SIL/HIL Control System Demo

Ricardo Ampudia

July 24, 2020

Contents

Note: Instructions on how to run the demo are found in the *Demo* section under the C and VHDL subsections respectively.

1 Overview

This is document explains the implementation and summarized results of a Hardware-in-the-loop / Software-in-the-loop (HIL / SIL) demo for a vision-in-the-loop system that performs spot tracking.

The system, depicted in figure 1, is composed of a plant (webcam) that can rotate on two separate axes, called pan and tilt. This webcam produces a video stream and by means of an image processing function it calculates the angle between the center of an object of a specified color and the center of the camera. This angle becomes the set point of the system and the controller of the motors allows the webcam to reach the point.

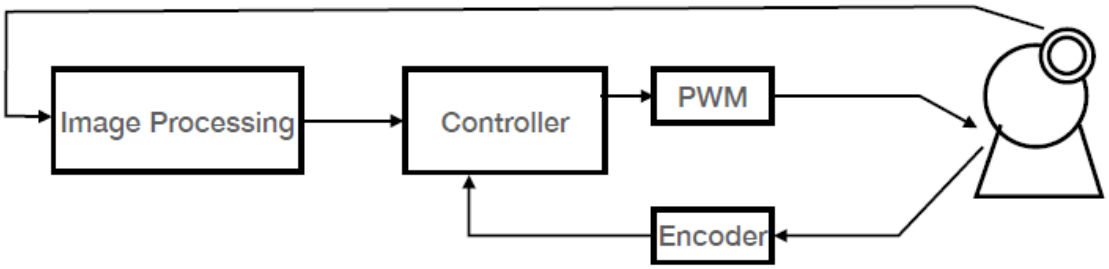
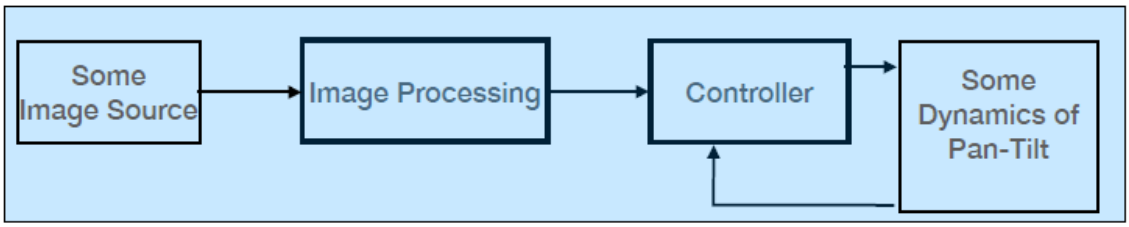


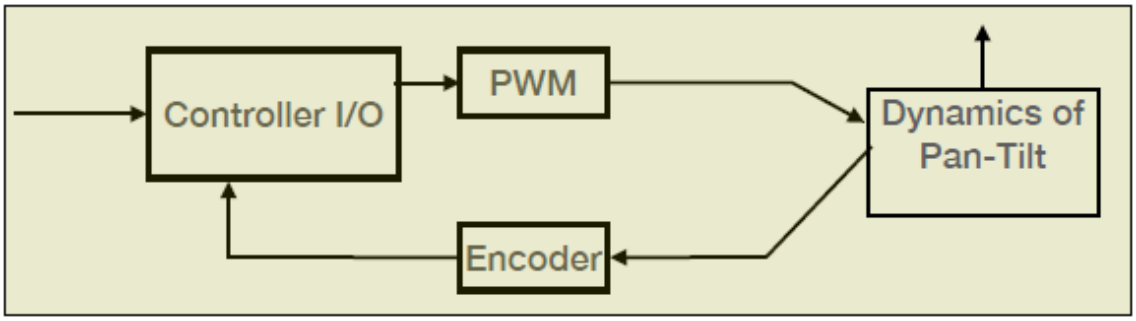
Figure 1: Overall model of the system

It consists of two parts (see figure 15), a complete SIL demo using a simulation in C that allows to process a live video stream, provide a set point using image processing and interact with the simulated plant and controller. Additionally, a HIL demo version was built in VHDL-2008 and simulated with ModelSim to test fully synthesizable sensors and actuators modules in a close to real situation. By doing so, it is technically still a SIL simulation that can be tested as a HIL by implementing its parts in and FPGA and establishing communication.

In particular, the SIL simulation in C will give a good indication if the video streaming, image processing and controller would be effective to control the plant in a real setting. The HIL in VHDL will serve to test that the hardware components would work correctly under the operating conditions. Together, both parts provide a solid framework that will give confidence that the parts designed such as set-point detection, sensor readers, controller and actuators would work correctly when the whole system is implemented physically.



(a) C model



(b) VHDL model

Figure 2: System model design

A summary of the results is presented in the *Demo* section, for a detailed explanation of how each part was implemented refer to the *Implementation* section.

2 Demo

2.1 C

Note: The demo part in C is found in the **/C** folder and can be compiled using the command "*make re*". After that, the program is executed by doing the command "*system_demo path_to_webcam_driver*". It is currently configured to detect light green color to determine the set point of the system.

When running the C demo, if an object of the specified color is detected, the center of this object and the angular distance, over both axis, to the center of the camera is printed on the screen. Also, all the inputs and outputs of the control-loop (controller and plant) are printed on the screen, as well as the pan and tilt errors calculated in the controller. Figure 3 shows an example of the program execution.

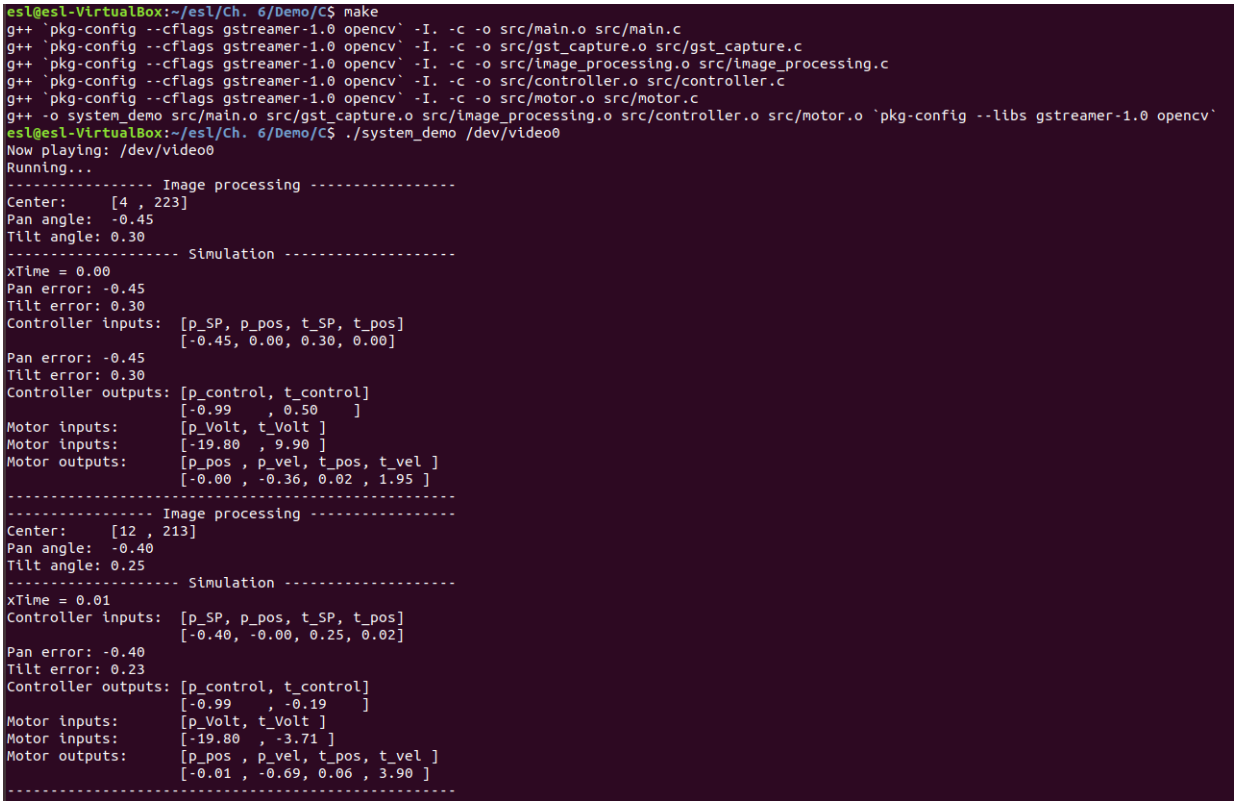
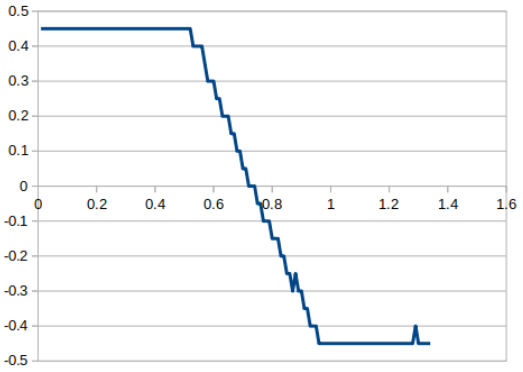


Figure 3: Example of an execution of the C program

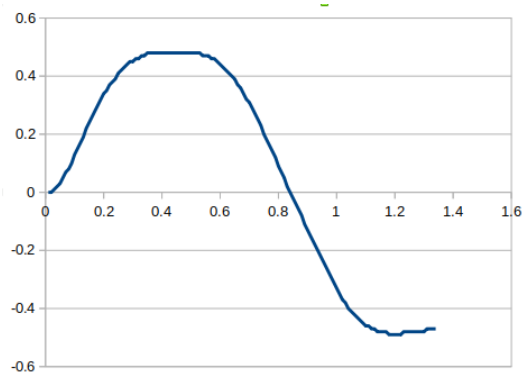
The program also registers and stores the values in CSV files to produce graph afterwards. Each file is composed of 2 columns, the first one is the value of the variable *xTime*, that acts as a timer incrementing by 0.01 each iteration and the second column values is described in the name of the file as follows:

- **ctrlIn[PAN/TILT]:** The input parameter of the controller, it is the value produced by the image processing function.
- **ctrlPosition[PAN/TILT]:** The current position of the plant, produced by the plant simulation.
- **ctrlError[PAN/TILT]:** The error difference between the set point and the feedback.
- **ctrlOutput[PAN/TILT]:** The controller output signal.
- **motVelocity[PAN/TILT]:** The velocity obtained from the plant simulation.
- **motOutput[PAN/TILT]:** The new position obtained from the plant simulation.

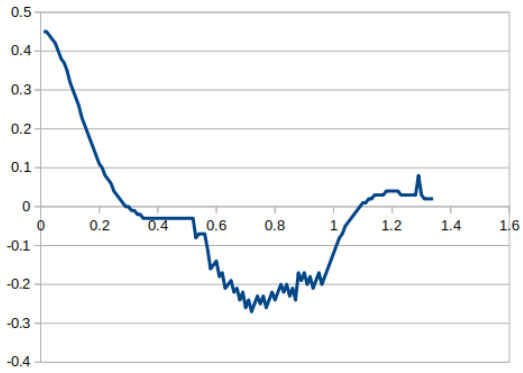
Figures 4 and 5 show the graphs that can be obtained after an execution of the demo program.



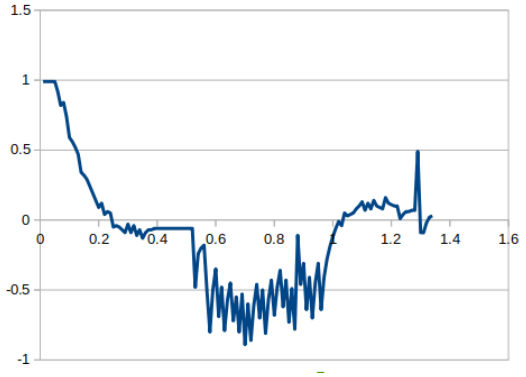
(a) Controller angle input



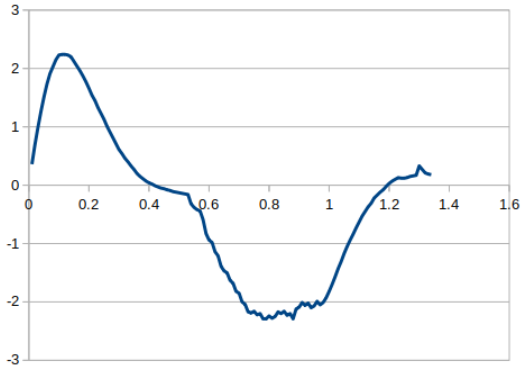
(b) Controller position input



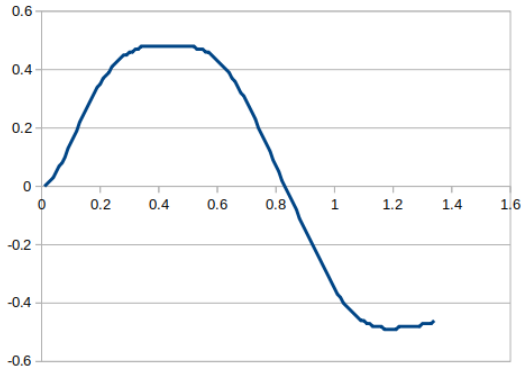
(c) Controller error



(d) Controller output

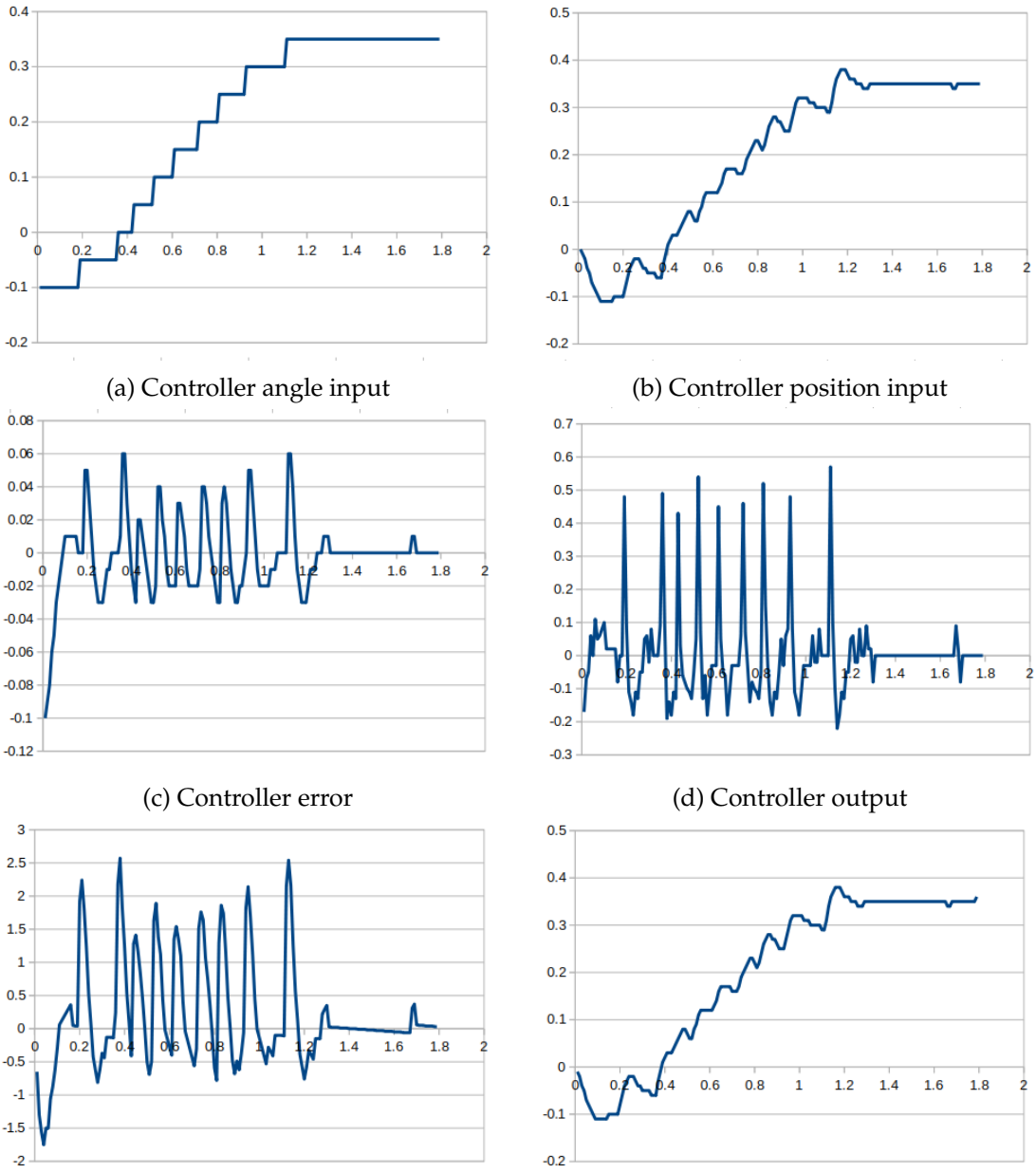


(e) Motor output velocity



(f) Motor output position

Figure 4: Pan graphs obtained by the C program



(a) Controller angle input

(b) Controller position input

(c) Controller error

(d) Controller output

(e) Motor output velocity

(f) Motor output position

Figure 5: Tilt graphs obtained by the C program

It can be seen that from both the Pan and Tilt cases (figures 4 and 5) the motor output position (b) follows the controller angle input (a) closely and the controller error (c) is close

to zero in the steady state where the angle set point remains steady for some milliseconds.

2.2 VHDL

Note: The demo part in VHDL is found in the **/VHDL** folder and can be executed by loading the *compile_demo.do* script and the running the simulation of *testbench_demo.vhd* for 1360 ms. In case the compile script doesn't work the compilation order should be the following: parameters.vhd, pid_to_pwm.vhd, pwm.vhd, average.vhd, plant.vhd, pulse_gen.vhd, encoder.vhd, pos_to_rad.vhd, teset_demo.vhd, testbench_demo.vhd.

As mentioned in the previous subsection, a closed loop version of the system was implemented in C and all the signals out of each block were logged in a CSV file. The data obtained from the controller's output was included in a testset in VHDL representing the controller. If the motion profiles of the motors position as measured by the encoder match with those obtained in C, we can conclude that the Software in the Loop simulation is appropriate and that the developed PWM and Encoder modules would work in a physical system.

Figures 4 a, d and 5 a, d from the previous subsection show the set points and PID output profiles that were obtained after doing some careful tests. The PID controller output for Pan and Tilt was translated to VHDL by creating a testset that sends the desired value every 10 ms to the respective next block, and an offset of 30ms was included to allow the simulation to initialize all of its components:

```
t_pid_out <= 0.0,
-0.17        after        30.0        ms,
-0.07        after        40.0        ms,
-0.05        after        50.0        ms,
 0.06        after        60.0        ms,
...

p_pid_out <= 0.0,
0.99        after        30.0        ms,
0.99        after        40.0        ms,
0.99        after        50.0        ms,
0.99        after        60.0        ms,
...
```

The demonstrator was then executed by simulating *testbench_demo.vhd* for 1360 ms and figures 6 and 7 show the results obtained:

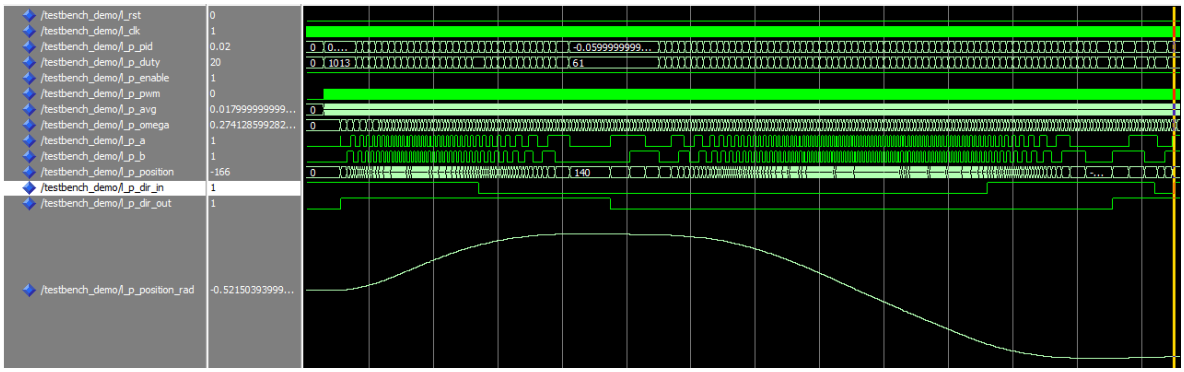


Figure 6: Pan open loop signal chain (PID to Encoder Position) - ModelSim

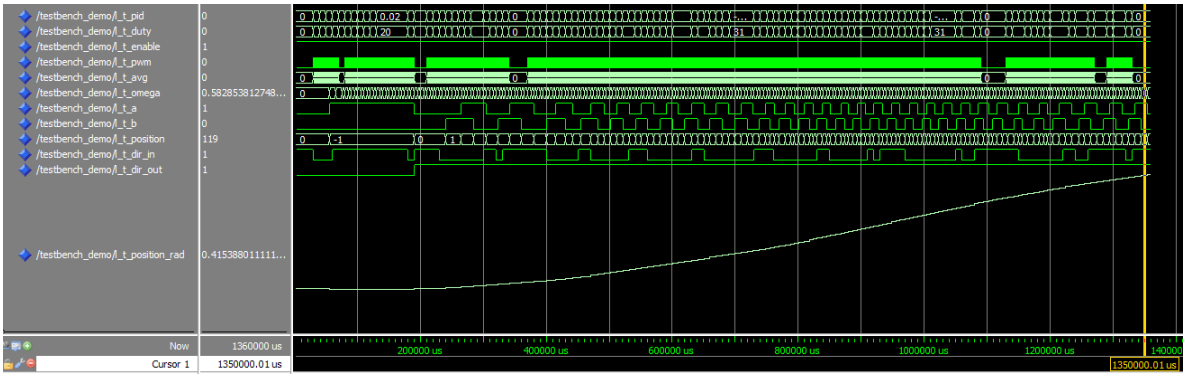


Figure 7: Tilt open loop signal chain (PID to Encoder Position) - ModelSim

It can be seen that the complete chain between between blocks works as expected. From top to bottom, the PID signal from the testset is converted to the PWM level equivalent and the PWM signal is generated, then the average block sends the moving average of the PWM signal to the plant where the velocity response (omega) is obtained. The pulse generator block creates the varying pulses of Channels A and B according to the velocity of the plant and the encoder registers the incremental pulses. The final signal of the chain, position in radians, is displayed in an analog format to easily view the resulting motion profile.

2.3 Results and Analysis

The resulting motion profiles from VHDL (figures 6 and 7 were compared to those obtained from the C simulation (figures 4b and 5b). It can be seen from the plots that the

profiles from the C and VHDL simulations match very closely in shape, range and in values.

From the plots and the table obtained it can be seen that the VHDL and C simulation give close results to each other and serves as a validity check that the overall closed loop system was well implemented. Figure 8 shows side by side the graphs from the motor position output and in the C and in VHDL simulations. It can be seen that the Pan profiles closely match each other. Although the values registered in both cases are very close, a noticeable difference between the two approaches is seen in the Tilt motor where the VHDL curve is very smooth compared to the one in C. This can be explained due to the fact that the chain of blocks in VHDL is longer than the one in C and some of its parts act as a low pass filter that dampen the peaks of the PID control signal. This is particularly noticeable in the Tilt motor which has very sensible dynamics (low moment of inertia and low friction causing quick changes in angular velocity) compared to the Pan motor.

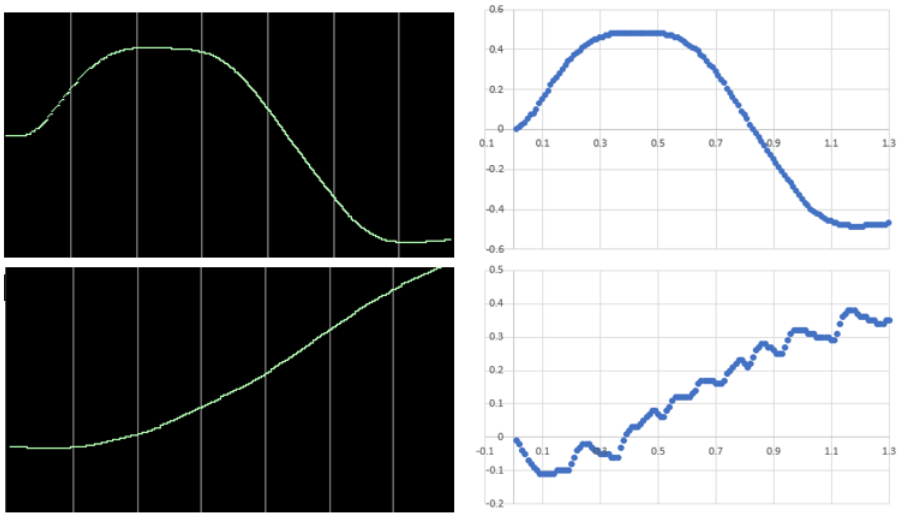


Figure 8: ModelSim vs C motor position profiles of Pan (upper half) and Tilt (lower half)

The results of the demo show that the parts implemented in C (Image processing and controller) as well as the parts implemented in VHDL (Encoder and PWM) function correctly within the whole system. Now a triple model-based verification with some real time characteristics has been made in 20sim, C and VHDL which gives us a very good confidence that the physical setup could work with minor tuning of the communication parts between modules (depending on the actual hardware used).

During implementation, it was also noticed that some assumption of the DSE were cor-

rect. For example, the development time for the C parts was considerably more flexible and faster than in VHDL. Also that VHDL implementation are better suited for real-time tasks since simulation is ModelSim is already suited to work naturally in the range of nanoseconds.

3 Appendix: Detailed implementation of parts

The code for both parts of the demo is located in the folder **Demo**.

3.1 C

The code relative to the C part of the design is located in the folder **C**.

3.1.1 Image acquisition

The image acquisition was made assuming a the choice of using a Logitech C250 webcam [5], as its the webcam that was used in the previous years in the lab. The code for the acquisition of frames from the camera is written in the file *src/gst_capture.c*. Done with **GStreamer** library, the program reproduces the pipeline of figure 9 and the behaviour of the following command line:

```
gst-launch-1.0 -v -e v4l2src device=/dev/videoX !
image/jpeg,width=320,height=240,framerate=30/1 ! jpegdec ! videoconvert !
video/x-raw,width=320,height=240,framerate=30/1,format=RGB ! appsink
```

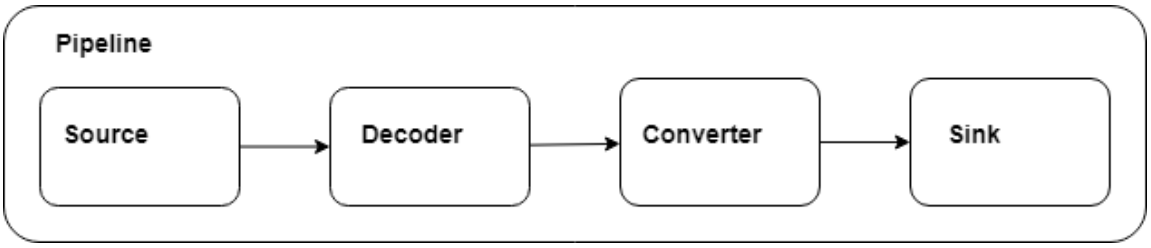


Figure 9: Image acquisition pipeline

The images are extracted at a resolution of 320x240 with a framerate of 30 frame per second. This choice of resolution was made because it is a resolution that can be processed by the majority of webcams on the market, including the Logitech webcam used as model for this design. The resolution is also of high quality enough to correctly detect objects in

the frame.

To return to the pipeline, the source webcam frames are decode in jpeg images, then converted to raw images of format RGB in order to have an easier and better detection of colors as mentioned earlier in section **??**. The converted frames are then pushed into the sink to be used by the image processing function.

3.1.2 Image processing

The image processing function (file *src/image_processing.c*) receives a RGB image and has to detect an object depending on a range of values. This range of values can be determined using the program **tune**, present in the folder *colorTune*. This program, made with the library OpenCV and compiled with the command line: *g++ tuneRange.c -o tune 'pkg-config –cflags –libs opencv'*. The program opens two windows and lets the user adjust the sliders until only the desired object to be detected is visible on the second window (see figure 10 for example).

When the thresholds values are determined, the script *colorRangeCustom.sh* can be called with the 6 values as arguments (in the same order as in the "Object Detection" window of the tune program). This script sets the values in the code of the final program. To reset the values to the default thresholds measured during testing, the script *colorRangeDefault.sh* can be called. Note that after using one of this script, the main program must be recompiled. **For now the color configuration is set to a light green color**.

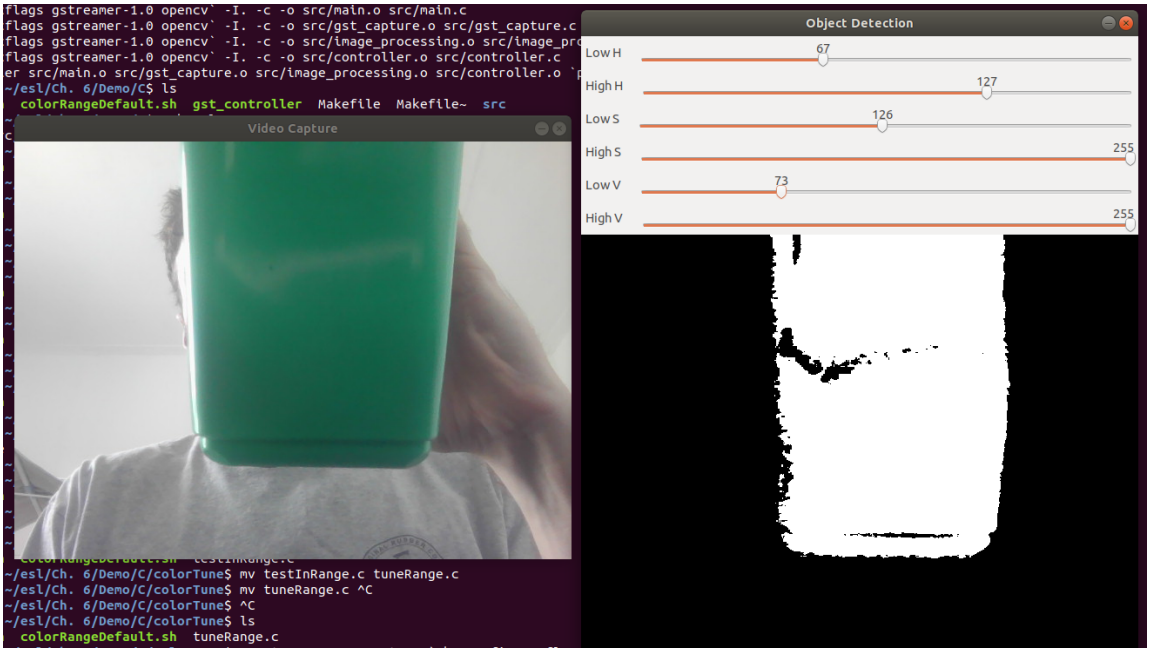


Figure 10: Tune program example

Using the OpenCV library, the image processing function uses the thresholds values, after converting from RGB format to HSV, to extract the pixels that are in the defined range. The center of the detected object is retrieved, then the pan and tilt angles of the object relative to the center of the frame are calculated using the FOV (Field Of View) defined in the hardware specification of the Logitech C250 webcam.

The image processing function returns the value of the pixels corresponding to the the center of the detected object. For the spot tracking application to work, the set point must be provided in radians (or degrees) so that the motor can turn and the position change be reported by the encoder. To do so a conversion between the pixel value and the relative pan and tilt angles of the object must be made. This can be done by using the concept of Field of View (FOV) [1] which describes the angular range that a camera can cover (see figure 11 for clarity).

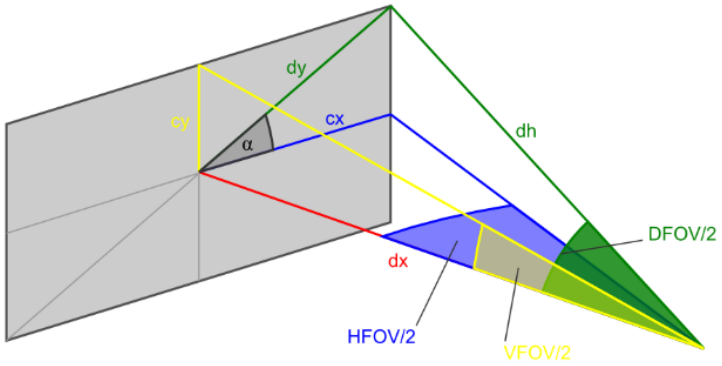


Figure 11: Screen FOV [1].

The Logitech camera chosen has a Diagonal FOV (DFOV) of 63° [5] and the resolution that will be used correspond to an aspect ratio of 4:3. By using this information and trigonometric formulas the Horizontal and Vertical FOV of the camera selected can also be known:

$$HFOV = 2tan^{-1}\left(tan\left(\frac{63}{2}\right)cos(tan^{-1}\left(\frac{4}{3}\right)\right) = 52.23degrees \tag{1}$$

$$VFOV = 2tan^{-1}\left(tan\left(\frac{63}{2}\right)sin(tan^{-1}\left(\frac{4}{3}\right)\right) = 40.37degrees \tag{2}$$

Now that the HFOV angles are known, the pan and tilt angles conversion can be done as follows:

$$Pan(rad) = \left(c.x - \frac{320}{2}\right)px\left(\frac{52.23degrees}{320px}\right)\left(\frac{\pi rad}{180degrees}\right) \tag{3}$$

$$Tilt(rad) = \left(c.y - \frac{240}{2}\right)px\left(\frac{52.23degrees}{240px}\right)\left(\frac{\pi rad}{180degrees}\right) \tag{4}$$

Where c.x and c.y are the calculated center of the detected object in the x and y coordinates respectively and the numbers 320 and 240 are the width and the height in pixels of the video resolution.

Finally the angle detected by the camera was found to be very sensitive and the resolution higher than the required for the application. For this reason the angle in radians was truncated to two decimals and restricted to step level of 0.05rad $\approx 3°$ to achieve a more stable performance of the set point. Due to the range of values obtained from the HFOV (approximately 52.23° or 0.9 rad) and VFOV (approximately 40.37° or 0.7 rad), the

resulting range of the set point for pan and tilt is:

- Pan = [-0.45, -0.40 , -0.35, ..., 0.40, 0.45]
- Tilt = [-0.35, -0.30 , -0.25, ..., 0.30, 0.35]

Note: For a spot tracking application the set point should be the current pan/tilt angular position (as measured by the encoder) plus the pan/tilt angle detected by the camera. Since in the simulation the motor movement won't cause the set point to be modified, it was decided to simply use the Set Point as the Pan/Tilt angle measured by the camera directly. This does not simplify the control problem but allows to observe more clearly and analyze the response obtained in simulation.

3.1.3 Controller

The PID controller is created using the 20sim JIWY model of figure 12. From this model, the C code of the pan controller and tilt controller is generated and regroup in one source file called *controller.c*. Figure 13 shows the sub-models of the servo controllers.

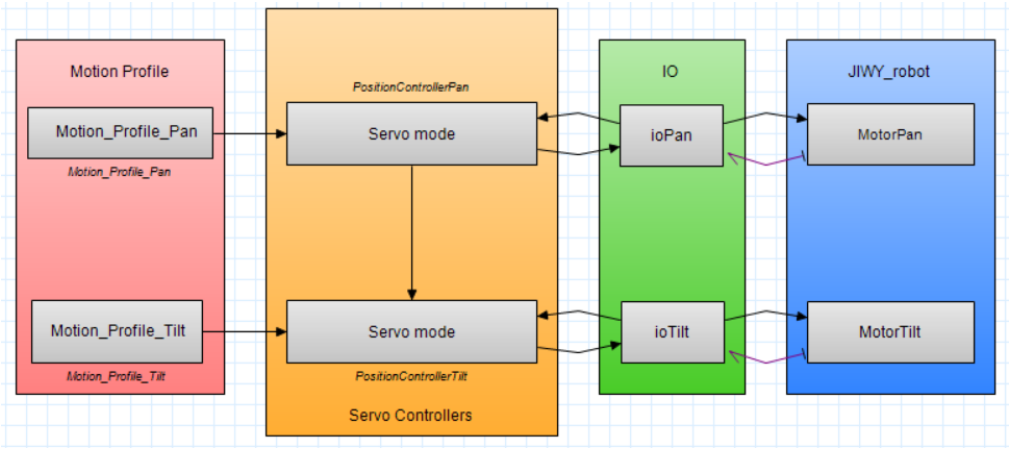
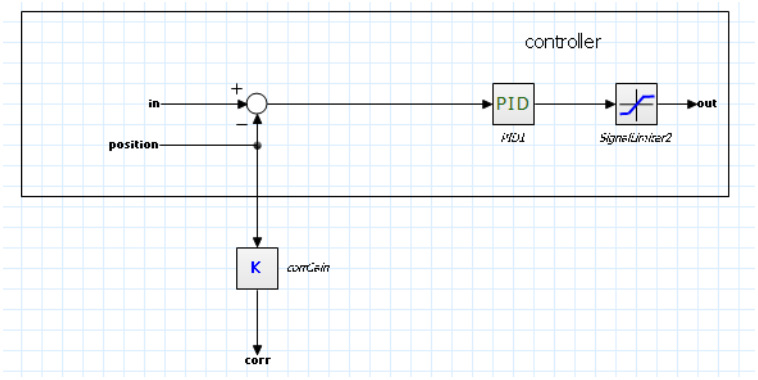
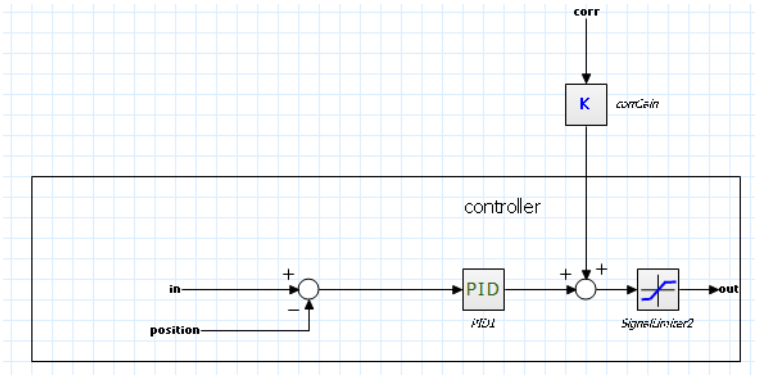


Figure 12: JIWY model in 20sim



(a) Pan controller



(b) Tilt controller

Figure 13: Pan and tilt controllers in 20sim

Using the pan and tilt angles measured by the image processing function, and the current position of the plant, the controller calculates a velocity for both axis, limited to a range of -0.99 to 0.99. The parameters for this calculation depends on the axis, and are shown in figure 14, and the PID equation are the following:

$$
\begin{aligned}
factor &= 1/(sampletime + tauD * beta);\\
uD &= factor * (tauD * previous(uD) * beta + tauD * kp *\\
&\quad (error - previous(error)) + sampletime * kp * error);\\
uI &= previous(uI) + sampletime * uD/tauI;\\
output &= uI + uD;
\end{aligned}
$$

Name	Val...
corrGain\K	0.0
PID1\beta	0.17
PID1\kp	2.6
PID1\tauD	0.05
PID1\tauI	9.0
SignalLimiter2\maximum	0.99
SignalLimiter2\minimum	-0.99

(a)

Name	Val...
corrGain\K	0.0
PID1\beta	0.001
PID1\kp	1.6
PID1\tauD	0.05
PID1\tauI	10.5
SignalLimiter2\maximum	0.99
SignalLimiter2\minimum	-0.99

(b)

Figure 14: Controller pan (a) and tilt (b) parameters

The sample time of the controller is not precised in the parameters table but a step size of 0.01 was used. In order to have a more realistic behaviour of the simulation with a camera that moves only when significant change exists, the precision of the input values has to be reduced. This is done by rounding the pan and tilt angles and positions to 2 decimal places.

3.1.4 Plant simulation

To simulate the behaviour of the plant, the JIWY model of figure 12 is reused. The motor sub-model is the same for the pan and tilt axis (figure 15), but values of the parameters differ (figure 16). The C code is again generated and written into the same C file, called *motor.c*.

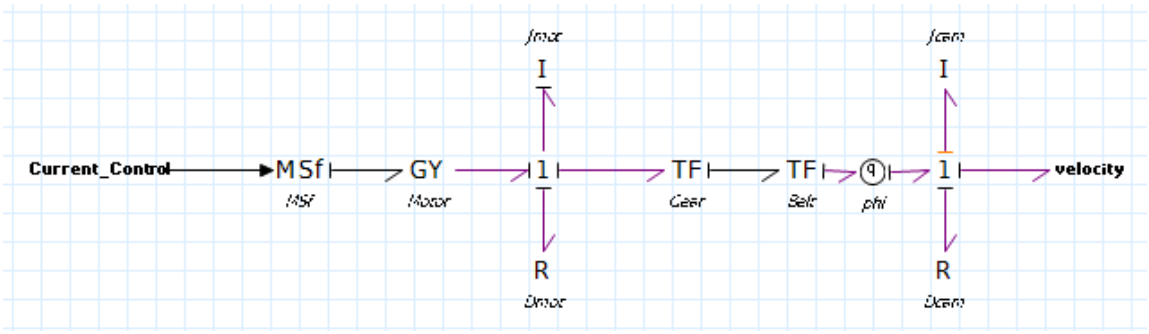
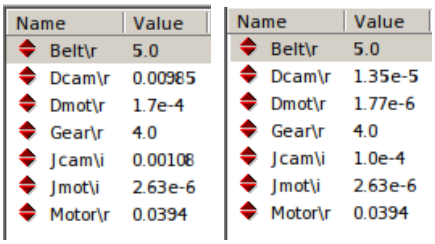


Figure 15: Motor model in 20sim



Name	Value
Belt\r	5.0
Dcam\r	0.00985
Dmot\r	1.7e-4
Gear\r	4.0
Jcam\i	0.00108
Jmot\i	2.63e-6
Motor\r	0.0394

(a)

Name	Value
Belt\r	5.0
Dcam\r	1.35e-5
Dmot\r	1.77e-6
Gear\r	4.0
Jcam\i	1.0e-4
Jmot\i	2.63e-6
Motor\r	0.0394

(b)

Figure 16: Motor pan (a) and tilt (b) parameters

The motor has two inputs parameters that are the outputs of the controller. To amplify the velocity out of the controller, the pan and tilt velocities are both multiplied by 20. The plant produces two outputs for each axis. The first one is the velocity of the motor calculated by the equations of 20sim. This velocity is then used to calculate the new position of the plant using the formula:

$$position = old\_position + velocity * STEP\_SIZE;$$

The value *STEP_SIZE* used is the same one as in the controller, 0.01. To keep the same precision as the controller, the output positions for the pan axis, and the tilt axis, are rounded to 2 decimal places.

3.2 VHDL

Note: The VHDL simulation was built in separate modules for each subsystems. For convenience, a *parameters.vhd* file was create with the concentrated adjustable parameters for the entire project.

Figure 17 shows a schematic of all the subsystems developed for the simulation in VHDL and their connecting signals. It is meant as a general layout to bring clarity of the main connections between blocks and the range of the signals (where appropriate) but it is not a complete schematic of all the connections in the design.

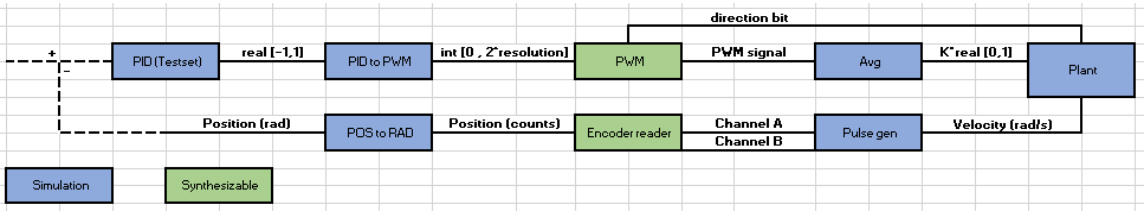


Figure 17: VHDL schematic of subsystems

3.2.1 Controller

The DSE exploration done previously indicated that a controller would be more appropriate in C, for this reason, the division was drawn in the PID controller block between the C and the VHDL parts and the controller in VHDL was simulated as a testset. The simulated controller in VHDL does not perform any calculations in this part and outputs hard-coded values in the range of -1 to 1 at specific times as the real PID controller would. By logging the output data of the C controller and using it as the testset in VHDL, the position profile obtained from the *pos_to_rad.vhd* is expected to be close to the one obtained in the C simulation and thus would confirm the validity of the simulation.

After a C simulation was performed, the output of the controller was logged in a CSV file and the values were formatted to provide the same output in VHDL:

```
pid_out <=  0.0,
            0.080       after       40.0       ms,
            0.370       after       50.0       ms,
           -0.990       after       60.0       ms,
           -0.010       after       70.0       ms,
            0.110       after       80.0       ms,
            0.990       after       90.0       ms,
            ...
```

3.2.2 PWM

The PWM module developed was designed in a flexible way that would allow to easily configure the design to work for different hardware components. The two configurable parameters defined were the PWM frequency desired and the resolution:

```
--Parameters for the PWM module
   CONSTANT p_pwm_freq    : INTEGER := 10_000;     --Desired PWM frequency (Hz)
   CONSTANT p_resolution  : INTEGER := 10;
```

With the parameter of 10,000 Hz the PWM module generates a signal with a period of $100\mu s$. The resolution of 10 in this case indicates that the duty cycle has 1023 levels (2^{10}), meaning that the average voltage will increment in steps of $1/1023 \approx 0.001$ (multiplied by the rated voltage of the signal). For the assumed H-Bridge *VNH2SP30-E* the PWM frequency specified is well within the allowed range between 0 and 20kHz [4]. For the application of motor position control, a resolution of (2^{10}) is sufficient to provide a wide range of control levels. Figure 18 shows a running example of the implementation with a duty cycle of 0.55.

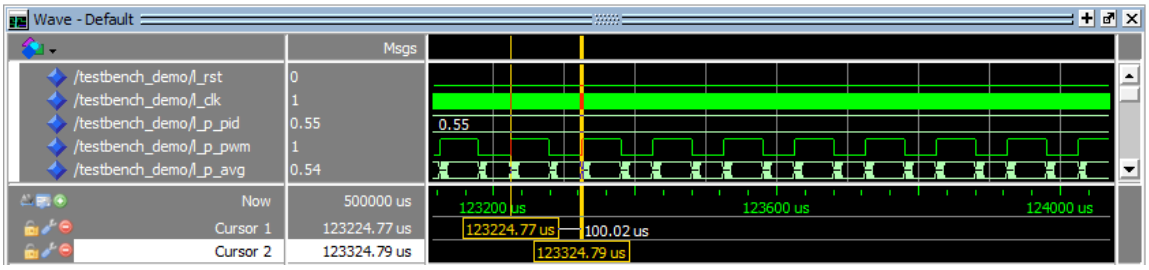


Figure 18: PWM with duty of 0.55 - ModelSim

In VHDL, the period of the PWM signal is calculated as the number of ticks equivalent to the desired time delay with respect to the internal clock frequency (a global parameter). Two counters keep track of the elapsed time for the PWM period and PWM resolution period to execute internal logic. The conditions evaluated within are to set the PWM signal output to 1 or 0 according to the current count and duty cycle, and once a complete PWM period has passed the counters are reset.

```
--Clk cycles in one pwm period
CONSTANT pwm_period     : INTEGER := clk_freq/pwm_freq;
--Clk cycles in one resolution step
CONSTANT pwm_period_res : INTEGER := pwm_period/2**resolution;
```

Since this module was designed to be programmed in an FPGA, the code was elaborated to be synthesizable and the input/outputs ports were given a range according to the set parameters for area efficiency of the bus. In a real hardware implementation this value would have to be matched to the width of the bus port that sends/receives data, but here it is not a concern since we have no hardware limitations.

3.2.3 Encoder reader

The encoder reader design uses 4x encoding which means that measures a pulse in both the rising and falling edges of the signals from either of its channels. This techniques achieves the highest possible resolution compared to 2x and 1x encoding. A configurable parameter was added to adjust to the pulses per revolution of the connected hardware making the implementation flexible. A parameter of 2000 pulses per revolution was selected corresponding to a high resolution encoder.

```
GENERIC (pulses_per_rev   : integer := 2000);
```

The encoder reader assumes the signal coming from figure 19 depending on the direction of rotation of the encoder. With this signals a state pattern can be recognized when going in the direction were Channel A leads B (state is formed with the value of ch_a concatenated with ch_b): 00, 10, 11, 01,... Another pattern can be recognized when going in the counter direction were Channel B leads A: 00, 01, 11, 10,... With this information, by knowing the previous state and the next state it is possible to know the direction of the encoder and if the counter should be incremented or decremented.

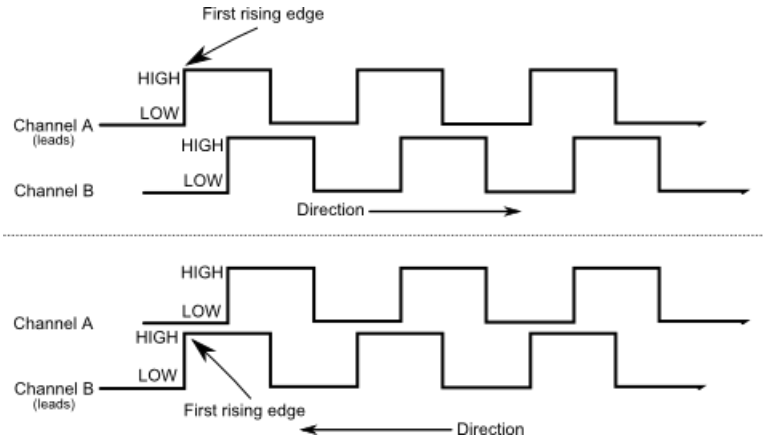


Figure 19: Signal of the channels depending on direction

To obtain the current state, signals from channel A and B are assigned immediately into the variable *state*. An intermediate variable *old_state_temp* is used to store and assign to *old_state* the previous state by means of a signal which places a register between each value read in the rising edge of the clock. A simple case statement is used to identify the patterns, set the direction bit and increment or decrement the position counter accordingly. The encoder was designed to be an incremental encoder (indicates the relative position with respect to a set origin) and is bounded to allow a maximum number of turns to avoid overflowing the position variable which has a set range.

```
state := a & b;
old_state := old_state_temp;
old_state_temp <= state;
```

Figure 20 shows that depending on the input signals of channels a and b, the position and direction changes accordingly.

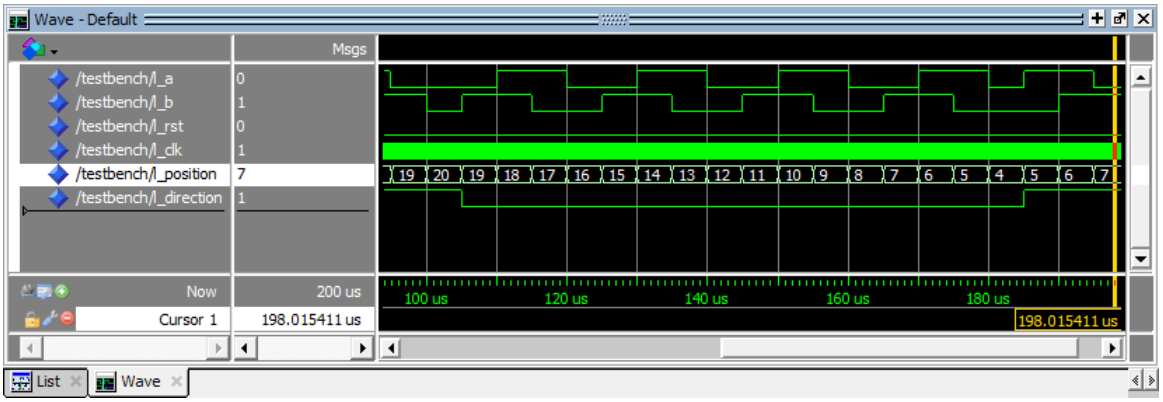


Figure 20: CW and CCW simulation of encoder reader - ModelSim

3.2.4 Plant dynamics

To simulate the functionality of our PWM module (for motor speed control) and Encoder reader (for motor position sensing) in a closed loop, the plant dynamics of a DC-motor must be also included in a separate module. This module doesn't need to be synthesizable since it is only for the purpose of simulation.

A DC-motor can be modelled as shown in figure 21. The following model and equations are obtained from [3].

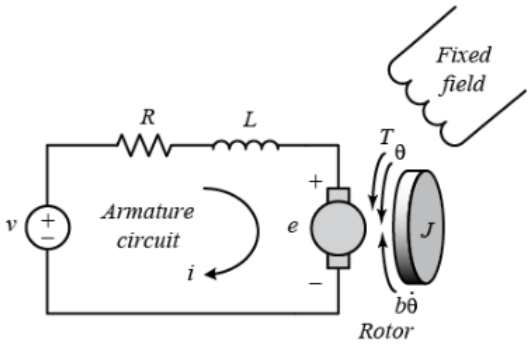


Figure 21: DC motor model [1]

where:
J : moment of inertia of the rotor [kg.m^2]
b : motor viscous friction constant [N.m.s]
e : back emf (electromotive force) Ke : electromotive force constant [V/rad/sec]

T : motor torque Kt : motor torque constant [N.m/Amp]
R : electric resistance [Ohm]
L : electric inductance [H]
θ : angular position [rad]
$\dot{\theta} = \omega$: angular velocity [rad/s]

Applying Newtons second law and Kirchoff's voltage the following equations can be derived:

$$J\frac{d^2\theta}{dt} + b\frac{d\theta}{dt} = K_t i \tag{5}$$

$$L\frac{di}{dt} + iR = V - K_c \tag{6}$$

Rewriting $\frac{d\theta}{dt} = \omega$ and rearranging the constants we can arrive at the following equations:

$$\frac{d\omega}{dt} = \frac{1}{J}(K_t i - b\omega) \tag{7}$$

$$\frac{di}{dt} = \frac{1}{L}(V - K_e\omega - iR) \tag{8}$$

All the constants that characterize the model can be obtained either through measurements or from the motor specification, and the voltage is the input of the plant. Then ω and i are the only unknowns left and can be calculated numerically with Euler's method by using equations *3* and *4* to obtain the value of the derivative. The following segment of VHDL code performs the described method:

```
dw <= (1.0/J)*(Kt*i - b*w);--dw=(1/J)*(Kt*i-b*w);
di <= (1.0/L)*(V*K - i*R - Ke*w); --di=(1/L)*(V-i*R-Ke*w);
w <= old_w + dw*dt_period*speedup_sim; --w=w+dw*dt;
i <= old_i + di*dt_period*speedup_sim; --i=i+di*dt;
```

Additionally, 2 new parameters were included to the equations. The first one, K, represents the power amplifier (H-Bridge) used for the motor and multiplies the average of the PWM control signal it receives (V). The second parameter *speedup_sim* can speed-up the simulation by a given factor specified in the *parameters.vhd* file if desired. Since the simulation respects the real time behavior of the physical plant, and the steady state response of the motor occurs in the order of seconds, this parameter is useful and makes it easier for ModelSim to handle (but for now it is set to 1). Figure 22 shows the step response of a

motor simulated in ModelSim.

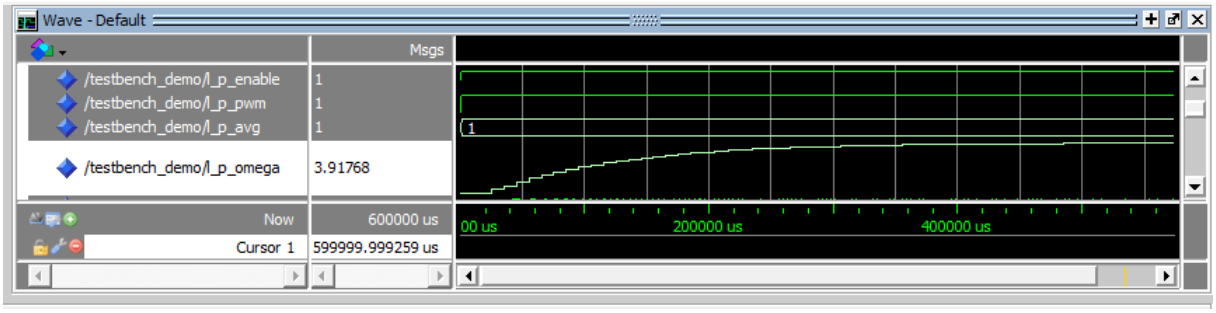
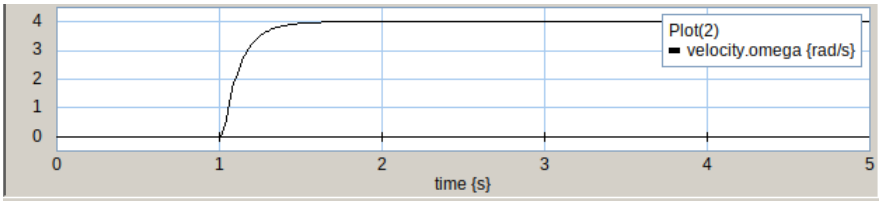
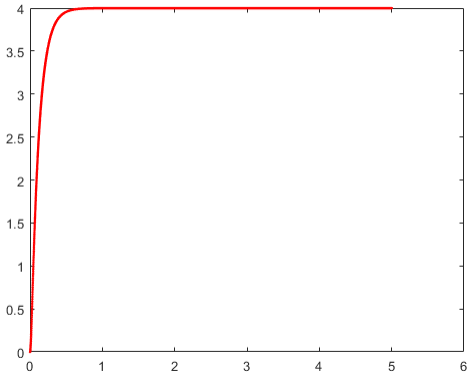


Figure 22: DC motor step response simulation

The DC-motor parameters were configured according to the 20sim model provided (bond graph representation) both for the pan and tilt motors and their values can be found in the *parameters.vhd* file. Figures 23 and 24 show that both models have an almost identical step response in terms of time constant and steady state values.

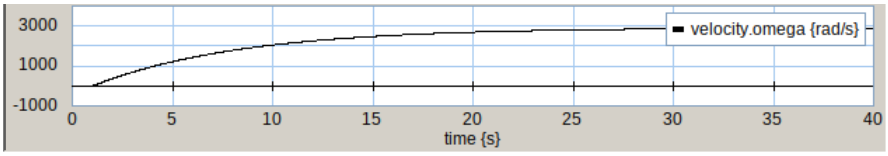


(a) Pan - 20 sim

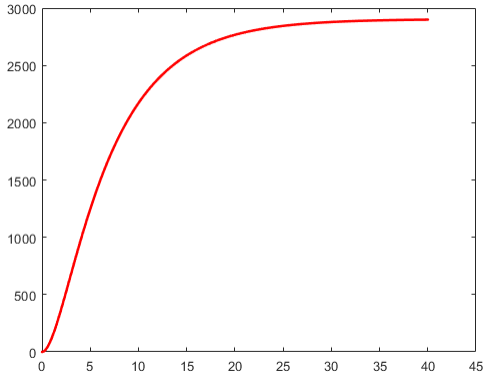


(b) Pan- Matlab

Figure 23: Step response of DC-motor bond graph vs transfer function model



(a) Tilt - 20sim



(b) Tilt - Matlab

Figure 24: Step response of DC-motor bond graph vs transfer function model

3.3 Connecting blocks

In order to test the closed loop chain from Controller to PWM to Plant to Encoder, some connecting blocks had to be developed. The VHDL blocks were designed to be as close as possible for the simulation to behave as a real system.

3.3.1 PID to PWM

Assuming a floating point signal in the range from -1 to 1 coming from the PID controller, this block (*pid_to_pwm.vhd*) converts it to the closest step level integer according to the chosen resolution for the PWM module by means of a simple multiplication and rounding. For example a controller signal of 1 corresponds to 1023 in a PWM with 10 bits of resolution and a value of 0.4 to a level of 409. According to the sign of the PID control signal, a new direction bit is sent through the bus to control the direction of the H-Bridge.

```
IF pid < 0.0 THEN
    direction <= '0';
    duty <= integer(-1.0*pid*Real(2**resolution-1));
ELSE
```

```
        direction <= '1';
        duty <= integer(pid*Real(2**resolution-1));
    END IF;
```

This block does not need to be synthesizable as it is only for simulation purposes and the previous logic could easily be implemented in the C-Controller.

3.3.2 Average block

Since the dynamics of a DC-motor are typically slow compared to the frequency of a PWM signal, it acts as a low-pass filter and the motor only *perceives* the average of the input voltage signal. For this reason, an average block (*average.vhd)* was implemented to calculate the moving average of a PWM signal, that is, the fraction of time it is set to 1 with respect to its period over a sampling period. It is a value between 0 and 1 representing that represents the analog value (before power amplification) that goes into the motor input. For this block, two parameters are mainly relevant:

```
--Parameters of the PWM Average block
    --Number of samples used to calculate average of PWM signal
    CONSTANT samples  : INTEGER := 500;
    --Number of samples per pwm period to calculate PWM (used for sampling period)
    CONSTANT samples_per_period  : INTEGER := 100;
```

Modifying this parameters has an effect on the reflected value of the average and tuning them is important as it has an effect on the plant dynamics. A high number of samples as the effect of making the average react slowly to new values and so does a low number of samples gives less weigh to previous values received. The number of samples should be higher than the PWM signal period to guarantee the precision of the accuracy of the measurement. It was found through testing that a good combination of values was 100 samples per period and with the moving average calculated with 500 samples it takes 5 PWM periods to reach a steady measurement.

The VHDL implementation is simple a simple moving average that every sampling period adds the value of the new sample to a cumulative sum variable while subtracting the oldest value of a shift register storing all the samples (the length of the shift register is determined by the *samples* parameter). The average is updated by dividing the cumulative sum by the number of samples.This block does not need to be synthesizable as it is only for simulation purposes.

```
    avg <= Real(sum) / Real(samples);
```

3.3.3 Pulse generation

The pulse generator block (*pulse_gen.vhd*) simulates a rotatory encoder attached to a motor shaft. It will provide the input signal of channels a and b to the encoder reader and generates the same amount of pulses that a real a motor rotating at a particular speed would after specifying some parameters. It requires the output of the plant (velocity of the motor in rad/s) and the parameter of pulses per revolution of the encoder. With this values and the clock frequency a *pulse_period* is calculated with the following formula in VHDL:

```
pulse_period <= clk_freq/INTEGER(omega*Real(pulses_per_rev)/(2.0*pi));
```

Then, every time a pulse period has passed, the outputs of channels a and b are set depending on the current state and direction. Figure 25 shows that depending on the input speed (*omega*) the frequency of the channels a and b is set with one leading the other according to the sign of the input. As expected, a higher revolution count means that more pulses will be generated and vice versa.

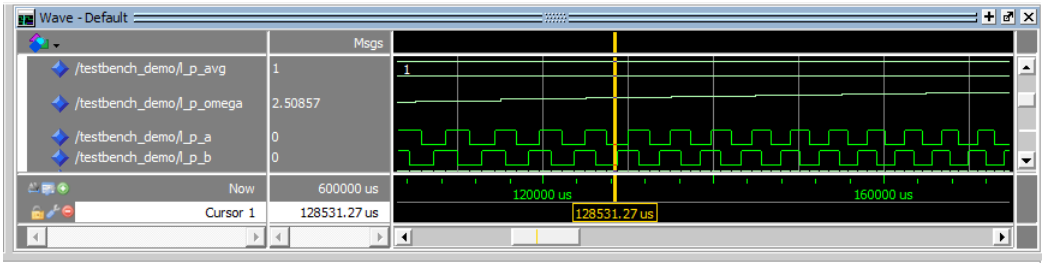


Figure 25: Pulses in channels a and b from a given input speed

This block does not need to be synthesizable as it is only for simulation purposes.

3.3.4 Position to radians

The incremental encoder reader developed outputs a value from 0 to a maximum equal to the parameter defined as pulses per revolution. As 1 revolution is equal to $2\pi radians$, a simple block *pos_to_rad.vhd* does the conversion as the error calculation of the feedback loop (set points minus position) requires both values to be in radians,

```
position_rad <= Real(position_count)*2.0*pi/Real(pulses_per_rev);
```

This block does not need to be synthesizable as it is only for simulation purposes and the previous conversion could easily be implemented in the C-Controller.

References

[1] Sanz, David. (2013). *Retrieved from http://therandomlab.blogspot.com/2013/03/logitech-c920-and-c910-fields-of-view.html*

[2] Thornton, Scott. (2017) Microcontroller Tips. *Retrieved from https://www.microcontrollertips.com/difference-between-fixed-and-floating-point/*

[3] DC Motor Speed: Simulink Modeling. Control Tutorials for MATLAB and SIMULINK. *Retrieved from http://ctms.engin.umich.edu/CTMS/index.php?example=MotorSpeedsection=SimulinkModeling*

[4] VNH2SP30-3 datasheet. *Retrieved from https://www.alldatasheet.es/datasheet-pdf/pdf/100179/STMICROELECTRONICS/VNH2SP30.html*

[5] C250 Technical Specifications. Logitech Support. *Retrieved from https://support.logi.com/hc/en-us/articles/360023305334-C250-Technical-Specifications*