FRANKFURT UNIVERSITY OF APPLIED SCIENCES

BALL BALANCING PLATFORM

Pawel Zakieta

TABLE OF CONTENTS

1 HARDWARE

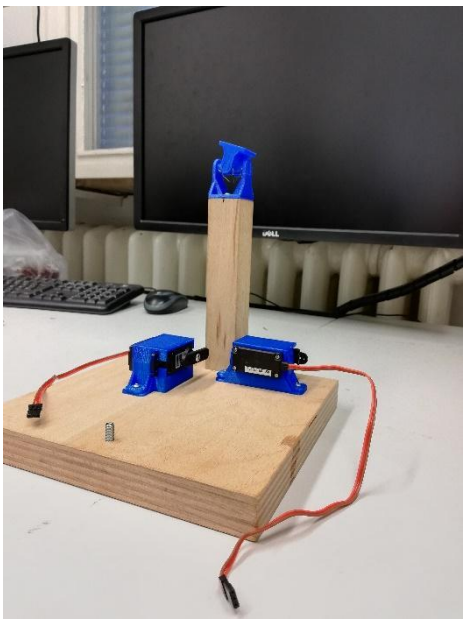
1.1 MAIN COMPONENTS OF THE ROBOT

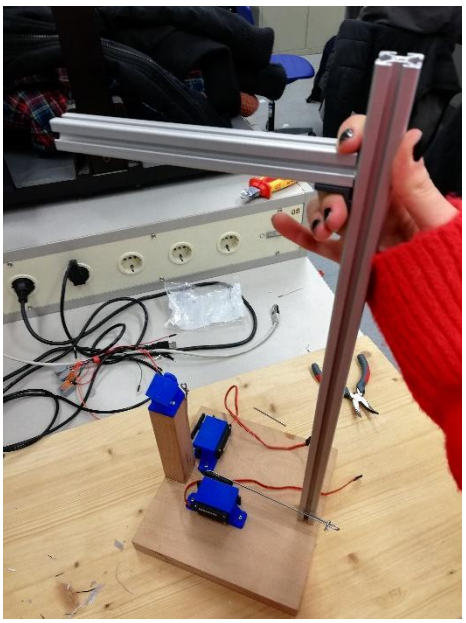
- Wooden base
- Wooden Column
- Joint printed in 3D
- Wooden platform
- Ball
- Metal column with L shape
- Camera Raspberry Pi
- Two servos
- STM32 microcontroller
- Power supply
- Screws and cables

The wooden base is where all the elements are supported. On one side of it there is a wooden column, at the end of which was mounted a universal joint (printed in 3D) to allow for platform movement. On the other side of the wooden base there is a metal column in the shape of an L, where we mounted Raspberry Pi as well as the camera. Two servos are mounted to the wooden base with 3D-printed sockets. The STM32 microcontroller is also in the wooden base with the power supply for the servos, which is just a 5 Volts, 2 Amps smartphone charger.

In our robot we can differentiate two parts: mechanical and electronic.

1.2 MECHANICAL PART





Procedure of the position of main mechanics parts

Our only requirement for the wooden base was that it had to be robust and heavy enough to ensure stability in case of rapid movements of the platform. We designed the platform in such a way that it provided enough room for eventual changes in the initial design.

What turned out to be quite a challenge to find a good material for were the linkages between servos and the platform- we required stiff, yet thin and long pieces that we could still bend without breaking them. We ended up using bicycle spokes. They have the perfect thickness and provide more than enough stiffness.

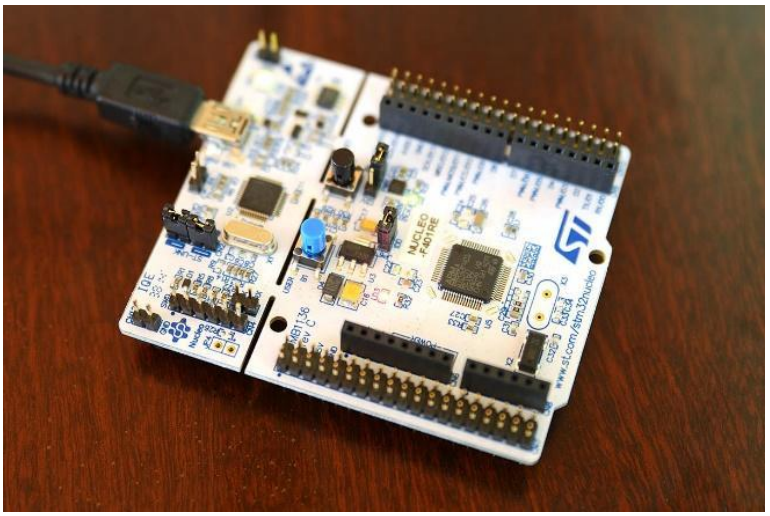
The 3D printed parts were designed using Fusion 360. Despite having some rough edges and requiring quite a lot of grinding, the parts do their job very well.

1.3 Electronic part

We used the following electronic parts:

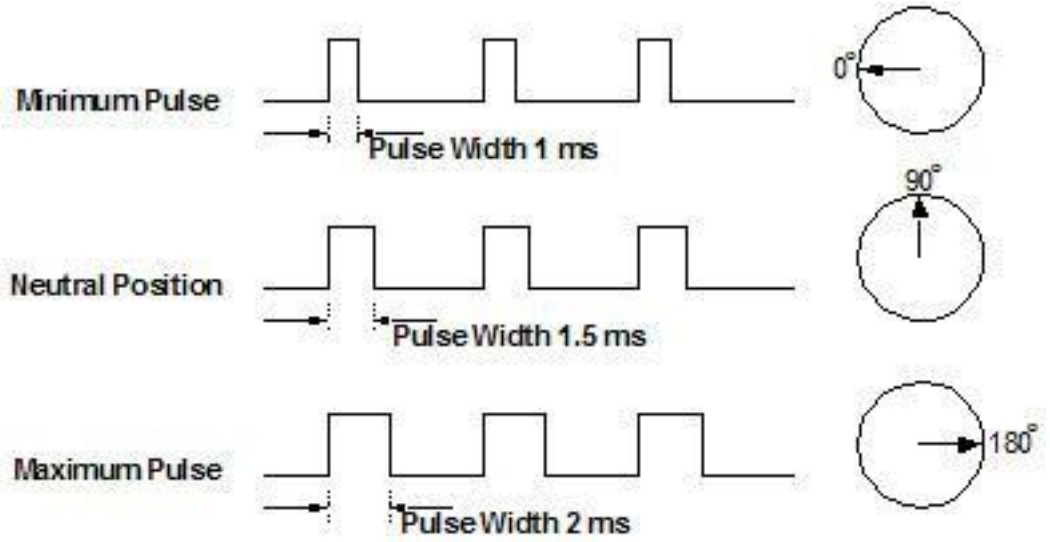
- PiCamera
- Raspberry Pi
- STM32F411RE
- Power supply
- 2 Servos
- Computer/laptop (for launching appropriate scripts on raspberry using ssh protocol)

PiCamera and Raspberry Pi



STM32 microcontroller

Servo



1.4 Run cycle overview

After running an appropriate script on Raspberry Pi, the camera is activated. The camera feeds a continuous stream of information to raspberry that processes the image. After all the processing we are ready to drive the servos. However, raspberry Pi doesn't provide a dedicated hardware for PWM generation. For that reason, we had to use the STM microcontroller. Raspberry sends the desired position of the servos to STM32 via UART using 2 bytes for every update (one for each servo) and then, using a dedicated timer, STM sends a PWM signal of the exact width. The 2 microcontrollers are connected using USB interface.

2 IMAGE PROCESSING

In order to work out the tilting of the surface and prevent the ball from falling off, the position of the ball must be constantly monitored. All the image processing operations are performed using functions provided by OpenCV library.

2.1 THROUGHPUT LIMITATIONS

The main requirement of the project is low latency- we want to have frequent measurements of the position of the ball without a significant delay. After spending some time with camera resolution, field of view and framerate it turned out that the maximum framerate we could achieve without dramatically decreasing the field of view was 40 frames per second at resolution of 360 x 480. After reading the image it is cropped to square proportions and resized to the resolution of 150 x 150 for faster processing.

2.2 DISTINGUISHING THE PIXELS OF SPECIFIC COLOR

First step of finding the center of the ball was to find all the pixels that "belong" to the ball. The ball we used is orange. The best way of detecting such pixels is by using HSV thresholding.

2.2.1 HSV Color Space

The HSV (Hue, Saturation, Value) model defines a color space in terms of three constituent components.

2.2.1.1 Hue

Hue is the color portion of the color model, expressed as a number from 0 to 360 degrees:

Color	Angle
Red	0-60
Yellow	60-120
Green	120-180
Cyan	180-240
Blue	240-300
Magenta	300-360

Orange color has a hue value between red and yellow. The acceptable range that worked the best was 20-60

2.2.1.2 Saturation

Saturation is the amount of "colorfulness" in the color, from 0 to 100 percent. Reducing the saturation toward zero to introduce more gray produces a faded effect. Sometimes, saturation is expressed in a range from just 0-1, where 0 is gray and 1 is a primary color. Since our ball is very highly saturated, the threshold is pretty high- we used the minimum value of 190 (out of 255)

2.2.1.3 Value (or Brightness)

Value works in conjunction with saturation and describes the brightness or intensity of the color, from 0-100 percent, where 0 is completely black, and 100 is the brightest and reveals the most color. Selecting an HSV color begins with picking one of the available hues, which is how most humans relate to color and then adjusting the shade and brightness values. Since the ball can be in shade and thus have low color value, the acceptable range of this parameter is quite wide- 10-255

2.3 Filtering the image

Since we are using camera input, we have to deal with the noise- not all detected pixels actually belong to the ball and vice versa- some pixels that do belong to the ball might be out of range of the thresholding. Two main techniques are used in order to reduce the noise.

2.3.1 Erosion

The first of them is erosion. With this technique the boundaries of foreground object are eroded away, this way the foreground is kept white. The kernel slides trough the image and the pixel in the original image will be considered 1 only if all the pixels under the kernel are 1, otherwise it is eroded, that is, made to zero. All the pixels near boundary will be discarded depending upon the size of kernel, so that the thickness or size of the foreground object decreases, or simply white region decreases in the image. This technique is utterly useful when it comes to remove small white noises.

2.3.2 Dilation

The second technique used is dilation, which is simply the opposite of erosion. A pixel element is 1 if at least one pixel under the kernel is 1, this way the white region in the image increases or the size of foreground object increases. Since the aim is to remove noise, erosion is followed by dilation in the code for the following reason. Erosion removes white noises, but at the same time it also shrinks the object, so it needs to be dilated. Once the noise is gone, it won't come back but the area of the object increases.

2.4 Finding the center of the ball

After all the processing we can assume that all the detected pixels actually belong to the ball. Now we need to find the center of it. To do that we use an object of class Moments. It provides a way of calculating the center of mass of detected pixels. In the picture below, the red circle is drawn around the center of mass of white pixels on the right.

3 Mathematics of the project

3.1 Creating mathematical model of the process

In order to be able to design the best control system, we needed to have an accurate mathematical model of the process. The goal of this part was to have a virtual version of the process. We needed to be able to capture its current state as well as simulate its behavior assuming some future steering signals.

3.1.1 Calculating inversed kinematics of the platform

Since it's the platform that directly influences the ball, not the servos, we need to have a way of determining what signal we should send to the servos in order to achieve desired platform tilt.

We express the desired platform tilt using 2 angles- Alpha and Beta. These are the angles in the joint in x and y axes respectively (Image 1).

The resulting orientation matrix is the following:

$$[1\ 0\ 0\ 0\ \cos cos\ \alpha\ \ -\ \sin sin\ \alpha\ \ 0\ \ \sin sin\ \alpha\ \ \cos cos\ \alpha\ ] \times [\cos cos\ \beta\ \ 0\ \ \sin sin\ \beta\ \ 0\ 1\ 0\ \ -\ \sin sin\ \beta\ \ 0\ \ \cos cos\ \beta\ ] = [$$

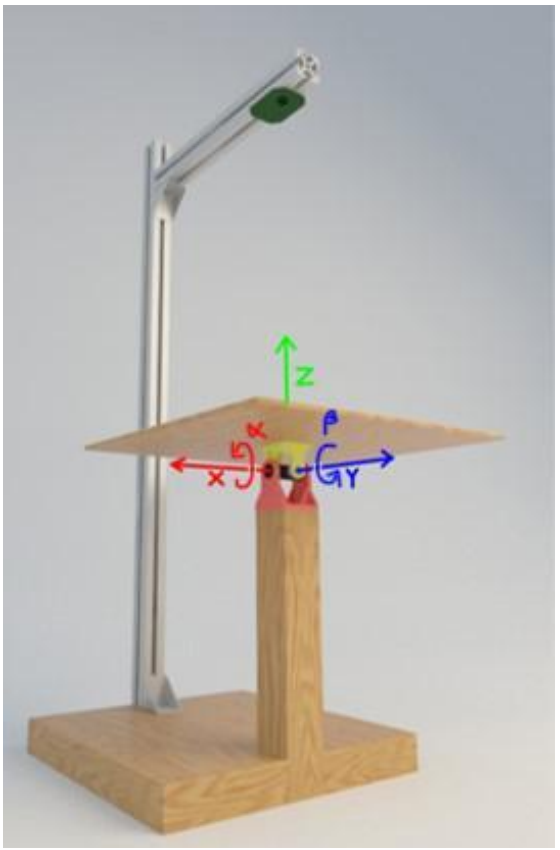


Image 1

Using the rotation matrix, we could calculate the position of the points of connection between the platform and the metal linkages (highlighted in Image 2) in world coordinates as following:

$$P_0 = [\cos cos\ \beta\ \ 0\ \ \sin sin\ \beta\ \ \sin sin\ \alpha\ \sin sin\ \beta\ \ \cos cos\ \alpha\ \ -\ \sin sin\ \alpha\ \cos cos\ \beta\ \ -\ \sin sin\ \beta\ \cos cos\ \alpha\ \ \sin sin\ \alpha\ \ c$$

Where P_1 is the position of the point in coordinate system relative to the platform with the origin at the point of intersection between axes of rotation (on Image 2 those are: $[x\ 0\ z\ ]$ and $[0\ \ -\ y\ z\ ]$)

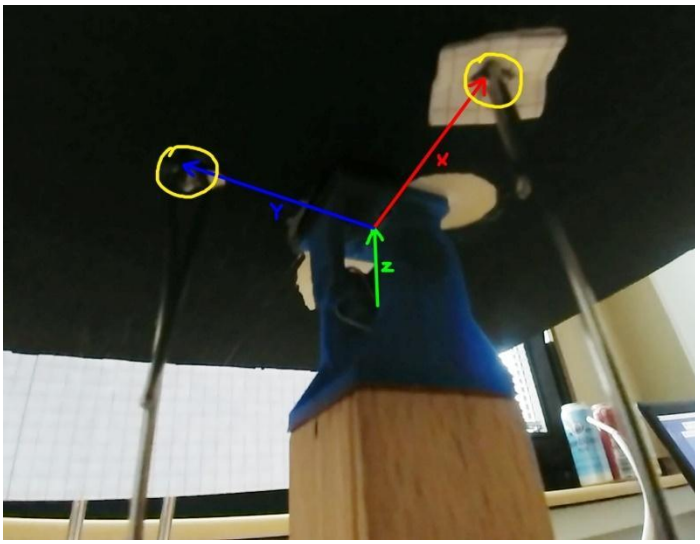


Image 2

The constraint is that the distance between the highlighted point and the tip of servo arm is equal to the length of metal rod. We can however simplify the equation by assuming that the links are always vertical. This way we just have to consider the z coordinate.

$$r\ sin\varphi\ = \Delta P_0 = ([-\ \sin sin\ \beta\ \cos cos\ \alpha\ \ \sin sin\ \alpha\ \ \cos cos\ \alpha\ \cos cos\ \beta\ ] - [-\ \sin sin\ 0\ \ \cos cos\ 0\ \ \sin sin\ 0\ \ \cos cos$$

$$\varphi\ = arcsin\Big(\big([-\ \sin sin\ \beta\ \cos cos\ \alpha\ \ \sin sin\ \alpha\ \ \cos cos\ \alpha\ \cos cos\ \beta\ \ -\ 1\ ]P_1\big)/r\Big)$$

φ – servo angle $\quad$ r – length of the servo arm

3.1.2 Calculating real world position of the ball

Since we want the model to be physically accurate, having the pixel coordinates of the ball won't be enough- we want to have the coordinates in real world. We decided to use for that purpose the coordinate system that's relative to the platform- the z coordinate is always 0.

The information from image processing tells us essentially at what angle the camera sees the ball but not how far away it is. In other words, using the pixel coordinates and the field of view of the camera we can write down the equations that describes the line at which the ball is (2 linear equations, red line in Image 3). We also know that the ball is on the platform as well as its tilt. Using normal vector to the platform we can find the equation of the surface – one linear equation. We are looking for the intersection point (highlighted in green in Image 3) so we just need to solve the system of those 3 linear equations.

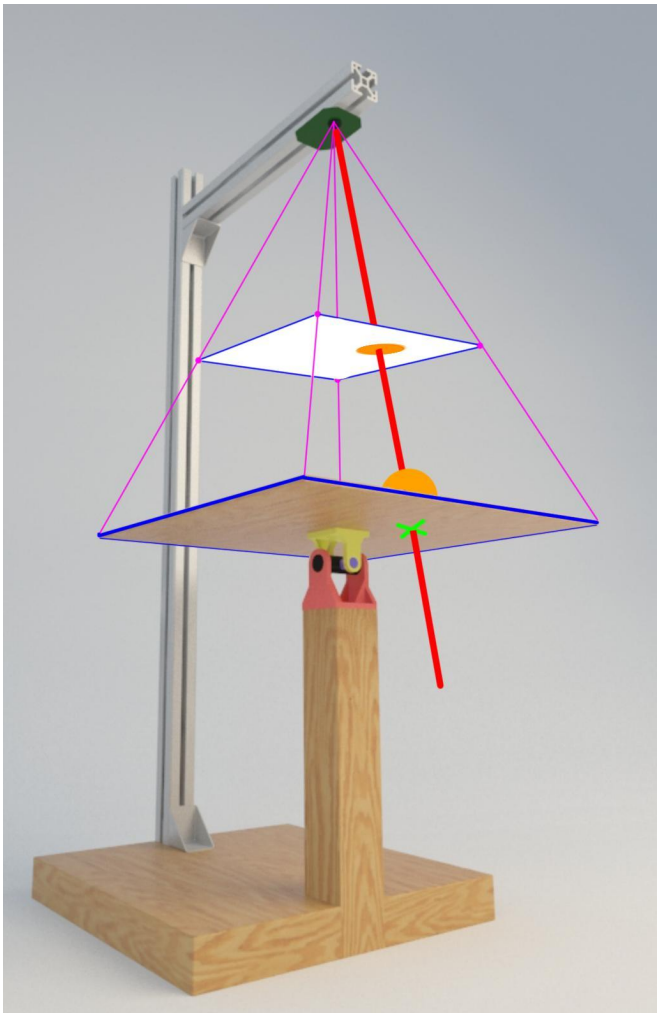


Image 3

The situation presented in Image 3 represents the simplest possibility, where $\beta\alpha = 0$. In this case, it's possible to solve the problem using simple proportions, but it doesn't apply in other cases.

After solving the system equations, what we get are the world coordinates of the ball. In order to have the position relative to the platform, we needed to multiply them by inversed rotation and translation matrix of the platform.

3.1.3 Calculating the acceleration in terms of α, β

Since we want to be able to predict what would happen if we applied some steering signal, we need to calculate what acceleration we can expect given α and β. Then, we can invert the function and set the desired acceleration of the ball and have the program calculate appropriate α and β and the the servo angles, using derived inversed kinematics.

First step in calculating the acceleration was to find the sine of the angle between platform and the horizontal plane. We can calculate it using normal vector.

$$n = [\cos cos\,\beta \;\; 0 \;\; \sin sin\,\beta \;\; \sin sin\,\alpha \;\; \sin sin\,\beta \;\; \cos cos\,\alpha \;\; - \sin sin\,\alpha \;\; \cos cos\,\beta \;\; - \sin sin\,\beta \;\; \cos cos\,\alpha \;\; \sin sin\,\alpha \;\; co$$

$$\sin sin\,\gamma = \sqrt{n_x^{\,2} + n_y^{\,2}}$$

Using this value as well as the moment of inertia of a sphere we were able to find the norm of acceleration vector. Next, we just had to align its direction so that it's parallel to the platform and directed towards the fastest descent.

$$n \cdot a = 0$$

$$\frac{a_x}{a_y} = \frac{n_x}{n_y}$$

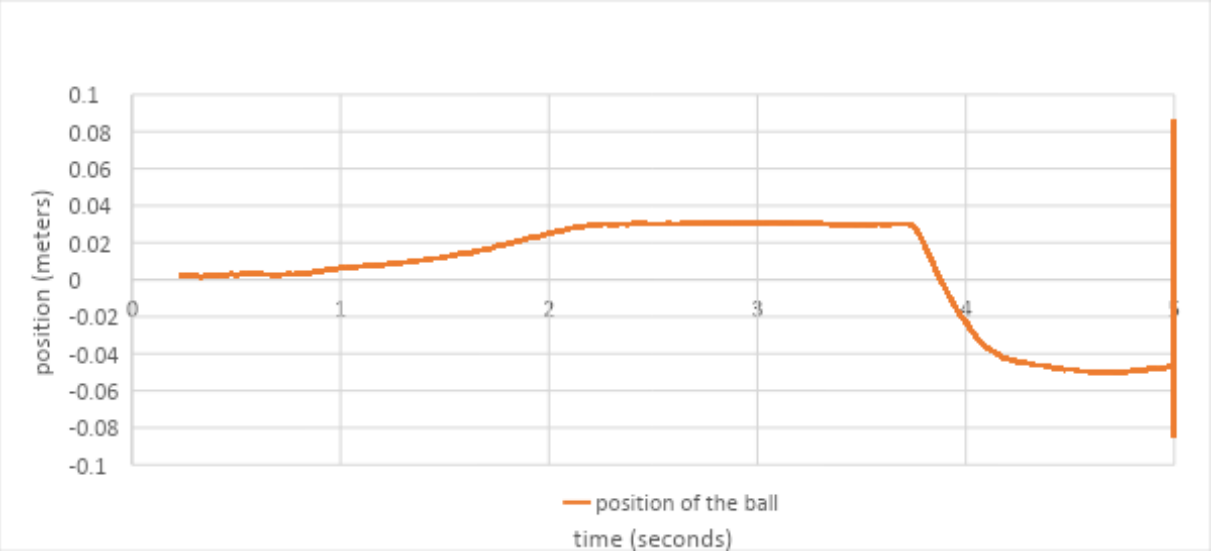
$$a_x^{\,2} + a_y^{\,2} + a_z^{\,2} = \|a\|$$

With the given set of equations we can find all the components of acceleration vector (in world coordinate system). Then we just had to multiply it by inverse rotation matrix of the platform to have it relative to the platform so that we can apply the formula to measured ball position. The calculated acceleration was:

$$a_p = \left[\frac{3}{5} g \sin sin\,\beta \;\; \cos cos\,\alpha \;\; - \frac{3}{5} g \sin sin\,\alpha \;\; 0\right]$$

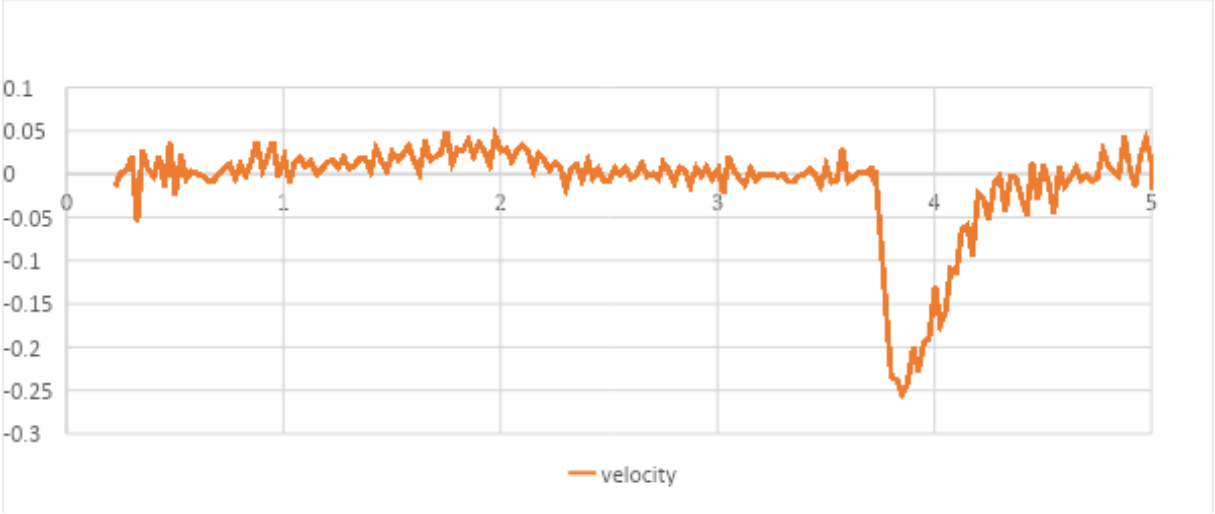
3.2 Improving the quality of input signal

Despite using denoising filters during image processing, the raw input signal still didn't look promising.



plot 1

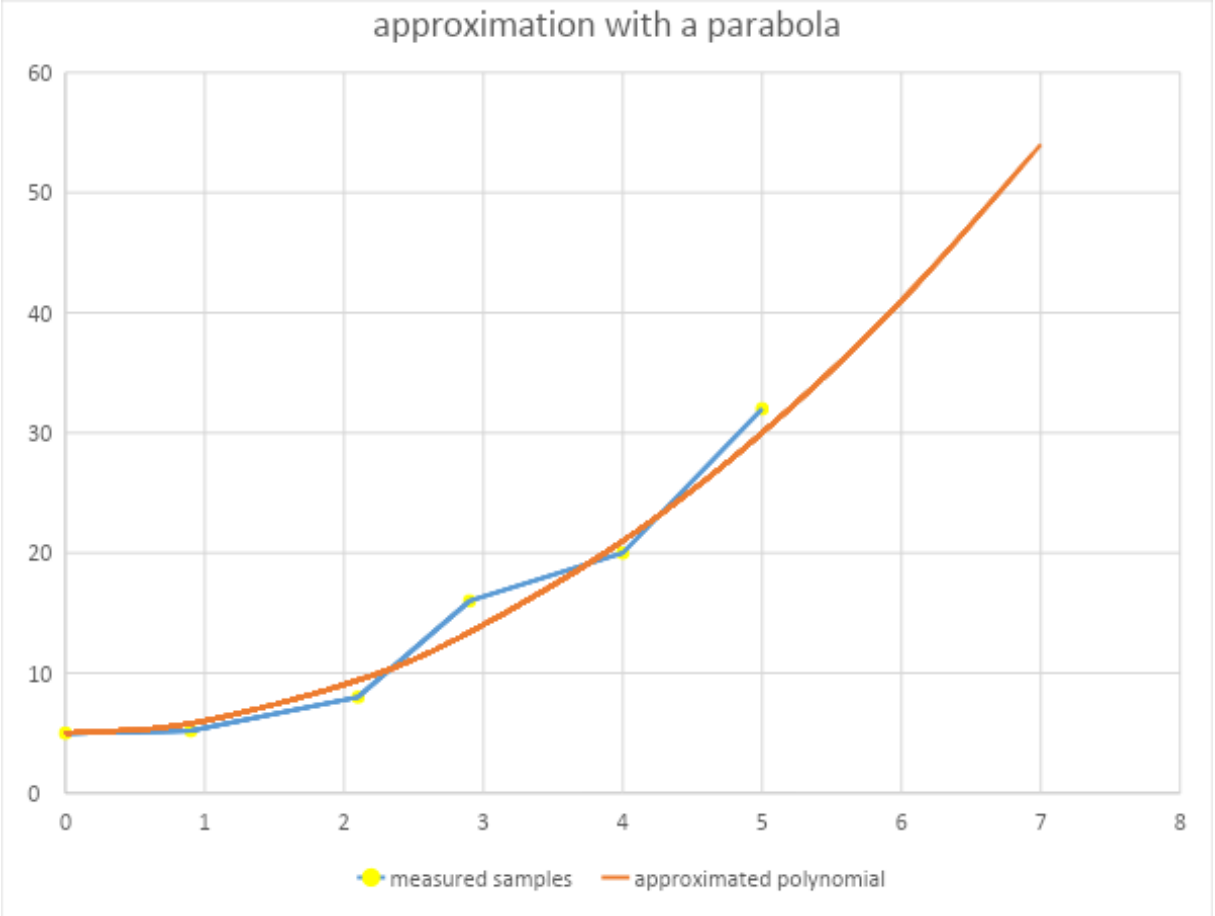
the visible noise theoretically didn't ruin the accuracy of the measurement that much (the amplitude of the noise is around 1 mm) but in order to stabilize the ball we need to have a reliable measurement of the ball's velocity. If we calculated it the traditional way (by calculating the difference between 2 subsequent samples and dividing it by the time difference), we would amplify the effect of the noise (as shown in plot 2)



plot 2

This amount of noise caused our first prototype to break (the rings that are connecting the metal linkages with the platform disconnected from the platform) so we had to find a way of filtering the signal.

Our solution to this problem is to approximate several most recent samples with a polynomial. We ended up approximating 5 previous samples with a parabola.



plot 3

There are 2 major advantages of this approach. First, we have a reliable way of measuring velocity (by calculating the derivative polynomial) and second, we can sample it at any given moment time. This turns out to be useful if we are aware that image processing takes some time- if we know when all the frames began to be processed, we can just sample the polynomial an appropriate amount of time ahead, thus reducing the input delay to some degree. In addition, the ability to sample the polynomial at any given time turns out to be useful when we need to have the measurements at constant frequency (the frames don't always come at consistent rate). In that case we can just sample the polynomial at desired timestamps.

The disadvantage to this solution is the fact that since it filters out high frequencies, sudden, unexpected changes are also filtered out in the beginning. If we hit the ball, the system will fully "believe" the unexpected change in velocity after 5 samples. This reduces the speed of reaction to such events

3.3 CONTROL SYSTEM

Having done all the mathematics (inversed kinematics, acceleration of the ball etc.) the control system's input signal is ball's position and the steering signal is the desired acceleration. Since x-acceleration doesn't influence y-position and vice versa, we can just use 2 separate control systems for those dimensions. The flow of the whole system is shown in Image 4

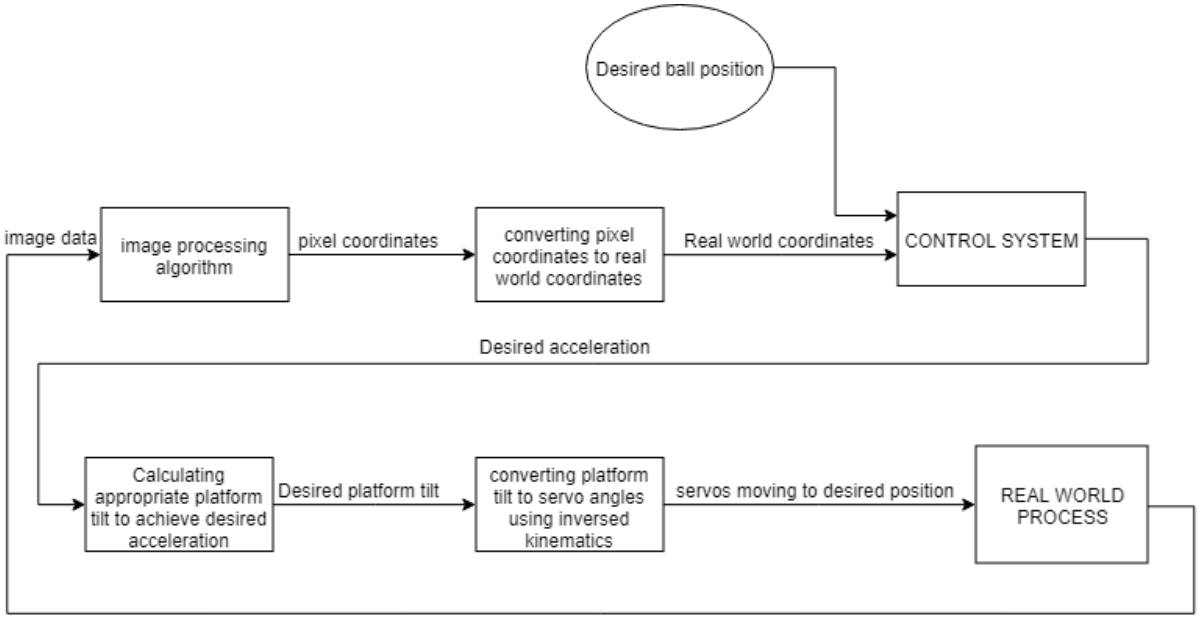


Image 4

3.3.1 PID controller

The object in question is a very basic unstable system. The "go-to" solution in such cases is a PID controller. The main advantage of this approach is the simplicity of implementation- it comes down to calculating the derivative and integral of the error and tuning just 3 variables. This approach has its limitations though. The most important in our case is the fact that it doesn't take into consideration the values of the setpoint in the future. In theory, the control signal, knowing we have some specific movement planned in the future, could start working towards achieving it in advance. This is especially visible when we want the ball to move in circles.

3.3.2 Model predictive control

3.3.2.1 Theoretical description

The second type of control system we have implemented is model predictive control (MPC). This controller takes advantage of the fact that we have a mathematical model to predict what effect the steering signal values will have in the future. Using this information, the program can find an optimal trajectory of steering signal. However, first we have to specify what "optimal" means. The cost function is the following:

$$\sum_{k} \left(y_r(k) - y(k)\right)^2 + \delta \bullet (\Delta u(k))^2$$

$y_r(k)$ – reference signal at sample k (setpoint)

$y(k)$ – predicted output signal (given some steering signal values)

δ – constant that balances the cost of error and changing steering signal value

This function sums up the squared errors, as well as steering signal values change (since we don't want the steering signal to change too rapidly) over some period of time. Turns out that if the model is linear we can express the predicted output signal in such a way that it's possible to analytically calculate the vector of steering signal value changes that minimizes the cost function value. Then all we need to do is apply the first element of derived steering signal vector and repeat this process every time we receive a new sample. Our model isn't linear though- servo angles don't have linear influence on the position of the ball detected by the camera. However, with the mathematical model of the process we ended up being able to set an acceleration and measure position in one coordinate system. This model, despite all the static non-linearities (in inversed kinematics, acceleration etc.) can be controlled with MPC. We "disconnect" the controller from all the non-linearities (like in Image 4). This kind of model is called Hammerstein-Wiener system.

The MPC requires describing a model with a difference equation. Thus, it is necessary to have measurements in very precise intervals. Here, the approximation with a polynomial comes in handy again- while receiving and processing the image can introduce a variable delay, after approximation with a polynomial we can simply sample it at precise points in time.

The difference equation has to have a following structure:

$$y(k) = a_1 y(k-1) + a_2 y(k-2) \dots a_n y(k-n) + b_1 u(k-1) + b_2 u(k-2) \dots b_m u(k-m)$$

In our case, y is position of the ball and u is its acceleration. First, we derived an according set of first order differential equations $y^{'(t)} = v(t)$

$$v^{'(t)} = u(t)$$

And calculated transfer function

$$G(s) = \frac{1}{s^2}$$

Next, using Matlab and zero-order-hold method, we were able to find a discrete transfer function equivalent. We also added some input delay since neither the camera nor the servos have instant response time.

```
Tp = 0.025;
Gc = tf([0,0,1],[1,0,0]);
Gc.InputDelay = 0.2;
Gd = c2d(Gc,Tp);
[num, den] = tfdata(Gd,'v');
```

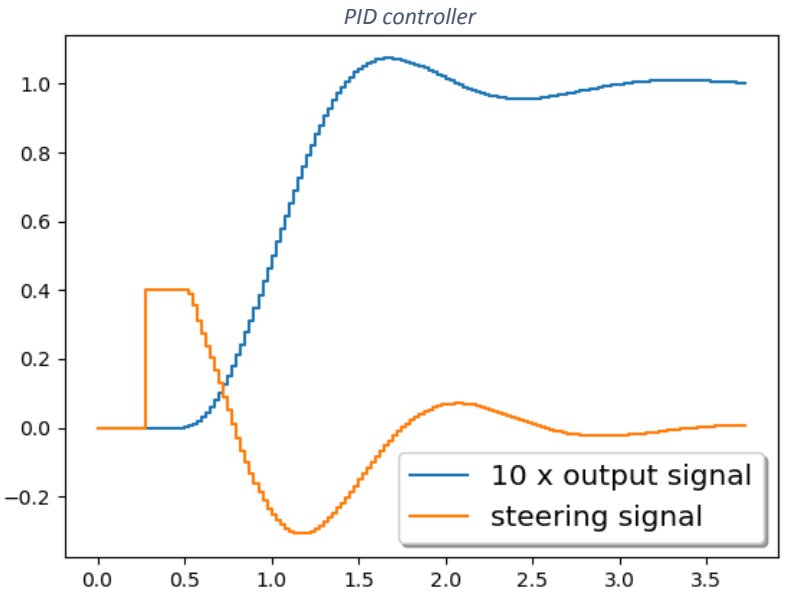
The resulting discrete transfer function was:

$$G(z) = z^{-delay/T_p} \frac{0.0035z^{-1} + 0.0035z^{-2}}{1 - 2z^{-1} + z^{-2}}$$

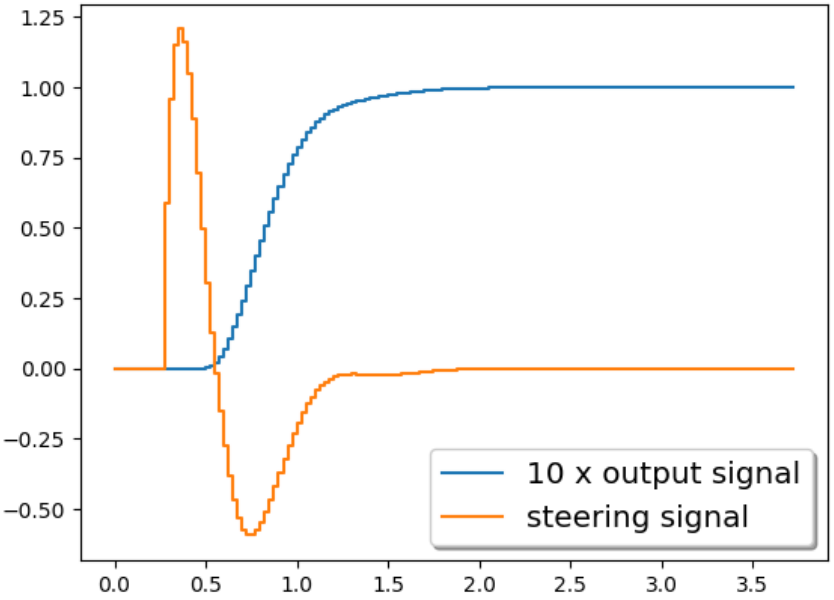
From here we can easily derive the suitable difference equation

$$y(k) = 2y(k-1) - y(k-2) + 0.0035u(k-1) + 0.0035(k-2)$$

The comparison between PID and model predictive control based on the simulation is shown below. The setpoint is a step signal at time t= 0.25s.



Model predictive control



As expected, the quality of control is way better. We can observe how the PID reacts only to the output signal value- only after the output starts moving towards the setpoint, the derivative component tries to damp the movement. MPC however "remembers" what steering signals have been applied in the past, even if their effect isn't visible yet. It starts braking even before the ball starts moving. In short, it comes up with an optimal policy, i.e. such that will minimize the cost function. Using MPC proves to be most effective in cases where system delay is significant or when we want the setpoint trajectory to change in pre-defined way.

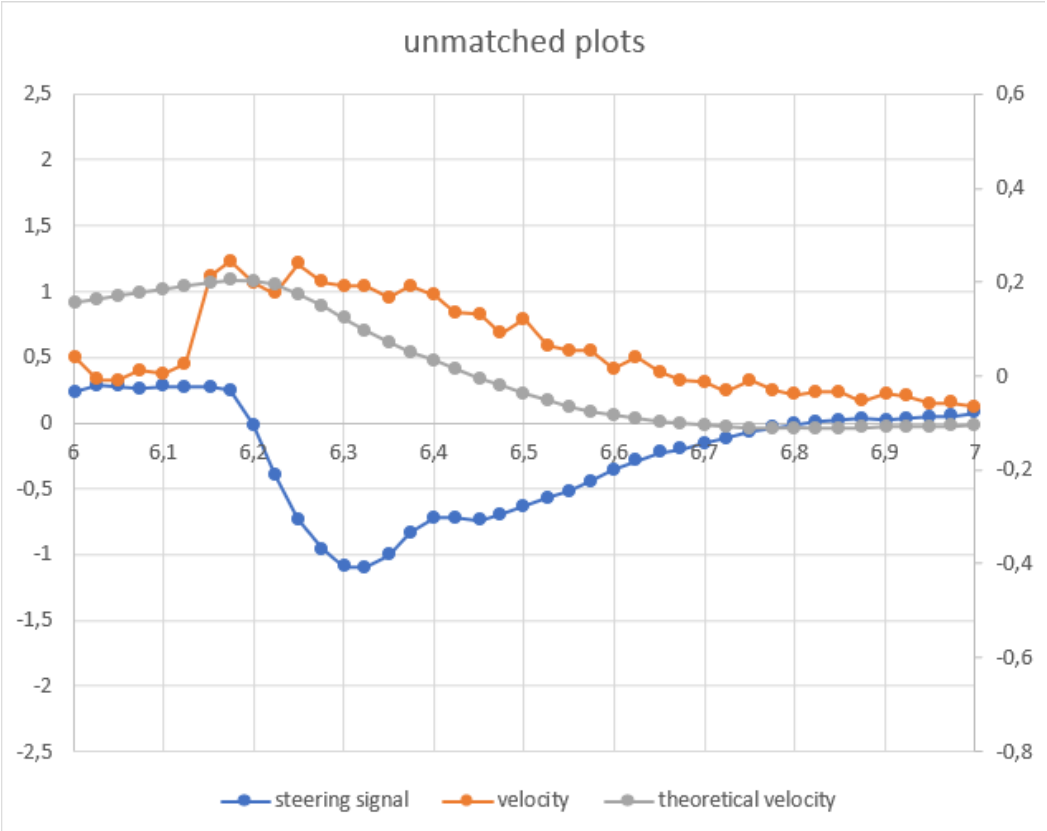
The disadvantage of this control system is how heavily it relies on the model- if it isn't accurate enough or we introduce too much interference to the system, it can perform worse than PID.

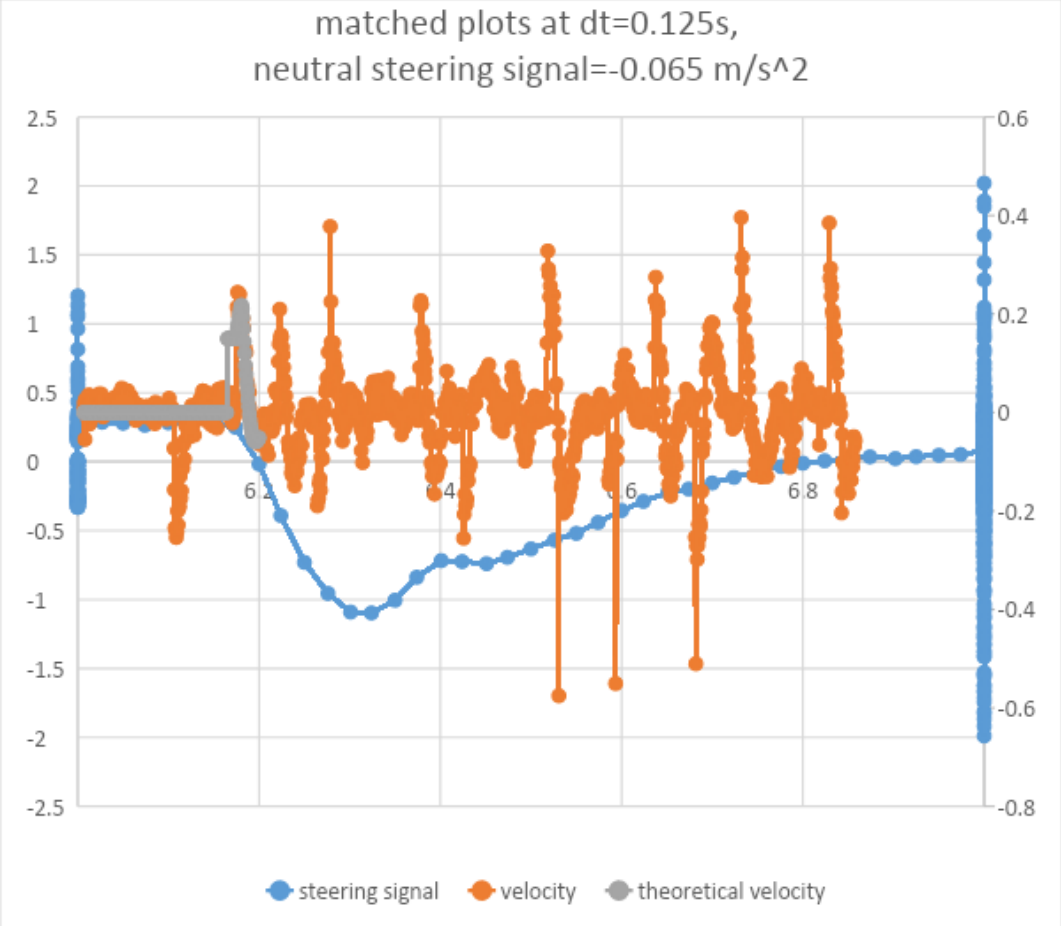
3.3.2.2 Practical implementation

One thing that turned out to be quite a challenge was figuring out the delay of the system. We decided to conduct a following experiment:

- Use any control system that can stabilize the ball (e.g. PID) and store velocity and steering signal data
- Hit the ball a couple times so that the changes in velocity are visible on the plot
- Measure average steering signal to cancel out steady state error (average applied steering signal is "neutral" and must be subtracted from current steering signal to get real acceleration)
- Integrate the corrected steering signal to get theoretical velocity
- Shift the integrated plot until it approximates directly measured velocity as closely as possible

The results before and after matching are presented below


unmatched plots
2,5
2
1,5
1
0,5
0
-0,5
-1
-1,5
-2
-2,5
0,6
0,4
0,2
0
-0,2
-0,4
-0,6
-0,8
6
6,1
6,2
6,3
6,4
6,5
6,6
6,7
6,8
6,9
7
steering signal
velocity
theoretical velocity


matched plots at dt=0.125s,
neutral steering signal=-0.065 m/s^2
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
6.2
6.4
6.6
6.8
steering signal
velocity
theoretical velocity

Using this data, we can assume that the delay is around 5 samples. However, the system worked a little bit better when we set it to 4.

3.3.3 Results analysis

Our benchmark for measuring the performance of both control systems was the ability to move the ball in circles. After analyzing the gathered footage, we can make some conclusions.

The PID control system, while achieving more circle-like path, can't maintain the set radius of the circle- it is bigger than it is supposed to. Moving the ball in circles requires a constant acceleration towards the center of the circle. If the ball is on the correct path, the only component that could contribute to such steering signal is the integral, but since the direction has to keep rotating, it isn't able to keep up with the change. Thus, the ball is following a path of a bigger circle and the proportional component is responsible for curving the movement.

MPC on the other hand is "aware" of the fact that it has to create a curvature to the path even if at current state the ball in its correct position. The followed path circle is actually a little bit smaller than the set trajectory. Since the required acceleration can be calculated with a formula

$$a = \omega^2 r$$

Where ω is constant, decreasing radius leads to decreasing required acceleration. Thus, the rate of change of steering signal decreases. As mentioned in the theoretical part, the MPC penalizes changes of steering signal, so it tries to find a correct balance between accuracy of trajectory and the rapidness of steering signal value.

What in our opinion could improve the quality of steering is using different, heavier ball- ping pong balls, due to very light weight are vulnerable to even the slightest irregularities on the platform- if the piece of paper covering it isn't completely flat, the ball doesn't have enough mass to press it against hard, even surface below.

3.3.4 Software structure

The most important class responsible for steering logics is Controller. Every time a new sample is gathered, a method "update" is called with ball's position as argument. Controller measures current time, uses the approximation with a parabola to filter the signal and then, using 2 separate control systems for 2 axes outputs desired acceleration. Objects of class Servos contain methods responsible for inversed kinematics that can be called with desired acceleration and send appropriate signal to serial port.

We have implemented 3 control systems – 2 versions of PID and MPC. The 2 versions of PID are the traditional version (PID_0) as well as one that calculates the derivative of the ball's position, not the error (PID_1).

Due to high complexity of the program it was extremely important to logically divide the code- we had to have a way of testing and debugging all the parts of the program separately, without involving actual hardware. Thus, all the classes could work individually.