LABORATORY L6 - REPORT

EMBEDDED & REAL TIME SYSTEMS

December 22nd, 2023

The sixth laboratory session aims to test the knowledge acquired from the first five practices. To do so. it proposes implementing the control of the *Ball in Tube* platform.

This problem aims to ensure that the ping-pong ball periodically changes between two setpoints (at 25 and 50 centimeters from the bottom of the tube) every 10 seconds.

To do so, we decided to use the Arduino IDE to code, and four different approaches:

- Simple ON/OFF control
- Control with hysteresis
- Simple feedback control
- PID controller

The videos of the developed projects can be seen in the following link:`https://drive.google.com/drive/folders/1jcUfE4kxGmwgYeyyOHwKt-4ZDBISoeMl?usp=sharing`. If for some reason the link in not working properly, please email us.

Overall structure.

As stated in the introduction, we decided to code using the *Arduino IDE*. Besides that, we used *FreeRTOS*. To do so, we implemented three tasks, listed below.

- **TaskSpeedSetpoint**: Task with the highest priority with a period of 10 seconds. Its main purpose was to change the height setpoint every 10 seconds.
- **TaskControl**: Task with the lowest priority with a period of 20 milliseconds. Its purpose is to compute the PWM value and send it to the fan.
- **TaskDistance**: Task with medium priority with a period of 20 milliseconds. Its purpose is to compute the distance using the ultrasound sensor.

Approach 1: Simple ON/OFF control.

For the first approach, the only objective is to have basic and easy control. The expected behavior is for the Arduino to send an *up* signal to the fan if the ball is below the current setpoint, and a *down* signal otherwise.

For these signals, we decided to first test the fan power to see the ball behavior when sending different *PWM* values. This experiment consisted of setting the ball at different random values and seeing if an arbitrary value for the signal made it go up, down, or stay in its place.

From that, we observed a weird behavior, different from some of our colleagues. While they had a value that kept the ball in a constant position, we couldn't find one. If we sent a value of *59* through the `AnalogWrite` command, the ball descended and if it was of *60*, the ball ascended, independently of its position.

With that experiment done, we had the values to code the ON and OFF control, where the ON signal corresponds to a value of *60* and the OFF signal to a value of *59*.

Therefore, the obtained code can be seen below.

```
#include <Arduino_FreeRTOS.h>
#include <semphr.h>

// define three tasks for setting the speed setpoint & computing both distance and control action
void TaskSpeedSetpoint( void *pvParameters );
void TaskFeedForward( void *pvParameters );
void TaskDistance( void *pvParameters );

// define global variables
float h_value;      // high distance to sensor value
float l_value;      // low distance to sensor value
bool x;             // toggle variable to change between distance values {True: 15; False: 40}
float pwm;          // speed variable (PWM send to the fan)
float setpoint;     // desired nominal speed value
float ECHO = 7;     // echo pin
float TRIGGER = 6;  // trigger pin
int y;              // high time of the received signal form the distance sensor
float cm;           // distance measured from the sensor

void setup() {
  pinMode(TRIGGER,OUTPUT);     // set trigger pin as output
  pinMode(ECHO,INPUT);         // set echo pin as input
  pinMode(9, OUTPUT);          // set digital pin 9 as output (PWM pulse to the fan)
  pinMode(3, INPUT_PULLUP);    // set digital pin 3 as input (F00 sensor)
  Serial.begin(9600);          // start serial port
  h_value = 65-25;             // set high distance to sensor value
  l_value = 65-50;             // set low distance to sensor value
  x = false;                   // initialize to start at high distance

  // create first task
  xTaskCreate(
  TaskSpeedSetpoint
  , "SpeedSetpoint" // name
  , 128             // stack size
  , NULL
  , 3               // high priority
  , NULL
  );

  // create second task
  xTaskCreate(
  TaskFeedForward
  , "FeedForward"   // name
  , 128             // stack size
  , NULL
  , 1               // low priority
  , NULL
  );

  // create third task
  xTaskCreate(
  TaskDistance
  , "Distance"      // name
  , 128             // stack size
```

```
  , NULL
  , 2              // medium priority
  , NULL
  );
  // now the task scheduler is automatically started
}

void loop()
{
  // empty loop
}

//--------------------------------------------------
// First task: Iterating through states
//--------------------------------------------------
void TaskSpeedSetpoint(void *pvParameters)
{
  (void) pvParameters;

  // forever loop
  for (;;)
  {
    if (x){                    // checking the current setpoint
      setpoint = l_value;    // toggle value
      x=false;
    }
    else {
      setpoint = h_value;    // toggle value
      x=true;
    }
    vTaskDelay(10000 / portTICK_PERIOD_MS);  // wait for 10 seconds
  }
}

//--------------------------------------------------
// Second task: Feed Forward Control
//--------------------------------------------------
void TaskFeedForward(void *pvParameters)
{
  (void) pvParameters;

  // forever loop
  for (;;)
  {
    if (cm < setpoint){         // checking if setpoint reached
      analogWrite(9, 59);       // reduce fan speed
    }
    else{
      analogWrite(9,60);        // increase fan speed
    }
    vTaskDelay(20 / portTICK_PERIOD_MS);  // wait for 20 miliseconds
  }
}

//--------------------------------------------------
// Third task: Computing Distance
//--------------------------------------------------
void TaskDistance(void *pvParameters)
{
  (void) pvParameters;

  // forever loop
  for (;;)
  {
    digitalWrite(TRIGGER, HIGH);           // send a 20 ms signal
    vTaskDelay(20 / portTICK_PERIOD_MS);
    digitalWrite(TRIGGER, LOW);

    y = pulseIn(ECHO, HIGH);               // Count time to recieve echo back
    cm = y/58;                             // Convert to centimeters
```

```

    Serial.print(cm);                     // display results
    Serial.print(",");
    Serial.println(setpoint);
  }
}
```

Listing 1: Code for simple ON/OFF control.

First, we specify the tasks to be done, the `TaskSpeedSetpoint`, the `TaskFeedForward` and the `TaskDistance`. Then, other global variables are defined to be used throughout the code. The tasks are created, specifying its name, stack size and priority, and the loop of this program is left empty.

The `TaskSpeedSetpoint`, changes the setpoint from one to the other each 10 seconds. As this is a feature that will not change on the different approaches implemented, this task will always be the same.

The `TaskDistance` sends a high value to the sensor through the trigger during 20 miliseconds. Then, it counts the time that the echo signal is at high value, to enable the distance computation using the same formula used in *Laboratory 2*. Also, this task sends the results through the Serial to represent the results.

The `TaskFeedForward` checks if the ping-pong ball reaches the current setpoint defined by `TaskSpeedSetpoint`. If the ball is above the current setpoint -which means closer to the sensor, so its distance is lower than the setpoint-, then the fan speed is reduced. Otherwise, if the ball is below, the fan speed is increased. Then a delay is added to enable the scheduler to perform another task. As happened with the first task explained, this one is also the same for all the approaches, as the sensor is not changed.

As expected, this code obtained poor behavior and had a lot of room for upgrades. The results of the program execution can be seen in the video `approach1.MOV`.

Approach 2: Control with hysteresis.

For this second approach, we decided to add a hysteresis. This means that the ON signal won't be sent until the ball is a threshold below the current setpoint, and the OFF signal won't be sent until it is a threshold over the current setpoint.

Despite that, when the ball goes from one state to the other it takes a bit to respond. This produced a worse behavior, having bigger oscillations around the setpoint. For that reason, we decided to do an *inverse* hysteresis, meaning that the *down* signal will be sent when the ball is ascending and it is a bit below the setpoint, and similarly, the *up* signal will be sent when the ball is descending and it is a bit over the setpoint.

```
#include <Arduino_FreeRTOS.h>
#include <semphr.h>

// define three tasks for setting the speed setpoint & computing both distance and control action
void TaskSpeedSetpoint( void *pvParameters );
void TaskFeedForward( void *pvParameters );
void TaskDistance( void *pvParameters );

// define global variables
float h_value;       // high distance to sensor value
float l_value;       // low distance to sensor value
bool x;              // toggle variable to change between distance values {True: 15; False: 40}
float pwm;           // speed variable (PWM send to the fan)
float setpoint;      // desired nominal speed value
float ECHO = 7;      // echo pin
float TRIGGER = 6;   // trigger pin
```

```
int y;                 // high time of the received signal form the distance sensor
float cm;              // distance measured from the sensor
float hist = 2;        // amplitude of the hysteresis set to 2cm
float state;           // fan at high or low speed {0: Ball descending; 1: Ball ascendding
    }

void setup() {
  pinMode(TRIGGER,OUTPUT);      // set trigger pin as output
  pinMode(ECHO,INPUT);          // set echo pin as input
  pinMode(9, OUTPUT);           // set digital pin 9 as output (PWM pulse to the fan)
  pinMode(3, INPUT_PULLUP);     // set digital pin 3 as input (F00 sensor)
  Serial.begin(9600);           // start serial port
  h_value = 65-25;              // set high distance to sensor value
  l_value = 65-50;              // set low distance to sensor value
  x = false;                    // initialize to start at high distance

  // create first task
  xTaskCreate(
  TaskSpeedSetpoint
  , "SpeedSetpoint"  // name
  , 128              // stack size
  , NULL
  , 3                // high priority
  , NULL
  );

  // create second task
  xTaskCreate(
  TaskFeedForward
  , "FeedForward"    // name
  , 128              // stack size
  , NULL
  , 1                // low priority
  , NULL
  );

  // create third task
  xTaskCreate(
  TaskDistance
  , "Distance"       // name
  , 128              // stack size
  , NULL
  , 2                // medium priority
  , NULL
  );
  // now the task scheduler is automatically started
}

void loop()
{
  // empty loop
}

//-----------------------------------------------
// First task: Iterating through states
//-----------------------------------------------
void TaskSpeedSetpoint(void *pvParameters)
{
  (void) pvParameters;

  // forever loop
  for (;;)
  {
    if (x){                   // checking the current speed level
      setpoint = l_value;     // toggle value
      x=false;
    }
    else {
      setpoint = h_value;     // toggle value
      x=true;
```

```
    }
    vTaskDelay(10000 / portTICK_PERIOD_MS);  // wait for 10 seconds
  }
}

//--------------------------------------------------
// Second task: Feed Forward Control
//--------------------------------------------------
void TaskFeedForward(void *pvParameters)
{
  (void) pvParameters;

  // forever loop
  for (;;)
  {
    if ((cm < setpoint + hist) && (state == 1)){    // checking if ball ascending and hysteresis reached
      analogWrite(9, 59);   // reduce fan speed
      state = 0;
    }
    else if ((cm > setpoint - hist) && (state == 0)){ // checking if ball descending and hysteresis reached
      analogWrite(9,60);    // increase fan speed
      state = 1;
    }
    vTaskDelay(20 / portTICK_PERIOD_MS);  // wait for 20 miliseconds
  }
}

//--------------------------------------------------
// Third task: Computing Distance
//--------------------------------------------------
void TaskDistance(void *pvParameters)
{
  (void) pvParameters;

  // forever loop
  for (;;)
  {
    digitalWrite(TRIGGER, HIGH);              // send a 20 ms signal
    vTaskDelay(20 / portTICK_PERIOD_MS);
    digitalWrite(TRIGGER, LOW);

    y = pulseIn(ECHO, HIGH);                  // Count time to recieve echo back
    cm = y/58;                                // Convert to centimeters

    Serial.print(cm);                         // display results
    Serial.print(",");
    Serial.println(setpoint);
  }
}
```

Listing 2: Code for control with hysteresis.

This code is really similar to the one of the previous section. In fact, only the `TaskFeedForward` is modified to include the hysteresis commented. To do so, first an additional variable to determine the state of the ball -whether it is ascending or descending- is created. Also, the distance reference -in this case, the sensor- must be taken into account. As the sensor is located on top of the tube, the upper bound of the hysteresis will be defined as `setpoint - hist`, whereas the lower bound will be defined as `setpoint + hist`.

The conditions to change the value of the fan speed are determined as explained before: when the ball is ascending and it is a bit below the setpoint (`setpoint + hist`) the fan speed will be decreased, whereas when the ball is descending and it is a bit over the setpoint (`setpoint - hist`) the fan speed will be increased.

This code shows a better behavior, shown in *Figure* 1. Despite that, it does not respond well to noise, for that reason, other approaches are proposed.

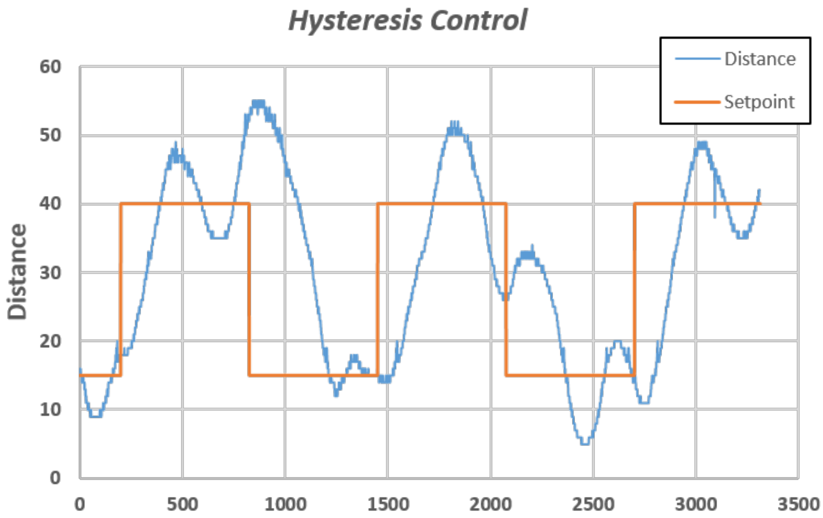


Figure 1: Behaviour of control with hysteresis.

As can be seen on the graph above, it is clear that the system has a lot of oscillations around the setpoints. This may be caused by external factors that are not considered, for example, the dynamics of the system can be slower than the time it has to stabilize -as each 10 seconds the setpoint is changed. Also, fluid mechanics properties could produce some unknown interactions caused because the diameter of the ball is a bit smaller than the diameter of the tube.

The results of the program execution can be seen in the video `approach2.MOV`.

Approach 3: Simple feedback control.

For this section, we implemented a proportional feedback control. To do so, we added a new function `control`, which computes the PWM value to send using the proportional constant, the desired setpoint, and the current distance. Therefore, the used function is:

$$PWM = -k \cdot (setpoint - distance) + 59 \tag{1}$$

Note that a *59* was added to the PWM value. This was made so the constant could be kept low and, therefore, the final PWM value was more stable, rather than oscillating through really different values. The proposed code can be seen below.

```
#include <Arduino_FreeRTOS.h>
#include <semphr.h>

// define three tasks for setting the speed setpoint & computing both distance and
    control action
void TaskSpeedSetpoint( void *pvParameters );
void TaskFeedback( void *pvParameters );
void TaskDistance(void *pvParameters);


// define global variables
float h_value;       // high distance to sensor value
float l_value;       // low distance to sensor value
```

```
bool x;               // toggle variable to change between distance values {True: 15; False: 40}
float pwm;            // speed variable (PWM send to the fan)
float setpoint;       // desired nominal speed value
float ECHO = 7;       // echo pin
float TRIGGER = 6;    // trigger pin
int y;                // high time of the received signal form the distance sensor
float cm;             // distance measured from the sensor
float k = 0.25;       // value for the proportional controller

void setup() {
  pinMode(TRIGGER,OUTPUT);     // set trigger pin as output
  pinMode(ECHO,INPUT);         // set echo pin as input
  pinMode(9, OUTPUT);          // set digital pin 9 as output (PWM pulse to the fan)
  pinMode(3, INPUT_PULLUP);    // set digital pin 3 as input (F00 sensor)
  Serial.begin(9600);          // start serial port
  h_value = 65-25;             // set high distance to sensor value
  l_value = 65-50;             // set low distance to sensor value
  x = false;                   // initialize to start at high distance

  // create first task
  xTaskCreate(
  TaskSpeedSetpoint
  , "SpeedSetpoint" // name
  , 128             // stack size
  , NULL
  , 3               // high priority
  , NULL
  );

  // create second task
  xTaskCreate(
  TaskFeedback
  , "Feedback"      // name
  , 128             // stack size
  , NULL
  , 1               // low priority
  , NULL
  );

  // create third task
  xTaskCreate(
  TaskDistance
  , "Distance"      // name
  , 128             // stack size
  , NULL
  , 2               // medium priority
  , NULL
  );
  // now the task scheduler is automatically started
}

void loop()
{
  // empty loop
}

//-----------------------------------------
// First task: Iterating through states
//-----------------------------------------
void TaskSpeedSetpoint(void *pvParameters)
{
  (void) pvParameters;

  // forever loop
  for (;;)
  {
    if (x){                    // checking the current speed level
      setpoint = l_value;    // toggle value
      x=false;
```

```
    }
    else {
      setpoint = h_value;    // toggle value
      x=true;
    }
    vTaskDelay(10000 / portTICK_PERIOD_MS);  // wait for 10 seconds
  }
}

//--------------------------------------------------
// Second task: Feedback Control
//--------------------------------------------------
void TaskFeedback(void *pvParameters)
{
  (void) pvParameters;

  // forever loop
  for (;;)
  {
    float input = control(cm,setpoint);     // compute control action
    analogWrite(9,input);                   // sending fan velocity
    vTaskDelay(20 / portTICK_PERIOD_MS);    // wait for 20 miliseconds
  }
}

//--------------------------------------------------
// Third task: Distance
//--------------------------------------------------
void TaskDistance(void *pvParameters)
{
  (void) pvParameters;

  // forever loop
  for (;;)
  {
    digitalWrite(TRIGGER, HIGH);            // send a 20 ms signal
    vTaskDelay(20 / portTICK_PERIOD_MS);
    digitalWrite(TRIGGER, LOW);

    y = pulseIn(ECHO, HIGH);                // Count time to recieve echo back
    cm = y/58;                              // Convert to centimeters

    Serial.print(cm);                       // display results
    Serial.print(",");
    Serial.println(setpoint);
  }
}


// auxiliar function to compute feedback control action
float control(float dist, float setpoint){
  float value = -k*(setpoint-dist) + 59;
  return value;
}
```

Listing 3: Code for simple feedback control.

This code is really similar to the ones of the previous sections. In this case, only the control task is changed to `TaskFeedback`. Also, the additional auxiliary function commented before and the variables it involves are added. The `TaskFeedback` calls the `control` function to compute the action value, expressed by *Formula* 1.

This code has the best results so far, shown in *Figure* 2. To obtain them, a proper tunning of the constant was done, trying different values to see how the program responded. The final value for the proportional controller was $k = 0{,}25$.

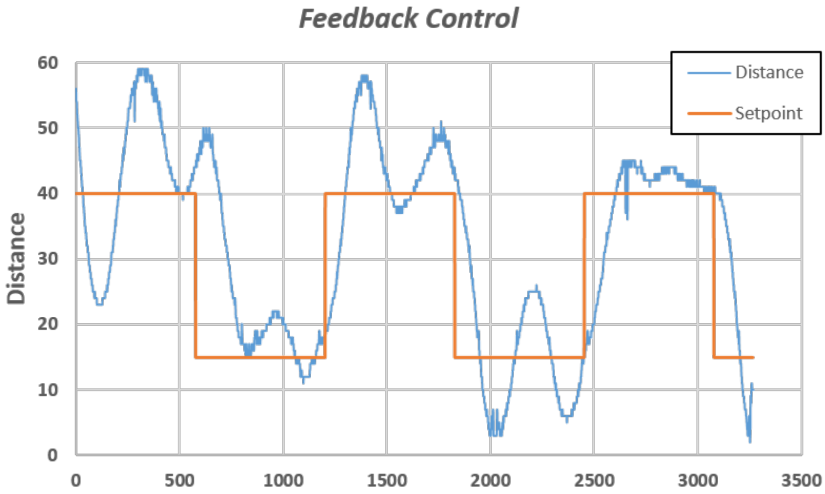


Figure 2: Behaviour of feedback control.

As can be seen on the graph above, the oscillations of the system remain, but sometimes these are really reduced (like between 2600 and 3100 approximately). It can also be observed that the system reacts faster to the setpoint changes.

The results of the program execution can be seen in the video `approach3.MOV`.

Approach 4: PID controller.

The final approach consisted of implementing a PID controller, as seen in previous practices of the subject. To do so, we changed the function `control` to implement the stated control method. Besides that, we also needed to input to the function the current error, the error of the two previous iterations, and the previous control signal sent.

The code used to implement this behavior can be seen below.

```
#include <Arduino_FreeRTOS.h>
#include <semphr.h>

// define three tasks for setting the speed setpoint & computing both distance and control action
void TaskSpeedSetpoint( void *pvParameters );
void TaskPID( void *pvParameters );
void TaskDistance(void *pvParameters);


// define global variables
float h_value;              // high distance to sensor value
float l_value;              // low distance to sensor value
bool x;                     // toggle variable to change between distance values {True: 15; False: 40}
float pwm;                  // speed variable (PWM send to the fan)
float setpoint;             // desired nominal speed value
float ECHO = 7;             // echo pin
float TRIGGER = 6;          // trigger pin
int y;                      // high time of the received signal form the distance sensor
float cm;                   // distance measured from the sensor
float input;                // fan velocity to send

```

```

// define PID parameters
float K = 0.2;              // proportional part value
float Td = 0.05;            // derivative part value
float Ti = 9;               // integral part value
float Ts = 0.02;            // sampling period
float prev_error[2];        // previous errors
float prev_control = 56;    // previous control (initialized at 56)


void setup() {
  pinMode(TRIGGER,OUTPUT);     // set trigger pin as output
  pinMode(ECHO,INPUT);         // set echo pin as input
  pinMode(9, OUTPUT);          // set digital pin 9 as output (PWM pulse to the fan)
  pinMode(3, INPUT_PULLUP);    // set digital pin 3 as input (F00 sensor)
  Serial.begin(9600);          // start serial port
  h_value = 65-25;             // set high distance to sensor value
  l_value = 65-50;             // set low distance to sensor value
  x = false;                   // initialize to start at high distance

  // create first task
  xTaskCreate(
  TaskSpeedSetpoint
  , "SpeedSetpoint" // name
  , 128             // stack size
  , NULL
  , 3               // high priority
  , NULL
  );

  // create second task
  xTaskCreate(
  TaskPID
  , "PID"           // name
  , 128             // stack size
  , NULL
  , 1               // low priority
  , NULL
  );

  // create third task
  xTaskCreate(
  TaskDistance
  , "Distance"      // name
  , 128             // stack size
  , NULL
  , 2               // medium priority
  , NULL
  );
  // now the task scheduler is automatically started
}

void loop()
{
  // empty loop
}

//-------------------------------------------------
// First task: Iterating through states
//-------------------------------------------------
void TaskSpeedSetpoint(void *pvParameters)
{
  (void) pvParameters;

  // forever loop
  for (;;)
  {
    if (x){                       // checking the current speed level
      setpoint = l_value;     // toggle value
      x=false;
```

```
    }
    else {
      setpoint = h_value;    // toggle value
      x=true;
    }
    vTaskDelay(10000 / portTICK_PERIOD_MS);   // wait for 10 seconds
  }
}

//------------------------------------------------------------
// Second task: PID Control
//------------------------------------------------------------
void TaskPID(void *pvParameters)
{
  (void) pvParameters;

  // forever loop
  for (;;)
  {
    float error = -(setpoint - cm);            // compute current error
    input = control(error,prev_error,prev_control);    // compute control action
    prev_error[1] = prev_error[0];             // update errors
    prev_error[0] = error;
    prev_control = input;                      // update previous control
    analogWrite(9,input);                      // sending fan velocity
    vTaskDelay(20 / portTICK_PERIOD_MS);       // wait for 20 miliseconds
  }
}

//------------------------------------------------------------
// Third task: Distance
//------------------------------------------------------------
void TaskDistance(void *pvParameters)
{
  (void) pvParameters;

  // forever loop
  for (;;)
  {
    digitalWrite(TRIGGER, HIGH);             // send a 20 ms signal
    vTaskDelay(20 / portTICK_PERIOD_MS);
    digitalWrite(TRIGGER, LOW);

    y = pulseIn(ECHO, HIGH);                 // Count time to recieve echo back
    cm = y/58;                               // Convert to centimeters

    Serial.print(cm);                        // display results
    Serial.print(",");
    Serial.println(setpoint);
  }
}


// auxiliar function to compute PID control action
float control(float error, float error_ant[2], float control_ant){
  float a1,a2,a3,control;
  a1 = K*(1+Ts/Ti+Td/Ts);
  a2 = K*(-1-2*Td/Ts);
  a3 = K*Td/Ts;
  control = control_ant + a1*error + a2*error_ant[0] + a3*error_ant[1];
  return control;
}
```

Listing 4: Code for PID controller.

This code is really similar to the ones of the previous sections. The control task is changed to `TaskPID`, and the additional auxiliary function commented before along with the necessary global variables it uses are added.

The `TaskPID` computes both the error and control action, calling the `control` function. This function computes the action value, expressed by the discrete-time PID formula. Then, it updates the variables of previous errors and controls to be used in next iteration. To make the initial response faster, we started the value of the `prev_control` variable at 56. In doing so, the ball starts responding way faster to the first signal.

The results of executing the code are shown in *Figure* 3. To obtain them, the different values of the controller had to be tuned. First, we started by using the proportional constant obtained in the previous control and kept the derivative part zero. With that, we iterated through different values of the integrator constant to find the best response. Later, we repeated the process but with the derivative constant. The final obtained values for the parameters are $K = 0{,}2$, $T_i = 9$ and $T_d = 0{,}05$.

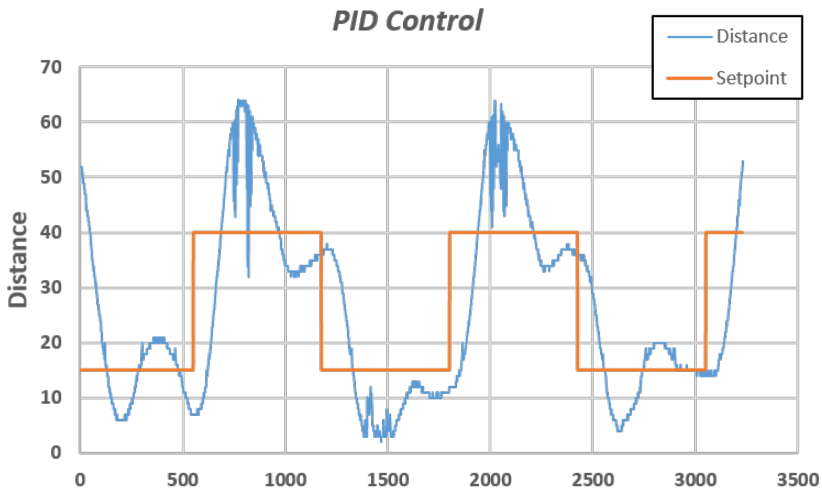


Figure 3: Behaviour of PID control.

As can be seen on the graph above, the PID control seems to produce really high peaks -the ball really high and closer to the sensor- when the setpoint is changed to high value, but for the low value of the setpoint this does not happen so noticeably. This means that in this case the control seems to produce different behaviours for the different setpoint values. It also has a good response velocity when the setpoint is changed, but the oscillations are still present.

The results of the program execution can be seen in the video `approach4_1.MOV`.

It still seems that due to the slow dynamics of the system the control is not done properly. In order to study this in more detail, the time between setpoint changes have been modified and defined as 20 seconds. With this new situation, the results of executing the code are shown in *Figure* 4.

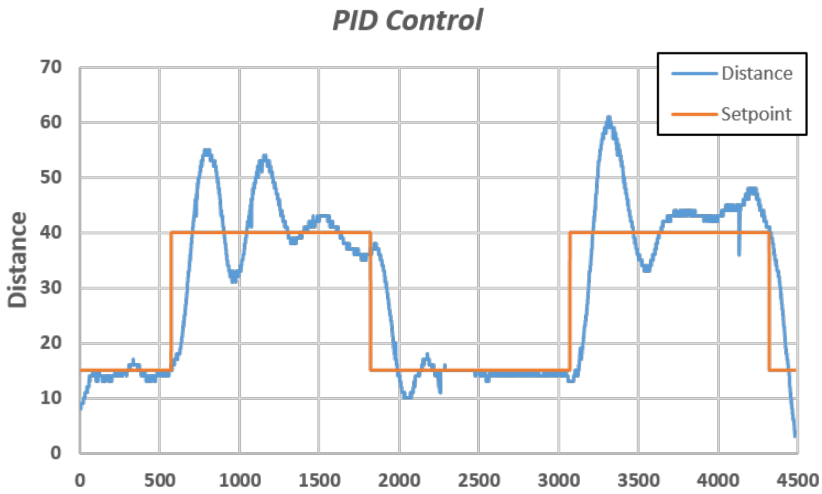


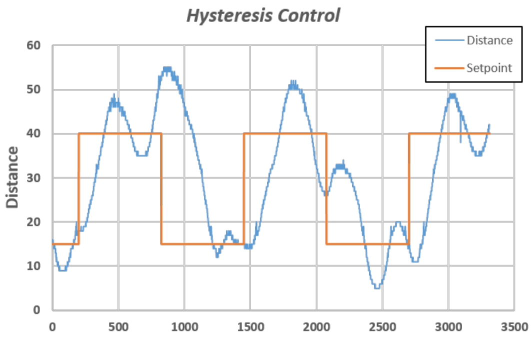
Figure 4: Behaviour of PID control.

As can be seen on *Figure* 4, as there is more time between setpoint changes, it seems that the systems arrives to a more stable position, and this produces less peaks and oscillations on the distance. Also, the behaviour on lower distances to the sensor -when the ball was at a higher position- seems to be more stable than for higher distances.

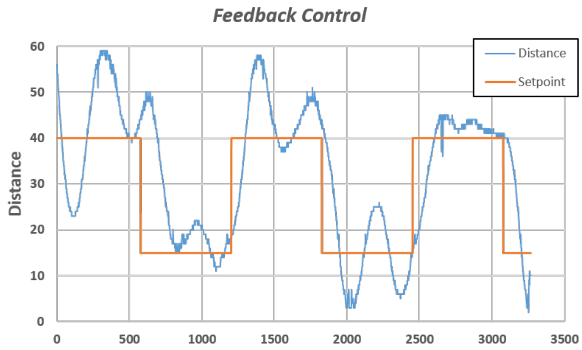
The results of the program execution can be seen in the video `approach4_2.MOV`.

Conclusions and future works

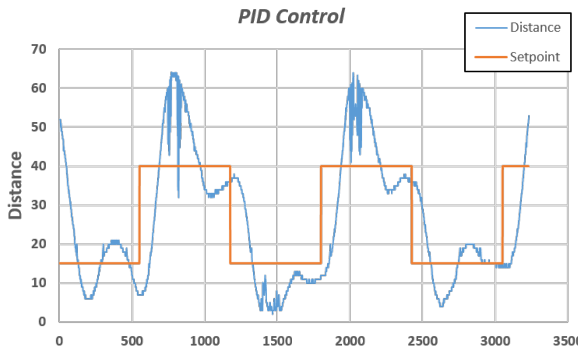
Once we completed the demanded exercises, we would like to compare the obtained results for those control approaches that we considered more interesting as they provided a better performance of the system *Ball in Tube*. The results are summarized in the figure below.



(a) Hysteresis



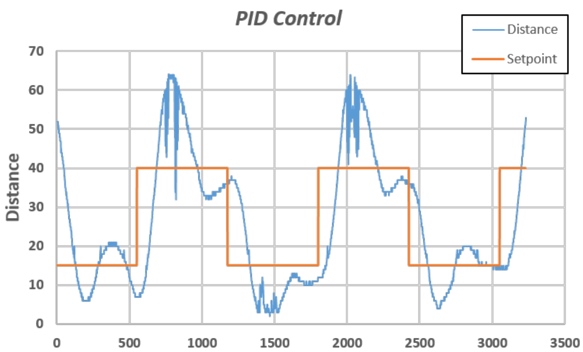
(b) Feedback



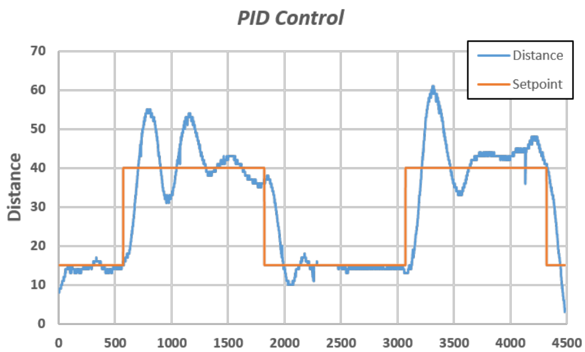
(c) PID

Figure 5: Obtained results for each control approach.

We would like to note that it is complex to analyze these results with the setpoint toggle each 10 seconds as the ball had not enough time to stabilize, so hard oscillations are observed in every approach. Hysteresis and Feedback control present very similar results, while PID reaches the highest peak response and noise values. We wanted to observe its behaviour increasing the toggle time from 10 to 20 seconds, and the obtained results are shown below.



(a) PID for 10 seconds



(b) PID for 20 seconds

Figure 6: Comparison of the PID approach toggling setpoint every 10 or 20 seconds.

This modification seems to help the stabilization of the system, especially on the lower setpoint, when the ball is closer to the sensor. However, the oscillations are not eliminated and there are some peak values of the distance that should be corrected. The time response of the system does not seem to be affected. We can say that the PID approach behaviour improves a lot when we consider a higher toggle time of the setpoint. Its performance could be considered the best among the studied algorithms.

The time at the laboratory room was limited, we did not have enough time to try all the proposals that we had on mind, so suggestions of possible future works and improvements are explained here.

Regarding the impossibility of finding a value for the fan speed to keep the ping-pong ball in a constant height, we considered the possibility of implementing the program with *Timer1*. It has a total of 16 bits and could consider a wider range of values, providing more precision to the fan's input and perhaps achieving a concrete value that lets the ball remain at a constant height.

Referring to the PID tuning, it is likely that with more trials we may find a better set of values for K, T_i, and T_d that mitigate the oscillations around the setpoint and also perform a faster settling time when converging into the setpoint.

To conclude, we could also consider the possibility of trying different types of controls and formulas than the used ones, in order to achieve a better performance of the system.