Project BALANCE: Design and Implementation of a Visual Servoing System

Matthew Ebert
ECE 490; Supervisor: Dr. Capson
dept. of Electrical and Computer Engineering
University of Victoria
Date: 2022-AUG-11

Repository of Project:
[ebertmx/BALANCE_CS: Control and SImulation of the BALANCE system. (github.com)]
Video Examples:
[BALANCE_Video_Examples]

Table of Contents

Table of Figures

Tables

Executive Summary

Project BALANCE is a visual servoing design project. The system consists of a ball and beam control system viewed by a camera. Using computer vision analysis, a feedback system is generated for use in a PID tuned control loop. The main controller of the design is a Raspberry Pi 3B. This controller connects to a DC motor and a raspberry pi camera module. Currently, the raspberry pi connects to an external computer running a MATLAB program which executes the control functionality. Testing of the BALANCE system was successful although not all requirement of the project were met. During testing, the ball was position at a desired location on the beam within a certain margin, but the settling time and steady state error objectives were not met. These problems with steady state error and long settling times were largely caused by a slow control rate, motor and beam dead zones, and camera leveling errors. Future development will aim to improve the control rate of the system by running the software directly on the Raspberry Pi controller. Additionally, the mechanisms of the design will be improved to reduce dead zone and overcome the steady state error.

I. Introduction

To demonstrate the use and limitations of visual servoing, a control system was developed and implemented with the use of computer vision (CV), control theory, and visual servoing (VS) techniques. Project BALANCE is a ball and beam control system. This system is well covered in control theory and allows for easy integration of CV. Project BALANCE has been developed to a successful state. Although not all the design goals and requirements were met, the system can control the motion of the ball and position it with some accuracy.

II. Background

Visual Servoing is the application of computer vision in control processes. VS uses cameras and CV image analysis to create a feedback signal for use in a control loop. When processing power is limited, these types of system can be difficult to implement. A full review of visual servoing can be found in an earlier report [1].

There are many examples of VS systems in varying environment and degrees of freedom [2] [3] [4]. These systems often involve identifying and manipulation object in scenarios where other sensors are not suitable. To simplify development and the control theory, Project BALANCE comprises only 2 degrees of freedom (DoF).

III. Relavent Theory

A summary of the theory applied in this system is covered in this section. For more detailed background and theory see earlier reports [1] [5].

A. Ball and Beam Control System

The ball and beam system is a fundamental system of control engineering. While not perfectly linear, it allows for an intuitive and visual display of control theory [6].

The ball and beam system can be separated into two control system: the actuator which controls the beam angle; and the beam and ball which dictates the ball's acceleration. A diagram of these systems is shown in Figure 1.

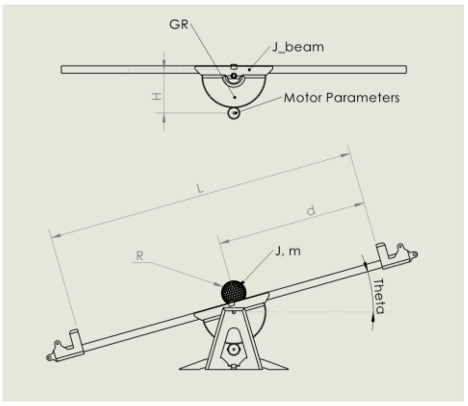


Figure 1: Control system diagrams and parameters for the actuator and ball and beam.

The actuator position transfer function can be found as

$$\frac{\boldsymbol{\theta}(\boldsymbol{s})}{\boldsymbol{V}(\boldsymbol{s})} = \mathrm{GR}\frac{\boldsymbol{K}}{\boldsymbol{s}\big((\boldsymbol{J}/(\boldsymbol{GR^2})\boldsymbol{s}+\boldsymbol{b})(\boldsymbol{Ls}+\boldsymbol{R}\big)+\boldsymbol{K^2}}$$

With the parameters defined in Table 1 [7].

Table 1: Actuator TF parameters

Symbol	Description
θ	Motor angle
V	Input Voltage (PWM controlled in this case)
K	Motor torque and back emf constant
J	Moment of inertia of the entire actuator assembly
b	Viscous friction constant
R	Motor Resistance
L	Motor Inductance
GR	Gear ratio

Note the gear ratio is used to reflect the moment of inertia from the beam to the motor shaft and the motor shaft inertia consider negligible compared to the inertia of the beam assembly [8].

The ball and beam system can be linearized with the sine small angle approximation to give the transfer function

$$\frac{\boldsymbol{D}(\boldsymbol{s})}{\boldsymbol{\theta}(\boldsymbol{s})} = \frac{-\boldsymbol{mg}}{\boldsymbol{s^2}(\frac{\boldsymbol{J}}{\boldsymbol{R^2}}+\boldsymbol{m})}$$

With the parameters defined in Table 2 [9].

Table 2: Ball and Beam TF parameters

Symbol	Description
theta	Beam angle
R	Ball Radius
m	Ball mass
J	Ball moment of inertia
g	-9.81m/s^2
d	Distance from objective

B. Image Based Visual Servoing

Image based visual servoing (IBVS) is one of the two common methods of VS. This method projects a 3D scene onto a 2-dimensional image plane to define an objects position and calculate the subsequent control signal. Given a set of pixel locations u and v, the 2D image plane coordinate can be found from the following formula

$$\begin{cases} x &= X/Z = (u - c_u)/f\alpha \\ y &= Y/Z = (v - c_v)/f, \end{cases}$$

Where c, f and α are intrinsic camera parameters [10]. These intrinsic parameters must be estimate accurately from a process called camera calibration. In addition, real world coordinates of the object may be found by multiplying the image plane coordinates by the distance from the image plane to the object plane, Z.

IV. Design Requirements

The design requirements set for the project are listed in Table 3. Additionally, since not all requirements were met, the status of each requirement is also shown.

Table 3: Functional Design Requirements of the Control Project

R No.	Title	Description	Status
1	Stable Control	The system should respond to input to control or drive a system to a steady, stable, and final state which corresponds to the input.	**Complete. System can position a ball to a stable position with occasional steady state error.**
3	Embedded Design	The system should be self-contained and use only integrated hardware (microcontroller, camera, motor ect.) to accomplish the task.	**Incomplete. The system currently requires a connected computer running MATLAB. However, MATALAB allows for C++ code generation which would allow the system to meet this requirement.**
4	Portable	The system will be used to demonstrate visual servoing; thus, it must be mobile and possible to deploy in many situations.	**Complete. The system is easily portable. Requires only a standard wall plug for power.**
5	Maximum Vision	The system will use camera(s) as the primary feedback sensor. Although other sensors might be necessary, they should be kept to a minimum.	**Complete. The camera comprises the entirety of the sensor equipment**
6	Multiple Configurations	The system should allow for several configurations to demonstrate different methods of visual servo including hand-in-eye and hand-to-eye setups.	**Incomplete. Only one configuration is currently operational**
7	Real Time	The system should operate smoothly in real time	**Complete. The system can operate in real time at a rate of 30Hz.**
8	Accuracy	The system should control the device to within 10% of the accuracy of the least accurate component (Camera or motor resolution).	**Complete. Due to the motor speed limitations, the ball is positioned within this requirement**
9	Speed	The system should complete the task as fast as possible with the hardware.	**Incomplete**
10	User Interface	The system should have an integrate UI which allows for easy setup and control.	**Partially complete. The system has a UI which allows for real time PID tuning.**
11	Data Storage	The controller should store and save state, performance, and configuration data for later analysis.	**Complete. The system can store and plot data.**
12	Rise Time	The rise time of the system should be less than 1.5 seconds (0.1 to 0.9)	**Complete. The system can consistently position the ball within 10% of the objective within 1.5 seconds**
13	Settling Time	Settling time of the system should be less than 5 seconds (5%)	**Incomplete. Currently the system takes over 5 seconds to control the ball to within 5% of the objective position**

V. BALANCE System Design Considerations

The BALANCE system is a 2 DoF ball and beam control system. The major components of the design are the beam and ball, a DC motor, a raspberry pi controller, and a camera sensor. BALANCE is an eye-to-hand IBVS controlled VS system. The externally mounted camera has a view of both the ball and beam which, using CV analysis, it uses to construct feedback signals for the controller. The Raspberry Pi 3B (Pi) is connected to the DC motor and camera. Currently, the Pi is connected to a computer running MATLAB which executes MATLAB functions to run on the Pi. In future, these MATLAB functions will be code generated into a C++ executable to execute directly on the Pi, making the system independent/embedded. Figure 2 and Figure 3 show the overall system.

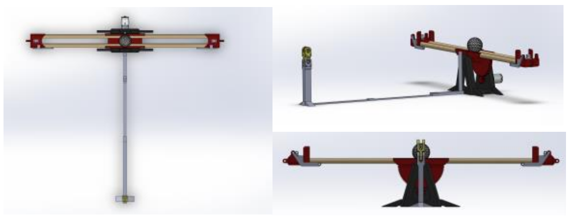


Figure 2: 3D CAD model of BALANCE System

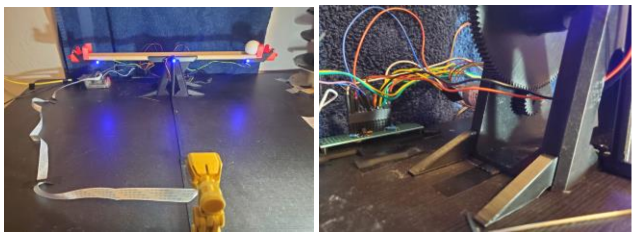


Figure 3: Realized BALANCE system

VI. Design

The design of the BALANCE system consists of four parts: The control model; the mechanical and electrical hardware; the CV system; and the software implementation.

A. Control Model

The control system of BALANCE consists of 2 nested PID tuned control loops (Figure 4). The outer loop controls the ball position, the inner loop controls the beam angle. The input to the outer loop is the objective ball position. This signal is fed through the outer PID block which computes the desired beam angle. The desired beam angle is fed to the inner PID controller which outputs the motor signal required to achieve the angle. The BALANCE CV system measures the ball position and beam angle and sends these as feedback signals in the respective control loops.

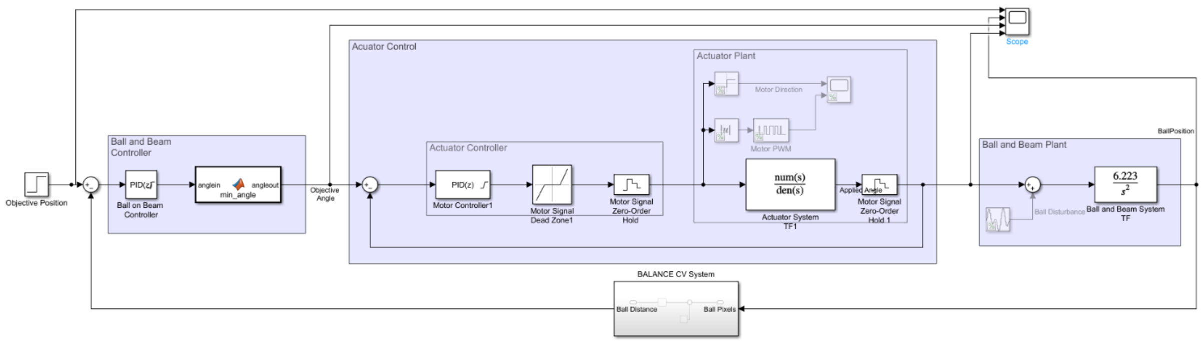


Figure 4: BALANCE Control Loop

Simulink was used to model the system and find initial tunings for the PID controllers.

1) *Actuator Controller*

The actuator control loop dictates the beam angle and was the first controller develop. Figure 5 shows the Simulink model for the actuator controller.

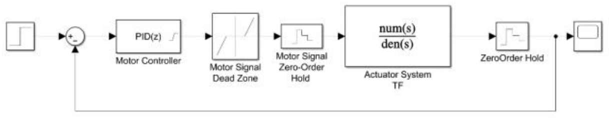


Figure 5: Actuator Control Loop

Upon initial implementation of the realized actuator system, it was found that the BLDC has a minimum speed of rotation. This speed corresponds to a PWM duty signal of 8% or 0.264V DC. This was modeled in Simulink with the addition of a dead zone. The actuator transfer function was modeled with the equation

$$\frac{\boldsymbol{\theta(s)}}{\boldsymbol{V(s)}} = \mathrm{GR}\frac{\boldsymbol{K}}{\boldsymbol{s\big((J/(GR^2)s+b)(Ls+R\big)+K^2}}$$

The moment of inertia J and the gear ratio were calculated from the SolidWorks models of the assembly. The constants related to the motor could not be found in documentation; consequently, standard approximations were used for simulation purposes [7]. The values of these parameters can be found in Figure 4.

Table 4: Parameter values for actuator transfer function

Symbol	**Description**	**Value**
K	Motor torque and back emf constant	0.0274
J	Moment of inertia of the entire actuator assembly	0.002 kg*m^2
b	Viscous friction constant	3.5077E-6
R	Motor Resistance	4 ohms
L	Motor Inductance	2.75E-6 H
GR	Gear ratio	110/20

The model was discretized with a zero-order-hold method. PID parameters were created using MATLAB's PID tuner app to produce the step response shown in Figure 6.

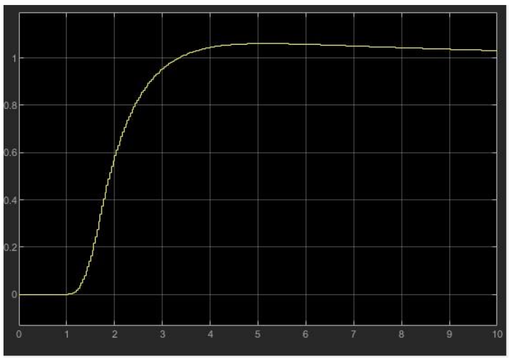


Figure 6: Actuator controller simulation results to step response

The results show a slight (<10%) steady state error. This result is due to the motor dead shown and the small integral of the PID controller. The integral gain was kept small to maintain stability.

These PID tunings were used as the initial settings for the realized system.

2) *Ball and Beam Controller*

The ball and beam controller dictates the acceleration and, hence, the position of the ball. Figure 6 shows the Simulink model used to develop the controller.

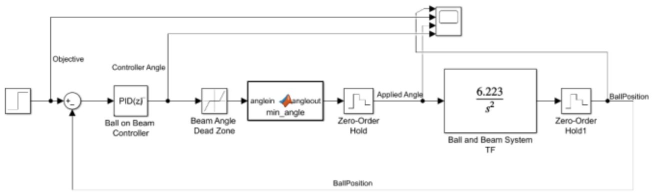


Figure 7: Ball and Beam Control Loop

When testing the realized system, it was observed that the beam required a minimum slope for the ball to achieve any motion. This was simulated in the model as a beam angle dead zone. The transfer function was inserted into the model with the values shown in Table 5

Table 5: Ball and Beam transfer function parameter values.

Symbol	Description	Value
R	Ball Radius	20/1000 m
m	Ball mass	2.7/1000 kg
J	Ball moment of inertia	6.207e-07 kgm^2
G	Acceleration of gravity	-9.81 m/s

The PID block was tuned with MATLAB's PID Tuner app to produce the simulation results shown in Figure 8.

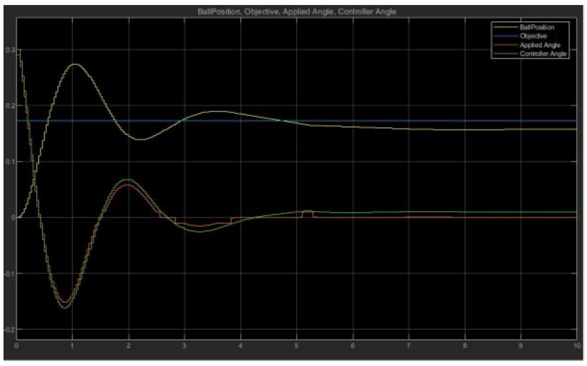


Figure 8: Ball and beam simulation results to step response

Again, this simulation shows a small steady state error which was caused by the small integral term and the angle dead zone of the beam and ball.

3) Full System Simulation

After the two sub systems were tuned, the full subsystem was simulated to produce the step response shown in Figure 9. This response is not ideal. The settling time required is over 5s and the action of the control prevents steady state. The reason for these oscillations is due to the dead zones in each sub system. When these zones are removed from the simulation, the response appears similar to Figure 8.

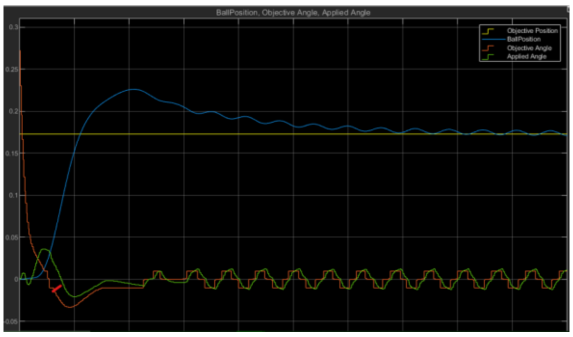


Figure 9: BALANCE system simulation response to step response discretized with the Tustin method.

Attempts to improve the step response of the simulation with different PID tunings were unsuccessful. Therefore, the model was accepted as a viable controller and implemented in the real system.

a) Addition of a acceleration term

After testing was preformed, insufficient dampening from the derivative gain prevented the ball position from reaching a steady state for a significant period (>10s). A custom solution was implemented with the addition of an acceleration term in the ball and beam control system. This term adds a signal using the acceleration of the position error to the ball and beam control signal which increases dampening. The custom modification is shown in Figure 10.

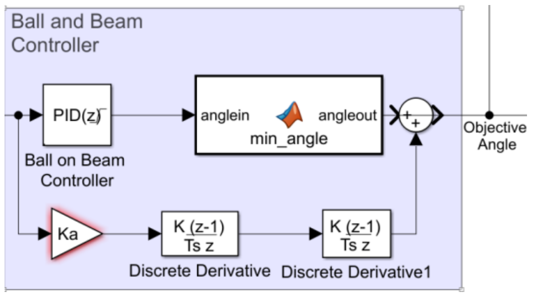
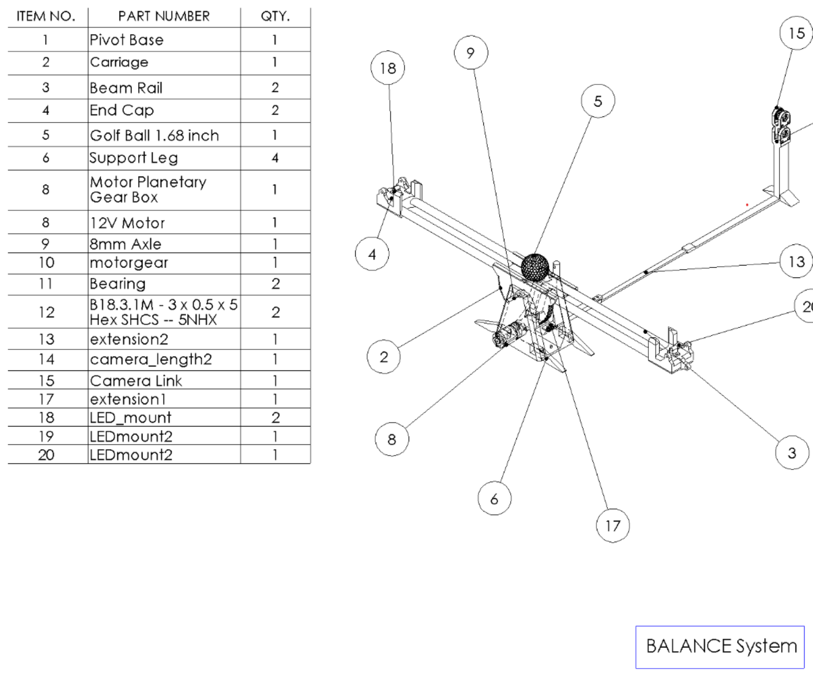


Figure 10: Acceleration term addition to ball and beam control loop

B. Mechanical and Electrical

The mechanical elements were design in SolidWorks. Most components were 3D printed. Figure 11 shows the BOM for the system.



ITEM NO.	PART NUMBER	QTY.
1	Pivot Base	1
2	Carriage	1
3	Beam Rail	2
4	End Cap	2
5	Golf Ball 1.68 inch	1
6	Support Leg	4
8	Motor Planetary Gear Box	1
8	12V Motor	1
9	8mm Axle	1
10	motorgear	1
11	Bearing	2
12	B18.3.1M - 3 x 0.5 x 5 Hex SHCS -- 5NHX	2
13	extension2	1
14	camera_length2	1
15	Camera Link	1
17	extension1	1
18	LED_mount	2
19	LEDmount2	1
20	LEDmount2	1

Figure 11: BOM for the BALANCE mechanical assembly.

The electrical system consists of a Raspberry Pi, Power supply, 3 LED's, a transistor inverter circuit, and a BLDC motor. The circuit diagram is shown in Figure 12.

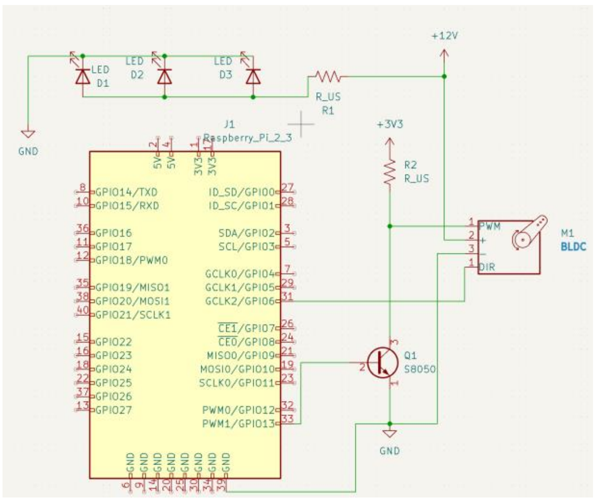


Figure 12: Electrical schematic for the BALANCE system

C. Computer Vision System

BALANCE uses a raspberry pi camera as its primary sensor. The camera is mounted orthogonally 370 mm away from the plane of action of the beam. This position gives the camera a complete view of the system and maximizes the pixel per mm resolution of the area of interest. The Raspberry Pi Camera operates at several resolutions and frame rates. For this application, the camera was configured for 60fps with a 640x480 frame size [11].

Running any advanced image processing through the raspberry pi proved very expensive and significantly decreased the sample rate of the process. Consequently, thresholding was employed to improve the speed of image processing. For this purpose, LED's were positioned at key points on the BALANCE system. This addition allowed for high thresholding values which removed all detail except these points. Additionally, the ball was chosen for its white reflective surface so it could be easily differentiated from the background. A black background was also employed to further emphasize the ball's shape.

The 3 LEDs were positioned with two on either end of the beam and one statically fixed to center of the assembly. The LED in the middle acts as a calibrating point for the system. Its location set the middle reference point of the beam. The two side LED can be used to find the angle of the beam either using both of their positions or one side and the center reference point.

To find the ball and beam positions, a binary blob searching function was employed. The MATLAB computer vision toolbox comes with a function which, given a binary image, can detect blobs or circles of a certain defined area. Through trial and error, this function and the thresholding of the camera image was tuned such that the ball and beam's location was consistently found. To further increase the speed of these functions, the captured image were resized based on the previous ball and beam locations so that fewer calculations were needed to find the new positions.

The pixel locations were converted to positions on the image plane using the formula's

$$\begin{cases} x &= X/Z = (u - c_u)/f\alpha \\ y &= Y/Z = (v - c_v)/f, \end{cases} .$$

Discussed in section III.B.

A summary of the imaging processing steps is detailed by the following algorithm

BALANCE CV Algorithm:

```
$count \leftarrow 1000$ //frame count
$ballarea \leftarrow pi * r_{ball}^2$
$LEDarea \leftarrow pi * r_{led}^2$

$xrange_{ball} \leftarrow 1\ to\ 640$
$yrange_{ball} \leftarrow 1\ to\ 480$
//set search area for beam to left side LED
$xrange_{beam} \leftarrow 1\ to\ 100$
$yrange_{beam} \leftarrow 1\ to\ 480$

$\boldsymbol{for}\ i\ from\ 1\ \boldsymbol{to}\ count$
        //get image and threshold to two levels
        $img \leftarrow grabframe\ from\ camera$
        $bimgLOW \leftarrow \boldsymbol{theshold}(img, 60\%)$
        $bimgHIGH \leftarrow \boldsymbol{theshold}(img, 90\%)$

        //resize image based on last known positions
        $ballsearchimage \leftarrow bimgLOW\ \boldsymbol{in}\ (xrange_{ball}, yrange_{ball})$
        $beamsearchimage \leftarrow bimgLOW\ \boldsymbol{in}\ (xrange_{beam}, yrange_{beam})$

        $ballpixelcenter \leftarrow$
                $\boldsymbol{findcirclewitharea}(ballsearchimage, ballarea)$
        $beampixelcenter \leftarrow$
                $\boldsymbol{findcirclewitharea}(ballsearchimage, LEDarea)$

        //update search range; Search area currently set to 200x200
        $xrange_{ball} \leftarrow (ballpixelcenter.x - 100)$
                $\boldsymbol{to}\ (ballpixelcenter.x + 100)$
        $yrange_{ball} \leftarrow (ballpixelcenter.y - 100)$
                $\boldsymbol{to}\ (ballpixelcenter.y + 100)$

        $yrange_{beam} \leftarrow (beampixelcenter.y$
                $- 100)\ \boldsymbol{to}\ (beampixelcenter.y + 100)$

        //calculate ball and beam positions; add offset based on the resized
        //image.
        $ballposition \leftarrow \boldsymbol{pixels2meters}(ballpixelcenter +$
                $(xrange_{ball}.min, yrange_{ball}.min))$
        $beamposition \leftarrow \boldsymbol{pixels2meters}(beampixelcenter +$
                $(xrange_{beam}.min, yrange_{beam}.min))$
$\boldsymbol{loop}$
end
```

D. Software Implementation

The design was implemented in a MATLAB class from the models and controller values. The BALANCE class takes a raspberry pi object and pin numbers as input. Initially, the class sets up pins on the raspberry pi, initializes the camera, calibrates the system, and prepares the PID controllers.

Setting up the pins involves setting the PWM and direction signals for the motor. To initialize the camera, the class captures several frames and insures successful camera operation. The calibration process of the BALANCE system finds the key points of the beam and ball and sets some initial constants. The program captures frames from the camera, processes the images, and attempts to locate the center reference point LED and the initial ball position. This process is repeated until canceled by the user or until these reference points are found. These reference points are used to calculate the angle of the beam and initialize the PIDs.

The PIDs are implemented from the difference equation from the inverse Z transforms of the simulated controllers. The gains are set from the properties of the class so that they may be adjusted by the main program. The PID controller methods from the BALANCE class must be called each time an update is required. These functions use an internal timer for the derivative and integral parts of the controllers. This timer can be paused if the control sequence is paused. When the PID signals are calculated, a method can be called by the user to run the motor with the calculated signal.

Other methods of the class allow for stopping and starting the motor, flushing the camera (see section VIII.A.4) for explanation), and viewing processed frames.

The main program for running the ball positioning procedure begins by establishing a connection to the raspberry pi and initializing the BALANCE class object. The BALANCE object is calibrated and the initial PID signals set. Next, the program loops for a set direction and uses the PID calculators and motor control methods to attempt to move the ball to the objective position.

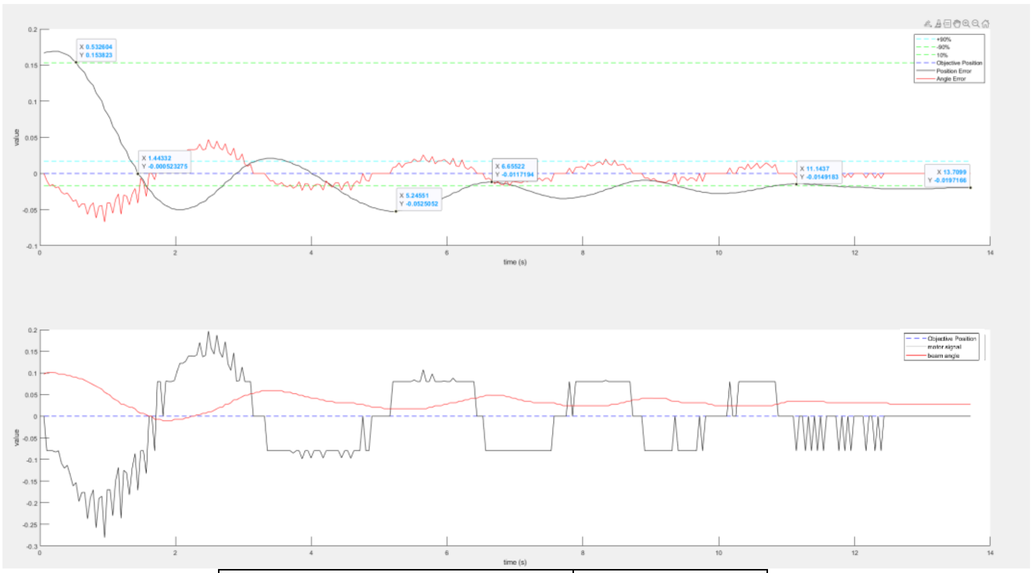
The full code listing can be found in Appendix B.

VII. Testing Results and Discussion

The BALANCE system was tested by placing the ball on one end of the beam and setting the system to balance the ball in the center. The beam was set to an unknown angle at the beginning of each test; however, this appeared not to affect the results.

A. *Standard Response*

Although better responses were recorded, this plot demonstrates the most common response of the system. BALANCE generally exceed the rise time requirement but has a slow settling time and often maintains a steady state error above 10%.

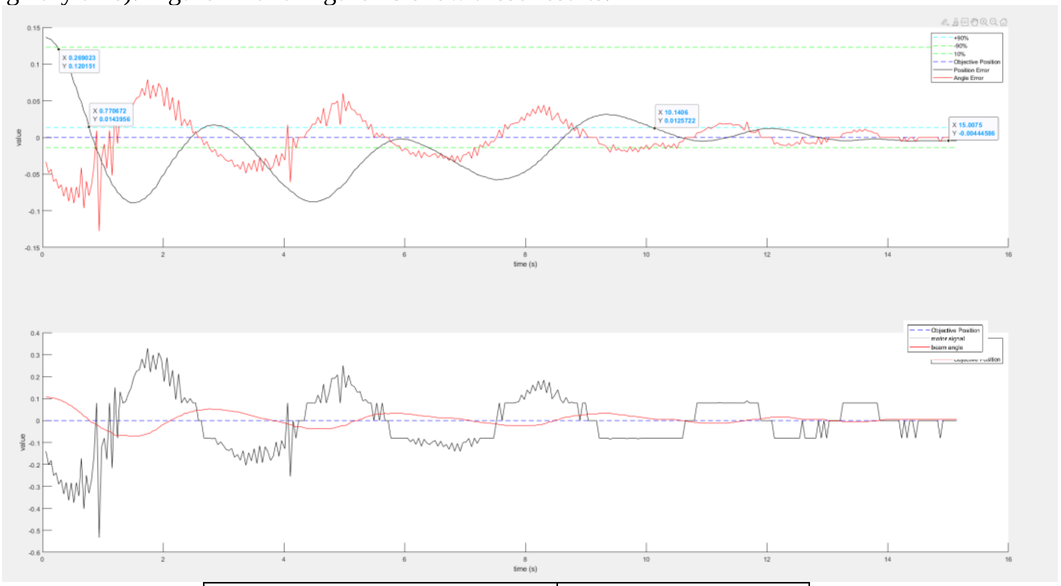


Result	**Value**
Rise Time	0.818 s
Settling Time (Oscillations <10%)	6.65 s
Final Value	-0.019 m
Steady State Error	0.019m

Figure 13: Standard step response of the BALANCE system

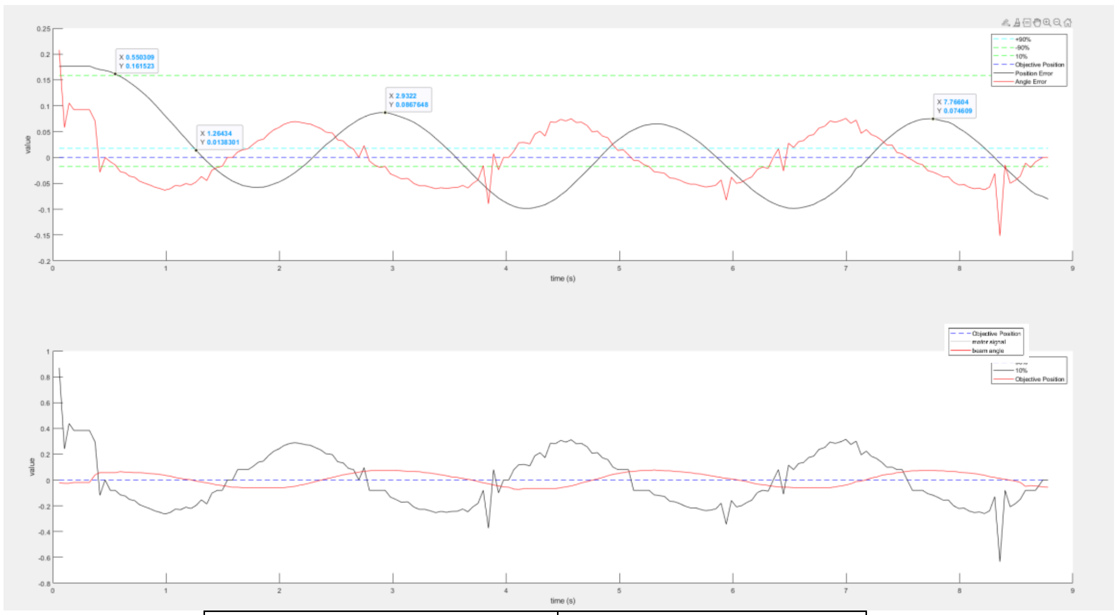
B. *20Hz and 15Hz Response*

The program was limited using software to demonstrate the effect of the control rate. Control rates above 20Hz converge and were stable. Any response below 15Hz became unstable. At 15Hz the BALANCE system exhibits steady state oscillations (pole on the imaginary axis). Figure 14 and Figure 15 show these results.



Result	**Value**
Rise Time	0.51 s
Settling Time	10.14 s
Final Value	-0.004 m
Steady State Error	0.004 m

Figure 14: Standard step response of the BALANCE system at 20Hz control rate

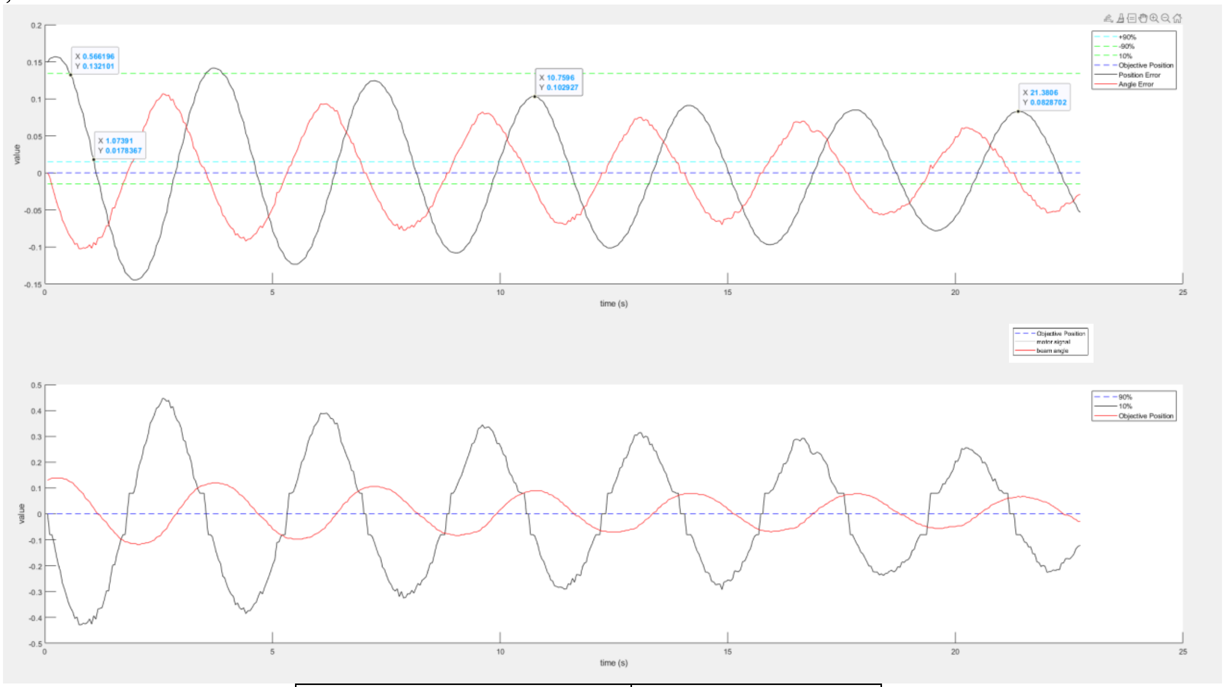


Result	Value
Rise Time	0.51 s
Settling Time (Oscillations <10%)	NA
Final Value	NA
Steady State Error	0.0746 m

Figure 15: Standard step response of the BALANCE system at 15Hz control rate

C. Results without acceleration dampening term

The acceleration term of the ball and beam PID was deactivated. The results show a very slowly converging system; however, it is stable.

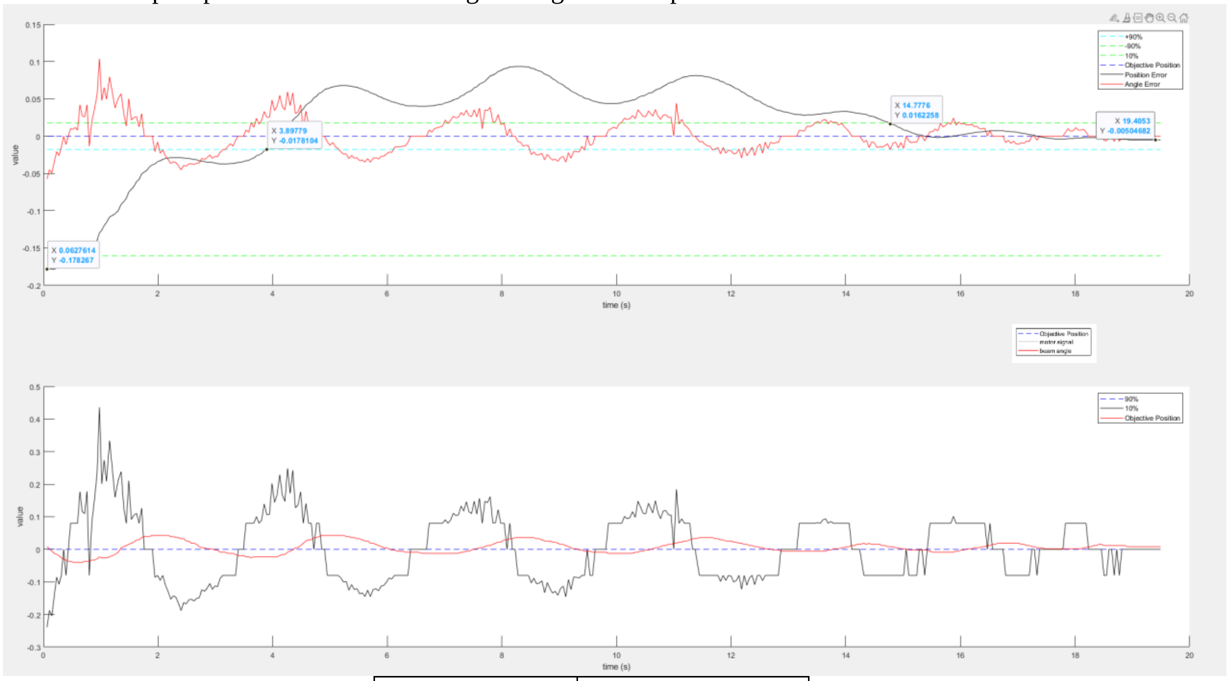


Result	Value
Rise Time	0.504 s
Settling Time (Oscillations <10%)	‘Attenuating Slowly’
Final Value	NA
Steady State Error	‘Decreasing’

Figure 16: Standard step response of the BALANCE system without acceleration term

D. *Increasing Kd for Ball and Beam PID*

The derivative gain was adjusted in an attempt to achieve less overshoot. Figure 17 shows the best results. These results do not show a desired step response because of the long settling time and poor rise time.

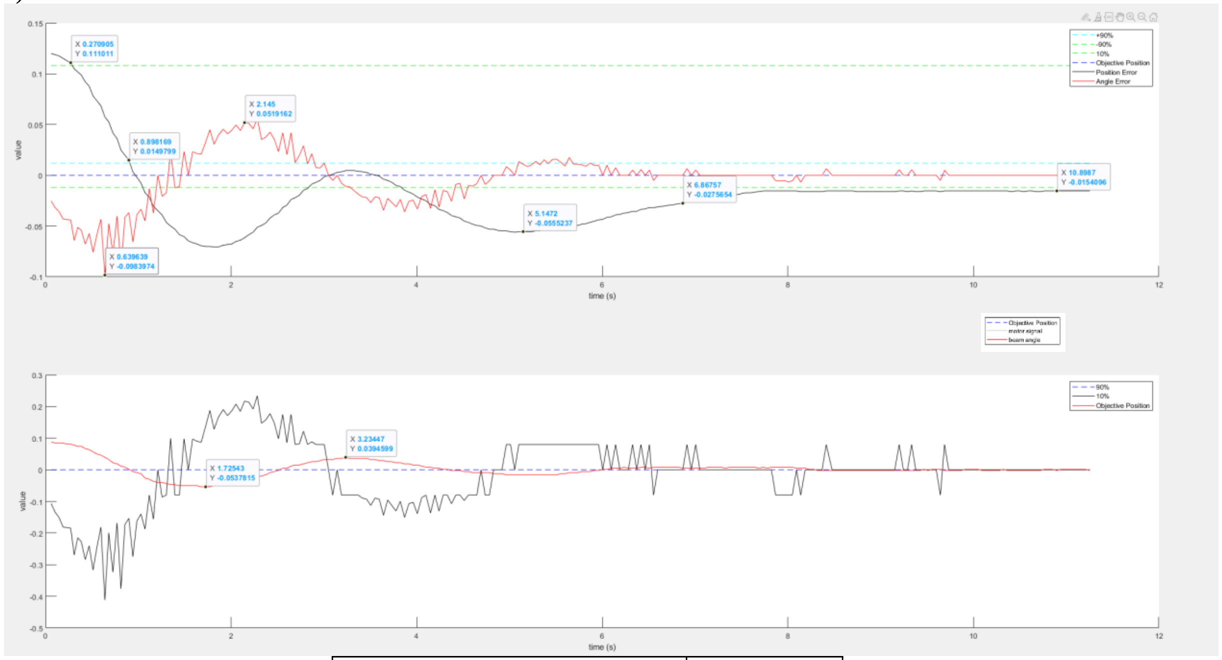


Result	Value
Rise Time	3.8343 s
Settling Time	14.77 s
Final Value	-0.005 m
Steady State Error	0.005 m

Figure 17: Standard step response of the BALANCE system with increased derivative gain.

E. *Results with Loose Gearing*

A loose motor gear was place in the assembly. The BALANCE system failed to overcome the steady state error. Additionally, there was increased lag from the measured angle and the angle signal from the PID controller (red signals in Figure 18).

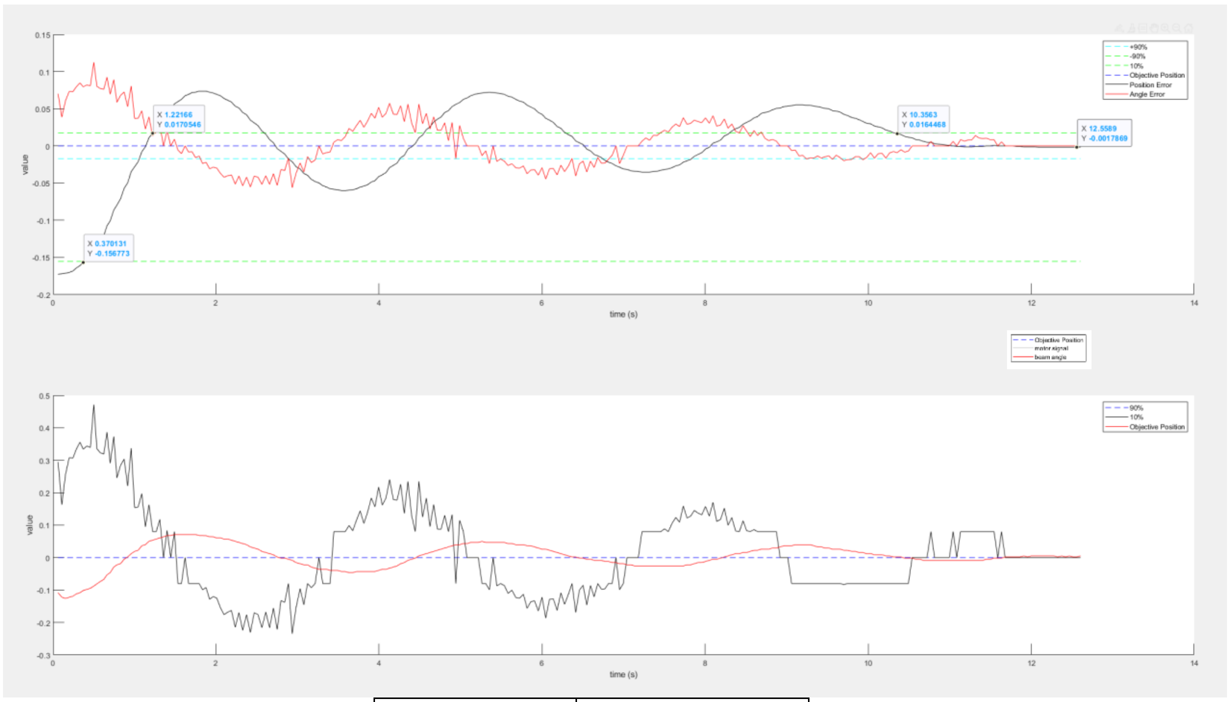


Result	Value
Rise Time	0.838 s
Settling Time (Oscillations <10%)	6.50 s
Final Value	-0.015 m
Steady State Error	0.015 m

Figure 18: Standard step response of the BALANCE system with loose gearing

F. *Increase Derivating Filtering by 25%*

The derivative filter was increased by 25 percent. The system response became less aggressive but more consistently settled to within 10%. Figure 19 shows the average step response with the increased derivative filtering. The lag of the beam angle to PID signal also increased.



Result	Value
Rise Time	0.85 s
Settling Time	10.35 s
Final Value	-0.0017 m
Steady State Error	0.0017 m

Figure 19: Standard step response of the BALANCE system with increased filtering

G. *Tracking and Locating of the Ball and Beam*

The ball and beam pixel locations are plotted in Figure 20. The smoothness of the lines demonstrates the precision and variance of the CV system.

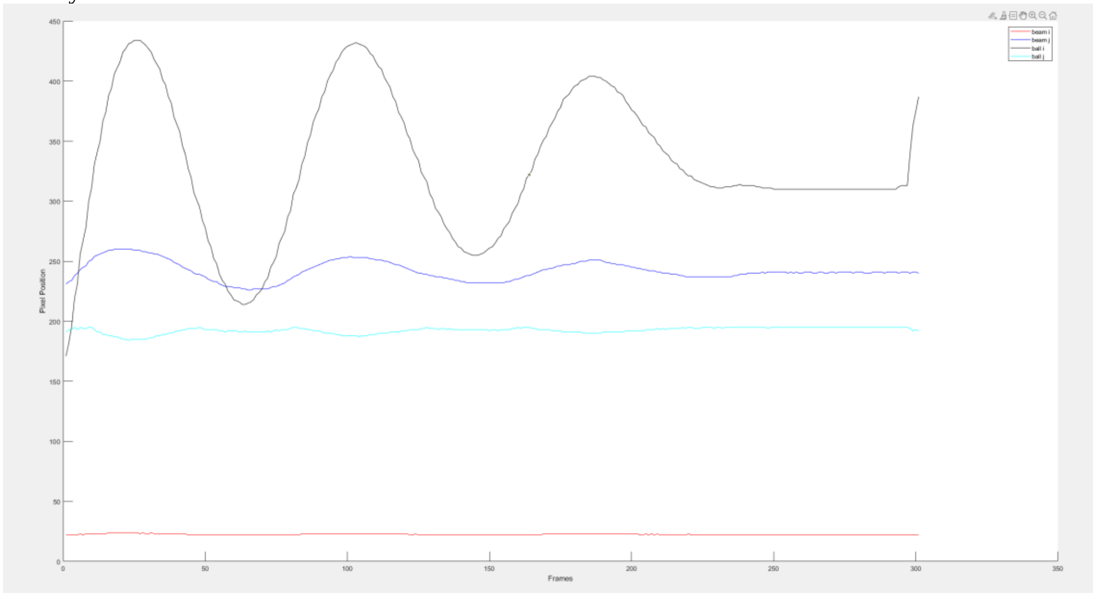


Figure 20: Ball and beam pixel locations

VIII. Discussion of Results

With all implementations of the BALANCE system, the worst control results were the steady state error and settling time. The steady state error occurred due to the dead zones, low integral gain of the ball and beam PID, and incorrect camera orientation. Higher integral gains produced unstable or undesirable step responses. Errors with the camera calibration or leveling increased the steady state error.

The results in Figure 19 show that a slower settling time led to higher precision and a more desirable final value; however, this passive response increased the settling time significantly

The acceleration term of the ball and beam PID proved necessary to dampen the system to steady state within a reasonable time. Without it, long attenuated oscillations are produced by the control signal.

There is a significant amount of high frequency signals in the PID outputs. These could be mitigated by increasing the derivative term; however, tests with any higher filtering proved unstable because the response was too passive.

The CV component of the BALANCE system limited the control rate to under 30Hz. As seen from Figure 14 and Figure 15, the stability of the system is highly dependent on this rate. Currently, if the proportional or derivative gain is increased any higher, the system becomes unstable or ineffective. If the rate was increased to higher values, perhaps these gains could be increased and allow for a faster settling time with less overshoot.

A. *Errors and Issues*

Several issues and systems errors were encountered during development.

1) *Camera Leveling Error*

Currently, the camera can rotate and leveled relative the ball and beam. With the current software implementation, the camera frame is assumed to be level with the ground. Consequently, if the camera is off level, the control signals will contain error. Testing of the effect of the camera's orientation showed that even slight misalignment caused large over or undershoot depending on the direction of the angle offset and the balls approach path.

2) *Motor Dead Zone*

It was discovered that a minimum PWM duty cycle of 8% was required for any motion of the motor. This created several problems for the control system. Since the ball and beam system is quite sensitive to changes in angle, when the ball position error became very small, a proportionally small control signal (beam angle) was needed. Consequently, the control signal was often smaller than the minimum motor signal and created an extended steady error as the integral term 'built up'.

3) *Gear Backlash*

The 3D printed gears have a large tolerance. The mesh of the 2 gears connecting the motor shaft and beam allow for backlash which corresponded to approximately 0.5 degrees. This backlash created a certain amount of lag when the motor direction changed. This was compensated for by with addition of an acceleration term in ball and beam controller. The term was added to the control signal with a small gain. This signal gives a small positive boost to the motor signal when the ball begins to change direction (when its acceleration is highest). This helps the system overcome the lag cause by the backlash.

4) *Undesired frame buffering*

During testing, on several occasions the code was stepped through line by line. During these debugging sessions, the captured image was displayed on each iteration of the loop. It was observed that when the image view changed, it would take several calls to the camera image capture function before the change was observed in the stored image. This suggests that the frames are buffered somewhere in the system before the function to capture the images is called. This was not confirmed to occur in regular run time; however, if this frame buffering occurs in the real time system, it may introduce additional lag and decreasing control performance.

IX. Future Development

Several improvements to the BALANCE system should be prioritized in future development. These should improve the accuracy and speed of control and help demonstrate different types of visual servoing.

1) *Improved gearing and switch to stepper motor*

The gearing backlash and the minimum motor speed increased the difficulty of controlling the system. Printing or acquiring a higher quality gear set would minimize the gear backlash. A DC motor was originally chosen for the higher resolution speed and position control; however, since the rate of the control system is so low (<30hz), this advantage was not experienced by the system. A stepper motor would provide more consistent angle position at an adequate speed. This would simplify the control loop and allow for easier experimentation with the VS portion of the system.

2) *Pose estimator for beam.*

If sufficient reference points were positioned on the beam, VS pose estimation could be conducted to determine its orientation. Then, if a universal, gravity aligned reference frame was established, the beam angle could be determined with respect to the horizontal regardless of the camera orientation – provided the beam and reference coordinates were within the cameras reference frame. This type of VS would be considered position based or a PBVS system.

3) *Run embedded software.*

Although not confirmed, it was observed during development that running the control software on an external computer (rather than directly on the raspberry pi) greatly increased the time of certain operations including writing to a pin on the Pi board. On occasion the time for writing the duty cycle and pin direction took upwards of 0.02s (in the same order as the computer vision component). If the system was run independently on the raspberry pi, the control frequency may noticeably improve.

X. Conclusion

The BALANCE system was created to demonstrate the use of visual servoing in a realized system. The BALANCE system consists of a ball and beam control system which uses a single camera to generate feedback signals. The device consists mostly of 3D printed components and uses a raspberry pi as the main controller. Currently, the device depends on an external computer running MATLAB; however, there is potential for the device to become completely independent through code generation. While not all the requirements set for this project were met, enough were satisfied to successfully position the ball at the desired beam location.

The BALANCE system was tested by placing the ball on the beam and setting the system to balance the ball in the center. The results showed a satisfactory rise time for all tests. The system often exhibited steady state error due to the motor and beam dead zone, the low integral gain, and poor camera orientation. Steady state error occurred less with more passive tuning, but this increased the settling time significantly. The settling time of the system was significant especially when the rate of control was low (<20Hz). Decreasing the rate below 15 Hz made the system unstable. The highest time cost in the system was the computer vision processing time and the pin writing time. The high pin writing time was due to the communication delay between the connected computer and the raspberry pi.

Several issues were noted while developing the system. The largest impact on the control system was the motor dead zone and poor orientation of the camera. Thus, it was suggested that a pose estimator for the beam be created such that the beam angle can be accurately calculated regardless of the camera’s orientation. Additionally, for unknown reasons, image frames were buffered somewhere in the software implementation. This buffering may be causing additional lag between the control signal and CV feedback signals. Consequently, future development should seek to avoid any such buffering by making the system fully embedded and run exclusively on the raspberry pi controller. To increase understanding of visual servoing systems, future project should seek to add more complexity and degrees of freedom.

XI. References

[1] M. Ebert, "Review of Visual Servoing Fundamentals," dept. of Electrical and Computer Engineering, University of Victoria, 2022.

[2] G. Palmieri, M. Palpacelli and M. Battistelli, "A Comparison between Position-Based and Image-Based Dynamic Visual Servoings in the Control of a Translating Parallel Manipulator," *Journal of Robotics,* vol. 2012, p. 11, 2012.

[3] Y. Peng, D. Jivani and R. Radke, "Comparing Position- and Image-Based Visual Servoing for Robotic Assembly of Large Structures," Department of Electrical, Computer, and Systems Engineering,Rensselaer Polytechnic Institute, New York.

[4] N. Lai, Y. Chen, J. Liang and e. al, "Image Dynamics-Based Visual Servo Control or Unmanned Aerial Manipulatorl With a Virtual Camera," *IEEE Robotics & Automation Society,* 2022.

[5] M. Ebert, "Survey and Review of Computer Vision Theory and Methods," Department of Electrical and Computer Engineering, University of Victoria, 2022.

[6] K. Nowopolski, "Ball-and-beam laboratory system controlled by Simulink model through dedicated microcontrolled-Matlab data exchange protocol," Poznań University of Technology, Computer Applications in Electrical Engineering.

[7] The University of Michagan, Carnegie Mellon University, "Example: Modeling DC Motor Position," [Online]. Available: https://www3.diism.unisi.it/~control/ctm/examples/motor2/motor.html. [Accessed 20 06 2022].

[8] ME 380, "Gear Train Analysis," 1 September 2004. [Online]. Available: http://www.me.unm.edu/~starr/teaching/me380/gears.pdf. [Accessed 14 AUG 2022].

[9] University of Michigan, Carnegie Mellon University, University of Detriot Mercy, "Control Tutorials for Matlab and Simulink," [Online]. Available: https://ctms.engin.umich.edu/CTMS/index.php?example=BallBeam&section=SystemModeling. [Accessed 20 June 2022].

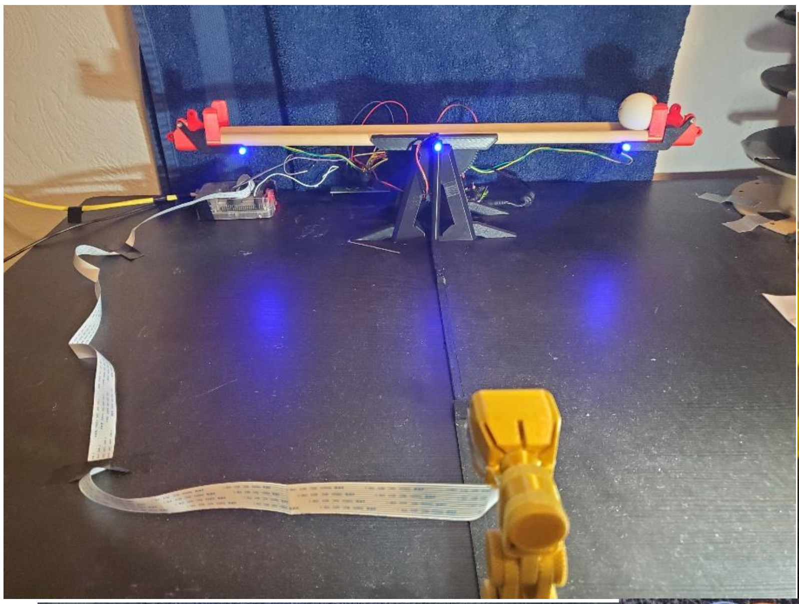
[10] F. Chaumette and S. Hutchinson, "Visual Servo Control Part 1: Basic Approaches," *IEEE Robotics & Automation Magazine,* pp. 82-90, December 2006.

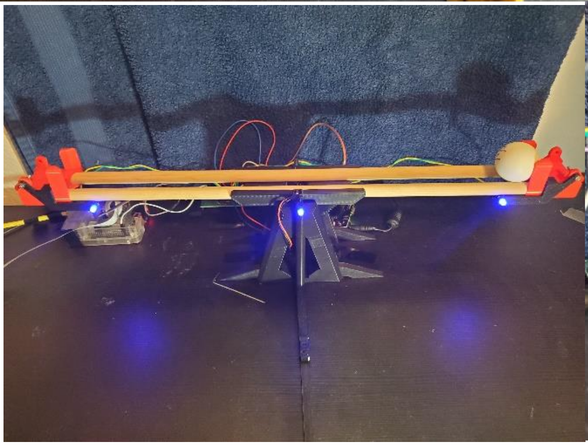
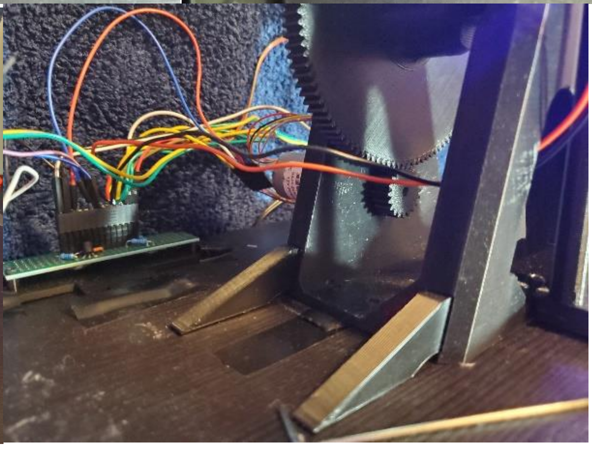
[11] Raspberry Pi, "Getting started with the Camera Module," Rapsberry Pi Foundation, [Online]. Available: https://projects.raspberrypi.org/en/projects/getting-started-with-picamera. [Accessed 15 AUG 2022].

[12] G. Pavlakos, X. Zhou, A. Chan, K. Derpanis and K. Daniilidis, "6-DoF object from Semantic Keypoints," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2017.

APPENDIX A: REALIZED BALANCE SYSTEM

Videos of the implemented BALANCE system can be found here: BALANCE_Video_Examples



APPENDIX B: MATLAB SOFTWARE PROGRAM

A complete repository of the BALANCE system can be found at:
[ebertmx/BALANCE_CS: Control and Simulation of the BALANCE system. (github.com)]

BALANCEControlSystem Class

About BALANCE Control System

```
%Matthew Ebert
%ebertmx@gmail.com

% Created
% 2022-AUG-16
% For
% University of Victoria ECE 490
% Visual Servoing and Computer Vision Control Systems
% Supervisor: Dr Capson
```

Description

This class implements models and provides tool to control the BALANCE system -- a ball and beam control model. An IBVS VS model is used to calculate error signals for the controller signals. The class is initiated with a raspberry pi object and the pins connected toa motor PWM signal and direction. Its methods include PID calculations, Control signal motor control, direct motor control, image capturing, and image processing.

```
%system -- a ball and beam control model.
% An IBVS VS model is used to calculate error signals for the controller
% signals.
%The class is initiated with a raspbery pi object and the pins connected to
%a motor PWM signal and direction.
%Its methods include PID calculations, Control signal motor control, direct
%motor control, image capturing, and image processing.

classdef BALANCEControlSystem <handle %#codegen
    properties (Access = public)
        %Constants and properties values
        %Pixel refers to image pixel coordinate
        %Point refers to image plane coordinate
        %Position refers to real world unit

        %Hardware configuration values
        cam, rpi, DIRpin, PWMpin;

        %Variables for ball, beam, and reference point values
        ballPixel, ballPoint, ballPosition, ballObjectivePosition;
        beamPixels, beamPoints,beamAngle;
        CenterReferencePixels, CenterReferencePoint, ballRadius;

        %image variables
        rawFrame, bimgLOW, bimgHIGH;
        lowThresh = 0.6;
        highThresh = 0.9;

        % Search range for ball and beam relocation
        ballDetectionRadiusRange = [20,40];
        beamSearchRange = 50;
        ballSearchRange = 150;

        %Real world offsets for BALANCE system
        Z = 0.5;
        LED_HYPO = 0.24309; %m
        LED_lineoffset = 9.85/1000; %m
        LED_refoffset = 8.3/1000 ; %m

        %Intrinsic Camera Paramters (Calibrated)
        px = 625.21687001741327;
        py = 623.78140876507746;
        u0 = 319.96647337485342;
        v0 = 243.30511332809766;

        %Ball and beam control system signals, gains, and limits
        positionError = zeros(1,2);
        thetaError = zeros(1,2);
        bKp = 0.2105;
        bKi = 0.05;
        bKd = 0.4771;
        bKa = 0.08;
        alphaFilter = 0.1164;
        bKc = 0.75;

        P_err = zeros(1,2);
        I_err = zeros(1,2);
        D_err = zeros(1,2);
        A_err = zeros(1,2);
        Dfilter_err = zeros(1,2);


        thetaSignal = 0;
        maxBeamAngle = 0.2;
        minBeamAngle = -0.2;


        %Actuator control system signals, gains, and limits
        mKp = 4.125;
        % integral disabled
        mKi = 0.0038528*(0);
        mKd = 0.27;

        mKc = 1;
        alphaFilter_sig = 0.01;

        P_sig = zeros(1,2);
        I_sig = zeros(1,2);
        D_sig = zeros(1,2);
        Dfilter_sig = zeros(1,2);

        actuatorControlSignal = 0, motorDuty, motorDir;
        maxABSMotorSig= 0.9
        minABSMotorSig = 0.08

        %Clock signals for derivative and interval PID calculations
        thetaclockSig = tic;
        controlclockSig = tic;

        %angle offset to correct for camera frame misalignment
        angleOffset = 0.005;
    end

    methods
        %          function delete(~)
        %          end
```

Initialization

```
       function obj = BALANCEControlSystem (setrpi, setPWMpin, setDIRpin)

            %Check if parameters are properly set
            if(isempty(setrpi) || isempty(setDIRpin)||isempty(setPWMpin))
                disp("need rpi and pin")
            else
                obj.rpi = setrpi;
                obj.DIRpin = setDIRpin;
                obj.PWMpin = setPWMpin;
            end
            %set the objective position to default (middle of beam)
            obj.ballObjectivePosition = [0;0];
        end
```

Hardware Configuration

```
     function status = SetUpHardware(obj)
            if(obj.ConfigureMotor() && obj.ConfigureCamera())
                status = true;
            else
                status = false;
            end
        end
        %configure the motor
        function status = ConfigureMotor(obj)
            %set PWM pin and frequency
            configurePin(obj.rpi, obj.PWMpin, 'PWM');
            writePWMDutyCycle(obj.rpi, obj.PWMpin, 0.0);
            writePWMFrequency(obj.rpi, obj.PWMpin, 8000);

            %set direction pin
            configurePin(obj.rpi,obj.DIRpin, 'DigitalOutput');
            writeDigitalPin(obj.rpi,obj.DIRpin,1);

            status = true;
        end

        %configure the camera
        function status = ConfigureCamera(obj)
            obj.cam = cameraboard(obj.rpi,'Resolution','640x480');
            %flush initial camera frames on start up
            for i = 1:10
                I = snapshot(obj.cam);
            end
            if( isempty(I))
                status = false;
            else
                status = true;
            end
        end
```

Image Acquisition, Processing, Object Detection, and Visual Servoing

```
%Get and process images from the camera
        function status = GrabFrames(obj)
            obj.rawFrame = snapshot(obj.cam);
            if(~isempty(obj.rawFrame))
                img = rgb2gray(obj.rawFrame);
                img = imgaussfilt(img,2);
                %threshhold the gray scale image to isolate the ball
                obj.bimgLOW = imbinarize(img,obj.lowThresh);
                %threshhold the gray scale image to isolate the LED points
                obj.bimgHIGH = imbinarize(img,obj.highThresh);
                status = true;
            else
                status = false;
            end
        end

        %Find calibration points in image plane
        function status = CalibrateImage(obj)
            %Set range of search for middle reference point
            cy1 = 220;
            cy2 = 280;
            cx1 = 280;
            cx2 = 360;
            obj.GrabFrames();

            if(~isempty(obj.bimgHIGH))
                %Conduct blob search on binary image
                simg = obj.bimgHIGH(cy1:cy2,cx1:cx2);
                sc   = regionprops(simg, 'Centroid','Area');

                if (~isempty(sc))
                    %find blob of correct area
                    scdata = struct2table(sc);
                    id = find(scdata.Area < 100 & scdata.Area>30);
                    if(~isempty(id))
                        center = scdata.Centroid(id(1),:);
                    else
                        status = false;
                        return;
                    end
                end
                %Correct for image resizing
                obj.CenterReferencePixels = [center(1)+(cx1-1), center(2)+(cy1-1)];
                obj.CenterReferencePoint = obj.pixel2point(obj.CenterReferencePixels());
                status = true;
                return;
            end
            status = false;
        end


        %find feature which dictate ball position and beam angle
        function [statusball, statusbeam] = LocateFeatures(obj)
            %             statusball = false;
            statusbeam = false;
            %Find the center of the ball
            if(obj.GrabFrames())
                if(~isempty(obj.ballPixel()))
                    %resize image based on previous location of ball
                    ballx1 =  min(max(uint32(obj.ballPixel(1) - obj.ballSearchRange),1),640);
                    ballx2 = min(max(uint32(obj.ballPixel(1)  + obj.ballSearchRange),1),640);
                    bally1 =min(max(uint32(obj.ballPixel(2)  - obj.ballSearchRange),1),480);
                    bally2 =min(max(uint32(obj.ballPixel(2) + obj.ballSearchRange),1),480);
                    searchimage = obj.bimgLOW(bally1:bally2,ballx1:ballx2);
                else
                    ballx1 = 0;
                    bally1 = 0;
                    searchimage = obj.bimgLOW;
                end
                [centers, radii] = imfindcircles(searchimage,obj.ballDetectionRadiusRange, ...
                    'ObjectPolarity','bright');

                if(size(centers,2)>=1)
                    %correct for image resize
                    obj.ballPixel(1,1) = centers(1,1)+ballx1;
                    obj.ballPixel(1,2) = centers(1,2) +bally1;
                    obj.ballRadius = radii(1);
                    statusball = true;
                else
                    statusball=false;
                    return;
                end

                %Find beam angle

                if(~isempty(obj.beamPoints()))
                    %resize image based on the previous position of the beam
                    beamx1 = min(max(uint32(obj.beamPixels(1)-obj.beamSearchRange),1),640);
                    beamx2 = min(max(uint32(obj.beamPixels(1) +obj.beamSearchRange),1),640);
                    beamy1 =min(max(uint32(obj.beamPixels(2) - obj.beamSearchRange),1),480);
                    beamy2 =min(max(uint32(obj.beamPixels(2) + obj.beamSearchRange),1),480);
                    left = obj.bimgHIGH (beamy1:beamy2, beamx1:beamx2);
                else
                    beamx1 = 0;
                    beamy1 = 0;
                    left = obj.bimgHIGH(1:480,1:100);
                end

                ls  = regionprops(left, 'Centroid', 'Area');
                if (~isempty(ls))
                    %blob search for LED of correct area
                    lsdata = struct2table(ls);
                    idb = find(lsdata.Area < 80 & lsdata.Area>10);
                    if(~isempty(idb))
                        centroidLED = lsdata.Centroid(idb(1),:);
                    else
                        statusbeam = false;
                        return;
                    end
                    %correct for image resize
                    obj.beamPixels(1,1) =centroidLED(1) + beamx1-1;
                    obj.beamPixels(1,2) =centroidLED(2) + beamy1-1;
                    obj.beamPoints(1,:) = obj.pixel2point(obj.beamPixels(1,:));

                    %Calculate the beam angle
                    ytemp = (obj.beamPoints(1,2) - (obj.LED_lineoffset)) -
obj.CenterReferencePoint(1,2);
                    xtemp = -1 * (obj.beamPoints(1,1)-obj.CenterReferencePoint(1,1));
                    obj.beamAngle = obj.angleOffset + atan2(ytemp, xtemp);

                    statusbeam = true;
                    return;
                end
            end
            statusbeam = false;
            statusball = false;
        end

        %Use IBVS to convert from pixels to image plane in meters
        function meter = pixel2point(obj, pixel)
            meter = zeros(1,2);
            meter(1) = (pixel(1) - obj.u0)/obj.px * obj.Z;
            meter(2) = (pixel(2) - obj.v0)/obj.py * obj.Z;
        end
```

Control Systems

```
%Get and process images from the camera
        function status = GrabFrames(obj)
            obj.rawFrame = snapshot(obj.cam);
            if(~isempty(obj.rawFrame))
                img = rgb2gray(obj.rawFrame);
                img = imgaussfilt(img,2);
                %threshhold the gray scale image to isolate the ball
                obj.bimgLOW = imbinarize(img,obj.lowThresh);
                %threshhold the gray scale image to isolate the LED points
                obj.bimgHIGH = imbinarize(img,obj.highThresh);
                status = true;
            else
                status = false;
            end
        end

        %Find calibration points in image plane
        function status = CalibrateImage(obj)
            %Set range of search for middle reference point
            cy1 = 220;
            cy2 = 280;
            cx1 = 280;
            cx2 = 360;
            obj.GrabFrames();

            if(~isempty(obj.bimgHIGH))
                %Conduct blob search on binary image
                simg = obj.bimgHIGH(cy1:cy2,cx1:cx2);
                sc  = regionprops(simg, 'Centroid','Area');

                if (~isempty(sc))
                    %find blob of correct area
                    scdata = struct2table(sc);
                    id = find(scdata.Area < 100 & scdata.Area>30);
                    if(~isempty(id))
                        center = scdata.Centroid(id(1),:);
                    else
                        status = false;
                        return;
                    end
                end
                %Correct for image resizing
                obj.CenterReferencePixels = [center(1)+(cx1-1), center(2)+(cy1-1)];
                obj.CenterReferencePoint = obj.pixel2point(obj.CenterReferencePixels());
                status = true;
                return;
            end
            status = false;
        end


        %find feature which dictate ball position and beam angle
        function [statusball, statusbeam] = LocateFeatures(obj)
            %             statusball = false;
            statusbeam = false;

            %Find the center of the ball
            if(obj.GrabFrames())
                if(~isempty(obj.ballPixel()))
                    %resize image based on previous location of ball
                    ballx1 =  min(max(uint32(obj.ballPixel(1) - obj.ballSearchRange),1),640);
                    ballx2 = min(max(uint32(obj.ballPixel(1)  + obj.ballSearchRange),1),640);
                    bally1 =min(max(uint32(obj.ballPixel(2)  - obj.ballSearchRange),1),480);
                    bally2 =min(max(uint32(obj.ballPixel(2) + obj.ballSearchRange),1),480);
                    searchimage = obj.bimgLOW(bally1:bally2,ballx1:ballx2);
                else
                    ballx1 = 0;
                    bally1 = 0;
                    searchimage = obj.bimgLOW;
                end
                [centers, radii] = imfindcircles(searchimage,obj.ballDetectionRadiusRange, ...
                    'ObjectPolarity','bright');
                if(size(centers,2)>=1)
                    %correct for image resize
                    obj.ballPixel(1,1) = centers(1,1)+ballx1;
                    obj.ballPixel(1,2) = centers(1,2) +bally1;
                    obj.ballRadius = radii(1);
                    statusball = true;
                else
                    statusball=false;
                    return;
                end

                %Find beam angle

                if(~isempty(obj.beamPoints()))
                    %resize image based on the previous position of the beam
                    beamx1 = min(max(uint32(obj.beamPixels(1)-obj.beamSearchRange),1),640);
                    beamx2 = min(max(uint32(obj.beamPixels(1) +obj.beamSearchRange),1),640);
                    beamy1 =min(max(uint32(obj.beamPixels(2) - obj.beamSearchRange),1),480);
                    beamy2 =min(max(uint32(obj.beamPixels(2) + obj.beamSearchRange),1),480);
                    left = obj.bimgHIGH (beamy1:beamy2, beamx1:beamx2);
                else
                    beamx1 = 0;
                    beamy1 = 0;
                    left = obj.bimgHIGH(1:480,1:100);
                end

                ls  = regionprops(left, 'Centroid', 'Area');
                if (~isempty(ls))
                    %blob search for LED of correct area
                    lsdata = struct2table(ls);
                    idb = find(lsdata.Area < 80 & lsdata.Area>10);
                    if(~isempty(idb))
                        centroidLED = lsdata.Centroid(idb(1),:);
                    else
                        statusbeam = false;
                        return;
                    end
                    %correct for image resize
                    obj.beamPixels(1,1) =centroidLED(1) + beamx1-1;
                    obj.beamPixels(1,2) =centroidLED(2) + beamy1-1;
                    obj.beamPoints(1,:) = obj.pixel2point(obj.beamPixels(1,:));

                    %Calculate the beam angle
                    ytemp = (obj.beamPoints(1,2) - (obj.LED_lineoffset)) -
obj.CenterReferencePoint(1,2);
                    xtemp = -1 * (obj.beamPoints(1,1)-obj.CenterReferencePoint(1,1));
                    obj.beamAngle = obj.angleOffset + atan2(ytemp, xtemp);

                    statusbeam = true;
                    return;
                end
            end
            statusbeam = false;
            statusball = false;
        end

        %Use IBVS to convert from pixels to image plane in meters
        function meter = pixel2point(obj, pixel)
            meter = zeros(1,2);
            meter(1) = (pixel(1) - obj.u0)/obj.px * obj.Z;
            meter(2) = (pixel(2) - obj.v0)/obj.py * obj.Z;

        end
```

Basic Control

```
function RunMotorFor(obj,dir,speed, time)
            temp = tic;
            while toc(temp)<time
                writeDigitalPin(obj.rpi, obj.DIRpin, dir);
                writePWMDutyCycle(obj.rpi, obj.PWMpin, speed);
            end
            writePWMDutyCycle(obj.rpi, obj.PWMpin, 0);
        end


        function RunMotor(obj,dir,speed)
            writeDigitalPin(obj.rpi, obj.DIRpin, dir);
            writePWMDutyCycle(obj.rpi, obj.PWMpin, speed);
        end

        function StopMotor(obj)
            writePWMDutyCycle(obj.rpi, obj.PWMpin, 0);
        end

        function RunToAngle(obj, angle,speed)
            obj.LocateFeatures
            if (angle -obj.beamAngle)>0
                dir = 0;
            else
                dir =1;
            end
            while abs(obj.beamAngle()-angle)>0.05
                RunMotor(dir,speed)
            end
            obj.StopMotor();
        end

        function ShowImage(obj)
            imshow(obj.bimgLOW)
            hold on
            plot(obj.u0,obj.v0, 'b*');
            plot(obj.CenterReferencePixels(1),obj.CenterReferencePixels(2),'r*')
            viscircles(obj.ballPixel,26);
            viscircles(obj.beamPixels, 5);
            hold off
        end
    end

end
```

Main Controller Program

About BALANCE MAIN

```
%Matthew Ebert
%ebertmx@gmail.com

% Created
% 2022-AUG-16
% For
% University of Victoria ECE 490
% Visual Servoing and Computer Vision Control Systems
% Supervisor: Dr Capson

function MAIN_BALANCEController() %#codegen
```

Define Parameters and Plot Arrays

```
cycles = 1000;%number of frames to run system
%Raspberry Pi
rpi = raspi();
PWMpin = 13;
DIRpin = 6;

positionError = zeros(cycles,1);
thetaError = zeros(cycles,1);
beamAngle = zeros(cycles,1);
motorSignal = zeros(cycles,1);
PIDsig = zeros(cycles,3);
PIDerr = zeros(cycles,4);
ballPixels = zeros(cycles,2);
beamPixels = zeros(cycles,2);
sampletime = zeros(cycles,1);
pause(0.001);
```

BALANCE Controller Creation and Calibration

```
%create BALANCE system controller and set it up
Controller = BALANCEControlSystem(rpi,PWMpin,DIRpin);
Controller.SetUpHardware();

%Calibrate CV system
while(~Controller.CalibrateImage())
    disp("Calibrating");
    %     imshow(Controller.bimgLOW);
    pause(0.001)
end
```

Initialize PID

```
%Located the ball and set the objective of the control system
Controller.LocateFeatures();
Controller.SetObjective();

for i=1:15
    [statusball, statusbeam] = Controller.LocateFeatures();
    if(statusball && statusbeam)
        Controller.CalcPositionError();
        Controller.CalcThetaError();
        Controller.CalcMotorSignal();
    end
end

%Reset Integral Signal
Controller.I_err = [0,0];
Controller.I_sig = [0,0];
```

Main Program

```
%run program for the number of cycles
for i=1:cycles
    %fps timer start
    t = tic;

    %attempt to locate the ball and beam
    [statusball, statusbeam] = Controller.LocateFeatures();
    if(statusball && statusbeam) %if both the ball and beam is located
        %calculate PID signals
        Controller.CalcPositionError();
        Controller.CalcThetaError();
        Controller.CalcMotorSignal();

        %control the motor
        Controller.ControlMotor();

    elseif (~statusball)
        disp("Cant find Ball")
        Controller.StopMotor();
        Controller.DelayPID();
    elseif (~statusbeam)
        disp("Cant find Beam")
        Controller.StopMotor();
        Controller.DelayPID();
    end
    %needed to display image in realtime
    %pause(0.0001)


    %save controller data
    PIDsig(i,1) = Controller.P_sig(1);
    PIDsig(i,2) = Controller.I_sig(1);
    PIDsig(i,3) = Controller.D_sig(1);

    PIDerr(i,1) = Controller.bKp* Controller.P_err(1);
    PIDerr(i,2) = Controller.bKi* Controller.I_err(1);
    PIDerr(i,3) = Controller.bKd* Controller.D_err(1);
    PIDerr(i,4) = Controller.bKa* Controller.A_err(1);

    positionError(i) = Controller.positionError(1);
    thetaError(i) = Controller.thetaError(1);
    beamAngle(i) = Controller.beamAngle;
    motorSignal(i) = Controller.actuatorControlSignal(1);
    ballPixels(i,:) = Controller.ballPixel;
    beamPixels(i,:) = Controller.beamPixels;
    sampletime(i) = toc(t);

    %Used to limit control rate

    %     while(toc(t)<(1/15))
    %     continue;
    %
    %     end
end

%stop the motor
Controller.StopMotor();
end
```