GE: Blisk Blade Root Fillet Quality Check

A Major Qualifying Project Report to the Faculty of the Worcester Polytechnic Institute, in partial fulfillment of the Bachelor of Science degree in cooperation with GE Aviation in Hookset, New Hampshire

Submitted by:

Matt Shanck

William Moore

Project Advisors:

Professor Susan Jarvis

Professor Craig Putnam

Professor Kenneth Stafford

Sponsor:

John Graham

Submitted on: September 1, 2015

This report represents the work of WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on its website without editorial or peer review. For more information about the projects program at WPI, please see http://www.wpi.edu/academics/ugradstudies/project-learning.html

Abstract:

GE Aviation, in New Hampshire, produces the bladed disks (blisks) that are necessary in the construction of jet engines. These blisks are manufactured from solid billets of titanium that are machined into the correct configuration. The blisks undergo various quality checks, including one to determine if the minimum and maximum radii of the blade root fillets are met. The goal of the project was to develop a robotic system which automates that portion of the Blisk inspection process.

Table of Contents

Abstract: 2
Chapter 1: Introduction 6
Chapter 2: Background 7
Chapter 3: Methodology 10
  Turntable: 11
  End of Arm Tooling (EOAT): 21
  Sensor and Motor System: 25
  Image Processing: 42
Results: 49
  Turntable: 50
  End of Arm Tooling: 54
  Sensor and Motor System: 55
  Image Processing: 56
Recommendations and Conclusion: 61
References: 65
Appendix: 66
  RAPID Code: 66
  MATLAB Code: 70
  Arduino Code: 78

Table of Figures

Figure 1: Blisk Model 6
Figure 2: Blade Root Fillet Light Inspection 8
Figure 3: ABB IRB 1600 9
Figure 4: Ring-Bearing turntable with Indexing Plate 13
Figure 5: Chain around Sprocket 14
Figure 6: Turntable Side View Model 15
Figure 7: Indexing Protrusions on Spacer Plate and Dowel Base 16
Figure 8: Turntable Sub-Assembly 17
Figure 9: Machined Turntable Parts Alligned for Assembly with Chain Tensioner 18
Figure 10: Chuck Jaws Spread 19
Figure 11: Full Turntable Model with Blisk 21
Figure 12: Initial Tool Design 22
Figure 13: First 3-D Printed Design 23
Figure 14: Current Design of End Effector 24
Figure 15: System Model in Robot Studio 26
Figure 16: EOAT Approaching Blisk in RobotStudio 26
Figure 17: Model of EOAT without Ball Bearing (older Robot Studio Code) 28
Figure 18: EOAT in Blisk Model 28
Figure 19: ABB and Arduino Wiring Diagram 30
Figure 20: OP-amp and Voltage-divider Circuit 30
Figure 21: Motor Driver Wiring Diagram (8) 31
Figure 22: Motor Wiring (9) 32
Figure 23: Arduino with Stepper Motor 33
Figure 24: Step Signal Diagram 35
Figure 25: Signal for No Motor Movement 35
Figure 26: Blisk Coverage Gaps 36
Figure 27: Full Signal System Diagram 37
Figure 28: Flow Chart 38
Figure 29: Half-Bridge Diagram 40
Figure 30: MATLAB Start Screen 43
Figure 31: Coding System Diagram 44
Figure 32: Sample of Maximum Ball Image 46
Figure 33: Sample of Maximum Cropped Image 46
Figure 34: Sample of Maximum Greyscale Image 47
Figure 35:Sample of Maximum Binary Image 47
Figure 36 Smile and Whiskers 48
Figure 37: Results Display Screen 49
Figure 38: Machined Turntable Assembly 49
Figure 39: Chuck Jaws 51

Figure 40: Unfinished Sprocket 52
Figure 41: Turntable with Blisk 53
Figure 42: Side View of Turntable Assembly (shown w/o chain) 54
Figure 43: Completed EOAT 55
Figure 44: Maximum Test Concave Sample Image A 59
Figure 45: Maximum Test Convex Sample Image B 59
Figure 46: Minimum Test Concave Sample Image C 60
Figure 47: Minimum Test Convex Sample Image D 60

Chapter 1: Introduction

General Electric is a multinational, technology engineering company that supplies a variety of products to consumers and private organizations. In particular, the GE Aviation division is a leading producer of special parts and engines used in the aviation world. One such facility in Hookset, New Hampshire, produces the bladed disks (blisks) that are necessary in the construction of jet engines (1).

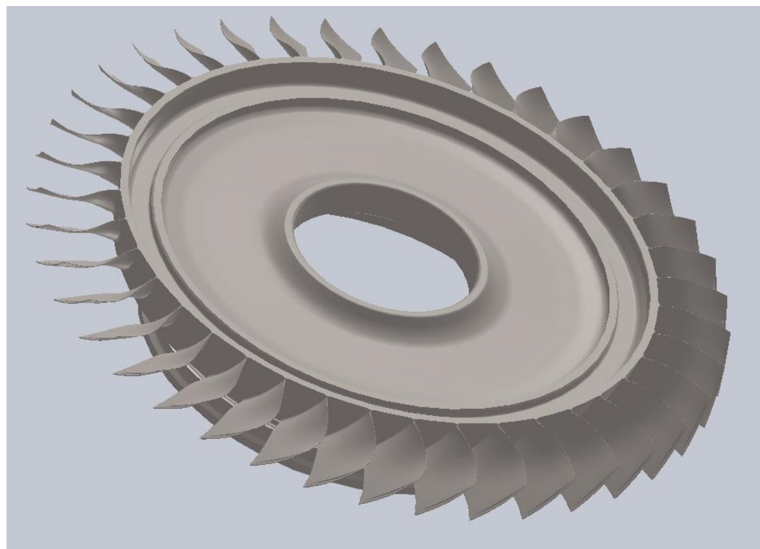


Figure 1: Blisk Model

A blisk is a part that consists of multiple blades mounted on a central disk for rotation (Figure 1). Traditionally, blisks are assemblies of blades around a center ring. However, there are some blisks that are milled out of a single billet of titanium. The billets are milled over the course of up to two weeks depending on the size. Once the finished blisks exit the milling machine, they are moved to another area to undergo various quality assurance checks. One such examination involves using a hand tool to verify that the blade root fillets meet specifications (2).

GE currently uses a manual inspection process to examine these blisks. The primary inspection procedure is the use of a developer spray and a ball-gage to determine if the minimum

and maximum radii of the root fillet are met. This is performed on both sides of each blade. If this check provides a negative result, then a light-based system can be used to confirm or deny the earlier test. The main procedure is quite long with each blade taking approximately two minutes to complete. With thirty or more blades in a blisk, using this procedure can take over an hour to complete. However, with GE's approval, we have chosen to automate the light-based inspection process because of its simplicity and speed.

The main deliverables of the project are to develop a robotic system which automates the blisk inspection process. This system worked on a single, 34-bladed blisk which served as a case study for extension to other blisks. The system needed to determine which blisk root fillets are out of specification thus requiring further examination. The system also had to be accurate in determining passing or failing blisks; the system should be 99.99% accurate; but we aimed for 99%. The system tried to perform the inspection at least as fast as the current manual system, which takes approximately one hour.

Chapter 2: Background

There are a number of sub-systems that comprise this project. One of the major aspects of this section is a thorough examination of the inspection process. The project needed to closely replicate the current approved process using multiple mechanical systems. An integral system used in the quality check is an ABB IRB 1600 industrial robot arm. The robot needed to examine the blade root fillets using a machine vision system.

This project studied GE Aviation's two-step quality check to examine the radius of the blade-root fillet. The first of the two processes is to utilize a developer spray around the blade root area. The spray is a powdery substance that coats the blade root with a thin layer. Using two

different sized ball-gages, the powder is rubbed off leaving either one or two lines. The smaller of the two ball bearings, the minimum measurement, should produce a single line in the spray. The larger of the two ball bearings, the maximum, should produce two lines in the spray. If either of these tests fails then the next stage of the blisk inspection is performed (3).

The second part of the inspection process, if the first fails, is to make use of a light-based inspection process. This works by placing the two ball bearings against the connection point and shining a light behind the bearing. For the minimum ball bearing the light should reveal no light between the ball bearing and edge of the blisk. For the maximum ball bearing, there should be a crescent of light surrounded by two connection points between the ball bearing and the blisk edge (Figure 2). This is the secondary check for the blisk (3). This entire process takes a varying amount of time based on the number of airfoils (blades) on the blisk. On average, it takes two minutes to check each airfoil using that process.

Figure 2: Blade Root Fillet Light Inspection

The full inspection process is looking for a number of different defects on the airfoils, but this project is only focused on the blade-root max-min defect. A defect of this type can occur with both milled blisks and electrochemically-machined blisks. Defects caused by the milling process vary between each airfoil, whereas electrochemically-machined defects are uniform throughout the blisk. Unfortunately there is not enough data to give a correct assessment on the accuracy of the inspection process because it is a pass or fail system. However, the percentage of blisks that fail the existing inspection process (i.e. having at least one defect) is around 2%.

The robot that is utilized for this project is the ABB IRB 1600 (Figure 3). This is a medium sized industrial robot with payloads of up to 6 kg, 1.45m arm-length, armload of up to 30kg, and the ability to mount additional hardware on the top arm.

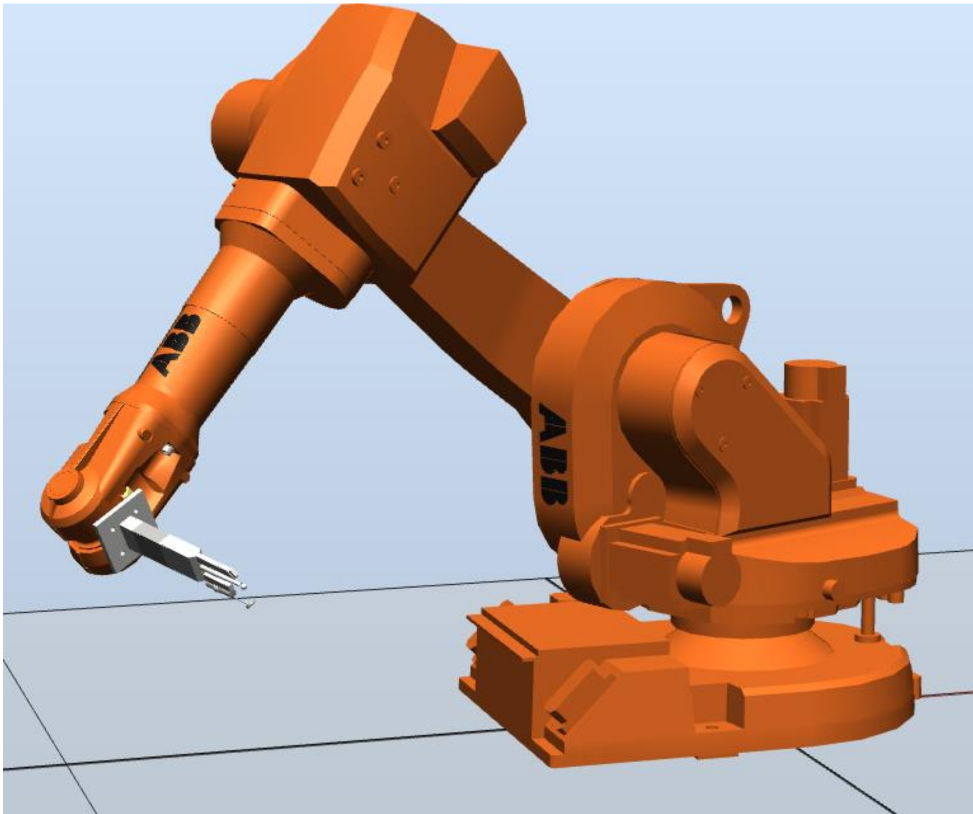


Figure 3: ABB IRB 1600

With these specifications, the robot was sufficient for our required weight loads. The robot's large working range and 6 degrees of freedom provided ample dexterity necessary to complete the task. It also served as a real world analog to other industrial robots that could use similar systems (4)(5)(6).

One of the major aspects of the project is the use of machine vision. Machine vision makes use of different images to determine the characteristics of them. These characteristics are analyzed for use within the system. One of the most basic levels of machine vision is the analysis of images in regards to pixels. Images are obtained and converted into basic pixel maps. Then the pixels are assigned numerical values that correspond to levels of light, which differentiates pixels from each other on the computer. This can be used to identify shapes in the image. This kind of machine vision is useful for light pattern recognition (7).

Chapter 3: Methodology

The current inspection process, using the light-based system, is a mundane task that requires up to two hours per blisk to complete. Once the blisk is placed on a small platform, the inspector picks up the ball-gage in one hand and a light pipe in the other. They position the ball-gage on the root fillet and shine the light from underneath, so that they can see how much light is visible. They then have to walk around the blisk to check the top part of the blade root, before flipping the blisk over to check the bottom. The estimated inspection time per blade is currently 2 minutes 20 seconds. Our project goal was to drastically reduce the effort it takes to perform a blisk root fillet inspection in the same amount of time as the current process.

A three-stage design approach was used to design the entire system. The first major design area was the construction and implementation of a turntable. This functioned as way to

hold the blisk in place while allowing access to each blade through rotation. The second major design area was the positioning of the end of arm tooling along the root fillet. This involved creating the end of arm tooling and using the ABB robot to properly position the tool. The final step was to successfully deploy the vision processing system. It used images acquired by the digital camera to determine if the root fillet passed or failed.

Turntable:

The major requirement for the turntable was to create the same positioning for each blade for the frame of references used. This allowed the robot to find each blade consistently. In order to secure the blisk on the platform, a custom chuck was designed. Initial designs involved a 4-armed chuck that was tightened by rotating it with a handle. The motor used to rotate this chuck was located underneath the platform and supported its weight. This design proved too inaccurate and prone to a centering error. Therefore, to revise the design, we changed the 4-jaw chuck into a 3-jaw chuck which is more commonly used to center circular objects. The new design involved three aluminum jaws that all had equal radii. They expand outward into the inner diameter of the blisk by tightening a screw on the top. The jaws needed to move linearly towards the blisk inner diameter, therefore, two steel dowels were inserted into the middle of each jaw, so that they can only move in a radial direction.

To rotate the blisk, the platform rested on top of a 10” metal turntable that was set into the base plate. Rotation was powered by a stepper motor that was controlled by an Arduino which received input from a sensor. The sensor was located in the middle of the base so that it did not interfere with the chain connecting the motor to the central sprocket in the turntable. The

size of the chain was chosen to be 3/8” because that was strong enough to not break while being compact. A chain tensioner was also added to prevent any slack in the chain. As for the motor, a stepper motor was a logical choice because it can precisely move small increments to rotate the blisk blade by blade. This allowed the robot to inspect all the blade root fillets without complicated robot arm positioning. The same two movement paths were used for the front and back of each blade, as opposed to having a unique set of movements for each blade. The entire turntable was machined out of aluminum.

The design of the turntable was created with simplicity and durability in mind. First and foremost, the system had to be inclusive and hold all pieces together. This task was solved by having the system built upon a solid base of aluminum. This ensured that the subassemblies will not shift positions relative to each other. The base was machined to have a raised, circular platform that is used to index the ring-style turntable properly.

The ring-style turntable was a key part in the overall success of the system. The turntable allowed the metal platform under the blisk to rotate smoothly.

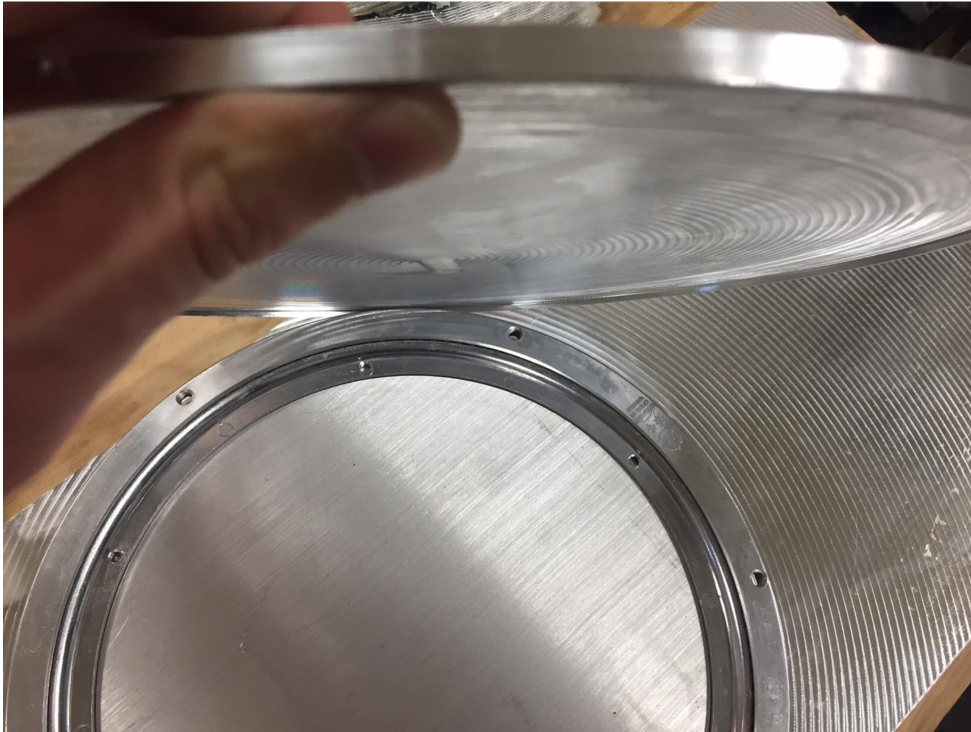


Figure 4: Ring-Bearing turntable with Indexing Plate

As seen in Figure 4, the turntable rested around the raised platform so it cannot shift laterally in any direction when rotating. The part being held in the photo is a 0.5” thick plate that was machined with an outer overhang that encompasses the outside edge of the turntable. With these two pieces screwed into the turntable, it became properly indexed and concentric.

The next layer of the system involved the sprocket for rotation and the bottom of the chuck base. The system needed to be able to rotate the blisk in a way that did not interfere with the blades. Therefore, the chain and sprocket assembly needed to be tucked away from the blisk, so that it was sandwiched between the upper turntable plate and the chuck base (see Figure 5).

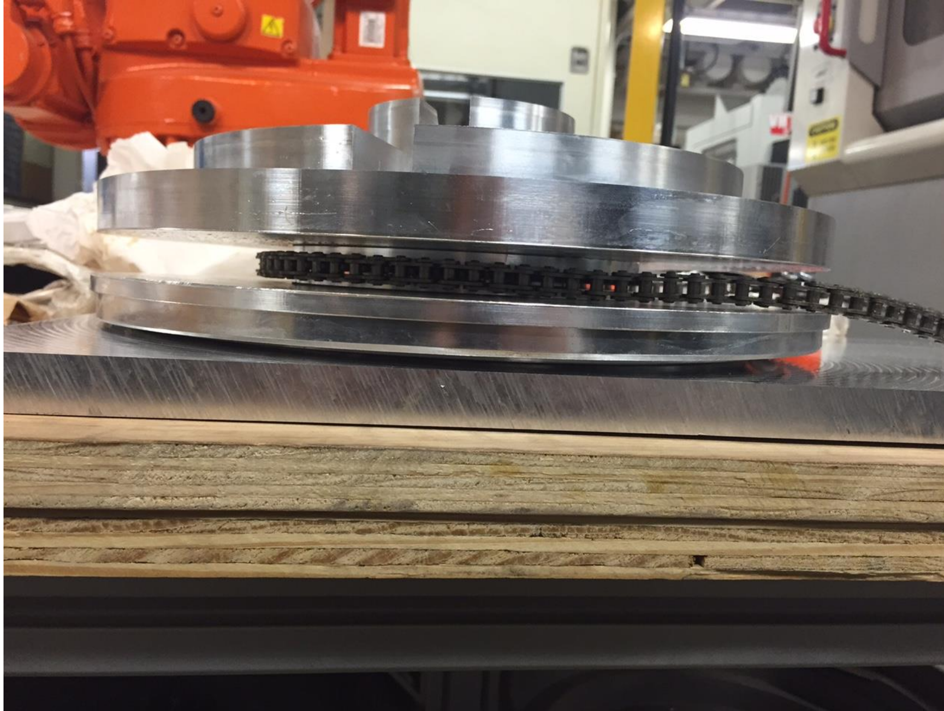


Figure 5: Chain around Sprocket

The chain would have enough clearance between the plates to not rub against them during rotation. However, during manufacturing, the bottom plate was cut too much and caused the chain to rub against it (see Results for more information). As for the sprocket, the turntable top

plate and the bottom of the chuck base were machined with indexing circles so that the sprocket would be set between them.

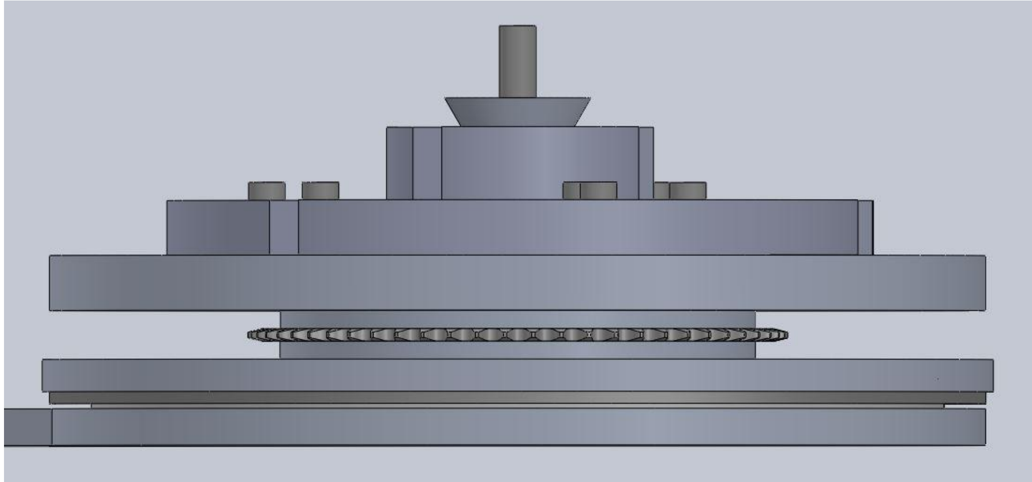


Figure 6: Turntable Side View Model

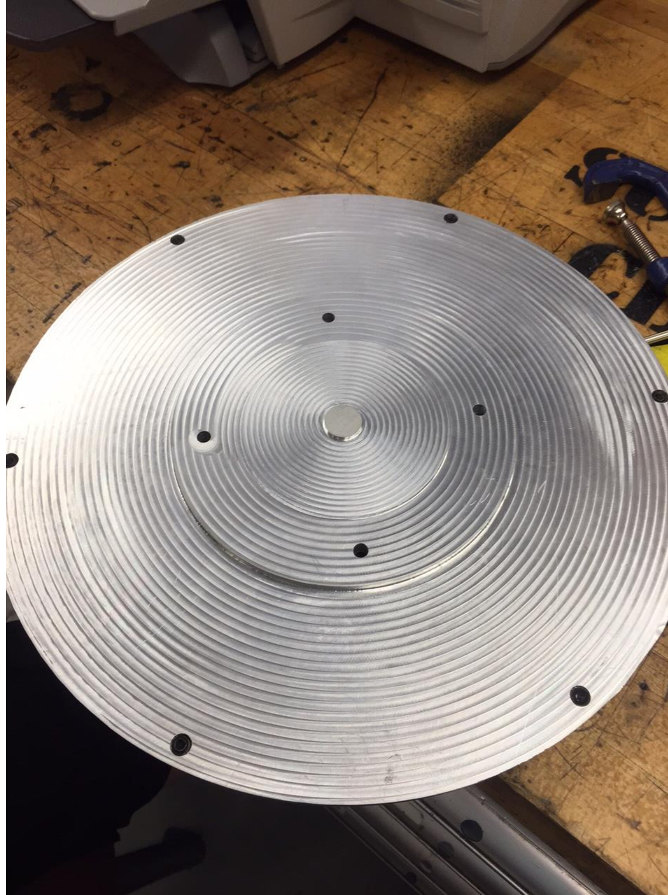


Figure 7: Indexing Protrusions on Spacer Plate and Dowel Base

Displayed in Figure 7, the turntable cover-plate and the chuck base bottom have identical protrusions that fit into the sprocket center hole. The sprocket bore had a diameter of 11/16”, so there was a close fit boss to go into the bore hole from each side of the sprocket. There were also bolt holes drilled around the perimeter for the turntable, and alignment holes in the center boss for the sub-assembly. The turntable-cover and sprocket have the four ¼” alignment holes drilled through them, while the chuck base has four holes ½” deep, and tapped so the screws can find

purchase. This completes the turntable sub-assembly as seen in the figure below.

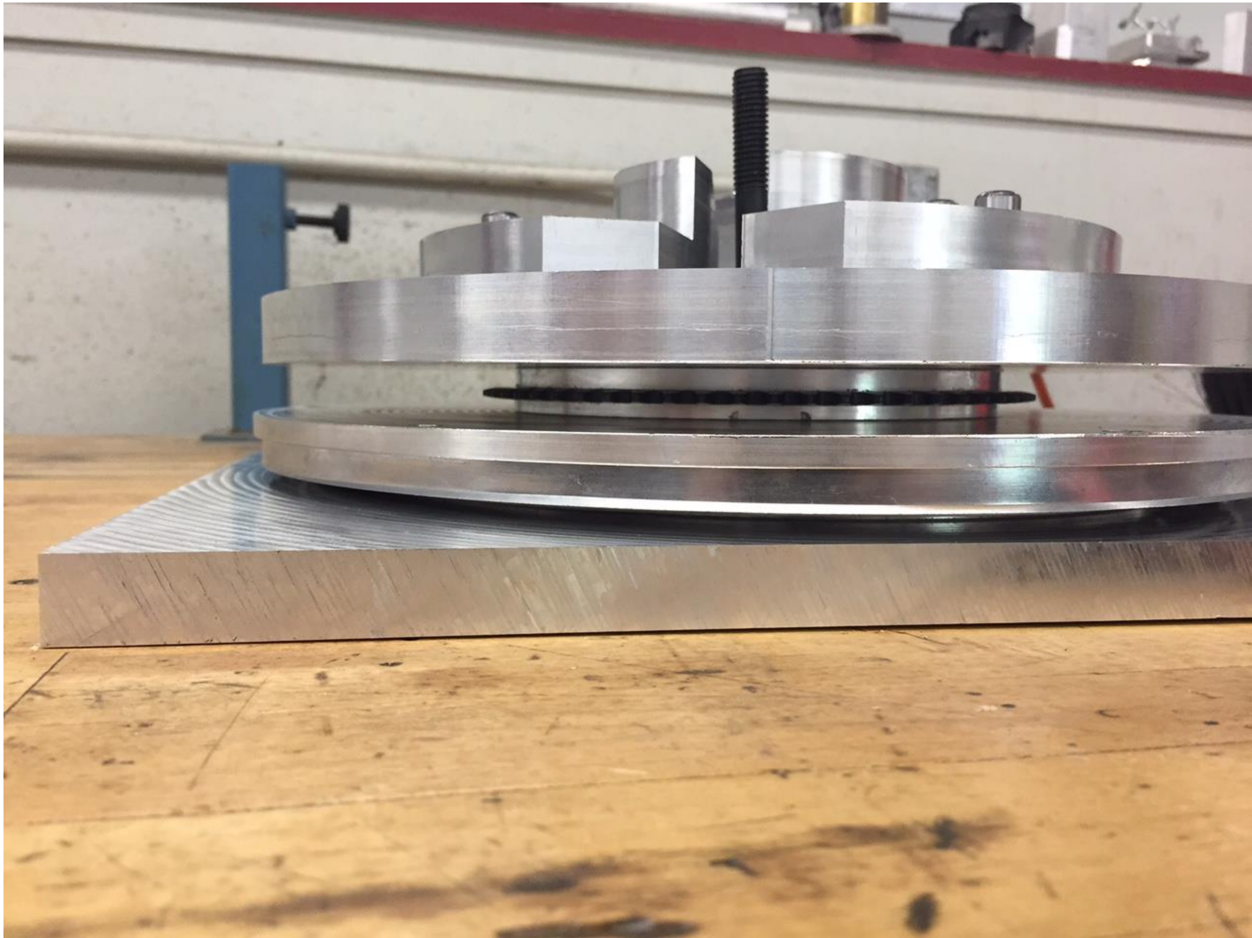


Figure 8: Turntable Sub-Assembly

The next layer of the system was the chuck base and chuck jaws which was where the blisk will have direct contact with the turntable assembly. As stated previously, the initial designs for the chuck jaws were based on a 4-jaw method, but that idea was scrapped as it did not provide enough concentricity. The new design involved a custom 3-jaw chuck that would apply pressure to the bottom lip of the blisk to hold it in place. The jaws were machined out of a 1.75” thick aluminum plate.

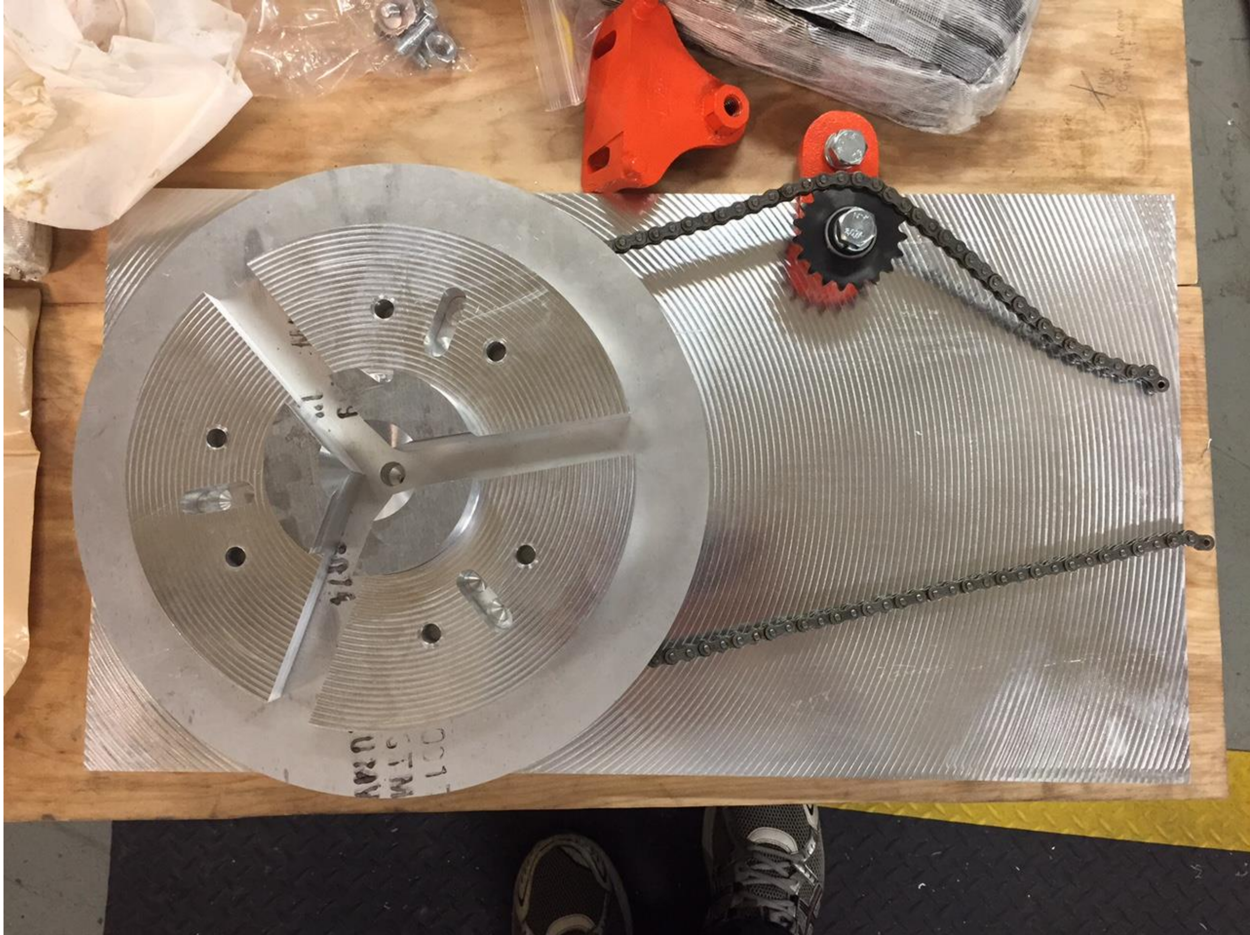


Figure 9: Machined Turntable Parts Alligned for Assembly with Chain Tensioner

The jaws were guided by ½” dowel pins that were press-fit into the chuck base. This allowed the jaws to slide in and out concentrically. To push the jaws apart evenly, a steel conical wedge was manufactured from a cylindrical stock using a lathe. The wedge has a through hole that straddles a dual-threaded 5/8” stud that is screwed into the chuck base.

Figure 10: Chuck Jaws Spread

The motor and photogate sensor had to be securely mounted to the base plate so they wouldn't move during operation. Both the motor and sensor mount were 3D printed out of a durable plastic and were secured into position with screws. Set screws were used because the head of normal machine screws interfered with the chain. The mounts were tapped so that the screw-threads gripped the part.

The sensor mount was a two part assembly: the base which was secured to the base plate, and the mount for the sensor. The photogate sensor mount needed to align the infrared beam between the blades and be able to shift away when placing and removing the blisk. The sensor was then attached to a swinging arm that rotates 90 degrees so that it can be turned to release the

blisk, and rotated back into position when a new blisk is placed on the turntable. The motor mount was a simple design that is a C-shape brace that holds the motor upside down so the drive sprocket is in line with the turntable sprocket. The motor was too tall to be oriented as seen in Figure 11, so it had to be turned upside down to account for the extra height. This design also included room for the shaft coupler that was needed to change from the motor's ¼" shaft to the ½" shaft of the drive sprocket.

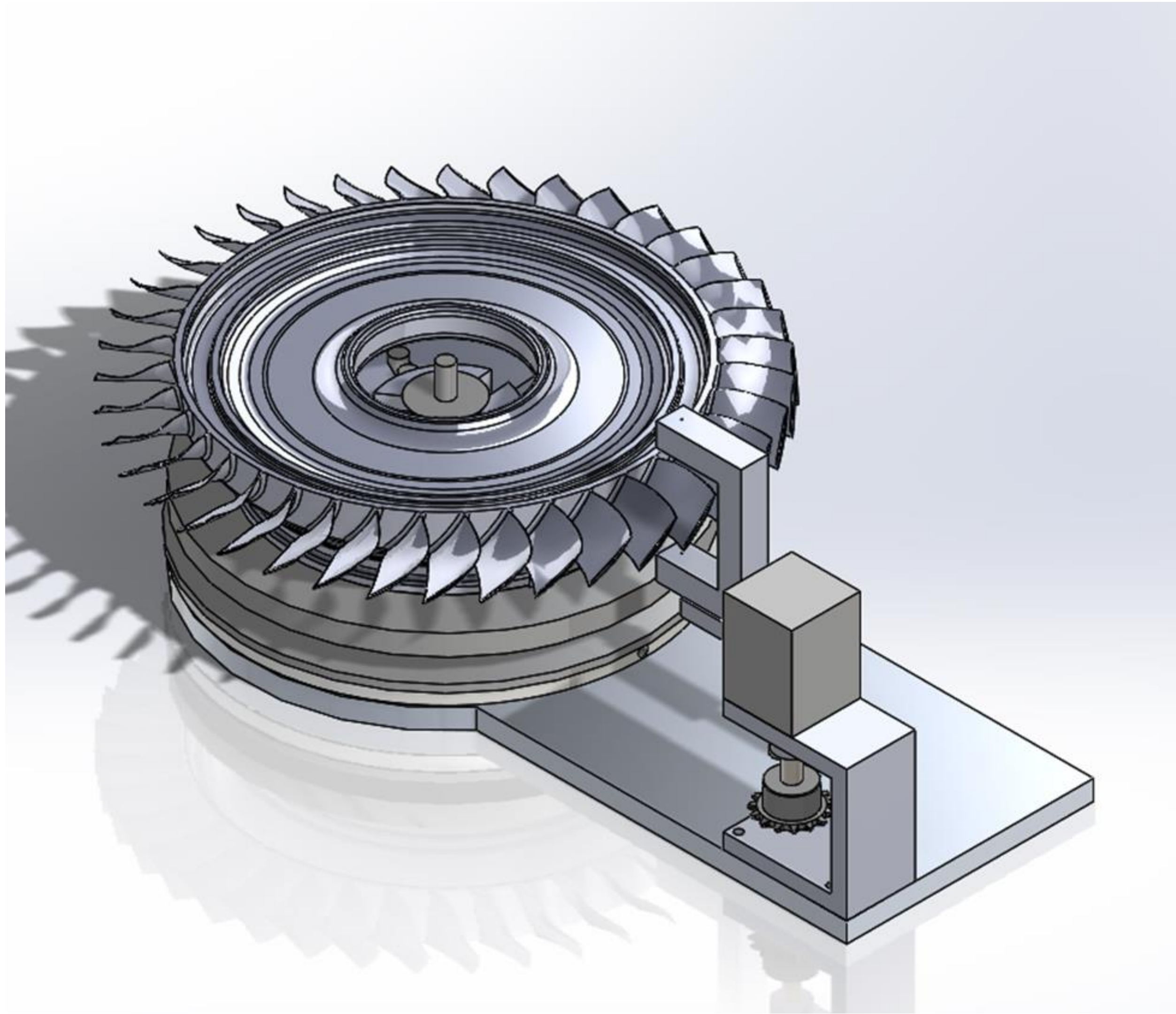


Figure 11: Full Turntable Model with Blisk

End of Arm Tooling (EOAT):

The pointer for the end of arm tooling needed to accurately trace the blade root fillet using the correct ball-bearing size. The end effector needed to have some compliance when the robot touched the blisk, as to not incur any damage, as well as apply enough pressure so that the ball gauge stayed firmly on the blade root fillet. This allowed the robot arm to not have to be 100% accurate when the tool was engaged. If the robot was off by a 1/100th of an inch, when the

bearing came in contact with the blade wall, the compliance allowed the ball-gage tip to slide into the fillet. The compliance of the elastomer allowed for this large tolerance.

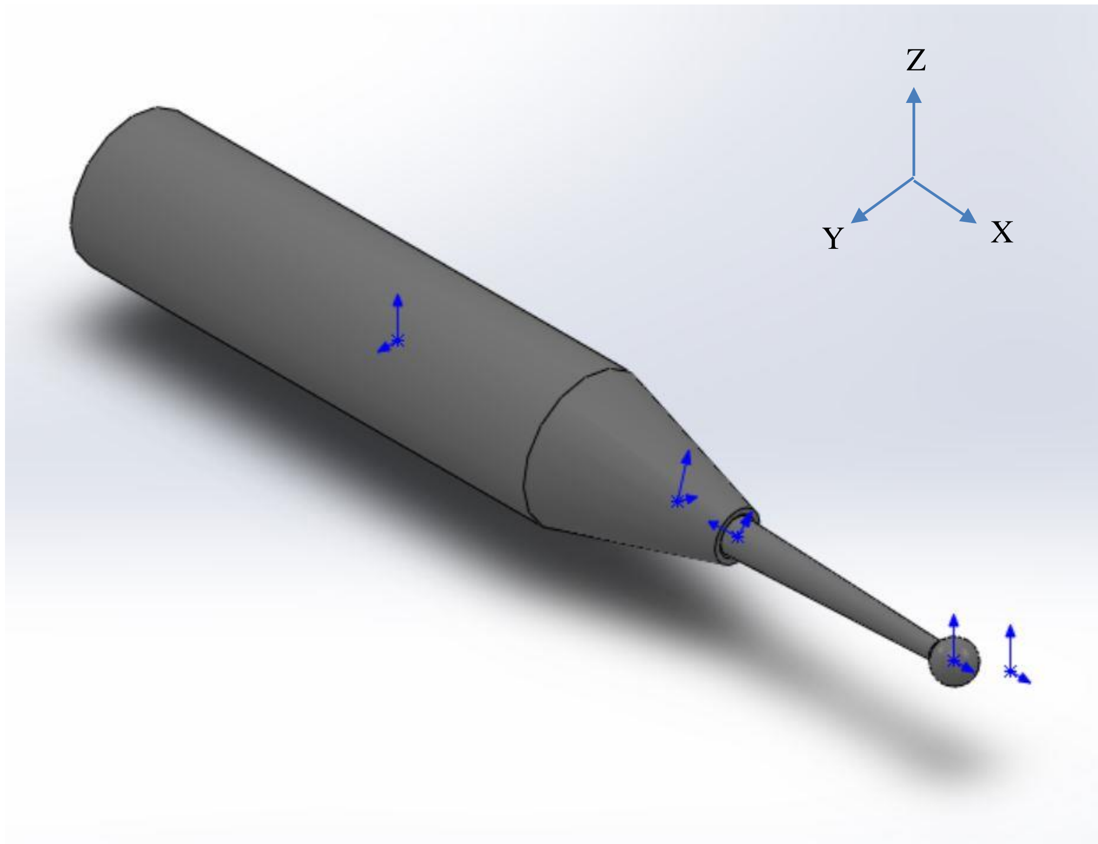


Figure 12: Initial Tool Design

At first, we thought of having a ball gage inserted into a spring-loaded cylinder. This design would give us the compliance we need in the x-axis, but it would be difficult to add compliance in the y and z-axis. Having the ball-gauge holder be a rigid structure and then have a compliant elastomer, between it and the robot, would provide the right amount of compliance with moderate error. The elastomer should allow the tool to bend into the root fillet even if it misses by ¼” at most, and also be able to compress, at most 1/8”, to not damage the tool when inspecting. The next challenge was deciding on a size and shape for the elastomer. We decided on a rectangular shape of dimensions 0.75” wide, 1.5” tall and 2” long. These dimensions make

the elastomer bend at most 2” in either direction on the Y-axis, ¾” up or down in the Z-axis, and 1/4” on the X-axis. These numbers (especially the Y-axis) were extremes for the tool and should not be reached during normal operation.

To manufacture the end effector, we decided it would be beneficial to 3D print it using a two-material design. Using a precision Objet printer, we printed the shafts of the tool with hard plastic to attach to the base and hard plastic for the ball-gage holder. The flexible material TangoBlackPlus was used as the elastomer in the middle of the two parts. The Objet printer allowed this design to be printed as a single part. Surrounding the ball gauge tip were arms that allow the endoscope camera and LED to be aimed at the ball gauge. The base serves as the attachment point for the tool and the robot base plates.

The first design is shown in the figure below, which used a box-like base, printed separately, to secure the tool to the robot arm.

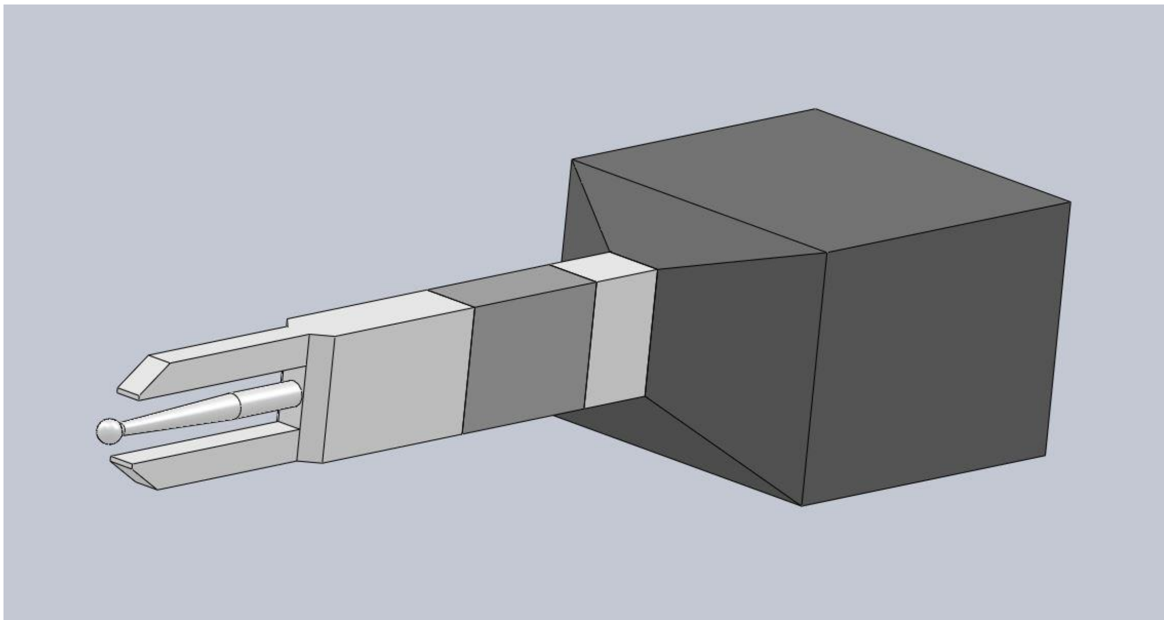


Figure 13: First 3-D Printed Design

We assembled this design and attached it to the robot arm, but found it was too large and unnecessary to be used further. The placement of the camera was initially planned to be vertical so that it would have a top-down view of the ball-gage tip. However, this made the front of the tool too large to fit in between the blades when inspecting the bottom of the root fillet. Therefore, we had to modify the design and print a new tool that solved this issue.

The next iteration for the end of arm tooling involved a re-design of the front to include a holder for the camera, as well as a slimmer arm. The idea was to lay the camera flat on the arm and to have a 0.5" mirror direct the image of the ball-gage tip at the camera. The base of the tool was also modified to be smaller and be printed as a single part.

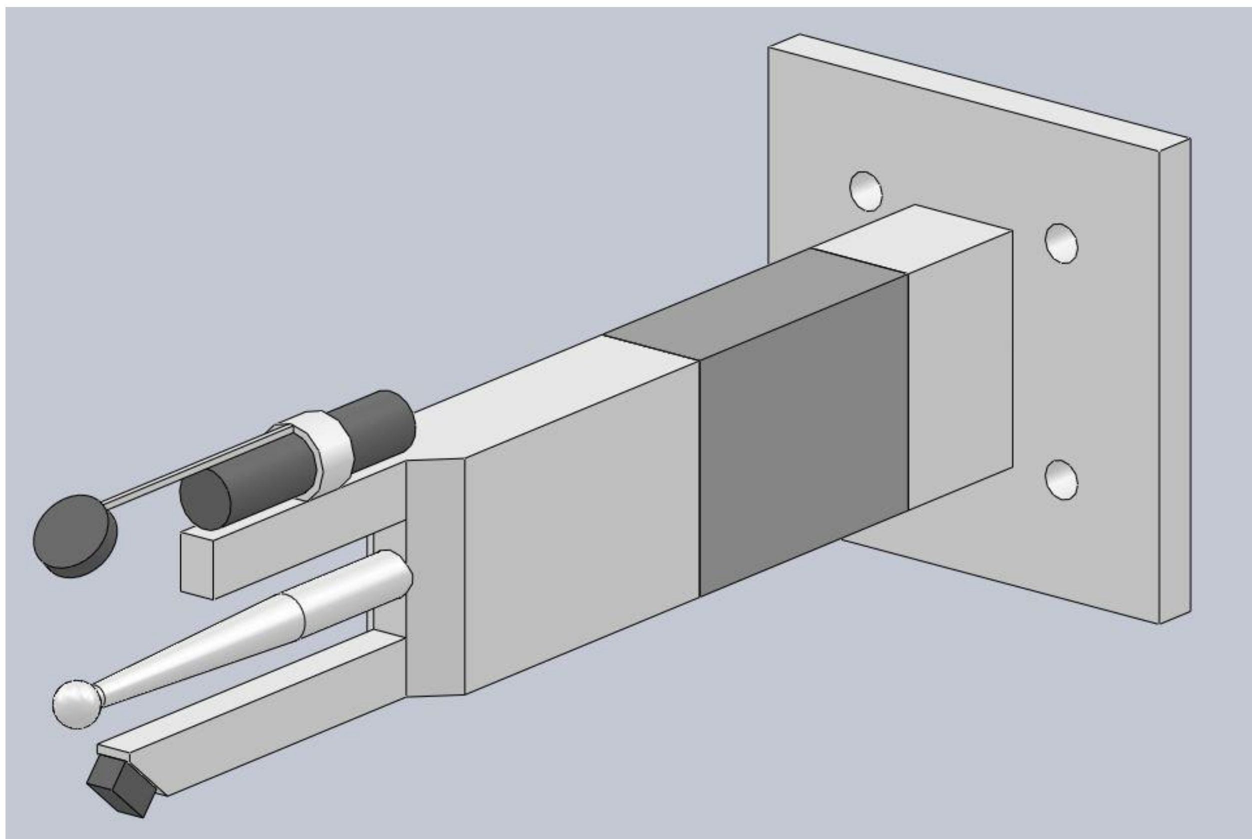


Figure 14: Current Design of End Effector

Figure 14 displays the current design of the end effector, complete with models of the camera (cylinder), mirror (puck), and LED (rectangular box). Stand-in (plastic) ball-gages were also 3D printed due to the difficulty in obtaining real ones. Although the plastic gages will wear, they still served the purpose for testing.

We used a high quality mirror to reflect the image in as much detail as possible. Although the camera resolution is low, we did not want the mirror to reduce it even further. With that in mind, the resolution of the camera did not have to be high, because the only information we needed was the general shape and amount of light coming through the space between the ball-gage and the blade root fillet (see Image Processing).

Sensor and Motor System:

For the programming of the ABB robot, RobotStudio is used to compile the code. RobotStudio allows the user to program the movements and functions of the ABB robot, it also serves as a fully realized simulation software. It utilizes the language RAPID which can allow for seamless transition between developing code in and out of the simulation environment. The majority of that coding involved using the robot to safely and accurately move the EOAT (End of Arm Tooling) along the blisk blades. The coding was primarily developed in the simulation side by first inputting the models of the EOAT, blisk and turntable into RobotStudio (Figure 15, 16). Then reference frames were created for the blisk and turntable for consistent robot arm poses, in the event that the turntable or blisk needed to be moved. A frame of reference was also created for the EOAT's tip to know the exact points of contact between it and the blisk.

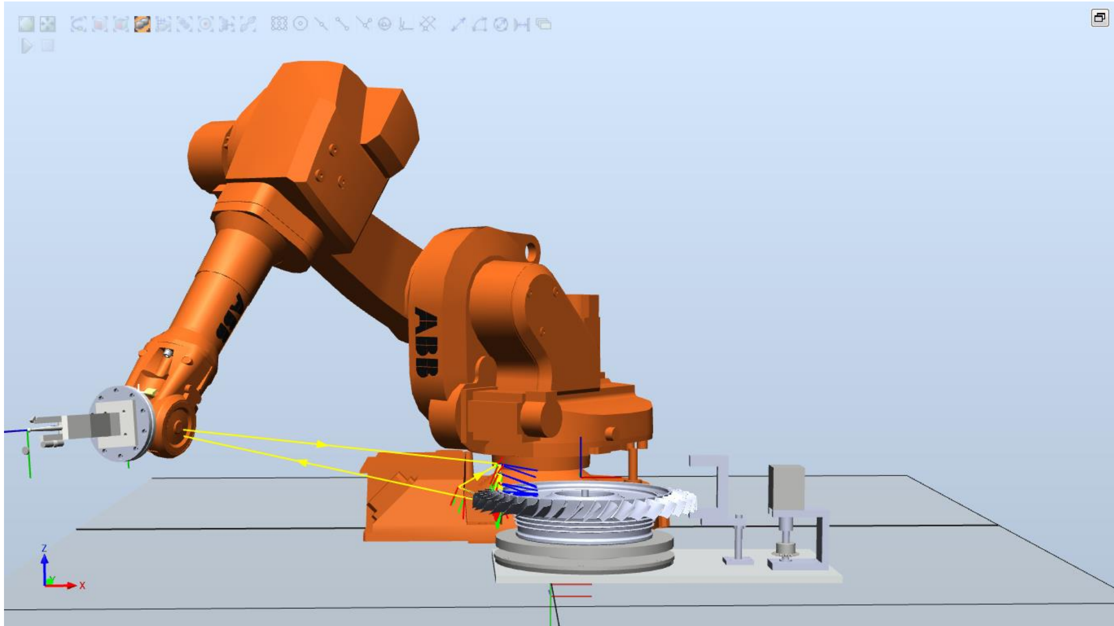


Figure 15: System Model in Robot Studio

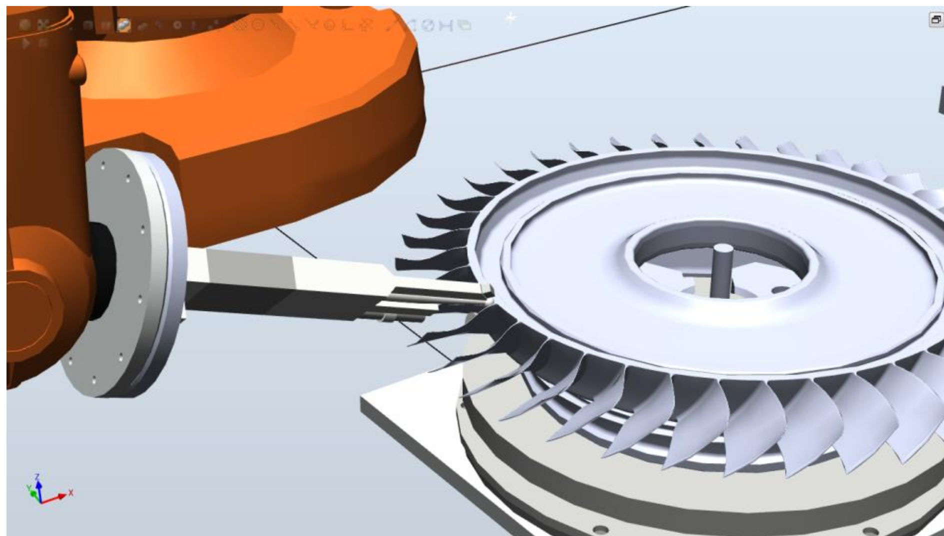


Figure 16: EOAT Approaching Blisk in RobotStudio

The code was developed using a series of target goals (poses) for the robot to reach in relation to the blisk and turntable's frames of reference. There were two different sets of poses

used by the robot one for the concave side of the blade and one for the convex side. Each goal had a positioning relative to the blisk and turntable, which needed to be hit by the tip of the EOAT. There were different 14 poses used for the concave side inspection, 17 for the convex side and 1 “home” pose. The blade root fillet was approximately 38.1mm in length, a precision of approximately 2.7mm per pose on the concave side and 2.2mm per pose on the convex side. Now the camera precision per frame, calculated later in the paper, was 0.635mm per frame for the concave side and 0.423mm per frame for the convex side. This yielded a ratio of approximately 4.2 frames per pose on the concave side and 5.2 frames per pose on the convex side. It is important to note that the distance and timing between the poses was not uniform, so this just yields an estimate. While more poses does yield more precision between each step, we feel that the ratio of 5.2 frames per pose will be sufficient for our needs.

For each point it was important to ensure that the ABB robot could hit that point when transitioning from its previous location. There were two separate types of movements available to the ABB robot linear and joint motion. Linear motion has the tool center point, i.e. the tip of the ball bearing, move in-between poses in a linear line. Joint motion simply moves the joint angles from their position in the current pose to their target pose. Given the very small space in-between poses, 2.7mm for concave and 2.2mm for convex, we felt that joint motion met the needs for our pathing.

It was also important to avoid a collision between the blisk and EOAT (Figure 18). Initially this was done using eyesight; however, an improved method was to use RobotStudio’s collision detection system. The problem this presented was that the EOAT’s tip is meant to touch the blisk. This meant the collision detection was live during all of the inspection processes. This issue was addressed by using an EOAT model without the ball bearing (Figure 17).

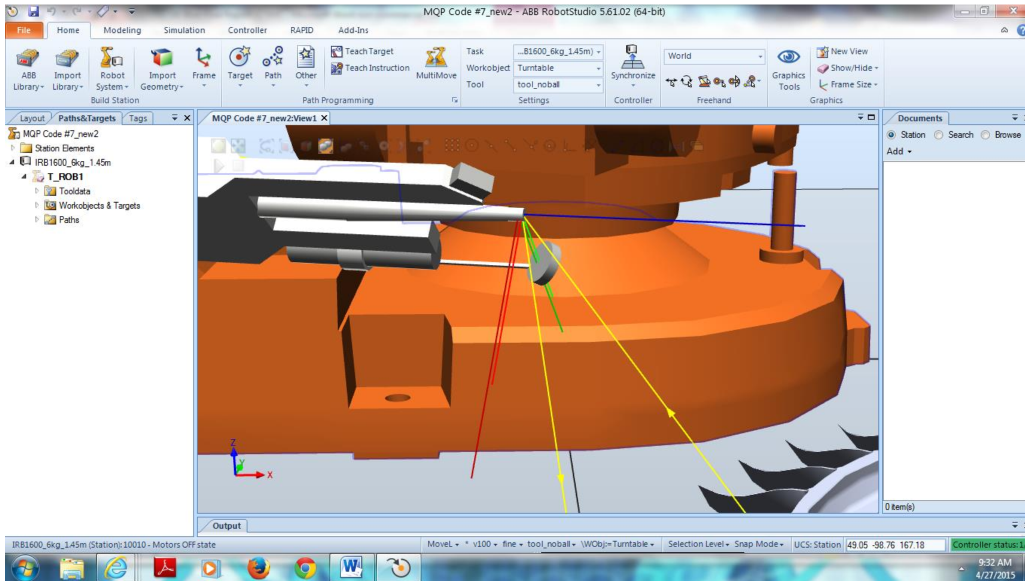


Figure 17: Model of EOAT without Ball Bearing (older Robot Studio Code)

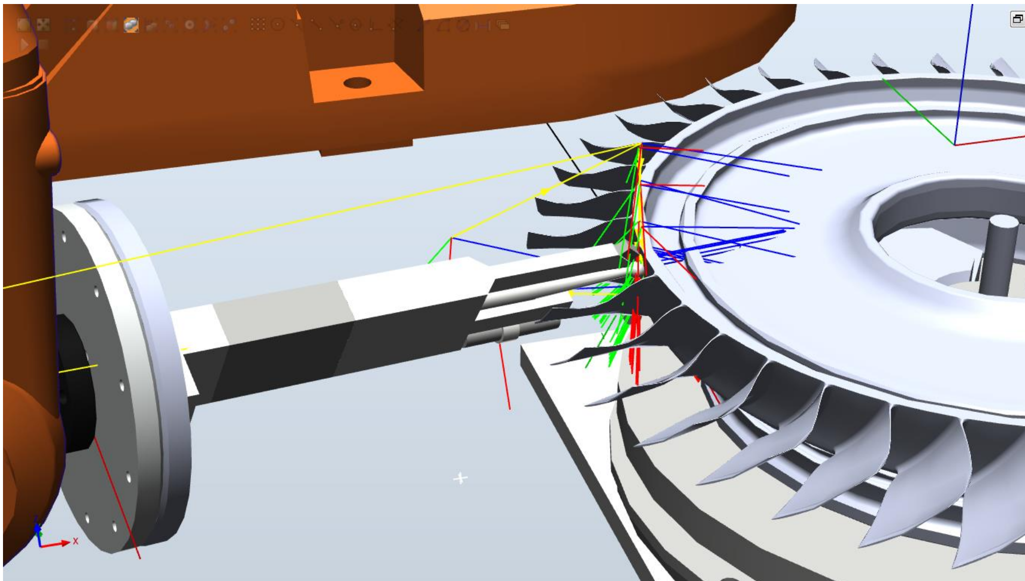


Figure 18: EOAT in Blisk Model

The IRC5 robot controller communicates with the sensors on the turntable via an Arduino Uno microcontroller. The Arduino Uno serves as the central hub for the photogate

sensor, the stepper motor, ABB robot and MATLAB. The Arduino Uno was the master processor during the inspection operation. It had a two way I/O connection to the robot controller, the slave processor, via a 25-pin connector. However, there was an interfacing issue with the two systems. The Arduino only outputs 5 volt logic signals at around 5 volts, while the inputs on the ABB robot used 24 volts. Similarly, the ABB output signal needed to be stepped down from 24 volts to 5 volts to interface with the Arduino. This was solved by utilizing an op-amp to help boost the signal and a voltage divider to decrease the signal. The diagram showing the layout of the circuitry is below (Figure 19). Note that the grounding line between the ABB and Arduino isn't in the physical system and they don't share a common ground (Figure 20). This was the most likely reason that the op-amp circuitry doesn't currently function properly. The 5 volt output going into the op-amp wasn't properly grounded with the rest of the circuit. This meant that when the Arduino Uno output the high 5 volt signal, the op amp didn't necessarily output the corresponding high 24 volts to the ABB I/O connector. The voltage divider worked in spite of this error.

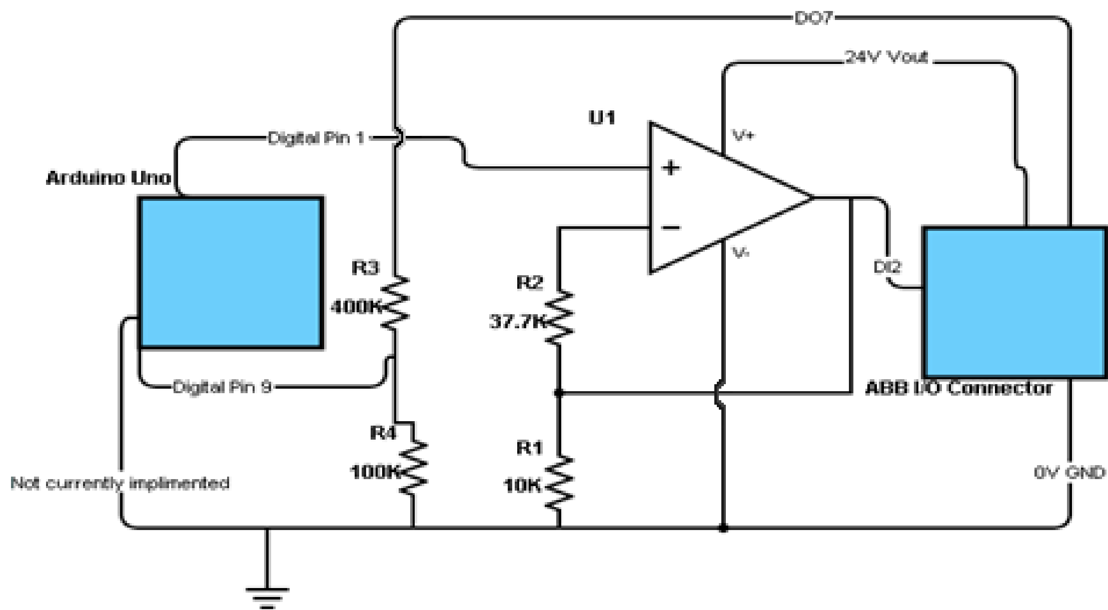


Figure 19: ABB and Arduino Wiring Diagram

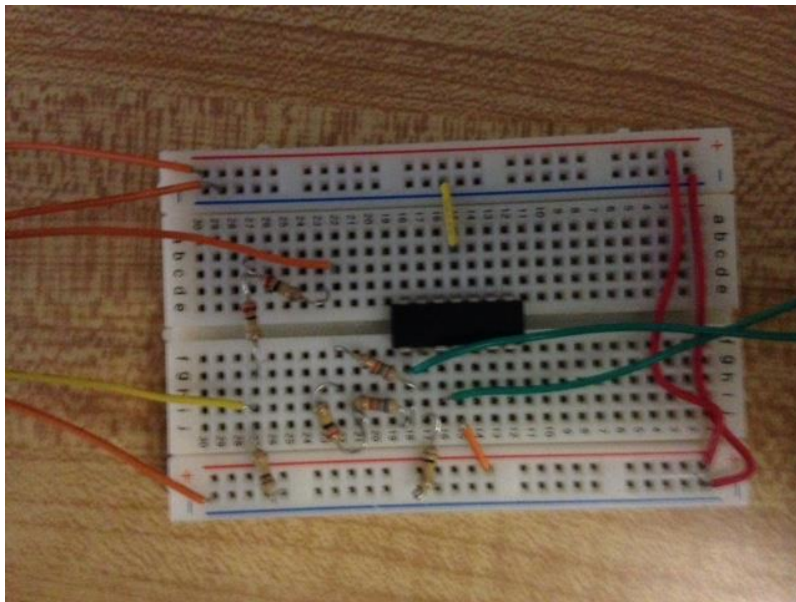


Figure 20: OP-amp and Voltage-divider Circuit

The Arduino connected with the stepper motor via one I/0 connection and its 5 volt power line as well, via a motor driver. The Arduino moved the motor by sending a step

command via digital pin 11 to the driver. Each time that a logic 1 was detected by the motor driver, it incremented the motor by one set amount of rotation or steps. The different step sizes were determined by the wiring configuration of the motor driver. The connection diagram for the Arduino and the motor driver is shown in Figure 21. It is taken from the motor driver's website (8). Note that for the project purposes, the direction pin is wired to ground since the turntable direction doesn't change. Also the logic power supply is provided by the computer attached to the Arduino via USB since the microcontroller needs to connect to the computer that performs the image processing anyway. The motor power supply was simply provided by a bench power supply running around 12 Volts and 0.5 A. Finally, the motor connections have points 1A and 1B connected to either the wires A (black) and C (green) or B (red) and D (blue). 2A and 2B were connected to whichever wire pair isn't connected to 1A and 1B. The connection points and their corresponding colors for the motor were also below (Figure 22, 23).

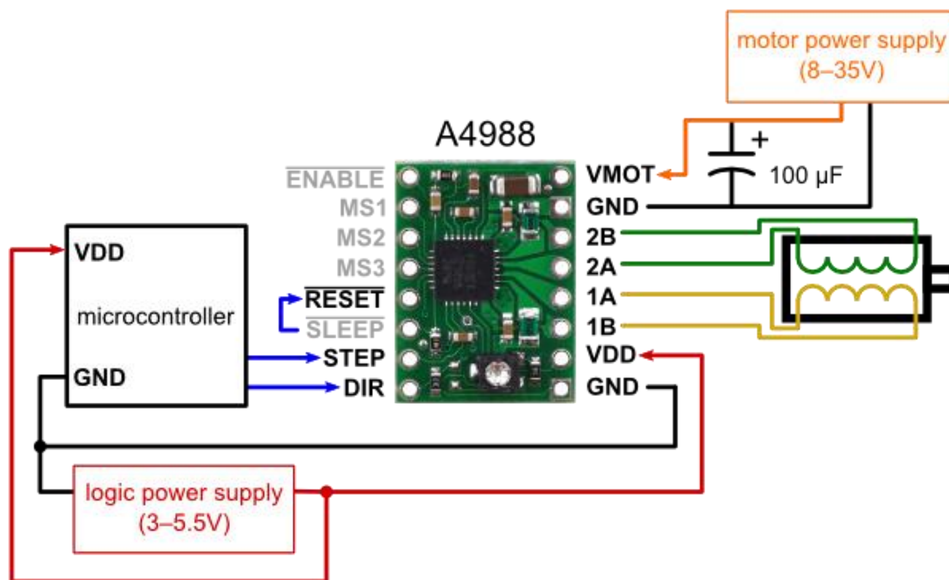


Figure 21: Motor Driver Wiring Diagram (8)

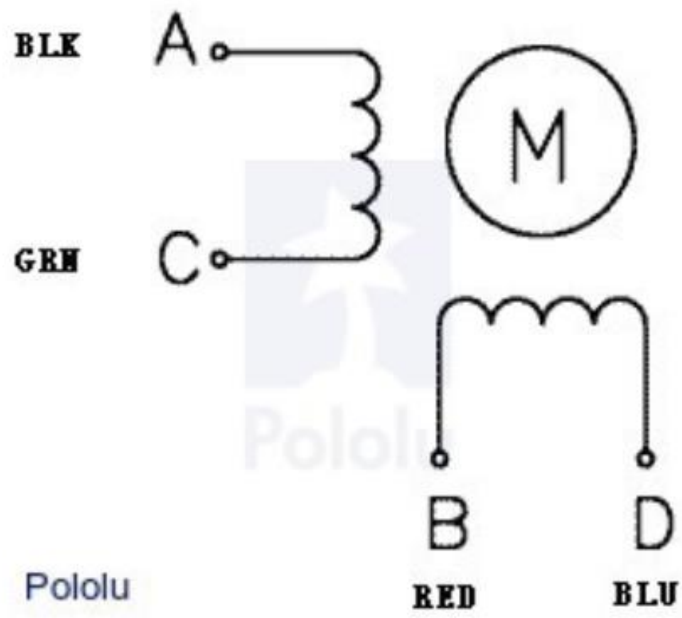


Figure 22: Motor Wiring (9)

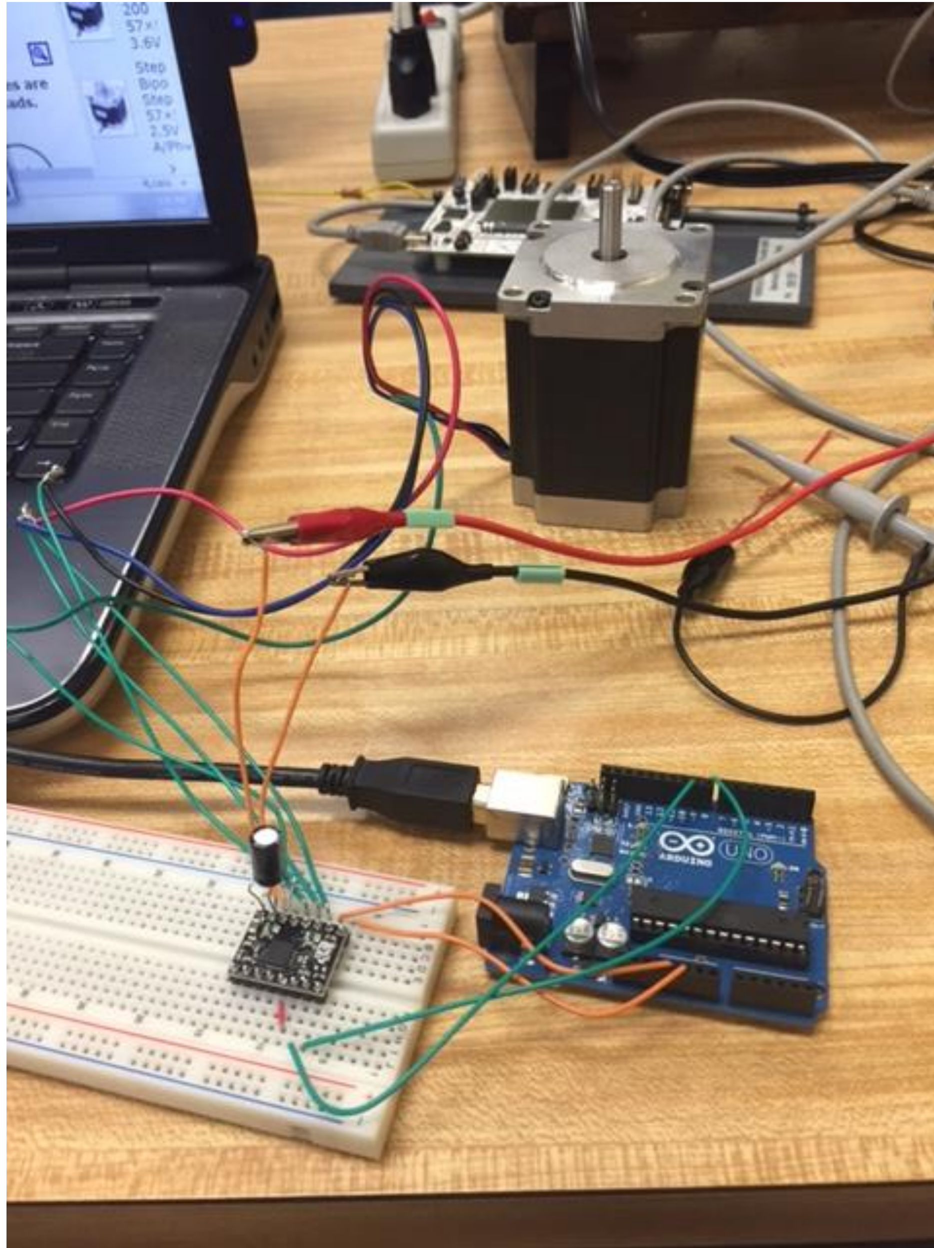


Figure 23: Arduino with Stepper Motor

The step ranges for the motor driver can vary from a full step to one sixteenth of a step depending on the configuration of the motor controller. This serves as a control of how far the motor turns when a high signal is sent to the motor driver's step input. So half-steps were half

the rotation step of a full step when signaled. We used full steps as we thought it allowed for the necessary precision we needed. However, one area for improvement could be to change the wiring to a smaller step increment for greater accuracy. In addition, it could also be possible to switch step sizes during the turntables operation, with the addition of more connections between the Arduino and motor driver. This can allow for extremely precise movements in the motor. The table for different step configurations, taken from the motor driver's website, were below (Table 1)(8).

Table 1: Step Resolution Table

MS1	MS2	MS3	Microstep Resolution
Low	Low	Low	Full step
High	Low	Low	Half step
Low	High	Low	Quarter step
High	High	Low	Eighth step
High	High	High	Sixteenth step

The Arduino code had two different step rates it uses for communicating with the driver (Figure 24, 25). It had "fast" steps and "slow" steps. The "fast" steps were done whenever the increment blisk command was initially done. They were a set number of steps, currently 10, to rotate the blisk most of the way to the next blade position. After those were completed the program sends out "slow" steps. These were half as fast and were used to rotate the blisk slower to avoid overshooting the desired position for the next blade. An area for improvement could have these slow steps be in a lower step size for more precision. The steps for moving the blisk

and the signal sent out when the motor is not moving are shown in Figures 24 and 25 respectively.

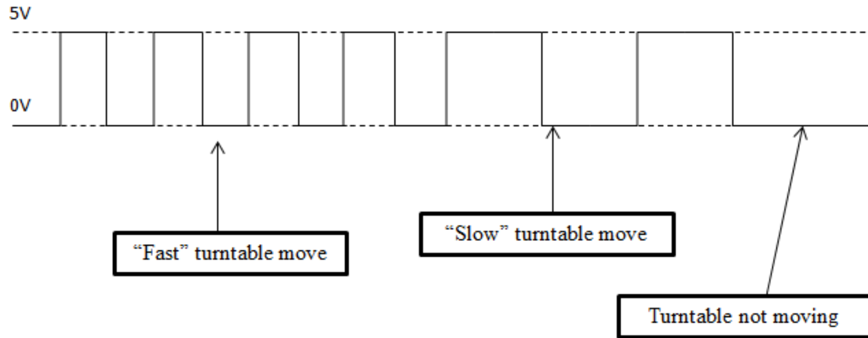


Figure 24: Step Signal Diagram

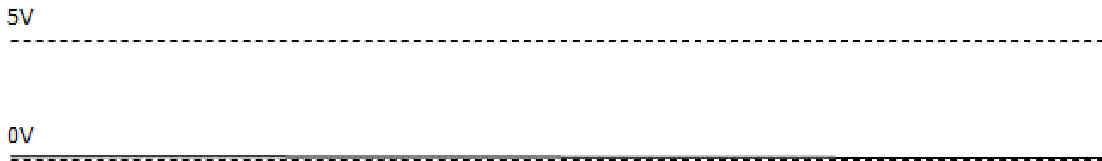


Figure 25: Signal for No Motor Movement

The photogate sensor used was a basic IO sensor which sent an infrared signal between a transmitter and receiver set a fixed distance apart. When the signal was interrupted, that indicated a blade was between the two sides of the photogate. The photogate was connected to the Arduino using a custom shield board which went on top of all of the pins of the Arduino. This meant that for the five pins used by other subsystems to connect to the Arduino, there were wire connectors sandwiched between the shield and the Arduino. These connectors allow for the Arduino to connect to the sensor, motor driver and ABB controller simultaneously.

For the photogate sensor there were two spots on the blisk which have gaps in between blades. When looking through the blades from the top of the blisk towards the bottom they do

not overlap. These were shown in Figure 26 below. The first was very close to the inner circle of the blades and the second was on the farthest edge. These were the possible locations for the photogate sensor's beam since they were present in between each blade. The photogate was placed on the outer edge of the blades to detect blade movement.

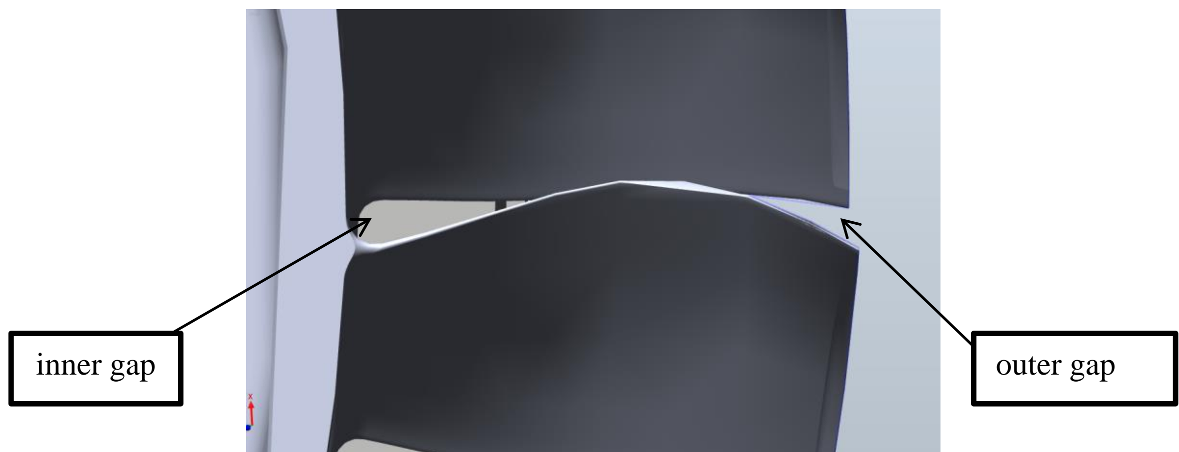


Figure 26: Blisk Coverage Gaps

Finally, the Arduino connects with MATLAB via the UART connection between it and the computer. The UART has a baud rate of 9600 and was only used one way to send signals from the Arduino to MATLAB. The Arudino doesn't send any data via the UART except for the signal telling MATLAB when to start recording using the camera. However, when it was not signaling to start recording, it periodically sent out a pulse to prevent MATLAB from timing out. The computer running MATLAB was also used to power the Arduino via USB.

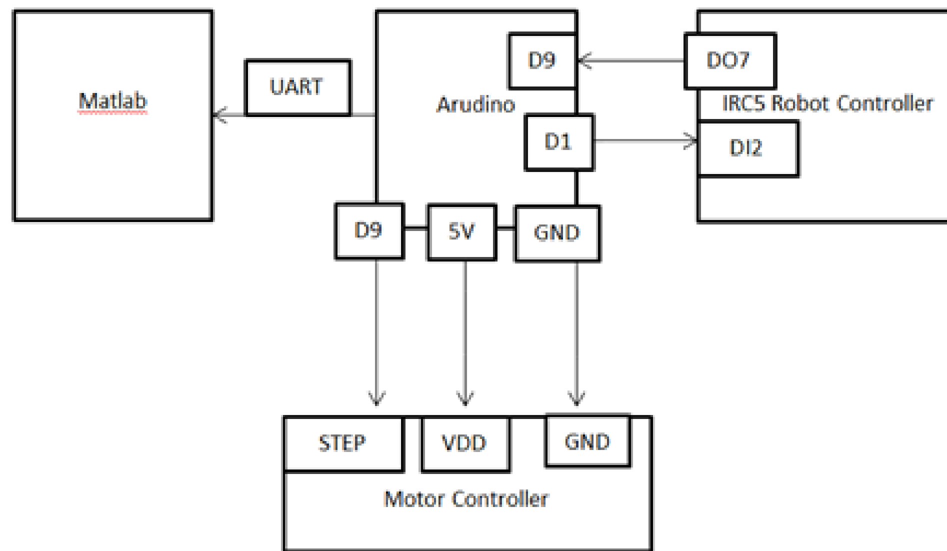


Figure 27: Full Signal System Diagram

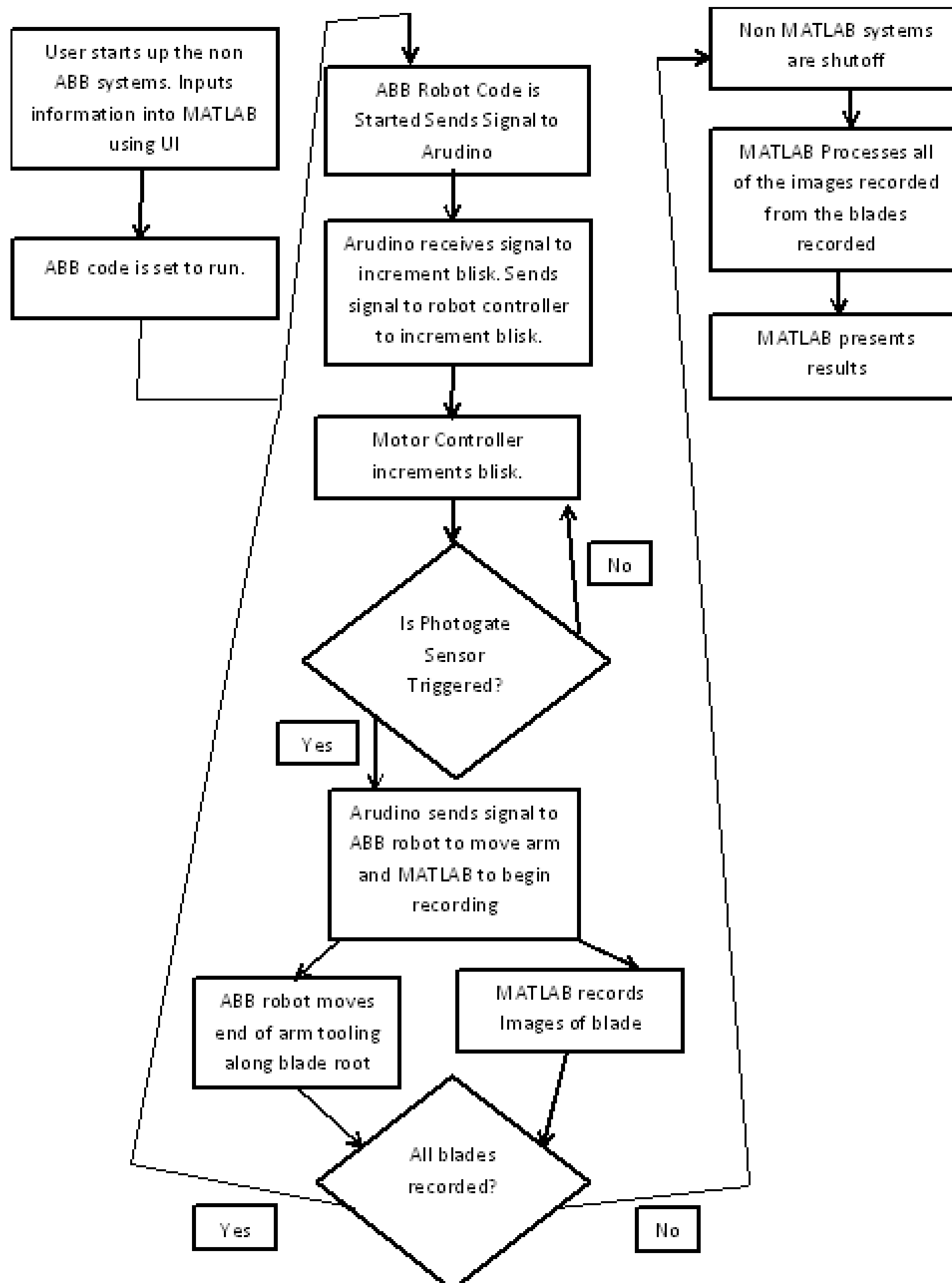


Figure 28: Flow Chart

Operation of the full system by a user was fairly straight forward (Figure 27, 28). To begin, the Arduino should be powered by plugging into the computer via USB. The motor power supply should also be turned on at the previously mentioned 12Volts and 0.5Amps. While the current limiting on the power supply doesn’t have to be exactly 0.5A it shouldn’t exceed 2A to avoid damaging the motor driver. Next, MATLAB should be run, running the start screen script.

Once the user inputs the desired values into the screen (elaborated in the MATLAB section), the user simply hits “OK” to ready MATLAB. Finally, the ABB robot should be turned on and running the ABB robot RAPID code. This will start the process of inspecting the blisk. The ABB robot code right now should be run continuously until every blade has been inspected the user wants to inspect can be inspected. A future expansion of this would allow for a user to input a set number of blisks to inspect. Any errors that were detected while running the RAPID code will appear on the ABB FlexPendant. Similarly any MATLAB -related errors will appear in MATLAB on the PC. When finished, the robot should be stopped first, then the power supply, then the Arduino, and finally MATLAB. Note: MATLAB’s processing time can be longer than the rest of the inspection process since MATLAB currently runs its image processing after all of the data collection. This was done for simplicity purposes. The time metrics regarding this were discussed later in the Image Processing section of the Methodology. Possible improvements in this area were discussed in the Recommendations.

We had planned for the Arduino to make use of two strain gauges attached to the elastomer of the EOAT. This would serve as a way to indicate contact with the EOAT and the blisk. The plan was to have the strain gauges measure the change in the elastomer’s strain when the tip of the EOAT was in contact with the blisk. The original design was to have vertically-mounted strain gauges under and over the elastomer connected via superglue or a similar adhesive.

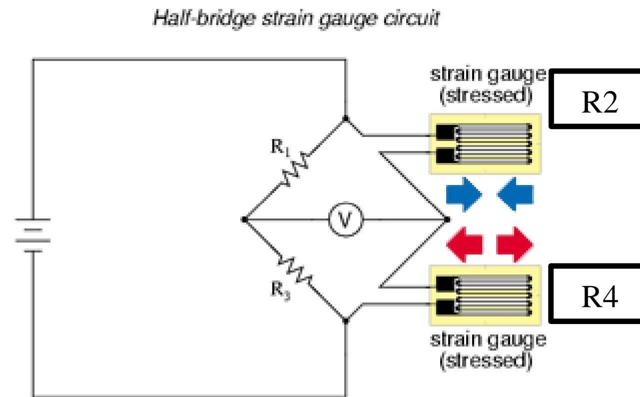


Figure 29: Half-Bridge Diagram

The strain gauges would be connected via a Wheatstone half bridge to show voltage changes (Figure 29). Strain gauges were used to measure forces applied, but they were sensitive to temperature fluctuations. A half bridge was chosen since that configuration eliminates the temperature problems that a quarter bridge would cause. It also offers greater sensitivity than the quarter bridge. Finally, while not as sensitive as a full bridge configuration, the half bridge only requires two strain gauges which was more size appropriate given the more limited space on the EOAT.

The strain gauges would also require amplifiers to boost the relatively small signal given from the bridge. Assuming the system uses the input voltage of 5 volts from the Arduino, as the strain gauges would certainly have to connect to the Arduino. Also we can assume a calculated force applied of around 1 gram's weight force or 0.0098N. This around the force of a human applying the ball bearing calculated via lightly pressing a finger to a force plate. We will assume a FSG Series Force Sensor 0-5N as it was a force sensor which can handle that level of sensitivity. (11) They have a typical resistance of 5K and a range of 4K – 6K so the nominal resistance of 5K will be the value of R1 and R3. We get the relationship of Vin and Vout using the equation below.

$$Vout = \left[\frac{R_3}{R_3 + R_4} - \frac{R_2}{R_1 + R_2}\right] Vin$$

For a baseline @ the max of 6K

$$Vout = \left[\frac{5K\Omega}{5K\Omega + 6K\Omega} - \frac{6K\Omega}{5K\Omega + 6K\Omega}\right] 5V$$

Vout = -1Volts

For a baseline @ the min of 4K

$$Vout = \left[\frac{5K\Omega}{5K\Omega + 4K\Omega} - \frac{4K\Omega}{5K\Omega + 4K\Omega}\right] 5V$$

Vout = 1Volts

This means that in order to get the 5Volt signal necessary to the Arduino a gain of approximately 5 would be necessary for the op amp to allow for the case of a swing from maximum to minimum. However, the strain gauge wouldn't be swinging from maximum to minimum rather, would be changing by its minimum level change 0.0098N. Using the published sensitivity of the device (7.2mv/V/N) and an excitation voltage of 5 Volts the gain can be calculated.

$$0.0072 \text{ V/V/N} * 0.0098N * 5V = 0.0003528$$

$$\frac{1}{0.0003528} \approx 2834.46$$

$$2834.46 * 5 = 14{,}172.3$$

With that being the change necessary we need to add another approximately 2,834.46 gain in order to compensate for the fact we would be looking for the smallest change possible on

the strain gauge. This would yield a gain necessary of around $2834 * 5$ or $14{,}172.3$ to detect the force we suspect. However, this device could probably not be used on the EOAT as currently designed.

Image Processing:

The last major sub-system in our design was the image processing. This begins once the images have been taken from the digital camera. The image capture process was done utilizing an endoscopic camera which saves data using a basic video capture program called VLC Media Player. VLC media player also has the ability to store frames from videos to images. It can also control whether to store every frame in a video, or skip frames. The images can then be loaded into MATLAB for analysis. All of the functionality on VLC media player would be handled by command line codes in MATLAB. This allows for MATLAB to fully automate the image capture process and analysis by itself.

The MATLAB code was operated by the user through a start screen with three basic parameters (Figure 30). The first was the test number which indicates which directory to store the test result. The next parameter is the blade root number which tells the program how many blade roots to test. This was the number of blades desired to test multiplied by two; the user would, for example, enter 78 to test the full 39 blades of the supplied sample blisk. Finally there was a toggle if the test was a minimum or maximum test. This indicates to the code which kind of test to run: max or min. The starting screen is shown below.

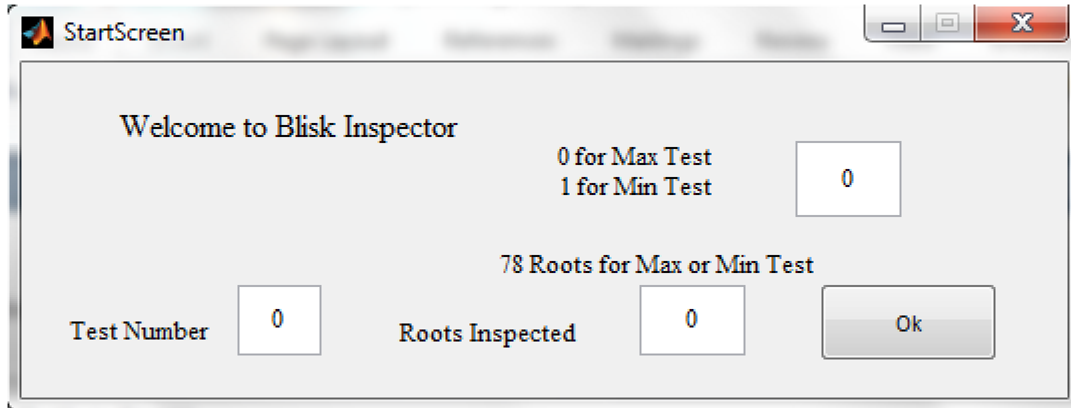


Figure 30: MATLAB Start Screen

One major concern was the frame rate needed to properly provide an analog for the human eye during the inspection process. One approach was to try and get a rate of around 30 frames per second. This would yield an analog similar to a human examining the blisk. This was based on one of the major digital TV and video frame rates of the 30p standard (12). This was based on the idea of requirements that the coding would be at least as effective as the current process of a human inspecting the blisk. While we chose 30 frames per second as a reasonable starting point based on the 30p standard, we do not have enough data argue whether or not this was sufficient for the examination process.

The length of the blade root fillet was around 1.5 inches or 38.1mm. The process takes approximately 2 seconds for the concave side and 3 seconds for the convex side. This yields a frame every 0.635mm for the concave side and 0.423mm for the convex side. They were different due to the increase in time for moving along the convex side from increased pose complexity. If necessary the concave side inspection could be slowed down to have a consistent 0.423mm spatial relation for both sides. Also if during more extensive testing this spatial resolution was found to not be high enough, the frame rate could be increased or the inspection process slowed down to increase it. It is also important to note that, if necessary, a buffer can be

implemented into the code. This can allow the processing to take place after the images were captured.

In terms of timing for the image processing under our current design a blade side takes just over a minute, 63.38 seconds to process. This was calculated using the tic and toc commands stopwatch timer commands in MATLAB. This was for 244 images on a side of a blade, with each image being 110 by 110 pixels. This was the time required to compute a 2D correlation function over 9 different image offsets. Computing the correlations for the 0-offset case only takes 5.74, or 0.04 seconds per image processed. Alternatively put we have an image processing rate of approximately 25 frames per second. Since the current image processing time exceeds the data acquisition time, we elected to collect all the images then batch post-process the data.

The image processing was done utilizing an image processing toolbox in MATLAB. MATLAB was chosen due to the variety of image based processing commands it can utilize. The code follows a basic structure shown in Figure 31 below.

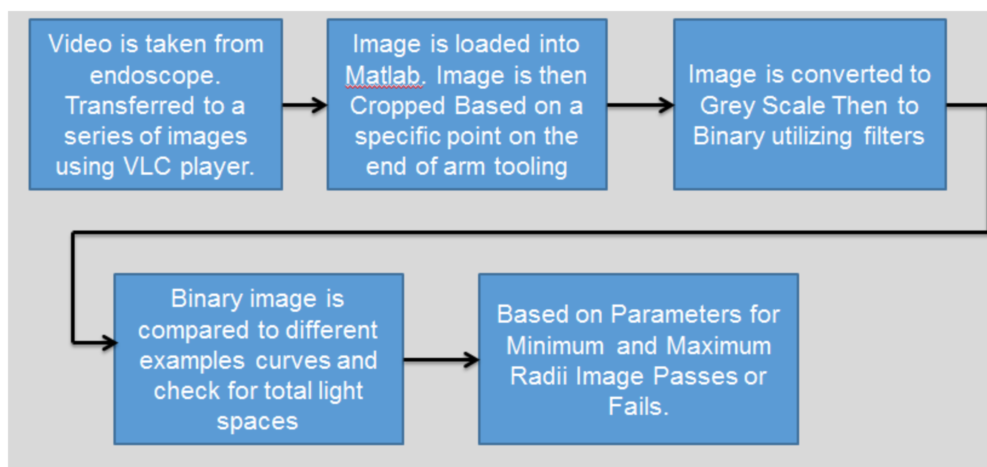


Figure 31: Coding System Diagram

Once the image was loaded into MATLAB the first major challenge was to properly crop the image. This is done utilizing a specific reference-point on the end of arm tooling. This point gives a consistent frame of registration regardless of the location of the ball bearing within the captured image. This point will be made using an easily recognizable object for the image-processing algorithm to detect. We chose a dot of lime green paint was applied to the ball bearing for this purpose (Figure 32). Our camera takes color images with a bit depth of 24 bits. The colors in the images were a combination of different amount of red, green and blue (RGB). Lime green has a color which was recognized by the computer as having an incredibly high green amount in comparison to red and blue. This makes it ideal for detecting it specifically in an image; the same concept was used in video production green screens.

After the reference point was found, the processing used the cropping procedure on the image. This consisted of finding all of the pixels which met the lime green RGB requirements. For our system that means a red value of less than 160, a green value of higher than 130 and a blue value of less than 60. These values where chosen as an approximating starting parameters for lime green in the pictures based on examining the sample images. More exhaustive testing would help better define those values. Those pixels were then averaged to find the approximate center of the lime green dot. Finally, the image takes a square image crop to the left and below that point (Figure 33). Next, the image was converted to greyscale, and finally black and white (Figure 34, 35). Once an image was converted into greyscale it had a value associated to its hue of grey with white being a value of one and black being a value of zero. This value was compared to a threshold when the image was then converted into binary. If the value was above the threshold the pixel was listed as a one or white pixel in the binary image, otherwise it was a

zero and was black. The greyscale to binary conversion threshold used was 0.6. The final binary image rendered the light and dark areas of the image as a matrix of ones and zeros.

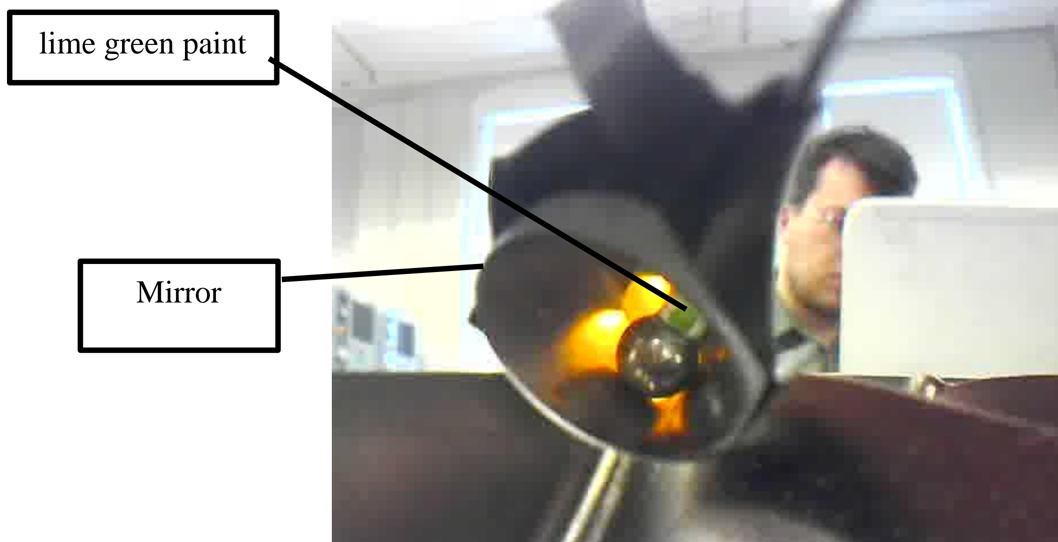


Figure 32: Sample of Maximum Ball Image

Figure 33: Sample of Maximum Cropped Image

Figure 34: Sample of Maximum Greyscale Image

Figure 35:Sample of Maximum Binary Image

Once the binary matrix was made, the image can be tested for the presence of a “smile”. A lack of that smile with light coming out of the side of the ball is known as “whiskers”. Examples of these were shown in Figure 36 below. A maximum ball bearing test should have the smile, while a minimum bearing test should not have a smile and instead have whiskers. The smile’s presence or lack thereof was determined by comparing with the sample images. A sample matrix (image) can be compared to the two template binary matrices (images) to determine similarity. It lists the similarity between the two images as a percentage. This test was done by comparing the sample image to known passing images. Passing images were picked

based from the sample images which best demonstrated the smile and whiskers. Areas for improvement in the validity of the selection of sample passing images were in the recommendations and conclusion.

Figure 36 Smile and Whiskers

The matrix comparison was done for each individual image. This was then expanded for the use of all the images in the video feed. Once all of the images were loaded, they were compared with an appropriately matching sample image. Depending on how well an image matches a sample image template, the matching percentage was calculated using the test program. This was explained later in the report. This was then averaged between the results of all of the test values for a given side of a blade. This was then compared to a certain threshold to determine if that side of the blade root fillet passes or not. This can be expanded upon for each side of each blade. Once all of this information was collected it was displayed to the operator so they know if a blisk has passed or not. It also indicates which blade has a fault and whether or not the error was on the convex or concave side of the blade. For an example see the image in Figure 37.

Figure 1

File Edit View Insert Tools Desktop Window Help

	Blade Number	% Match Example	Pass 1/Fail 0
Concave	1	0.9670	1
Convex	1	0.9441	1
	2	0.9283	1
	2	0.9441	1
	3	0.9652	1
	3	0.9441	1
	4	0.9652	1
	4	0.9441	1

Figure 37: Results Display Screen

Results:

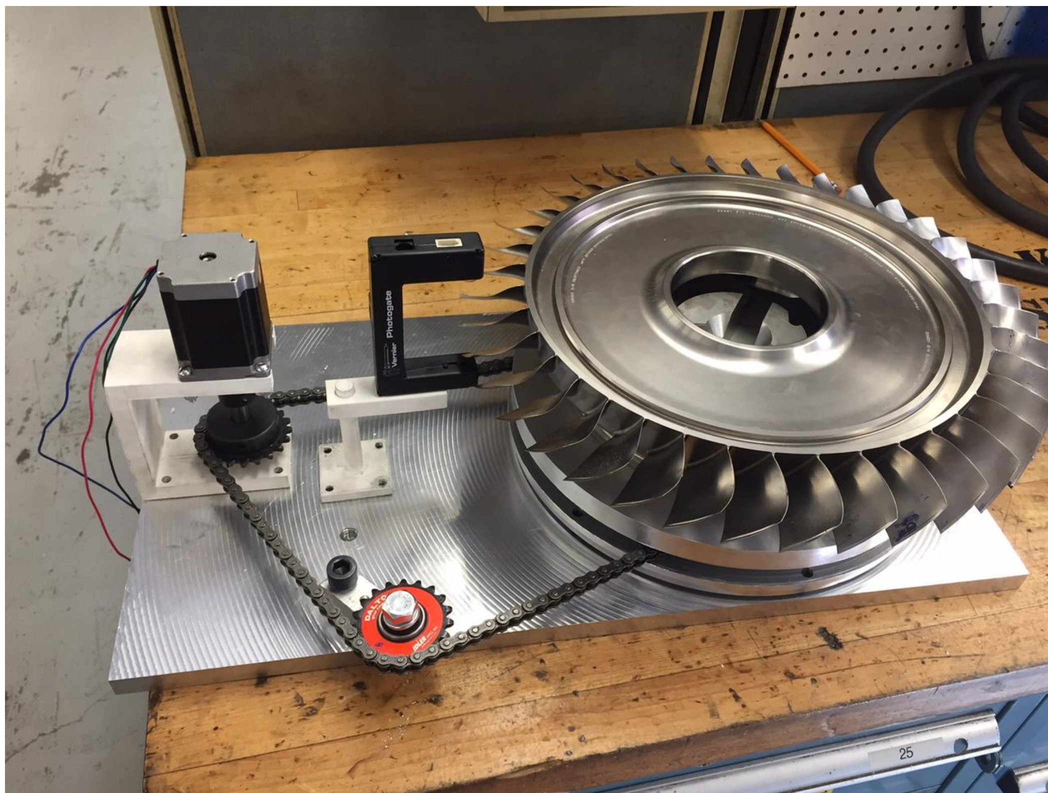


Figure 38: Machined Turntable Assembly

This section will discuss the outcome of the project and what the status of each system was currently. The overall results were an unfinished product, with each subsystem completed to varying degrees, but not working together to the fullest extent. This section will be divided into subsections which will describe each subsystem.

Turntable:

The turntable can be seen in Figure 38 in its current state. We were able to machine all parts needed for the design, but there were flaws that need to be addressed.

The first part that needs attention were the jaws themselves. Due to tolerance error, the jaws don't quite grip the blisk when extended fully. This problem was temporarily fixed by applying three strips of masking tape to each jaw. The tape adds 0.003" to the jaws and provides a higher coefficient of friction between the aluminum and the titanium. The jaws would need to be placed back into the milling machine and lengthen the slots by a few thousandths of an inch in order to permanently address the issue. Also a tool-pathing error led to a broken drill bit, in one of the jaws. This was not a problem as it did not mar an important section of the jaw (see Figure 39).

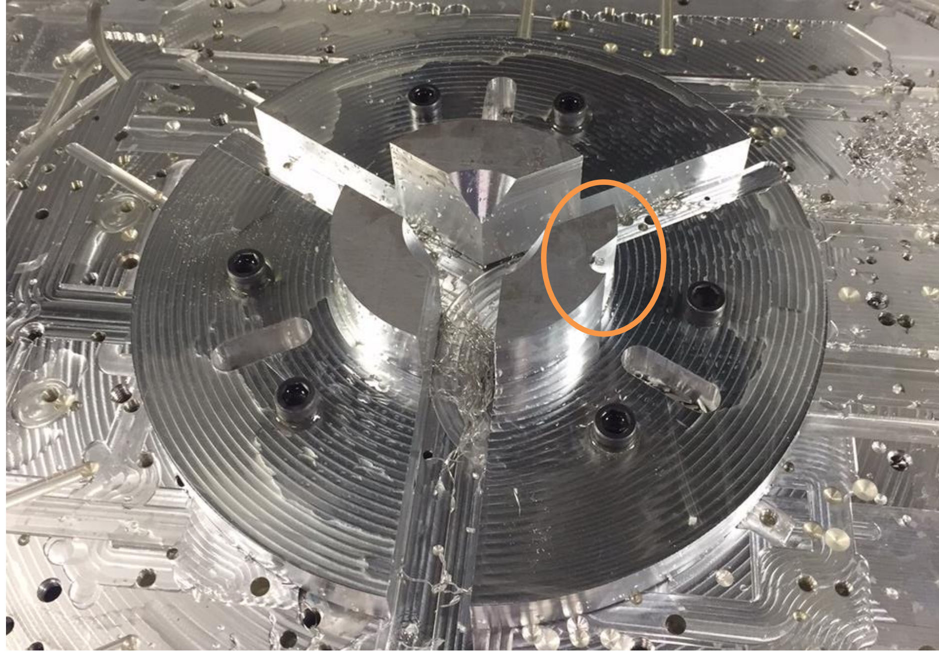


Figure 39: Chuck Jaws

Figure 40: Unfinished Sprocket

The chuck jaws' points needed to be changed as shown in the figure above; this change was to make it possible for the blisk to fit around the jaws. Due to the initial design of the jaws, when they were closed, there were three points that remain the full diameter of 13". The tips were cut to eliminate this issue, so now the blisk can be fit around the jaws.

Figure 41: Turntable with Blisk

The main issue that remains in the turntable was the chain assembly. During the manufacturing of the turntable cover, it came to our attention that the wrong size aluminum stock was ordered and, due to time constraints, the design was changed instead of ordering more stock. However, these changes left very little space for the chain to move without hitting the base plate. The chain can also catch on the edges of the plate which causes the motor to skip. After closer inspection, another problem was discovered that cannot be solved without re-manufacturing the turntable. The sprocket is actually not entirely concentric with the assembly, which causes the

chain to tighten and loosen when rotated. This issue probably occurred when the plates were machined.

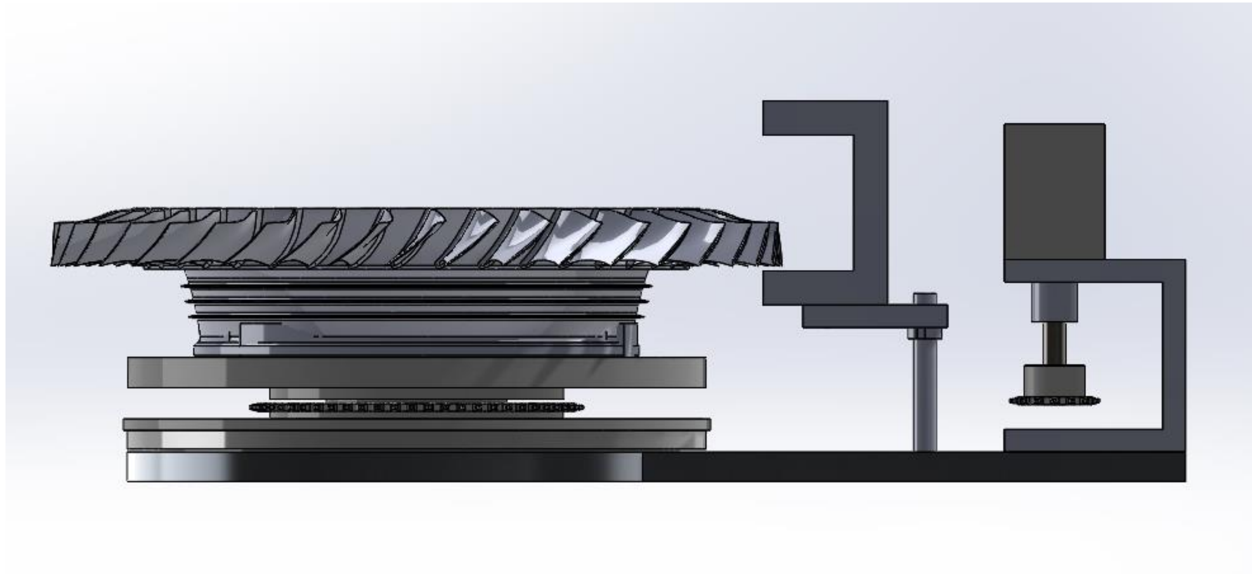


Figure 42: Side View of Turntable Assembly (shown w/o chain)

End of Arm Tooling:

The end of arm tooling was printed successfully including the new ball-gage shafts. The tool was completed and assembled as shown in Figure 38. The LED and mirror have to be positioned carefully in order for the ball to be seen by the camera. An issue we ran into was with the elastomer and its qualities. The weight of the tool attached to the elastomer makes the tool sag by about 10 degrees when the inspection is being performed. This leads to complications with the tool path during the inspection. To temporarily amend this, small strips of durable tape were stretched across the elastomer to prevent sagging. The elastomer dimensions and material needed to be reviewed to prevent that problem from occurring. The material used for this iteration was the only option for printing on campus.

Further improvements to the tool could include repositioning of the camera to be under the branch it is on currently. The same could be changed of the LED position. A smaller LED could be positioned closer to the ball, or a stationary light source attached to the base plate. Any of these new configurations would give the tool a smaller profile to fit in between the blades.

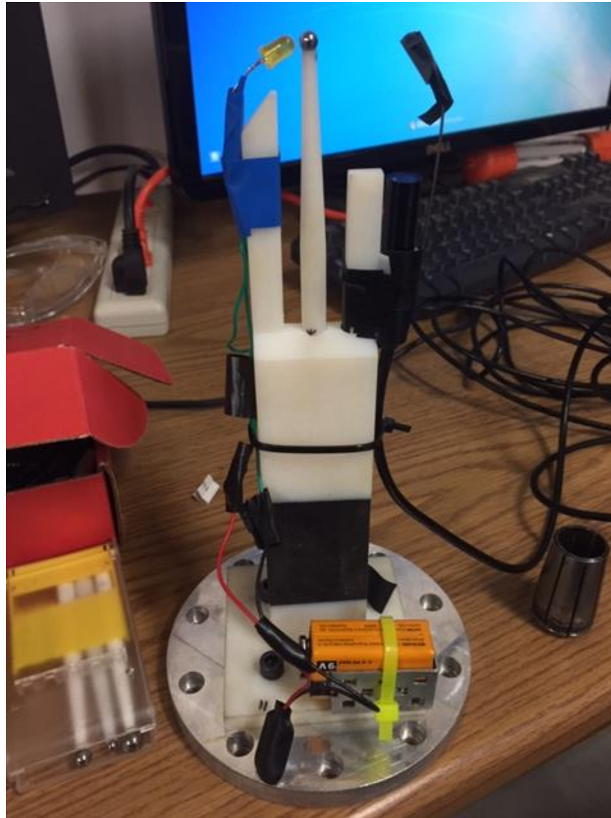


Figure 43: Completed EOAT

Sensor and Motor System:

The sensor and motor were almost complete with a couple of issues. First of all as previously mentioned in the Methodology section, the op-amp I/O connecting the Arduino to the ABB robot controller was still a little buggy. That would need to be worked on to improve the system. Also the timing of the video capture for MATLAB and the movement of the ABB robot

need to be worked on to get that more seamless. Finally, the RAPID code could be worked on to more accurately place the EOAT into the blisk. These problems impact the project such that the MATLAB testing was using manually acquired data versus data acquired from a fully functioning automated system. Besides these main areas of concern the individual subsystems of the sensor and motor as well as their connections seem to work reasonably well.

Image Processing:

The vision system was fully functioning based on the data we have access to. The cropping function uses a green dot as the target for the image cropping operation. Unfortunately, the project has yet to yield system-generated data, so hand-acquired data was used. Therefore the testing that has been done was tried on images that had the EOAT physically placed and recorded in the blisk. The hand-acquired data seemed to be very close to data that would habe been acquired from the fully functioning system.

The algorithm for detecting passing or failing was used on the binary array gained from filtering and cropping. The basics for the test was first to convert both the binary image from the camera and the image of the template from an array of 1's and 0's to -1's and 1's. An area for improvement of this aspect was covered in recommendations and conclusions. This is done via multiplying each element in the grid by two and subtracting one.

$$x = x * 2 - 1$$

$$x = \begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} \end{bmatrix} * \mathbf{2} = \begin{bmatrix} \mathbf{2} & \mathbf{0} & \mathbf{2} \\ \mathbf{0} & \mathbf{0} & \mathbf{2} \\ \mathbf{0} & \mathbf{2} & \mathbf{0} \end{bmatrix} - \mathbf{1} = \begin{bmatrix} \mathbf{1} & -\mathbf{1} & \mathbf{1} \\ -\mathbf{1} & -\mathbf{1} & \mathbf{1} \\ -\mathbf{1} & \mathbf{1} & -\mathbf{1} \end{bmatrix}$$

Once this was done, the two matrices (test and template) were then subtracted from each other, element by element. This gave a correlation between the two matrices.

$$x = \begin{bmatrix} 1 & -1 & 1 \\ -1 & -1 & 1 \\ -1 & 1 & -1 \end{bmatrix} - \begin{bmatrix} 1 & -1 & -1 \\ 1 & 1 & -1 \\ -1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 2 \\ -2 & -2 & 2 \\ 0 & 0 & 0 \end{bmatrix}$$

The algorithm then gets the absolute value of the resulting matrix.

$$abs \begin{bmatrix} 0 & 0 & 2 \\ -2 & -2 & 2 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 2 \\ 2 & 2 & 2 \\ 0 & 0 & 0 \end{bmatrix}$$

Next the elements of the matrix were added together.

0+0+2+2+2+2+0+0+0=8

Next the sum of the elements of the matrix were divided by two to get the total number of matches. This was then divided by the total number of elements in the array. This was the percent difference of images. This was then subtracted by one to calculate a correlation.

1- (2/8)/9= 1- (4/9) ≈ 0.56

This was the individual test for each un-shifted image array. However, when doing an image test it was important to offset the cropping in case the starting cropping doesn't work perfectly. This was done via a correlation function performed over a subrange of offsets. To do this the grid uses an offset pattern shown below. This served as a compensation mechanic and allows for small errors in the mechanical system. Each offset increases the processing load by one image at the previously calculated rate of 0.04 seconds. The number of additional pixels considered for each offset was 110 for the non-diagonal offsets and 219 for the diagonal offsets. One pixel was hit by the change in both directions for the diagonal offsets which was why it was 219 new pixels and not 220. This yields a processing load increase rate of approximately 0.363

milliseconds per pixel for the non-diagonal offsets and 0.183 milliseconds per pixel for the diagonal offsets.

(-1,1)	(0,1)	(1,1)
(-1,0)	(0,0)	(1,0)
(-1,-1)	(0,-1)	(1,-1)

The correlation function was calculated for each of the offsets of interest and the highest value was chosen. This was the highest correlation obtained for that image to the test template. The peak correlation values of all the images in the test route were then averaged together. This average serves as a net rating for the side of the blade being examined. If the average value was above a certain threshold the blade side passes. If the average value was below that threshold the blade side fails. A possible improvement to the system could be to implement a failsafe for a small set of very anomalous data. Since all of the images were averaged together, the rest of the images could compensate for an area of the blade failing when the rest of the blade was correct. This could be done by ensuring that each individual image meets a certain threshold in addition the blade as a whole meeting the threshold.

The current threshold was set at .72 passed on our initial testing. This was set by examining the following images. The .72 was taken as a midway point “right answer” between passing and failing sample images. It serves only as a starting point for where the algorithm might fall given more testing. More on how the algorithm would develop given more testing was in recommendation and conclusions. We have 16 data sets with four of the data sets each being compared to one of the four sample images shown below. (Figures 44-47)

Figure 44: Maximum Test Concave Sample Image A

Figure 45: Maximum Test Convex Sample Image B

Figure 46: Minimum Test Concave Sample Image C

Figure 47: Minimum Test Convex Sample Image D

The results of the image tests are shown in the table below. (Table 2)

Table 2: Results of MATLAB Tests

A Library Concave Max	% Match	Blisk Good/Bad
Test 1	0.8197	Good
Test 2	0.7506	Good
Test 5	0.7131	Bad
Test 6	0.7268	Bad
B Library Convex Max		
Test 3	0.476	Good
Test 4	0.5098	Good
Test 7	0.4139	Bad
Test 8	0.3922	Bad
C Library		
Test 9 Concave Min	0.7471	Good
Test 10	0.7879	Good
Test 13	0.7114	Bad
Test 14	0.7222	Bad
D Library Convex Min		
Test 11	0.5945	Good
Test 12	0.5773	Good
Test 15	0.7259	Bad
Test 16	error	Bad

The results show some good and bad news. For both of the concave tests the passing check of 0.72 works well as passing blades were over it and failing blades were under it. However, the convex tests were quite a mishmash especially test 15 which was over the threshold despite being a bad blisk. This means that while the image searching algorithm seems to have a general start of promise for concave side, the convex side could really use work.

Recommendations and Conclusion:

For the project there definitely seems to be a lot of promise in terms of an automated check system for Blisk Inspection. However, as the project has shown, there was still more work

to be done to refine the implementation of the inspection system before a final decision can be made. In general, we would recommend more work on the system to try and continue the research. We do feel that this project does demonstrate some feasibility on a proof of concept level. This should serve as a starting point to move forward.

One aspect of the project for further work was the concept of the reliability and validity of the image search algorithm. First the project needs to be able to demonstrate test and retest reliability. This means similar results should be able to be gained from further retesting of the results. Currently the reliability of the tests was unsure due to the very small amount of data. More tests would indicate that the test was repeatable. Also important was the validity of the data. While the project does have some face validity, more criterion-based validity needs to be in place. This could be done via examining how many passing blisks were passed and how many were failed. This and the reverse determine if the image processing was providing its task and the results were valid. (13)

One of the aspects of validity was the presence of false positives and negatives in our search result. For this application false positives, i.e. a blisk fails when it’s actually fine were much less damaging than false negatives i.e. a blisk passes when it’s actually damaged. In general the idea would be any blisk indicated as damaged could be examined to verify the test results. However, any blisk thought of as fine would presumably be used. Therefore the threshold value for the test wants to be on the aggressive side. With proper tuning from repeated

tests the proper threshold could be found to keep the false negatives as low as possible while not having a massively high false positive value.

Given this basic concept tuning the algorithm would involve a tradeoff between the false positives and negatives. The sensitivity of the data needs to be nearly 100% as anything less was unacceptable for a part which could have serious implications if any defect was not caught. This means more than likely the threshold of failure would need to be so strict that every single failing blisk would fail. Because of this the specificity of the test would suffer. However, the cost of having a 5% rate of unnecessary further inspections happen vs one bad part going through was more than likely worth it. Although as more testing was done it becomes more likely that the threshold was closer to the value to remove most false positives as well. However, this will have diminishing returns as testing goes on (14).

One of the important aspects about the current process was the collection of data and processing was sequentially in two separate steps. All of the data was collected and then the data was analyzed during post-processing. Both of these steps were implemented in the same code. This was a potential major area for improvement of the system. If some more real time elements were to be implemented that could speed up the processing considerably. Specifically, one idea was if the system could take the first blade's data and then after process it while simultaneously getting the data from the second blade. This should be an area examined for future work once the system gets refined enough to worry about bettering performance.

Another processing problem was that of over template processing. Currently the system analyzes the template for its binary matrix every time a blisk was compared with one of the templates. This was very undesirable in terms of unnecessary processing. An area for future

improvement would to instead saving the template images as images save them as matrices. This would also have the added benefit of reducing the memory used by the system as the way the system currently works the sample images were added to the image files of each blisk.

There were a couple of areas to further expand the project once the original specifications were fully implemented. One of the areas would be to work on developing a system to work with blisks of different sizes. This could allow more research to be done on verifying if the method can work on other blisks. On a similar notion, research could be done on multilevel blisks which have multiple layers of blades. This could provide an interesting challenge in terms of EOAT placement. Also, the implementation of a force sensing system on the EOAT elastomer would add greater control with performing the inspection.

References:

(1) About Us. (2014). from www.ge.com

(2) Bladed Disks. (2011) (Vol. 20, pp. 299-325). Dordrecht: Springer Netherlands.

(3) GE Internal Document

(4) Industrial Robots - Robotics | ABB. (2014). from http://new.abb.com/products/robotics/industrial-robots

(5) IRB 1600 - Industrial Robots - Robotics | ABB. (2014). from http://new.abb.com/products/robotics/industrial-robots/irb-1600

(6) Oh, S., Oh, J.-K., Jang, G., Lee, J. H., Lee, J. S., Yi, B.-J., . . . Choi, Y. (2009). Bridge inspection robot system with machine vision. Automation in Construction, 18(7), 929-941. doi: 10.1016/j.autcon.2009.04.003

(7) Mahdi, A., Mahmoud, O., Alireza, K., & Seyed Saeid, M. (2010). Sorting Raisins by Machine Vision System. Modern Applied Science, 4(2), 49. doi: 10.5539/mas.v4n2p49

(8)"Pololu - A4988 Stepper Motor Driver Carrier." *Pololu - A4988 Stepper Motor Driver Carrier*. Web. 20 July 2015

(9)"Pololu - Stepper Motor: Bipolar, 200 Steps/Rev, 57×76mm, 3.2V, 2.8 A/Phase." *Pololu - Stepper Motor: Bipolar, 200 Steps/Rev, 57×76mm, 3.2V, 2.8 A/Phase*. Web. 20 July 2015.

(10) "THE STRAIN GAUGE." *THE STRAIN GAUGE*. Web. 9 Aug. 2015. <http://web.deu.edu.tr/mechatronics/TUR/strain_gauge.htm>.

(11) "FSG005WNPB." *Product Page*. Web. 9 Aug. 2015.

<http://sensing.honeywell.com/product-page?pr_id=145659>.

(12)"Video Frame Rates – 60i vs 60p vs 30p vs 24p – What It Means." *FS Photography Video Longmont Boulder Photographer*. Web. 16 Aug. 2015.

(13)"Reliability and Validity " *Reliability and Validity*. Web. 20 July 2015.

<https://www.uni.edu/chfasoa/reliabilityandvalidity.htm>.

(14)"Sensitivity and Specificity." *Sensitivity and Specificity*. Web. 20 July 2015.

<http://www.med.emory.edu/EMAC/curriculum/diagnosis/sensand.htm>.

Appendix:

RAPID Code:

This code is a series of targets and directions to the targets used by RobotStudio to move the ABB robot. It is written in RAPID and contains of two main parts defining the movement target and the order of target movements.

```
MODULE Module1
	/*These targets are there to demonstrate both the location and positioning of the end of arm tooling for each target. They were determined via the positioning of the end of arm tooling and the blisk in the robot studio model. The first three values are the x,y and z location of the target. The second three values are the positioning of the angles along those axes. The last values remain constant and are unchanged. */

	CONST robtarget Target_11:=[[-94.412081934,-214.405763859,48.280344636],[0.516224851,0.266726249,0.769156242,0.266021967],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_11_2:=[[-94.275075722,-197.406745875,48.28334136],[0.610023784,0.037096588,0.746936796,0.261878309],[-2,0,1,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_11_2_2:=[[-94.12187049,-178.406934291,48.287296524],[0.591712042,0.121962254,0.72963034,0.320377333],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_12:=[[-94.033194771,-167.407291923,48.289420712],[0.567922268,0.172561599,0.719345341,0.360872655],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
```

```
	CONST robtarget Target_13:=[[-93.659093232,-165.813823046,48.950151812],[0.567922268,0.172561599,0.719345341,0.360872655],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_14:=[[-93.323299635,-164.479044104,49.533045445],[0.567922268,0.172561599,0.719345341,0.360872655],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_40:=[[-91.782718521,-158.916526671,52.217334224],[0.571049811,0.166277636,0.720823815,0.35590292],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_50:=[[-91.023535034,-156.343307686,53.55774405],[0.571049811,0.166277636,0.720823815,0.35590292],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_60:=[[-90.580433657,-154.869169221,54.349347364],[0.571049811,0.166277636,0.720823815,0.35590292],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_70:=[[-90.062147895,-153.174542466,55.295706222],[0.594485295,0.115590465,0.730666866,0.315201538],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_80:=[[-89.311607768,-150.789867079,56.722644475],[0.586030565,0.134677484,0.727390679,0.330655337],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_90:=[[-88.634874033,-148.703391708,58.070627108],[0.586030565,0.134677484,0.727390679,0.330655337],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_100:=[[-88.200099534,-147.38543531,58.972592895],[0.586030565,0.134677484,0.727390679,0.330655337],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_110:=[[-88.590775639,-144.88895457,59.27880081],[0.586030565,0.134677484,0.727390679,0.330655337],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_120:=[[-86.375308483,-141.915862966,63.177344631],[0.586030565,0.134677484,0.727390679,0.330655337],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_130:=[[-85.696917679,-140.038389346,64.951278066],[0.586030565,0.134677484,0.727390679,0.330655337],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_130_2:=[[-168.558597591,-168.478315234,73.264677666],[0.586030565,0.134677484,0.727390679,0.330655337],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_137:=[[-96.838273485,-214.301007989,45.158810636],[0.727121954,-0.255257091,0.636516766,-0.031366991],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_138:=[[-96.30651049,-198.102807371,43.372624024],[0.72646378,-0.266326964,0.631964971,-0.044052241],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
```

```
	CONST robtarget Target_139:=[[-96.065231274,-
180.330941955,40.050229428],[0.7363246,-
0.083919374,0.668991928,0.056863196],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_140:=[[-94.428362959,-
164.738301719,40.054772642],[0.751114355,0.126954862,0.606413531,0.22797438],[0,0,0,0],[
9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_150:=[[-94.362835061,-
164.307284705,42.040918613],[0.751114355,0.126954862,0.606413531,0.22797438],[0,0,0,0],[
9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_160:=[[-94.031414669,-
163.131197891,42.285721621],[0.751114355,0.126954862,0.606413531,0.22797438],[0,0,0,0],[
9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_170:=[[-93.172133559,-
160.436023769,43.17604083],[0.751114355,0.126954862,0.606413531,0.22797438],[0,0,0,0],[9
E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_180:=[[-92.852076052,-
158.147608589,43.623518552],[0.75682464,0.111037302,0.609529019,0.208234377],[0,0,0,0],[
9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_190:=[[-90.34939532,-
155.716630671,45.837724108],[0.762016236,0.095043642,0.612226767,0.188351661],[0,0,0,0]
,[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_210:=[[-91.12727458,-
150.30997849,46.591053666],[0.765187367,0.084344336,0.613792262,0.175023957],[0,0,0,0],[
9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_220:=[[-90.572282316,-
147.276408557,48.001568019],[0.765187367,0.084344336,0.613792262,0.175023957],[0,0,0,0]
,[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_230:=[[-89.654925152,-
143.866974459,50.385594897],[0.765187367,0.084344336,0.613792262,0.175023957],[0,0,0,0]
,[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_240:=[[-88.595143059,-
140.653443823,53.122081652],[0.765187367,0.084344336,0.613792262,0.175023957],[0,0,0,0]
,[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_250:=[[-88.106439826,-
139.210999016,54.570424116],[0.765187367,0.084344336,0.613792262,0.175023957],[0,0,0,0]
,[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_260:=[[-87.513380143,-
137.544232367,56.425600814],[0.765187367,0.084344336,0.613792262,0.175023957],[0,0,0,0]
,[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_270:=[[-87.113096851,-
136.522883714,57.657843745],[0.765187367,0.084344336,0.613792262,0.175023957],[0,0,0,0]
,[9E9,9E9,9E9,9E9,9E9,9E9]];
	CONST robtarget Target_270_2:=[[-157.171031348,-
152.315524227,51.14030457],[0.662903094,0.200200432,0.628834974,0.353618227],[0,0,0,0],[
9E9,9E9,9E9,9E9,9E9,9E9]];
```

```
 /* Path_10 is the path along the inner or concave part of the blisk. It consists of a series of
targets to move across the blisk.*/
        PROC Path_10()
            MoveAbsJ [[-126.2,65.6,-
14.6,169.4,53.4,102.4],[9E9,9E9,9E9,9E9,9E9,9E9]]\NoEOffs,v1000,z50,tool0;
 /* Workobject_1 is the given frame of reference for all of the targets and movements. It is based
around the blisk such that if the blisk is moved the code can follow it.*/
 /* tool_ball is the tool made from the end of arm tooling on the robot.*/
 /* v100 is a speed metric with a TCP(tool center point) speed of 100mm/s.*/

            MoveJ Target_11,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_11_2,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_11_2_2,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_12,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_13,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_14,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_40,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_50,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_60,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_70,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_80,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_90,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_100,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_110,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_120,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_130,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_130_2,v100,fine,Tool_T\WObj:=Workobject_1;
            MoveJ Target_11,v100,fine,tool_ball\WObj:=Workobject_1;
        ENDPROC
 /* Path_20 is the path along the outer or convex part of the blisk. */
        PROC Path_20()
            MoveJ Target_137,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_138,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_139,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_140,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_150,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_160,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_170,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_180,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_190,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_210,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_220,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_230,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_240,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_250,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_260,v100,fine,tool_ball\WObj:=Workobject_1;
```

```
            MoveJ Target_270,v100,fine,tool_ball\WObj:=Workobject_1;
            MoveJ Target_270_2,v100,fine,Tool_T\WObj:=Workobject_1;
            MoveAbsJ [[-126.2,65.6,-
14.6,169.4,53.4,102.4],[9E9,9E9,9E9,9E9,9E9,9E9]]\NoEOffs,v1000,z50,tool0;
        ENDPROC
/* This is the main loop which runs both Path_10 and Path_20 when given the go ahead from the
Arduino. This command is meant to be run in a loop during operation.*/

        PROC main()
                SetDO D652_10_DO7, 1;
                WaitDI D652_10_DI2, 1;
                SetDO D652_10_DO7, 0;
                Path_10;
                Path_20;
        ENDPROC
ENDMODULE
```

MATLAB Code:

```
function varargout = StartScreen(varargin)
% STARTSCREEN MATLAB code for StartScreen.fig
%      STARTSCREEN, by itself, creates a new STARTSCREEN or raises the
existing
%      singleton*.
%
%      H = STARTSCREEN returns the handle to a new STARTSCREEN or the handle
to
%      the existing singleton*.
%
%      STARTSCREEN('CALLBACK',hObject,eventData,handles,...) calls the local
%      function named CALLBACK in STARTSCREEN.M with the given input
arguments.
%
%      STARTSCREEN('Property','Value',...) creates a new STARTSCREEN or
raises the
%      existing singleton*.  Starting from the left, property value pairs are
%      applied to the GUI before StartScreen_OpeningFcn gets called.  An
%      unrecognized property name or invalid value makes property application
%      stop.  All inputs are passed to StartScreen_OpeningFcn via varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help StartScreen

% Last Modified by GUIDE v2.5 10-Jul-2015 05:35:03

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
```

```
                   'gui_OpeningFcn', @StartScreen_OpeningFcn, ...
                   'gui_OutputFcn',  @StartScreen_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end

% End initialization code - DO NOT EDIT


% --- Executes just before StartScreen is made visible.
function StartScreen_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to StartScreen (see VARARGIN)

% Choose default command line output for StartScreen
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes StartScreen wait for user response (see UIRESUME)
% uiwait(handles.figure1);


% --- Outputs from this function are returned to the command line.
function varargout = StartScreen_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Takes in the three inputs from the start screen
a = str2double(get(handles.edit2,'String'));
b = str2double(get(handles.edit1,'String'));
```

```
c = str2double(get(handles.edit3,'String'));
% Closes the screen after the ok button is pressed with a little delay
close all;
% Begins the test code.
Final(a,b,c);


function edit1_Callback(hObject, eventdata, handles)
% hObject    handle to edit1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit1 as text
%        str2double(get(hObject,'String')) returns contents of edit1 as a
double

% --- Executes during object creation, after setting all properties.
function edit1_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function edit2_Callback(hObject, eventdata, handles)
% hObject    handle to edit2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit2 as text
%        str2double(get(hObject,'String')) returns contents of edit2 as a
double


% --- Executes during object creation, after setting all properties.
function edit2_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

```
% --- Executes on button press in Min.
function Min_Callback(hObject, eventdata, handles)
% hObject    handle to Min (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of Min


% --- Executes on button press in Max.
function Max_Callback(hObject, eventdata, handles)
% hObject    handle to Max (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of Max



function edit3_Callback(hObject, eventdata, handles)
% hObject    handle to edit3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit3 as text
%        str2double(get(hObject,'String')) returns contents of edit3 as a
double


% --- Executes during object creation, after setting all properties.
function edit3_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% Function for running the comprehensive test from the start screen
% Takes in three parameters the test number to save the data, the
% root num which is how many tests to perform or the number of blades
% inspected times 2 and whether using the min or max ball gauge.
function [] = Final(testNum, rootNum, Min)
% Creates the blank  three column array for displaying the results
b = zeros(rootNum, 3);
% Creates the first column of the array which gives the blade number
corresponding to each test. In the format 11 22 33 … . Is limited to 39 39
after which it goes back to 11 corresponding to inspecting a full blisk.
for i = 1:rootNum,
```

```
    x = mod(i,78);
    x = x - .5 * x ;
    x = round(x);
    if x == 0
       x = 39;
    end
    b(i,1) = x;
end

% Serial connection setup to the Arduino
s = serial('com8');
set(s, 'DataBits', 8);
set(s, 'StopBits', 1);
set(s, 'BaudRate', 9600);
set(s, 'Parity', 'none');

% Code to run the camera twice for every blade root.
% The Arduino side outputs a periodic 'b' character pulse
% and an 'a' to trigger the MATLAB code.
% Once the MATLAB code is triggered then the get scenes code
% runs twice during blade inspection and then awaits until
% another 'a' is sent. Still a bit buggy.
fopen(s);
tz = 2;
for i = 1:rootNum,
    a = 'b';
    if (tz == 2)
        while (a ~= 'a')
            a = fread(s,1,'uchar');
            tz = 0;
        end
    end
    getScenes(testNum,i);
    fscanf(s);
    tz = tz + 1;
end
fclose(s);

% Code for performing the inspection for each half of the blade root.
for i = 1:rootNum,
    % Code for changing to the directory where the data was stored
    % Defined in get scenes
    x = strcat('Test', num2str(testNum));
    % Should change y to correspond to where user wants to save data.
    y = 'C:\Users\William\Documents\MATLAB\Data\';
    z = strcat('Root',  num2str(i));
    d = strcat( y , x);
    e = strcat( d, '\');
    f = strcat( e, z);
    cd(f);
    % Copies the images in the library file which serve as example passes
    % Should change to correspond to where user has their library saved.
    copyfile('C:\Users\William\Documents\MATLAB\Data\Library\*',f);
    % Code to define which sample pass to compare each image to.
    % If a min inspection alternates between mode 3 and 4 which is the min
    % passing value for concave and convex respectively.
    % If a max inspection alternates between mode 1 and 2 which is the max
```

```
    % passing value for concave and convex respectively.
    % The mode values and their corresponding images are defined in
    % Test_Multiple().
    if(Min == 1)
        Mode = mod((i-1), 2) + 3;
    else
        Mode = mod((i-1), 2) + 1;
    end
    % The inspection done on one video from a root fillet.
    K = Test_Multiple(Mode);
    % Code which inputs the result from the test into the output display
    % The percentage value of % match to the library image stated as a
    % a decimal is the 2nd column.
    % The code then determines if the match% is a pass or fail or not by a
    % threshold in this case .73. It then puts that result in the third
    % column 1 for pass and 0 for fail.
    b(i, 2) = K;
    if(K > .73)
        A = 1;
    else
        A = 0;
    end
    b(i, 3) = A;
end
    % Moves the data from the test to be put in the display table
bd = b;
    % Column headings
cnames = {'Blade Number','% Match Example','Pass 1/Fail 0'};
    % Row headings for first two rows
rnames = {'Concave', 'Convex'};
    % Sets up table
t = uitable('Position', [0 250 375 175], 'Data', bd, ...
'ColumnName', cnames, 'RowName',rnames);
    % Displays table
disp(t);


% Calls VLC media player and gets the scene values. Assumes the test
% and root directories don't already exist.
function [] = getScenes( testnum, rootnum)
% Sets up the directories for the test creating both the test folder
% and the corresponding root folders. Test num is the testing number
% session where root number is the value for each individual test.
% Should change b to correspond to where user wants to save data.
a = strcat('Test', num2str(testnum));
b = 'C:\Users\William\Documents\MATLAB\Data\';
c = strcat('Root',  num2str(rootnum));
% Some string attachment to have the Test# and root# directories properly
% setup to cd to them.
d = strcat( b , a);
e = strcat( d, '\');
f = strcat( e, c);
% These files are to commandline access the vlc media player. They
% should be changed to the location of the users vlc media player.
% run time parameter indicates length of record currently set to 6.
g = 'cd \Program Files (x86)\VideoLAN\VLC\ & vlc dshow:// vdev="Default" --
video-filter=scene --scene-path="';
```

```
h = '" --run-time 6 --start-time 0 vlc://quit';
i = strcat( g, f);
j = strcat( i, h);
% Goes to a test number and makes the directory where the test is if
% it doesn't already exist. Needs to check so that it only creates the test
% directory once.

cd(b) ;
if ~exist(a, 'dir')
   mkdir(a);
 end
cd(a);
% Makes the root directory
mkdir(c);
cd(c);
% rootnum naming convention is the number of the root
% divided by two rounded up
% The root num being odd indicates convex and
% even indicates concave
% There are 78 total roots for a full concave or covex test
% Runs VLC media player using the defined command line code.
dos(j);

end

function [K] = Test_Multiple(Mode)
% Code to compare multiple images in a directory to a library image.
% Mode is used to indicate which library image to compare the data to.
% Code Assumes MATLAB is currently in the directory you want to test.
% The 'final' code pre moves to the correct directories when using
% this code.
% Code used to find out total images in the directory
imagefiles = dir(pwd);
nfiles = length(imagefiles)-1;
x = 0;
% Presets an array to store the results from the test
J = zeros(nfiles,1,'double');
% Used to determine which library image to compare to based
% on chosen mode inputted
if(Mode == 1)
    Pass = 'A.jpg';
elseif(Mode == 2)
    Pass = 'B.jpg';
elseif(Mode == 3)
    Pass = 'C.jpg';
else
    Pass = 'D.jpg';
end
% Loop used for each image in the directory
for i=1:nfiles
filename = imagefiles(i).name;
% For each image the following tests are undergone with the (0,0)
% having the test with the images exactly lined up and the rest
% being offsets. The bytes being > 0 is to only run the test when
% an image is captured in case of dropped frames
    if imagefiles(i).bytes > 0
            A = Test(filename, Pass, 110, 110, -1, 1);
```

```
            B = Test(filename, Pass, 110, 110, 0, 1);
            C = Test(filename, Pass, 110, 110, 1, 1);
            D = Test(filename, Pass, 110, 110, -1, 0);
            E = Test(filename, Pass, 110, 110, 0, 0);
            F = Test(filename, Pass, 110, 110, 1, 0);
            G = Test(filename, Pass, 110, 110, -1, -1);
            H = Test(filename, Pass, 110, 110, 0, -1);
            I = Test(filename, Pass, 110, 110, 1, -1);
            % This compares and finds the best match out of the
            % offsets which is the value for that image
            J(i) = max([A,B,C,D,E,F,G,H,I]);
    else
        % Counter for the amount of dropped frames so they aren't
        % included in the calculated total.
        x = x + 1;
    end
end
% The total percent for a given directory taking the average of all the
% percentages. Also subtracts the dropped frames from the image total.
K = sum(J)/(nfiles - x);
end

function [ D ] = Test( image1, image2, X, Y, offsetX, offsetY)
% This code is used to compare two images and determine their percentage
% of matching tiles.
% The inputs are two different images and the output is the percentage of
% them that matches.
J = Crop_Point(image1, X, Y, offsetX, offsetY);
% The cropping program
K = rgb2gray(J);
% The filter to greyscale
L = im2bw(K, 0.6);
% The filter to binary
L = 2.*L;
L = L - 1;
% Converting the binary matrix from ones and zeros to positive and
% negative ones
M = Crop_Point(image2, X, Y, offsetX, offsetY);
N = rgb2gray(M);
O = im2bw(N, 0.6);
O = 2.*O;
O = O - 1;
% The same process for the other image.
P = 0;
if size(O) == size(L)
    P = L - O;
    P = abs(P);
end
% The images are then compared
D = 1- ((sum(sum(P))/2)/(X * Y));
% The sum of the matrix is added divided by the total possible squares
% based on the cropping.
return ;
end

function  [J] = Crop_Point( image, Xcrop, Ycrop, offsetX, offsetY )
```

```
% This program finds the green point and crops from there.
K = imread(image);
% The image is inputted
[Y,X] = find(K(:, :, 1) <= 160 & K(:, :, 2) >= 130 & K(:, :, 3) <= 60);
% The points with the incredibly high green value.
% Probably could be further tweaked for better results.
 A = (round(mean(X)));
 B = round(mean(Y));
 C = A + offsetX;
 D = B + offsetY;
 % The points that are found with those values are averaged and rounded.
Position = [(C - Xcrop + 20), D, Xcrop, Ycrop];
 % The cropping point is made based on the position from the point and the
 % length of the desired cropping.
J = imcrop(K, Position);
% The image is cropped
% subplot(1,2,1), imshow(K)
% subplot(1,2,2), imshow(J)
% commands to show the original image and the cropping.
end
```

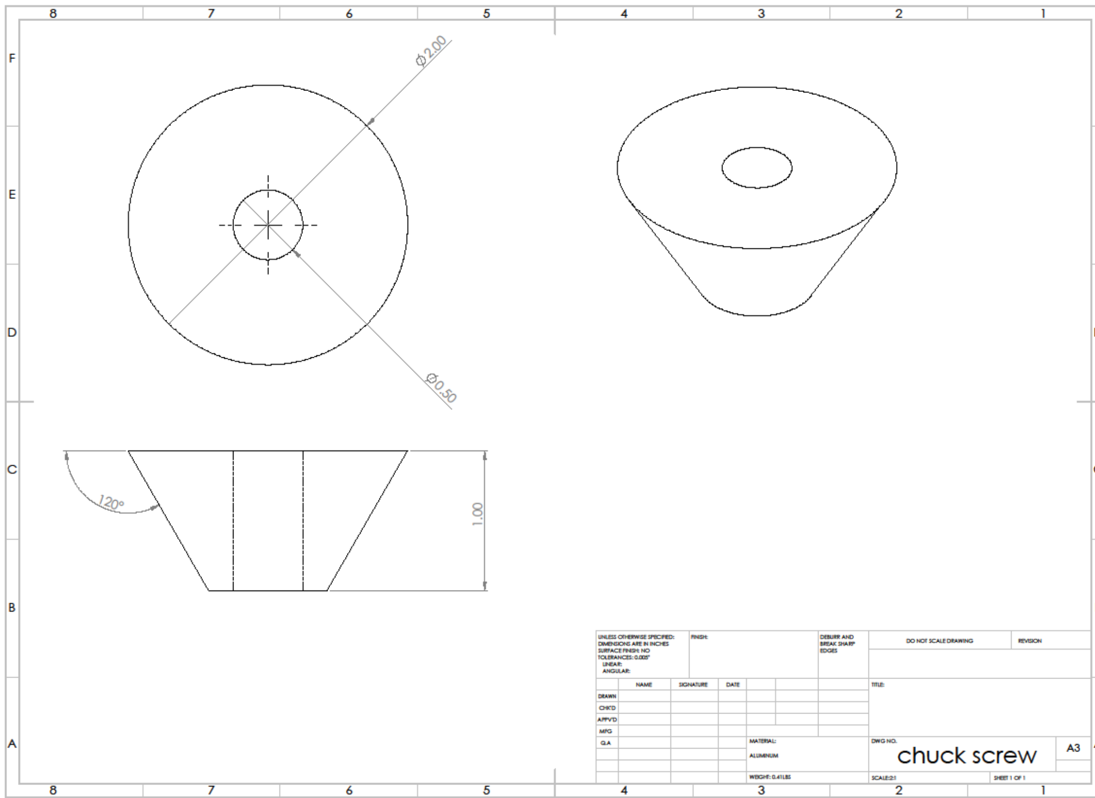
Arduino Code:

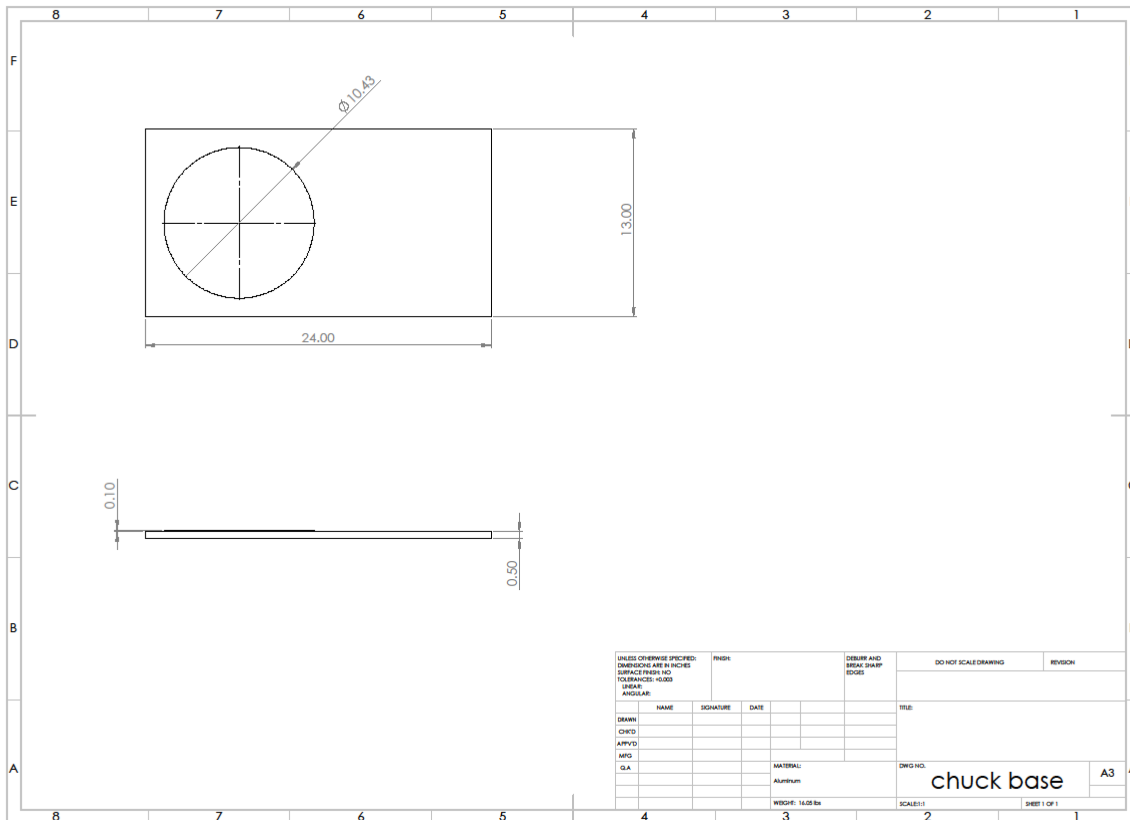
```
/*
Code uses some basic elements from the following code
 VernierPhotogateTimmer (v 2013.12.09)
 Monitors a Vernier Photogate connected to BTD connector.

 Modified by: B. Huang, SparkFun Electronics
 December 9, 2013
 */


#define bufferSize 150  // Sets the size of the circular buffer for storing interrupt data events. Increasing this may cause erratic behavior of the code.
#define DELIM '\t'   // this is the data delimitter. Default setting is a tab character ("\t")


const int ABBoutputPin = 1;    //the pin used to send an output value to the abb robot
const int photogatePin = 2; // default pin if plugged into Digital 1 on SparkFun Vernier Shield
const int stepPin = 11; // pin used to send pulses to the stepper motor
const int ABBinputPin = 9; // the pin used to get an input value from the abb robot
const int numSteps = 10;// the number of steps used at “fast” speed in between blisks
int button = HIGH;

/* These variables are all accessed and modified by the interrupt handler "PhotogateEvent"
```

```
Variables used by the Interrupt handler must be defined as volatile. */
volatile int photogate = LOW;
volatile int dataIndex = 0;
volatile int pace = 0;
 // First state (0) meant to represent the motor stopped waiting for the go
 // ahead from robot studio
 // Second state (1) meant to represent the motor running waiting for the
 // stop from the photogate

void setup()
{
 attachInterrupt(0, photogateEvent, CHANGE); // photogate_event
 pinMode(ABBoutputPin, OUTPUT);
 pinMode(stepPin, OUTPUT);
 pinMode(photogatePin , INPUT);
 pinMode(ABBinputPin, INPUT);
 digitalWrite(ABBoutputPin, LOW);
 Serial.begin(9600);        // set up Serial library at 9600 bps
};// end of setup

void loop ()
{
 if (dataIndex == 0){
 //Motor doesn't move
 digitalWrite(stepPin,LOW);
 delay(10);
 //Command to pulse to MATLAB
 Serial.println('b');
//Code to start moving if the go ahead from abb
// Done by polling instead of isr because the photogate shield
// only allows the use of one isr although acceptable since this is less
// time sensitive than the photogate.
 button = digitalRead(ABBinputPin);
 if(button == HIGH){
  dataIndex = 1;
  digitalWrite(ABBoutputPin, LOW);
//Code that the tells the abb it is moving
 }
 }
 else{
 //Motor moves
 if(pace == 0){
 for(int x = 0; x < numSteps; x++){
// The fast moving speed done for the given numSteps
 digitalWrite(stepPin,HIGH);
 delay(50);
 digitalWrite(stepPin,LOW);
 delay(50);
```

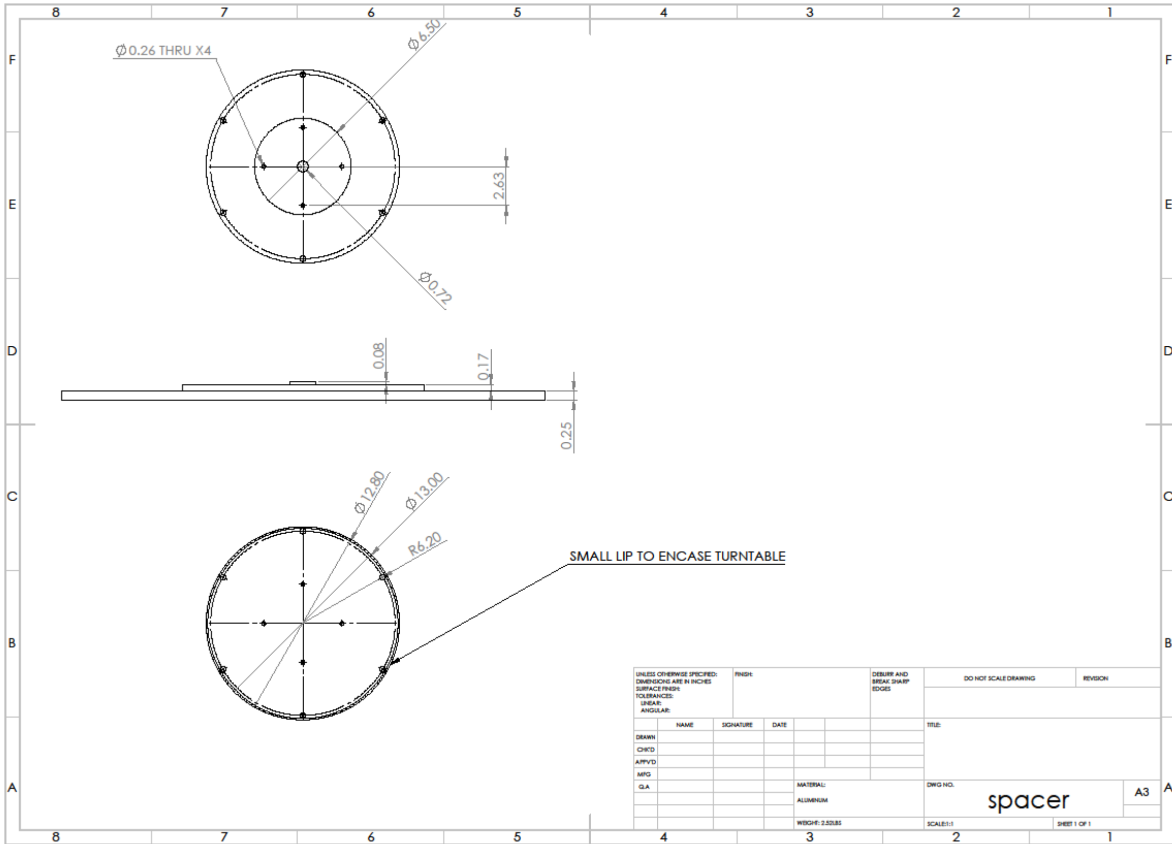
```
 //Command to pulse to MATLAB
 Serial.println('b');
 }
 pace = 1;
 }
// The slow moving speed used to precisely move into place.
 digitalWrite(stepPin,HIGH);
 delay(100);
 digitalWrite(stepPin,LOW);
 delay(100);
 //Command to pulse to MATLAB
 Serial.println('b');
}
} // end of loop



/**************************************************
 * photogateEvent()
 *
 * Interrupt service routine. Handles capturing
 * the time and saving this to memory when the
 * photogate issues an interrupt on pin 2.
 *
 * As it is currently written, the photogate
 * will only work on Digital Port 1.
 **************************************************/
void photogateEvent()
{

 photogate = digitalRead(photogatePin);
   if (photogate == 1)  // used when photogate becomes un tripped
   {
     dataIndex = 0;
     digitalWrite(ABBoutputPin, HIGH);  // tells the abb it is done moving
     pace = 0; //tells the turntable to move “fast” mode
 //Command to indicate to start video in MATLAB
     Serial.println('a');
   }
}
```

Part Drawings:

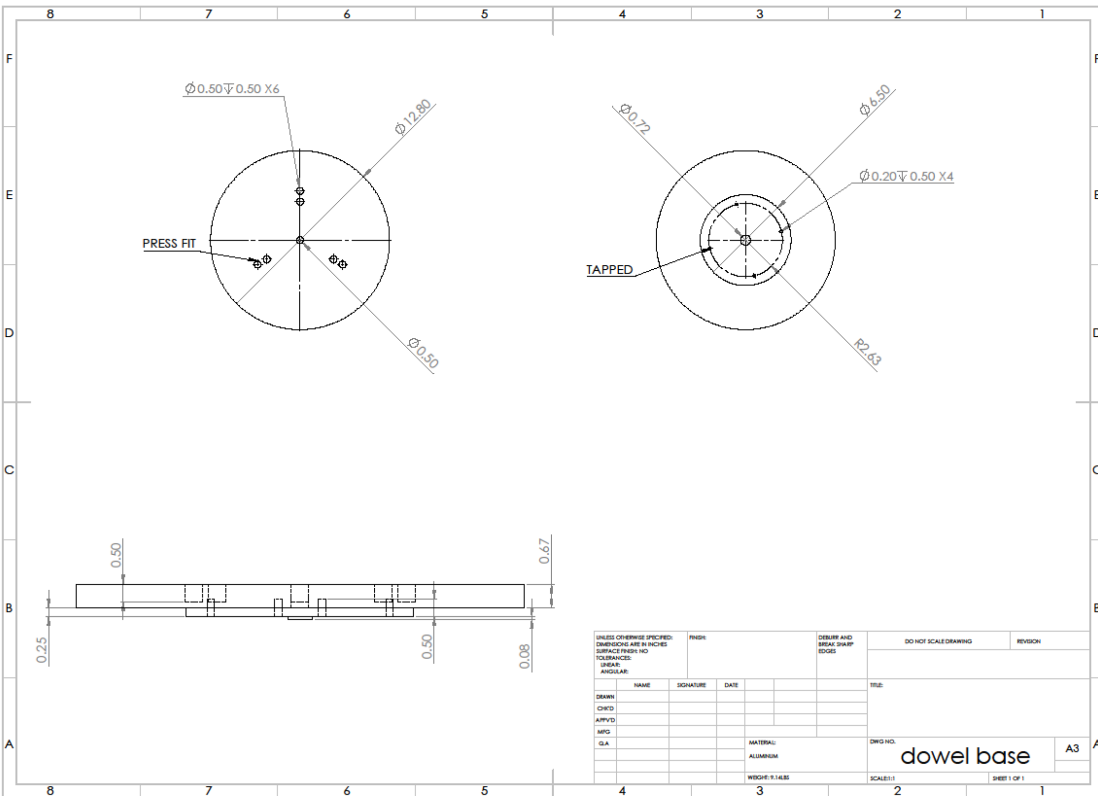




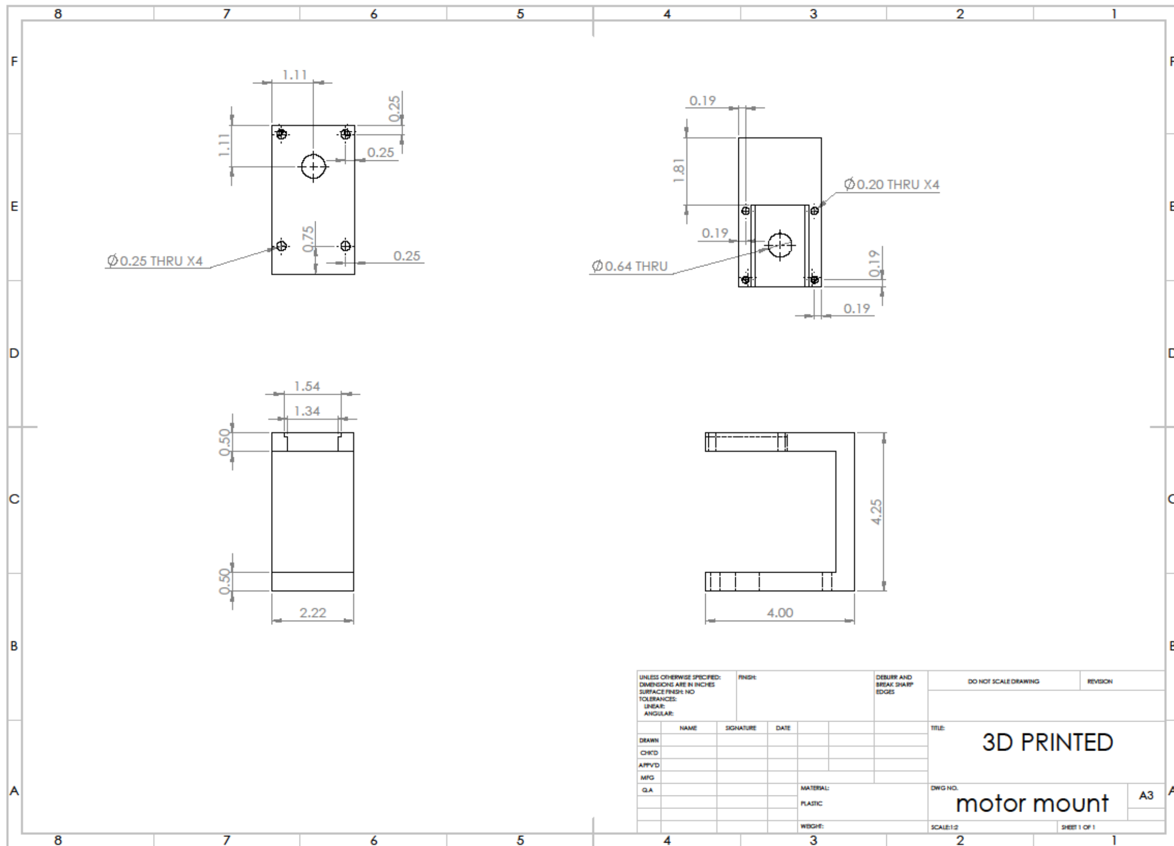
s:

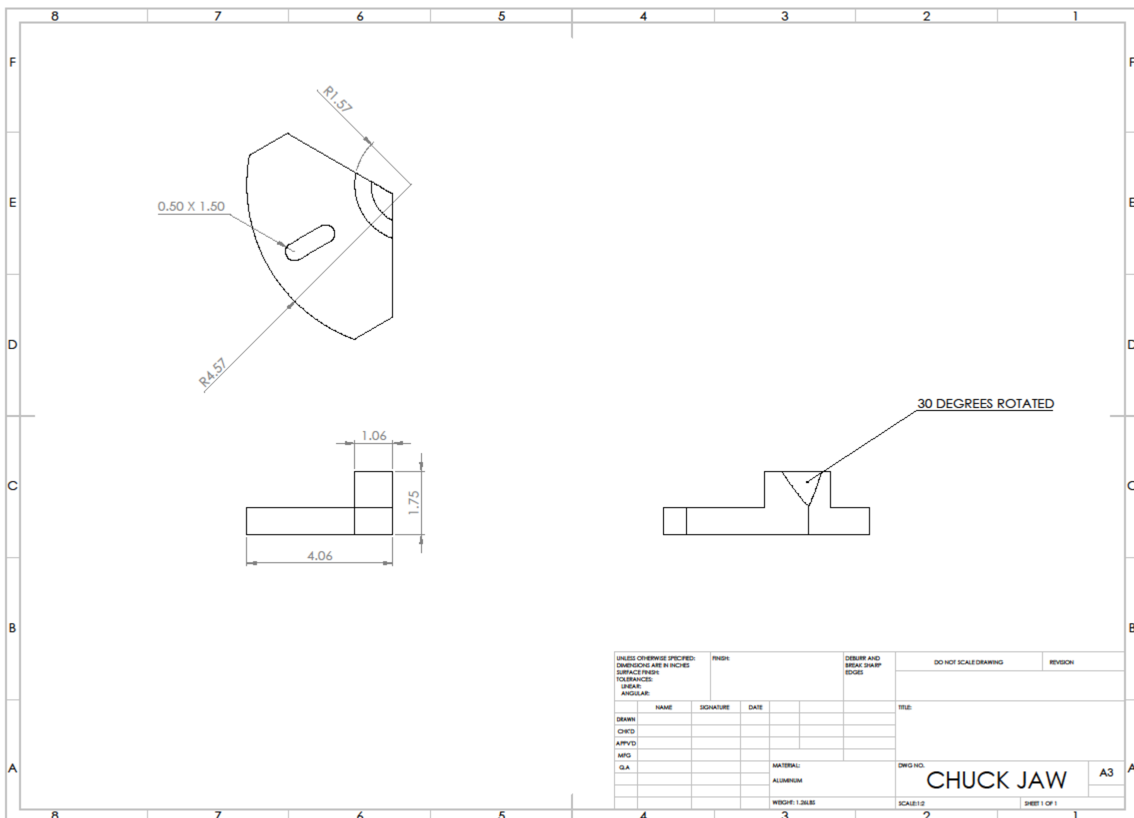


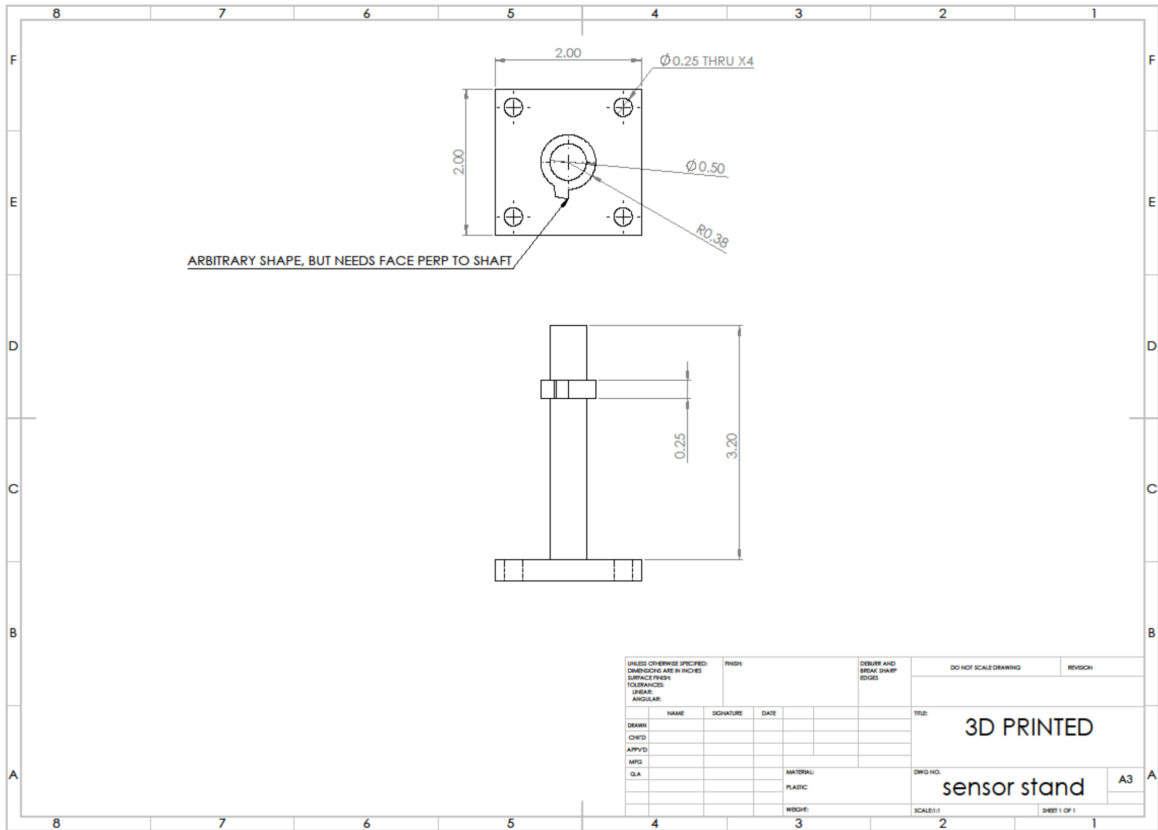
Ø0.26 THRU X4
Ø6.50
2.63
Ø0.72
0.08
0.17
0.25
Ø12.80
Ø13.00
R6.20
SMALL LIP TO ENCASE TURNTABLE
MATERIAL:
ALUMINUM
DWG NO.
spacer
A3



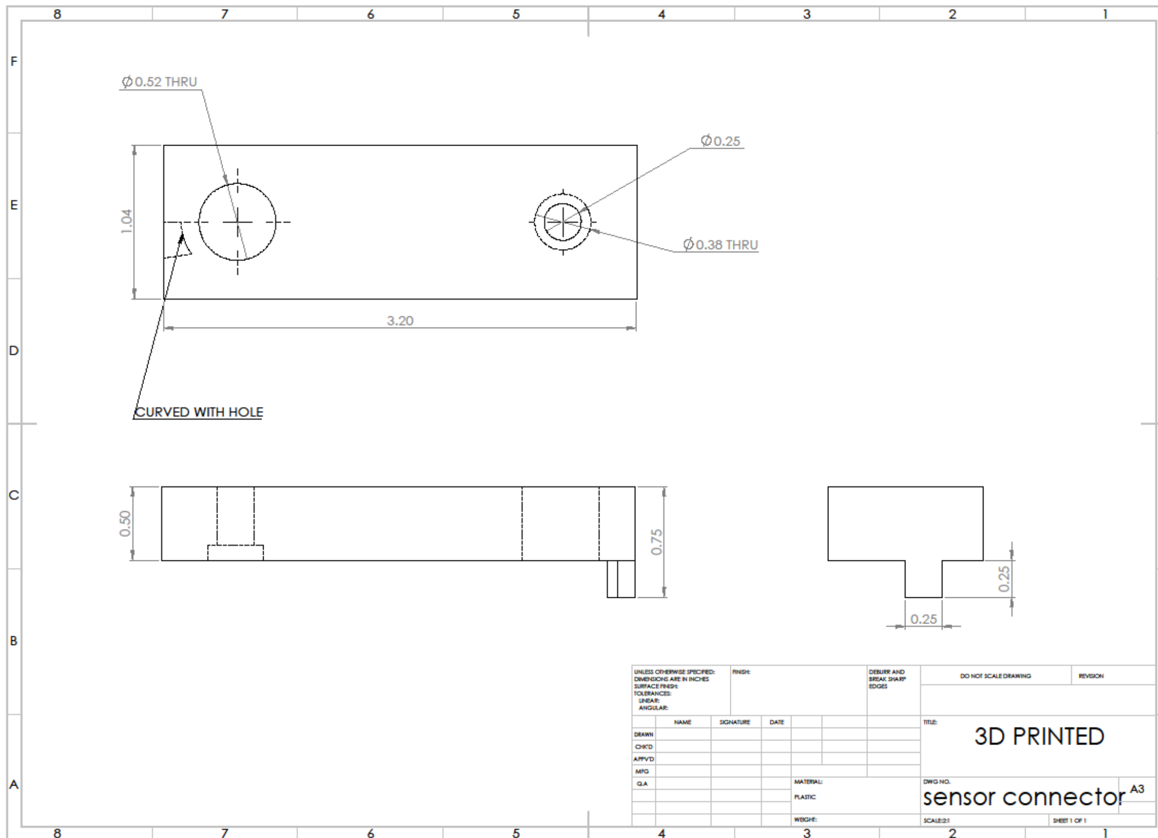
Ø0.50 ↧ 0.50 X6
Ø12.80
PRESS FIT
Ø0.50
Ø0.72
Ø6.50
Ø0.20 ↧ 0.50 X4
TAPPED
R2.63
0.50
0.67
0.25
0.50
0.08
MATERIAL:
ALUMINUM
DWG NO.
dowel base
A3


1.11
0.25
1.11
0.25
Ø0.25 THRU X4
0.75
0.25
0.19
1.81
Ø0.20 THRU X4
0.19
Ø0.64 THRU
0.19
0.19
1.54
1.34
0.50
0.50
2.22
4.25
4.00
UNLESS OTHERWISE SPECIFIED:
DIMENSIONS ARE IN INCHES
FINISH:
DEBURR AND BREAK SHARP EDGES
DO NOT SCALE DRAWING
REVISION
TITLE:
3D PRINTED
MATERIAL:
PLASTIC
DWG NO.
motor mount
A3
SCALE:1:2
SHEET 1 OF 1

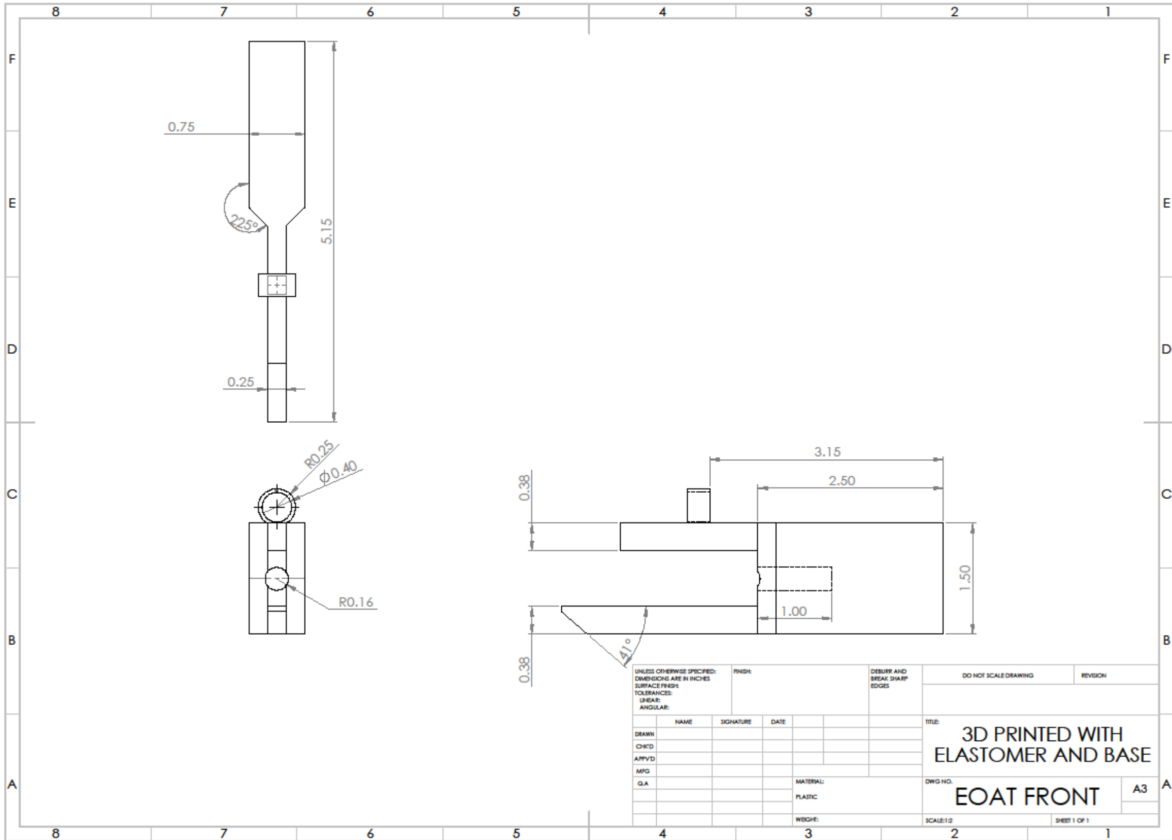

R1.57
0.50 X 1.50
R4.57
1.06
1.75
4.06
30 DEGREES ROTATED
UNLESS OTHERWISE SPECIFIED:
DIMENSIONS ARE IN INCHES
FINISH:
DEBURR AND BREAK SHARP EDGES
DO NOT SCALE DRAWING
REVISION
TITLE:
MATERIAL:
ALUMINUM
DWG NO.
CHUCK JAW
A3
SCALE:1:2
SHEET 1 OF 1

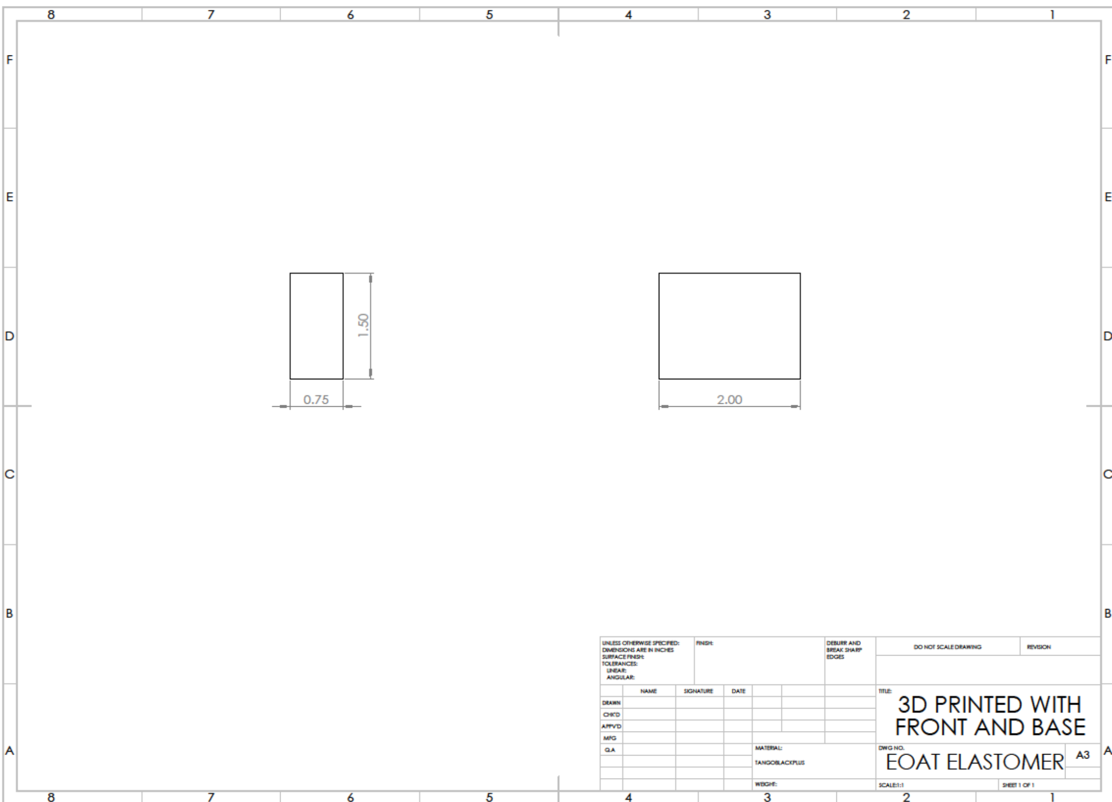


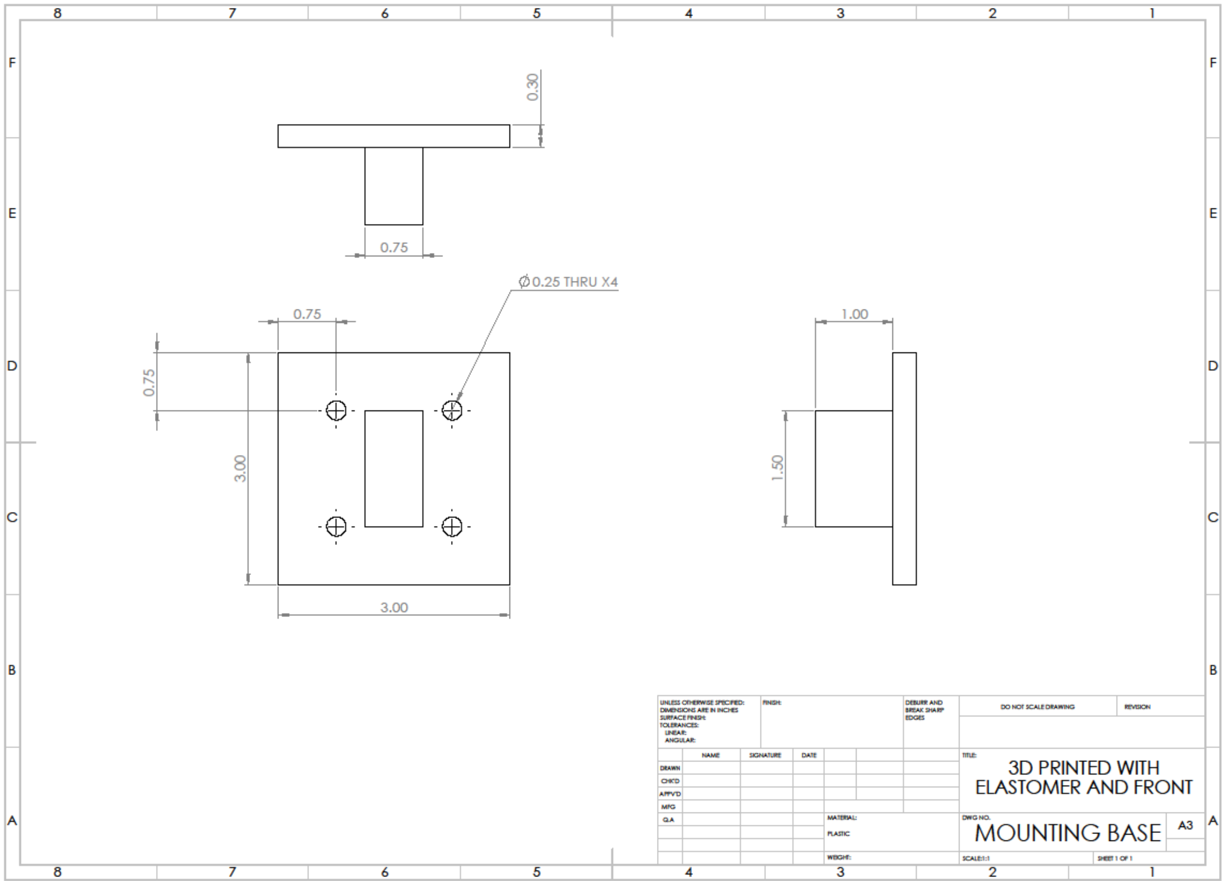
2.00
∅0.25 THRU X4
2.00
∅0.50
R0.38
ARBITRARY SHAPE, BUT NEEDS FACE PERP TO SHAFT
0.25
3.20
3D PRINTED
sensor stand
A3
PLASTIC



∅0.52 THRU
∅0.25
1.04
∅0.38 THRU
3.20
CURVED WITH HOLE
0.50
0.75
0.25
0.25
3D PRINTED
sensor connector
A3
PLASTIC


0.75
225°
5.15
0.25
R0.25
Ø0.40
R0.16
3.15
2.50
0.38
1.50
1.00
41°
0.38
3D PRINTED WITH
ELASTOMER AND BASE
EOAT FRONT
A3
PLASTIC

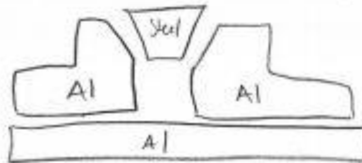

1.50
0.75
2.00
3D PRINTED WITH
FRONT AND BASE
EOAT ELASTOMER
A3
TANGOBLACKPLUS


0.30
0.75
Ø0.25 THRU X4
0.75
1.00
0.75
3.00
1.50
3.00
3D PRINTED WITH
ELASTOMER AND FRONT
MOUNTING BASE
A3
PLASTIC

Written Analysis:

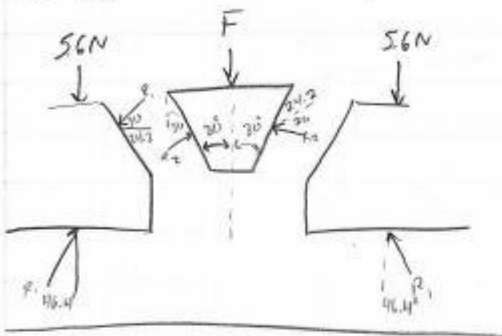
1.26 lbs = 5.6 N

$F_f = \mu N$



$\mu_{Al\text{-}Al} = 1.05 \rightarrow \angle_f = 46.4^\circ$

$\mu_{Al\text{-}Steel} = 0.45 \rightarrow \angle_f = 24.2^\circ$



$\Sigma F_y = 0$

$R_1 \sin 46.4 = R_2 \cos 54.2$

$R_1 (0.72) = R_2 (0.58)$

$R_1 = 0.81 R_2$

$\Sigma F_x = 0$

$R_1 \cos 46.4 = R_2 \sin 54.2 + 5.6$

$(0.81 R_2) \cos 46.4 = R_2 \sin 54.2 + 5.6$

$(0.81 R_2)(0.68) = R_2 (0.81) + 5.6$

$0.55 R_2 = 0.81 R_2 + 5.6$

$-0.26 R_2 = 5.6$

$R_2 = 21.5 \text{ N}$

$\Sigma F_y = 0$

$F = 2 R_2 \sin 54.2$

$= 2(21.5)(0.81)$

$\boxed{F = 34.83 \text{N}}$

$D = 0.5''$

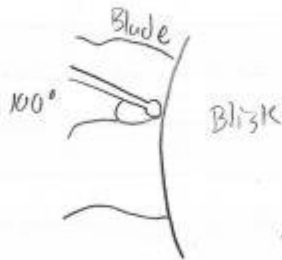
$\mu_f = 0.2$

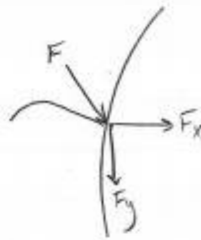
$F = 34.83N$

$T = \mu_f D F$

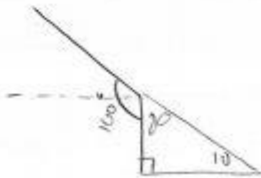
$T = 0.2(0.5)(34.83)$

$= \boxed{3.48 \text{ in/lbs}}$





$F_{measured} = 0.25 \text{ lbs} = 1.11 N$



$F_y = 1.11 \sin(10) = 0.19 N$

$F_x = 1.11 \cos(10) = 1.09 N$

Chain Tension

15 teeth   4:1   60 teeth

1 step/sec   200 steps/rev   200s/60s = 3.33 min

1 step = 1.8°

$D_1 = 1.61''$ $C_1 = \pi D_1 = 5.06''$ $\omega = \frac{\theta}{t}$

$D_2 = 7.01''$ $C_2 = \pi D_2 = 22.02''$

Torque = 270 oz-in

3.33(4) = 13.32 min/rev of driven gear

$G_1 = 0.3$ rpm

$G_2 = 0.07$ rpm

v = chain velocity (in/s)

$v_1 = \frac{D_1(\pi)(0.3)}{60} = 0.025$ in/s