S-PAVe: Self-Parking Autonomous Vehicle

Irving Lin, Jonathan Fong, Timothy Liu, Tyler Latzke, Boyang Zhang | Wenchao Li, Professor Sanjit Seshia
EECS 149 Embedded Systems, University of California Berkeley

Abstract

Parking is a problem that plagues millions of drivers and has, until recently, been quite ignored by the auto industry, and the ones that do exist are severely limited. Therefore, we propose a Self-Parking Autonomous Vehicle (S-PAVe), a proof-of-concept proportionally-scaled prototype vehicle built using a Lego Mindstorms environment and NXT framework, to encourage and further the goal of fully autonomous parking regardless of parking lot structure (parallel or head-in) and situation (surface conditions, slope). S-PAVe's goal is to model real world dynamics of vehicular parking as accurately as possible within the confines of the hardware and software limitations of Lego Mindstorms and NXT while retaining the potential for technological advancement, scalability, and deployment of technology. As such, S-PAVe contains four primary components integrated neatly into package: a vehicle chassis, four sensors (two ultrasonic and two bump), two motors, and a goal-based reflex algorithm. This packaging is intentionally modular to allow for the possibility of future modifications, additions, and development.

Introduction

A parking lot is an invitation for trouble. Reckless drivers, blind spots, inanimate objects, moving hazards, oblivious children – you name it, a parking lot contains it. Compound that to awkward angles and oft-unintuitive (to many drivers) backwards driving and turning, it's no surprise that these dangers are even troublesome for good drivers. It's only now that organizations have started taking notice of the risks associated with these relatively low speed environments. According to a recent 2009 study, the National Highway Traffic Safety Administration (NHTSA) estimates 35% of all fatal 'Nontraffic' accidents occur in parking lots with around 400 annual deaths. In fact, roughly 40% of all minor accidents occur during parking maneuvers. But, according to national insurance claims, 70% of these minor accidents could have been avoided with the aid of new in-car technologies. And while certain driver-assisting technologies like ABS and traction control have been around for a while now, many In-car technologies that mitigate many avoidable accidents are relatively new to the industry. In particular, one of the more recent ones is the introduction of sensor technology to vehicles as early detection warnings. This sensor system can be extended to autonomous parking systems, which we'll see is just now started to be introduced into some vehicle systems.

Comparison

Vehicular technology has greatly improved, but assisted parking has only recently been introduced. However, there is still much room for improvement, as these latest technologies are still missing numerous features – for instance, neither unassisted parking or situation-independent algorithms has yet to be attained. Most current technologies still require human input and are hardcoded to certain conditions. S-PAVe will automate the entire parking process by itself – without needing manual control of the gas and brake pedals, and will be able to tackle virtually any parking spot presented to it, parallel. However, certain parts of S-PAVe will still need to be partially calibrated to the particular car.

Algorithms and Formal Models Used

Overview: Goal Based Reflex Agent

The design for a Self-Parking Autonomous Vehicle (S-PAVe) took into account several considerations.

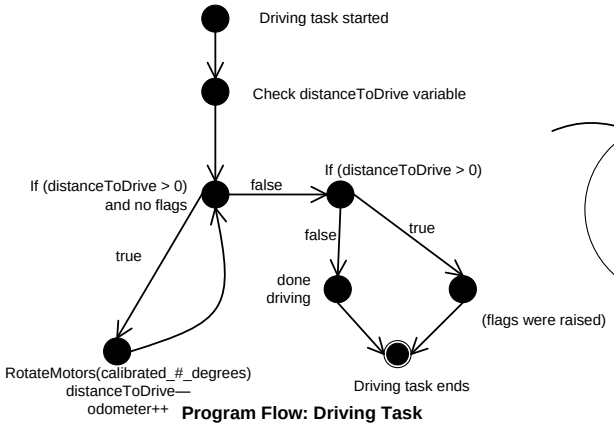
- Model real-time real-world dynamics (friction, velocity, turning radius, etc.)
- Function as an embedded system (use preceptors and actuators)
- Perform concurrent actions robustly (move, sense, think, and speak at the same time)
- Autonomously reach a goal state (correctly park the car in a valid parking spot)

Each of these concerns was successfully modeled through finite state machines, concurrency models, and the actor model. The actor model helped to translate the inputs and outputs of the sensors and motors into relevant mathematical signals. Specifically, angles and distances could be computed to determine arc length, velocity, and other types of metadata. Concurrency models helped to separate and abstract away functionality between low level motor and sensor control from the high-level parking algorithm. Lastly, finite state machines were heavily used to accurately define the transitions between goal states. S-PAVe can be best described as a goal based reflex agent.

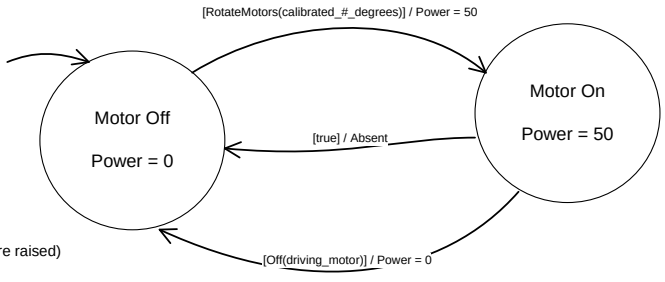
Following real world dynamics, S-PAVe's environment is only partially observable, so the vehicle does not know if a valid parking spot exists. S-PAVe can only rely on the current percepts to determine what transition to make and how to best approach its goals. S-PAVe successfully implements concurrency by abstracting away sensor and motor control into low-level finite state machines. These state machines perform real-time actions while interfacing with embedded preceptors. S-PAVe's highest level FSM contains several states ending with the final goal state PARK. From reachability analysis, the goal state cannot always be achieved due to some nondeterminism, but S-PAVe remains a goal based reflex agent.

Driving Control Algorithm (*see DrivingControl_continuous.nxc*)

The requirements for driving functionality were the ability to: 1) drive specified distances; 2) drive continuously while tracking distance travelled; 3) stop at anytime and keep no state of previous driving commands; 4) drive concurrently to other tasks. Under the restrictions of the NXC development environment (described further in Part 4, below), the driving algorithm developed utilizes a loop to command the motors to continually drive smaller known distances (see below figure). This algorithm is implemented as a task that is protected with helper functions that form a layer of abstraction.



Program Flow: Driving Task



State Machine: Motor for Driving

Sensor Algorithm (*see Sensys.nxc*)

The sensor algorithm is a black box (relative to the main parking algorithm) function that obtains, computes, and then repackages the two ultrasonic and two bump sensor raw output data in a manner that allows easy comprehension. It utilizes four global variables for the four respective sensors, which are constantly updated once the sensor algorithm computes the new raw data. The sensor algorithm is comprised of two methods, SENSYS for ultrasonic and TOUCHSYS for bump, each of which is its own thread.

SENSYS
A while loop that continually polls. SENSYS keeps a buffer of past ultrasonic sensor readings for each of the two sensors. At each poll, update the global variable for the particular ultrasonic sensor with the median of the buffered values.

TOUCHSYS
Continually check to see if there's a true reading on each of the two bump sensors. If there is, set the particular global bump variable to be true, else, to false.

Parking Algorithm (*see main_continous.nxc* or *main_smooth.nxc)*

The Parking algorithm must perform robustly within the model of a real world environment to keep the vehicle and any passengers safe. Therefore, several criteria must remain satisfied throughout the entire parking process.

- Firstly, the parking spot must be deep enough to position the car either for head-in or parallel parking.
- Secondly, the width of the spot must also correspond to the specified depth. A spot that is deep enough to fit the car lengthwise, but is not wide enough should not be considered a spot at all. Similarly, spots that are not very deep must be long enough to fit a car for parallel parking.
- Finally, the spot must be empty.

The Parking algorithm uses a high-level finite state machine which consists of six states that must be traversed for each successful park: RESET, LOOKING, DETECTING, ALIGNMENT, PARK, and STOP.

LOOKING
The vehicle begins in the state LOOKING; however, it is already performing an important action. S-PAVe polls the ultrasonic sensors to determine the initial distance from the right side of the car and sets it as the *InitialBase* which will be used later. At this time, the algorithm has also initiated concurrent tasks: sensors (excluding secondary ultrasonic), driving, and a progress indicator (the Mario theme song). Assuming the vehicle can drive straight (further explanation below), S-PAVe's desired action is to drive straight. S-PAVe will continue to drive straight until a desired transition condition is met. The transition condition is a specific spike in sensor readings or a jump in distance for the depth between the right side of the car and a wall or open space. The vehicle sets a flag *isParallel* based on the sensor reading, sets the distance to *DROP,* and transitions into the state DETECTING.

DETECTING
Inside DETECTING, S-PAVe looks to satisfy two conditions based on the type of spot for parking that *isParallel* denotes. The first condition is that the space maintains a depth as previously indicated by *DROP* or else an obstacle has appeared in the spot. The second condition is that the car has traveled a

specific distance, *distance_remembered,* which it increments after driving at each time step. This ensures that the space is wide enough for the head-in or parallel parking. If either of these conditions is violated, the vehicle will go back into LOOKING. Once the spot has been successfully detected, S-PAVe will now enter into ALIGNMENT.

ALIGNMENT

The ALIGNMENT state is more dichotomous than DETECTING depending on what *isParallel* indicates. For parallel parking, the vehicle pulls forward to align its back wheels with the beginning of the parking spot, which accounts for the arc length computed by the turning radius for an S-turn. S-PAVe then proceeds to the state PARK. For head-in parking, the vehicle needs to align itself to the optimal position given the environment and mechanical specifications. Therefore, the desired position is to reverse back and away from the parking spot, similar to a half S-turn, which is also optimal for the turning radius of the vehicle. S-PAVe then proceeds to the state PARK.

PARK

The state PARK is fairly simple in comparison to the amount of work done before hand. If *isParallel* is set for head-in parking, the vehicle drives forward at a desired angle for the distance of the parking spot's width and depth or more specifically the *distance_remembered* while traveling. Also, the vehicle must account for the *InitialBase* distance that it was initially at. For parallel parking, the car must travel in a similar manner except in reverse. To accomplish an S-turn, the vehicle turns its wheels in one direction and travels half the distance required and turns its wheels in the opposite direction for the rest of the distance. Thus, the vehicle successfully parked inside a spot, achieved its goal state, and transitioned into the state STOP. However after various testing, S-PAVe was not robust enough to model real world conditions where an ideally parked car is both aligned widthwise but also lengthwise. This required that the STOP state, now misnamed, perform parking adjustments or the nicknamed 'wiggle'.

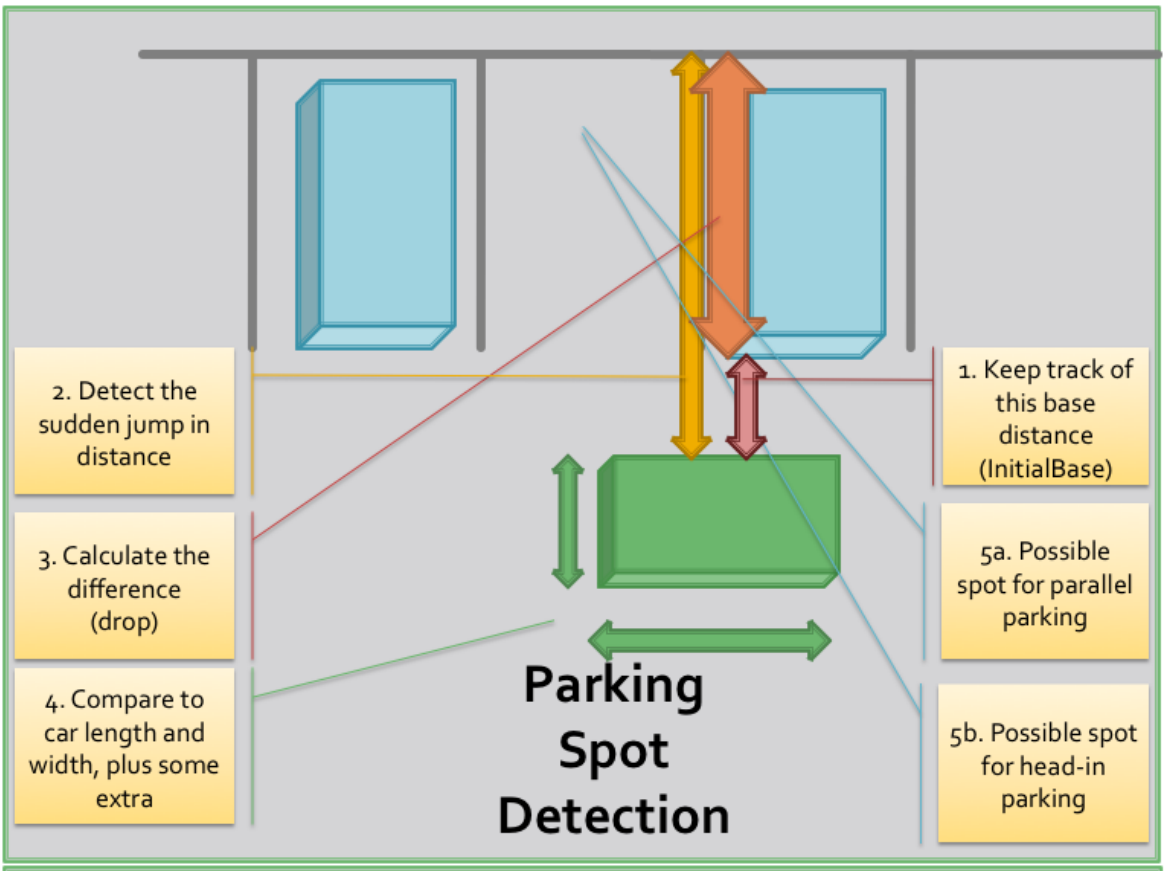
STOP, Wiggle, or Parking Adjustments

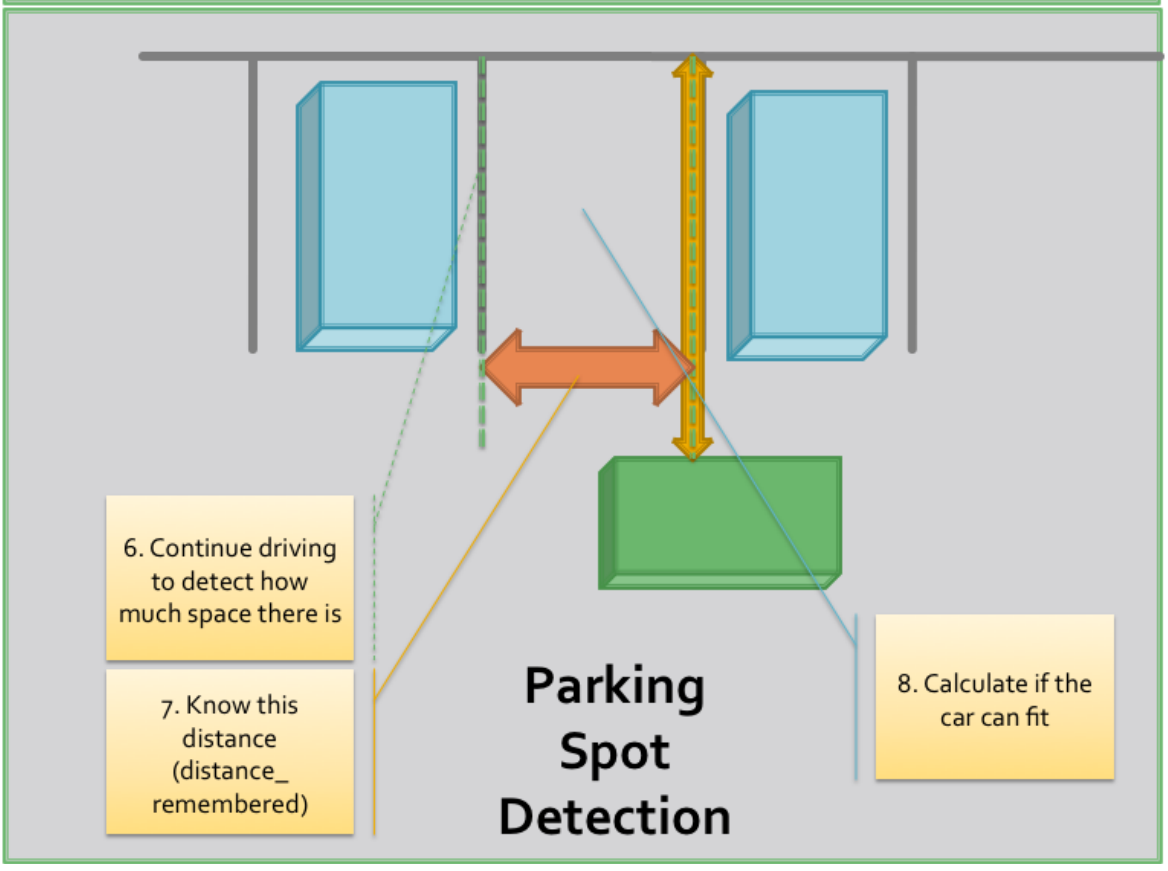
In the STOP state, S-PAVe begins by turning on the second ultrasonic sensor, so as not to interfere with previous readings. Based on both ultrasonic sensor readings, the vehicle can now determine whether or not it is parallel to the surface on the right, as the differential determines the angle of misalignment and the direction to realign. S-PAVe can also now maintain an optimal distance to the curb if parking parallel or to the neighboring cars if parking head-in. Furthermore, the vehicle must also maintain correct distances in the perpendicular axis. Using the two touch sensors, the vehicle will slowly drive or reverse and use information for a parking spot's defined width and depth to optimally wiggle its way into alignment. Thus the car is finished and plays the flag-grabbing theme from Mario and ends in state RESET, so it can be reset.

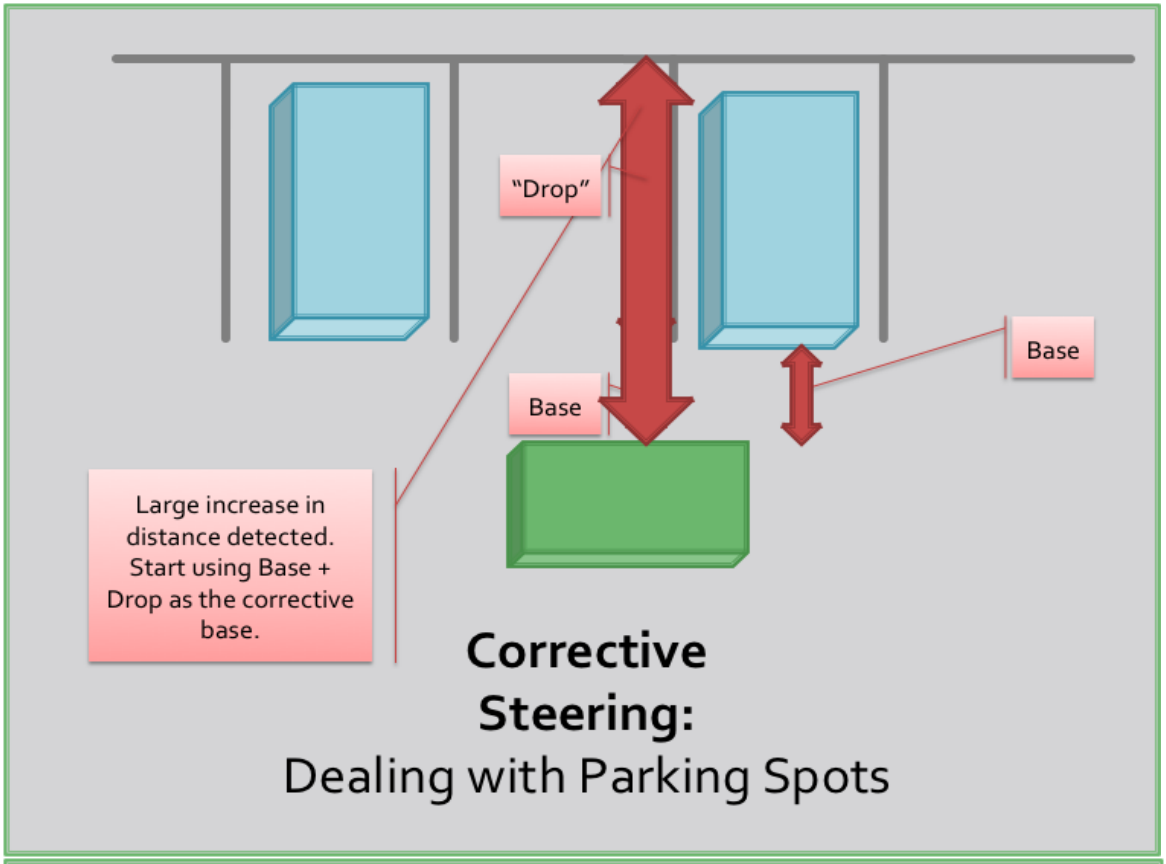
Corrective Steering

As previously mentioned, S-PAVe assumes that it can drive straight; however, this is not the case. The motors allow for drift among the wheels due to imprecise hardware. Therefore, corrective steering is implemented in software to account for mechanical limitations. In almost all states, corrective steering is implemented by noting the *InitialBase* and calibrated *ERROR.* If at any time the ultrasonic sensor reads a value that is beyond *InitialBase* +/- *ERROR*, the vehicle will take the necessary adjustments so that it falls within the *InitialBase +/- ERROR*. A specific corner case must also be taken care of for any drops or spikes in readings, as this will reset the *InitialBase*. Therefore, the vehicle maintains or snaps to the most current parallel surface.

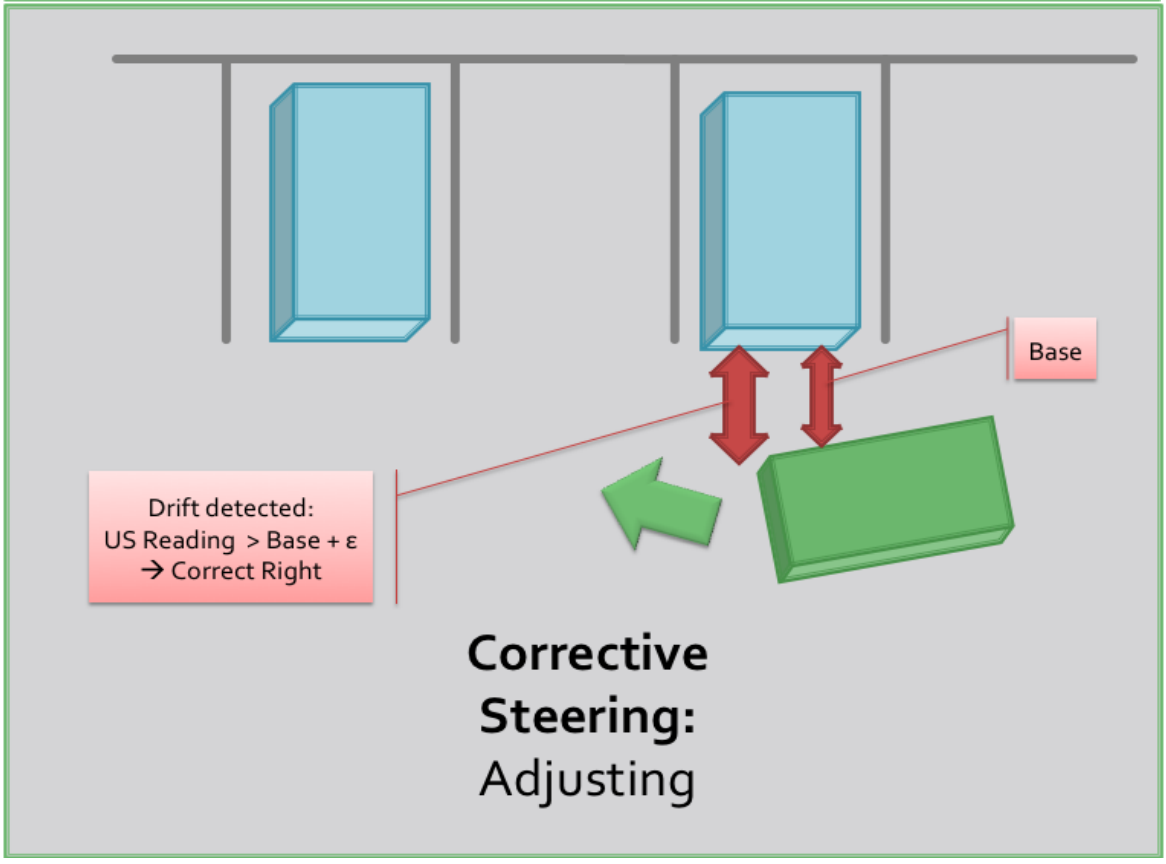
A pictorial explanation of the algorithm can be seen in the following figures:


2. Detect the sudden jump in distance
1. Keep track of this base distance (InitialBase)
3. Calculate the difference (drop)
5a. Possible spot for parallel parking
4. Compare to car length and width, plus some extra
5b. Possible spot for head-in parking
Parking Spot Detection


6. Continue driving to detect how much space there is
7. Know this distance (distance_ remembered)
8. Calculate if the car can fit
Parking Spot Detection



Corrective Steering:
Dealing with Parking Spots



Corrective Steering:
Adjusting

Major Technical Challenges

Design

Realistic Conditions
The goal of S-PAVe focused on designing a system that would improve on current technology, but remain robust without the need for a large overhead of sensors. Although more information about the world is beneficial, it does not translate for a less complex and faster solution. Therefore, a specific design challenge was to propose a solution that would integrate current technology (hard-coded parking), partially-observable environment model (human-parking with errors), and reflex-agent (pure simulation). This design would function off of very cheap sensors and be easily integrated into new situations.

Concurrency and Abstraction
After achieving a workable design, even if additions were required later, the next challenge revolved around parallelizing and running concurrent processes. A large part of this was how to correctly abstract away functionality, so that the algorithm would not need to be worried about the motor speeds and wheel feedback just to drive straight. Another significant portion concerning concurrency design centered on the shared usage of sensor data, which needed to be extracted, filtered, and processed. Overcoming this specific challenge became the most difficult part of integration.

Chassis
Since S-PAVe needs to be realistic, the dimensions and features of the chassis need to be to scale. As a result, S-PAVe has similar dimensions to the BMW 5 series, which is widely considered the perfect sedan. Also, to turn similarly to a regular car, the chassis must utilize single axel turning – the two front wheels must turn together on a single axel. This was a design challenge due to physical constraints and high-level design requirements. Numerous chassis designs had to be redrawn or scrapped because two cars could not be built, allowing for parallel development, and did not meet requirements.

Sensor Choice and Placement
From only four sensor inputs, the robot collects all the data from the physical world and makes a judgment on the state of itself in relation to the world. Important design issues include the selection and placement of sensors. In the final version of the car, there are two touch sensors and two ultrasonic sensors. The two touch sensors were selected because they were accurate and reliable in short range and they are mounted on the front and rear bumpers. This placement was chosen because the algorithm did not need front or rear distance values, except at extremely close ranges where the ultrasonic was most inaccurate. Two ultrasonic sensors were selected because they provided distance readings in a range of distances, and both are mounted on the right side of the car.

Motors
The turning mechanisms available to NXT robots were not designed to be modeled after real cars. They were designed to be used much like the iRobots where a circular turn in place is desired. In this way the wobbling wheels would hardly be noticed. However, designing proper motor control based off of single-axel wheels proved to be extremely difficult. A major concern of this design was the give on the motor, which did not register in the default motor feedback. Therefore, a significant portion of time was spent approaching a solution and designing optimal motor controls. This resolved into having two modes, mainly a continuous and smooth iteration.

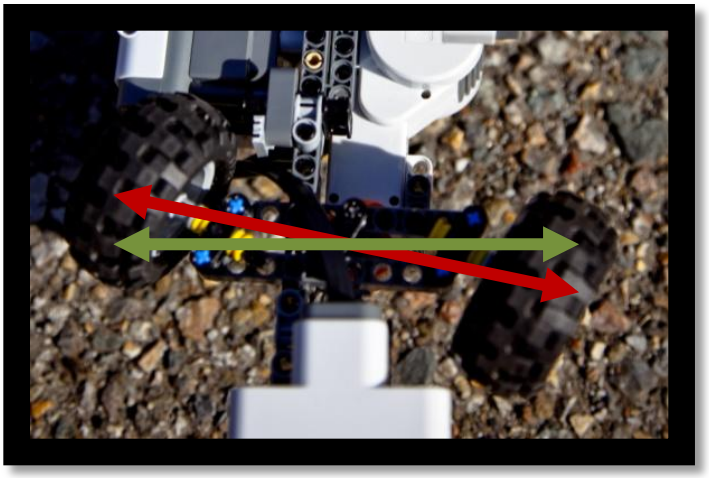
Implementation

No Floating Point

Although the development requirement specifically mentioned having no floating point, the proposed solution of scaling up by 1000 and then dividing again did not facilitate implementation at all. A significant amount of precision was lost in every calculation where this scale could not be used, essentially when data from sensors and to actuators could only be fixed point. Coupled with various errors in sensor readings, mechanical instability, and numerous computations revolving around pi, the software challenge to account for all of these errors was immense. After finally receiving a car, coding the implementation was not easy.

Chassis

Normally, implementing a workable chassis would not be a challenge. However, using Lego Mindstorms pieces turned out to be a nightmare. The pieces at disposal were not meant to create a scale model of a car. Pieces do not necessarily fit together. The plastic parts bend and create flex in the car, causing the car to drift. Also, the motors are gigantic compared to the wheels. On top of that, these pieces are especially not meant to create a single axel turn. Using the pieces provided, it took five different models to halfway approximate that type of turn. The final implementation is still not completely accurate; as the line intersecting the center of the wheels is not invariable, it shifts with the turning of the wheel (the green arrow depicts an ideally implemented wheel axel while the red arrow is the actual):

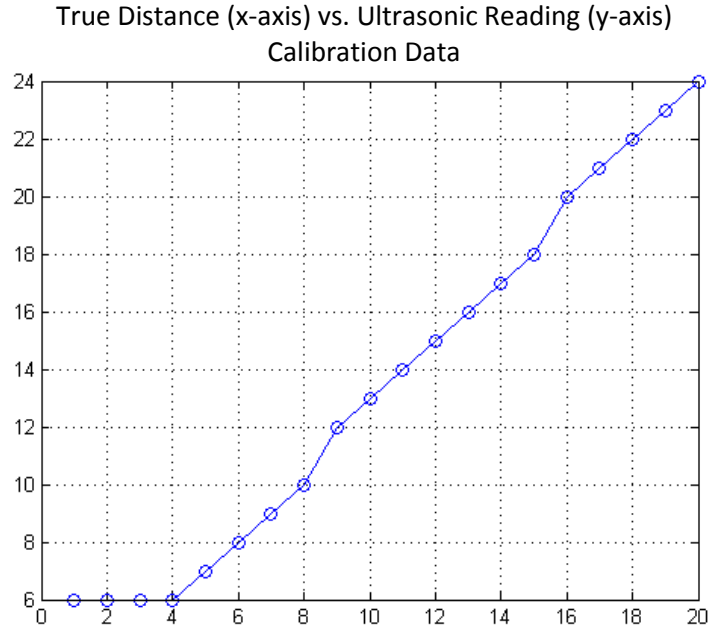


With these requirements, it is nearly impossible to fit everything into a configuration that models the dynamics of a real car without resorting to brute force and super glue, and even then, there were compromises made.

Sensor Usage

The sensors, despite being calibrated extremely well, would incur a low signal-to-noise ratio. When first implementing with two ultrasonic sensors, the interference would create spurious results where a mid range value 25-30 cm would always appear even though the only distances from surfaces were below ten and above 40. Even without a second sensor, the technical challenge of implementing a robust algorithm, which overcame nondeterministic results where a sensor reading could read +/- 1 cm, was extremely difficult. Due to the overhead of NXC interfacing with the sensors, an overhead time could be seen when retrieving data, which meant the overall state machine would need to handle a less robust system. Most of the reason for a design that would revolve around a ‘time-step,’ (note the slight stop in movement) was to account for any noise or spurious sensor errors.

Also, a significant implementation challenge occurred with low-level sensor usage. The results returned by the ultrasonic sensors when driving were not very impressive, especially compared to controlled, simple test conditions. Possible reasons include cheap hardware, a moving vehicle, suboptimal conditions, interference, and other random environmental distractions. However, the final implementation smoothed the data as best as possible.



On the x-axis, values 0-3, the ultrasonic is virtually useless (<1 returns 255). From 4 on, the real distance is linear with ultrasonic reading.

Motor Usage

Two methods were available to control the servomechanism motors: 1) command the motor to turn a specific number of degrees at a set power level; or 2) command to motor to turn indefinitely at a set power level. While these two choices appear to make implementation of a driving control system very easy, in fact, a crucial element is missing: lack of floating point and measurement of distance travelled.

Having no floating point really hurt S-PAVe when it came to turning because of the default gear parts in the NXT kit. The wheels could only physically turn due to the gears at specific increments that NXC did not allow. For example, the *turnWheelsTo(1)* function with a parameter 1 turned a lot more than 1 degree. This was with the lowest power setting such that the gears and motors could overcome the tension and coefficient of friction. Overall, after changing to different designs, the high-level motor usage could only allow for +/- 8 turns. This gave us eight degrees of freedom to work with, which limited the range of where the car could start to park, i.e. the turning radius is not ideal for a real car.

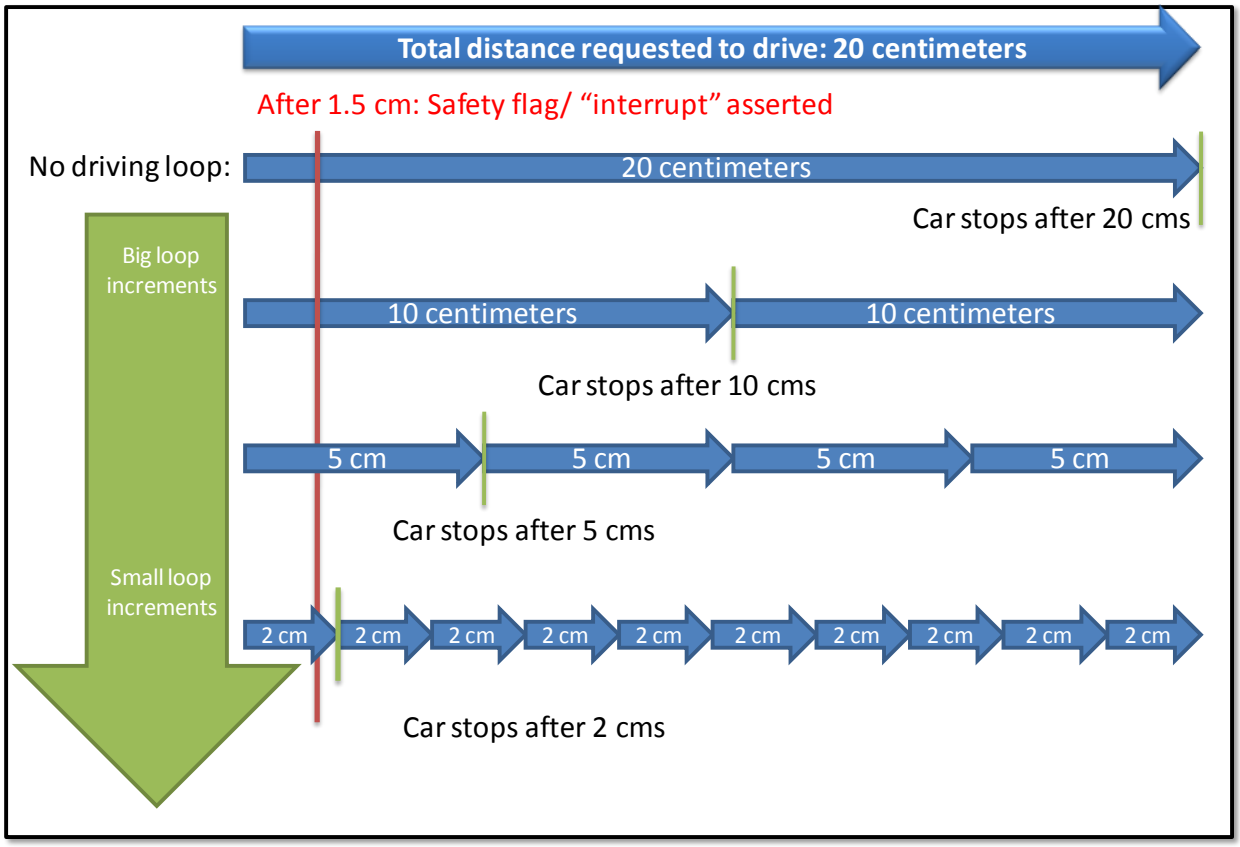
Also due to the complex data gathering and maneuvering needs of our system (i.e. the car needs to know relative location to measure distance, and must travel relative distances accurately to maneuver safely into parking spots), implementing the “odometer” was both the biggest priority and the biggest challenge. (An “odometer” is a built-in feature of the iRobot’s, but there is no such feature for NXT).

Creating a robust and modular driving system was the first requirement. The relationship between the degrees of the motor and the rotation of the tires was determined through calibration testing. Then, this

calibration constant was used to translate a desired distances to travel into the correct number of degrees to turn the motor. (See DrivingControl_continuous.nxc, lines 7, 227, 243). The fact that the servomechanism motors could precisely and exactly turn a specified number of degrees helped greatly: without needing to build a feedback system the motor was guaranteed to turn exactly as much as was commanded. This became the foundation for implementing a robust odometer.

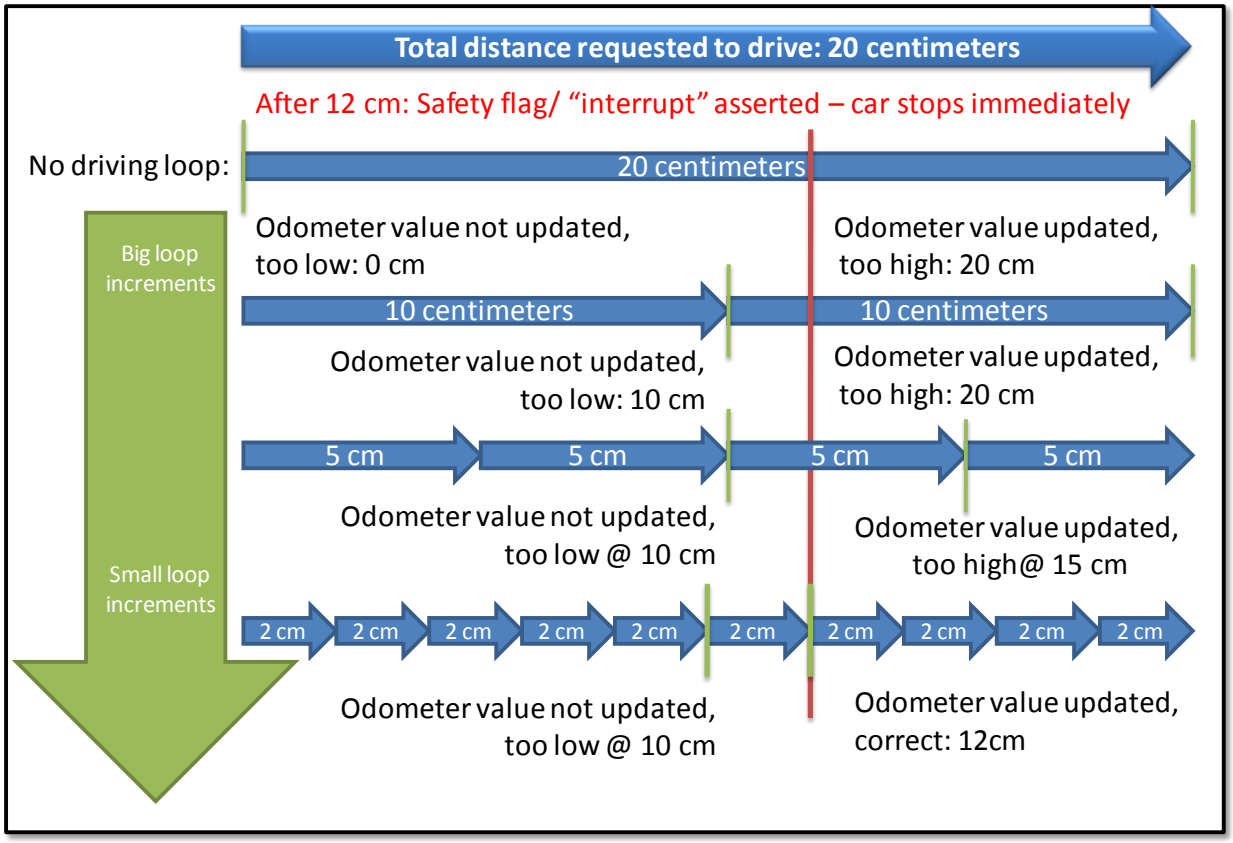
To have the car drive as well as update an odometer value at the same time, the solution, as shown in Figure X (Program Flow for the Driving Task) above, is to divide driving into small pieces which are executed in a loop while updating the odometer. For example, if a distance of 30 centimeters needed to be driven, this could be broken up into 6 iterations of a loop that tells the car to drive 5 centimeter increments, and adds 5 centimeters to the odometer upon finishing each loop iteration.

This method of driving also alleviated safety concerns. Due to the limited nature of the native NXT firmware we used along with the NXC development environment, concurrent tasks could not be stopped arbitrarily. They only stopped when they ran to completion. This meant that once we commanded the motor to drive a certain distance, we could not stop the motor until that distance was travelled. Additionally, there is no interrupt system. Instead, our development team used Boolean global variables as "flags" to pass interrupt-style data to tasks. Because the driving occurred within a task, a loop of smaller increments created more opportunities for to check flags before commanding the motor to drive. As show in the example in below figure, the smaller driving increment is certainly safer.



Example – Why Smaller Driving Distance Increments are Safer

The tradeoff, however, is that a precise odometer would only result from smaller increments, which in turn, create more loop iterations that consume more processor resources/cycles to run. In the end, however, the precise odometer was a very high priority: with large increments, if the loop was "interrupted", the car could have travelled some of the increment, but the odometer value might not be updated to reflect this distance. This is because the command to drive is not atomized with the code to increase the odometer (See DrivingControl_continuous.nxc, lines 23-240). Further in the development cycle, a way was discovered to stop the motors arbitrarily at any time; thus, the safety reason for smaller loop increments was no longer a factor. The decision to continue using smaller increments became solely based on the necessity for an accurate odometer.



Example – Why Smaller Driving Increments are Better Due To Non-Atomized Code

Results

What Worked

Our final prototype and software system is able to detect and park into both head-in and parallel parking spots. This is done without hard-coding the path the car should take, but instead, interfacing with the environment with sensors, and moving in the environment adaptively.

The final prototype is an accurate model of automobile mechanics. For the basic specifications, the prototype is a four-wheeled vehicle, with front steering and rear-wheel drive. The dimensions of the prototype are to scale with a typical American sedan. Additionally, because of the nature of this project to simulate parking, the turning/steering mechanism was carefully designed. The front wheels that turn

utilize an axle to turn both wheels at the same time. The pivot point, however, is not the center of the axle, but at each wheel at a point similar to where a suspension system would be connected.

Following from the physical build of the prototype car, the mechanical dynamics are also realistic. The steering mechanism is not stiff, but is loose enough to cause the car to drift, just as a real car would veer unless the driver proactively corrects the orientation of the steering wheel / tires. While this complex mechanical issue was something this project did not set out to include, it became unavoidable, and a corrective steering system was implemented successfully to compensate.

This project also successfully interfaced with NXT sensors and motors, including implementing sensor data filtering and a high-level driving control system on top of low-level library functions.

What Didn't Work

This project were only mildly successful in detecting a parking spot and parking the car by driving continuously while reading sensor data and calculating how to maneuver. The successful prototype, as discussed in the rest of the report and as demonstrated in our final presentation, drives a short distance, stops, collects sensor data, "thinks," then repeats this process. We were able to program the car to drive continuously instead, but its behavior was not always successful.

This was because the NXC development environment was not ideal for development of fast/high-performance applications. The NXC environment relies on the native Lego Mindstorms NXT firmware. Other development environments, on the other hand, such as RobotC which a project team used last year, include custom firmware which is optimized for speed. According to some calculations, RobotC is more than 25 times faster than NXC[1]. In short, the software just could not run fast enough to poll sensors and calculate what to do next at the same time. This problem was exacerbated by the fact that the car had to drive at a relatively fast speed. This is due to the fact that the motors often encountered difficulties overcoming inertia and/or the coefficient of static friction.

Future work for this project would include optimization of existing code while staying in the NXC environment. Alternatively, however, the algorithms developed for the project could be re-implemented in the RobotC environment which could be more conducive to a smoother, natural-looking parking.

Roles

A major facet of this project was the allocation of work. Although having 5 people gave us a theoretical numerical advantage, the widespread backgrounds and range in capabilities required careful delegation of tasks to best optimize work efficiency.

Irving: With Jonathan, shared project manager duties scheduled and led group meetings, delegated tasks, and kept everyone on time with deadlines and goals. He built the numerous different iterations of the car chassis, helped with high-level code logic, design, and review, took over milestone report duties from Jonathan halfway, and also made the intro and final presentations as well as the final poster.

Timothy: Shared duties with Tyler on design and implementation of the parking algorithm with all of the functions, spent hours fine-tuning calibrations, helped Irving design the chassis that would be the basis of the final prototype model, and also assisted Jonathan and Boyang with motor and sensor code.

Jonathan: Started off sharing project manager duties with Irving, scheduled and led group meetings with Irving, and helped delegate and schedule work. He worked on initial milestone reports, made 2nd and 3rd

[1] http://www.teamhassenplug.org/NXT/NXTSoftware.html

PowerPoint presentations, aided with the final presentation, designed and implemented motor module control and algorithm, and also helped Tim and Tyler on algorithm and Boyang on sensor code.
Tyler: Shared duties with Tim on design and implementation of the parking algorithm with all of the functions, spent hours fine-tuning calibrations, helped design what would be the final environment for the car, and also worked on backup car.
Boyang: Implemented sensor module, researched coding environments, and developed manual Bluetooth control. He also helped improve backup car.

Key Concepts

Concurrency

The Mindstorm NXT Brick has an ARM processor capable of 256 threads and running at 48MHz, and the car always had a few threads running, including one for the FSM, two for sensors, one for music output, and one for the motors. We were unable to verify the real time properties of the operating system for timing under one millisecond. Running half a dozen threads, the NXT seems to be able to meet timing deadlines for 10 milliseconds. Wait instructions were inserted in the code because the sampling frequency needed to be limited in cases of sensor reading and music output, which had to dual effect of reducing its priority and thus allow other more important threads to execute. For global variables shared across multiple threads, mutexes were initially used, but it turns out the operating system was already using mutexes under the hood to preserve read and write consistency. On our part, we made sure that only one thread was responsible for changing values to a global variable.

Real Time Behavior Governed by FSM:

As the project was to demonstrate a self parking car, the goal was to drive the car as a human would in a parking structure. Given the limitations of the physical components of the car, it behaved in a manner similar to what one would expect in real world conditions. The car drove slowly, stopping sometimes to gain more accurate sensor data, and rarely appearing to be confused. The FSM for the car modeled the conditions for searching, validating and then executing a park.

Design Methodologies for Embedded System Design

The project was split into four components: the chassis, sensor, motor, and parking FSM so that work could begin immediately on each of the four components. The chassis, sensor and motor were the first to be completed so that the parking FSM could be implemented. Modular code allowed for updates to the chassis, sensor and motor with minimal changes to the parking FSM. One feature we made extensive use of was the LED display which was used to debug almost all aspects of the project. After a relatively stable car was built, an environment was set up to fine tune the parameters and ensure that the car would work as planned.

Feedback

Topics that came in handy:

- Sensors: Types, Calibration
- Modal Behavior: FSM's
- Concurrency
- Scheduling
- Project Management

Topics that would have been useful:

- Basic mechanical prototyping