The 15th Winona Computer Science Undergraduate Research Symposium

April 28, 2015

11:00am to 1:00pm

Watkins 105

Winona State University
Winona, MN

Sponsored by the Department of Computer Science at
Winona State University

WINONA
STATE UNIVERSITY
Computer Science Department
http://cs.winona.edu

Table of Contents

Light Sensor Performance Comparisons

Jason Carpenter
Computer Science Department
Winona State University
Winona, MN 55987

JCarpenter11@winona.edu

ABSTRACT

This paper gives a consumer focused review of several commercial grade light sensors: A Photo resistor, UV/IR/Visible Light(SI1145), High Dynamic Range LUX(TSL2591), Digital Luminosity LUX(TSL2561), and an Analog Light(GA1A1S202WP). Many consumer reviews for sensors exist and continue to grow, as the market expands in the area of home automation. This paper examined a subset of sensors for a specific micro controller and assess their qualities with an approximation of what an average consumer may use them for, in an attempt to create a more consumer friendly quality assessment. The assessment is based on many traits present in the sensors, one such quality is light sensitivity. The assessment points to the Analog sensor as preferable for basic level applications.

Keywords

Light Sensing, Light sensors, Light spectrum.

1. INTRODUCTION

In the modern digital age, home-automation with small micro-computers is a common sight. Many of these devices operate using basic sensors, for example, light sensors. A light sensor is a device that responds to changes in light. Most detect light changes through changes in voltage coming through the circuit. Light hits the sensor, and the photosensitive materials change the resistance offered by the sensor. This change in resistance can be coded to represent data. Light sensors come in a variety of types, ranging from UV capable, analog based, digital based, simple, and complex. Light Sensors are present in light fixtures, burglar alarms, garage doors, solar panels, and various other devices and applications [8-12]. Advance applications of light sensors such as, fiber optic cables, optical computers, wireless direct-line-of-sight devices, and bar code scanners have been around for several decades. The versatility of light makes sensors that use it popular with home-automation individuals. As a result, many light sensors have come into the market and finding the best one for a job may be difficult. One such light sensor is the photo resistor. It is a device that costs less than $1.00 that decreases electrical resistance when sensing light [1]. It is simple in all respects: cost, complexity, and light spectrum capability. Some sites like AdaFruit.com, provide detailed specifications for each sensor. These traits are esoteric and may not well indicate the quality of the sensor in an approachable way. Price may not always indicate quality, and you may make an unnecessary expenditure. Taking more variables into account, like implementation complexity, library dependency, and traits not stated in the manufacturer's specs can give consumers a more complete assessment of the light sensor from a consumer prospective. My research gives manufacturers better insight into what consumers need or want. It is common for researchers to tackle this type of project, but uncommon to attempt to address it from a beginner perspective [7-8].

2. Hypothesis

The Photo resistor, based on traits compared among a subset of selected sensors, is the highest performing light sensor, given the quality criteria.

Using an assessment based on cost, complexity, light capabilities, and other traits, among the subset of selected sensors, we determined the quality criteria of the sensor. The Photo resistor was the control due to its simplicity.

3. Methodology

3.1 Sensors

We obtained a set of light sensors from the distributor Adafruit.com: The Photo resistor, UV/IR/Visible Light(SI1145), High Dynamic Range LUX(TSL2591), Digital Luminosity LUX(TSL2561), and Analog Light(GA1A1S202WP).

3.2 Sensor Specifications

The sensor manufacturers provide data on each sensor. Temperature range, dynamic range, voltage range, are a few examples. Information used for assessment from manufacturers is recorded in Table 1. This was recorded and factored into the later quality criteria calculation. Some information present in the manufacturer specs is not present because of the beginner perspective limitation. Information sampled with our testing methodology is present in the documentation for each sensor already. This test retested these in an effort to standardize and simplify the data for the lower level perspective.

3.3 Sensor Testing

For the quality assessment, there are two important variables for a standard light sensing application: light turnover and light sensitivity range. Both of these specs are provided by the manufacturers, but retested to simplify and standardize the environment variables. To approximate a common application of a light sensor, we build a simple light mount and sensor holder (Figure 2), the process of constructing this mount, and programing the sensor processor, gives us the same steps a common user may follow. This helps build a consumer perspective: purchase, receive, prepare sensor, implement hardware, program for hardware, and finally run.

We constructed our light mount's light sensor processor using an Arduino Uno microcontroller, purchased from AdaFruit.com (Figure 1). We used a White LED inside a (Samsung Galaxy S4 Assistive light) for our light source. We used an algorithm written specifically for this project for assessing light values. The algorithm attempted to correct for ambient light values by first taking ten samples of ambient light and then using the average of those lights to adjust the incoming sensor value before actual sensing begins. We used the Arduino to sample a light value from the sensor once every .0001 seconds. The light range sensitivity was measured by how high and low the sensor value changes with respect to a light. Light turnover was measured by the speed of the sensor adjusting to a changed light value. Simply, start with a light off, turn it on, and measure the time it takes to reach the high. All the sensors were implemented using their defaults or minimum required setup process.

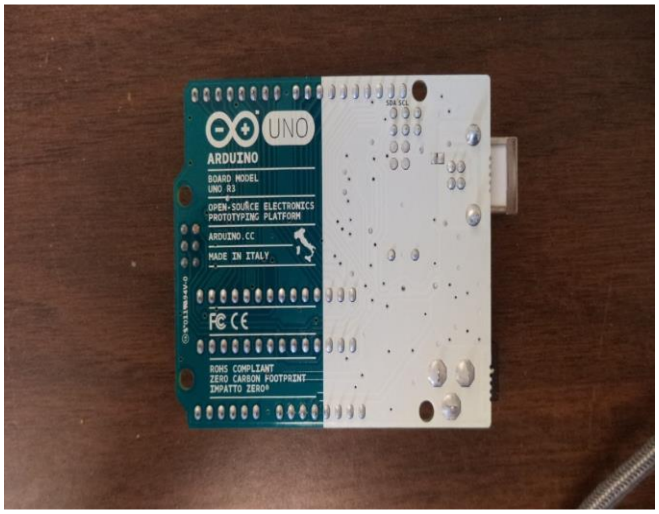


Figure 1. Arduino Microcontroller, Arduino Uno.

Starting with the photo resistor as the control (For it is the simplest sensor), we test each sensor against it. The apparatus indicated in figure 2 acted as a stand in for a generic sensor application a common person may use. Measuring light intensity readings and responding to rapid changes in intensity can indicates the capability of the sensor. This process was repeated a number of times for each sensor.

Process

1. Install Light Sensor into apparatus
2. Collect light range
3. Calculate light turnover
4. Repeat for each Sensor

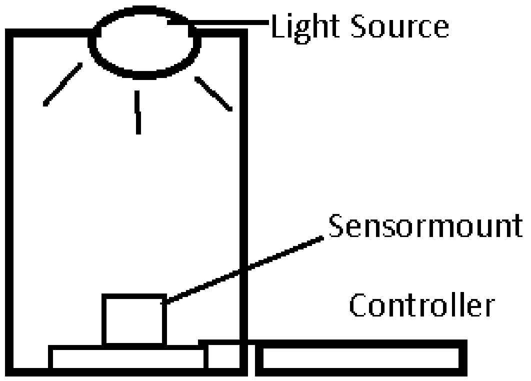


Figure 2: Test Apparatus Diagram.

3.4 Quality Criteria

We took the results from our test and the values from the constants together and created the assessment of a sensor's quality. The quality factor assumed that spectrum range, cheapness, sensitivity range, light turnover, command read accuracy, and simplicity of sensor implementation was regarded as ideal. The final number calculated was an aggregate of all these traits, rated, and summed on a ranking 0-5, 0 is not ideal, 5 is ideal. The following are some definitions of what is ideal in a sensor from our set quality assessment. Explanations for each rating was given in the analysis, much of this was relative to the sensors.

3.5 Quality Criteria Definitions

Spectrum Range: Quality was be measured in how wide the sensible spectrum of light is. *Price*: The lower the price the more ideal. *Simplicity of Implementation*: How difficult is setting up the sensor, wiring it up, fitting it to the devices, and implementing code for it. Less steps of implementation and a smaller easier to manipulate are ideal. This is the least empirical trait, the ratings for this trait was accompanied by an explanation. The basic assessment looked at number of steps to go from receiving the device to receiving values from it. *Sensitivity Range*: How much the sensor reacts to light changes. A wider range indicated a greater degree of sensitivity. Each sensor measures light differently, the Photo Resistor measures in the change in voltage, the Arduino maps the voltage output as given by the Arduino page as, 5 Volts/1024 units yielding a .0049 Volts/unit [6], the LUX sensors output a lux value which is a fairly complex unit of light intensity over an area; the IR/UV/VS sensor gives a unit less value for visible light; the analog sensor operates similarly to the Photo resistor, but it can be converted to LUX given the log-scale nature of the resistor. This trait was analyzed separately due to the difference in units of measurement. *Light Turnover*: How fast does the sensor detect change when the light changes. The Arduino can read in an analog voltage at a rate of .1 ms [6]. The formula for calculating the rise/fall values are stated in formula 1. Some of the sensors may need to be sampled at different rates, the actual numbers for the turnover calculation is included in the data table.

Formula 1: (.0001 second + sample rate (.0001 second) * number of samples). The Arduino's input speed plus the sample rate times the number of samples that indicate a light change [15-16].

4. Results and Analysis

These explanations are the basis for the scores given to the sensor traits. If a consumer places a greater emphasis on a certain trait apply a multiplier.

Spectrum Range

The Photo Resistor is not sensitive to other parts of the visible light spectrum, compared to the other sensors it can give the least responsivity to incoming light variations. It has only a 300 nm range. This is the rationale for the lower 1 rank. The IR sensor has the widest useable range of all the sensors, most of them max out near the edges of visible light. This sensor can read above and below the standard visible spectrum quite far. The LUX(TSL2561) sensor has a comparable range to the photo resistor, but has a much more sensitive intensity readout. This can detect varying degrees of light intensity much better than the photo resistor. The range on the LUX(TSL2591) is comparable to the other LUX capable sensor (TSL2561). It access primarily the visible light spectrum with little overlap in the IR or UV sides. The spectrum for Analog sensor is comparable to the photo resistor, hovering around the visible spectrum (Table 2).

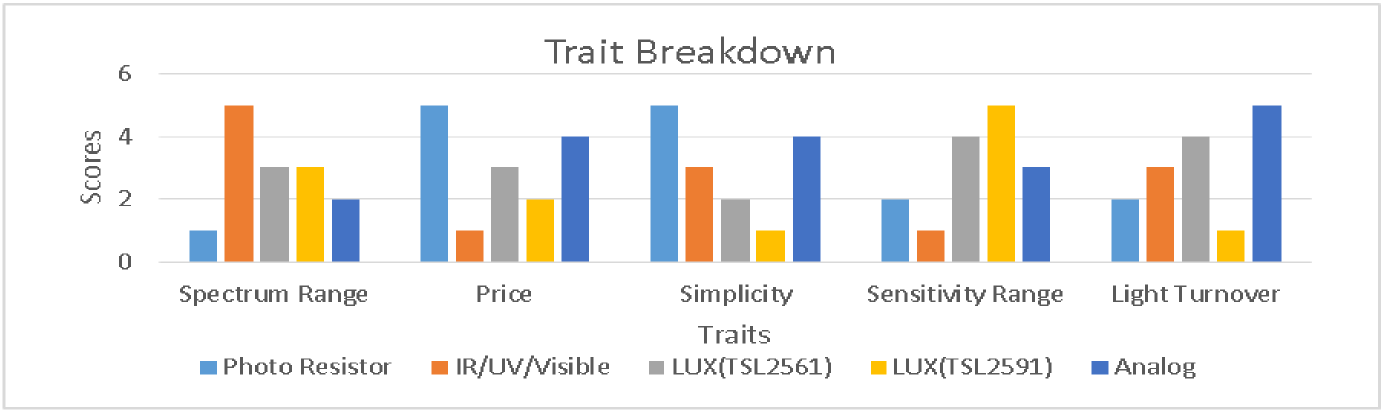


Figure 5. Trait Graph.

Price

The Photo Resistor is the cheapest sensor. Lower prices according to the criteria are valued, hence the rank 5. The IR sensor is the most expensive of the sensors, my best guess would be that the added cost is for the circuitry to detect the other spectrum sections. The high price is the rationale for the low rank. The LUX(TSL2561) sensor relative to the others is somewhat expensive, much higher than the photo resistor ($.99) but a few dollars short of the IR ($10.00). Its dollar cost is competitive for reasonability. The TSL2591 has a price point comparable to the mid-level sensors. The Analog's price point is the second lowest in the sensor subset. This is the rationale for the rank 2 (Table 2).

Simplicity

The Photo Resistor has the least installation overhead of any of the tested sensors. No soldering, no library setup, just plug in the Photo Resistor, wire it to the board, and read from the analog pins. This is the rationale for the 5 rank. The UV/IR/Vis takes a library import, initialization, and you have to differentiate the voltage units coming in (Source Code 1) [7]. The design specs state that the values from the sensor when taken for visible spectrum are "unit less." [7]

The lack of a unit and the need for library imports with device initialization also lowers the final score. The TSL2561 sensor, similarly to the IR sensor, requires a library import, and some special functionality to work. It requires extra wiring on top of basic soldering. This sensor requires one less step than the IR (Device initialization) (Appendix: Source Code 1). This sensor has virtually the same complexity of implementation as the TSL2561 (Appendix: Source Code 1). The Analog device operates in a similar manner to the photo resistor, in that it is a device primarily intended to give output in voltages related to changing resistance from light intensity. This translates to a simple plug in, and read analog values, which requires no library imports or device initialization.

Sensitivity

For the simple light application, the photo resistor does the job of detecting a light or no light well. It can't give intensity, and in side tests, ambient light runs the chance of maxing out the sensor's low LUX cap (Table 2), rendering bright ambient conditions difficult when attempting to sense new light presence. It still has value in a wide numeric range, hence the rank 2 rating.

The IR sensor is not very sensitive to visible light changes, the graph of sensor values is shaky a best (Appendix: Figure 7). Enough to do rudimentary work, but unless you are sampling along the larger spectrum it may be lost to ambient noise. One way to possibly mitigate this is to sample across the entire spectrum it is capable of, visible, IR, and UV, this may bulk up the sensitivity, but given the default load it is not sufficient. The TSL2561 sensor has a significant range reaction to light intensity. Not as strong as the TSL2591, but for basic applications it is more than enough. It far exceeds the maxed 10 LUX of the photo resistor, coming in between 3 and 55,000 LUX value. The LUX(TSL2591) sensor has the highest sensitivity of the sensors, recorded values easily outpaced the other sensors, and the stated LUX value is the highest. Unlike the photo resistor the Analog sensor can readily have its voltage value converted to LUX ratings. This sensor also has a wider range of values (Table 1-2).

Light Turnover

The Photo Resistor's rise is consistent with the other sensors and there is not a noticeable difference at the tested speeds. It is fast enough for most applications in the common world. The fall however is significantly slower than the other sensors, and depending on the sensitivity of the job may result in miss read signals (Figure 5). This is the rationale for the rank 3 rating (Figure 3). The IR sensor has a favorable light reaction time. Barring retest* it is competitive with the other sensors for good speed turnaround. The LUX(TSL2561) sensor rapidly reacts to changes in light, barring faster timing retests, it is consistent with the other sensors. The default device configuration for the LUX(TSL2591) seems to cap sample speeds at .001 Seconds. This means that it cannot match the defaults of the other devise in this type of testing. The Analog sensor has one of the best rise and fall values of the sensors, with rapid up and down, there is little wait to normalize.

The final values assessed and summed give the relative quality order from best to worst, assuming that all the traits are equally desired: Analog, TSL2561, Photo Resistor, IR/UV/Vis, and TSL4591 (Figure 6).

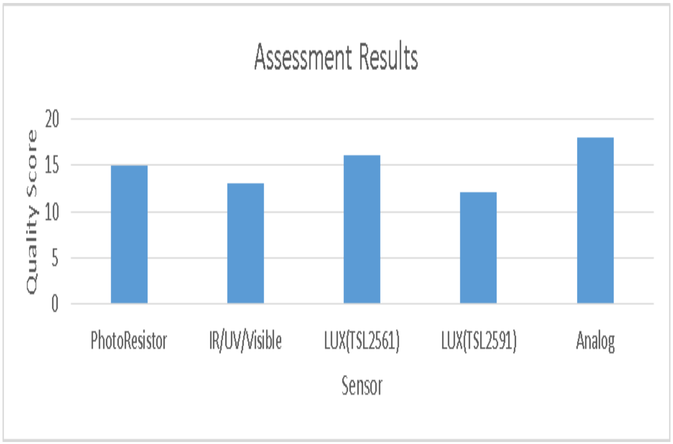


Figure 6. Assessment values graphically.

5. Conclusion and Future Work

From our testing and assessment it seems that the Photo Resister fared well among the sensors. Its simplicity and cheapness overcame the small spectrum sensitivity and the inability to measure large lux values (Table 2). The Analog sensor may be the better choice, for just a few more dollars, you gain the ease of translating voltage to LUX, a wider spectrum, and better sensitivity rating, and better light turnover. For the more basic jobs the analog sensor proves more than adequate and preferable for its well-rounded and responsive trait qualities (Table 3).

This research is useful to those not technically capable, and to industries seeking to reach out to customers through better product education. Possible gaps in my research include problems with how the wiring of each sensor is setup. The Arduino's input capacity may be shortening some of the other sensor's response times, the light source is a fixed value in the visible light spectrum and some sensors may not pick it up properly. The algorithm's ambient light processing may be damaging the low values and the highs by adjusting the voltages incorrectly. The sensing apparatus is not a perfect Faraday cage and ambient light and noticeable shifts occur from outside sources. The different sensors sometimes use different base measurements. Optimizations of these light sensors, better testing environments, and a more thorough quality assessment may be undertaken in the future. Other possible avenues for research are testing across the spectrum sensitivity and testing with variable distances light sources, and testing readability of various documentation styles.

6. ACKNOWLEDGMENT

Thanks to Dr. Debnath, Dr. Cichanowski, Dr. Zhang, of the Computer Science Department and Christopher Bischke for their active help in preparing this manuscript.

7. REFERENCES

[1] Advanced Photonix, Inc., “CdS Photoconductive Photocells PDV-P8001,” 18 2 2015. [Online]. Available: http://www.advancedphotonix.com/wp-content/uploads/PDV-P8001.pdf. [Accessed 18 2 2015].

[2] l. ada, “Photocells,” 29 7 2012. [Online]. Available: https://learn.adafruit.com/. [Accessed 18 2 2015].

[3] PerkinElmer Optoelectronics, “Photocells A 9950, A 7060, B 9060 Epoxy encapsulated Series,” PerkinElmer Optoelectronics, Vaudreauil-Dorion, 2008.

[4] AdaFruit, “Products Page,” 18 2 2015. [Online]. Available: http://www.adafruit.com/products/439. [Accessed 18 2 2015].

[5] TAOS | Texas Advanced Optoelectronic Solutions, “TSL256,” 2009. [Online]. Available: http://www.adafruit.com/datasheets/TSL256x.pdf. [Accessed 25 2 2015].

[6] Arduino, “analogRead,” 9 3 2015. [Online]. Available: http://arduino.cc/en/Reference/analogRead. [Accessed 9 3 2015].

[7] L. Ada, "si1145-breakout-board-uv-ir-visible-sensor/overview," 21 3 2014. [Online]. Available: https://learn.adafruit.com/adafruit-si1145-breakout-board-uv-ir-visible-sensor/overview. [Accessed 10 3 2015].

[8] A. a. K. T. a. K. Y. a. S. S. Yamawaki, "A Method Using the Same Light Sensor for Detecting Multiple Events Near a Window in Crimes Involving Intrusion into a Home," *Artif. Life Robot.*, vol. 15, no. 1433-5298, pp. 30--32, August 2010.

[9] J. a. B. D. a. W. K. Lu, "Using Simple Light Sensors to Achieve Smart Daylight Harvesting," in *Proceedings of the 2Nd ACM Workshop on Embedded Sensing Systems for Energy-Efficiency*, New York, NY, USA, ACM, 2010, pp. 73--78.

[10] S. a. P. F. Makonin, "An Intelligent Agent for Determining Home Occupancy Using Power Monitors and Light Sensors," in *Proceedings of the 9th International Conference on Toward Useful Services for Elderly and People with Disabilities: Smart Homes and Health Telematics*, Berlin, Heidelberg, Springer-Verlag, 2011, pp. 236--240.

[11] J. C. a. D. P. H. a. M.-A. D. a. R. R. a. H. S. E. Lee, "Automatic Projector Calibration with Embedded Light Sensors," in *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology*, New York, NY, USA, ACM, 2004, pp. 123--126.

[12] Z. a. Z. F. a. W. Q. Cao, "Size Measurement Based on Structured Light Sensor," in *Proceedings of the 2011 Second International Conference on Digital Manufacturing \& Automation*, Washington, DC, USA, IEEE Computer Society, 2011, pp. 1098--1101.

[13] Oda, Y. Sharp Corporation, "GA1A1S202WP OPIC Light Detector," November 2007. [Online]. Available: http://www.adafruit.com/datasheets/GA1A1S202WP_Spec.pdf. [Accessed 25 2 2015].

[14] Silicon Labortories, "Proximity/UV/Ambient Light Sensor IC with I^2C Interface," Silicon Labortories, 2013.

[15] Arduino, "Learning Delay," 9 3 2015. [Online]. Available: http://arduino.cc/en/Reference/Delay. [Accessed 9 3 2015].

[16] B. Earl, "Adafruit GA1A12S202 Log-scale Analog Light Sensor Overview," 16 3 2013. [Online]. Available: https://learn.adafruit.com/adafruit-ga1a12s202-log-scale-analog-light-sensor. [Accessed 16 3 2015].

[17] Y. Oda, "Device specifications for OPIC Light Detector Model No. GA1A1S202WP," Sharp Corporation, Camas, 2007.

APPENDIX

Table 1. Manufacturer's Specifications.

Sensors	Photo Resistor [1] [2] [3]	IR/UV/Vis [7] [10-11]	LUX(TSL2561) [4] [5]	LUX(TSL2591) [4]	Analog [4][13-14]
Dimensions (mm)	4.46(L)*5(W)*2.09(H)	20*18*2	10*13*1.5	19*16*1	10*13*1.5
Weight (g)	.25	1.4	.2	1.1	.2
Price ($)	1.00	9.95	5.95	6.95	3.95
Operating Temp. (C)	-30, 75	-40, 85	-30, 80,	-30, 80	-30,70
Spectrum (nm)	400 (Min, Violet) - 700 (Max, Orange) 520 (Peak, Green)	550–1000 400-800 200-400	Measured LUX (Vis)	Measured LUX (Vis)	555
Sensitivity	LUX 0 -10	Non Unit for Vis	LUX 0.01 – 40,000	LUX .01 – 88,000	LUX 3-55,000

Table 2. Result table from Apparatus testing.

Sensors	Photo Resistor	IR/UV/Vis	LUX(TSL2561)	LUX(TSL2591)	Analog
Sensitivity Range	544 V/Unit ~Maxes at 10 lux [2]	11 Unit less	719 LUX	507 LUX	93 LUX
Light Turnover (second)	.04 (Fall) .0008 (Rise)	.0006 (Rise)/ (Fall)	.0006 (Rise)/(Fall)	.02 (Rise/Fall)	.0004 (Rise)/(Fall)

Table 3. Assessment Values.

Sensors	Photo Resistor	IR/UV/Visible	LUX(TSL2561)	LUX(TSL2591)	Analog
Spectrum Range	1	5	3	3	2
Price	5	1	3	2	4
Simplicity	5	3	2	1	4
Sensitivity Range	2	1	4	5	3
Light Turnover	2	3	4	1	5
Final Score	15	13	16	12	18

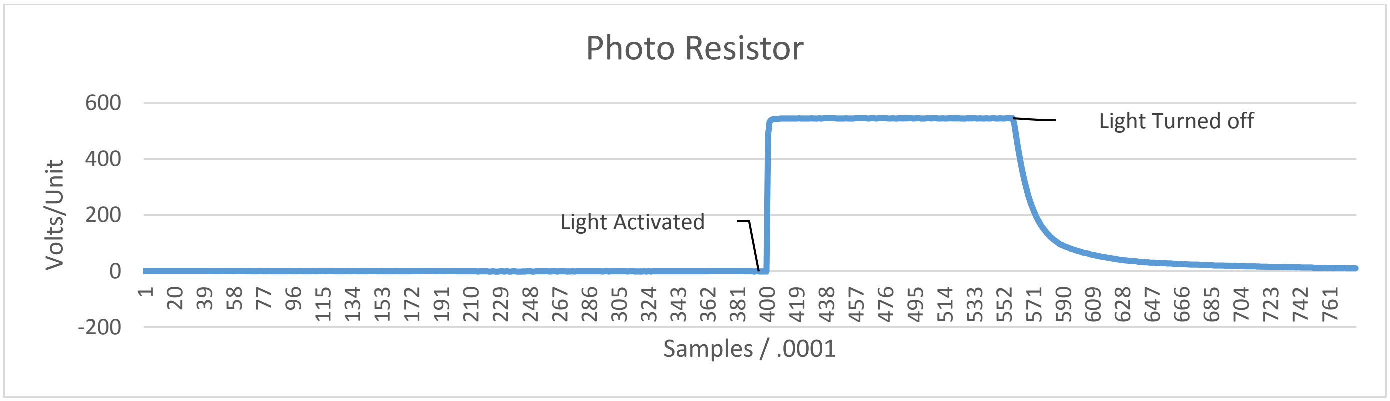


Figure 3. Photo Resistor sensor values during light activation. Light is on from sample 400, and off at 568~.

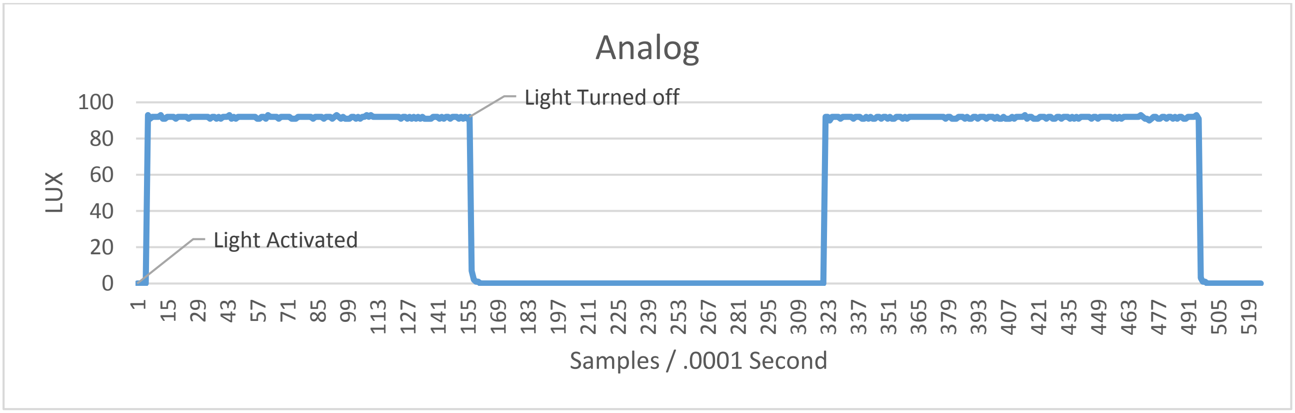


Figure 4. Analog sensor values during testing. Light was activated between 5th-7th samples. Light deactivated around 155th-160th samples.

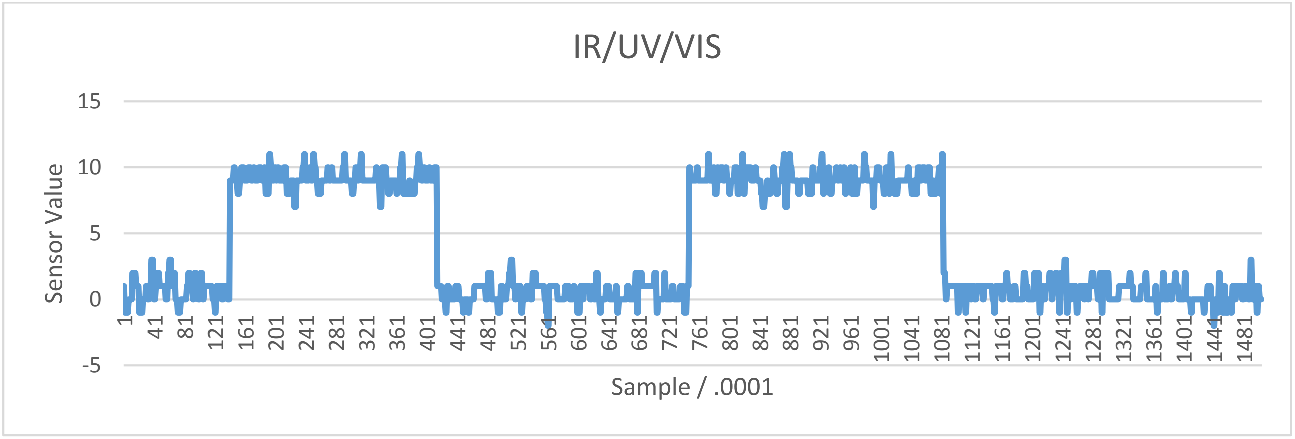


Figure 7. IR/UV/VIS sensor values during testing. Light is activated twice, corresponding to jumps.

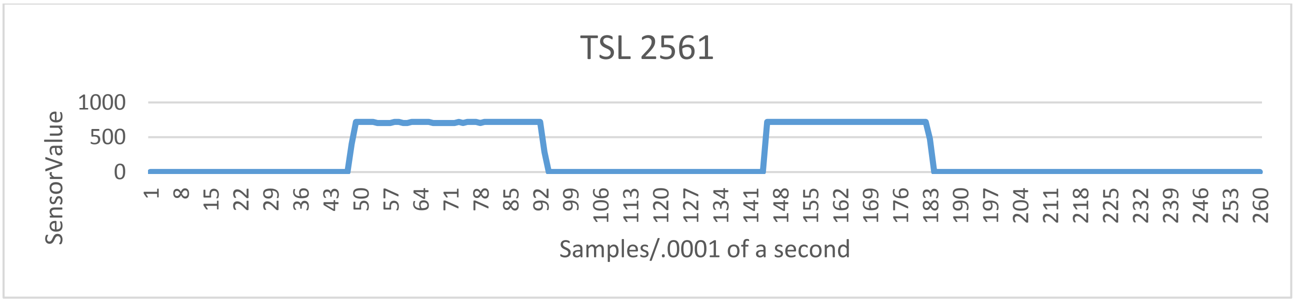


Figure 8. TSL 2561 sensor values during testing.

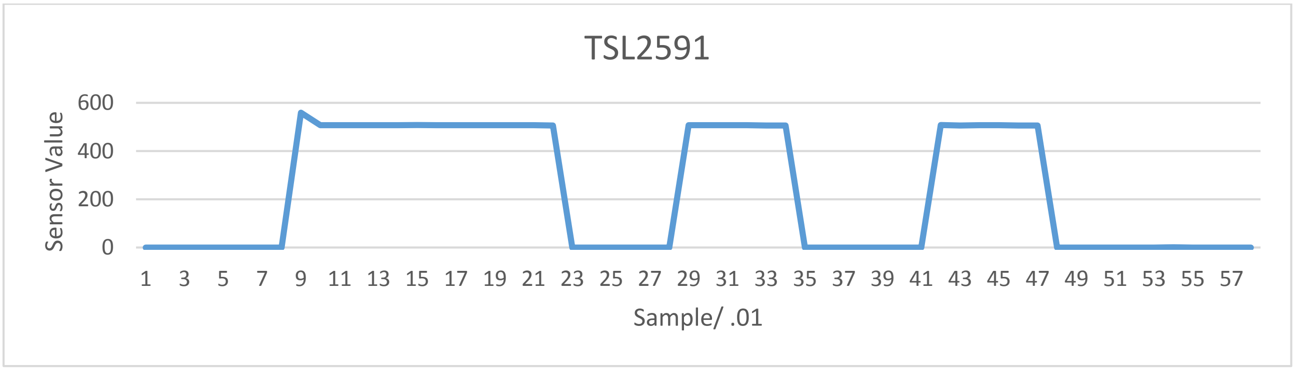


Figure 9. TSL 2591 sensor values during testing.

MapReduce Efficiency Between Operating Systems.

Robert Rutscher
Computer Science Department
rrutscher11@winona.edu

ABSTRACT

Since the MapReduce paradigm was introduced over a decade ago it has gained popularity and importance throughout the software development community due to ever-changing scalability requirements, and the need for easily deployable distributed computing. Hadoop is a popular open-source distribution of MapReduce and was used during research. Hadoop's efficiency between different operating systems and their distributions has not had an in depth study done and is the main point in this research. The top Linux and Windows server distributions were tested within a 3-node cluster with six different benchmarks. Data analysis has provided evidence of greater efficiency on Linux.

Keywords

Hadoop, MapReduce, Efficiency, Comparison

1. INTRODUCTION

In recent years we have asked systems to process larger and larger sets of data. The problem with handling such large amounts of data across thousands of machines is the near impossibility to efficiently distribute, process in a reasonable amount of time, and handle data failures. In an effort to reduce the need to repeatedly implement parallel/distributing processing, Google created MapReduce. The fundamental idea can be abstracted into two separate functions, Map and Reduce, which were first introduced in Lisp and can be found in many other functional programming languages [6].

The core concept of MapReduce is the ability to generate and process data sets using parallel and distributed processing within a computational cluster. The fundamental procedures that implement and achieve this are: the Map() function and Reduce() function. The job of Map() is to perform the filtering and sorting on a given data set, while the job of Reduce() is to perform a summary operation on the data. An example of the Map() procedure is sorting a list of students in a University by first name using queues, there is one queue for each name. An example of the Reduce() procedure would be counting the number of students in each queue, which ultimately would return the number of times a given name occurs in the given text [6]. The major advantage of this is splitting up of tasks efficiently so that large data sets are processed in a reasonable amount of time.

The MapReduce programming model was introduced in 2003 by Google in an academic paper, from that point on it has rapidly evolved due to a large amount of support received from both the Open Source community and the Apache Software Foundation [8]. Yahoo spearheaded development of the most popular MapReduce implementation, Hadoop in 2007 [8]. Hadoop as well as the MapReduce programming model's adoption has been accelerated due to the need to be able to handle "Big Data" in today's enterprises. Big data is the information all around us, it can be trivial statistics such as how many times you unlock your phone each day, to the important ones such as the IP address that is accessing a bank account. The collection of all this data is considered "Big Data" it is characterized by the sheer size of what it encompasses and the lack of structure present within [5]. The ability to process and make inferences from this unstructured and messy coagulation of data is where a system such as Hadoop can excel.

The majority of servers worldwide are running either Windows or Linux as their operating system, functionality between the two is similar but differences are present. The server owner makes the choice, that being said, there are pros and cons to each. Windows Server gives users support should any problems arise, and is generally considered easier to configure. These benefits come with a price tag, literally. Window's licenses can be prohibitively expensive to deploy across an entire cluster. In comparison, Linux is free and can be distributed and modified by anyone, it also has a lighter resource footprint that Windows. The benefits Linux provides come with the con of little to no support compared to Windows. Although these non-technical differences matter when deciding upon operating system there are many technical differences that differentiate Linux and Windows. Linux and Windows have completely different file structures, Linux does not have a C: Drive and instead relies on the user to setup drive access. Linux also does not have a registry, this is a master database that on Windows holds settings for all users and applications. Settings on Linux are stored on a program-by-program basis under the user. This gives greater modularity when performing or duplicating installations. Linux may only run on some hardware configurations while Windows has very wide driver support and supports nearly any hardware. These characteristics give rise to a choice that a company must make between the two operating system. The conclusions reached in this paper can help provide proper insight into efficiency when making operating system choice.

Hadoop is an application that requires the proper hardware resources to do its job properly. This can mean spending a considerable amount of a company's budget to have the correct hardware for Hadoop. In an attempt to minimize the cost, and/or maximize the money spent, a question can be raised and hypothesis given: What operating system will run Hadoop the most efficiently? The operating system that will run Hadoop most efficiently is a Linux server distribution. The ability to change the operating system is always possible and in the long run even 1% more efficiency can lead to large monetary and energy savings [1]. The investigation following compares efficiency between the most widely used Linux and Windows server distributions.

2. HYPOTHESIS

I hypothesize the Linux server distribution will have greater efficiency metrics for Memory Usage, Network Usage, CPU Usage, I/O Usage, and Time for Job completion compared to its Windows counterpart.

3. METHOD

Testing involves running six tests and recording performance data on four-machine Hadoop cluster (3 nodes, 1 master). Operating systems compared are the most common server distributions for Linux and Windows. These are Ubuntu 14.04 (Linux) and Windows Server 2012. The following sections are an in-depth explanation of the testing, data gathering and data analysis methodology.

3.1 Testing Environment

The testing being done is meant to be as controlled, accurate and precise as possible. The environment was created with these principles in mind as well as the overall goals of minimizing unknowns and environmental variables that could skew data. The hardware configuration remained the same throughout testing while the software changed with the respective operating system. Hardware was provided by Winona State University Technical Support and comprised of four HP 8470w laptops that natively support Windows and Linux. Native support means the hardware within the machine specifically supports said operating system, in this case both Windows and Linux. It was important that both operating systems ran natively on the hardware. Non-native support would have caused confounding variables leading to inaccurate results. Table 1 is the hardware specifications for the machines used.

The machines also had their BIOS updated to the most recent version. BIOS is the interface between the operating system and hardware, and is important to have updated to ensure no problems occur. The nodes will also be connected to a private local area network (LAN). A private LAN will not have any other computers connected and will not have a connection to the internet. The following figure is a visualization of cluster setup. Figure 1 shows a diagram of the closed network that was used for testing. Three machines considered nodes were configured to receive workload within the Hadoop cluster, one machine was configured to direct workload and is referred to as master. Commands throughout the testing process will be run on the master node to perform benchmarks and distribute the workload for the benchmarks. The operating systems tested were the top server distributions from both Linux and Windows, these were Ubuntu Server 14.04 for Linux and Windows Server 2012 for Windows. The software portion of the testing environment had many similarities but there were steps unique to each operating system. These differences are described but are not believed to have affected efficiency.

The installation and setup process for Hadoop between Windows and Linux had several similarities and several differences. First both were installed from flash drives prepared with UNetbootin, the test machines had labels attached to ensure node1 was considered node1, node2 was node2, etc. for testing on both Linux and Windows. Operating systems were installed without difficulty and updates were run; using "apt-get upgrade" on the Linux machines and Windows Update on the Windows Machines. The HOSTS files for each set of machines were configured to allow for easy network name resolution. This means instead of using an IP address to connect to another machine, a machine would be addressed with a plaintext name such as node1 or master.

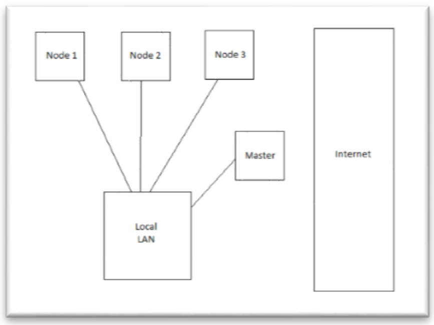


Figure 1. Diagram of cluster setup.

Table 1: Hardware Configuration for testing machines

Computer	HP 8470w
Processor	2.90 Ghz Intel Core i7-3520M
Hard Drive	HGST HTS725959A7E630 500GB 7200 RPM
Memory	2x4GB PC3L-12800S (8GB) RAM
Network	Intel 82579LM Gigabit Ethernet
Graphics Card	AMD FirePro M2000

Java 7u76 JDK & SDK was installed on each set of machines. The differences between the setup processes began to surface here. The Hadoop installation process on Linux involved downloading the pre-compiled Hadoop 2.6.0 binaries and extracting. On Windows, Hadoop ran natively as of version 2.2.0 but does not have pre-compiled binaries. cMake and the Windows SDK were used to compile Hadoop 2.6.0. Windows also needed access to a few of the basic Linux commands, to enable this Cygwin was installed. Cygwin is a collection of GNU and Open Source tools that provide functionality similar to a Linux distribution on Windows. Hadoop was added to the PATH variable on both sets of machines. SSH configured setup on both sets of systems, and keys were added to authorized key folders to allow for password-less connections between systems. BASH and Powershell terminal were used throughout the process for each respective operating system, this ensured that commands being run were the same, and provided uniformity throughout the setup process.

3.2 Testing

The benchmarks being used for testing purposes are built into Hadoop 2.6.0. This allows for exactly the same test to be run on both Linux and Windows without needing to worry about differences that may have sprung up if new code had needed to be written for each respective test. The tests are piTest(), DFSIOTest() and teraSort(). These three tests are comprised of one, two and three functions respectively.

piTest() - This test will be a simple work delegation and data calculation. It is designed to have the digits of pi calculated to a specific decimal place. The CPU as well as network adapter will be taxed during this test, data collected should help provide data on the different between average CPU and network usage between the operating system [7]. This test is comprised of a single function and will be run from the master node. Results will be gathered from each of the nodes with data gathering commands. A sample command for piTest on Linux looks like this:

```
“yarn jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-
mapreduce-examples-2.6.0.jar pi 16 500000000“
```

The configuration flags on the piTest() command are settings for a Monte-Carlo function that calculates pi. The command creates a job to distribute the drawing of 500,000,000 points 16 times across the three nodes [2]. The data is then reduced and given back to the master node for output.

DFSIOTest() - This test will involve the writing and reading of data, it is largely I/O based and is a good indicator of the read and write ability of each operating system. As discussed previously this test has two functions, these are: DFSIOwrite() which create a specified amount of files of a specific size, and the other being DFSIOread() which reads back these files [7]. This test also provides statistics on its job completion which were helpful for data gathering purposes. Figure 2 in the appendix is an example of output from this job. The commands for DFSIOtest looked like this:

```
“yarn jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-
mapreduce-client-jobclient-2.6.0-tests.jar TestDFSIO -write  -
nrFiles 10 -fileSize 100

yarn jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-
mapreduce-client-jobclient-2.6.0-tests.jar TestDFSIO -read  -
nrFiles 10 -fileSize 100”
```

The configuration flags attached to the testDFSIO() commands first specify the amount of files to write, in this case it is 10, and the size of each file in megabytes, 100. This test is largely I/O and network traffic based and not based off of calculation.

teraSort() - The final test was an overall benchmark of performance as it uses a considerable amount of each resource including CPU, networking, storage, and I/O. All resources being used simultaneously give a good overall perspective of system activity and performance during load. TeraSort() similar to TestDFSIO() is comprised of multiple functions. These are teraGen() which is a write/generate command, teraSort() which sorts the data, and teraValidate() which reads the test data and verifies its validity [7]. An example of the commands necessary to run this benchmark are:

```
“yarn jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-
mapreduce-examples-2.6.0.jar teragen 10000000 /teraInput

yarn jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-
mapreduce-examples-2.6.0.jar terasort /teraInput /teraOutput

yarn jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-
mapreduce-examples-2.6.0.jar teravalidate /teraOutput
/teraValOut”
```

The single numerical parameter used in the teraSort() command directs the creation of rows equal to that number. Each row has a size of 100 bytes. The amount of data being generated during this command is equal to 1 gigabyte. The additional parameters used are directories for data input and output.

The testing environment (includes hardware and software) has been created to reduce any confounding variables. Operating systems were setup in as close to duplicate as possible fashion. This has been described in the previous two sections. Identical hardware was procured for the testing process, however small differences in how identical hardware performs will have to be dealt with through percent error and will be calculated into the metric values. As for confounding variables that can occur because of software, they will be minimized through having default installations of operating systems, with only the necessary runtimes installed for Hadoop to run. The benchmarks being run also were the same between the two operating system which helps reduce any inconsistencies we may have experienced using custom made tests for each operating system.

3.3 Data Processing

As the tests were being run, data was gathered related to system usage. The data needed for comparison purposes was utilization data for: CPU, network, I/O, and memory. Time for job completion was also a statistic gathered but was separate from the previous metrics in the collection process. The data gathering process gathered more data than necessary and needed further processing to eliminate unnecessary statistics and congregate necessary parts.

The testing process involved running each benchmark three times. The performance data was gathered on the three nodes in each cluster. The operating systems were polled every 5 seconds for 60 seconds giving us 12 data entries for each node for each test for each trial. The testing generated 54 .csv files for each operating system (3 nodes * 6 functions * 3 repetitions) for a total of 108 .csv files that needed processing.

The data gathering on Linux used the sysstat command. This command allows for all system data to be gathered at specific intervals. The command recorded more data than was needed, this excess was stripped during the data processing step. A sample command looked like this:

```
“sar -u -r -d -q -b -n DEV 5 12 | grep -v Average | grep -v Linux
|awk '{if ($0 ~ /[0-12]/) { print
$1","$2","$4","$5","$6","$7","$8","$9","$10","$11","$12; }  }' >
"$(hostname)pi1.csv"”
```

This command is comprised of four parts: first, it is using sysstat with parameters that provide information about CPU, Memory, Network, Disk, and system load every five seconds twelve times over for a sample period of one minute. The command then uses grep to remove two columns containing unnecessary data from the output. Next awk is used to select the columns and delimit them with commas, the final portion pipes this output to a text file named with the host name of the machine and the test being run.

The data gathered on Windows used the built-in utility called Performance Monitor. This was manually configured using the Windows Server 2012 GUI on the master node. The data was also exported to a .csv file. The Windows data had even more information was unneeded and had a lot of processing done to extract the relevant data. Raw data is table 2 in the appendix.

The raw files that were output from data gathering were not formatted in a fashion that would allow for data analysis. It was necessary to parse the many csv files generated and remove excess data as well as average out necessary data. Excel and Visual Basic were used to process the data and a sample of the scripts used are available in the appendix. Excel files for both Windows and Linux were formatted different and thus needed different processing was run on each. The large number of files made it necessary for the scripts to be written, they prevented errors and a lot of copying and pasting. Once data was correctly formatted, it was easily comparable between operating systems. Table 3 in the appendix is an example of properly formatted data.

After the data was correctly formatted it was averaged. The averaging process took the data from each node and each trial and congregated it into six values along with the nine metrics. An example of this is table 4 in appendix. The data from each metric was the sorted and brought into separate sheets to prepare for a two-sample t-test with unequal variances. This was used because it was not known if the data would have equal variances.

4. RESULTS

The results provided are based off the statistical significance of each of the nine metrics. The null and alternative hypothesis for the results follows:

H_o = Linux Efficiency is equal to Windows Efficiency

H_a = Linux Efficiency is greater than Windows Efficiency

The two-sample t-test with unequal variances was used. If the t stat value was greater than the t critical one-tail value the difference was considered significant. Three of the metrics tested were found to have a significant difference. These were %idle, %system and kbmemused. Six of the metrics were found to have no evidence of a statistical significant difference, these were: %memused, bwrtn/s, bread/s, txkB/s, rxkB/s, and job completion time. Table 5 in the appendix provides the values calculated from statistical analysis. The results showed a statistically significant difference in efficiency for the %system, %idle, and kbmemused metrics. There was not sufficient evidence to show a significant efficiency difference for the %memused, I/O, Job Completion time or Network utilization metrics.

The metrics being used in measuring operating system efficiency were selected for the accurate measure of system performance they provide. The metrics following were compared between operating systems to find a statistically significant difference in efficiency. Each metric is important because anyone of them can cause bottlenecks in system performance, the entire system can become slower just because one part is not able to do its job fast enough [3].

CPU utilization is a key performance and efficiency metric. It is often used to track CPU performance regressions and improvements, it also is directly correlated to energy usage of a system. **%system** is a metric that describes the amount of current CPU in use currently. **%idle** describes the amount the CPU is not being used during operation, it is directly correlated to %system. Memory footprint is another metric able to gauge system efficiency. It can be used to track performance hiccups and could be the cause of bottlenecks in a system. **%memused** is the amount of available memory currently used by applications currently running. **Kbmemused** is a discrete number describing the amount of memory in use. Network utilization is a vital efficiency metric used to troubleshoot bottlenecks that may occur in computing clusters. The local network is the primary form of communication in computing and is important to keep from becoming congested. **txkB/s** is the amount of kilobytes being sent per second by the network adapter. **rxkB/s** is the amount of kilobytes received per second by the network adapter. Disk utilization is important when processing and generating data, it is the rate at which data will be able to written as well as read. The I/O speed of a device is generally completely dependent upon the speed of hard drive within the system. **bwrtn/s** is the amount of bits written to the hard drive by the operating system per second. **bread/s** is the amount of bit read from the hard drive by the operating system per second. Job completion time is largely self-explanatory. It is the amount of time it will take for a particular operation being run to finish. This may differ largely based off of OS. This metric is measured in seconds for each job. [4]

5. ANALYSIS

The results provide interesting insight into operating system efficiency. Both CPU metrics showed a significant difference between operating systems while memory was shown to only have a significant difference for one of its metrics. The two memory metrics should coincide as they are pulling from similar data sources. The fact that they are different does not necessarily mean that there was an error collecting data. The data says for %memused there was not significant evidence that it was different between the two operating systems. Differences between metric usage can be seen in table 6 in the appendix.

The three significant and six non-significant metrics can lead to a few conclusions: 1. Hadoop on Linux has an overall lower CPU usage than on Windows. 2. Hadoop on Linux has lower memory usage than on Windows. 3. Network, I/O, and Job Completion do not have a significant difference between the two operating systems. With lower memory and CPU usage a system can do more at the same time, this across a large cluster can lead to very large savings in both time for completion and energy usage.

6. CONCLUSION

The Linux operating system was more efficient than its Windows counterpart in regard to memory used, %idle and %system. There was not sufficient evidence to show greater efficiency for Network Utilization, I/O Usage, and Job completion time. Though only three out of nine metrics displayed greater efficiency, the difference small gains can make in a normal sized computing cluster can lead to large energy and monetary savings. Further study could be directed towards configuration of Hadoop in an operating system specific manner.

7. ACKNOWLEDGMENTS

Thanks to Dr. Zhang and Dr. Debnath for guidance throughout the research process, also thank you to Winona State Technical Support for access to test systems.

8. REFERENCES

[1] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. 2008. Improving MapReduce performance in heterogeneous environments. In Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08). USENIX Association, Berkeley, CA, USA, 29-42.

[2] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. 2010. The performance of MapReduce: an in-depth study. Proc. VLDB Endow. 3, 1-2 (September 2010), 472-483. DOI=10.14778/1920841.1920903 http://dx.doi.org.wsuproxy.mnpals.net/10.14778/1920841.1920903

[3] Greg Malewicz. 2011. Beyond MapReduce. In Proceedings of the second international workshop on MapReduce and its applications (MapReduce '11). ACM, New York, NY, USA, 25-26. DOI=10.1145/1996092.1996098 http://doi.acm.org.wsuproxy.mnpals.net/10.1145/1996092.1996098

[4] Michael Cardosa, Chenyu Wang, Anshuman Nangia, Abhishek Chandra, and Jon Weissman. 2011. Exploring MapReduce efficiency with highly-distributed data.
In Proceedings of the second international workshop on MapReduce and its applications (MapReduce '11). ACM, New York, NY, USA, 27-34.
DOI=10.1145/1996092.1996100
http://doi.acm.org.wsuproxy.mnpals.net/10.1145/1996092.1996100

[5] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. 2012. Parallel data processing with MapReduce: a survey. SIGMOD Rec. 40, 4 (January 2012), 11-20. DOI=10.1145/2094114.2094118 http://doi.acm.org.wsuproxy.mnpals.net/10.1145/2094114.2094118

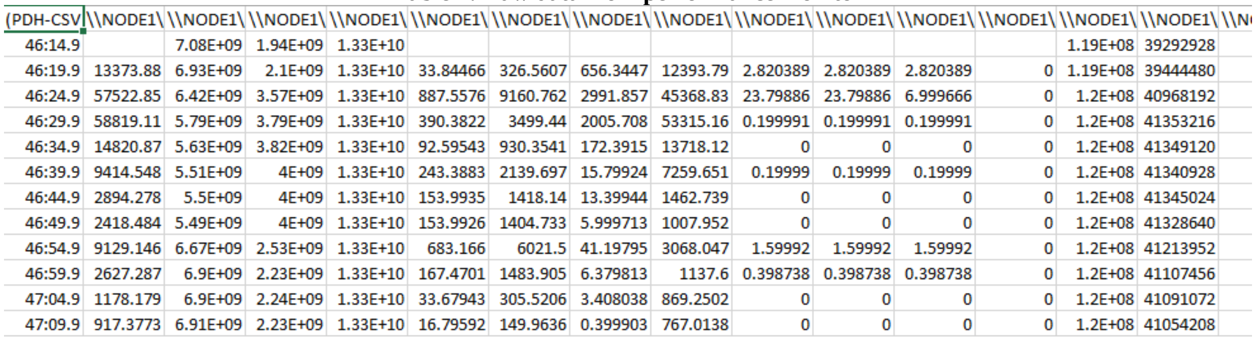
[6] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008), 107-113.
DOI=10.1145/1327452.1327492
http://doi.acm.org.wsuproxy.mnpals.net/10.1145/1327452.1327492Conger., S., and Loch, K.D. (eds.). Ethics and computer use. Commun. ACM 38, 12 (entire issue).

[7] MSIT SES Enterprise Data Architect Team. 2013. Performance of Hadoop on Windows in Hyper-V, Microsoft Inc.
http://download.microsoft.com/download/1/C/6/1C66D134-1FD5-4493-90BD-98F94A881626/Performance%20of%20Hadoop%20on%20Windows%20in%20Hyper-V%20Environments%20%28Microsoft%20IT%20white%20paper%29.docxSchwartz, M., and Task Force on Bias-Free Language. Guidelines for Bias-Free Writing. Indiana University Press, Bloomington IN, 1995.

[8] Darrick Harris. 2013. The history of Hadoop: From 4 nodes to the future of data. Gigacom (March 2013). https://gigaom.com/2013/03/04/the-history-of-hadoop-from-4-nodes-to-the-future-of-data/

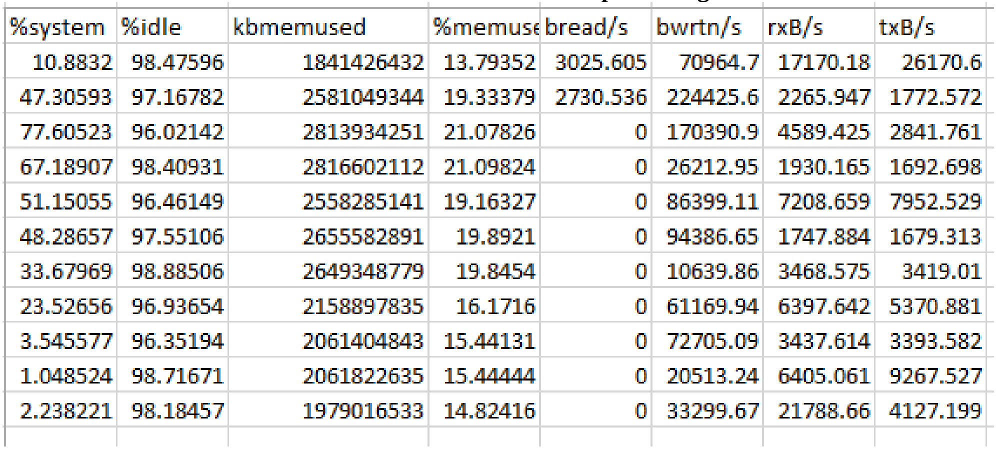
Appendix

Table 2: Raw data from performance monitor



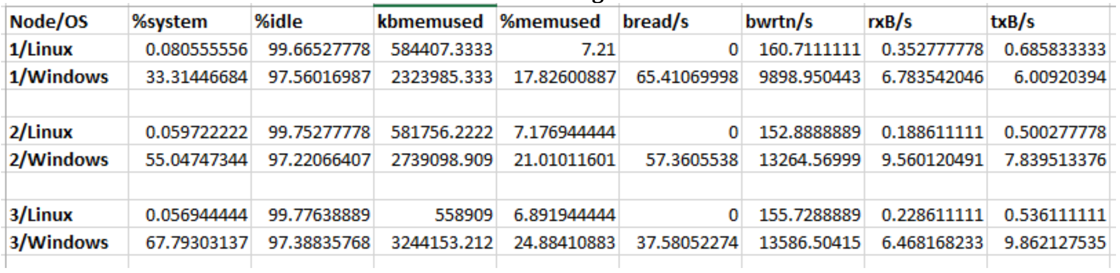
(PDH-CSV	\\NODE1\	\\NODE1\	\\NODE1\	\\NODE1\	\\NODE1\	\\NODE1\	\\NODE1\	\\NODE1\	\\NODE1\	\\NODE1\	\\NODE1\	\\NODE1\	\\NODE1\	\\NODE1\	\\N
46:14.9		7.08E+09	1.94E+09	1.33E+10									1.19E+08	39292928	
46:19.9	13373.88	6.93E+09	2.1E+09	1.33E+10	33.84466	326.5607	656.3447	12393.79	2.820389	2.820389	2.820389	0	1.19E+08	39444480	
46:24.9	57522.85	6.42E+09	3.57E+09	1.33E+10	887.5576	9160.762	2991.857	45368.83	23.79886	23.79886	6.999666	0	1.2E+08	40968192	
46:29.9	58819.11	5.79E+09	3.79E+09	1.33E+10	390.3822	3499.44	2005.708	53315.16	0.199991	0.199991	0.199991	0	1.2E+08	41353216	
46:34.9	14820.87	5.63E+09	3.82E+09	1.33E+10	92.59543	930.3541	172.3915	13718.12	0	0	0	0	1.2E+08	41349120	
46:39.9	9414.548	5.51E+09	4E+09	1.33E+10	243.3883	2139.697	15.79924	7259.651	0.19999	0.19999	0.19999	0	1.2E+08	41340928	
46:44.9	2894.278	5.5E+09	4E+09	1.33E+10	153.9935	1418.14	13.39944	1462.739	0	0	0	0	1.2E+08	41345024	
46:49.9	2418.484	5.49E+09	4E+09	1.33E+10	153.9926	1404.733	5.999713	1007.952	0	0	0	0	1.2E+08	41328640	
46:54.9	9129.146	6.67E+09	2.53E+09	1.33E+10	683.166	6021.5	41.19795	3068.047	1.59992	1.59992	1.59992	0	1.2E+08	41213952	
46:59.9	2627.287	6.9E+09	2.23E+09	1.33E+10	167.4701	1483.905	6.379813	1137.6	0.398738	0.398738	0.398738	0	1.2E+08	41107456	
47:04.9	1178.179	6.9E+09	2.24E+09	1.33E+10	33.67943	305.5206	3.408038	869.2502	0	0	0	0	1.2E+08	41091072	
47:09.9	917.3773	6.91E+09	2.23E+09	1.33E+10	16.79592	149.9636	0.399903	767.0138	0	0	0	0	1.2E+08	41054208	

Table 3: Formatted data from processing



%system	%idle	kbmemused	%memuse	bread/s	bwrtn/s	rxB/s	txB/s
10.8832	98.47596	1841426432	13.79352	3025.605	70964.7	17170.18	26170.6
47.30593	97.16782	2581049344	19.33379	2730.536	224425.6	2265.947	1772.572
77.60523	96.02142	2813934251	21.07826	0	170390.9	4589.425	2841.761
67.18907	98.40931	2816602112	21.09824	0	26212.95	1930.165	1692.698
51.15055	96.46149	2558285141	19.16327	0	86399.11	7208.659	7952.529
48.28657	97.55106	2655582891	19.8921	0	94386.65	1747.884	1679.313
33.67969	98.88506	2649348779	19.8454	0	10639.86	3468.575	3419.01
23.52656	96.93654	2158897835	16.1716	0	61169.94	6397.642	5370.881
3.545577	96.35194	2061404843	15.44131	0	72705.09	3437.614	3393.582
1.048524	98.71671	2061822635	15.44444	0	20513.24	6405.061	9267.527
2.238221	98.18457	1979016533	14.82416	0	33299.67	21788.66	4127.199

Table 4: Averaged data



Node/OS	%system	%idle	kbmemused	%memused	bread/s	bwrtn/s	rxB/s	txB/s
1/Linux	0.080555556	99.66527778	584407.3333	7.21	0	160.7111111	0.352777778	0.685833333
1/Windows	33.31446684	97.56016987	2323985.333	17.82600887	65.41069998	9898.950443	6.783542046	6.00920394
2/Linux	0.059722222	99.75277778	581756.2222	7.176944444	0	152.8888889	0.188611111	0.500277778
2/Windows	55.04747344	97.22066407	2739098.909	21.01011601	57.3605538	13264.56999	9.560120491	7.839513376
3/Linux	0.056944444	99.77638889	558909	6.891944444	0	155.7288889	0.228611111	0.536111111
3/Windows	67.79303137	97.38835768	3244153.212	24.88410883	37.58052274	13586.50415	6.468168233	9.862127535

Table 5: Statistical Analysis Results

Metric	t stat	t-Critical one-tail	p-value
%idle	2.496919931	1.724718243	0.010686812
%system	4.423229218	1.739606726	0.000186085
%memused	-0.987264239	1.703288446	0.166139206
kbmemused	5.624648547	1.690924255	.0000001324
bwrtn/s	-1.474419366	1.70561792	0.076187639
bread/s	1.720597957	1.739606726	0.051736128
txkB/s	1.650168435	1.690924255	0.054058994
rxkB/s	0.756649403	1.701130934	0.227789925
Job Time	-0.118222423	1.690924255	0.453293561

```
15/03/29 16:18:02 INFO fs.TestDFSIO: ----- TestDFSIO ----- : write
15/03/29 16:18:02 INFO fs.TestDFSIO:            Date & time: Sun Mar 29 16:18:02 CDT 2015
15/03/29 16:18:02 INFO fs.TestDFSIO:        Number of files: 10
15/03/29 16:18:02 INFO fs.TestDFSIO: Total MBytes processed: 1000.0
15/03/29 16:18:02 INFO fs.TestDFSIO:      Throughput mb/sec: 10.665415257943067
15/03/29 16:18:02 INFO fs.TestDFSIO: Average IO rate mb/sec: 10.66562557220459
15/03/29 16:18:02 INFO fs.TestDFSIO:  IO rate std deviation: 0.047367220052642704
15/03/29 16:18:02 INFO fs.TestDFSIO:     Test exec time sec: 96.921
15/03/29 16:18:02 INFO fs.TestDFSIO:

15/03/29 16:21:00 INFO fs.TestDFSIO: ----- TestDFSIO ----- : read
15/03/29 16:21:00 INFO fs.TestDFSIO:            Date & time: Sun Mar 29 16:21:00 CDT 2015
15/03/29 16:21:00 INFO fs.TestDFSIO:        Number of files: 10
15/03/29 16:21:00 INFO fs.TestDFSIO: Total MBytes processed: 1000.0
15/03/29 16:21:00 INFO fs.TestDFSIO:      Throughput mb/sec: 11.087580800745085
15/03/29 16:21:00 INFO fs.TestDFSIO: Average IO rate mb/sec: 11.08777141571045
15/03/29 16:21:00 INFO fs.TestDFSIO:  IO rate std deviation: 0.04630139928117125
15/03/29 16:21:00 INFO fs.TestDFSIO:     Test exec time sec: 92.653
15/03/29 16:21:00 INFO fs.TestDFSIO:
```

Figure 2: Data from DFSIOtest output

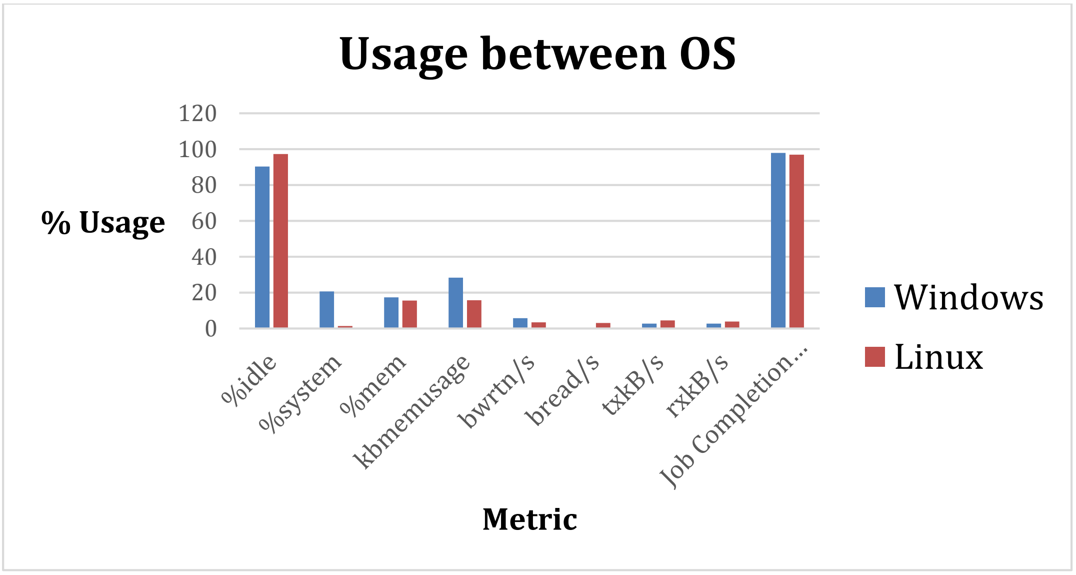


Figure 3: Usage percent between the different metrics.

Cygnus Flash X-Ray Performance

Alex Hanneman
ABHanneman10@winona.edu
Winona State University- Computer Science/Math

ABSTRACT

Cygnus is a flash X-ray generator that is used by National Security Technologies in Las Vegas, Nevada to record images of small-scale nuclear tests. The nuclear tests going on take a lot of preparation and unfortunately sometimes the x-ray shot produced is of poor quality, which is unknown for several hours. Meanwhile, as the x-ray produces a shot, machine diagnostics data is captured containing voltage and current, which can be used right away. Thus, signal-processing techniques such as cubic splines and filters can be applied along with general statistical methods to find trends and correlations for the machine diagnostics and x-ray images. These results will save ample time on Cygnus X-ray experiments.

Keywords

Cygnus, X-ray, Sub-critical nuclear, Cubic spline, Filter, Signal processing.

1. INTRODUCTION

The National Security Technologies LLC (NST) is a leader in research in homeland security, nuclear and nonnuclear experiments, physics modeling, and radiological detection. They are contracted by the Department of Energy and these areas of research are very important as they relate to the nation's security. As apart of the Defense Experimentation and Stockpile Stewardship Directorate, the NST conducts time-consuming, expensive sub-critical nuclear experiments at the Nevada testing site just outside of Las Vegas. During these experiments they will use, high-speed diagnostic instruments to measure radiography. This instrument is called the Cygnus Flash X-ray. Cygnus is a flash X-ray generator that is used by the NST to record images of small-scale nuclear tests. The X-ray is pulsed through a scene where a scintillator (a substance that exhibits luminescence when struck by a light of certain wavelength; that produces a spark or flash) collects the unabsorbed X-ray pulses and the data is read using an oscillator (a circuit that produces an alternating output current of a certain frequency determined by characteristics of the circuit components) [1]. These measurements are used for the machine diagnostics (such as current and voltage readings) which are helpful and produced immediately, but do not measure performance. The machine performance is a separate process that takes almost an hour to complete, which is a problem because bad X-ray shots can be produced. Bad shots are typically due to a component in the X-ray failing but can take quite a bit of work to figure out.

The NST would like to save time by figuring out if a part is failing and prevent a bad shot from being produced. There has been no work done on this particular project so far. So, the goal of this project is to characterize the performance of the Cygnus machine using machine diagnostic data. To achieve the goal each diagnostic is to be characterized as a function of time (shot), and each shot is to be characterized as a function of diagnostic.

The diagnostic data includes current and voltage at roughly around 28 different sensors along each of the Cygnus X-ray machines. Each of the sensors has a digitizer (used to convert to digital form for use in a computer) that captures a reading around every 5 nanoseconds. Below is a plot of the data given from one sensor for voltage readings. The data follows the trend of have a spike of energy pass through followed by a ringing that the sensor captures. The ringing is due to the energy that moves through leaves a rebound that bounces in the machine and slowly dampens to return to the baseline. The vertical lines on the plot indicate what is considered the start and stop of the peak; this will be discussed later.

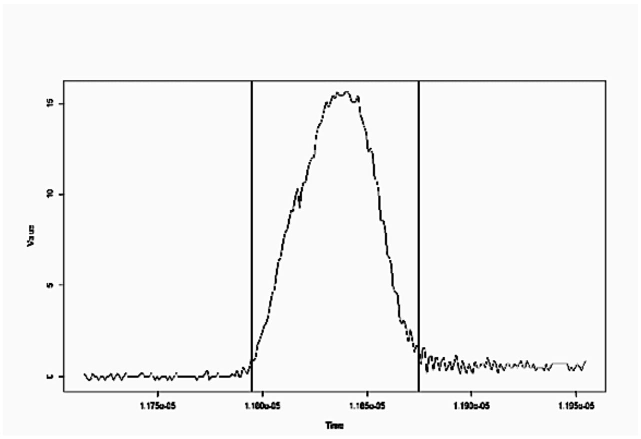


Figure 1. Data given by Cygnus x-ray machine diagnostics.

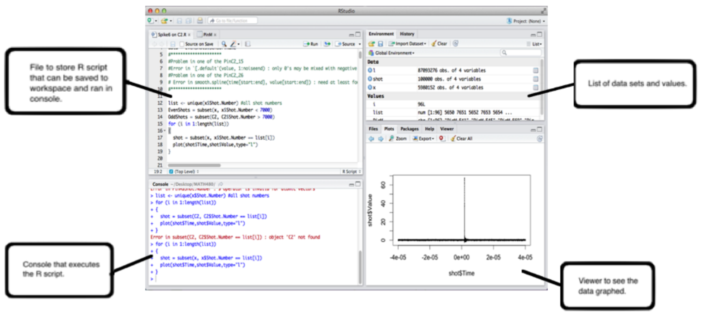
2. Hypothesis

Machine diagnostics from the Cygnus x-ray will correlate to a measurable radiation dose produced by the x-ray shot.

3. Methods

This research project incorporates some software development, in R, in data analysis. Specific areas of research will include how the Cygnus X-ray machine works with components correlating to diagnostic data. Figure 2 shows a screen shot of R Studio (software for R) running. R Studio is a beneficial integrated development environment because it provides everything you need to use in one window, as depicted in figure 2.

Another benefit to R is the availability of packages to help speed up the statistical programming process.



```
#######################
#Problem in one of the PinC2_15
#Error in `[.default`(value, 1:noiseend) : only 0's may be mixed with negative
#Problem in one of the PinC2_26
# Error in smooth.spline(time[start:end], value[start:end]) : need at least fo
#######################

list <- unique(x$Shot.Number) #all shot numbers
EvenShots = subset(x, x$Shot.Number < 7000)
OddShots = subset(C2, C2$Shot.Number > 7000)
for (i in 1:length(list))
{
  shot = subset(x, x$Shot.Number == list[i])
  plot(shot$Time,shot$Value,type="l")
}
```

```
Data
1      87093276 obs. of 4 variables
shot   100000 obs. of 4 variables
x      5980152 obs. of 4 variables
Values
i      96L
list   num [1:96] 5650 7651 5652 7653 5654 ...
```

```
Console ~/Desktop/MATH480/
> list <- unique(x$Shot.Number) #all shot numbers
> for (i in 1:length(list))
+ {
+   shot = subset(C2, C2$Shot.Number == list[i])
+   plot(shot$Time,shot$Value,type="l")
+ }
Error in subset(C2, C2$Shot.Number == list[i]) : object 'C2' not found
> for (i in 1:length(list))
+ {
+   shot = subset(x, x$Shot.Number == list[i])
+   plot(shot$Time,shot$Value,type="l")
+ }
>
```

Figure 2. R Studio integrated development environment window.

3.1 Prepare Data For De-noising

First, before we even begin applying de-noising methods, we first need to eliminate the data from where the data plots where the sensor reading is unusable. One example of unusable data is shown in figure 3, which includes only picking up the baseline noise and not reading the actual burst of energy passing through the Cygnus. This could be due to the time of the sensor was not started and stopped at the correct time. Figure 4 is when the sensor picks up the spike, but then the ringing never returns to baseline. This could be due to remaining energy still rebounding in the x-ray. Figure 5 is when the sensor shows more than one spike, which is not what should be shown.

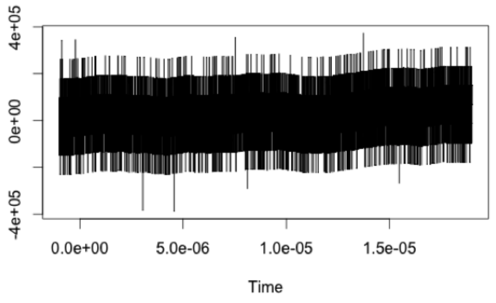


Figure 3. Unusable data, only baseline is picked up.

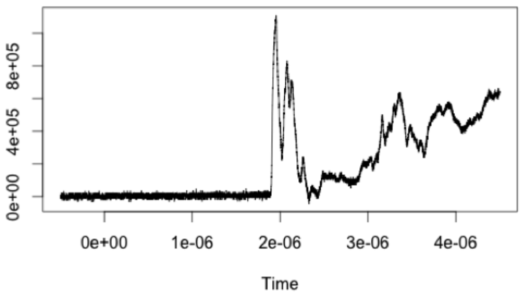


Figure 4. Unusable data, ringing never returns to baseline.

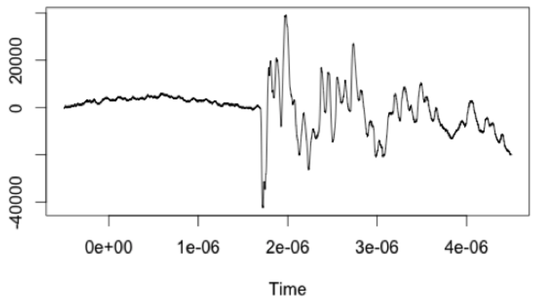


Figure 5. Unusable data, more than one spike.

The data is then checked to see if the noise is background noise or if it is random. To do this, we graphed the values of how far the data is away from where the average value. Figure 6 shows the distribution of noise. This graph shows that the distribution of the noise is slightly positive, and evenly distributed. The digitizer of the sensor causes the positive average. The digitizer is constantly picking up energy so the ground zero is actually slightly positive. Thus, this means that the noise is background noise is not to be included in the data.

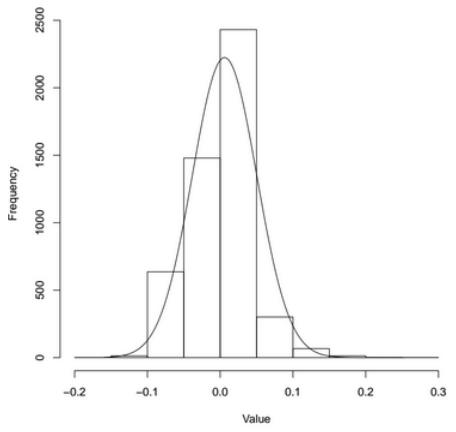


Figure 6. Column graph showing the noise is approximately Gaussian (even distribution).

Before the data can be de-noised, we must find where the spike starts and ends since that is the important part of the data we want to examine. To do this, we find the beginning and ending of the peak by finding where it jumps above and returns to the baseline.

3.2 Apply De-noising Filter and Cubic Splines

Machine diagnostics must be cleaned since there is a lot of noise in the data. This is important to get a "clean" set of data, since it is hard to know right away if the data is deviating because of noise or not [2]. In de-nosing the data we used a few different methods, which include cubic splines, moving average filter, and a weighted-binomial filter.

The moving average and binomial filters analyze data by calculating a series of averages of different subsets of the full data set. The difference between the two filters is the weighted value distributed to the points being evaluated. These filters are crucial for removing the noise and these filters will be discussed in further detail in Adam Grupa's (partner) paper.

After one of the filters is applied to the noisy data, we applied a cubic spline. Cubic splines apply polynomial curves the data; where the polynomials curves are connected at evenly distributed points among the data, called knots. At each knot where both polynomial curves meet, both cubic polynomials on each side of the knot must have matching signs of their first three derivatives [3]. This means if one cubic polynomial has the signs 1: +, 2: -, 3: +, then the other cubic polynomial must have the signs 1: +, 2: -, 3: +. Having matching signs ensures that when one polynomial connects to the other they are continuous and smooth.

Figure 7 shows a cubic spline applied to noisy data. To apply the cubic splines we used *smooth.spline()*, which is a built-in function in R. The vertical lines on the plot indicate the start and end of the spike. Identifying the spike is crucial when using cubic splines because if the spline is applied to the baseline and spike, the curves do not fit very well. But, when applied just to the spike, the curve came out to be very close. The cubic spline are important because they reduce the mean square error and so we will be apply the splines after we apply the moving average or binomial filter.

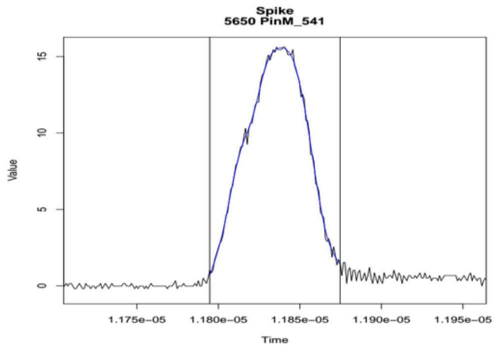


Figure 7. Cubic spline applied to noisy data excluding baseline, the smooth curve is blue.

3.3 Analysis of Filter Choices

To compare the 2 data de-noising methods, we used three metrics-full-width half-height, area under the curve, and rise time, all of which will be explained later. To compare these un-filtered, moving average filter, and binomial filter, we made graphs showing the percent of total of what the metrics came out to be for all shots on one specific sensor. From this comparison it was found that the filters concentrated the data more to the spike of the data. Also the binomial and moving average filter produce very curves. Thus, these results show the filters are doing what we need. The comparison of filter choices is discussed in greater detail in Adam Grupa's (partner) paper.

4. Metrics Collected for Analysis

For pages other than the first page, start at the top of the page, and continue in double-column format. The two columns on the last page should be as close to equal length as possible.

Full-width half-height is important because it will show us the general shape of the peak, without including the extra ringing before returning back to baseline. It calculates the area under the curve from the top of the spike, down to half the height of the peak.

Area under the curve tells us how large the spike is and is found by using a trapezoidal sum function.

Rise time is the amount of time it takes to reach the peak.

Baseline is the value of which nothing is happening in the data being required, so this any data a part of the baseline is set to zero because we do not want that included in the data being evaluated.

Standard deviation is included for the noise.

Peak is indicating whether the spike was positive or negative, which is important for calculating other metrics. Some sensors recorded negative spikes and are still usable.

Max and Min height is the maximum height and minimum height of the data.

Range is the range of time values for start to end of the spike.

5. Results

To see if the machine diagnostics correlate to a shot radiation dose, we developed a model using linear regression (independent variables being used to linearly predict the dependent variables). The model uses area under the curve and full-width half-height, because those two metrics were found to have the lowest percent error. The model resulted in reasonable shot radiation dose predictions with a few outliers. This model will continue to be improved and the analysis is discussed in detail in Adam Grupa's (partner) paper.

6. Future Work

Correlations will be tested against data but it will not be a complete model because we are given a limited data set. If we believe we have found a reasonable model we may also decide to use cross-validation to improve the model. Solutions found by our research group will be given to the NST where they will continue researching the solutions in the future. One thing to note is when analyzing make sure the data from one sensor can be compared to other sensors. For example, at different sensors, the value of the voltage and current readings is scaled as a security precaution of NST. Thus, when comparing data the data can only be compared for each specific sensor with the same scale. The NST should be able to eliminate the various scales and then compare across multiple sensors.

7. Acknowledgements

Dr. Luttman (Industry contact), Dr. Schmidt (Mathematics), Dr. Zhang (Computer Science), and Dr. Debnath (Computer Science) for direction on the project. The Mathematical Association of America, National Science Foundation, and Society for Industrial Applied Math for funding of PIC Math program.

8. References

[1] B. V. Oliver, et al., "Characterization of the rod-pinch diode X-ray source on Cygnus," In: Pulsed power conference 2009, 2009, p. 11-16.

[2] Prandoni, Paolo. Vetterli, Martin. "Signal Processing for Communications," EPFL Press, 2008. P. 19-143.

[3] Gareth, James, et al. "An Introduction to Statistical Learning with Applications in R," Springer, p. 59-297.

[4] B. West et al., "Linear Mixed Models: A Practical Guide Using Statistical Software," Second Edition, CRC Press, 2014.

[5] D. Mosher, et al., "Rod-pinch X-radiography for diagnosis of material response," in: 42nd Annual Meeting of the APS Division of Plasma Physics combined with the 10th International Congress on Plasma Physics. #PO2.001.

[6] J. Smith, R. Carlson, et al., "Cygnus dual beam radiography source," In: Pulsed power conference, 2005 IEEE, 2005, p. 334-337.

[7] C. Courtois, et al., "Characterization of an MeV Bremsstrahlung X-ray source produced from a high intensity laser for high areal density object radiography," Physics of Plasmas, 2013, p. 20.

9. Appendix

The source code includes not the complete program, but a generalized version to show the essential components.

spike6:

Purpose: Driver function of the program, which switches between using the data as unfiltered, moving average filtered, and binomially filtered. It then applies the cubic spline and assigns the metrics using external functions.

Input: *list*- a list of all shots containing a specified sensor name. *dataset*- the complete data set containing shot values corresponding to shot times.

```
spike6=function(list,dataset)
{
  for(h in 1:length(list)){
    data=dataset[dataset$Sensor.Name==list[h],]
    value=data$Value
    time=data$Time
    for(aaa in 1:3){
      switch(aaa,
        value <- value,
        value <- avgFilter(value, 10),
        value <- binFilter(value, 6))

        #apply cubic spline to curve
        smooth=smooth.spline(time[start:end],value[start:end])
        fit=smooth$y
        fittime=smooth$x
        start=0
        end=length(value)

        #calculate metrics
        fwhh=calcFwhh(fittime,fit)
        int=integrate(fittime,fit)
        risetime=fittime[which.max(fit)]-fittime[1]
        baseline = findBaseline(fit[1:100])
        standarddev=sd(noise)
        max=max(fit)-mean(noise)
        min=min(fit)-mean(noise)
        peak=calcPeak(value,max,min)
        range=time[end]-time[start]
      }
    }
}
```

integrate:

Purpose: Calculates area under the curve of the function by using a trapezoidal sum.

Input: *shotTimes*- time values for data in the shot. *shotValues*- values corresponding to shot times.

Output: *int*- the floating point value of area under the curve.

```
integrate=function(shotTimes,shotValues)
{
        install.packages("zoo")
        library("zoo")
        id <- order(shotTimes)
        int <- sum(diff(shotTimes[id])*
        rollmean(shotValues[id],2))
        return(int)
}
```

calcFwhh:

Purpose: Calculates the full-width half-height of the curve.

Input: *fittime*- time values for data in the shot. *fit*- values corresponding to shot times.

Output: *int2*- the floating point value of area under the curve of the peak of the curve.

```
calcFwhh=function(fittime,fit){
    midend=findEnd(fittime,fit)
    midstart=findStart(fittime,fit)

    fwhh=fittime[midend]-fittime[midstart]
    fwhhfit=fit[midstart:midend]
    fwhhfittime=fittime[midstart:midend]
    int2 = integrate(fwhhfittime,fwhhfit)
        return(int2)
}
```

calcPeak:

Purpose: Calculates the sign of the peak.

Input: *value*- value of the peak point. *tempmax*- the highest value found on the curve. *tempmin*- the lowest value found on the curve.

Output: *Peak*- the string indicating whether the curve is “Positive” or “Negative”.

```
calcPeak(value, tempmax, tempmin)
    baseline=mean(value)+sd(value)
    baseline2=mean(value)-sd(value)
    if(baseline<tempmax | baseline2>tempmin){
        Peak = “Negative”
    }
    else{
        Peak="Positive"
    }
    return(Peak)
}
```

findNoise:

Purpose: find the start and end of the noise.

Input: *shotValue*- values of data points.

Output: *noise*- number of data points ranging from the start of the noise to the end of the noise.

```
findNoise = function(shotValues){
    upperThreshold = mean(values) + sd(values)
    lowerThreshold = mean(values) - sd(values)
    noiseBegin = 1
    signalBegin = 1
    while((values[signalBegin]<upperThreshold)&&
        (values[signalBegin] >  lowerThreshold)){
        signalBegin = signalBegin + 1
    }
    noiseEnd = signalBegin - 100
    if(noiseEnd > 300){
        noise = values[noiseBegin:noiseEnd]
        return(noise)
    }
    else {
        return(NULL)
    }
}
```

findBaseline:

Purpose: calculate the baseline of the curve.

Input: *shotValue*- values of data points.

Output: *baseline*- the average of the filtered data

```
findBaseline = function(shotValues){
    noise = findNoise(values)
    filt = avgFilter(noise, 10)
    baseline = mean(filt)
    return(baseline)
}
```

findStart:

Purpose: calculate where the spike starts and jumps above the baseline.

Input: *fittime*- time values of the shot. *fit*- values corresponding to time values.

Output: *midStart*- time value where the spike starts.

```
findStart(fittime,fit){
    midstart=0
    max=max(fit)
    mid=max/2
    k=1
    while(midstart==0){
        if(fit[k]>mid){
            midstart=k
            for(l in 1:5){
                if(is.na(fit[k+l])){ }
                else{
                    if(fit[k+l]<mid){
                        midstart=0
                    }
                }
            }
        }
    }
    return(midstart)
}
```

findEnd:

Purpose: calculate where the spike ends and returns to the baseline.

Input: *fittime*- time values of the shot. *fit*- values corresponding to time values.

Output: *midEnd*- time value where the spike ends.

```
findEnd(fittime,fit){
    midend=0
    max=max(fit)
    mid=max/2
    k=1
    while(midend==0){
        if(fit[k]>mid){
            midend=k-1
        }
        k=k+1
        if(k==length(fit)){
            midend=k
        }
    }
}
```

Analysis and Inference of Cygnus Shot Quality Using Machine Diagnostics

Adam Grupa
Winona State University
agrupa06@winona.edu

ABSTRACT

Cygnus is a flash X-ray generator used by National Security Technologies at the Nevada National Security Site to record images of subcritical nuclear experiments in support of the U.S. Stockpile Stewardship program. Experiments performed with Cygnus are expensive and must be scheduled far in advance, so it is critical Cygnus performs correctly during each experiment. However, the process used to determine the usability of an experiment takes several hours. During the course of an experiment, machine diagnostics are collected from electrical sensors along Cygnus. These diagnostics are available immediately, so it would be useful if experiment usability could be determined from them. We analyze these electrical diagnostics using signal processing techniques to determine characteristics that exist in the signals, and then use those characteristics to create a predictive model that allows us to infer the usability of an experiment.

Keywords

Cygnus, flash X-ray radiography, signal processing, filters, smoothing splines, linear regression

1. INTRODUCTION AND HISTORY

1.1 Supercritical Nuclear Testing

In 1945, the United States (US) government began testing nuclear weapons, starting with Trinity test in New Mexico, which arose from the Manhattan Project [1]. These supercritical tests were mainly focused on figuring out the military applications of the weapons, as well as experimenting with new weapon designs. Until 1963, most of the nuclear tests that were conducted were atmospheric or exoatmospheric [1]. Even though precautions were taken to test weapons in unpopulated areas, such as the Nevada Test Site, the atmospheric nature of the tests often meant radioactive fallout would be dispersed far from the detonation zone of the weapon. Coupled with unpredictable weather patterns, tests would lead to unintended contamination of inhabited areas or water and food supplies.

Figure 1: Upshot-Knothole Grable Test

A prime example of this was the Castle Bravo test in 1954, where the US detonated a fusion hydrogen weapon in the Marshall Islands. Researchers underestimated the yield of the bomb and the amount of radioactive fallout it would produce. In addition, the weather pattern changed, leading to the spread of fallout over nearby populated islands. Even though the islands were evacuated as soon as possible, many of the inhabitants suffered from radiation burns and poisoning, resulting in an increase in radiation related illnesses, such as cancer and birth defects.

The negative effects of unabated atmospheric testing were cause for concern for many of the worlds countries, eventually resulting in the Partial Test Ban Treaty in 1963, which the US signed. As part of the treaty, the US halted all atmospheric tests and moved testing underground [1]. Instead of open air detonations, holes would be drilled hundreds of meters into the ground and nuclear weapons would be placed in them. The resulting explosions excavated huge amounts of earth, creating large subsidence craters up to a kilometer in diameter.

1.2 Subcritical Nuclear Testing

Part of the impetus for continuing nuclear testing in the US was the arms race between them and the Soviet Union during the Cold War. With the end of the Cold War in sight around 1991, aggression between the two blocs was rapidly declining, so the Soviet Union declared a moratorium on all future nuclear weapons testing. This led the US to reexamine its own nuclear testing policy, resulting in the US' nuclear testing moratorium of 1992, after which no nuclear weapons have been produced or tested. In addition, the

US signed (but did not ratify) the Comprehensive Test Ban Treaty in 1996 [1], which ended nuclear testing for most countries around the world. However, this created a problem for the caretakers of the US aging nuclear stockpile. How could they be sure the weapons will still perform as expected if they cannot conduct supercritical tests?

To solve this problem, the US Department of Energy (DOE) began the Stockpile Stewardship program, which employs national laboratories overseen by the National Nuclear Security Administration (NNSA), such as Los Alamos National Laboratory and Sandia National Laboratory, to test and maintain the current stockpile of nuclear weapons without conducting any supercritical tests [5]. Part of the solution to this problem involved the use of computers for computational modeling. Computer simulations of nuclear tests are used to verify the reliability of the stockpile. But this of course creates another problem. How do researchers know that the models are correct?

The answer to this question lies in breaking the use of the model into separate stages and performing small-scale physical tests analogous to the simulation created by the model. First, the model is used to create a small-scale subcritical nuclear test simulation. Second, focused experimentation with physical subcritical tests are conducted to verify the model. If the physical experiments agree with the computational model, then the model is scaled up and used to create simulations of full supercritical nuclear tests.

National Security Technologies (NSTec) is a company that is contracted by the DOE and, as part of its contract, helps perform the focused experiments used to verify the computational models produced through the Stockpile Stewardship program [1]. These experiments involve performing subcritical nuclear tests and using flash X-ray radiography to help analyze the results. X-rays are useful since not only can they be used to capture images of subcritical explosions, they can also reveal the internal structure of the explosion, which is not available using regular photography.

1.3 Cygnus

The machine that generates the X-rays used for these experiments at NSTec is called Cygnus (Figure 2), which is actually two X-ray generators that run in parallel (Cygnus 1 and Cygnus 2), meaning there are two X-ray shots per experiment, resulting in two images [5]. The quality and usability of the produced images is correlated with the dose of radiation produced by the X-rays for an experiment. This dose is often called *shot quality*, and is recorded separately for both machines. Cygnus must maintain a high degree of accuracy in order to provide useful information for verifying the computational models and must produce usable results in 199 out of every 200 experiments. When an experiment is unusable, the images that it produced are unclear and the corresponding shot quality is low.

A problem faced by the researchers working with Cygnus is that the process used to determine the shot quality can take up to several hours to complete, meaning they must wait after an experiment to determine if it has produced usable information or not. As might be expected, it takes a substantial amount of time and money to schedule and

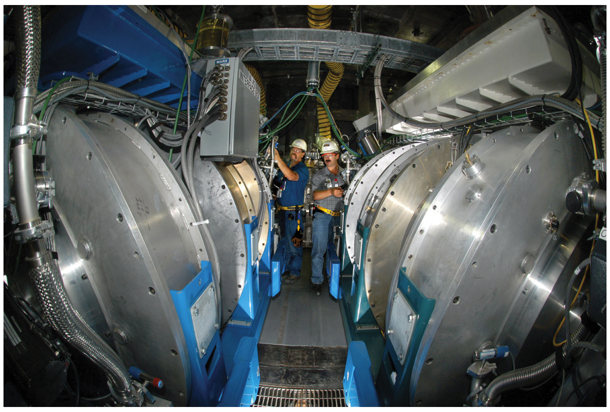


Figure 2: Technicians working on Cygnus

perform an experiment on Cygnus, so it would be beneficial if researchers could tell if an experiment was usable or not immediately, without having to determine shot quality. Luckily, Cygnus produces a series of machine diagnostics for each experiment which are available immediately after the experiment is conducted.

These machine diagnostics are discrete electrical signals produced by a battery of approximately 28 sensors (sensors are sometimes added or taken away) that measure voltage or current and are placed along Cygnus at various points until the point where X-rays are produced. Sensors near the end of Cygnus tend to be more important, since they are close to the point of X-ray production. In addition, the sensors are digitizers, meaning they have a finite resolution and are only capable of producing certain values, so certain signals may be quantized to values that do not represent the true signal values.

We worked with NSTec to see if it was possible to predict the shot quality of an experiment based upon these machine diagnostics. We were given machine diagnostic data for 100 experiments from both Cygnus 1 and Cygnus 2 along with the corresponding shot quality for the experiment. The data was also separated into two time periods, with the first being before Cygnus underwent significant maintenance, and the second being after the maintenance was finished. Therefore, the data can be divided into four distinct subsets indexed by what machine was used and time period. Using techniques from signal processing, we were able to obtain useful metrics from the diagnostics generated during the experiments. These metrics were then applied to a linear regression model which had moderate success at predicting shot quality.

In section 2, we give a brief overview of the characteristics of the data we worked with. In section 3, we go over the details of the methods we used to remove noise from the data. Finally, in section 4, we present the predictive model we created to predict shot quality and analyze the results of the model.

2. DATA DESCRIPTION

Examples that are characteristic of the signals obtained from the machine diagnostics are given in Figure 3. Note the general trend of the signals is to begin with pure noise, rapidly increase or decrease in value, and then return to some baseline. The noise is caused by fluctuations of energy in the sensor circuit primarily due to thermal agitation or defects. What this means is that even when the machine is not producing X-rays, the sensors are still reading some electrical values. Once Cygnus is activated, this noise is then compounded on top of the actual signal which distorts it slightly.

In addition, since the sensors never stop recording electrical noise and settle down, there is no true zero for the sensors. Instead, we must determine what an appropriate baseline is for each signal based on its noise distribution and use this as the zero. We can see in the examples that the noise hovers around a zero value before the spike in the signal, where the spike is the first time the signal rapidly increases or decreases and then returns to baseline. Some signals, such as the second example in Figure 3, never returned to a baseline value, which made them harder to use in our analysis. In addition, not all of the spikes begin and end at the same time, instead occurring as electricity moves along Cygnus to eventually produce X-rays.

Since the signals are digital, they are made up of a discrete number of points which represent the value of the signal at a certain place in time. We have added lines connecting each point to its adjacent neighbor to make the graphs more visually pleasing, but this is not necessary. We want to be able to refer to the value of any particular point in a signal, so we define notation to do so. Given a signal y, then $y[i]$ refers to the value of the i^{th} point in the signal. For example, if $y = (4, 6, 2, 8)$, then $y[1] = 4$ and $y[3] = 2$.

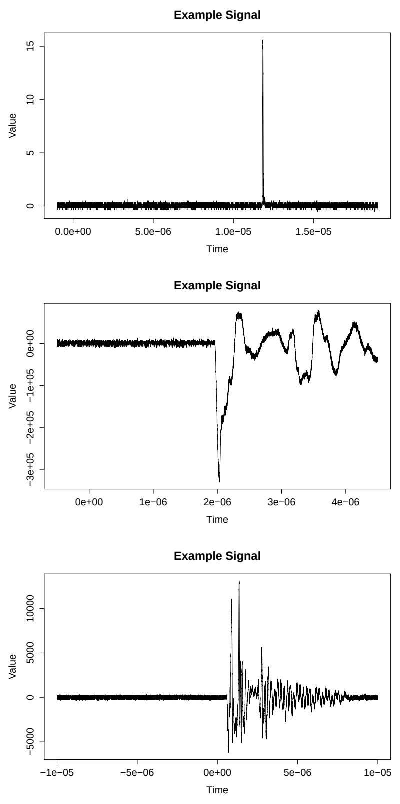


Figure 3: Example Signals

3. NOISE REMOVAL TECHNIQUES

Because the inherent noise in the signals was affecting the signal values, statistics and metrics we wanted to calculate from the signals would be changed, so we looked for ways to remove the noise while keeping the signal intact. We primarily used two methods to remove the noise; digital signal filters and smoothing splines. These would in turn help us determine an appropriate baseline for each signal which could then be used to calculate metrics on the signal.

3.1 Digital Signal Filters

Digital signal filters are essentially any process that removes an unwanted feature from a signal. In general, the most common filters are those that attempt to remove noise from a signal, thereby revealing the true values of the signal. Other common filters include bandpass filters, such as high-pass and low-pass filters, which attenuate parts of a signal with certain frequencies while letting other frequencies pass [4]. A direct application of this kind of filter would be a radio tuner, which filters out all radio frequencies except the one being tuned to.

In signal processing, filters take the form of functions designed with a specific purpose in mind which are then applied to an input signal to produce an output signal [6]. To apply the filter to a signal, the signal is represented as a function and convoluted with the filter. For an analog signal f and an analog filter g, their continuous convolution is defined as

$$(f * g)(t) = \int_{-\infty}^{\infty} f(x)g(t-x)dx. \tag{1}$$

Here, x is some dummy variable for use in the functions and t usually represents time, but it does not have to. What is essentially happening in the convolution operation is that the function f, the input signal, is being weighted by the function $g(t-x)$, the filter, where g has been horizontally shifted along the x-axis by some amount t. Increasing or decreasing t 'slides" g along the x-axis and emphasizes where it overlaps with f due to $f(x)$ and $g(t-x)$ being multiplied. Since our signals are discrete, however, we use the discrete convolution, which is defined as

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m] \tag{2}$$

for some input signal f and filter g and where m and n are integers [4]. In addition, our input signal and filters are represented by a finite sequence of values, so we can imagine padding the sequences with zeros in order to use them in the infinite definition of the discrete convolution. However, this is not necessary, since whenever the two sequences do not overlap, they will multiply to zero anyway. Thus, we need only evaluate the sum in the convolution from the lower bound of g to its upper bound. This gives us the definition

$$(f * g)[n] = \sum_{m=A}^{B} f[m]g[n-m] \quad (3)$$

where A and B are the locations of the first and last non-zero values of g [4].

The filters we examined were a moving average (MA) filter and a binomial filter.

3.1.1 Moving Average Filter

The MA filter is one of the simplest filters that can be implemented. The general idea of a moving average filter is to take in an input signal, x, and for each data point $x[i]$ take the average value of the data points of the signal in a small band around that point [6]. The definition of a MA filter is

$$y[i] = \frac{1}{2M+1} \sum_{j=-M}^{M} x[i+j] \quad (4)$$

where x is the input signal, y is the output, or filtered, signal, and $2M+1$, the width, is the number of data points around the i^{th} data point that are being averaged, where M is the number of data points to look ahead or behind. Notice we can also define the MA filter as

$$y[i] = \frac{1}{2M+1}x[i-M] + \cdots + \frac{1}{2M+1}x[i] + \cdots + \frac{1}{2M+1}x[i+M].$$

Here, it is apparent the filter is just assigning weights to the points and taking their sum. This means we can also define a MA filter by the weights it places on data points, listed as $(\frac{1}{2M+1}, \ldots, \frac{1}{2M+1})$ where the number of weights is $2M+1$. This also means we generalize the MA filter to a weighted average filter, with weights a_i listed as $(a_{-M}, \ldots, a_i, \ldots, a_M)$ [6].

As an example, consider the signal represented by the sequence $x = (0, 5, 1, 9, 2, 7)$. We would like to apply an MA filter of width 3 to x. This means, for each data point i,

$$\begin{aligned} y[i] &= \frac{1}{3} \sum_{j=-1}^{1} x[i+j] \\ &= \frac{1}{3}(x[i-1] + x[i] + [i+1]) \\ &= \frac{1}{3}x[i-1] + \frac{1}{3}x[i] + \frac{1}{3}[i+1]. \end{aligned}$$

For example, the third data point in y is $y[3] = \frac{1}{3}(5+1+9) = 5$. The stem plots of x and the filtered signal y are given in Figure 4. Notice the missing values near the edges. This is always a problem when using filters on finite signals that attempt look ahead or behind the current data point. Eventually, they will reach the end of the signal and will not be able to compute a new value. Usually this is remedied by padding the signal with extra data points with value 0 on both ends that are as wide as the filter being used.

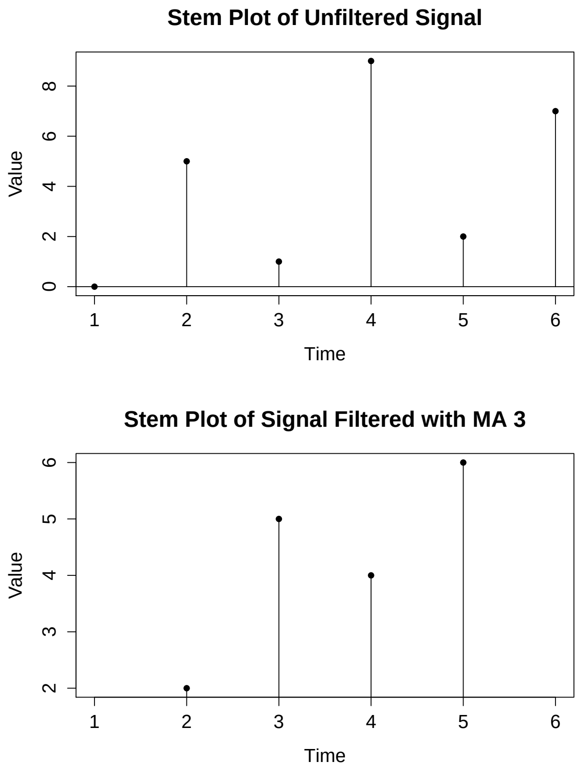


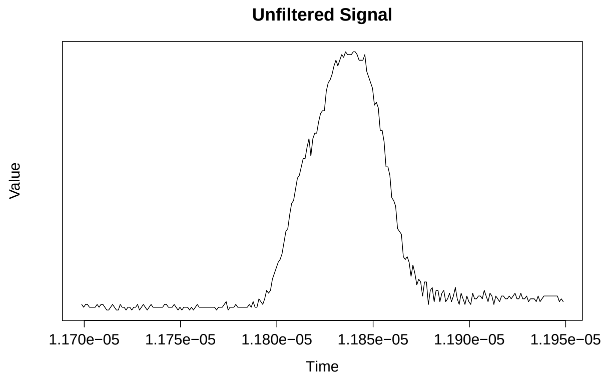
Figure 4: Moving Average Example

In Figure 5 is an example of a MA filter being applied to one of the signals in our dataset. Notice that the filter does a good job of reducing the variance of the baseline noise before and after the spike in the signal, however, it does not accurately follow the curve of the spike, and even reduces the peak value of the spike significantly. The reduction in peak value occurs because the filter is picking up some of the values in the baseline noise, which are near zero. We would like to use a filter that helps to eliminate the variance in the signal caused by noise, but that also does not affect the true value of the signal too much. Because of this, we examined binomial filters.

3.1.2 Binomial Filter

A binomial filter is fairly similar to a moving average filter, in that, instead of weighting each data point evenly, it applies more weight to data point closer to the input point, and less weight to those point farther away. A binomial filter is defined as

$$y[i] = \sum_{j=-M}^{M} \frac{x[i+j]}{2^{|j|+1}} \quad (5)$$



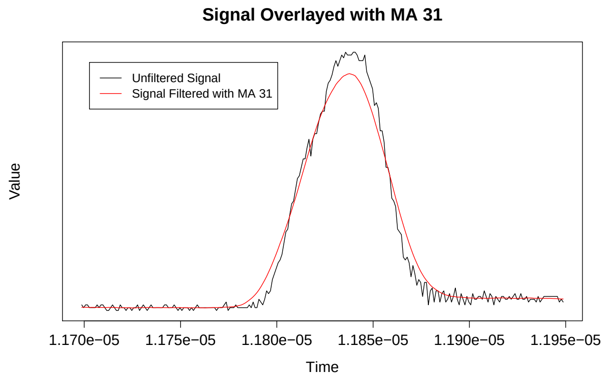


Figure 5: Effects of Moving Average Filter on Signal

Thus, the weights for the binomial filter look like

$$(\frac{1}{2^M}, \dots, \frac{1}{2^2}, \frac{1}{2^1}, \frac{1}{2^2}, \dots, \frac{1}{2^M}).$$

The binomial filter is more useful than the normal MA filter for two reasons. First, it weighs points more heavily that are closer to the point being calculated. This means that the filtered peak value of a signal will not be as affected by data points that are drastically different in value but far away. Second, the weights in the binomial filter approximately follow the probability density of a Gaussian distribution. The distribution of the noise in our signals is Gaussian, so the binomial filter will do a good job of removing that noise. Figure 6 shows an example of how the binomial filter affects our signals, using the same signal as in Figure 5. Notice that the variance of the noise is reduced significantly, but the filtered signal still closely follows the values of the original signal.

3.2 Cubic Splines

Cubic splines apply polynomial curves to continuous subsets of the data. The polynomials curves are connected at evenly distributed points among the data, called knots. At each knot, the two cubic polynomials on each side of the knot must have matching signs of their first three derivatives [3]. This means if one cubic polynomial has the signs 1: +, 2: -, 3: +, then the other cubic polynomial must have the signs

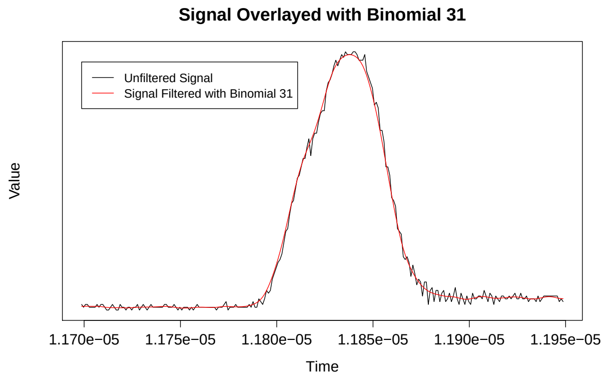


Figure 6: Effects of Binomial Filter on Signal

1: +, 2: -, 3: +. The reason for this is to ensure that the entire fit is continuous and smooth.

Figure 7 shows a cubic spline applied to noisy data. The vertical lines on the plot indicate the calculated start and end of the spike. It is crucial to find these when using cubic splines. When the spline is applied to the baseline and spike, the curves do not fit very well. But, when applied just to the spike, the curve came out to be very close. Cubic splines are useful because they minimize the mean square error while retaining the original characteristics of the curve.

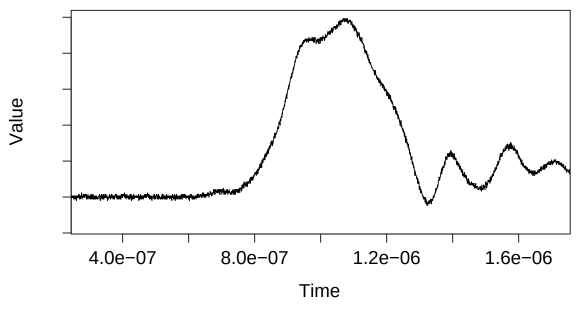
4. MODEL AND ANALYSIS

Using the metrics that we calculated on the signals, we produced a simple linear regression model to try and predict the shot quality of the experiments [3]. The data we used to fit our model came from the subset of experiments for the first Cygnus machine in the earlier time period. For the features of our model, we decided to use two important sensors near the end of the Cygnus machine.

We chose these two because these sensors are closer to where the X-rays are being produced and we believe this means the signals captured by those sensors will have higher correlation with the shot quality. From those two sensors, we used the full-width half-height (FWHH) and the area under the curve (AOC) values calculated on those signals as the features of the model. In this case, FWHH is the width or range of the first spike of the signal, where the value of the spike is greater than half the maximum value for the entire signal. AOC is simply the area under the spike using the calculated baseline. We also include an interaction term between the FWHH and AOC, since part of the calculations for FWHH and AOC deal with deciding where the spike of a signal begins and ends.

The full model looks like

$$\begin{aligned} y_i = \beta_0 &+ \beta_1 FWHH_{1_i} + \beta_2 AOC_{1_i} \\ &+ \beta_3 (FWHH_{1_i} \times AOC_{1_i}) \\ &+ \beta_4 FWHH_{2_i} + \beta_5 AOC_{i}{}_2 \\ &+ \beta_6 (FWHH_{2_i} \times AOC_{2_i}), \end{aligned} \tag{6}$$



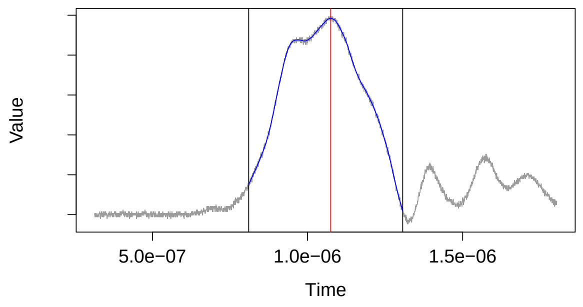


Figure 7: Cubic Spline Fitted to Spike

where y_i is the predicted shot quality of the i^{th} experiment, the β_n are the coefficients for the features, a subscript of 1 indicates a metrics from the first sensor, and a subscript of 2 indicates a metric from the second sensor.

We attempted to fit this model using metrics calculated from both unfiltered signals and signals filtered with a binomial filter to see if the filtering offered any appreciable difference in the fit of the model. Surprisingly, when we applied the model to the unfiltered metrics, we observed that the model performed fairly well.

4.1 Unfiltered Model

All of the p-values associated with the model feature coefficients were below 0.05, suggesting that the coefficients are all close to their true value, assuming the model is correct [3]. Notable features were AOC_1 and $FWHH_1 \times AOC_1$, whose associated coefficients both had p-values below 1.0×10^{-7}. The multiple R-squared statistic calculated for the model was approximately 0.9, indicating the model was able to predict shot quality fairly well [3]. In addition, the p-value of the F-statistic associated with the model was below 5.0×10^{-9}.

Looking at the residual plot for the model in Figure 8, we can see that there is no discernible pattern occurring in the residuals, indicating that a linear relationship probably models the true relationship between sensor values and shot quality well.

4.2 Binomial Filtered Model

We next attempted to fit the model using metrics that had been calculated from signal that were filtered with a binomial filter. All of the p-values associated with the feature coefficients were still below 0.05, however, most of the p-values increased slightly, meaning there is less certainty that the calculated values are closer to the true values for the coefficients [3]. In particular, the p-values for AOC_1 and $FWHH_1 \times AOC_1$ were now below 1.0×10^{-8}, an increase by an order of magnitude. Additionally, the p-value associated with the F-statistic decreased slightly and was below 3.0×10^{-9}. Finally, the multiple R-squared value also increased slightly, and was approximately 0.9049, indicating that the model based on the filtered metrics was slightly better at predicting shot quality [3].

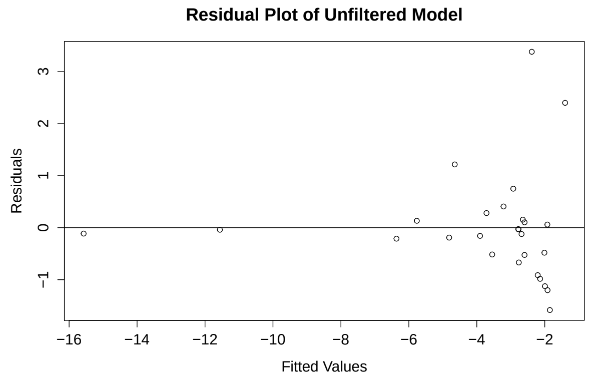


Figure 8: Residual Plot for Unfiltered Model

In Figure 9, we have the residual plot associated with the binomial filtered model. It is fairly similar to the residual plot for the unfiltered model, with the main difference being that most of the residuals have moved closer to 0, echoing the increased multiple R-squared value.

4.3 Analysis

The failure of binomial filtering to create a significantly superior model to the unfiltered model suggests a few things. First, is that the metrics we chose and the way we calculated them turned out to be robust to noise, meaning that filtering the signals would have little effect on the calculations of the metrics anyway. Second, is that the noise may not be having as strong of an effect on the signal as we supposed, so the filtering isn't actually removing a deleterious feature that is obscuring the true signal. Third, is that we have failed to verify some assumptions with regards to the model we have produced and so we are observing a spurious correlation between this signals and the shot quality.

5. CONCLUSION

We attempted to produce a model based on Cygnus' electrical machine diagnostics that can predict the shot quality of an experiment performed by Cygnus. After smoothing out the signals of the machine diagnostics using digital signal filters and smoothing splines, we calculated metrics from the signals with which to create a simple linear regression model. Surprisingly, the model we created was able to predict the shot quality of an experiment fairly well, even when fit using metrics from unfiltered signals. This shows that there is at least some correlation between the machine di-

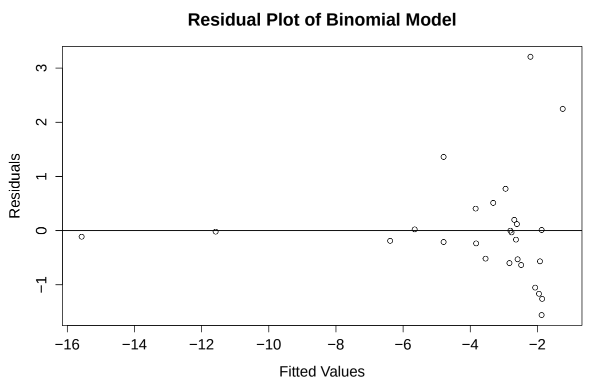


Figure 9: Residual Plot for Binomial Filtered Model

agnostics collected from Cygnus during an experiment, the experiment's shot quality and, by extension, the quality of the image captured during the experiment.

6. FUTURE WORK

There is still much work to be done regarding Cygnus and its machine diagnostics. We will further examine our model to see if it holds up under scrutiny. Notice that in Figure 8, there exist some points that seem like outliers, due to high leverage or high residual value [3]. We will see if the model remains intact after removing outliers that might cause bias in the fit of the linear regression. We will also apply the model we have created to the other remaining data subsets to see if it has the same degree of predictive ability. Finally, we will also try modifying the model by adding or removing other possible features or metrics to see how they affect the fit of the regression line.

7. ACKNOWLEDGEMENTS

We would like to thank Dr. Aaron Luttman from NSTec for providing us with the problem, the data to solve it, and suggesting possible avenues for research, Dr. Samuel Schmidt from Winona State University for finding the problem and administering the PIC Math course in which we worked on this problem, and the NSF, MAA, and SIAM for helping fund the PIC Math program.

8. REFERENCES

[1] United states nuclear tests july 1945 through september 1992. Tech. Rep. DOE/NV–209-REV 15, U.S. Department of Energy Nevada Operations Office, December 2000.

[2] de Boor, C. *A Practical Guide to Splines*. No. v. 27 in Applied Mathematical Sciences. Springer-Verlag, 1978.

[3] James, G., Witten, D., Hastie, T., and Tibshirani, R. *An Introduction to Statistical Learning: with Applications in R.* Springer Texts in Statistics. Springer, 2013.

[4] Prandoni, P., and Vetterli, M. *Signal Processing for Communications.* Communication and information sciences. EFPL Press, 2008.

[5] Smith, J., Carlson, R., Fulton, R., Chavez, J., Ortega, P., O'Rear, R., Quicksilver, R., Anderson, B., Henderson, D., Mitton, C., Owens, R., Cordova, S., Maenchen, J., Molina, I., Nelson, D., and Ormond, E. Cygnus dual beam radiography source. In *Pulsed Power Conference, 2005 IEEE* (June 2005), pp. 334–337.

[6] Smith, S. W. *The Scientist and Engineer's Guide to Digital Signal Processing.* California Technical Pub., 1997.

wARriorDrone API Development: AR.Drone 2.0 Node.js API Supporting Color Detection

Jeff Brookshaw, Joan Francioni, Mingrui Zhang, Narayan Debnath

Winona State University
715 338 5217

JBrookshaw09@winona.edu

ABSTRACT

The AR.Drone 2.0 is a remote-controlled quadcopter that is operated by computer instruction via Wireless Ad Hoc Network. There are existing API's to reduce the complexity of communications between the AR.Drone 2.0 and the computer. Developers have achieved impressive autonomous flight software using existing AR.Drone SDK's and OpenCV. However, implementing this software requires experience in computer graphics and computer vision, as well as the configuration of multiple independent libraries. This can be a difficult process for someone unfamiliar with the chosen toolsets. The wARriorDrone API provides the core functionality to access the pixel data of the AR Drone's video stream, and provides an interface for mapping specified colors to common drone flight commands.

1. INTRODUCTION

The AR.Drone 2.0 is a remote-controlled quadcopter that is operated by computer instruction via Wireless Ad Hoc Network. The drone comes equipped with two video recording cameras that can be read and processed by an application server. Because of the ability to control the drone over a wireless network, it has a great sum of applications, surveillance, natural disaster monitoring and relief, even package delivery with more advanced drone hardware. [1] Programming the AR.Drone provides learning opportunities in a variety of new programming languages and technologies, WebGL, Computer Vision (OpenCV), UDP Networking, Accelerometers, Gyroscopes, and experience with reverse engineering the drones base SDK. [2] Its very important to understand how drones function and are programed, they are already a huge part of military defense systems around the world, and are slowly becoming more widespread in private industry as first of their kind regulations are being defined by the Federal Aviation Administration.

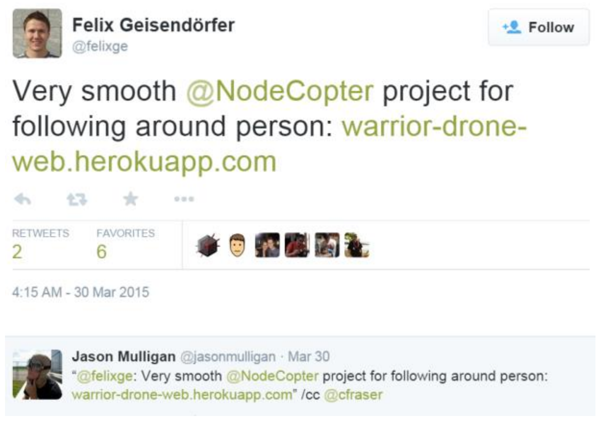
There are several SDK's/API's that are available for a variety of programing languages, these provide an interface to send and receive data to and from the AR Drone 2.0. The API's that support autonomous flight and image processing, almost all use OpenCV. OpenCV or Open Source Computer Vision, is a very popular open source library that supports real time computer vision applications. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, and more [5-7]. Two of the more interesting uses of OpenCV I came across were node-copterface and ardrone-panorama. Node-copterface uses OpenCV to detect faces and attempt to keep the detected face in the center of drones view. Panorama allows for the creation of 360 degree panoramic photos using OpenCV photo stitching libraries. Both showcase the power of processing the AR Drone video stream with OpenCV. But, between installing and configuring the API and independent OpenCV library you are looking at a lengthy set up process even for someone with system administration and computer graphics experience. [2-3, 5]

The DronePilot.NET SDK supports building AR Drone applications using C# and Visual Basic, the API provides a great coverage of all the available commands available to the Drone, and supports advanced image processing through OpenCV. It was created in Japan by researchers at Niigata University in an effort to simplify the process of creating a GUI and sending commands to the AR Drone, as compared to using the C/C++ SDK released by the AR Drone manufacturer Parrot. It successfully achieved this, but, is dependent on having OpenCV installed and set up to do any image processing. One big downfall of the DronePilot.NET SDK is that the amount of available open source code for programing the drone with C#/Visual Basic does not even compare to the number of open source packages available for Node.js [2, 5].

Figure 1. AR.Drone 2.0 Power Edition

Because the proposed API is built using Node.js, installation is quick and a verbose set of open-source libraries can be easily installed and managed via NPM (Node Package Manager). Making the API great for quickly getting a feel for programming the drone and basic image processing without having to configure complex computer vision libraries. I have measured the performance of my proposed API for processing the AR Drone's video stream and have verified its satisfactory performance, to detect and follow colored objects.

One of the biggest reasons for node's popularity with AR Drone developers is the NodeCopter community. NodeCopter is a full day event where fifteen to sixty developers team up in groups of three, and are given one Parrot AR.Drone 2.0 and spends the day programming and having fun with it. At the end of the day each team presents a demo of their work to the other attendees. NodeCopter was founded by Felix Geisendorfer and the first event was held in Berlin, Germany on October 5th, 2012. It gained support quickly, and twenty-six more NodeCopter events have occurred across Europe and the United States since. Felix is also the author of node-ar-drone a Client API for interacting with the drone, which was the foundation for coding at NodeCopter events. The node-ar-drone library is available on GitHub and provides a great API for programing the drone at both high and low levels. However, unlike my proposed API it does not have any tools to simplify image processing, and the few plugins available require OpenCV set up in your development environment [3-4].


Felix Geisendörfer
@felixge
Follow

Very smooth @NodeCopter project for following around person: warrior-drone-web.herokuapp.com

RETWEETS 2 FAVORITES 6

4:15 AM - 30 Mar 2015

Jason Mulligan @jasonmulligan · Mar 30
"@felixge: Very smooth @NodeCopter project for following around person: warrior-drone-web.herokuapp.com" /cc @cfraser

Felix Geisendorfer founder of NodeCopter, giving the wARriorDrone API a shout out on twitter

2. HYPOTHESIS

An AR Drone 2.0 quadcopter can be programmed to recognize and autonomously follow a colored object in a controlled environment.

3. METHODS

I expressed my interest and submitted a proposal to the Computer Science Department requesting funding to purchase an AR.Drone 2.0. After successfully gaining the Computer Science Department's support an AR Drone 2.0 Power Edition was ordered online through the WSU CS Department, which came with 2 HD Batteries, 4 sets of propellers, as well as tools for any needed repairs. At the end of the semester I will be returning the drone to the Computer Science department making it available for future student projects.

3.1 Specifications

Dependencies:

Node.js: v0.12.0

node-ar-drone: v0.3.3
node-dronestream: v1.1.1

Angular.js: v1.3.15

Drone Specs:

Model: AR.Drone 2.0 Power Edition

Network: Wi-Fi (802.11)

Processor: 1 Ghz CPU

Memory: 125MB

OS: Linux (BusyBox)

Cameras: 720p front, 480p bottom

Machine Specs:

OS: Windows 7 Enterprise

Processor: Intel(R) Core i7-4820k CPU @ 3.7GHz

RAM: 8 GB

System: 64 bit

IDE: JetBrains: WebStorm (JavaScript IDE)

AR.Drone Features Diagram [10]

3.2 Software Design and Implementation

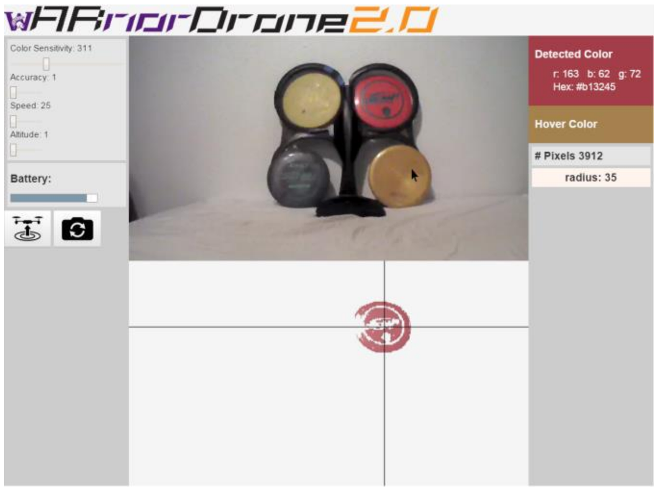
The wARriorDrone API extends two open source node libraries, node-ar-drone, and node-dronestream [4].The node-ar-drone library developed by Felix Geisendorfer, provides a client API for sending movement commands and reading the drone video stream and navigation data. The node-dronestream handles rendering the drone's video stream in a WebGL canvas using broadway.js. Great open source libraries like these are the primary reason I chose Node over languages like Java or C#. On the other hand Node.js is not built for CPU-intensive operations; in fact, using it for heavy computation will annul nearly all of its advantages. However, Node excels in building fast, scalable applications, and has the capability to handle a large number of simultaneous connections with high throughput, which equates to high scalability. As a result Node is very efficient at maintaining and reading video and data streams, making it great for reading the AR.Drone's two video streams and the navigation data all asynchronously.

To avoid configuring any external libraries such as OpenCV all image processing and autonomous flight algorithms are written in

JavaScript within the API. This makes the API and example application ready to use as soon as it is installed with no need for additional setup. Making the application great for someone who has little experience in computer graphics but also is a great platform to start from to gain experience with the core concepts of computer vision and autonomous flight in the simplest and quickest way possible. In addition to my personal preference and familiarity, I use Angular.js primarily to take advantage of the built in two way data-binding for keeping the various control variables in the front and backend in sync without any extra code.

3.2.1 Demo Application

I have developed a web application using my proposed API and Angular.js to demonstrate its proficiency as well as visualize the data being processed by the server to aid in understanding the underlying algorithms. The application allows the user to track a colored object by clicking the video stream to define the desired color range. An overview of the applications functionality can be seen below in Figure 2.



Color Sensitivity: 311
Accuracy: 1
Speed: 25
Altitude: 1
Battery:

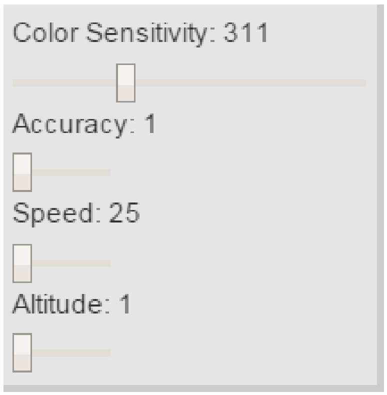
Detected Color
r: 163 b: 62 g: 72
Hex: #b13245
Hover Color
Pixels 3912
radius: 35

Figure 2. wARriorDrone Web App

In the center of the application interface, the AR.Drone's video stream is rendered in the user's browser of choice using broadway.js and a WebGL canvas. The HTML5 canvas below renders any pixels that match the detected color range and estimates the center of the object by calculating the average x/y coordinate of all detected pixels. The visualization provided by the bottom canvas was crucial to debugging the color detection algorithms early on, and provides instant feedback when adjusting color detection settings ensuring you are detecting the most possible pixels on the object without detecting undesired pixels outside the object.

The application provides the user with controls to optimize color detection and several options to modify how the drone will interact with the detected color. Initially I planned to allow the user to choose between a few pre-defined colors (red, blue, yellow, etc.) that the drone could detect. This approach had two major flaws. First it is difficult to find an RGB color range that will consistently detect the object without picking up other unwanted pixels. Second the color of an object will change midflight from something as simple as a shadow or change in lightning, causing the drone to lose the object regularly. To solve this I have implemented three color detection settings (Figure 3) that may be adjusted at any time while using the application. The Color Sensitivity slider allows the user to adjust how closely a pixels color must match the detected color. This is extremely useful allowing you to hone down to very specific color range, or be extend the color range to detect more of the object when no similar colors are in the environment. The Accuracy slider is used to determine how many pixels from the drone's stream are processed (1 = every pixel, 2 = every other pixel, etc.). Allowing a user to increase accuracy when using a computer with enough processing power, or reduce accuracy to free up CPU on a slower machine. The Speed slider Controls how often in milliseconds the program computes the detected pixels and sends commands to the drone. Setting the speed slider to the fastest speed twenty-five milliseconds will cause the drone to react faster to movement of the object, but even more importantly adjusts the color range more often decreasing the chance of losing the detected object. A slow speed like five-hundred (two times per second) will not be as effective when tracking the drone, but allows for each tick of the dynamic color detection algorithm can be seen frame by frame, an extremely useful feature when debugging or improving the color detection algorithms.

My goal was to enable any AR.Drone owner, to be able to experiment with computer vision and autonomous flight quickly and without a lengthy and complex configuration process. The application meets both these goals, assuming Node is already configured it can be installed with NPM (Node Package Manager) that comes bundled with Node.js on all major OS (Windows, Linux, IOS) with one command.



Color Sensitivity: 311
Accuracy: 1
Speed: 25
Altitude: 1

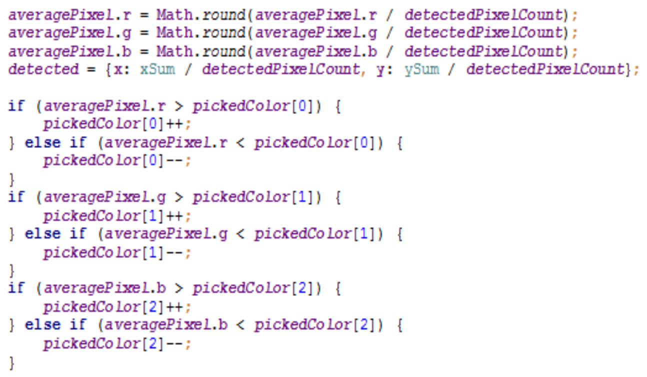
Figure 3. Color Detection Settings

3.2.2 Color Detection

As I have mentioned previously, choosing to manually code the color detection algorithms in JavaScript eliminates the dependency on OpenCV or another computer vision library. Details on the functionality of the color detection algorithm can be seen in Figure 2. As the drone is tracking an object around a room the lighting on the object and available to the drone change frequently, this also changes the shade of the object being detected which can cause total loss or inconsistent detection of the object. To solve this, the algorithm automatically adjusts the color range the drone is detecting by moving the RGB values slightly towards the average

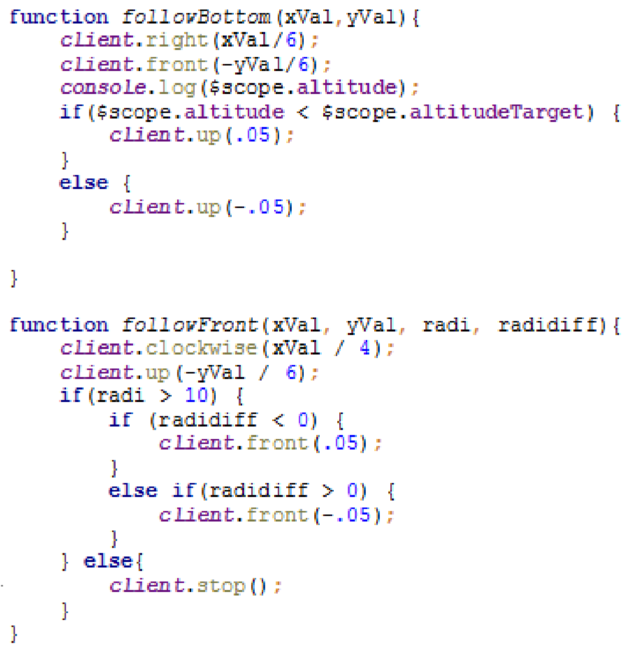
```
funct
    c
    c
    c
    i

    }
    e

    }
}

funct
    c
    c
    i





    }

    }
}
```

of all detected pixels each frame. This provides a level of consistency and flexibility that far surpasses attempting to track a set color range that does not adjust to a changing environment.



```
averagePixel.r = Math.round(averagePixel.r / detectedPixelCount);
averagePixel.g = Math.round(averagePixel.g / detectedPixelCount);
averagePixel.b = Math.round(averagePixel.b / detectedPixelCount);
detected = {x: xSum / detectedPixelCount, y: ySum / detectedPixelCount};

if (averagePixel.r > pickedColor[0]) {
    pickedColor[0]++;
} else if (averagePixel.r < pickedColor[0]) {
    pickedColor[0]--;
}
if (averagePixel.g > pickedColor[1]) {
    pickedColor[1]++;
} else if (averagePixel.g < pickedColor[1]) {
    pickedColor[1]--;
}
if (averagePixel.b > pickedColor[2]) {
    pickedColor[2]++;
} else if (averagePixel.b < pickedColor[2]) {
    pickedColor[2]--;
}
```

Figure 4. Dynamic Color Detection Algorithm



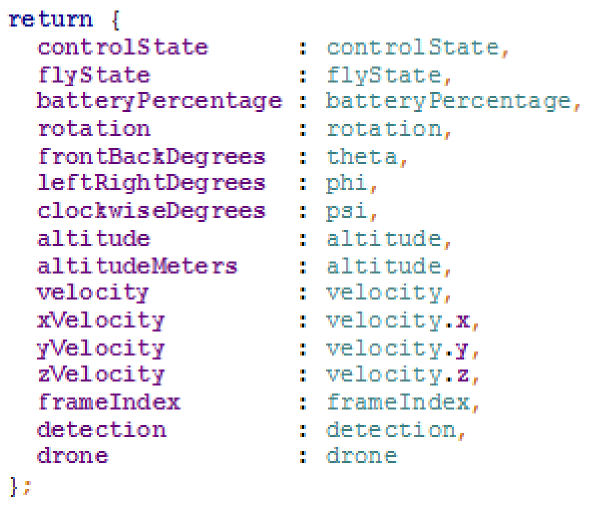
```
function followBottom(xVal,yVal){
    client.right(xVal/6);
    client.front(-yVal/6);
    console.log($scope.altitude);
    if($scope.altitude < $scope.altitudeTarget) {
        client.up(.05);
    }
    else {
        client.up(-.05);
    }

}

function followFront(xVal, yVal, radi, radidiff){
    client.clockwise(xVal / 4);
    client.up(-yVal / 6);
    if(radi > 10) {
        if (radidiff < 0) {
            client.front(.05);
        }
        else if(radidiff > 0) {
            client.front(-.05);
        }
    } else{
        client.stop();
    }
}
```

Figure 5. Autonomous Flight Algorithms

The section of code that adjusts the color range is included in the Appendix Figure A. The averagePixel is a temporary object that at this point is populated with the average of each of the r, b and g values of all currently detected pixels (count = # of detected pixels). The RGB values of the computed averagePixel and the current color being tracked (pickedColor) are compared one at a time, and the RGB range is modified to be slightly closer to the average color that was computed. By only incrementing or decrementing the RGB values by 1 each iteration the adjustment is smooth and will not overreact to quick temporary changes in the detected color. This combined with tweaking the Color Sensitivity settings seen in Figure $, allows for optimized detection no matter the environment. The object rendering on the white canvas gives you immediate feedback as you adjust the sliders, allowing you to easily see how effectively the object's color is being detected.

3.2.3 Autonomous Flight

To achieve autonomous flight I used only pixel data from the video stream and navigation data gathered by the drone's sensors. Basic drone movement commands take in a speed argument between one and negative one. The speed arguments are based on the distance as a percentage between the detected objects center, and the center of the drone's field of view (Figure 5: xVal, yVal). I divide by /6/determine how fast the drone should move in the required direction determining the speed and direction the drone will move. The demo application supports three tracking modes. When the default mode, follow-front is active, the drone will track the detected object using the front camera, while maintaining a consistent distance and keeping the object in the center of its view.



```
return {
  controlState       : controlState,
  flyState           : flyState,
  batteryPercentage  : batteryPercentage,
  rotation           : rotation,
  frontBackDegrees   : theta,
  leftRightDegrees   : phi,
  clockwiseDegrees   : psi,
  altitude           : altitude,
  altitudeMeters     : altitude,
  velocity           : velocity,
  xVelocity          : velocity.x,
  yVelocity          : velocity.y,
  zVelocity          : velocity.z,
  frameIndex         : frameIndex,
  detection          : detection,
  drone              : drone
};
```

Figure 6. Example AR.Drone 2.0 Navigation Data [4]

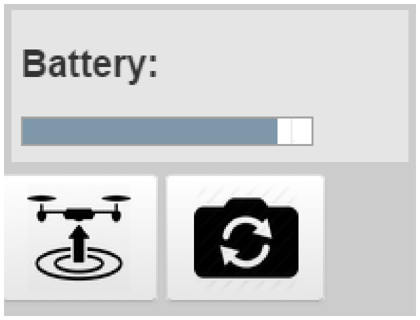


Figure 7. Battery Meter and Flight Commands

The second mode follow-bottom, is activated by pressing the "switch camera" button shown in Figure 7. This mode utilizes the drones down facing 480p camera to track an object from above. The drone will hover at an adjustable altitude (Figure 3) above the detected object. An on board Altimeter that measures atmospheric pressure is used to determine the drone's current altitude. The greater the altitude the lower the pressure *[2]*. This can be retrieved through the drone's real-time navigation data, as well as the fields shown in Figure 6. The battery meter in Figure 7 created using an

HTML5 progress bar that is updated automatically by parsing the navigation data (Figure 6: batteryPercentage). And third, “orbit” mode, commands the drone to circle the detected object while keeping the camera centered on the object. Movement along the x/y axis (up, down, left, and right) is based on the calculated center of the object, calculated by taking the average location of all detected pixels.

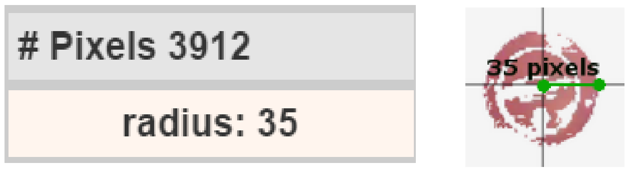


Figure 8. Distance Metrics

Keeping a set distance from the detected object is not quite as straight forward. I use two metrics to determine distance, first I get a general size of the object based total pixels detected, more pixels than previously detected indicates the object is closer, the inverse is true for less. Unfortunately the total pixels detected alone is not consistent enough to maintain distance, so I introduce a second metric, a rough width, or “radius” of the detected object. Illustrated by Figure 8, I obtain a rough estimate of the radius, starting at the detected center of the object I read the row of pixels to the right one at a time. The distance between the location of the last pixel to match the detected color and the center of the object is the radius. While the drone is tracking an object the originally detected radius is compared to the current radius (Figure 5: radidiff) based on the live video, moving forward or backwards to remain at the correct distance.

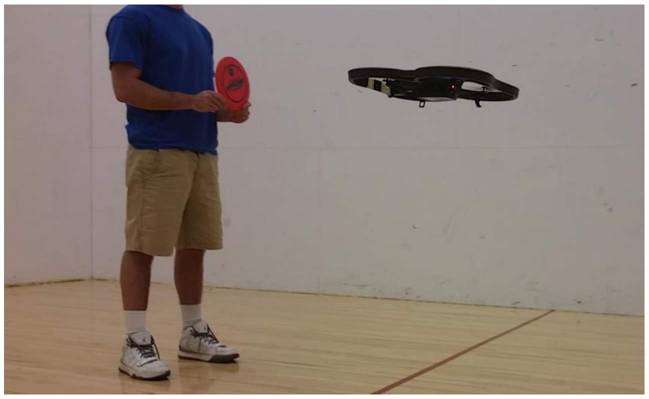


Figure 9. Autonomous Flight Trial

4. TESTING PHASES

4.1 Color Detection Algorithm Optimization

To avoid damage to the AR Drone and my early testing environment, most of the color detection development and testing was completed without the need to fly the drone. For my own convenience while testing different color sensitivities and how often I process the current video buffer, (every 25 ms-500ms), I added the slider controls that became a major feature of the finished application. For test objects to detect I used mostly discs I use when playing disc golf, for their variety of colors and consistent size (Figure 2).

4.2 wARriorDrone Trials

Four autonomous flight sessions were conducted. The first two tests focused on tracking with the front camera (Figure 9), and switching between colors while flying using the three colored squares in Figure 10. At first the drone over compensated the distance required to center the object. To correct this I lowered the speed of the movement commands sent to the drone. Another common issue I experienced was inverting or forgetting to invert the movement commands which would send the drone the opposite direction of the detected object. The last two rounds tested improvements to the front tracking algorithm as well as showcased the bottom tracking functionality and orbit mode for the first time. I use Camestia Studio to capture the Application view during tests, and Google Chromes built in profiling tools to evaluate how much CPU I had to spare for additional processing or features [10].

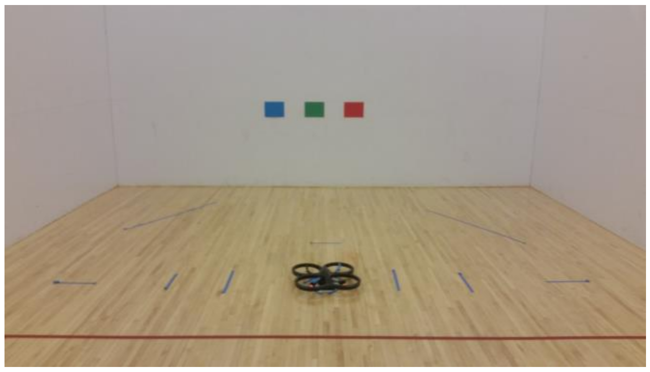


Figure 10. Racquetball Test Environment

Tests were conducted in a standard racquetball court that measures 40 feet long, 20 feet wide, and 20 feet high with red lines defining the service and serve reception areas. This provides multiple advantages, first it is an open but enclosed space, to have room to fly but if the program has errors the drone can’t damage itself or others. Second it provides a consistent white background in all directions, perfect for detecting colored objects against. Last the striped lines in the service area give the drone a good pattern on the ground to stabilize against.

5. CONCLUSION

I have successfully implemented a demo application using the wARriorDrone API, which supports autonomous flight through color tracking. To aid potential users I have also created a website with documentation on how to get started working with the drone, and videos of the three main trials. The application is open source and is published to NPM, making installation and configuration, faster and easier than any existing AR.Drone API. As continued research I would like to implement more detection modes as well as implement PID controllers, a common data structure used in autonomous flight systems to improve tracking accuracy and consistency. Programming the AR.Drone 2.0 is a great way to start experimenting with cutting edge Computer Science and Engineering. On top of that, the hands on nature of the programming makes it a great way to get younger generations excited about programming and robotics. As new drone technologies continue to improve drones will quickly integrate themselves into our daily lives. Making drone and computer vision research crucial to taking advantage of new opportunities they provide, and equally if not more importantly understand their flaws, for situations when they will inevitably be used for the wrong reasons. My hope is that Winona State Computer Science students will continue to develop applications using the drone and continue to update the website with their new projects.

6. ACKNOWLEDGEMENT

I would like to thank, Dr. Joan Francioni, Dr. Mingrui Zhang, and Dr. Narayan Debnath for helping with the purchase of the drone and guiding me through the research process.

7. REFERENCES

[1] Randal W. Beard, Timothy W. McLain, Small Unmanned Aircraft: Theory and Practice, Princeton Oxford Press, C 2012

[2] Takuya Saito, Kenichi Mase, DronePilot.NET Development: AR.Drone SDK Supporting Native and Managed Code, IEEE Journal, Niigata University, Japan, 2013

[3] Felix Geisendorfer, What Is This?, http://www.nodecopter.com/, 2014, 2/13/2015

[4] Felix Geisendorfer, Documentation, https://github.com/felixge/node-ar-drone, 2014,

1/14/2015

[5] OpenCV Developers Team, About, http://opencv.org/about.html, 2015, 2/10/2015

[6] Laurent Eschenauer, Facial Recognition, https://github.com/eschnou/webflight-copterface , 2013, 2/7/2014

[7] Laurent Eschenauer, Panorama, https://github.com/eschnou/ardrone-panorama , 2013, 2/8/2014

[8] Google Angular Team, Angular.js, https://angularjs.org/, 2015, 1/1/2015

[9] Parrot, AR.Drone 2.0 Diagram, http://ardrone2.parrot.com/ardrone-2/specifications/, 2014

[10] TechSmith, Camestia Studio, https://www.techsmith.com/camtasia.html, 2015, 2/13/2015

8. APPENDIX

Source Code: https://github.com/JBrookshaw/node-wARriorDrone
Videos/Documentation: https://warrior-drone-web.herokuapp.com/#!/
WebStorm: https://www.jetbrains.com/webstorm/