Supply and Demand Parking Meter

Team #33
Nick Johanson - njohans2
Adam Barbato - barbato2
TA - Yuchen He

Spring 2017

Abstract

Our project uses computer vision combined with a backend control algorithm to prototype a system that fits parking prices to a supply and demand model. The parking meter hardware takes a picture which is sent over a wireless to a computer which uses background subtraction to determine the number of cars present in a scene. This information can then be sent to the control algorithm to adjust prices based on local demand of parking locations by comparison to the global average parking density. Our results show a system like this could be implemented to better distribute parking across a city by modifying prices in accordance with the demand of the location.

Contents

1. Introduction

1.1 Motivation

Parking in dense urban areas is a problem where no obvious solution exists. Many approaches attempt to make it easier to find parking spots, however we believe the problem is that when people spend copious amounts of time searching for the spot it doesn't exist. Thus, the real issue is that there isn't enough parking, or perhaps parked vehicles are inadequately distributed. If parking is equivalently priced across an entire city people are incentivized purely on the distance between their parking spot and their destination, so they park as close as possible. This results in particularly high parking densities near popular locations and the inevitable 30 minute quest for a parking spot.

We propose a parking meter which can be implemented with a backend capable of monitoring prices across a city and creating economic incentives for a more favorable parking distribution. This parking meter would monitor the usage of spots it's responsible for and the prices that were applied, then this information would be relayed to another computer which will alter the prices proportionally. Prices would be reduced in areas with low density and increased in places with high density. The algorithm will have a goal density in mind and attempt to reduce the maximum density below that goal.

1.2 Project Overview

The project itself consists of two main parts, the computerized parking meter prototype and computer vision software that collects the parking data from the street, and the backend software that takes that data and predicts a correct price. Those two parts consist of smaller blocks as shown below in Figure 1.
The hardware consists of three main blocks, the Parking Meter Processing Unit, or PMPU, the Xbee Transceivers and the Camera, with the software that controls each being included in that block. The PMPU is the main processing unit that is powered by an ATMega2560 and controls the rest of the hardware to take pictures via the camera and transmit them to the backend via the Xbee.
The Computer Vision software is the part of the backend that receives the pictures from the PMPU and decides if and how many cars are present in the picture and then reports that textual information to the pricing model. This is the only part of the block diagram to have changed

since the Design Review, where it used to be part of the PMPU's software before we realized that the ATMega did not have sufficient memory to store the required data. Finally, the pricing model software and control algorithm take inputs from multiple theoretical parking meters and tries to ascertain the correct price for that street in order to reach an optimal density of cars within that area. Usually that optimal density will be such that a few parking spots are always open in every area.

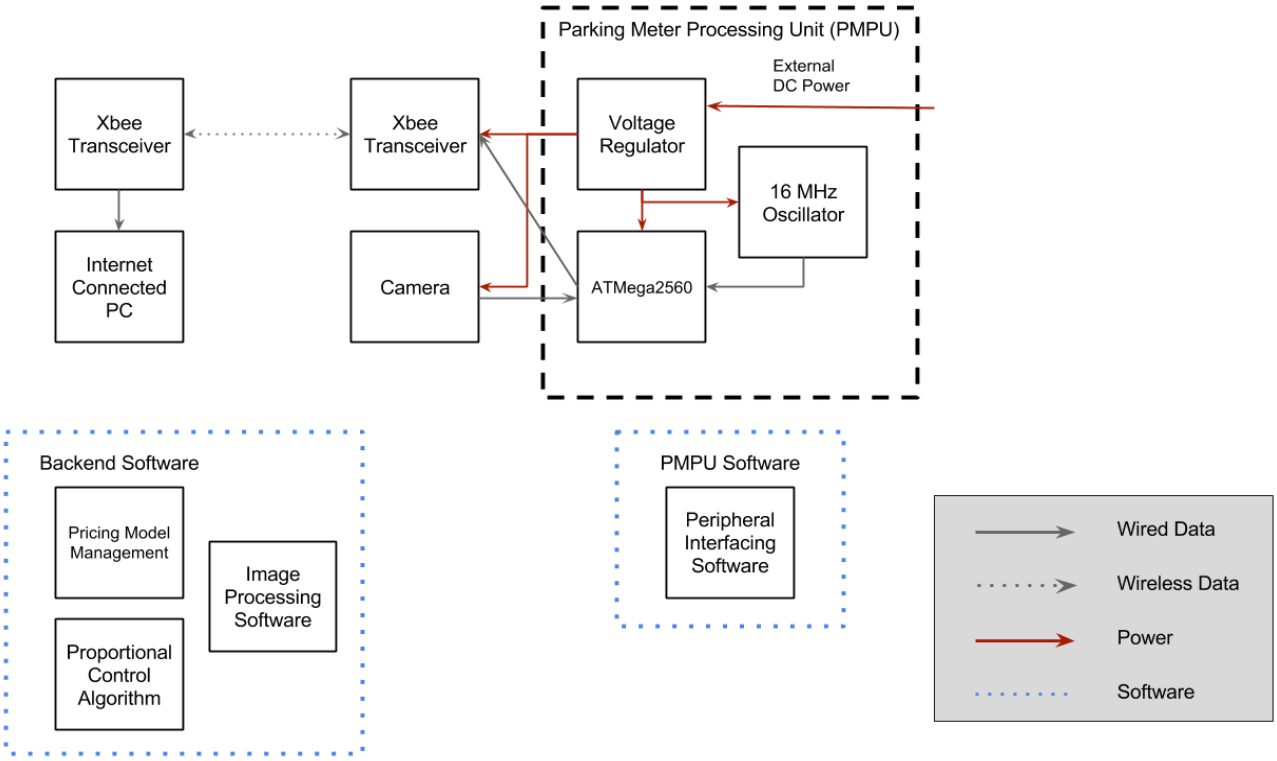

Figure 1: Block diagram of project

2. Design

Full Requirement and Verification tables can be found in Appendix A, and links to any referenced videos can be found in Appendix B.

2.1 Parking Meter Processing Unit

2.1.1 Design Procedure

The main design choice for the PMPU was what processor to use. The most obvious choice was the ATMega328, the same ship that is used in the very popular Arduino Uno. However, after we decided we wanted to attempt to run the Frame Differencing algorithm on it, we decided to use the most powerful ATMega with an easily usable bootloader, the ATMega2560. Once that was picked, the rest of the components we chosen via the ATMega2560 data sheet [1]. It required a 16 MHz oscillator and recommended a crystal oscillator, so one of those was chosen and it required a 5V input, so one of those was put into the design. The capacitor and resistor values were all chosen according the to user guides of the ATMega, crystal, and voltage regulator.

2.1.2 Design Details

As mentioned above, the PMPU consists of three main parts: the ATMega2560, a 16 MHz Crystal Oscillator, and a 5V Voltage Regulator. It acts as the main processing power for the hardware part of the project and interfaces with both the camera and Xbee modules. The full circuit diagram is shown below in Figure 2. The entire circuit was designed to be fit onto a PCB in order to ease circuit construction and cable management, since it needs to interface with so many devices. The final PCB design is seen in Figure 3.

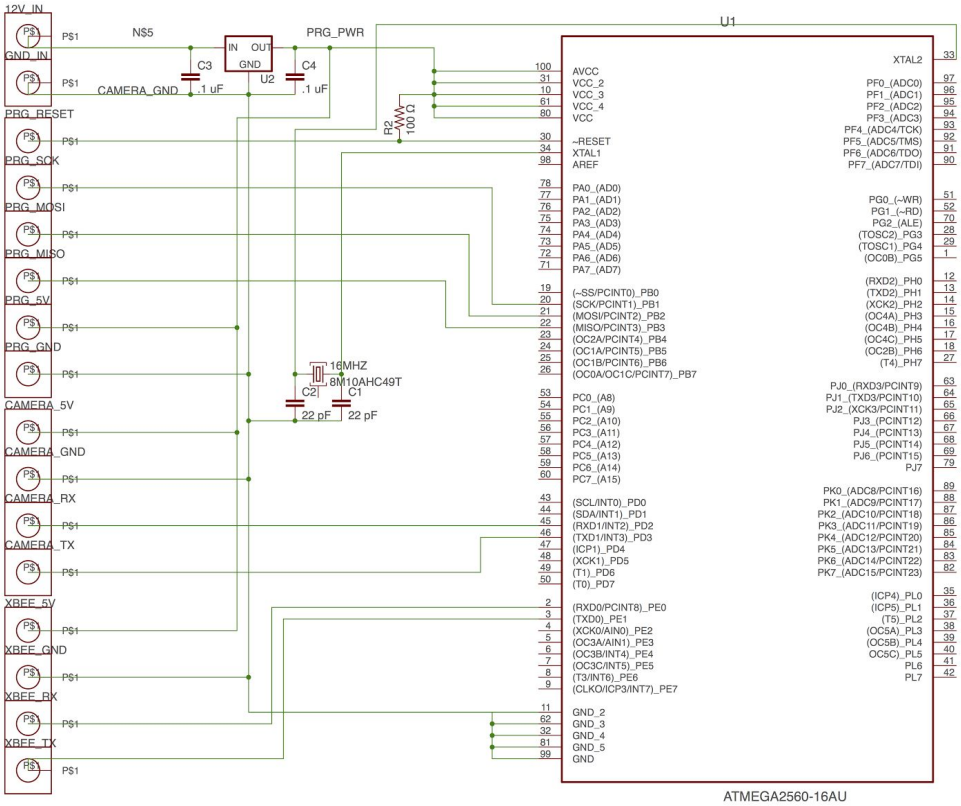

Figure 2: PMPU Circuit Diagram - modules used from element14[2]

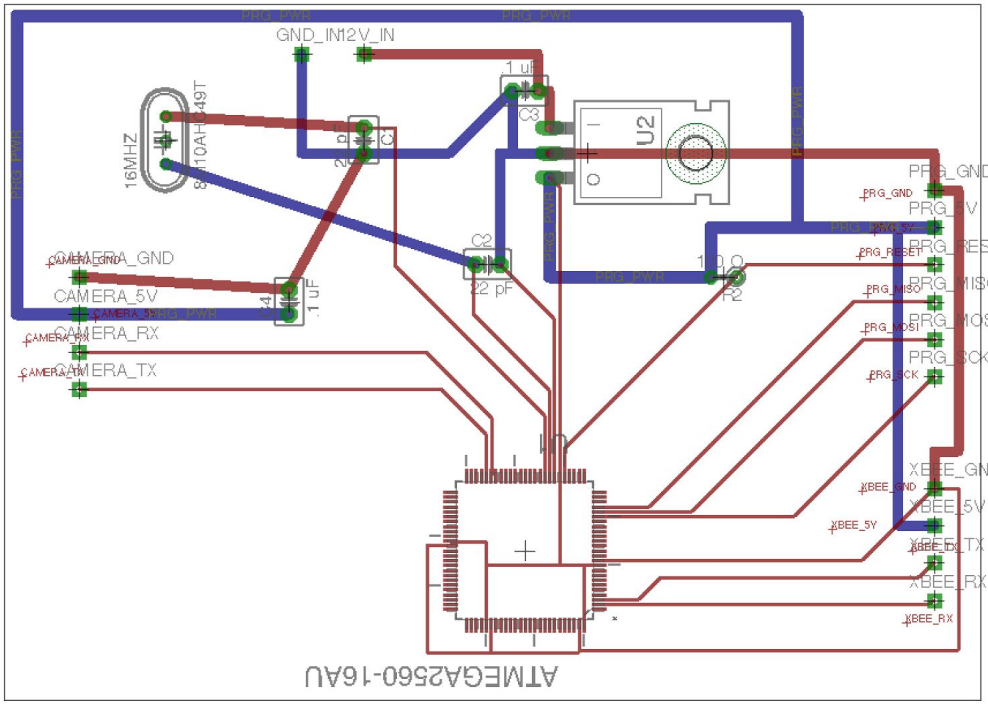

Figure 3: Final PCB Design

2.1.3 Requirements and Verification

The full requirements and verification table can be found in Appendix A, but the summarized version will be presented here. The completed PMPU on the PCB needed to accomplish three main things: be able to regulate an in coming 12V DC signal to 5V, produce a 16 MHz signal usable by the ATMega, and the ATMega2560 must be properly wired such that it can be programed with and execute arbitrary arduino code.

The first requirement was easily verified by a simple voltage reading. A 12.07 V signal was applied to the input pins on the completed PCB and the output power pins showed an output of 5.01 V. The next two were harder to verify. They were tested by uploading the “blink” Arduino sketch to the PMPU, which switches the state of a digital pin after a certain amount of time. Once uploaded, a probe was taken to the blinking pin to verify the changing voltage and it was correct, as shown in video one. The timing was also compared to the same sketch on an Arduino Mega to ensure the oscillator was functioning at the correct frequency, which it was.

2.2 Xbee

2.2.1 Design Procedure

The decision to use Xbees as our radio transceivers was an easy one for a few reasons. First, they were long range enough and had enough throughput to handle the data we needed to transmit. However, most importantly, they were easy to configure using the XCTU software that comes with them and work seamlessly with both the built in libraries in Python and in the Arduino IDE.

2.2.2 Design Details

Most of the work done relating to the Xbees was all in software. As mentioned, setting up the Xbees started by using the XCTU software to put the two Xbees on the same channel and having each other as their target device. Once completed the Xbees could simply be plugged into the PMPU or computer and communicated to over *Serial* libraries. Support software was then written for handshaking and acknowledging, since the Serial communication only send plain text.

2.2.3 Requirements and Verification

As mentioned above, the requirements on the Xbee were more requirements on the software that ran the Xbees, since the Xbees themselves were pre-made. The two requirements were that the software had to use the Xbees to talk to each other at a distance of 50 ft and that they

could communicate at up to 1 kb/s. The distance requirement was verified in video two, where the Xbees retained communication at up to 400 ft away. The speed requirement was tested by rapidly sending a single character over the channel for 10 seconds. The results after 10 seconds were 9385 bytes, which in kb/s would be:

$$9385\ bytes/10\ seconds = (9385 * 8)\ bytes/10\ seconds = 75056\ bits/10\ seconds\ =\ 7.5056\ kb/s$$

which is well over the required 1 kb/s.

2.3 Camera

2.3.1 Design Procedure

When trying to decide on a camera for the project, the adafruit TTL arduino camera was the most recommended one we could find. It worked well because adafruit supplied sample code [3] for interfacing it with an Arduino Mega, which would run on the PMPU and it could be adjusted to work at QQVGA, QVGA, and VGA resolutions, which would give us options depending on what resolution we found worked best.

2.3.2 Design Details

As with the Xbee, the main part of the design of the camera was software design to control the camera. As mentioned above, a large part of this software was provided by adafruit[3], which had sample code for taking a picture with the camera and storing it on an SD card. We modified that code such that it would wait for an incoming signal from the Xbee to execute and then send the resulting picture over the Xbee instead of to an SD card.

2.3.3 Requirements and Verification

The camera only had two requirements, one of which was a requirement on the system as a whole and one on the software. The first was that the camera had to be positioned in order to have a field of view such that two cars could fit within the picture. This is shown below in Figure 4.

Figure 4: Sample photo from camera, showing two cars in FOV

The second was that it had to be able to transmit a QVGA picture to the backend when instructed over Xbee, this is shown in video three and shows that this was accomplished in around 20 seconds. This time held fairly consistent throughout subsequent tests.

2.4 Computer Vision Software

2.4.1 Design Procedure

The Computer Vision Software block contains two main parts: the actual Computer Vision algorithm, and the communication software that enables the backend to receive pictures from the PMPU.
Since this project was time constrained to a semester, the circumstances of our project had to be constrained in order to make a working Computer Vision system, as a more robust one would have taken far longer. Our constraints were that the system had to have consistent lighting conditions and a static background. With these constraints in mind, the best and easiest solution for a Computer Vision system was a Frame Differencing algorithm with a threshold value. We decided that once the Frame Differencing was complete, we would decide if a car was present if the following equation returned a value larger than 5 dB:

$$10 * log(avgCar/avgBackground) \qquad \textit{(Eq. 1)}$$

where *avgCar* is the average pixel value of the area of the photo where the car would be parked, and *avgBackground* is the average pixel value of the area of the photo where any car would not be parked

2.4.2 Design Details

As mentioned in the *Xbee* section, communication with the backend computer connected Xbee occurred over the *Serial* python library. Due to previous configuration, data written to the Xbee serial address would be directly sent to the PMPU and its data would likewise be sent directly to the backend. This was used to set up a handshake system where the backend would send a request for a picture to the PMPU and would start receiving picture data soon after. The backend would then continue to store the picture data in a new jpeg file until it received the end-of-file indicator. It would then save the new jpeg file.

The Frame Differencing algorithm was implemented like other similar algorithms. Two pictures were taken and store by the previous method. The first picture, taken at boot up, would be considered the “background” image. The second picture, taken right before the frame differencing occurs would be the “current” image. Once both images are stored and ready, they are compared pixel by pixel and if the difference between the two pixel values exceeds the threshold (in this case, 100) a white pixel is inserted in that location on a newly created third image, otherwise a black pixel is inserted. To determine if a car is present in a particular spot, the average value of pixels of the parking spot is compared with the rest of the picture according to *Eq. 1*. We know where the parking spot is beforehand, because they do not move and we assume that cars park within those spots. If *Eq. 1* shows a value of greater than 5 dB, we assume that a car is parked there, otherwise we assume the spot is empty.

2.4.3 Requirements and Verification

There were four main requirements for this block, two of which were about data transmission and two of which were about the Frame Differencing algorithm. The first was that the backend must be able to request a picture from the PMPU, receive, and store it within 1.5 minutes. This was demonstrated in video three, and was completed in around 20 seconds. The next requirement was that the backend could send pricing data to the PMPU and receive a response within 30 seconds. This is show in video four and shows that when tested, the round trip response can occur in just a few seconds.
Next, we have the large requirement that the Frame Differencing algorithm must be able to complete the frame difference on a QVGA image within 1.5 minutes and have the background attenuated by 5 dB, as shown in *Eq. 1*. An example of the algorithm can be seen below in Figures 5-7.

Figure 5: The “background” image in the example Frame Difference

Figure 6: The “current” image in the example Frame Difference

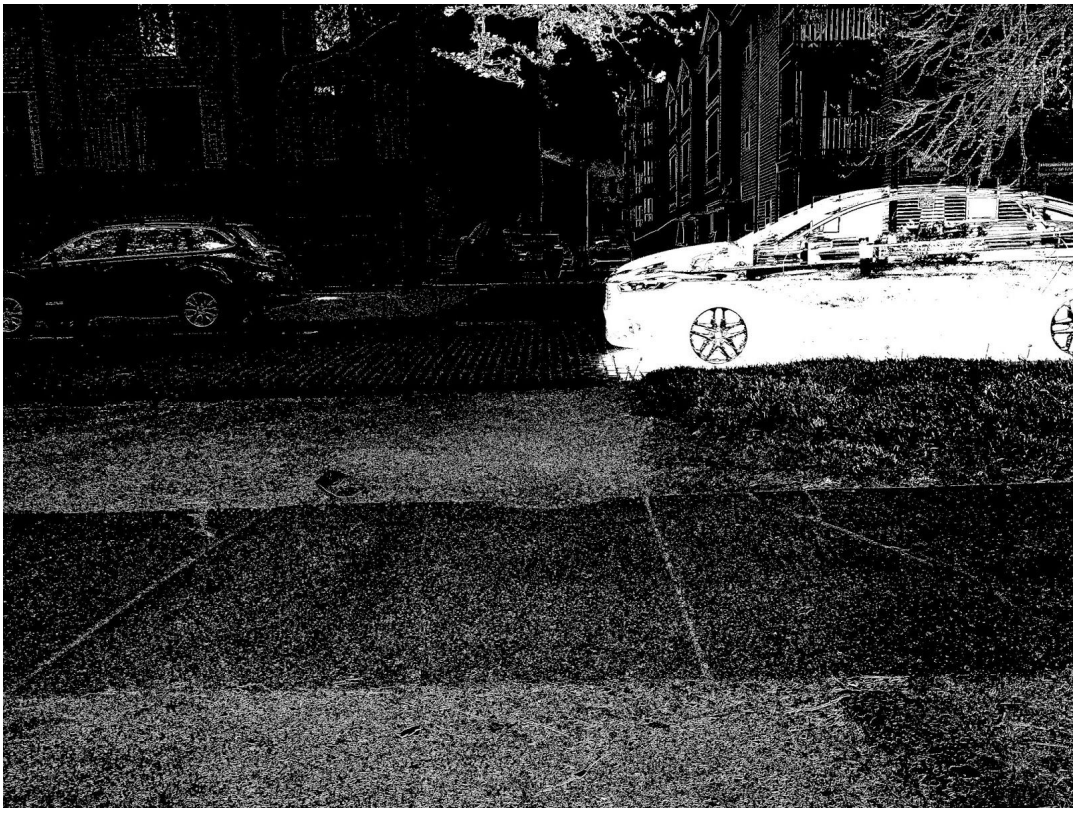

Figure 7: The resultant image from the example Frame Difference

In the above example, the parking spot area as an average pixel value of 156 and the background as an average pixel value of 19, thus by *Eq. 1*:

$$10 * log(156/19) = 21.05\ dB$$

which is above 5 dB and therefore would assume that a car is parked, which is true.

Finally, there was one last requirement and the only one that we failed. There was a requirement that the Frame Differencing run on the PMPU, which is why we initially chose the stronger ATMega2560 over a smaller ATMega chip. However, we eventually realized that even with the ATMega2560's increased memory capacity, that it was not enough to hold two full images and process them, so we had to move the Frame Differencing to the backend computer.

2.5 Control Algorithm

2.5.1 Design Procedure

We chose to use proportional control with a linear backtracking search to keep values strictly positive. Given our lack of experience in control theory this implementation keeps the problem simple enough for our timeline. The linear backtracking method was chosen because it is low in complexity but sufficiently solves the problem of keeping prices strictly positive during proportional control.

2.5.2 Design Details

The proportional controller computes the difference of a local density and the global average density as a percentage of the maximum possible density, then multiplies this value by the price range and a constant scalar value. This value becomes the new price,

$$P' = K_p * (D - D_{avg})/100 * P_{range} \qquad \textit{(Eq. 2)}$$

where P' is the new price to be used, K_p is the scalar value, D is the current density, D_{avg} is the global average density, and P_{range} is the price range.

Linear backtracking was implemented by checking to see if the new price was below zero, if so then we reduce K_p iteratively until a non zero value is found. Our implementation resembled the following pseudo code:

```
while new_price < 0:
      Kp = Kp*0.5
      new_price = Kp * (density-average)/100 * price_range
price = new_price
```

2.5.3 Verification

The control algorithm was verified by generating agents with a price to use for decisions. Then a single “parking lot” was set up with 100 available spots. If the price to park in the lot was below

an agent's price they parked in the lot, if it wasn't they did nothing. After all agents were given a change to park the price was adjusted in attempt to get a specific number of agents parked.

For the algorithm we desired a proportional change in price compared to the distance between the current density and the goal density. This change would need to be positive (increase in price) if the density was above the goal, and negative (decrease in price) if the density was below the goal. The following graphs show the behavior of the test explained in the first paragraph:

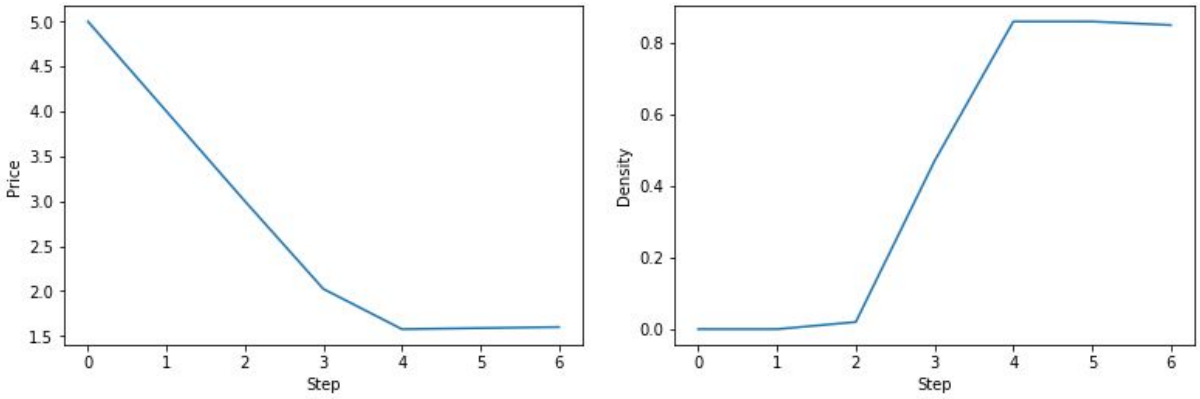

Figure 8: Price and density at each iteration

In the above graphs we can see that as the simulation executes the density increases in response to the price change. For this test the goal density was set to 85%. We can also see how the price changes when the density is below the goal. The following graphs show that the control algorithm behaved appropriately when the current density was above the goal.

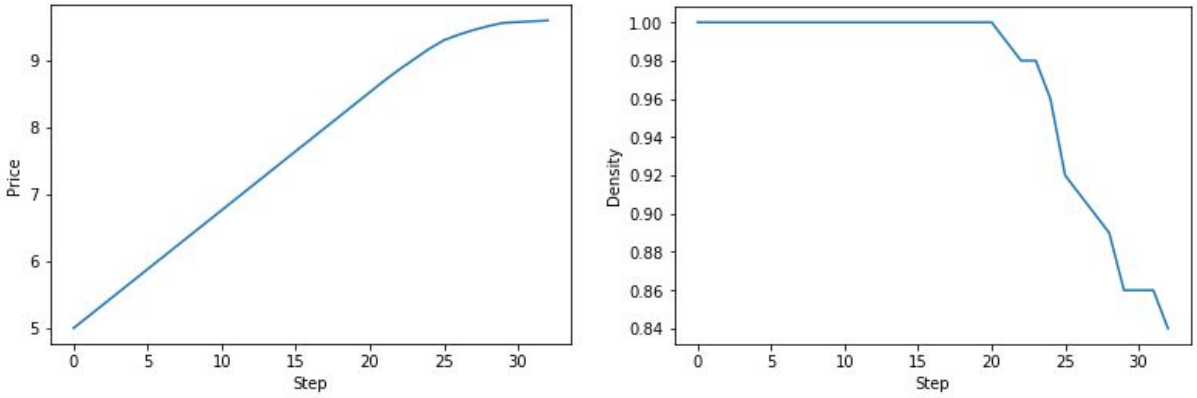

Figure 9: Price and density at each iteration

2.6 Simulation

2.6.1 Design Procedure

The simulation was designed to verify the control algorithm could be implemented on a larger scale and still produce meaningful results. We ran the control algorithm over one hundred iterations using a 10x10 grid and five thousand generated agents which choose where to park. This scenario simulates the price changes for one hundred parking lots over one hundred days.

The simulation was coded in python using Spyder as the development environment. Grid locations, price values, and agent features were stored in parallel inside of arrays that they could be accessed using a loop counter when the simulation was run.

2.6.2 Design Details

Each agent has three features: a desired location to park on the grid, a price they are willing to pay, and a search radius each one uses search for a parking spot. The latter two of the features are generated using a normal distribution to randomly assign the values.

Each agent decides where to park by first looking at their desired location. If they are unable to do so, they search all grid squares within their search radius for the lowest price and park there. Then, if they were still able to find a location to park they increase their search radius by one and repeat the process until they are parked.

Once all agents are parked the new pricing model is calculated using the control algorithm specified above and the agents repark in the next iteration with the updated prices.

2.6.3 Verification

Using the numpy library we were able to generate agent features with a discrete or continuous normal distribution as outlined in our requirements. We decided on using a discrete distribution for modeling desired prices and a continuous distribution for search radius as shown below:

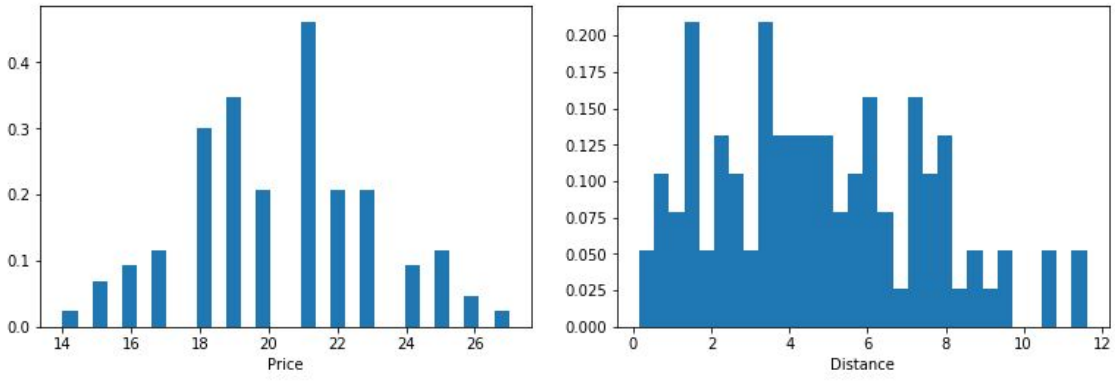

Figure 10: Distributions of price and search radius features of all agents

After one hundred iterations we graphed the average of the non-zero densities and the median density at each iteration. This result showed our algorithm produced a marginally stable system and the non-zero average converged to ~65% and the median converged to ~53%:

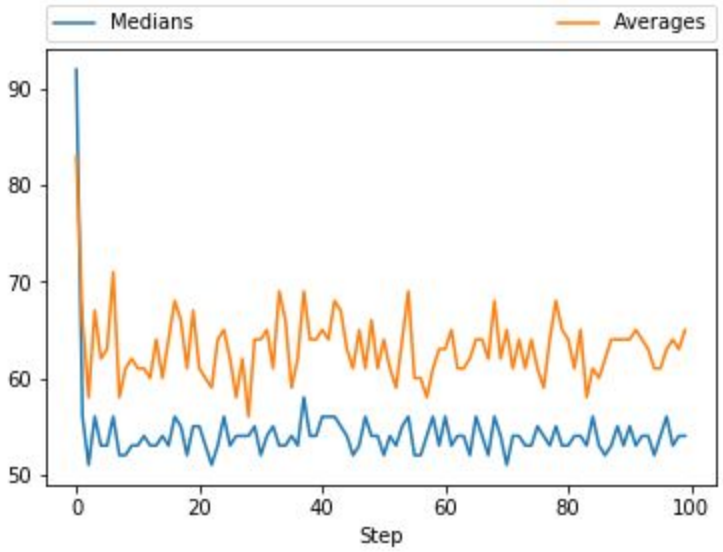

Figure 11: Average of non-zero densities and median of all densities at each iteration

The heat maps for the first and final iteration respectively are shown below:

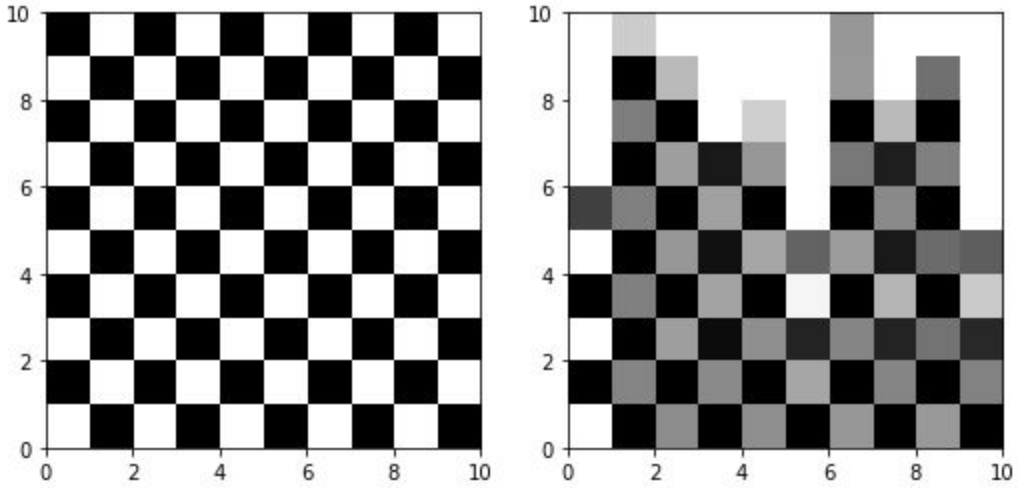

Figure 12: Heat maps at iteration 0 (left) and iteration 100 (right)

From this we can see that while the algorithm wasn't able to produce a perfect distribution (all grid squares have the same density) the final distribution shows improvement over the first since previously unused parking spots are now being used.

3. Costs

Table 1: Costs of Labor

Name	**Hourly Rate**	**Hours Invested**	**Cost**
Nick Johanson	$35	95	$3,325
Adam Barbato	$35	90	$3,150
		Total Labor Cost	$6,475

Table 2: Costs of Parts

Part	**Quantity**	**Unit Cost**	**Total Cost**
ATMega2560	2	$13.25	$26.50
XBee 1mW Trace Antenna (802.15.4)	2	$0	$0
TTL Serial Camera	1	$54.95	$54.95
Voltage Regulator-5V	2	$0.95	$1.90
16MHz crystal oscillator	1	$0.75	$0.75
Arduino Mega	1	$45.95	$45.95
PCB	5	$30.00	30.00
		Total Cost of Parts	$160.05

4. Conclusion

Given our reduction in scope, background subtraction with average pixel value comparison was sufficient to determine the number of cars in front of the camera. The image could be captured on the hardware and quickly transferred over a radio transceiver to a computer which could run both the computer vision algorithm and the control algorithm. Our initial design had the computer vision running on the selected microprocessor, however our software libraries didn't allow for this functionality. This problem was discovered too late for us to implement a different hardware scheme which would allow our hardware to meet the initial design.

The control algorithm produces acceptable results given the nature of the simulation. Prices changed proportionally and the average of non-zero densities dropped to a more acceptable level. However, since the agents used are very poor models for human behavior, more time would need to be devoted to developing the control algorithm in a real world scenario.

In a real implementation of this system we acknowledge the possibility for our system to be abused to price gouge, or to manipulate prices to increase monetary gain. Any malicious act like this would go directly against the IEEE Code of Ethics' point 1 about keeping the public's welfare in mind and point 8 by discriminating against those who cannot pay [1]. However, the system was not designed to work that way and changing the system to accomplish either of these unethical goals would be against its main purpose: to make easy parking more available for everybody.

5. References

[1] “ATMega2560 DataSheet,” *atmel.com*. [Online]. Available: http://www.atmel.com/Images/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf.

[2] “Atmel CAD Library for Cadsoft EAGLE Software” Eagle CAD Libraries. element14. N.d. Web. 23 Feb. 2017. <https://www.element14.com/community/docs/DOC-64259/l/atmel-cad-library-for-cadsoft-eagle-software>

[3] O., “oskarirauta/Adafruit-VC0706-Serial-Camera-Library,” *GitHub*, 18-Dec-2012. [Online]. Available: https://github.com/oskarirauta/Adafruit-VC0706-Serial-Camera-Library. [Accessed: 04-May-2017].

[4] "7.8 IEEE Code of Ethics." IEEE Code of Ethics. IEEE. n.d. Web. 21 Feb. 2017.

Appendix A: Requirement and Verification

Data Collection

Requirements	Verification
1. Data from all meters should be collected successfully every 5 minutes.	1.1 Set up a data packet and send it to backend using transceiver. 1.2 Confirm it was received. 1.3 Backend can send acknowledge 1.4 Use timing data to establish upper bound on number of meters.
2. If data is missed, it should poll the meters it doesn't have data for.	2.1 Repeat process above but have backend randomly ignore some information on first pass 2.2 Have backend print message when all data is received. 2.3 Check array to confirm all data was received.

Data Storage

Requirements	Verification
1. Arrays should contain correct data at correct index. After being received by collection process.	1.1 Generate meter data and send it sequentially over transceiver. 1.2 Check array values and confirm they are in the appropriate location

Control Algorithm

Requirements	Verification
1. Changes in prices should reflect the difference between the current density and the goal. If the current density is too high the price should proportionally increase, if it's too low the price should proportionally decrease.	1.1 Setup the algorithm and give it a parking density 1.2 Confirm the correct change happens. 1.3 Set the density close to the goal 1.4 Monitor changes as density changes

	towards goal to confirm the change decreases.

Simulation Test

Requirements	Verification
1. Generated personalities should resemble the designated distribution with error <10%	1.1 Generate 10, 100, 1000 personalities based on distribution. 1.2 Graph personalities and calculate average error.
2. Parking densities should converge and form a stable system within 100 iterations.	2.1 Run simulation for a single grid tile using a park or no park decision 2.2 If this system converges expand to 25 tiles (5x5) 2.3 If this converges expand to 10x10 for final test. Graph densities after at various numbers of iterations (5, 10, 20, 30... etc) to confirm system is converging.

Camera

Requirements	Verifications
1. Angle and placement of lens must allow three cars to be captured in one picture	1.1 As shown in Figure 5 below, make sure camera is positioned such that 60 degree angle of lens could capture two cars. 1.2 Using another method, move a picture to a screen and verify that two cars are in the picture
2. Must be able to transmit a QVGA picture to arduino when instructed via software	2.1 Hook up camera to arduino and USB connection from laptop to arduino 2.2 Run picture software on PMPU/arduino 2.3 Wait for picture to appear on laptop 2.4 Verify that resulting picture is QVGA

Xbee Transceiver

Requirements	Verifications

1. Software that runs the transceiver must be able to transmit data both directions between the parking meter and a waiting computer within 50 ft of the parking meter	1.1 Hook up one Xbee to Arduino or PMPU and the other to a laptop using the provided Xbee software 1.2 Use our software to transmit continuous data while moving Xbees away from each other 1.3 Confirm on the laptop that the Xbee receives data until at least 50ft away
2. Must transmit at least 1 kb/s	2.1 Repeat verification 2, ensuring that the transmission rate is greater than 1 kb/s using the Xbee software on the laptop

PMPU

1. Must be able to regulate incoming 12V DC to at least 5±.5V and provide at least 1A output	1.1 Hook up 12V DC power supply 1.2 Measure output current and voltage, pins 80 and 99 on ATMega2560, ensuring that they are within the acceptable ranges
2. Must produce a 16±.5MHz clock signal	2.1 Hook up output of oscillator circuit, pins 33 and 34 on ATMega2560, to sprectrum analyzer and measure the frequency for at least 30 seconds 2.2 Ensure that the 16MHz signal is the main output component
3. ATMega2560 must be corrected designed and integrated into the PCB such that arbitrary Arduino code can be uploaded to and run on it	3.1 Hook up PMPU to Arduino Mega 3.2 Load modified “Blink” sketch onto the PMPU through Mega pass-through 3.2 Connect LED to “Xbee RX” pin 3.3 Ensure that LED blinks

Computer Vision Software

Requirements	Verifications
1. Must be able to request a picture from the camera, receive and store it within 1.5 minutes	1.1 Start software with PMPU/Arduino connected to laptop via Xbee 1.2 Request camera take and store one picture 1.3 Record the time it takes to finish and ensure it is less than 1.5 minutes
2. Must be able to perform a Frame Differencing background subtraction using two separate QVGA pictures and attenuate	2.1 Start software with PMPU connected to laptop via Xbee 2.2 Load a background and modified image

the background by 5 dB of the foreground within 1.5 minutes	with different foreground into software 2.3 Request it to background subtract 2.4 Ensure it finished in 1.5 minutes 2.4 Request the returned picture 2.5 Verify picture's background has been attenuated by 5 dB
3. Frame Differencing must run on PMPU	3.1 Repeat verification 2, but ensure it is running on the PMPU
4. Must be able to transmit parking data via Xbee to backend and receive response within 30 seconds.	4.1 Hook PMPU/Arduino up to laptop and launch provided software 4.2 Make sure that the receiving Xbee attached to the PC running the backend software is in range and running 4.3 Request that the PMPU send parking data 4.4 Check the PC to make sure it received the data 4.5 Ensure it returned in 30 seconds

Appendix B: Videos

Video 1: PMPU Blink
Video 2: Xbee distance
Video 3: Picture transmission
Video 4: Xbee response