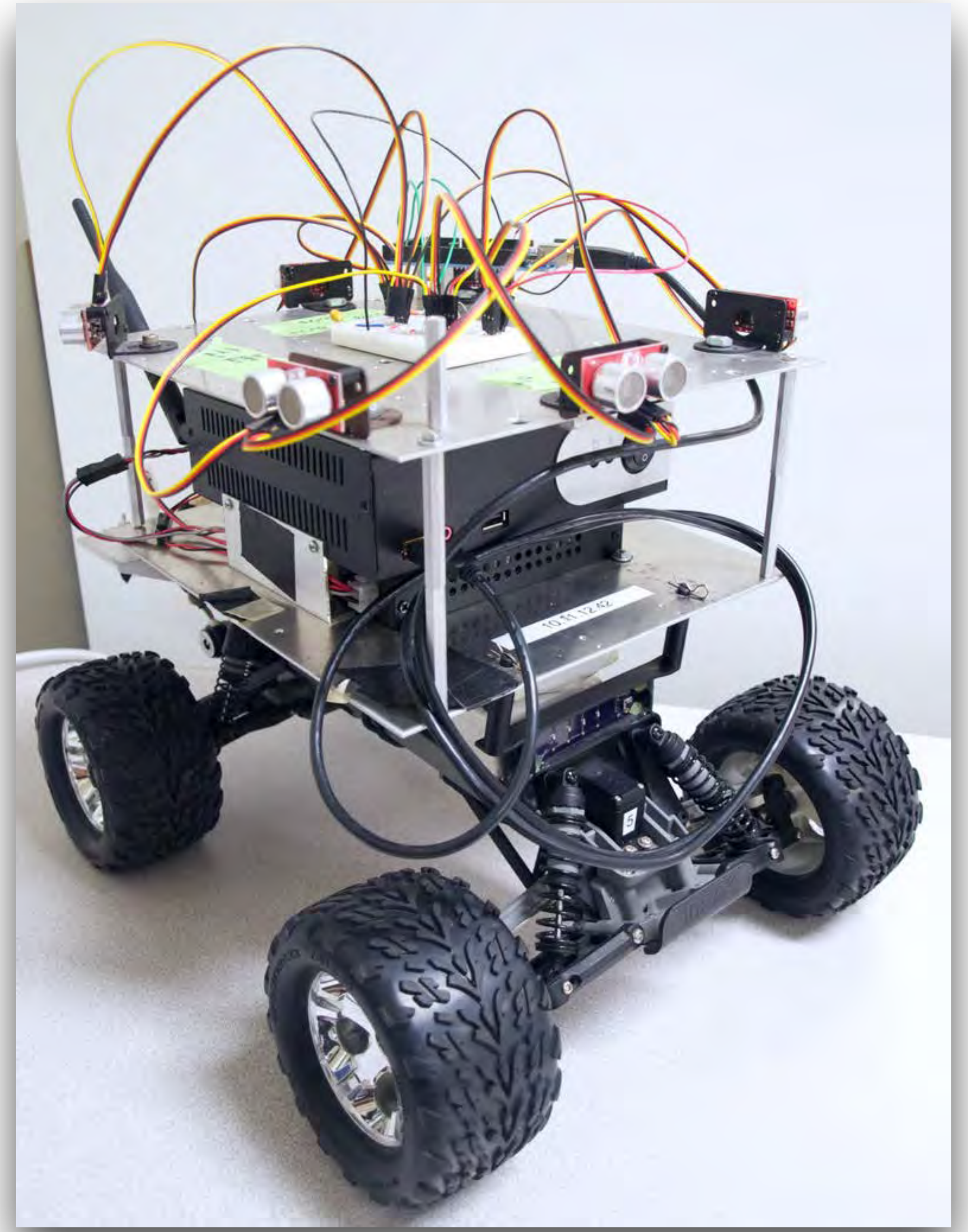


```
def monitor_cb(data):
    global next, cpath
    global pdist, cdist
    global path

    cdist = data.distance

    # Update current path node.
    if cpath is None:
        # Do nothing if we have finished following path.
        if path is None or next >= len(path):
            driver_pub.publish(AckermannDriveMsg())
            path = None
            return

        # Update current path node and previous distance.
        pdist = cdist
        cpath = path[next]
        next += 1
        rospy.loginfo('angle %s dist %s dir %s' % cpath)


    msg = AckermannDriveMsg()
    diff = cdist - pdist

    # Check if we have completed the current arc.
    if cpath[2]*diff > scale*cpath[1]:
        print scale*cpath[1]
        cpath = None
        return

    # Set the speed and angle specified by current path node.
    msg.speed = cpath[2] * speed
    msg.steering_angle = cpath[0] + offset

    # Update the Traxxas driver.
    driver_pub.publish(msg)
```

Publish speed and steering

Range approach

- Identifies obstacles, edges, and empty spaces that are large enough to park using the ultrasound range finder
- Once space is identified, the robot turns into the space at low speed while monitoring distance to obstacles around it
- Robot situates itself equidistant from obstacle at either side and stops 30 cm away from front obstacle



Range approach

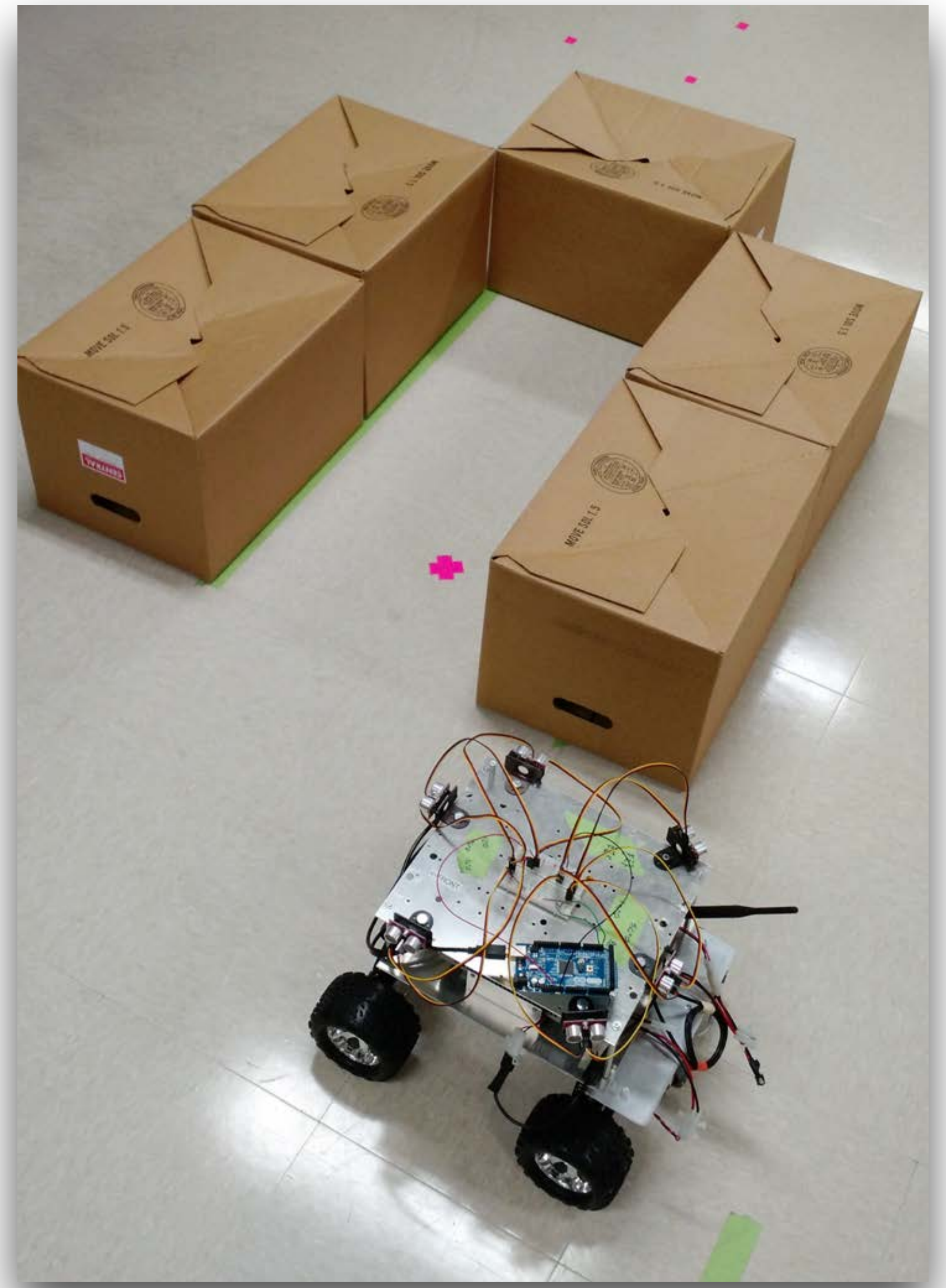
- Identifies obstacles, edges, and empty spaces that are large enough to park using the ultrasound range finder
- Once space is identified, the robot turns into the space at low speed while monitoring distance to obstacles around it
- Robot situates itself equidistant from obstacle at either side and stops 30 cm away from front obstacle

Main Sensor - Ultrasound Range Finder

Devantech Ultrasound Range Finder SRF-08

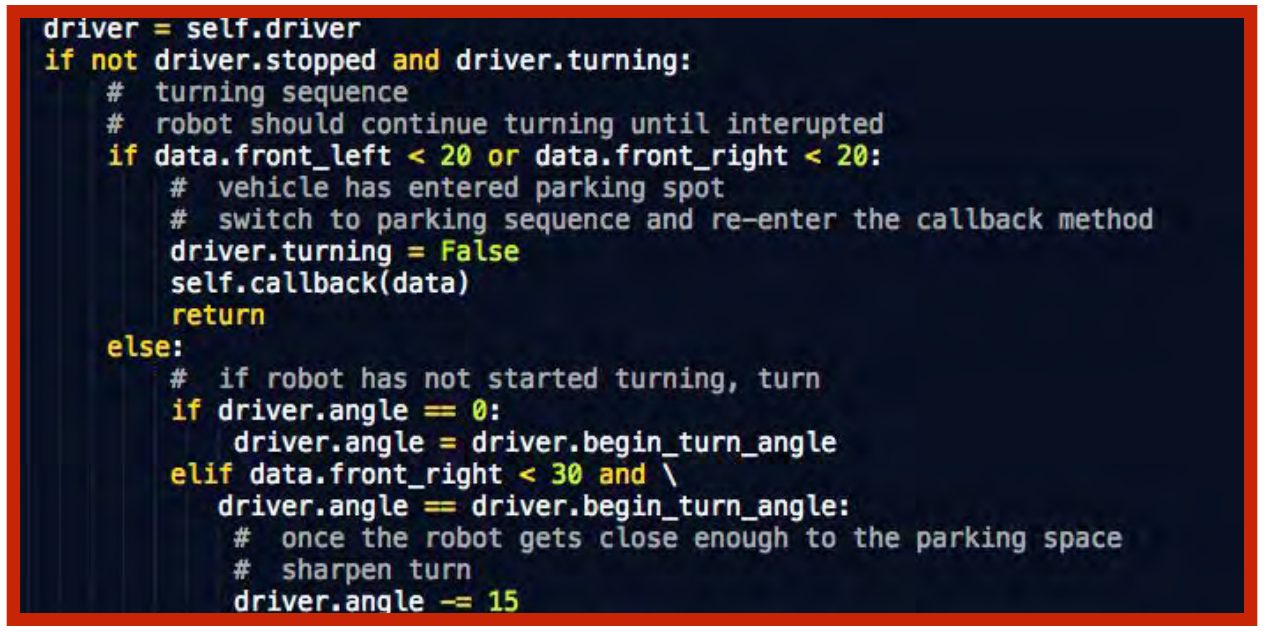
Range approach

- The first step in the range approach is to find edges and the shape of the space
    - In this beginning phase, the robot is assumed to be perpendicular to the parking spot, and so it must turn into the spot



Range approach

- The first step in the range approach is to find edges and the shape of the space
    - In this beginning phase, the robot is assumed to be perpendicular to the parking spot, and so it must turn into the spot

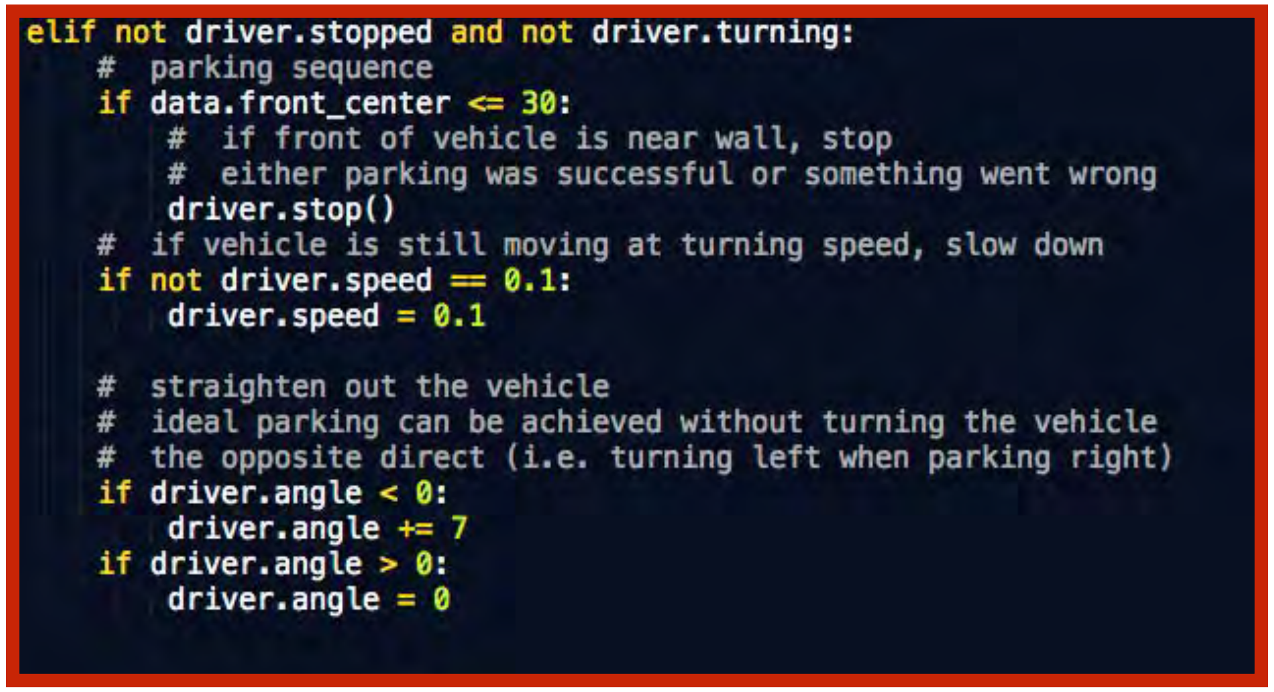


```
driver = self.driver
if not driver.stopped and driver.turning:
    #  turning sequence
    #  robot should continue turning until interupted
    if data.front_left < 20 or data.front_right < 20:
        #  vehicle has entered parking spot
        #  switch to parking sequence and re-enter the callback method
        driver.turning = False
        self.callback(data)
        return
    else:
        #  if robot has not started turning, turn
        if driver.angle == 0:
            driver.angle = driver.begin_turn_angle
        elif data.front_right < 30 and \
           driver.angle == driver.begin_turn_angle:
            #  once the robot gets close enough to the parking space
            #  sharpen turn
            driver.angle -= 15
```

Robot turning into a parking spot

Range approach

- Once the robot has entered the spot, it slowly parks itself while avoiding collisions with neighboring obstacles
    - The robot parks by aligning itself equidistant to both sides, straightening, and stopping 30 cm from the front obstacle

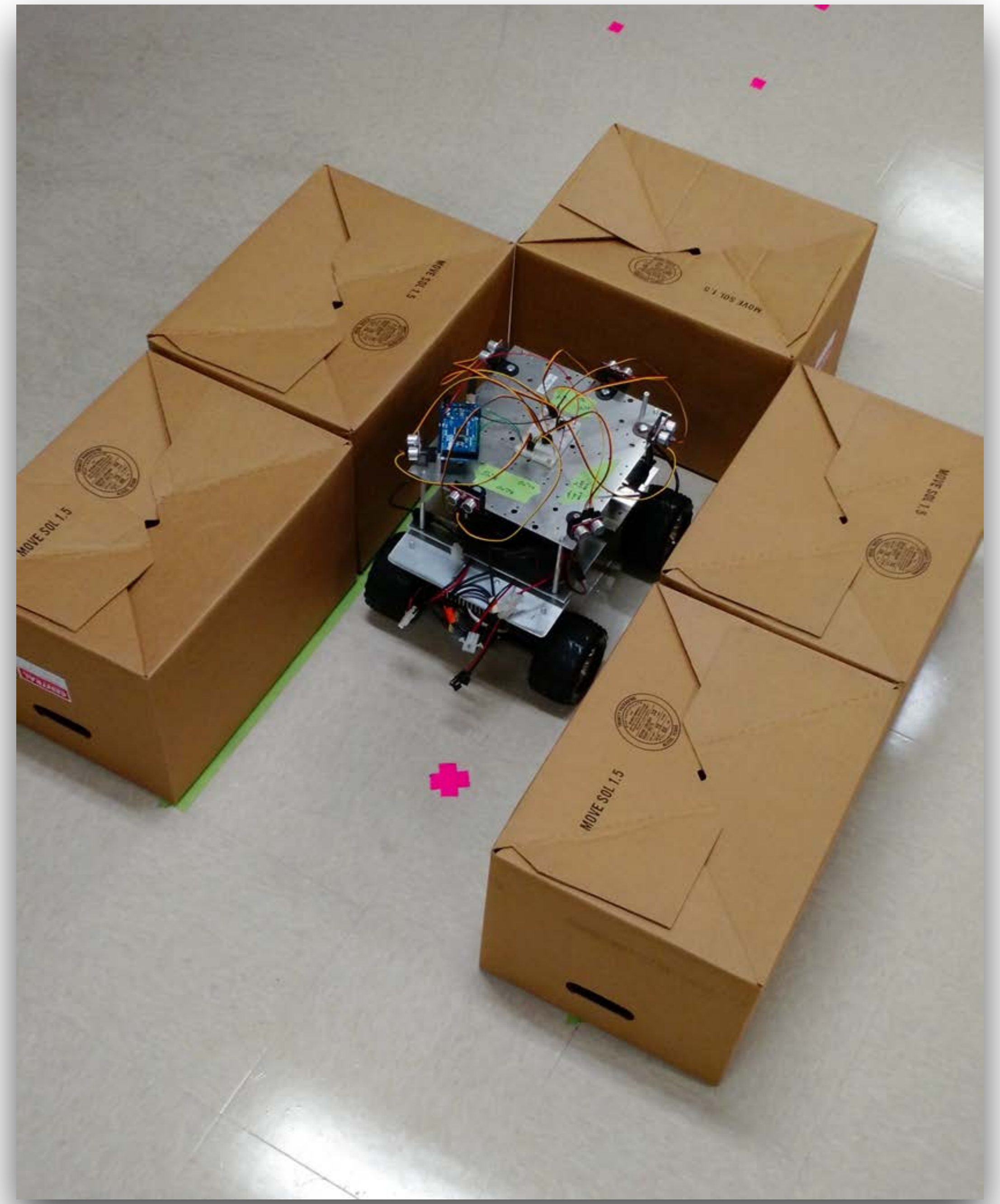


```
elif not driver.stopped and not driver.turning:
    #  parking sequence
    if data.front_center <= 30:
        #  if front of vehicle is near wall, stop
        #  either parking was successful or something went wrong
        driver.stop()
    #  if vehicle is still moving at turning speed, slow down
    if not driver.speed == 0.1:
        driver.speed = 0.1

    #  straighten out the vehicle
    #  ideal parking can be achieved without turning the vehicle
    #  the opposite direct (i.e. turning left when parking right)
    if driver.angle < 0:
        driver.angle += 7
    if driver.angle > 0:
        driver.angle = 0
```

Parking phase of range approach

Range approach

- Once the robot has entered the spot, it slowly parks itself while avoiding collisions with neighboring obstacles
  - The robot parks by aligning itself equidistant to both sides, straightening, and stopping 30 cm from the front obstacle



Next Steps

- We want to combine the strengths of each of the approaches
    - Visual approach gives us more flexibility of starting point, but because of computational complexity, it is more costly to update often
    - Range approach presents more constraints in terms of starting conditions, but once the conditions are satisfied, it can update its current status in real time making it more reliable
- We will use visual approach to *find* and *get to* a parking spot, while the range approach will be used to *enter into* the spot

Next Steps

- Continue on the next steps of the overall project
- Stages of involvement and requirements:

★ 1. Self-parking capabilities for Proteus robots

Next 2. Communication framework between robots for algorithm execution

3. Algorithm deployment on network of robots in simulated parking lot environment