Design and Control of a 3RPS-Stewart Platform for Precision Ball Balancing Using PID and Computer Vision

Jose Ismael Ramirez
Department of Mechanical Engineering
California State Polytechnic University, Pomona,
Pomona, USA
jiramirez@cpp.edu

Ethan D. Lendo
Department of Mechanical Engineering
California State Polytechnic University, Pomona,
Pomona, USA
edlendo@cpp.edu

Abstract*— *This paper presents the design and implementation of a Stewart platform, a 3-DOF parallel manipulator with three linear actuators, combining revolute, prismatic, and spherical joints. The project aims to dynamically stabilize a ball on the platform by precisely regulating its tilt angles using a combination of mechanical design, mathematical modeling, and digital control system implementation.

The core of the control system is a proportional-integral-derivative (PID) algorithm, optimized to maintain the ball's position at a designated setpoint on the platform. Real-time feedback is provided by a computer vision camera sensor, operating at 60 frames per second, which tracks the ball's position and sends coordinates to an Arduino Uno R4 microcontroller. Corrective actions calculated by the controller achieved a settling time of 8.6 seconds, demonstrating efficient stabilization.

The system's nonlinear dynamics were modeled using the Euler-Lagrange method to derive stability criteria and inform the PID design. Continuous-time stability analysis revealed marginal stability, necessitating precise parameter tuning. A zero-order hold method with a 0.2-second sampling time enabled discretization for microcontroller implementation while maintaining performance.

Inverse kinematics translated control commands into stepper motor actuator positions, allowing precise control of the platform's orientation. Iterative prototyping addressed challenges like joint friction, resolution constraints, and motor limitations. The final design featured a 1.5-foot diameter platform constructed of 3D-printed components, and an aluminum platform, demonstrating a balance between manufacturability and functionality.

This project underscores the integration of low-cost components and open-source tools to develop scalable control systems. Potential applications include robotics, automation, and precision handling. Future work aims to incorporate faster microcontrollers for improved response and expand functionality to handle more complex dynamics.

Keywords—Stewart Platform, parallel manipulation, computer vision tracking, dynamic stabilization, motion control system

I. INTRODUCTION

The Stewart platform is a type of parallel manipulator widely recognized for its precision and versatility in applications such as motion simulation, robotics, and automation. This project aims to develop a simplified Stewart platform configured for three degrees of freedom (3-DOF) to dynamically balance a ball. The system combines mechanical design, mathematical modeling, and control system implementation to achieve precise motion control and stability.

The platform's design focuses on cost-effectiveness and scalability by integrating low-cost components such as an Arduino microcontroller and a Pixy2 camera for computer vision. A proportional-integral-derivative (PID) control algorithm uses real-time feedback to adjust the platform's tilt angles and stabilize the ball. The system demonstrates how advanced mechatronic concepts can be implemented with accessible tools, making it a valuable case study for robotics and educational research.

The following subsections detail the system's key components, including position detection and mechanical construction, and outline their contributions to the overall design and functionality.

A. Position Detection

Accurate position detection is vital for the success of the ball-balancing system. The primary requirement of the position sensor is to provide real-time feedback with sufficient resolution and speed to track the ball's movement. After evaluating various alternatives, including piezoelectric sensors and computer vision, the Pixy2 camera sensor was selected due to its onboard image processing capabilities and 60 frames per second speed. This camera allows for effective tracking of the ball's position in the X, Y, and Z axes without overburdening the microcontroller.

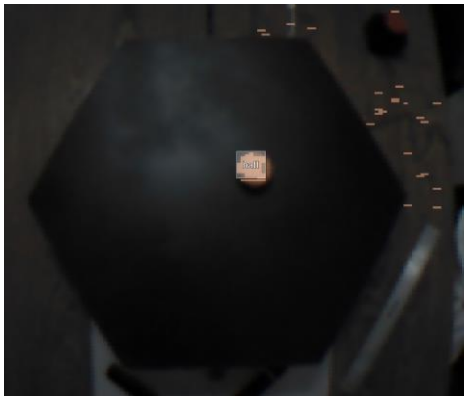


Figure 1: PixyMon Camera Feed

The Pixy2 camera simplifies image processing tasks by filtering pixels, adjusting brightness, and setting signature thresholds. These features reduce computational demands and enable quick and accurate data transfer to the Arduino Uno R4 microcontroller. It simply sends out x, y, and z coordinates to be interpreted by the Arduino. This combination ensures that corrective actions can be calculated and executed in near real-time. In Figure 1 above, the live camera feed can be seen as it looks it the PixyMon software that is used to tune the cameras parameters. Parameters such as pixel filtering, brightness, and signature thresholds can all be modified in the software that is included with the Pixy2 camera.

Alternative approaches, such as using multiple infrared or ultrasonic and piezoelectric sensors, were considered but deemed less effective due to their limited precision in continuous tracking and increased system complexity. The camera's ability to provide detailed positional data proved crucial in achieving the project's accuracy and responsiveness goals.

B. Mechanical Construction

The mechanical construction of the platform, referred to as the plant, plays a central role in ensuring stability and precision. The primary requirements of the plant are rigidity, low friction at joints, and compatibility with the chosen actuation methods. A Stewart platform design was chosen for its inherent stability and ability to achieve three degrees of freedom. Compared to alternative designs, such as six-degree-of-freedom (6-DOF) platforms, the simplified 3-DOF configuration reduced cost and complexity while still meeting the project's goals. Looking at Figure 2, the different possible configurations for a Stewart Platform can visualized. We each considered these designs when designing our application.

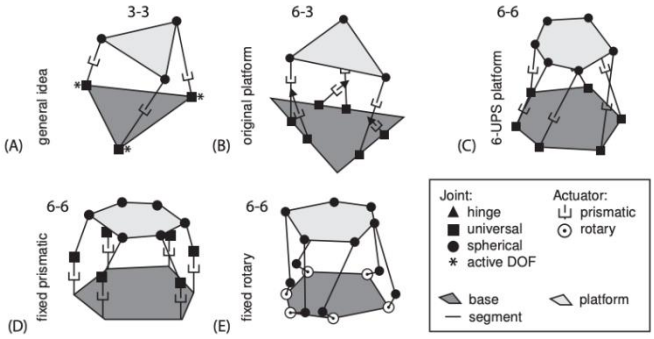


Figure 2: Common Stewart Platform Configurations

The plant incorporates three linear actuators, each powered by Nema 17 stepper motors with integrated encoders for closed-loop control. Bearings were added at moving joints to reduce friction and wear, ensuring smooth operation and prolonged durability. The frame and actuators were constructed using aluminum for its strength-to-weight ratio, while 3D-printed components were used for custom fittings to maintain cost efficiency and manufacturability.

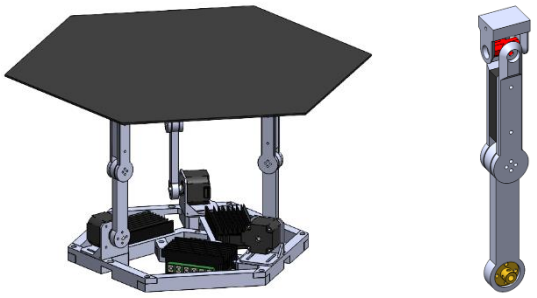


Figure 3: CAD Model of our system

The chosen actuation method relies on revolute, prismatic, and spherical (RPS) joints, which allow precise control of the platform's orientation. The actuators translate control signals into platform tilt adjustments, enabling accurate compensation for ball movement. Alternative configurations, such as 6-DOF platforms or systems with pneumatic actuators, were considered but excluded due to their higher cost and increased system complexity. As shown above in Figure 3, the actuators and entire system CAD model can be seen, showing the actuators and their design.

This design strikes a balance between functionality, affordability, and manufacturability, making it a practical choice for both research and educational purposes.

II. System Design

The design of the Stewart platform emphasizes the integration of mechanical precision, effective actuation, and vision-based feedback to achieve dynamic ball balancing with three degrees of freedom (3-DOF). This section focuses on the platform's mechanical configuration, actuation systems, and vision integration.

The mechanical structure of the platform is constructed primarily from aluminum, chosen for its combination of strength, durability, and lightweight properties. The circular platform has a diameter of 1.5 feet, providing a controlled area for ball movement. The base plate, also made of aluminum, serves as a rigid foundation, ensuring stability and reducing vibrations during operation. Bearings are incorporated at all moving joints to minimize friction and mechanical wear, enhancing long-term performance and reliability.

Actuation is achieved through three linear actuators, each driven by NEMA 17 stepper motors equipped with a 5.18:1 planetary gearbox for enhanced precision and torque. These actuators are connected to the platform and the base plate using revolute, prismatic, and spherical (RPS) joints. The revolute joint enables angular control at the motor, the prismatic joint facilitates linear motion, and the spherical joint allows multi-directional tilting. This arrangement ensures that the platform can achieve precise tilt and height adjustments, as required for maintaining the ball's stability.

The Pixy2 camera is employed as the primary feedback sensor, offering real-time tracking of the ball's position in three dimensions. Mounted directly above the platform, the camera provides an unobstructed view, capturing positional data at 60 frames per second. Onboard processing within the camera extracts the ball's coordinates, which are then transmitted to an Arduino Uno R4 microcontroller for use in the control algorithm. This approach reduces computational demands on the microcontroller and simplifies the system architecture compared to alternative sensor configurations. The final design and linear actuators can be seen below in Figure 4, showing the tall camera

stand which served as the main source of feedback for our system as well as the stepper motors and drivers.

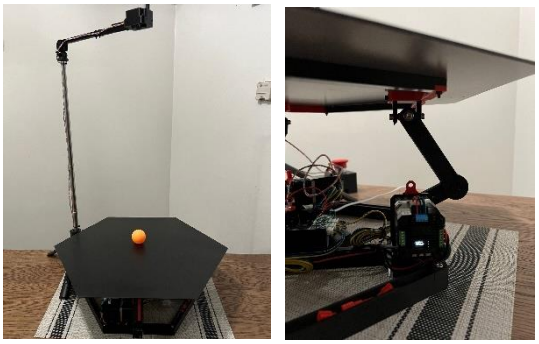


Figure 4: Final Design

The actuation process is driven by the results of inverse kinematics calculations, which determine the necessary adjustments in actuator lengths to achieve the desired platform tilt and height. Stepper motor movements execute these adjustments, translating control signals into precise physical changes in platform orientation. The closed-loop feedback system continuously monitors the ball's position and updates the actuator commands, allowing for rapid compensation of disturbances.

Power is supplied by a 24V system that provides consistent energy to the stepper motors, ensuring smooth and reliable operation. The Arduino Uno R4 coordinates all components, integrating vision feedback with actuation control to maintain dynamic stability. The platform's compact and robust design, combined with efficient vision-based feedback, enables precise motion control while maintaining simplicity and manufacturability.

III. Mathematical Modeling

A. *Inverse Kinematics*

In order to accurately model the dynamic behavior of a mobile platform, it is crucial to establish the relationship between the platform's position and orientation, the joint variables, and its geometric parameters. For platforms like the Stewart platform, two primary approaches are used: inverse kinematics and direct kinematics.

Direct kinematics involves solving a set of nonlinear equations to determine the position and orientation of the manipulator when the lengths of the actuators are known. However, this method can lead to multiple solutions. For example, a servo-controlled Stewart platform generates a system of 18 equations with up to 40 possible solutions [1]. In contrast, inverse kinematics provides a more practical and efficient approach. It simplifies the process by calculating the actuator lengths and motor parameters directly from the desired position or tilt angles of the upper platform. This avoids the complexity of determining the platform's position and orientation based on actuator conditions and operating parameters.

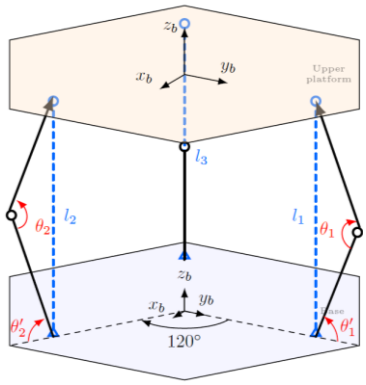


Figure 5: Physical Structure of platform with linear actuators

Given these advantages, inverse kinematics is particularly well-suited for deriving the dynamic model of the platform. By focusing on the variations in distances between the upper platform and the fixed lower platform, as well as the motor parameters, the desired position of the platform can be effectively achieved. This approach ensures precise and stable control in dynamic applications.

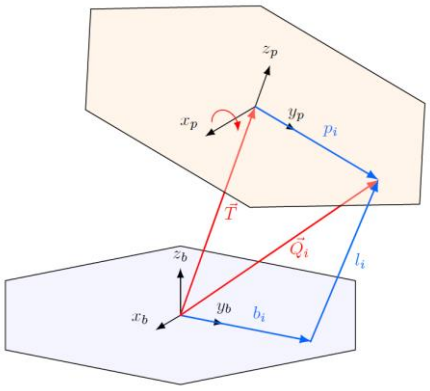


Figure 6: Platform Rotation i_{th} linear actuator

In Figure 6, relative to the center of the platform, the i_{th} vector p_i denotes the location of the spherical joint in the platform and the vector b_i denotes the location of the rotational joint at the motor or base. The vector $\vec{T}$ is the vertical translational reference distance between the base and the platform. The vector l_i denotes the length of the i_{th} actuator, required to tilt the platform based on the roll or pitch angle of the platform. The vector $\overrightarrow{Q_i}$ defines the coordinates of the anchor point p_i relative to the base plate given by,

$$Q_i = \overrightarrow{T_i} + R_b^p \cdot \overrightarrow{P_i} \quad (1)$$

where, $\overrightarrow{P_i}$ is the location of the of the spherical end-effector from the origin of the platform in the reference frame of the platform. $\overrightarrow{B_i}$ is the location of the rotational joint expressed in the reference frame of the base plate.

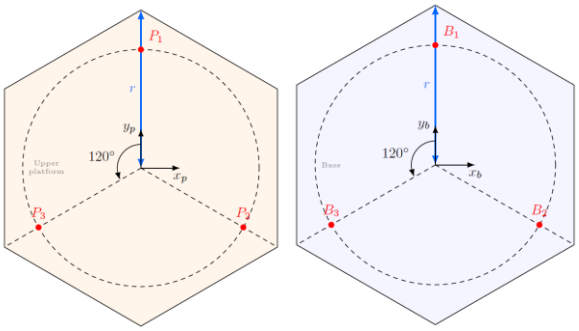


Figure 7: Platform and Base location of end-effectors

The rotational matrix R_b^p, is the rotation matrix, of the platform frame with respect to the base, given by the combination of each rotation R_x (roll angle α), R_y (pitch angle β), and R_z (yaw angle ψ),

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \tag{2}$$

$$R_y = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \tag{3}$$

$$R_z = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

Note that the rotation matrix about the yaw angle is evaluated as the identity matrix since there is no rotation about the z-axis due to design of the system, therefore $\psi = 0$. Thus, the rotational matrix is given by,

$$R_b^p = R_z \cdot R_y \cdot R_x$$

$$R_b^p = \begin{bmatrix} \cos\beta & \sin\alpha \cdot \sin\beta & \sin\beta \cdot \cos\alpha \\ 0 & \cos\alpha & -\sin\alpha \\ -\sin\beta & \cos\beta \cdot \sin\alpha & \cos\alpha \cdot \cos\beta \end{bmatrix} \tag{5}$$

Finally, the length of the i_{th} parallel actuator is found to be,

$$\vec{l_\iota} = \vec{T_\iota} + R_b^p \cdot \vec{P_\iota} - \vec{B_\iota} \tag{6}$$

However, the length of the parallel manipulator is achieved by the rotational angle of the stepper motors as denoted in figure 8,

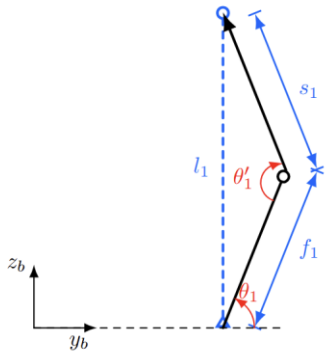


Figure 8: Linear RPS joint

By applying, the law of cosines and the fixed lengths of the parallel manipulator, results in the following equation that relates the angle of the motor θ_i and length of the parallel manipulator l_i.

$$f_i^2 = s_i^2 + l_i^2 - 2s_i l_i \cos\theta_i \tag{7}$$

Finally, the angle θ_i of the i_{th} motor based on the platforms rotation can be determined by,

$$\theta_i = \cos^{-1}\left(\frac{l_i^2 + s_i^2 - f_i^2}{2s_i l_i}\right) \tag{8}$$

B. Dynamic Modeling

The Stewart platform plant model serves as the foundation for analyzing the dynamics of a ball-and-plate control system. This system is a common benchmark in control theory and robotics due to its complexity and multi-variable nature. It involves balancing a ball on a rigid plate by adjusting the plate's tilt angles through actuators. The dynamic model provides a mathematical framework for designing controllers, simulating system behavior, and predicting performance under various conditions. The following section outlines the assumptions, energy analysis, and derivation of the plant model, culminating in its transfer functions.

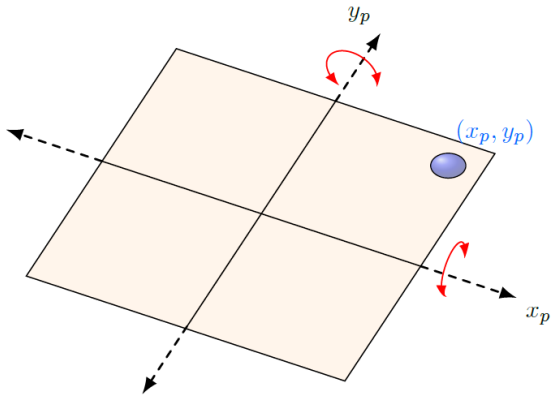


Figure 9: Ball in plane with pitch and roll rotation

1) Assumptions:

The ball and plate system analyzed under the Stewart platform model operates under the following assumptions:

1. The ball rolls on the platform without slipping.
2. The ball is a solid, symmetric, and homogeneous sphere.
3. Friction between the ball and the plate is negligible.
4. The ball maintains continuous contact with the plate.
5. The plate is rigid and homogeneous.

2) Energy Analysis

The Lagrangian formulation simplifies the analysis of our system by expressing dynamics in terms of kinetic and potential energy. For the ball-and-plate system, this approach captures the interaction between the ball's motion and the plate's tilt using generalized coordinates. The resulting equations of

motion form the foundation for control design and stability analysis.

The dynamic behavior of the system is characterized using the Lagrange equation, expressed as:

$$L(t) = \sum T(t) - \sum V(t) \tag{9}$$

Where $L(t)$ is the difference between the kinetic energy T and the potential energy V of the system.

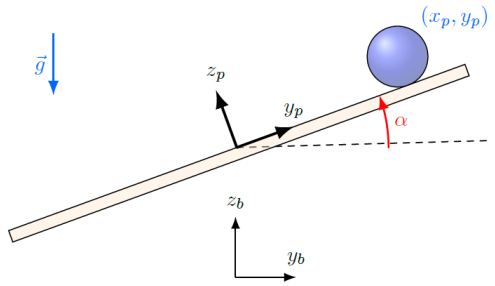


Figure 10: Ball rolling without slipping (pitch $\angle\alpha$)

a) Kinetic Energy

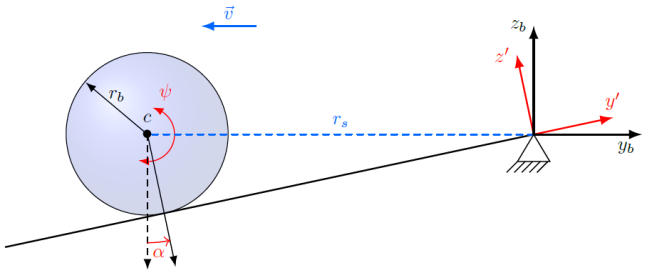


Figure 11: FBD Ball rolling w/out slipping axis rotation

The total kinetic energy consists of contributions from the ball and the plate:

$$T = T_{ball} + T_{plate} \tag{10}$$

Where, the ball's kinetic energy accounts for translational and rotational motions:

$$T_{ball} = \frac{1}{2}mv_b^2 + \frac{1}{2}I_b\omega_b^2 \tag{11}$$

We can define the position vector of the ball from the origin of the plate, as $\vec{s} = [x' \quad y' \quad 0]^T$, and the linear velocity vector of the ball from the origin as $v_b = \frac{d\vec{s}}{dt} = [\dot{x}' \quad \dot{y}' \quad o]^T$. The ball translational kinetic energy can be modeled as,

$$T_{b_{trans}} = \frac{1}{2}m\left(\dot{x}'^2 + \dot{y}'^2\right) \tag{12}$$

The angular velocity of the ball can be simplified by the fact that the linear velocity is proportional to the angular velocity and the distance from the rotation axis, given by $\overrightarrow{v_b} = \overrightarrow{\omega_b} \times \overrightarrow{r_b}$, where $\overrightarrow{r_b} = [0 \quad 0 \quad r_b]^T$ is the radius of the ball described in the 3D reference frame. The ball rotational kinetic energy can be modeled as,

$$T_{b_{rot}} = \frac{1}{2}I_b\left(\frac{\dot{x}'^2}{r_b^2} + \frac{\dot{y}'^2}{r_b^2}\right) \tag{13}$$

The total kinetic energy of the ball can be modeled by the following,

$$\therefore T_{ball} = \frac{1}{2}m\left(\dot{x}'^2 + \dot{y}'^2\right) + \frac{1}{2}I_b\left(\frac{\dot{x}'^2}{r_b^2} + \frac{\dot{y}'^2}{r_b^2}\right) \tag{14}$$

The plate's kinetic energy includes translational and rotational motions.

$$T_{plate} = \frac{1}{2}mv_p^2 + \frac{1}{2}I_{sys}\left(\omega_{plate}\right)^2 \tag{15}$$

The angular velocity of the plate can be modeled by observation, which is simply the combination of pitch and roll, $\omega_p = \frac{\Delta\theta}{\Delta t}$ in the 3-dimensional coordinate system results in $\overrightarrow{\omega_p} = [\dot{\alpha} \quad \dot{\beta}]^T$. Furthermore, the moment of inertia for the system can be represented as the sum of moments between the ball and plate given by $I_{sys} = I_{ball} + I_{plate}$, thus the plate's rotational kinematic motion after linearization can be represented by,

$$T_{p_{(rot)}} = \frac{1}{2}\left(I_p + I_b\right)(\dot{\alpha}^2 + \dot{\beta}^2) \tag{16}$$

The linear velocity of the plate is dependent on the reference location on the plate from the center of the plate $\vec{r} = [x \quad y]$ and is given by $v_p = \vec{r} \cdot \vec{\omega}_p{}^T = x\dot{\alpha} + y\dot{\beta}$. The plate translational kinetic energy can be modeled as.

$$T_{p_{(trans)}} = \frac{1}{2}m\left(x\dot{\alpha} + y\dot{\beta}\right)^2 \tag{17}$$

The total kinetic energy of the plate can be modeled by the following,

$$\therefore T_{plate} = \frac{1}{2}m\left(x\dot{\alpha} + y\dot{\beta}\right)^2 + \frac{1}{2}\left(I_p + I_b\right)(\dot{\alpha}^2 + \dot{\beta}^2) \tag{18}$$

b) Potential Energy

The total potential energy consists of contributions from the ball and the plate:

$$V = V_{ball} + V_{plate} \tag{19}$$

However, since the plate is static and always in contact with the ball, it has no potential energy contribution to the system $V_{plate} = 0$. The only potential energy contribution to the system is gravitational effect on the ball given by $V_{plate} = -mgh$, where h, is the modeled about the combination of pitch and roll angles of the plate $h = x\sin\alpha + y\sin\beta$.

$$\therefore V_{ball} = -mg(x\sin\alpha + y\sin\beta) \tag{20}$$

3) Langrian Dynamics

The Euler-Lagrange equation denotes the relationship between the Lagrangian of a system and the system's equations of motion. The equation provides a systematic way to derive the equations of motion for a system using the principle of least action. The Lagrangian for the ball and plate system is,

$$L = \frac{1}{2}m\left(\dot{x}'^2 + \dot{y}'^2\right) + \frac{1}{2}I_b\left(\frac{\dot{x}'^2}{r_b^2} + \frac{\dot{y}'^2}{r_b^2}\right) + \frac{1}{2}m\left(x\dot{\alpha} + y\dot{\beta}\right)^2 + \frac{1}{2}\left(I_p + I_b\right)(\dot{\alpha}^2 + \dot{\beta}^2) + mg(x\sin\alpha + y\sin\beta \tag{21}$$

For a system described by a Lagrangian, $L(q, \dot{q}, t)$, the Euler-Lagrange equation is,

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}}\right) - \frac{\partial L}{\partial q} = Q \tag{22}$$

where q represents the generalized coordinates, $\dot{q}$ represents the time derivatives of these coordinates (generalized velocities) and t is time. Q represents the generalized non-conservative forces of the system, in our system they are the torques generated by the rotational movement of the plate about the origin represented.

$$q = \begin{bmatrix} x \\ y \\ \alpha \\ \beta \end{bmatrix}, \dot{q} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\alpha} \\ \dot{\beta} \end{bmatrix}, Q = \begin{bmatrix} 0 \\ 0 \\ \tau_x \\ \tau_y \end{bmatrix} \tag{23}$$

After evaluating the Euler-Lagrange with the derived Lagrangian, and taking the partial derivatives respectively, it yields the following second order system of equations,

$$\left(m + \frac{I_b}{r_b^2}\right)\ddot{x}' - m\left(x\dot{\alpha}^2 + y\dot{\alpha}\dot{\beta} + g\sin\alpha\right) = 0 \tag{24}$$

$$\left(m + \frac{I_b}{r_b^2}\right)\ddot{x}' - m\left(x\dot{\alpha}^2 + y\dot{\alpha}\dot{\beta} + g\sin\alpha\right) = 0 \tag{25}$$

$$dsdfd(26)$$

$$dd(27)$$

However, equations (26 & 27), are related to the non-conservative forces of the system, their solutions are coupled with the torques of the plate and are not useful for our application.

Furthermore, the equations can be simplified, by employing the small angle theorem to approximate $\sin\alpha \approx \alpha$, since the angle tilt of the platform is very small (much less than 15 degrees), $\dot{\alpha}^2 = 0$, $\dot{\beta}^2 = 0$, $\dot{\alpha}^2 = 0$ and $\dot{\alpha}\dot{\beta} = 0$. This results in simplified equations,

$$m\ddot{x}' + \frac{I_b}{r_b^2}\ddot{x} - mg\alpha = 0 \tag{28}$$

$$m\ddot{y}' + \frac{I_b}{r_b^2}\ddot{y} - mg\beta = 0 \tag{29}$$

where, the moment of inertia is defined as, $I_b = \frac{2}{5}mr_b^2$, it is important to note that after substituting the moment of inertia the resulting mathematical model is determined to be independent of the mass of the ball, which in counter to the natural assumption that a heavier ball falls faster, however, this proves that the system is behaving under the assumption of rolling without slipping.

Finally, the ball and plate Stewart platform can be modeled by the following set of second order linear equations,

$$\ddot{x}' = \frac{5}{7}g\alpha \tag{30}$$

$$\ddot{y}' = \frac{5}{7}g\beta \tag{31}$$

C. Controller Design

In order to design a controller for this system, a prerequisite is to check the system's stability and verify the criterion for stability. Applying a Laplace transformation to the mathematical model results in the following continuous time domain system,

$$G_{p_x}(s) = \frac{\mathrm{A}(s)}{X(s)} = \frac{5g}{7s^2} \tag{32}$$

$$G_{p_y}(s) = \frac{\mathrm{B}(s)}{X(s)} = \frac{5g}{7s^2} \tag{33}$$

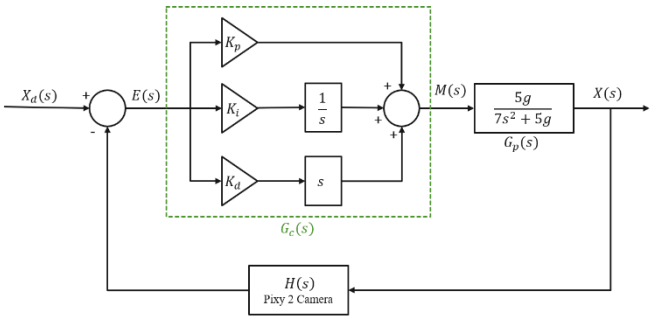


Figure 12: Block Diagram of PID System

The block diagram shown in Figure 12 represents the feedback control loop, where the desired position inputs $X_D(s)$ and $Y_D(s)$ are compared to the actual positions $X(s)$ and $Y(s)$ to compute the error signal $E(s)$. This error is processed through the PID controller $G_C(s)$, which generates the control input $u(t)$ based on proportional, integral, and derivative components of the error. Feedback from the Pixy2 camera system $H(s)$ ensures real-time updates of the ball's position, enabling continuous correction of platform tilt to maintain dynamic stability. The diagram highlights the integration of the vision system, control algorithms, and mechanical actuation, forming a cohesive feedback loop critical for achieving precise ball balancing.

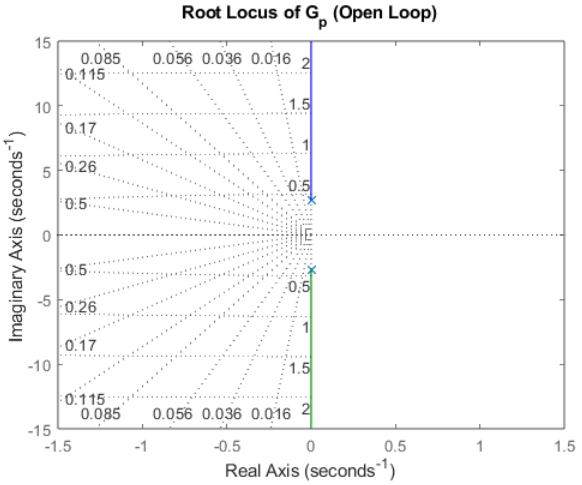


Figure 13: Root Locus of Open Loop

The root locus in Figure 13 shows that the system is marginally stable as indicated by the two poles located as zero, this demonstrates that a controller could stabilize the system.

D. Control Law (PID controller)

The chosen controller for this application to stabilize the system is a PID controller which operates on the principle of minimizing the error between a desired setpoint and the actual system output. The control law is defined as:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \quad (34)$$

Where, $e(t)$ is the error at time t, calculated as the difference between the desired and actual position or orientation. $K_p$is the proportional gain, which scales the error, K_i is the integral gain, which accumulates past errors. K_d, is the derivative gain, which predicts future error trends.

In Figure 14, it demonstrates the step response in continuous time domain of the tuned system with a PID controller utilizing MATLAB's autotune feature.

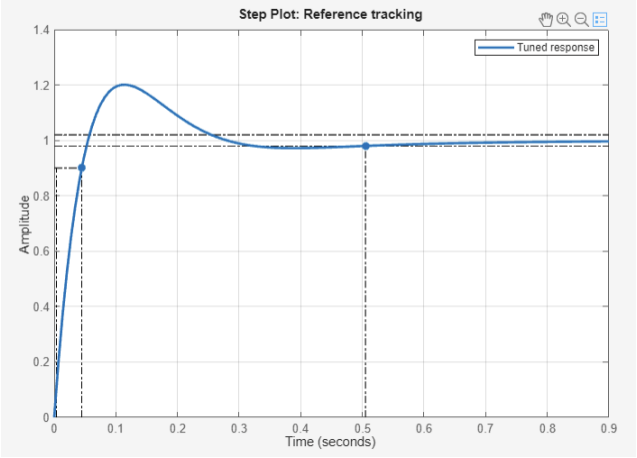


Figure 14: Step Response with tuned PID

The exact values of this tuned controller are not relevant for our application, it only demonstrates the system's response to an applied PID controller and the effectiveness of applying a control law to stabilize the system. In the real world, the applications of a continuous time domain controller are not practical and therefore a discrete time domain controller is needed for practical applications.

E. Discrete time domain analysis ZOH (Z-Domain)

In modern control systems, the use of a discrete PID controller is not only advantageous but often necessary. Discrete controllers are essential because most real-world systems rely on digital hardware, such as microcontrollers or microprocessors, which operate in discrete time. These controllers are inherently designed to process data and compute outputs at specific intervals, aligning seamlessly with the periodic nature of sensor feedback and actuation in the system. For example, the Stewart platform's vision-based feedback system provides positional data at discrete intervals, making a discrete PID controller an ideal choice for directly handling this sampled data.

To design a PID controller for the Stewart platform, a zero-order hold (ZOH) method was used to discretize the system for practical application. It works by holding each discrete control value constant over the sampling interval, creating a piecewise constant signal that approximates the desired continuous control input. The z-transform for the discretized plant from continuous time domain $G_p(s)$ to discrete-time, becomes,

$$G_p(z) = (1 - z^{-1})\mathbb{Z}\left\{\frac{G_p(s)}{s}\right\}$$

$$G_p(z) = (1 - z^{-1})\mathbb{Z}\left\{\frac{7}{s^3}\right\}$$

$$G_p(z) = \frac{3.5T^2(z+1)}{z^2-2z+1} \quad (35)$$

The resulting discretized plant can be applied to the control of the ball on the platform along the coordinate axis and can therefore be used to describe the motion of the ball along the x- and y- axes due to symmetry.

The PID control must also be discretized and for this method, a ZOH with a forward Euler configuration was applied and resulted in the following control law,

$$C(z) = K_p + K_i \frac{T}{z-1} + K_d \frac{z-1}{T} \quad (36)$$

This technique is essential for interfacing digital controllers with continuous physical systems, ensuring compatibility and enabling accurate system response. While simple and effective, ZOH introduces a piecewise nature to the signal, which may lead to minor lag or high-frequency effects in fast-changing systems.

IV. IMLEMENTATION AND TESTING

The resulting Discrete-Time functions provided us with a start for the tuning process. For this application, a sample time of 0.2 seconds was used. The limiting factor for our feedback control system being the Pixy2 camera which operates at a frequency of 60fps which leads to the sample time of 0.2 seconds as our limiting factor. after discretization of our closed loop feedback system, it results in the following discrete-time closed-loop transfer function with a tuned PID,

$$H(z) = \frac{0.07708z^3 - 0.05606z^2 - 0.07694z + 0.0562}{1.077z^3 - 3.056z^2 + 2.923z - 0.9438} \quad (37)$$

Figure 15 shows the step response of two discretized tuned responses that were within our desired specifications. The compared step responses highlight distinct differences between the fast and slow systems in their closed-loop step responses. The fast system demonstrates a slower rise time of 1.4 seconds compared to 1.0 second for the slow system, but it stabilizes significantly quicker, with a settling time of 8.6 seconds versus 12.8 seconds. Additionally, the fast system exhibits lower overshoot (22.57%) and a smaller peak amplitude (1.2257), indicating more controlled and stable behavior. In contrast, the slow system shows a more aggressive response with higher overshoot (44.52%) and a peak amplitude of 1.4452, though it reaches this peak earlier at 2.4 seconds compared to 4.0 seconds for the fast system. However, the increased overshoot and wider steady-state range of the slow system (0.8386 to 1.4452) suggest reduced precision compared to the fast system, which maintains a narrower steady-state range (0.9048 to 1.2257).

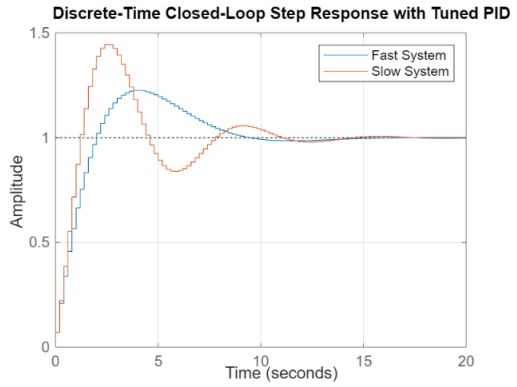


Figure 15: Discrete Time Closed Loop Step Response

Overall, the fast system is better suited for applications requiring stability and precision, while the slow system prioritizes rapid initial response at the expense of overshoot and settling time.

A. Physical Performance of PID controller

After the system was tuned with a PID controller, the values of the PID gains were programmed into Arduino Uno (See Appendix A), the Stewart platform robot. The resulting response matched our expectations of our designed discrete-time system. Effectively validating our design of the PID controller. The fast tuned system physically demonstrated a settling time of about 8 seconds with some overshoot.

In the Stewart platform application, overshoot allows the system to reach the desired position or target state faster. In systems where rapid responses are essential for controlling the ball's movement on the platform. The presence of overshoot minimizes the time it takes for the ball to reach a new position. Overshoot effectively deals with the dynamic and fast-changing ball's position while maintaining stability and achieving the control objective. For Example, when the platform is tilted to counteract the ball's motion, a slight overshoot in the platform's tilt can ensure the ball's movement is reversed promptly, minimizing errors before the system settles. However, the overshoot must be carefully balanced to avoid instability or excessive oscillations.

V. CONCLUSION

This study successfully developed and implemented a 3RPS-Stewart platform, a compact parallel manipulator, to dynamically stabilize a ball using precision actuation and a PID-based closed-loop control system. By leveraging low-cost components such as the Arduino Uno R4 microcontroller and Pixy2 camera for real-time vision tracking at 60 frames per second, the platform achieved a settling time of 8.6 seconds with controlled overshoot of 22.57%. These metrics were realized through meticulous integration of mechanical, electronic, and software systems.

Inverse kinematics computations translated control inputs into precise platform tilts, enabling seamless dynamic adjustments. The design optimizes manufacturability and functionality by combining lightweight aluminum components with custom 3D-printed fittings. Mathematical modeling using the Euler-Lagrange method provided a rigorous foundation for control design, while discretization through a zero-order hold (sampling time of 0.2 seconds) ensured compatibility with digital control systems.

The experimental outcomes validated the effectiveness of the proposed control strategy. The system demonstrated rapid stabilization with a rise time of 1.4 seconds, minimal steady-state error, and reliable performance under dynamic conditions. These achievements highlight the feasibility of creating high-performance robotic platforms with accessible and cost-effective technology.

This work paves the way for further exploration in fields like robotics, automation, and precision handling. Future enhancements could include the adoption of faster microcontrollers for improved response times, as well as expanded capabilities to address more complex dynamic systems. The results underscore the potential of scalable and robust mechatronic designs for research, industrial, and educational applications.

ACKNOWLEDGMENTS

We extend our heartfelt gratitude to Professor Dr. Benham Bahr for his exceptional guidance and support as our advisor throughout this project. His invaluable expertise and constructive feedback greatly contributed to the successful design and implementation of the 3RPS-Stewart platform. Dr. Bahr's encouragement and dedication were pivotal in helping us overcome challenges and achieve our objectives. We deeply appreciate his mentorship, which has been a source of inspiration and learning throughout this journey.

REFERENCES

[1] Unknown. The mathematics of the stewart platform. Wokingham U3A MathGroup.[Online].Available:https://web.archive.org/web/20130506134518/http://www.wokinghamu3a.org.uk/Maths%20of%20the%20Stewart%20Platform%20v5.pdf

[2] Pixy2, "Color connected components – Pixy2 documentation," [Online]. Available: https://docs.pixycam.com/wiki/doku.php?id=wiki:v2:color_connected_components.

[3] A. Koszewnik, K. Troc, and M. Stowik, "PID Controllers Design Applied to Positioning of Ball on the Stewart Platform," Acta Mechanica et Automatica, vol. 8, no. 4, pp. 214–218, 2015. [Online]. Available: http://www.acta.mechanica.pb.edu.pl/volume/vol8no4/39_2014_016_KOSZEWNIK_TROC_SLOWIK.pdf

[4] (2002) Creating a stewart platform model using simmechanics.

[5] Technical Articles and Newsletters. MathWorks. [Online]. Available: http://www.mathworks.com/company/newsletters/articles/creating-a-stewart-platform-model-using-simmechanics.html?requested

[6] S. Kucuk, Serial and Parallel Robot Manipulators - Kinematics, Dynamics, Control and Optimization. InTech, 2012, ch. 10, pp. 179–202. [Online]. Available:http://cdn.intechopen.com/pdfs-wm/34400.pdf

[7] A. S. Jackson, "Design and control of a Stewart platform for dynamic stabilization," M.S. thesis, Mechanical Engineering Dept., California Polytechnic State Univ., San Luis Obispo, CA, USA, 2018. [Online]. Available: https://digitalcommons.calpoly.edu/theses/2124/.

APPENDIX 1

MAIN.ino

```
/************************************************
  3RPS Stewart Platform Ball Bounce & Balance
  ---------------------------------------------
  @description: this program operates a 3DOF RPS Stewart Platform to balance a ball on a platform
                using computer vision for position detection & PID controller to control the balls position.

  @authors: Jose Ramirez & Ethan Lendo
  @institution: Cal Poly Pomona, Dept. of Mech. Engineering
  @date: 07-22-2024
  @version: 1.0

 ************************************************/
 /* TO-DO
    -> tune the PID
    -> Develop bounce
    -> Develop ball patterns

*/
#include <Arduino.h>
#include <Wire.h>
#include <Pixy2.h>
#include <AccelStepper.h>
#include <MultiStepper.h>
#include "InverseKinematics.h"

#define DEBUG 0 // for debugging 1 is on, else off 0

#if DEBUG == 1
#define debug(db) Serial.print(db)
#define debugln(db) Serial.println(db)
#else
#define debug(db)
#define debugln(db)
#endif

// Camera Object
Pixy2 pixy;

// Pixy2 camera offsets (camera's center in pixels)
constexpr double xOffset = 158.0;     // Updated x offset of the platform's center in pixels
constexpr double yOffset = 104.0;     // Updated y offset of the platform's center in pixels
constexpr double zOffset = 0.0;

// Platform and camera parameters
constexpr double circumscribedRadius = 240.0;   // 240 Radius of the circumscribed circle for the hexagon in mm
constexpr double distanceToPlatform = 460.0;    // Distance from camera to the platform in mm

double ball[2];               // X and Y coordinates of the ball
constexpr int x = 0, y = 1;   // Define x, y array indexes
bool detected = false;        // flag to verify ball is detected

// Stepper Motors Object
AccelStepper stepperA(AccelStepper::DRIVER, 9, 8);  //(driver type, STEP, DIR) Driver A
AccelStepper stepperB(AccelStepper::DRIVER, 5, 4);  //(driver type, STEP, DIR) Driver B
AccelStepper stepperC(AccelStepper::DRIVER, 3, 2);  //(driver type, STEP, DIR) Driver C

// Create instance of MultiStepper
MultiStepper steppers;

// Stepper Motor Variables
double speed[3] = {0, 0, 0};     // Motor speed array
double speedPrev[3] = {0, 0, 0}; // Previous motor speed array
int pos[3] = {0, 0, 0};          // Array to hold angle position of each servo
constexpr double angleOrigin = -2.24;  // Origin angle at the start
constexpr double ks = 500.0;           // Speed multiplier constant
constexpr double angleToStep = 3200.0 / 360.0; // Convert angle to step count (microsteps * gearRatio / 360 degrees)
constexpr double gearRatio = 5.18;

// PID Constants (2s settling time)
double kp = 0.072, ki = 0.0025, kd = 0.07;

//double kp = 0.075, ki = 0.0025, kd = 0.07; //pretty good
//double kp = 0.068, ki = 0.0025, kd = 0.078; // almost perfect
//double kp = 0.068, ki = 0.002, kd = 0.25
//double kp = 0.03263, ki = 0.001886, kd = 0.1411;
//double kp = 0.01658, ki = 0.0008346, kd = 0.0823;
//double kp = 0.04302, ki = 0.002855, kd = 0.1621;
//double kp = 0.07756, ki = 0.004005, kd = 0.3755;

double error[2] = {0, 0}, errorPrev[2] = {0, 0}, integr[2] = {0, 0}, deriv[2] = {0, 0}, out[2] = {0, 0}, outf[2] = {0, 0};
// PID terms for X and Y directions
```

```
// Add a sample time for PID calculations
constexpr unsigned long sampleTime = 20; // Sample time in milliseconds

// Variables for PID timing
unsigned long lastPIDTime = 0;
// Variables to capture initial times

/*  Plant(b, p, f, g)
      b = distance from the center of the base to any of its corners
      P = distance from the center of the platform to any of its corners
      f = length of link #1
      s = length of link #2
*/
Plant plant(125.0, 125.0, 100.0, 60.0);

// Function Declarations
void moveTo(double hz, double nx, double ny);
void findBall();
void PID(double setPointX, double setPointY);
void convertToMillimeters(double &pixelX, double &pixelY);

////////////////////////////////////////////////////////////////////////////////
///
/// @brief Arduino setup() Function.
///
///   This function initializes serial communication with PIXY2 and the initial servo
///   parameters needed to initialize the platform.
///
////////////////////////////////////////////////////////////////////////////////
void setup() {
  // Initialize Serial communication
  Serial.begin(115200);

  // Initialize Pixy2
  pixy.init();

  // Optionally turn on Pixy2's LED
  // pixy.setLamp(1, 1);

  // Set max speed and acceleration for steppers
  stepperA.setMaxSpeed(6000);
  stepperA.setAcceleration(3000);

  stepperB.setMaxSpeed(6000);
  stepperB.setAcceleration(3000);

  stepperC.setMaxSpeed(6000);
  stepperC.setAcceleration(3000);

  // SteppersControl instance for multi stepper control
  steppers.addStepper(stepperA);
  steppers.addStepper(stepperB);
  steppers.addStepper(stepperC);

  moveTo(37.75, 0, 0);  // Moves the platform to the home position located at hz=37.75mm
  delay(100);
}

////////////////////////////////////////////////////////////////////////////////
///
/// @brief Main Arduino Loop() Function.
///
///   This loop executes the PID control of the Platform.
///
////////////////////////////////////////////////////////////////////////////////
void loop() {
  PID(0, 0);  // (X setpoint, Y setpoint)
}

/// @fn Find the location of the ball using the pixy2 cam
void findBall() {
  // Request blocks (objects) from Pixy2
  int numBlocks = pixy.ccc.getBlocks();

  debugln((String)"Number of Balls Detected: " + numBlocks);

  if (numBlocks == 1) {
    detected = true;

    // Set current X and Y coordinates for object 0 (assuming this is the ball)
    ball[x] = pixy.ccc.blocks[0].m_y;  // Absolute Y location of the ball, now treated as X
    ball[y] = pixy.ccc.blocks[0].m_x;  // Absolute X location of the ball, now treated as Y
```

```
    debugln((String)"Absolute (X, Y) pixels: (" + ball[x] + ", " + ball[y] + ")");
    Serial.println((String)"Absolute (X, Y) pixels: (" + ball[x] + ", " + ball[y] + ")");

    // Calculate relative coordinates based on the center of the camera's field of view
    ball[x] -= yOffset;
    ball[y] -= xOffset;

    debugln((String)"Relative (X, Y) pixels: (" + ball[x] + ", " + ball[y] + ")");

    // Convert relative coordinates (pixels) to real-world coordinates (mm)
    convertToMillimeters(ball[x], ball[y]);

    debugln((String)"Real-world (X, Y) mm: (" + ball[x] + ", " + ball[y] + ")");

  } else {
    detected = false;
    if (numBlocks > 1) {
      debugln("MULTIPLE BALLS DETECTED!");
    } else {
      debugln("NO BALL DETECTED");
    }
  }
}

/// @fn PID control to calculate platform movement based on setpoints
// Variables for time management
unsigned long previousTimePID = 0;
double elapsedTimePID;

// Updated PID Function
void PID(double setPointX, double setPointY) {
  findBall();  // Find the location of the ball

  if (detected) {
    // Calculate elapsed time since last PID calculation
    unsigned long currentTimePID = millis();
    elapsedTimePID = (currentTimePID - previousTimePID) / 1000.0;  // Convert ms to seconds
    previousTimePID = currentTimePID;

    for (int i = 0; i < 2; i++) {
      // Determine which axis we're calculating for (0 = X, 1 = Y)
      double setPoint = (i == 0) ? setPointX : setPointY;
      double currentPos = (i == 0) ? ball[x] : ball[y];

      // Calculate the error
      error[i] = currentPos - setPoint;

      // Proportional term
      double Pout = kp * error[i];

      // Integral term with windup guard
      integr[i] += error[i] * elapsedTimePID;
      // Limit for the integral to avoid windup
      integr[i] = constrain(integr[i], -1000, 1000);
      double Iout = ki * integr[i];

      // Derivative term (rate of change of error)
      deriv[i] = (error[i] - errorPrev[i]) / elapsedTimePID;
      deriv[i] = isnan(deriv[i]) || isinf(deriv[i]) ? 0 : deriv[i];  // Check for validity
      double Dout = kd * deriv[i];

      // PID output for the axis
      out[i] = Pout + Iout + Dout;
      //outf[i] = constrain(out[i], -5, 5);
      Serial.println((String) "I = " + Iout);

      // Store the current error as previous error for next loop
      errorPrev[i] = error[i];
    }
  } else {
    // If ball not detected, retry finding
    findBall();
  }


  // Move the platform based on PID output
  moveTo(37.75, -out[0], -out[1]);
  delay(5);

  // Debugging output to monitor PID terms and output
  debugln((String) "X OUT = " + out[0] + "   Y OUT = " + out[1]);
  Serial.print("X_Input:");
  Serial.print(ball[x]);
```

```
  Serial.print(",");
  Serial.print("X_Output:");
  Serial.println(out[0]);

  Serial.print("Y_Input:");
  Serial.print(ball[y]);
  Serial.print(",");
  Serial.print("Y_Output:");
  Serial.println(out[1]);
}

/// @fn moves the steppers to the desired position based on the given X & Y coordinates
/// @param hz height of the z translation from the platform.
/// @param nx X- coordinate position of the ball relative to origin
/// @param ny Y- coordinate position of the ball relative to origin
void moveTo(double hz, double nx, double ny) {
  // Check if ball is detected
  if (detected) {
    debugln("Ball Detected :) ");

    // Calculate Stepper position
    for (int i = 0; i < 3; i++) {
      pos[i] = round((angleOrigin - plant.theta(i, hz, nx, ny)) * angleToStep * gearRatio);

      // Calculate Stepper Motor Speeds
      speedPrev[i] = speed[i];
      speed[i] = abs(pos[i] - ((i == 0) ? stepperA.currentPosition() : (i == 1) ? stepperB.currentPosition() :
stepperC.currentPosition())) * ks;
      speed[i] = constrain(speed[i], speedPrev[i] - 500, speedPrev[i] + 500);  // Limit speed change to smooth movement
      speed[i] = constrain(speed[i], 0, 5000);
    }

    debugln((String) "Angle A = " + pos[0] / (angleToStep * gearRatio) + " degrees");
    debugln((String) "Angle B = " + pos[1] / (angleToStep * gearRatio) + " degrees");
    debugln((String) "Angle C = " + pos[2] / (angleToStep * gearRatio) + " degrees");

    // Set Calculated speed and target position
    stepperA.setMaxSpeed(speed[A]);
    stepperA.setAcceleration(speed[A] * 0.70);
    stepperA.moveTo(pos[A]);

    stepperB.setMaxSpeed(speed[B]);
    stepperB.setAcceleration(speed[B] * 0.70);
    stepperB.moveTo(pos[B]);

    stepperC.setMaxSpeed(speed[C]);
    stepperC.setAcceleration(speed[C] * 0.70);
    stepperC.moveTo(pos[C]);

    // Run steppers to target position
    while (stepperA.distanceToGo() != 0 || stepperB.distanceToGo() != 0 || stepperC.distanceToGo() != 0) {
      stepperA.run();
      stepperB.run();
      stepperC.run();
    }
  } else {
    debugln("Ball NOT Detected :( ");

    for (int i = 0; i < 3; i++) {
      pos[i] = round((angleOrigin - plant.theta(i, hz, 0, 0)) * angleToStep * gearRatio);
    }

    debugln((String) "Angle A = " + pos[0] / (angleToStep * gearRatio) + " degrees");
    debugln((String) "Angle B = " + pos[1] / (angleToStep * gearRatio) + " degrees");
    debugln((String) "Angle C = " + pos[2] / (angleToStep * gearRatio) + " degrees");

    // Set Stepper Max Speed and target position
    stepperA.setMaxSpeed(6000);
    stepperA.moveTo(pos[A]);

    stepperB.setMaxSpeed(6000);
    stepperB.moveTo(pos[B]);

    stepperC.setMaxSpeed(6000);
    stepperC.moveTo(pos[C]);

    // Run steppers to target position
    while (stepperA.distanceToGo() != 0 || stepperB.distanceToGo() != 0 || stepperC.distanceToGo() != 0) {
      stepperA.run();
      stepperB.run();
      stepperC.run();
    }
  }
```

```
}

/// @fn Convert pixel coordinates to real-world coordinates in millimeters
void convertToMillimeters(double &pixelX, double &pixelY) {
  // Convert the pixel positions to positions within the hexagonal platform's circumscribed circle
  // Assuming linear transformation, map the pixel range to the circumscribed radius

  pixelX = (pixelX / 208.0) * 2 * circumscribedRadius;  // Scaling X based on pixel range and radius
  pixelY = (pixelY / 316.0) * 2 * circumscribedRadius;  // Scaling Y based on pixel range and radius

  // Ensure ball position stays within the platform's circular bounds
  double distanceFromCenter = sqrt(pixelX * pixelX + pixelY * pixelY);
  if (distanceFromCenter > circumscribedRadius) {
    // Ball is outside the platform bounds
    debugln("Warning: Ball is outside the platform!");
   }
}
```

InverseKinematics.cpp

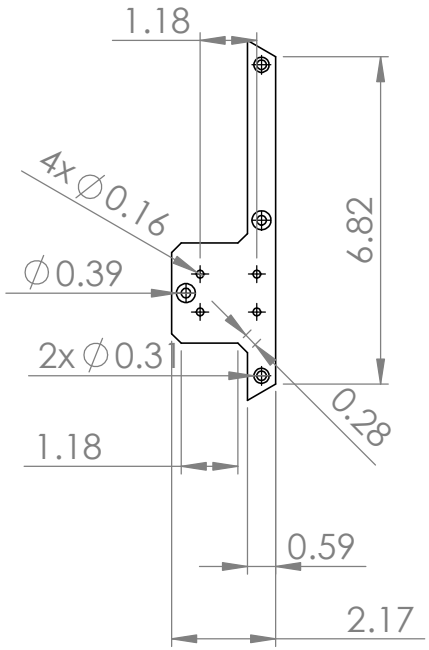
```
#include "InverseKinematics.h"

// Define constants for calculations
double SQRT_3_OVER_2 = sqrt(3.0) / 2.0;

constexpr double HEIGHT_OFFSET = 77.69;  // Offset for height calculation
constexpr double GRAVITY = 9.81;
constexpr double COEFF = 7.0 / (5.0 * GRAVITY);

// Constructor initializes constants and end effector/base positions
Plant::Plant(double _b, double _p, double _f, double _s) : b(_b), p(_p), f(_f), s(_s) {
    // Initialize end effector positions
    endEffector[A][0] = p;
    endEffector[A][1] = 0;
    endEffector[B][0] = -0.5 * p;
    endEffector[B][1] = SQRT_3_OVER_2 * p;
    endEffector[C][0] = -0.5 * p;
    endEffector[C][1] = -SQRT_3_OVER_2 * p;

    // Initialize base positions
    basePrisJoint[A][0] = p;
    basePrisJoint[A][1] = 0;
    basePrisJoint[B][0] = -0.5 * p;
    basePrisJoint[B][1] = SQRT_3_OVER_2 * p;
    basePrisJoint[C][0] = -0.5 * p;
    basePrisJoint[C][1] = -SQRT_3_OVER_2 * p;
}

double Plant::theta(int leg, double H, double xe, double ye) {
    // Calculate height and tilt angles
    double Hz = H + HEIGHT_OFFSET;
    double alpha = ye * COEFF * DEG_TO_RAD;
    double beta = xe * COEFF * DEG_TO_RAD;

    // Precompute trigonometric values
    double cos_alpha = cos(alpha);
    double sin_alpha = sin(alpha);
    double cos_beta = cos(beta);
    double sin_beta = sin(beta);

    double nx, ny, nz, length;

    // Calculate length based on the leg
    switch (leg) {
        case A:
            nx = endEffector[A][0] * cos_beta - basePrisJoint[A][0];
            nz = Hz - endEffector[A][0] * sin_beta;
            length = sqrt(nx * nx + nz * nz);
            break;

        case B:
            nx = endEffector[B][0] * cos_beta + endEffector[B][1] * sin_beta * sin_alpha -
basePrisJoint[B][0];
            ny = endEffector[B][1] * cos_alpha - basePrisJoint[B][1];
```

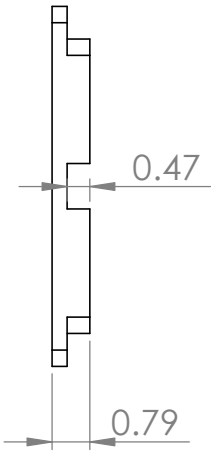
```
            nz = Hz - endEffector[B][0] * sin_beta + endEffector[B][1] * cos_beta * sin_alpha;
            length = sqrt(nx * nx + ny * ny + nz * nz);
            break;

        case C:
            nx = endEffector[C][0] * cos_beta + endEffector[C][1] * sin_beta * sin_alpha -
basePrisJoint[C][0];
            ny = endEffector[C][1] * cos_alpha - basePrisJoint[C][1];
            nz = Hz - endEffector[C][0] * sin_beta + endEffector[C][1] * cos_beta * sin_alpha;
            length = sqrt(nx * nx + ny * ny + nz * nz);
            break;

        default:
            return 0.0;  // Invalid leg, return 0
    }

    // Calculate the angle and return it in degrees
    double angle = HALF_PI - acos((length * length + s * s - f * f) / (2 * s * length));
    return angle * RAD_TO_DEG;
}
```

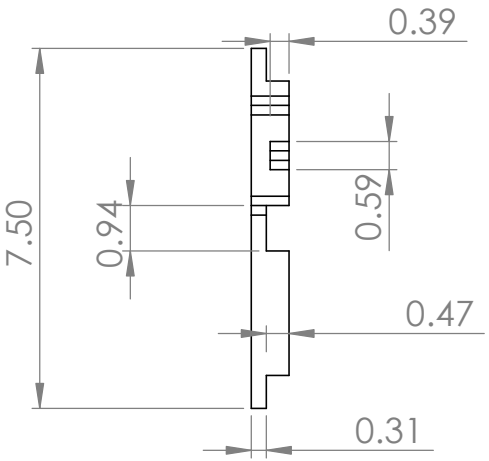
InverseKinematics.h

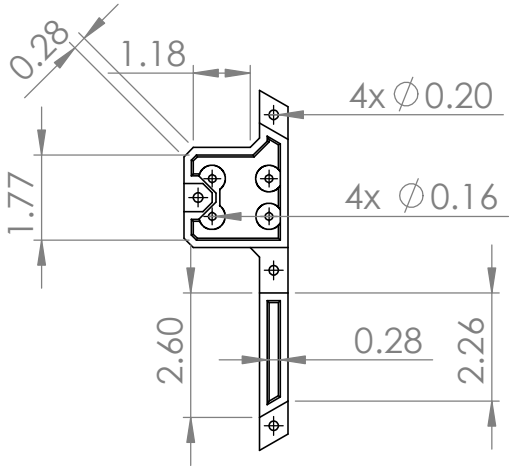
```
#ifndef InverseKinematics_H
#define InverseKinematics_H

#include <Arduino.h>
#include <math.h>

// Define servo constants
#define A 0
#define B 1
#define C 2

class Plant {
  public:
    Plant(double b, double p, double f, double s);
    double theta(int leg, double h, double xe, double ye); // returns the angle value of each servo A, B, C

  private:
    double b;  // distance from the center of the base to any of its corners
    double p;  // distance from the center of the platform to any of its corners
    double f;  // length of link #1
    double s;  // length of link #2
    double endEffector[3][2];
    double basePrisJoint[3][2];
};

#endif
```

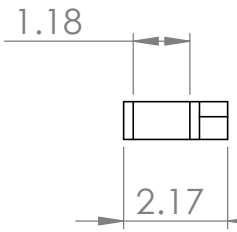
APPENDIX 2

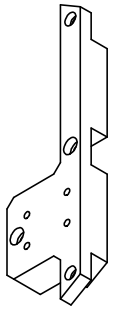


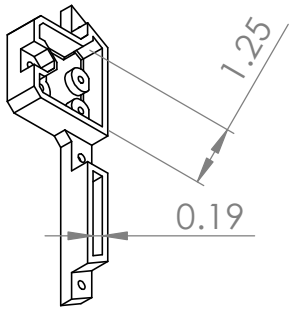


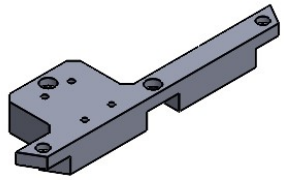










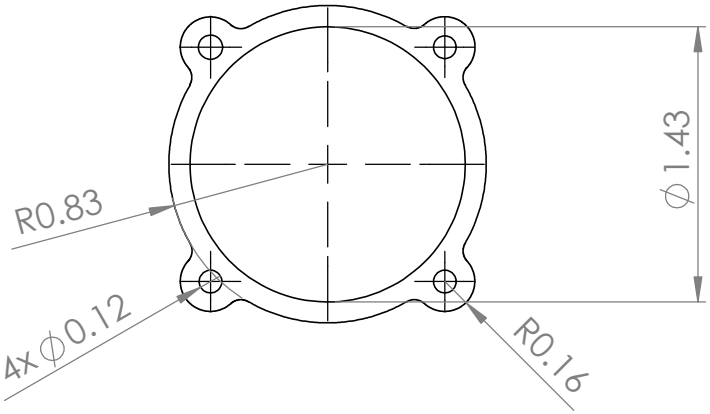


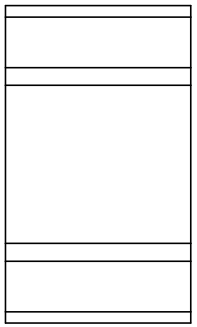
		UNLESS OTHERWISE SPECIFIED:		NAME	DATE
		DIMENSIONS ARE IN INCHES TOLERANCES: FRACTIONAL ± ANGULAR: MACH ± BEND ± TWO PLACE DECIMAL ± THREE PLACE DECIMAL ±	DRAWN		
			CHECKED		
			ENG APPR.		
			MFG APPR.		
		INTERPRET GEOMETRIC TOLERANCING PER:	Q.A.		
		MATERIAL	COMMENTS:		
NEXT ASSY	USED ON	FINISH			
APPLICATION		DO NOT SCALE DRAWING			

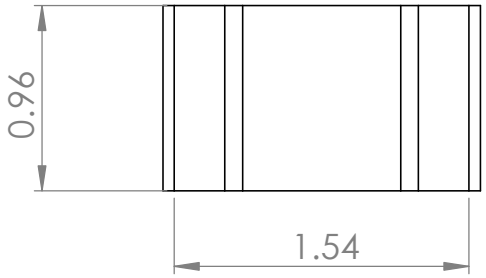
TITLE:

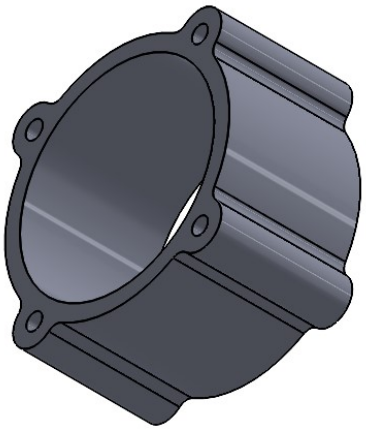
Base-Stepper

SIZE	DWG. NO.	REV
A		
SCALE: 1:4	WEIGHT:	SHEET 1 OF 1









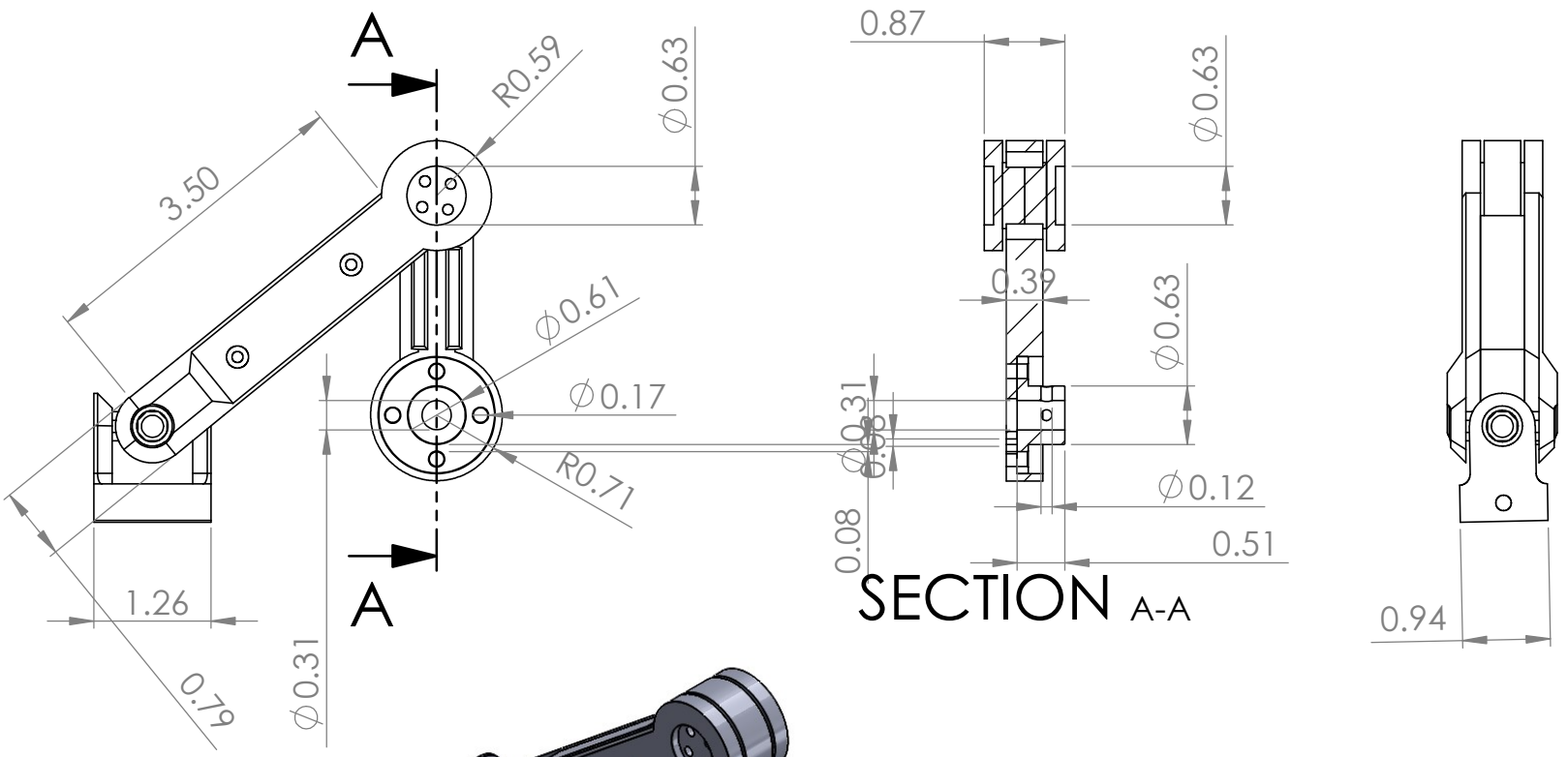
		UNLESS OTHERWISE SPECIFIED:		NAME	DATE
		DIMENSIONS ARE IN INCHES TOLERANCES: FRACTIONAL ± ANGULAR: MACH ± BEND ± TWO PLACE DECIMAL ± THREE PLACE DECIMAL ±	DRAWN		
			CHECKED		
			ENG APPR.		
			MFG APPR.		
		INTERPRET GEOMETRIC TOLERANCING PER:	Q.A.		
		MATERIAL	COMMENTS:		
NEXT ASSY	USED ON	FINISH			
APPLICATION		DO NOT SCALE DRAWING			

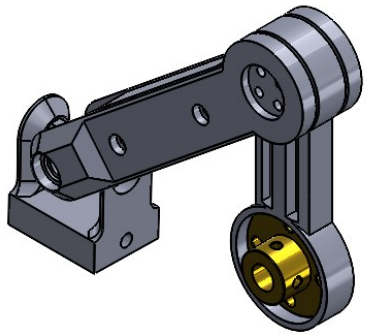
TITLE:

Stepper_Spacer

SIZE	DWG. NO.	REV
A		

SCALE: 1:1	WEIGHT:	SHEET 1 OF 1





		UNLESS OTHERWISE SPECIFIED:		NAME	DATE
		DIMENSIONS ARE IN INCHES TOLERANCES: FRACTIONAL ± ANGULAR: MACH ± BEND ± TWO PLACE DECIMAL ± THREE PLACE DECIMAL ±	DRAWN		
			CHECKED		
			ENG APPR.		
			MFG APPR.		
		INTERPRET GEOMETRIC TOLERANCING PER:	Q.A.		
		MATERIAL	COMMENTS:		
NEXT ASSY	USED ON	FINISH			
APPLICATION		DO NOT SCALE DRAWING			

PROPRIETARY AND CONFIDENTIAL

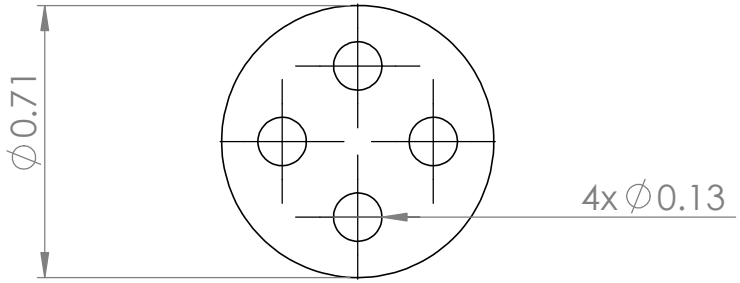
THE INFORMATION CONTAINED IN THIS DRAWING IS THE SOLE PROPERTY OF <INSERT COMPANY NAME HERE>. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF <INSERT COMPANY NAME HERE> IS PROHIBITED.

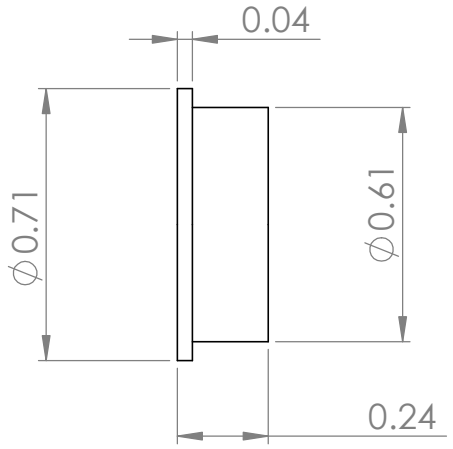
TITLE:

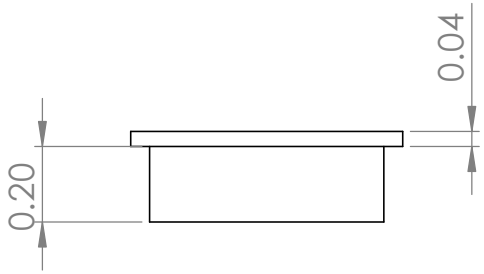
Actuator Assem

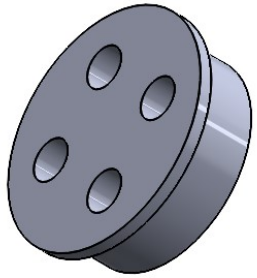
SIZE	DWG. NO.	REV
A		

SCALE: 1:2	WEIGHT:	SHEET 1 OF 1





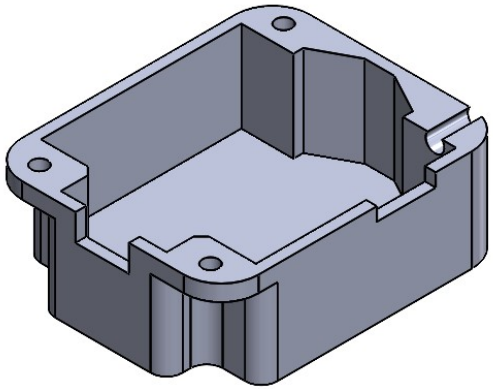


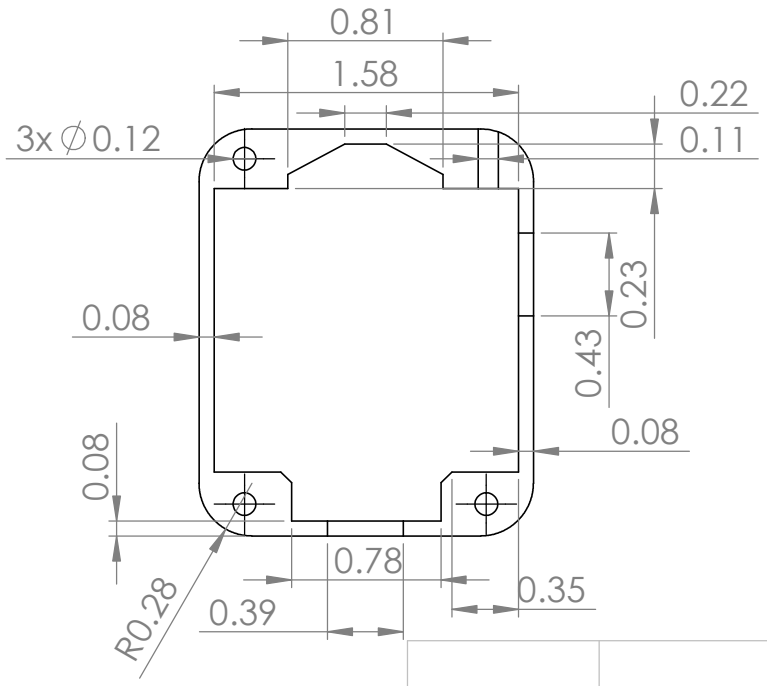


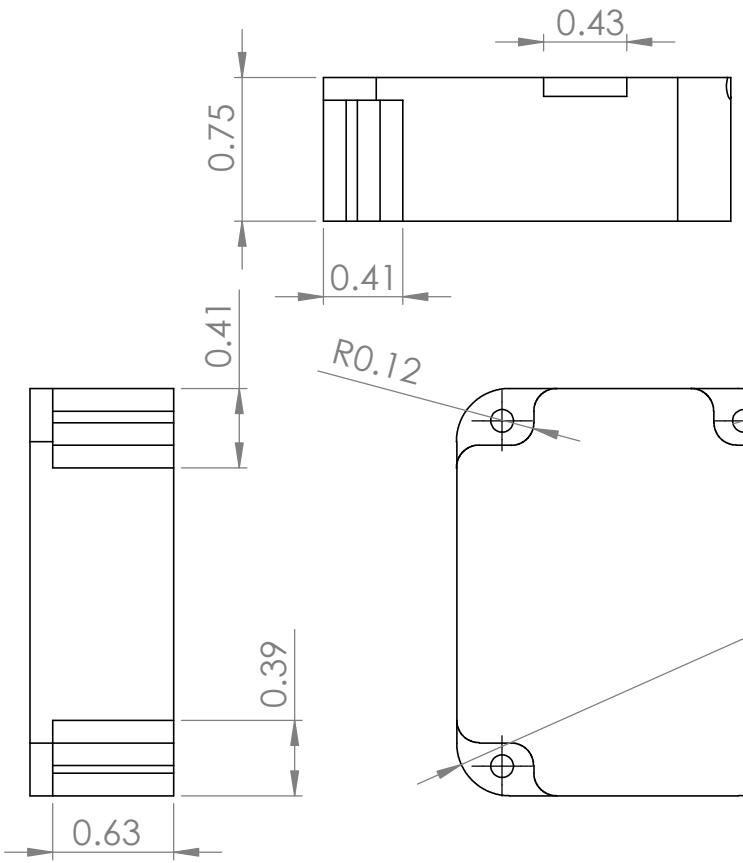
		UNLESS OTHERWISE SPECIFIED:		NAME	DATE
		DIMENSIONS ARE IN INCHES TOLERANCES: FRACTIONAL ± ANGULAR: MACH ± BEND ± TWO PLACE DECIMAL ± THREE PLACE DECIMAL ±	DRAWN		
			CHECKED		
			ENG APPR.		
			MFG APPR.		
		INTERPRET GEOMETRIC TOLERANCING PER:	Q.A.		
		MATERIAL	COMMENTS:		
NEXT ASSY	USED ON	FINISH			
APPLICATION		DO NOT SCALE DRAWING			

TITLE:

SIZE	DWG. NO.	REV
A	Bearing_shaft	
SCALE: 2:1	WEIGHT:	SHEET 1 OF 1





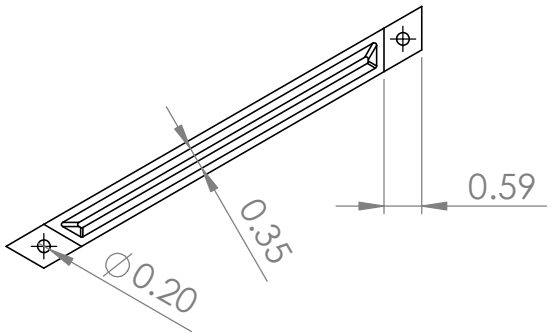


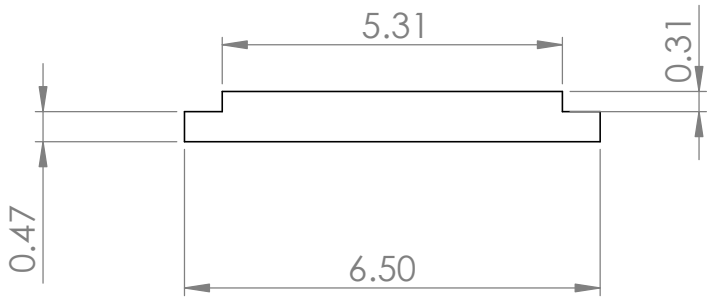
		UNLESS OTHERWISE SPECIFIED:		NAME	DATE
		DIMENSIONS ARE IN INCHES TOLERANCES: FRACTIONAL ± ANGULAR: MACH ± BEND ± TWO PLACE DECIMAL ± THREE PLACE DECIMAL ±	DRAWN		
			CHECKED		
			ENG APPR.		
			MFG APPR.		
		INTERPRET GEOMETRIC TOLERANCING PER:	Q.A.		
		MATERIAL	COMMENTS:		
NEXT ASSY	USED ON	FINISH			
APPLICATION		DO NOT SCALE DRAWING			

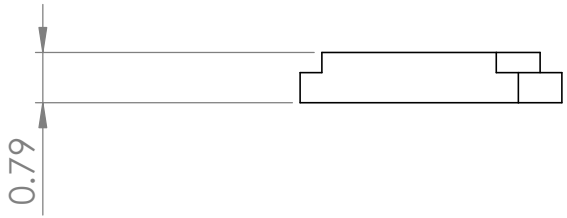
TITLE:

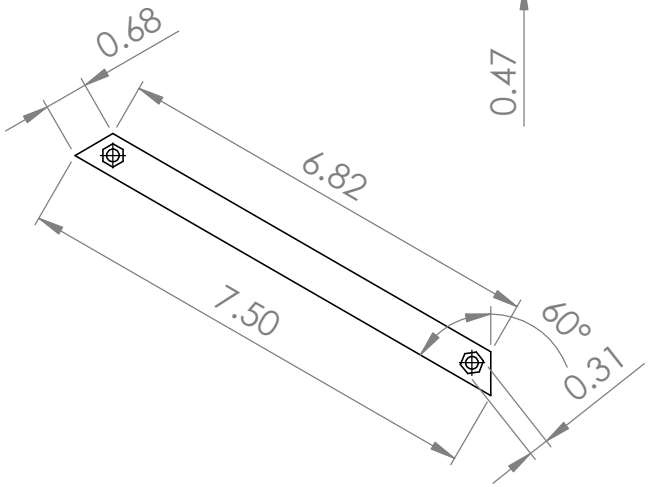
Pixy2_case

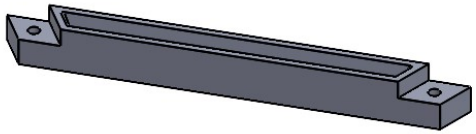
SIZE	DWG. NO.	REV
A		
SCALE: 1:1	WEIGHT:	SHEET 1 OF 1











		UNLESS OTHERWISE SPECIFIED:		NAME	DATE
		DIMENSIONS ARE IN INCHES TOLERANCES: FRACTIONAL ± ANGULAR: MACH ± BEND ± TWO PLACE DECIMAL ± THREE PLACE DECIMAL ±	DRAWN		
			CHECKED		
			ENG APPR.		
			MFG APPR.		
		INTERPRET GEOMETRIC TOLERANCING PER:	Q.A.		
		MATERIAL	COMMENTS:		
NEXT ASSY	USED ON	FINISH			
APPLICATION		DO NOT SCALE DRAWING			

PROPRIETARY AND CONFIDENTIAL

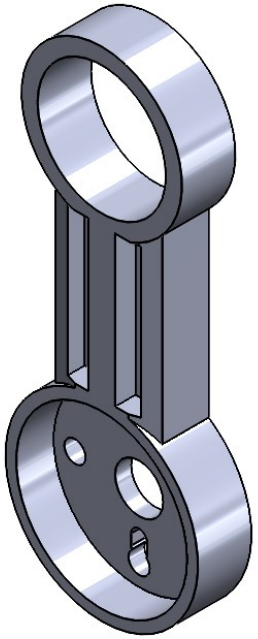
THE INFORMATION CONTAINED IN THIS DRAWING IS THE SOLE PROPERTY OF <INSERT COMPANY NAME HERE>. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF <INSERT COMPANY NAME HERE> IS PROHIBITED.

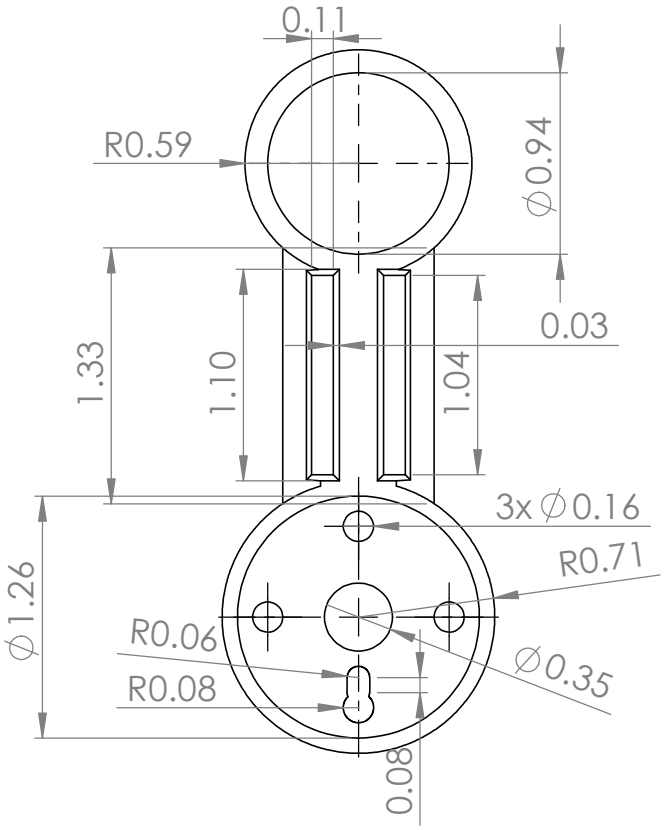
TITLE:

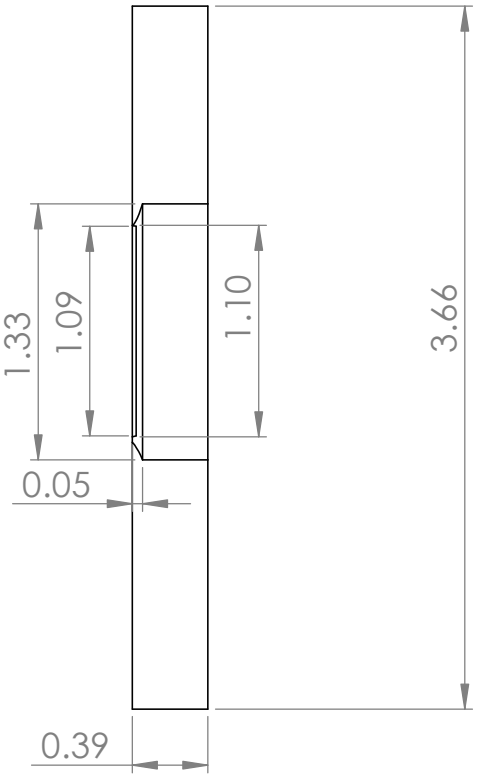
Base-Connector

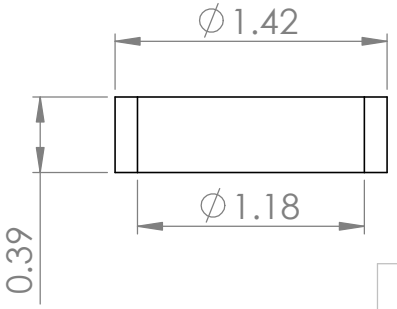
SIZE	DWG. NO.	REV
A		

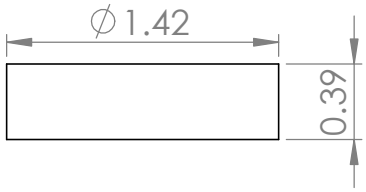
SCALE: 1:3	WEIGHT:	SHEET 1 OF 1











		UNLESS OTHERWISE SPECIFIED:		NAME	DATE
		DIMENSIONS ARE IN INCHES TOLERANCES: FRACTIONAL ± ANGULAR: MACH ± BEND ± TWO PLACE DECIMAL ± THREE PLACE DECIMAL ±	DRAWN		
			CHECKED		
			ENG APPR.		
			MFG APPR.		
		INTERPRET GEOMETRIC TOLERANCING PER:	Q.A.		
		MATERIAL	COMMENTS:		
NEXT ASSY	USED ON	FINISH			
APPLICATION		DO NOT SCALE DRAWING			

TITLE:

Input_actuator

SIZE	DWG. NO.	REV
A		
SCALE: 1:1	WEIGHT:	SHEET 1 OF 1