Group Dec 15-19

Final Document

Efficient Measurement of Soil Microtopography to Aid the Verification of European Space Agency and NASA Satellite Observations of Soil Moisture

Brant Walsh (bmwalsh@iastate.edu)
Dillon McDowell (dkm@iastate.edu)
Yan Yao Chan (yanyao@iastate.edu)

Advisor/Client: Dr. Brian Hornbuckle
Collaborative Advisor: Josh Bertram

Table of Contents

Problem Statement

The world's first satellites able to remotely observe near-surface soil moisture from space either have recently been launched (European Space Agency, 2010) or will soon be launched (NASA, 2015). Near surface soil moisture is the water content of the top 3 to 5 cm of the soil surface. Near-surface soil moisture affects how water and energy move between Earth's surface and atmosphere, and observations of this phenomenon will be used to make better weather and climate forecasts. Before they can be used, the near-surface soil moisture observations must be validated using on-the-ground measurements. As we have begun to validate these observations, we have found that soil surface "roughness" or the mm-scale variations in the height of the soil surface (microtopography) can "confuse" satellites and therefore must be taken into account. The problem is that it is difficult and time-consuming to make good measurements of soil microtopography. I've heard rumors that the Xbox Kinect can be hacked to make scans of surfaces. Or perhaps commercial camera technology can be used to measure microtopography. Hence I challenge a senior design group to come up with a method of measuring soil microtopography that is quick, accurate, and precise using commercially-available technologies that may have not been intended for this use.

Solution

The solution that our team has come up with utilizes an Xbox Kinect's depth sensors to take an image of a surface. From this image the depth data is gathered and placed into a two-dimensional array where it is further processed to remove any error in the data. This process will utilize MatLab to help visualize the data and create a value that can be applied to the data to correct it. Since the client wants the device to be portable we planned to mount the system onto and inside of a stripped power drill case. This allows it to be handheld and lightweight while still being able to use the power drill's battery as the power supply.

Minimal Viable Product

The minimal viable product that we will produce should be able to take images of a desired area, process the data for any error, and be able to save the output to a location that is easy to get to.

System level design

System requirements

The system requirements include a set goal for the accuracy of our measurements, the amount of time that it takes to gather measurements, the area covered by one measurement and a final output style.

The project's goal for the soil roughness measurement accuracy is millimeter vertical accuracy for every centimeter by centimeter horizontal area within a three square meter area. The client, Dr. Hornbuckle, would like us to use the Xbox Kinect to take the soil roughness measurements. To achieve this we used the IR depth sensor that is built into the Kinect to get raw data that we will analyzed. From the data we could see that we were getting even more accurate than cm^2 tiles.

Our client is requiring the system to display the information that we receive from the Kinect scanner into a format that is easy for the user to read and understand. To achieve this we designed our final output to be a csv file where each value corresponds to the vertical height at a specific pixel on the output image of the area. From this csv file, it's easy to place it into a plotting software to visual the data.

Our system is required to meet a time requirement of under an hour for taking the soil measurements. This is to see if the Kinect is able to have a faster scan time than our client's current soil measurement device which can take around an hour to finish scans for one location. The system that we have created takes a few seconds to take the image and return the output files.

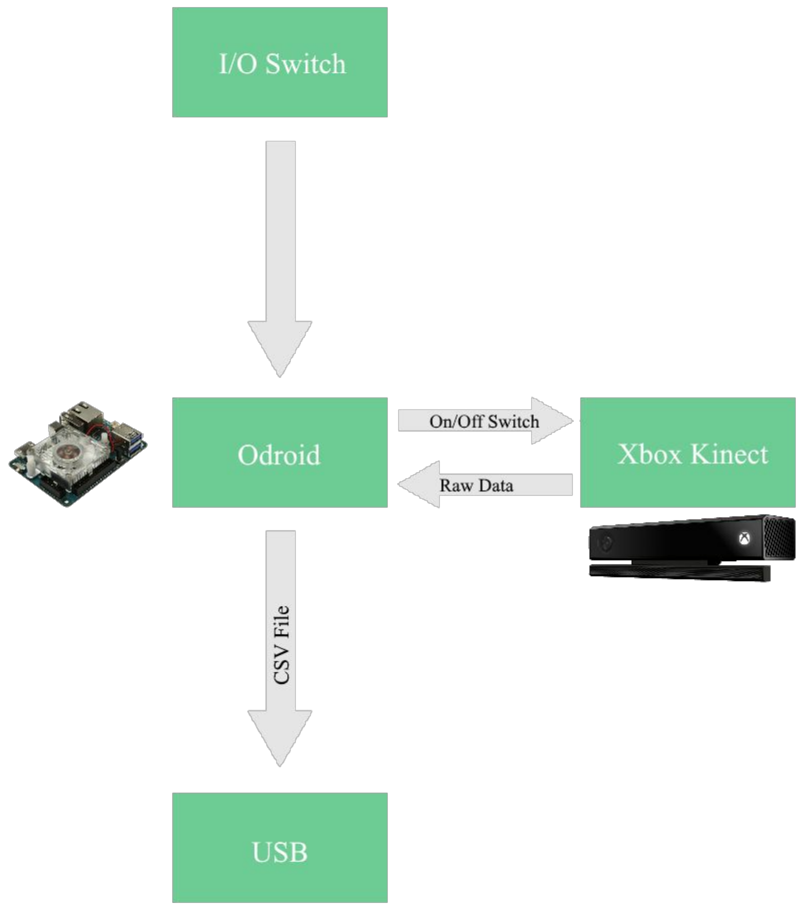
Our system was required to use the Xbox Kinect and a development board. This requirement was to allow the system to be mobile and operate in the fields. The development board that we needed to use was required to have multiple USB slots and have at least one USB 3.0 slot for the Kinect.

The software was required to be as simple to use as possible. Because of this, we made the software collect data and process it with a single click or trigger pull. This helps to make the project easy to use as well since our client wanted the device to be used by people that don't know anything about the system or how it is set up and so we designed it to be easy to use.

System analysis

Our system is comprised of three major parts. The first is the Kinect, the second is the Odroid XU4 development board, and the third are our two buck converters and the drill battery. To start the program on the Odroid, it will be as simple as turning on the Odroid. The program will then receive a signal from a GPIO pin that tells it to run the Kinect and return depth data. When this data is returned, the Odroid will process the data and correct any error around the edges. This data is sent to a CSV file for the user. The way to get the CSV off of this system is to have a USB drive plugged in that supports a Linux file system since our Odroid is running a version of Ubuntu. With this USB, the system will save the data it gathers onto the USB for the user.

System block diagram



Detail description

I/O specifications

Input

The system that we designed does not need any inputs except for a trigger, which will tell our program to capture the data that it currently has and process the data. This trigger should be hooked up to one of the Odroid's GPIO pins. When the trigger is pulled it should change the value of the pin, which will allow our software to know when to run the Kinect.

Output

Our system will output three files. The first is an IR image of the ground. The second is the depth data that the IR sensors returned and that has had it's error removed. The last one is a text file that the Odroid will display. This text file will have information about how much of the data is reliable. The C# program should store any data that it gathers into a flash drive that we will be able to remove from the Odroid and insert into another computer.

User Interface specifications

The user interface for the system will be a screen that comes with the Odroid. It is a small LED screen that is capable of displaying information to the user. We will use this screen to show the user the percentage of data that is unreliable in the most recent data capture. This is so the user can determine if there is enough reliable data in the image or if they need to take another image.

Hardware/Software specifications

For this project we will be using an Xbox Kinect v2, Xbox Kinect to PC adapter, a power drill battery and power drill case, two buck converters, and an Odroid XU4. The software that will be used for this project include two programs.

Software

We will use the Kinect SDK to help create two software programs on the Odroid. One will be written in C and the other will be C#.

The C program will run on startup. Inside this program will be in a loop that runs until the Odroid is powered down. This loop will constantly check one of the GPIO pins that we will have set up to a trigger. When the trigger is pulled it will tell the C program to create a child process that will run the C# program and after that program is done it will open the output file ending with "..-Output.txt" and read the data from it to display it on the Odroid's screen.

The C# program will be made so that we can access the Kinect's drivers and tell it to collect data and process the data. This program is a modified form of a simple depth data program that comes with the Kinect SDK. In the program, the program waits until there is a signal. In the main version of this, it should have run immediately. The version that runs consistently uses a button click listener before capturing data. Both versions will have the Kinect send out its IR points and read that data. This data will then be sent back to us in the form of a 1D array to a function where we loop through it and break that array apart and create a 2D array that is easier for the developer to understand. During that loop we also check for how many of the data values are 0. This means that they had unreliable data and were thrown out by the Kinect. After it has been broken up, it then gets sent to two sets of nested for loops. In these for loops we apply the polynomials that we found through our testing. These polynomials will give us a value that will be applied to the depth data. That value is stored in another two-dimensional array of the same size as the data array. Since we are using two polynomials for getting a correction value, we couldn't just immediately applying the correction values or the depth would be overcorrected. To fix this we average the value that is returned from the x-axis and y-axis polynomials. This value is then applied to the value in the depth data array to correct the data.

Hardware

In this senior design project we primarily utilize and modify an existing power drill to suit the project's needs. First and foremost, the power drill itself is disassembled and all the unnecessary parts such as the default motor and moving parts are removed from the chassis. By doing this we essentially have an "empty" power drill, which allows us to house some of the required parts inside the chassis such as the Odroid, the simple switcher circuits and any essential cables. Before the chassis can be properly utilized further, it has to be slightly modified such that it is able to fit all the required parts better. As such, several hinges between the chassis are altered so that the parts can easily fit into the chassis, which indirectly make the entire assembly easier.

An embedded computer is also required for the system. The embedded computer is required to run an operating system that is capable to use the Xbox Kinect. The computer is also required to have USB 3.0 because it is required for the Xbox Kinect. We chose to use the Odroid XU4 development computer. We chose it for its ability to run Linux and its availability of USB 3.0. The Odroid XU4 was also known to work with libfreenect2, a Linux driver for the Xbox Kinect.

Aside from the Odroid, buck converter circuits are also used in powering up the Odroid and the Kinect using a mobile power source. For the sake of mobility and portability, a rechargeable power drill battery is used. However, the battery could only provide an output voltage of 18V when the Odroid and the Kinect only require 5 V and 12 V to power up, respectively. Since this represents a major issue, buck converter circuits are utilized for both the Odroid and the Kinect to ensure that the suitable output voltage is provided for both devices. As mentioned earlier, the Odroid only needs 5V to power up, in order to get this voltage we built a buck converter using a TI Simple Switcher (LM2678-5). Similarly, we made a buck converter using another TI Simple Switcher (LM2678-12) reducing the voltage down to 12V.

Implementation issues/challenges

Software

Our challenged us to get to millimeter scale on the data results. The Xbox Kinect was originally returning all of the data in what appeared to be inches. We were confused as to why this was happening because all of the documentation about the Kinect that we could find was showing that it should be in millimeters. The issue was finally tracked down to a function in the C# program that we created that was dividing the depth data by an unknown number. There wasn't any documentation for where this number came from or why this was being done, however, removing this division allowed us to read data in millimeters.

When we were finally able to read millimeters we noticed that the data was slightly curved as if you were looking at piece of a sphere meaning the corners and edges of the data read numbers that were farther away than what they really were. From what we could see, it was data error from the IR sensors building up as we got closer to the edges. This was eventually fixed by testing and calibrating the data using MatLab and its Curve Fitting Tool.

However, MatLab was not our first attempt at fixing the data. Our first method that we tried was attempting to fix the data using geometry. We thought that maybe the values are wrong because the Kinect was reading the data back as the hypotenuse of a right triangle formed between the IR sensors, center of the viewable area, and that data point. To fix this we came up with multiple methods that didn't work. We eventually found out that it is not the hypotenuse of a triangle and instead seemed to be following a polynomial line.

Following this information we used MatLab to find a slice (array) of our data and get an average for that slice. This slice was then used as the y-axis in the curve fitting tool. The x-axis was an array where each element just held its element number ID. So for example xaxis[1] = 1 since MatLab is 1 based. Using these two arrays we were able to find a suitable polynomial that could then be applied to the data to correct it on the x-axis. The process needed to be repeated for the y-axis.

The data still has some issues that occur at set heights. The problem that is happening looks to be certain height ranges where the data starts being unreliable. It appears to be a 10 cm range that occurs periodically every 90 cm. We are unclear what is causing these "deadzones" to occur. To help fix this issue, we have a file that is created during the data generation stage that will output how much of the screen is currently unreliable. From this information the user should be able to tell if they are in a "deadzone" and then change the height of the program to take another image.

Hardware

Our client, Dr. Hornbuckle, wants our device to be highly portable so getting a weight that isn't heavy could be a challenge. The majority of the weight will probably be in the battery since it will need to be big enough to power the Odroid and the Xbox Kinect at the same time over a period of time in the fields.

In order to have the Kinect interface with the Odroid running Ubuntu we needed to install open drivers. The only drivers available for linux currently are open source drivers called Libfreenect2. These drivers are not developed by Microsoft (creators of the Xbox Kinect) and they are not made to work exactly the same for every device. Since we are using the Odroid XU4 using Ubuntu there is compatibility issues regarding the method that Protonect (default program for libfreenect) displays the data from the Kinect.

The problem with the default program is that it uses GL which is not supported by Ubuntu running on the Odroid XU4. In order to get around this the driver needed to compiled in a way so that it would not use the GUI. After the driver was compiled without the GUI we were able to tell that it was getting depth data from the Kinect.

We had several challenges with the mobile power. The issues that we faced were caused by both the Kinect and the Odroid normally take 120VAC right out of the wall and convert it to the voltages that each device use independently. The challenge arises from not having a 120VAC power source easily able to become mobile.

The solution that we came up with was to get a power source that will be able to power both devices at the individual post converted DC voltages (12V for the Kinect and 5V for the Odroid). Because the two devices use different voltages and we wanted only one power source we would need to step down the voltages from the source voltage. The source that we chose was 18V so we needed to step down the voltages for both the Kinect and the Odroid. Originally we thought that we would be able to use voltage regulators to step down the voltage for both devices but we soon realized that it would not be feasible to step 20V (max input voltage) down to 5V when the maximum current would be 4A ((20V-5V)*4A = 60W of power dissipated as heat!). We soon realized that a better solution would be to use buck converters in order to step down the voltage. We decided to use 2 different TI simple switcher chips (LM2676-12 and LM2678-5). For some reason still unknown the buck converter for the Kinect would supply the correct voltage with no load but once a load was put on it the voltage would drop and we were not able to fix this issue within a reasonable time. To resolve the issue of the Kinect buck converter not working we decided to use a 12V linear regulator (LM340) to supply the Kinect. The issue with this solution is that the voltage regulator is very inefficient and therefore has a large amount of heat dissipation.

Even after testing all of our output voltages with and without loads (see testing), we had an issue with the Kinect for windows adaptor stop working. We had tested the Kinect voltage regulator with the Kinect using a laptop to make sure we were getting output so we know that the regulator circuit works with the Kinect but for some unknown reason the regulator in the Kinect adaptor no longer works and is shorting the voltage input to ground, causing the voltage regulator to overheat very quickly. This issue may have been caused because the voltage regulator is on a prototype board and something may have shorted out causing the full battery voltage to go to the input of the adaptor or something else may have happened. This error happened very late in the semester and because the adaptor is required to run the Kinect it has put a stop to any further development until a replacement adaptor can be acquired.

Testing

Modeling and Environment

We have constructed small test models for use in our testing of the system. These models have allowed us to test the device in a controlled testing environment. To create these models we used Lego bricks since most bricks' height falls within a fraction of a millimeter from each other making them good for getting a consistent height for each model. The models that ended up being used and that can be seen
in the figures below, are 3 cm in depth, 3 cm in width, and 5 cm in length.

The test environment that was used the most was setting our models at set distances from a center point. We took the height measurement of these models so that we knew that we knew the millimeter height of each one. These models were spaced out from the center starting at 10 cm away and then placing them every 10 cm from that point until we were out of models. These models allowed us to see objects on the ground that we knew were at a specific distance from center and had a known depth.

Software data testing and calibration procedure

To start our tests, we would first have our environment setup how we wanted it and made sure we knew what values to expect by measuring specific points in the environment with a meter stick that could measure in millimeters. We would then position the Kinect so the center point of the IR depth sensors was aligned with our test environment's center point. From this location we would capture our data.

This data was then placed into MatLab to help us see things that would otherwise be hidden in the data had we tried to look at the file without plotting it. From this data we then took slices of it and zoomed in on them to see the depth errors that occurred near the edges of the data. Our assumption is that the error occur because as the three IR sensors get data back there is some error in it and it builds up more as they get further away from their center point. We would place the data slices we took earlier into the Curve Fitting Tool in MatLab and generate a polynomial that would allow us to account for the error in the data. Since the polynomial would change every time based on what is in view of the sensors and what the environment looks like, we decided to base our final polynomial off of a test environment that had a flat surface in all directions. By having this environment, we assumed that a polynomial based off of it should be able to be used as a base for our corrections. Figure 3 shows the curve that we used for the final calibration in the x-axis direction.

We then took this polynomial and placed it in our C# code to allow the program to adjust for the error before creating the csv output file. Since this polynomial was used by capturing x-axis data, we also needed a second polynomial for the y-axis data. This made us realize that if we applied both corrections to a pixel that it would be over-corrected so we decided to average the correction from both polynomials before applying the correction to the depth data. Figure 1 shows what some of our data would look like without any of the calibration and Figure 2 shows what happened to it after correcting the data.

From the data that we had from before the data and after correcting it, we were able to run a Root Mean Square (RMS) function on their differences in MatLab. This returned a one-dimensional array for the RMS of the x-axis values. Then running the mean on this to get the average of this RMS value, we found that for different CSV files that the final RMS value that we got was on average 3.7620.

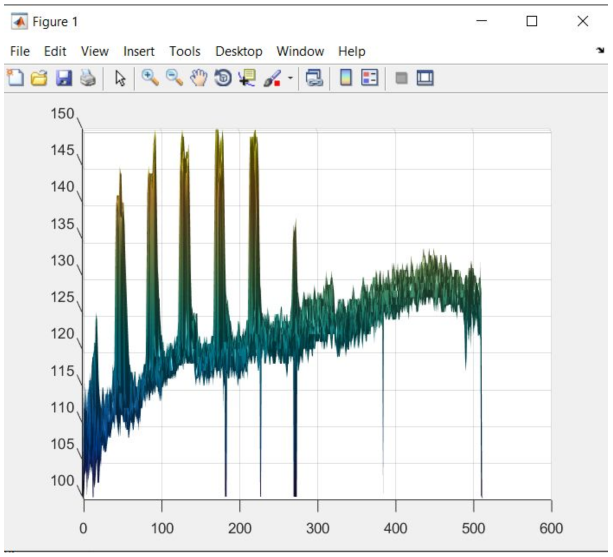


Figure 1 :No correction

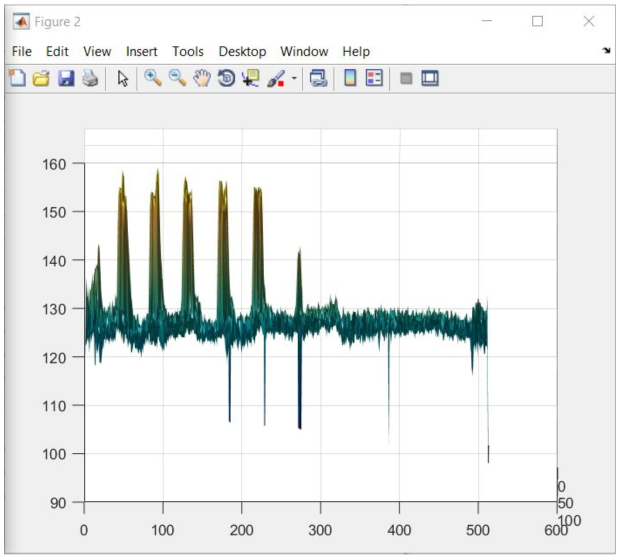


Figure 2: With correction

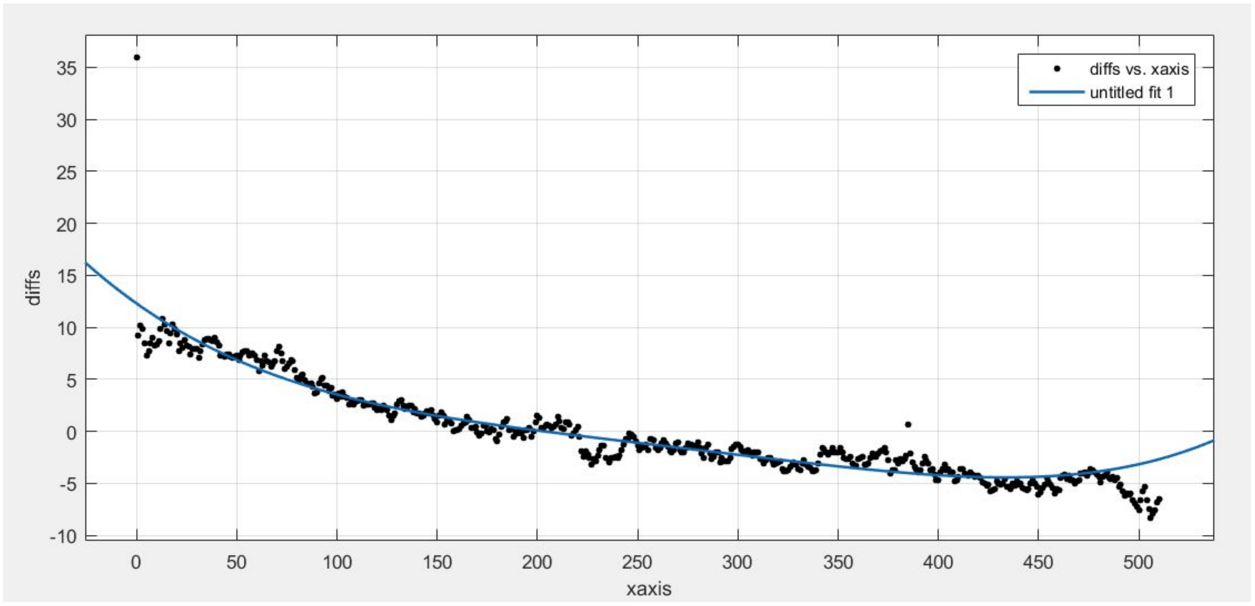


Figure 3: Polynomial fit curve of the data

Hardware Testing

The Odroid and Xbox Kinect both require different voltages and currents to work properly. If the voltage or current is not supplied properly, then the devices can be damaged or might not work properly. In order to ensure that the devices would work properly we needed to make sure that the power supplied to the devices was correct. In order to do this we tested the output of the voltage regulating circuits with and without a load. We used Ohm's law in order to find the appropriate load resistances that would pull the maximum current that the devices can use.

Kinect:
$i_{max} = 1.2A \quad V_{in} = 12V$ so to pull the max power $R_L = 12V/1.2A = 9.6\Omega$

Odroid:
$i_{max} = 4A \quad V_{in} = 5V$ so to pull the max power $R_L = 5V/4A = 1.25\Omega$ (almost a short circuit)

After finding these values we were able to test our circuits with no load voltage (0 A drawn) as well as with a load resistance (able to pull the max current). In order to do the actual testing we used the battery source and tested that the output of the circuits was within the range that is allowed for each device. We tested the voltage at the output of our circuits and made sure that there was not too much ripple voltage on the outputs for both cases, no load and load. We made adjustments to our circuits accordingly in order to correct for "bad" output (e.g too much ripple voltage). After testing our circuits in this way we felt comfortable using them with our devices.

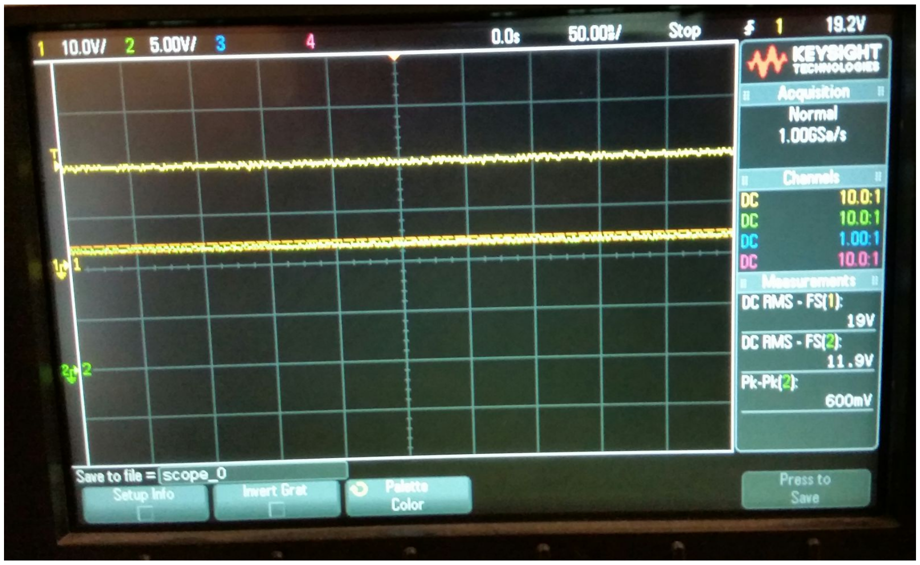


Figure 4: Kinect voltage regulator output showing 19V to 12V

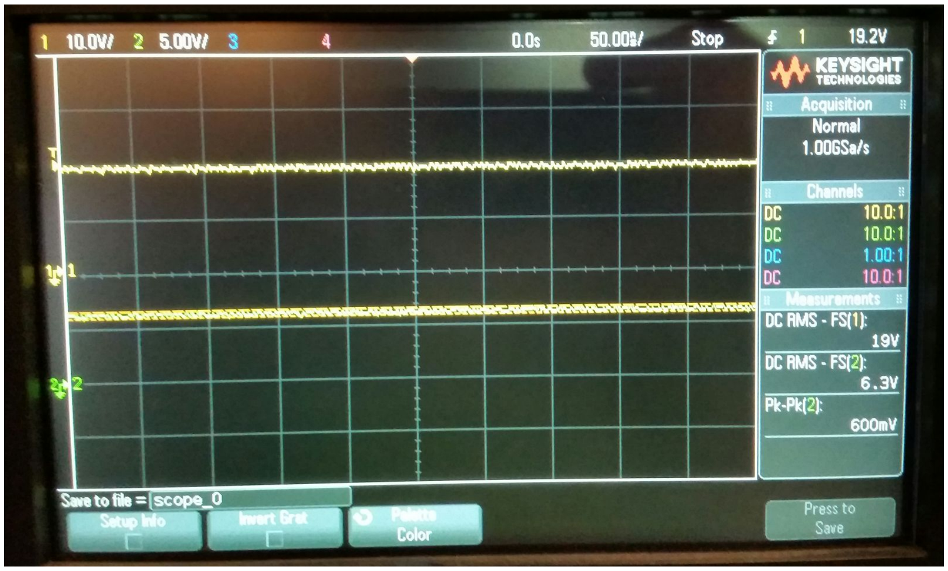
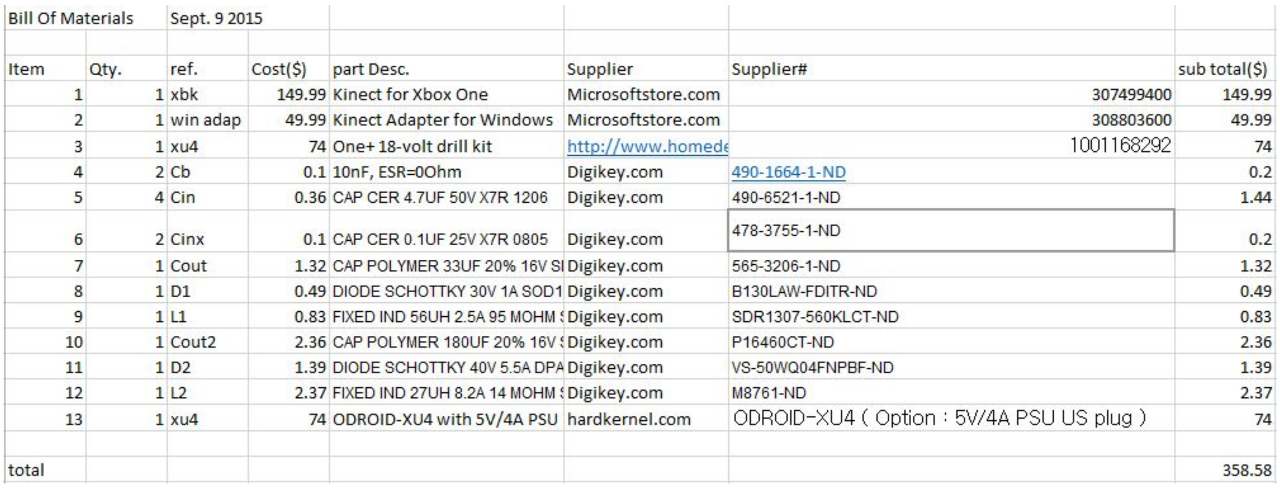


Figure 5: Odroid buck converter output showing 19V to 6.3V (5V with a load)

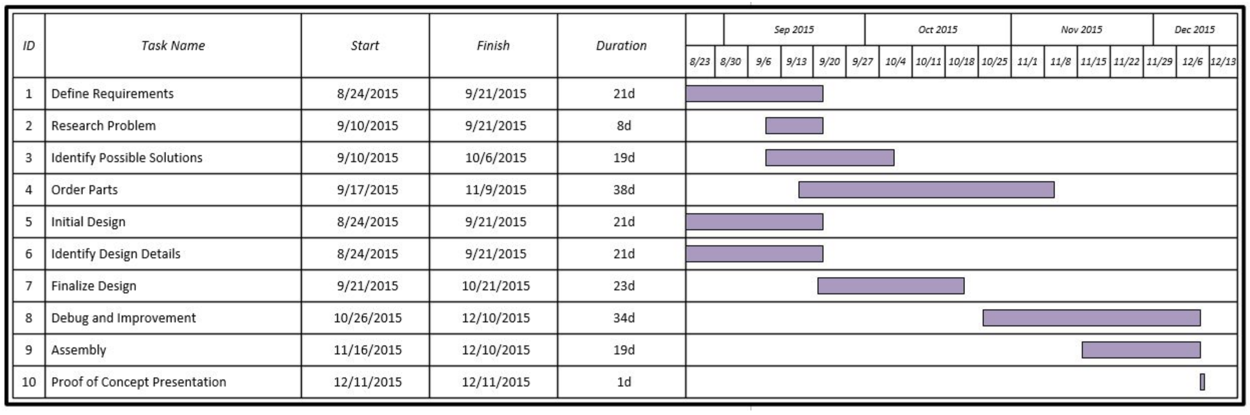
Other documents and figures

Bill of materials



Bill Of Materials		Sept. 9 2015					
Item	Qty.	ref.	Cost($)	part Desc.	Supplier	Supplier#	sub total($)
1	1	xbk	149.99	Kinect for Xbox One	Microsoftstore.com	307499400	149.99
2	1	win adap	49.99	Kinect Adapter for Windows	Microsoftstore.com	308803600	49.99
3	1	xu4	74	One+ 18-volt drill kit	http://www.homede	1001168292	74
4	2	Cb	0.1	10nF, ESR=0Ohm	Digikey.com	490-1664-1-ND	0.2
5	4	Cin	0.36	CAP CER 4.7UF 50V X7R 1206	Digikey.com	490-6521-1-ND	1.44
6	2	Cinx	0.1	CAP CER 0.1UF 25V X7R 0805	Digikey.com	478-3755-1-ND	0.2
7	1	Cout	1.32	CAP POLYMER 33UF 20% 16V SI	Digikey.com	565-3206-1-ND	1.32
8	1	D1	0.49	DIODE SCHOTTKY 30V 1A SOD1	Digikey.com	B130LAW-FDITR-ND	0.49
9	1	L1	0.83	FIXED IND 56UH 2.5A 95 MOHM	Digikey.com	SDR1307-560KLCT-ND	0.83
10	1	Cout2	2.36	CAP POLYMER 180UF 20% 16V	Digikey.com	P16460CT-ND	2.36
11	1	D2	1.39	DIODE SCHOTTKY 40V 5.5A DPA	Digikey.com	VS-50WQ04FNPBF-ND	1.39
12	1	L2	2.37	FIXED IND 27UH 8.2A 14 MOHM	Digikey.com	M8761-ND	2.37
13	1	xu4	74	ODROID-XU4 with 5V/4A PSU	hardkernel.com	ODROID-XU4 (Option : 5V/4A PSU US plug)	74
total							358.58

Gantt chart (timeline of project design)



ID	Task Name	Start	Finish	Duration
1	Define Requirements	8/24/2015	9/21/2015	21d
2	Research Problem	9/10/2015	9/21/2015	8d
3	Identify Possible Solutions	9/10/2015	10/6/2015	19d
4	Order Parts	9/17/2015	11/9/2015	38d
5	Initial Design	8/24/2015	9/21/2015	21d
6	Identify Design Details	8/24/2015	9/21/2015	21d
7	Finalize Design	9/21/2015	10/21/2015	23d
8	Debug and Improvement	10/26/2015	12/10/2015	34d
9	Assembly	11/16/2015	12/10/2015	19d
10	Proof of Concept Presentation	12/11/2015	12/11/2015	1d

Appendix I : Operation Manual

Mobile setup using Odroid

- Connect the Xbox Kinect to the top of the power drill case and place the Odroid inside the case.

- From here plug the Kinect's USB 3.0 into the Odroid.

- Then plug in the USB drive that you wish to save the files to into another of the Odroid's USB slots.

- Slide a charged drill battery into the battery slot of the drill case.

- Power on the Odroid.

- The C program should open and start running after the Odroid starts.

- When the program has started, you are ready to use the Kinect.

- The next step is to aim the Kinect at the location that you want to get data from.

- From here you need to pull the drill trigger.
    - This will tell the C program to start the Kinect and save the data. Allow a couple seconds for the Odroid to process the data.

- When it is done, the information about the reliability of the image will be displayed on the Odroid's screen.
    - If the data is acceptable, that's great!
    - If the data is too unreliable, then try lowering/raising the system towards/away from the ground and pull the trigger again to take another data sample.

- To retrieve the data from the system, power down the Odroid and then remove the USB drive.
    - The data should be stored inside that drive for use later.

Alternate setup using laptop

- Our system has an alternate setup for getting the Kinect to work.

- For this setup the user will need to download the Windows Kinect SDK v2 onto a laptop or desktop computer running Windows 8 or later and have a folder named KinectData located at C:\KinectData\.
  - This will install the necessary libraries and resources that our program will need to run and allow the program to save to the correct location.

- Then when that is done they will need to have the DepthBasics-WPF.exe file that has been created by our C# program from Visual Studio or they can download our source code and open the project in Visual Studio.
  - There is a Github link below and also on our team website for our source code.

- Run the program DepthBasics-WPF.exe.
  - If the SDK is installed correctly, then this executable should display a window that has another display window in the center of it as well as a button in the bottom right corner.
  - The inner display will show as black fn the Kinect is not already connected to a USB 3.0 slot on the computer, otherwise, it will display what the Kinect sensors are seeing.
    - This will help the user see what the data will show and will allow them to see the dead zones.

- The User should then position the Kinect over the area that they are wanting to get data from.
  - We have attached a bubble level to the back of the Kinect that should help with keeping the IR sensors level with the ground.
  - Also keep in mind that it is very possible that you might get your feet in the image.

- When the user is happy with the data seen in the inner display window, they can press the button to capture the data and save the information to a location on their computer.
  - The window will tell the user where the files are saved to in the bottom left corner of the main display window.

Appendix II: Older versions

Initial Development Board

Our initial design was the same as the current design except it included using a Raspberry Pi development board in place of the Odroid XU4. When we were originally reading up on development boards that we could use, the Raspberry Pi seemed like the best choice because of low power use and it had plenty of USB ports. Unfortunately this design was scrapped because after attempting to get working Kinect drivers on the Raspberry Pi it was found that it couldn’t support the Kinect because it lacked a USB 3.0 slot which the Kinect required for data transfer.

Initial Power Supply

Another one of our original design choices was to use an uninterruptible power supply for supplying power to the system. This was supposed to make the system mobile. We scrapped this design because we couldn’t get one that would support the system like we wanted it to and we couldn’t find one that was lightweight. Hence we changed the design to use a power drill battery instead.

Appendix III: Other Considerations

Future iterations

The major thing for the next iteration would be to get the buck converters for the Odroid and Kinect to work correctly so that we can further test the two programs running together on a more mobile platform. We discovered too late that the buck converters weren't supplying the power correctly and couldn't get them working so we opted for a simpler form and used a voltage regulator so that the Kinect could still use a mobile drill battery and work without a wall plugin allowing users to use their laptop in the field.

Another big part of a future iteration would be to complete the C program that would start the C# program. This wasn't able to be finished because without the Kinect and Odroid running on the same power we were unable to finish what was needed. Also because the Kinect adapter stopped working so late, it just added another obstacle in the way of getting this accomplished.

The next iteration could also work on improving the accuracy of the data. The data is currently being corrected but there is still a chance that some of the error is getting through.

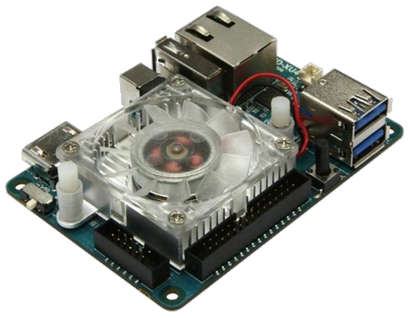
Technology used

Visual Studio - This is an IDE for C based languages. We did all of the programming for the C# Kinect program using Visual Studio since the original demo was a Visual Studio solution.

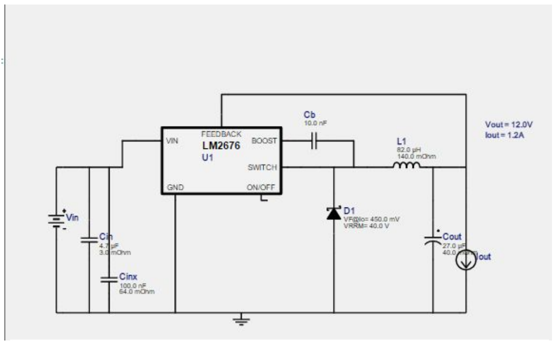
MatLab - This is a great tool for engineers that allows the user to view and make graphs, find curve fits, and it has many more applications. We used this application to help see things that were hiding in our CSV file data. We later used it to find our polynomials that corrected the depth data.

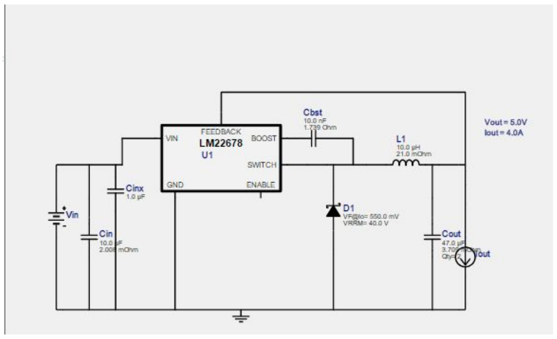
Kinect - This is an accessory to the Microsoft Xbox One. We acquired the Windows adapter to allow us to connect it to our laptops and debug programs. The Kinect is capable of capturing RGB, IR, and Depth data using its sensors. The ones we cared about were the IR and Depth sensors. For the Kinect to find depth data, it blasts out IR beams from the IR blasters at the front of it. The depth sensors then read the IR points as they come back to the Kinect.

Odroid XU4 - This is a development board made by Hardkernel Co., Ltd. This board is loaded with Ubuntu Linux and has the necessary power and ports to support the Kinect depth sensors. It was loaded with libfreenect2 and mono so that we could eventually work on the Kinect from this board.



TI simple switchers/buck converters - This is a circuit design that steps down the voltage input into a desired voltage output. By utilizing simple switcher circuits, we can take the initial voltage source, split the source into two separate outputs and step it down to two different voltage outputs such that each voltage output can power up different hardware.





Kobalt power drill - The drill was removed because we only wanted the chassis and its power source. Since our system needs to run on portable power the drill's battery and wiring were used to supply the needed power.

What we learned

Start early and expect delays

We had some misfortune at the beginning of the semester that put us back by a couple weeks. Then some of our parts didn't get in as soon as we would have liked pushing us back farther.

Things never go as planned

For getting the Kinect's drivers on the Odroid we used the open source project Libfreenect2. When researching this project it was said that the Odroid XU4 that we used would work without a problem but when we actually went to install the drivers, we ran into problems with the Odroid and it's display drivers not working for the standard version of Libfreenect2. We eventually got them working after researching our problems more.

Don't be afraid to ask questions

At first we were very scared and unsure of what to ask people for where to get started on this project. Getting our advisor Josh Bertram really helped as he is an industry professional who has done this sort of work before and got us on the right track. We also asked the CPR E parts department questions about our buck converters and power needs that really helped us.

Lock up your stuff

This was the big misfortune at the beginning of the semester. We discovered that our original system as well as some other teams' work had been stolen over the summer so we had to get all of our parts again. This did allow us to take another look at what was working and what wasn't working and only get things that had worked but also set us back several weeks as we waited for parts.

Appendix IV: Code

Our project is on Github: https://github.com/dmcdowell30/492KinectDepth.git

C#: DepthBasics-WPF

```
//------------------------------------------------------------------------------
// <copyright file="MainWindow.xaml.cs" company="Microsoft">
//     Copyright (c) Microsoft Corporation.  All rights reserved.
// </copyright>
//------------------------------------------------------------------------------

namespace Microsoft.Samples.Kinect.DepthBasics
{
    using System;
    using System.ComponentModel;
    using System.Diagnostics;
    using System.Globalization;
    using System.IO;
    using System.Windows;
    using System.Windows.Media;
    using System.Windows.Media.Imaging;
    using Microsoft.Kinect;

    /// <summary>
    /// Interaction logic for MainWindow
    /// </summary>
    public partial class MainWindow : Window, INotifyPropertyChanged
    {
        /// <summary>
        /// Map depth range to byte range
        /// </summary>
        private const int MapDepthToByte = 1;

        private double theUntouchables = 0;

        /// <summary>
        /// Active Kinect sensor
        /// </summary>
        private KinectSensor kinectSensor = null;

        /// <summary>
        /// Reader for depth frames
        /// </summary>
        private DepthFrameReader depthFrameReader = null;
        /// <summary>
        /// Description of the data contained in the depth frame
        /// </summary>
        private FrameDescription depthFrameDescription = null;

        /// <summary>
        /// Bitmap to display
        /// </summary>
        private WriteableBitmap depthBitmap = null;

        /// <summary>
        /// Intermediate storage for frame data converted to color
        /// </summary>
        private byte[] depthPixels = null;
        private ushort[] depthStuff = null;

        /// <summary>
        /// Current status text to display
        /// </summary>
        private string statusText = null;

        /// <summary>
        /// Initializes a new instance of the MainWindow class.
        /// </summary>
        public MainWindow()
        {
          // get the kinectSensor object
          this.kinectSensor = KinectSensor.GetDefault();

          // open the reader for the depth frames
          this.depthFrameReader = this.kinectSensor.DepthFrameSource.OpenReader();

          // wire handler for frame arrival
          this.depthFrameReader.FrameArrived += this.Reader_FrameArrived;

          // get FrameDescription from DepthFrameSource
          this.depthFrameDescription = this.kinectSensor.DepthFrameSource.FrameDescription;

          // allocate space to put the pixels being received and converted
          this.depthPixels = new byte[this.depthFrameDescription.Width * this.depthFrameDescription.Height];
          this.depthStuff = new ushort[this.depthFrameDescription.Width * this.depthFrameDescription.Height];

          // create the bitmap to display
          this.depthBitmap = new WriteableBitmap(this.depthFrameDescription.Width,
this.depthFrameDescription.Height, 96.0, 96.0, PixelFormats.Gray8, null);

          // set IsAvailableChanged event notifier
```

```
        this.kinectSensor.IsAvailableChanged += this.Sensor_IsAvailableChanged;

        // open the sensor
        this.kinectSensor.Open();

        // set the status text
        this.StatusText = this.kinectSensor.IsAvailable ? Properties.Resources.RunningStatusText
                                                        : Properties.Resources.NoSensorStatusText;

        // use the window object as the view model in this simple example
        this.DataContext = this;

        // initialize the components (controls) of the window
        this.InitializeComponent();
    }

    /// <summary>
    /// INotifyPropertyChangedPropertyChanged event to allow window controls to bind to changeable data
    /// </summary>
    public event PropertyChangedEventHandler PropertyChanged;

    /// <summary>
    /// Gets the bitmap to display
    /// </summary>
    public ImageSource ImageSource
    {
        get
        {
            return this.depthBitmap;
        }
    }

    /// <summary>
    /// Gets or sets the current status text to display
    /// </summary>
    public string StatusText
    {
        get
        {
            return this.statusText;
        }

        set
        {
            if (this.statusText != value)
            {
                this.statusText = value;
                // notify any bound elements that the text has changed
                if (this.PropertyChanged != null)
                {
                    this.PropertyChanged(this, new PropertyChangedEventArgs("StatusText"));
                }
            }
        }
    }

    /// <summary>
    /// Execute shutdown tasks
    /// </summary>
    /// <param name="sender">object sending the event</param>
    /// <param name="e">event arguments</param>
    private void MainWindow_Closing(object sender, CancelEventArgs e)
    {
        if (this.depthFrameReader != null)
        {
            // DepthFrameReader is IDisposable
            this.depthFrameReader.Dispose();
            this.depthFrameReader = null;
        }

        if (this.kinectSensor != null)
        {
            this.kinectSensor.Close();
            this.kinectSensor = null;
        }
    }

    /// <summary>
    /// Handles the user clicking on the screenshot button
    /// </summary>
    /// <param name="sender">object sending the event</param>
    /// <param name="e">event arguments</param>
    private void ScreenshotButton_Click(object sender, RoutedEventArgs e)
    {

        string time = System.DateTime.UtcNow.ToString("hh'-'mm'-'ss",
CultureInfo.CurrentUICulture.DateTimeFormat);

        try
        {
            double[,] pixels = new double[this.depthFrameDescription.Width, this.depthFrameDescription.Height];
            double[,] correctionMatrix = new double[this.depthFrameDescription.Width,
this.depthFrameDescription.Height];

            int i = 0;
int j = 0;
int k = depthPixels.Length - 1;

for (i = this.depthFrameDescription.Height - 1; i >= 0; i--)
{
   for (j = 0; j < this.depthFrameDescription.Width; j++)
   {
      pixels[j, i] = this.depthStuff[k];
      double pix = pixels[j, i];

      if (depthStuff[k] == 0)
      {
         theUntouchables++;
      }

      k--;
   }
}

int c, d;

//Correct for the x-axis
for (c = 0; c < depthFrameDescription.Width; c++)
{
   for (d = 0; d < depthFrameDescription.Height; d++)
   {
      /*
      f(x) = p1*x^5 + p2*x^4 + p3*x^3 + p4*x^2 + p5*x + p6
      Coefficients (with 95% confidence bounds):
      p1 =   5.237e-12  (3.014e-12, 7.459e-12)
      p2 =  -7.065e-09  (-9.924e-09, -4.207e-09)
      p3 =   3.218e-06  (1.89e-06, 4.545e-06)
      p4 =  -0.0004947  (-0.0007625, -0.000227)
      p5 =    -0.03215  (-0.05422, -0.01009)
      p6 =       9.665  (9.107, 10.22)
      */

      double correction = 5.237e-12 * Math.Pow(c, 5) +
         -7.065e-09 * Math.Pow(c, 4) +
         3.218e-06 * Math.Pow(c, 3) +
         (-0.0004947) * Math.Pow(c, 2) +
         (-0.03215) * c +
         9.665;

      correctionMatrix[c, d] = correction;
      double pix = pixels[c, d];
   }
}
```

```
//Correct for the y-axis
for (d = 0; d < depthFrameDescription.Height; d++)
{
   for (c = 0; c < depthFrameDescription.Width; c++)
   {
      /*
      Linear model Poly4:
      f4(x) = p1 * x ^ 4 + p2 * x ^ 3 + p3 * x ^ 2 + p4 * x + p5
      Coefficients(with 95 % confidence bounds):
      p1 = 2.085e-09(1.394e-09, 2.776e-09)
      p2 = -2.043e-06(-2.756e-06, -1.33e-06)
      p3 = 0.000717(0.0004741, 0.0009599)
      p4 = -0.1262(-0.1567, -0.09566)
      p5 = 9.532(8.406, 10.66)
      */
      double correction = 2.085e-09 * Math.Pow(d, 4) +
         -2.043e-06 * Math.Pow(d, 3) +
         0.000717 * Math.Pow(d, 2) +
         -0.1262 * Math.Pow(d, 1) +
         9.532;

      double avg = correctionMatrix[c, d];
      double pix = pixels[c, d];

      avg += correction;
      avg /= 2;

      correctionMatrix[c, d] = correction;
   }
}

//Apply the correction matrix to the depth data
for (c = 0; c < depthFrameDescription.Width; c++)
{
   for (d = 0; d < depthFrameDescription.Height; d++)
   {
      double abc = pixels[c, d];
      abc += correctionMatrix[c, d];
      if (abc < 0) { abc = 0; }
      pixels[c, d] = abc;
   }

}

//Create the .CSV file
double csvValue = 0.0;
```

```
                using (StreamWriter outfile = new StreamWriter("C:\\KinectData\\KinectScreenshot-Depth-" + time +
"-Output.csv"))
                {
                    for (int t = 0; t < this.depthFrameDescription.Height; t++)
                    {
                        string content = "";
                        for (int s = this.depthFrameDescription.Width - 1; s >= 0; s--)
                        {
                            csvValue = (pixels[s, t]);
                            content += csvValue + ",";
                        }
                        outfile.WriteLine(content);
                    }
                }

                //Create the information text file
                double kinectX = (pixels[depthFrameDescription.Width/2, depthFrameDescription.Height/2] +
                    pixels[(depthFrameDescription.Width/2)+1, depthFrameDescription.Height/2] +
                    pixels[depthFrameDescription.Width/2, (depthFrameDescription.Height/2)+1] +
                    pixels[(depthFrameDescription.Width/2)+1, (depthFrameDescription.Height/2)+1]) / 4;

                using (StreamWriter file = new StreamWriter("C:\\KinectData\\KinectScreenshot-Depth-" + time +
"-Output.txt"))
                {

                    file.Write("Fraction unreliable: " + theUntouchables + " / " + (depthFrameDescription.Height *
depthFrameDescription.Width) + "\n");
                    file.Write("Percentage unreliable: " + (theUntouchables / (depthFrameDescription.Height *
depthFrameDescription.Width)) * 100 + "%\n");
                    file.Write("Center distance: " + kinectX + "\n");
                    file.Write("\n");
                }
            }
            catch (Exception Ex)
            {
                Console.WriteLine(Ex.ToString());
            }

            if (this.depthBitmap != null)
            {
                // create a png bitmap encoder which knows how to save a .png file
                BitmapEncoder encoder = new PngBitmapEncoder();

                // create frame from the writable bitmap and add to encoder
                encoder.Frames.Add(BitmapFrame.Create(this.depthBitmap));

                string path = "C:\\KinectData\\KinectScreenshot-Depth-" + time + "-Image.png";
            // write the new file to disk
            try
            {
               // FileStream is IDisposable
               using (FileStream fs = new FileStream(path, FileMode.Create))
               {
                  encoder.Save(fs);
               }

               this.StatusText = string.Format(CultureInfo.CurrentCulture,
Properties.Resources.SavedScreenshotStatusTextFormat, path);
            }
            catch (IOException)
            {
               this.StatusText = string.Format(CultureInfo.CurrentCulture,
Properties.Resources.FailedScreenshotStatusTextFormat, path);
            }
         }
      }

      /// <summary>
      /// Handles the depth frame data arriving from the sensor
      /// </summary>
      /// <param name="sender">object sending the event</param>
      /// <param name="e">event arguments</param>
      private void Reader_FrameArrived(object sender, DepthFrameArrivedEventArgs e)
      {
         bool depthFrameProcessed = false;

         using (DepthFrame depthFrame = e.FrameReference.AcquireFrame())
         {
            if (depthFrame != null)
            {
               depthFrame.CopyFrameDataToArray(depthStuff);
               // the fastest way to process the body index data is to directly access
               // the underlying buffer
               using (Microsoft.Kinect.KinectBuffer depthBuffer = depthFrame.LockImageBuffer())
               {
                  // verify data and write the color data to the display bitmap
                  if (((this.depthFrameDescription.Width * this.depthFrameDescription.Height) == (depthBuffer.Size /
this.depthFrameDescription.BytesPerPixel)) &&
                     (this.depthFrameDescription.Width == this.depthBitmap.PixelWidth) &&
(this.depthFrameDescription.Height == this.depthBitmap.PixelHeight))
                  {
                     // Note: In order to see the full range of depth (including the less reliable far field depth)
                     // we are setting maxDepth to the extreme potential depth threshold
                     ushort maxDepth = ushort.MaxValue;
```

```
                    // If you wish to filter by reliable depth distance, uncomment the following line:
                    //// maxDepth = depthFrame.DepthMaxReliableDistance

                    this.ProcessDepthFrameData(depthBuffer.UnderlyingBuffer, depthBuffer.Size,
depthFrame.DepthMinReliableDistance, maxDepth);
                    depthFrameProcessed = true;
                  }
                }
              }
            }

            if (depthFrameProcessed)
            {
              this.RenderDepthPixels();
            }
          }

          /// <summary>
          /// Directly accesses the underlying image buffer of the DepthFrame to
          /// create a displayable bitmap.
          /// This function requires the /unsafe compiler option as we make use of direct
          /// access to the native memory pointed to by the depthFrameData pointer.
          /// </summary>
          /// <param name="depthFrameData">Pointer to the DepthFrame image data</param>
          /// <param name="depthFrameDataSize">Size of the DepthFrame image data</param>
          /// <param name="minDepth">The minimum reliable depth value for the frame</param>
          /// <param name="maxDepth">The maximum reliable depth value for the frame</param>
          private unsafe void ProcessDepthFrameData(IntPtr depthFrameData, uint depthFrameDataSize, ushort
minDepth, ushort maxDepth)
          {
            // depth frame data is a 16 bit value
            ushort* frameData = (ushort*)depthFrameData;

            // convert depth to a visual representation
            for (int i = 0; i < (int)(depthFrameDataSize / this.depthFrameDescription.BytesPerPixel); ++i)
            {
              // Get the depth for this pixel
              ushort depth = frameData[i];

              // To convert to a byte, we're mapping the depth value to the byte range.
              // Values outside the reliable depth range are mapped to 0 (black).
              this.depthPixels[i] = (byte)(depth >= minDepth && depth <= maxDepth ? (depth / MapDepthToByte) : 0);
            }
          }

          /// <summary>
          /// Renders color pixels into the writeableBitmap.
          /// </summary>
```

```
        private void RenderDepthPixels()
        {
            this.depthBitmap.WritePixels(
                new Int32Rect(0, 0, this.depthBitmap.PixelWidth, this.depthBitmap.PixelHeight),
                this.depthPixels,
                this.depthBitmap.PixelWidth,
                0);
        }

        /// <summary>
        /// Handles the event which the sensor becomes unavailable (E.g. paused, closed, unplugged).
        /// </summary>
        /// <param name="sender">object sending the event</param>
        /// <param name="e">event arguments</param>
        private void Sensor_IsAvailableChanged(object sender, IsAvailableChangedEventArgs e)
        {
            // on failure, set the status text
            this.StatusText = this.kinectSensor.IsAvailable ? Properties.Resources.RunningStatusText
                                                            : Properties.Resources.SensorNotAvailableStatusText;
        }
    }
}
```

MatLab - CalibrationTests.m

```
%Change the file name based on the what output file you are testing
M = csvread( 'C:\KinectData\KinectScreenshot-Depth-06-38-37-Output.csv' )
N=1000-M
O=N
O(O>400)=0
O(O<-875)=0
P=O(200:250,1:end)
P(P<100)=100

slice=P(1:15,1:end)

line=mean(slice)

diffs = mean(line) - line
plot(diffs)

xaxis=(0:size(P,2)-1)

%f is for the x direction, f4 is for the y direction
%plot(diffs); hold; plot(f4); hold;
```

```
%coeffs=[f4.p1 f4.p2 f4.p3 f4.p4 f4.p5]
plot(diffs); hold; plot(f); hold;
coeffs=[f.p1 f.p2 f.p3 f.p4 f.p5 f.p6]

x=0:size(P,2)-1

polyvector=polyval(coeffs,x)

correction=repmat(polyvector,51,1)

altered=P+correction

surf(P, 'FaceColor', 'interp', 'EdgeColor', 'none', 'FaceLighting', 'gouraud')
camlight left

%find the root mean square error for the output
e=mean(rms(altered-P))
d = (e + 3.2930) / 2
```

C - CProg.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

int main() {
	int a;

	while(1) {

		scanf("%d", &a); //Replace with checking GPIO pin

		if(a >= 0){

			pid_t childPID;
			childPID = fork();

			if(childPID >= 0) {
				if(childPID == 0) {
					int maxLineLength = 128;
					char* lineBuffer = (char*)malloc(sizeof(char) * maxLineLength);
					if(lineBuffer == NULL) {
						printf("malloc failed for lineBuffer");
						exit(1);
```

```
                    }

                    FILE * file;
                    file = fopen("test.txt", "r");
                    if(file == NULL) {
                        printf("open file failed");
                        exit(1);
                    }

                    char ch = getc(file);
                    int count = 0;

                    while(ch != EOF) {
                        if(count == maxLineLength) {
                            maxLineLength += 128;
                            lineBuffer = realloc(lineBuffer, maxLineLength);
                            if(lineBuffer == NULL) {
                                printf("failed to realloc for lineBuffer");
                                exit(1);
                            }
                        }
                        lineBuffer[count] = ch;
                        count++;

                        ch = getc(file);
                    }

                    lineBuffer[count] = '\0';
                    char line[count+1];
                    strncpy(line, lineBuffer, (count+1));
                    free(lineBuffer);
                    printf("%s\n", line);

                    //TODO place odroid show screen printing here

                    fclose(file);
                }
            }
        }
    }

    return 0;
}
```