Rubik's Cube Automatic Solver

Mechatronic System Design Project
University of Pisa

Alessandro Palla
Leonardo Bove

December 2024

Contents

Chapter 1

Introduction

The code and STL files of this project can be found on GitHub.

1.1 Project Objectives

The objective of the project is to read and solve a 3x3 Rubik's cube using an automatic robot, programmed via MATLAB and Simulink.
Robot management is handled by the NXP S32K144EVB-Q100 microcontroller thanks to Simulink's Model-Based Design Toolbox for S32K1xx.

1.2 Requirements

In order to properly control the robot, we faced the following challenges:

- Manage the communication between PC and microcontroller through PIL execution.

- Acquisition of colors from the cube's faces using a webcam.

- Control of four servo motors.

1.2.1 Hardware

In the project, the following hardware has been used:

- The NXP S32K144EVB-Q100.

- A 3D printed support (PLA), downloaded from an other project found on thingiverse.

- 4 Miuzei MZ996 180° servo motors controlled by a PWM with a duty-cycle, whose values lay in the range between 0.025 and 0.125. The datasheet is available here.

- The Braccio Shield v4 from Arduino robotic arm Tinkerkit, a 5V Power board where the 4 servo motors are connected. This object, connected to the power supply, is necessary because of the limited available power of the PC's USB port. The schematic can be found here.
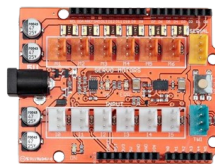
- A webcam for scanning the cube's faces.

1.2.2 Software

To use the NXP S32K144EVB-Q100 functions, Matlab and Simulink needed the following extension:

- NXP Support Package S32K1xx
- Model-Based Design Toolbox for S32K1xx
- Stateflow
- Simulink Embedded Coder
- Simulink Test
- Simulink SIL/PIL Manager

In addition, two more toolboxes have been added: the **MATLAB Support Package for USB Webcams** with the aim of managing the webcam, and the **Rubik's Cube Simulator and Solver** toolbox. The latter was of significant importance for the project, because it allowed us to generate a digital model of the cube (see section 2.4) and to implement the *Thistlethwaite 45* resolution algorithm on MATLAB.

1.3 Electrical Wiring

The *Arduino Braccio Shield* can be directly mounted on top of the NXP board, which has an Arduino compatible pin header. In order to supply the servo motors with the 5V external power supply connected to the shield, it is necessary to enable it, by setting to a high logic level the PTB3 NXP GPIO, which corresponds to the D12 digital pin on the Arduino compatible pin header of the shield. This is shown in figure 1.1.
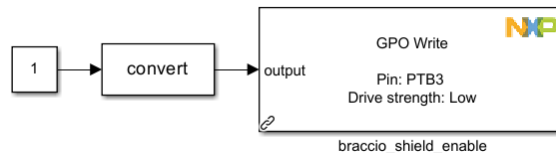

Figure 1.1: Shield enabling

Unfortunately, the servo motors pin headers on the shield have a different pinout (5V-SIGNAL-GND) compared to the one of the motors (SIGNAL-5V-GND), as you can see in figure 1.2. For this reason, instead of re-wiring the motors, male-female jumper wires were used to connect the right lines.

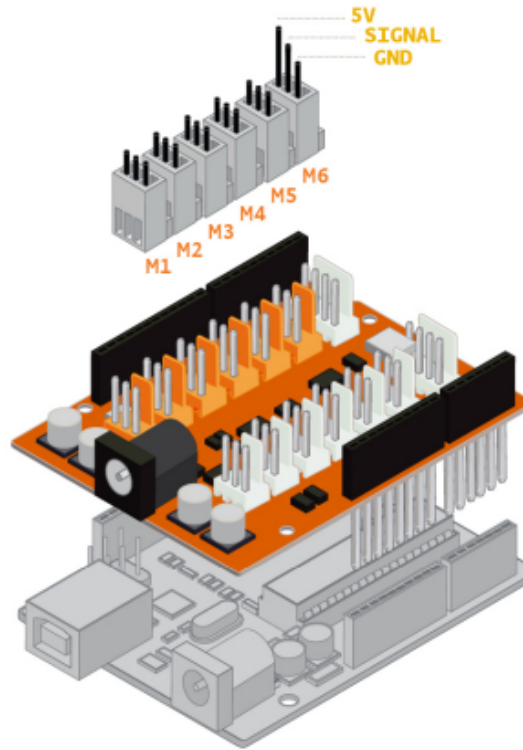In particular, the following connections were made:

Figure 1.2: Servo motors wiring to the driver shield.

- The left arm motor was connected to the M1 connector, whose signal line corresponds to the **PTB4** PWM output on the NXP board.

- The left grip motor was connected to the M2 connector, whose signal line corresponds to the **PTB5** PWM output on the NXP board.

- The right arm motor was connected to the M3 connector for the power supply, but, given that its signal line didn't lead to any PWM output on the NXP board, its PWM signal input was directly connected to the **PTD15** PWM output of the board.

- The right grip motor was connected to the M4 connector for the power supply, but, for the same reason as the previous one, its PWM signal input was directly connected to the **PTD16** PWM output of the board.

For the correspondence between the shield signal lines and the GPIO pins on the NXP board, the following schematic was used (1.3):
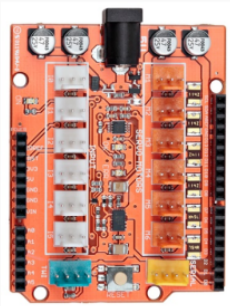
Nome connettore	Pin della shield		Nome connettore	Pin della shield
I0	A0		M1	11
I1	A1		M2	10
I2	A2		M3	9
I3	A3		M4	6
I4	A4		M5	5
I5	A5		M6	3
TWI	SDA, SCL		SERIAL	RX0, TX0

Figure 1.3: Arduino Braccio shield v4 pinout

Chapter 2

System Description

2.1 PIL System

The system was implemented using a **PIL** (*Processor-in-the-Loop*) execution. This means that the system is subdivided into different models, some of which are directly run by the MATLAB engine on the PC whereas others are used to generate C source code, which is compiled and downloaded onto the target hardware (in this case the S32Kxx microcontroller); afterwards, the user can simultanously run the code on the target and the other models on the PC through a Simulink Processor-in-the-Loop Simulation, which enables a UART communication between the two hardwares, allowing data exchange and verification.

   This execution mode is generally used to test an embeddable model on the target device, while the PC models the environment where the microcontroller will be deployed, without the need of testing it directly in the real environment. This helps in testing the embedded software faster and without compromising the real environment, which can sometimes result in dangerous consequences. In this case, the PIL simulation environment is actually used as part of the under-test system, for the reasons that will be explained in the next subsection.

2.1.1 Pros and Cons of PIL Simulation

The choice of the PIL simulation was pursued for the following advantages:

- **Delegation of complex tasks** to the PC: some functions, e.g. the webcam acquisition and the Rubik's cube solver algorithm, are easier to implement on the MATLAB environment because they were already available as application libraries or they would have required a computation effort that made it not implementable on an embedded system.

- **Easy debugging** and signal **logging**: Simulink PIL simulation provides a real-time data viewer and allows to compare logged signals from different runs of the simulation.

- **Simplified** set-up of the communication interface between PC and microcontroller via **UART**: Simulink automatically includes APIs in the generated embedded code that allow to exchange data via UART.

- Highly **customizable user interface**: in the simulation enviroment, there are many dashboard blocks that allow the user to easily interact with the model deployed on the target hardware.

On the other hand, this choice implies some disadvantages:

- Not perfect synchronization between PC and microcontroller: due to the overhead introduced by the MATLAB engine, sometimes there can be a pace error between the PC and the microcontroller simulation times.

7

- UART interface not optimized: even though it is easier to set-up for complex data exchange, it is not optimized. This problem can be overcome by defining a custom UART data exchange protocol.

- Impossibility to realize a stand-alone system: this problem can be solved by using a more powerful embedded system (e.g. a *Raspberry Pi*), so that the webcam management and solver algorithm can be implemented directly on the target hardware, making it independent from the MATLAB engine.

2.1.2 System Overview

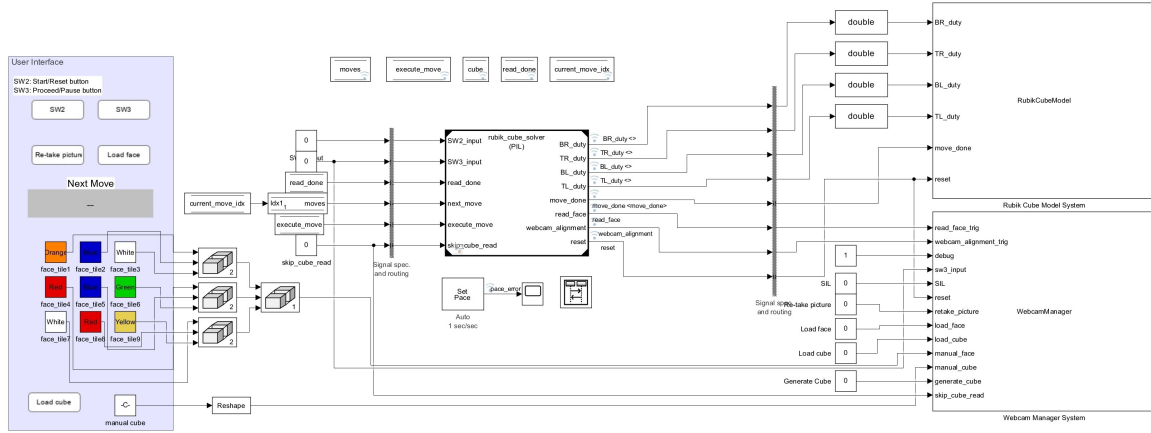In figure 2.1 is presented the overall system.

Figure 2.1: Overall system overview

This view is taken from the Simulink **Test Harness** manager, because, for the way the PIL simulation is originally thought, the actual DUT is the model that will be implemented on the microcontroller, while the other non-embedded blocks are built in the testing environment.

We can distinguish four main blocks in the system:

- **Cube Actuator**: this block models the actuator for the motors and the main finite-state machine of the system and this is the model that will be embedded into the microcontroller.

- **Webcam Manager**: this block is responsible for the webcam management and the cube's colors acquisition. It is run by the MATLAB interpreter.

- **Rubik's Cube Model**: this block manages the virtual model of the physical cube and the sequence of solving moves.

- **User Interface**: this area implements the user interface using blocks from the *Simulink Dashboard* resources.

As we can see, these blocks communicate through Simulink signals, which are delivered to the embedded model as well, thanks to the PIL simulation UART interface. Exchange of data happens also through *Data Store Memory* blocks (global variables) which are written by some models and read by others.

The two execution times are set to be equal (i.e. 1 second on the PC equals to 1 second on the microcontroller) and the models' tick time is set to 0.1 seconds.

2.1.3 MATLAB System Block

This block was used to implement both the *Webcam Manager* and the *Rubik's Cube Model*. It allows to implement algorithms using the MATLAB scripting language inside the Simulink environment. Moreover, it lets the user decide whether to generate source C code from it or to run it using the interpreter: the latter was the option that was helpful for our system.

Other relevant features of this block are that it allows to define state variables, to keep track of the current state of the subsystem, and the user can define an arbitrary number of input and output ports. Unfortunately, output ports were not used, because, apparently, they forced the system to be compiled into C code, especially if those output signals were connected to the *Cube Actuator* model. To solve this issue, we opted for system global variables (*Data Store Memory*), written by the two MATLAB Block Systems and read by the other embedded model.

The software implementation of this kind of block is made by means of a particular class, called **System object**. The default System object template is reported in 2.1.

Listing 2.1: System object template

```matlab
classdef untitled < matlab.System
    % untitled Add summary here
    %
    % This template includes the minimum set of functions required
    % to define a System object with discrete state.

    % Public, tunable properties
    properties

    end

    properties (DiscreteState)

    end

    % Pre-computed constants
    properties (Access = private)

    end

    methods (Access = protected)
        function setupImpl(obj)
            % Perform one-time calculations, such as computing constants
        end

        function y = stepImpl(obj,u)
            % Implement algorithm. Calculate y as a function of input u and
            % discrete states.
            y = u;
        end

        function resetImpl(obj)
            % Initialize / reset discrete-state properties
        end
    end
end
```

As we can see, the user can define `properties` that can be used to store state variables: unfortunately, their type can only be logical, numerical or an enumeration. These `properties` get initialized

inside the `resetImpl` function. The main body of the system is the `stepImpl` function, which is executed at every simulation tick. The user can define input and output signals for the system, just by changing the input and output arguments of this function. From here, any MATLAB script can be called, gaining access to all the Simulink toolboxes.

2.2 Cube Actuator

The block **Cube Actuator** is the heart of the system, which is embedded in the NXP board. As we can see from figure 2.2, the model was implemented on Simulink, using the *NXP's Model Based Design* toolbox. Here, we can distinguish four main blocks. The first is the **Stateflow**, which will be described in the next section. Its output is a **microcode** composed of four digits, one for each servo motor.

- first digit [0, 1]: open/close left grip;

- second digit [0, 1]: open/close right grip;

- third digit [0, 1, 2]: set left arm to 90/0/180 degrees

- fourth digit [0, 1, 2]: set right arm to 90/0/180 degress

  The following block is the **Cube Operation Decoder**, which takes as input the microcode and separates it into four different outputs. These are fed to the **Actuate Servos**, which converts the numbers in the corresponding duty-cycle values for the servos control. This way, a discrete positions actuation is implemented.

  The duty-cycle values are assigned as input of the **NXP PWM Config block**, which generates the C source code to control the PWM registers on the board.
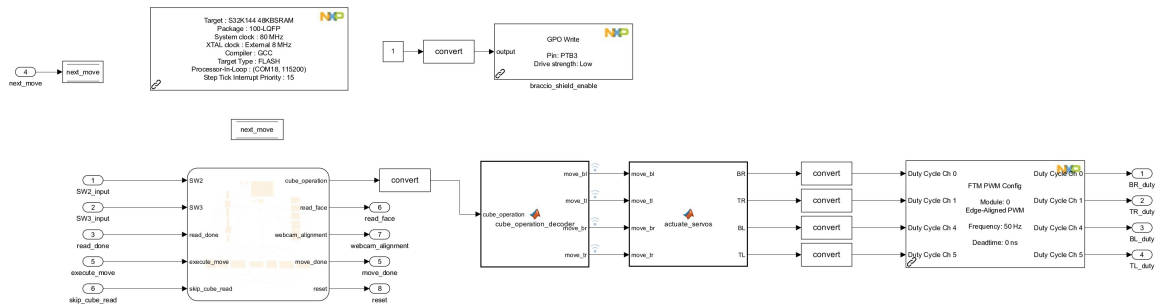

Figure 2.2: Cube Actuator blocks

2.2.1 Stateflow

Looking at the Stateflow, it is possibile to distinguish two macro groups: the **control section** and the **execution section**. The control section manages the operational phases of the robot, while the execution section sequentially executes the commands to perform a move on the cube.

Control Section

As you can see in figure 2.4, the control section is composed by 5 states:

- **wait_cube**: all local variables are initialized, the output `reset` is set to 1, initializing all variables in the *Webcam Manager* and in the *Rubik's Cube Model* (see sections 2.3 and 2.4). The `microcode`
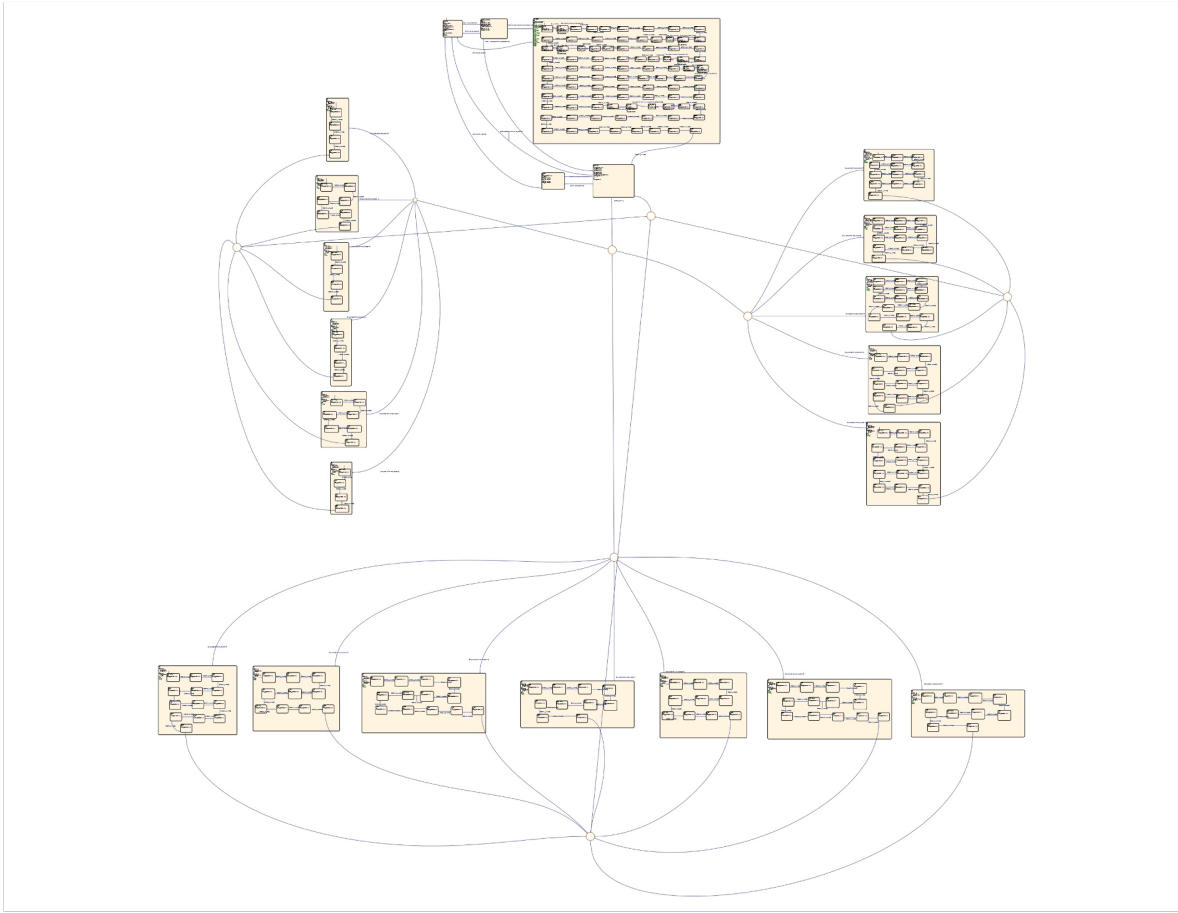
Figure 2.3: Stateflow

sent as output in this state is 1000, therefore one of the pliers is open and the other is closed. In this way, the user can easily place the cube on the arms. The cube has to be placed with the face having the red central tile in front of the webcam, and the face with the white central tile on the top.

- **webcam_alignment**: `reset` is set to 0 and will keep this values for all the other states. The `microcode` changes to 1100, closing both pliers. The output `webcam_alignment_trig` is set to 1, starting the thread `webcam_alignment`, see 2.3.1.

- **read_cube**: in this state the output `webcam_alignment_trig` is set to 0, the instantiated states are activated and they rotate the cube, in order to sequentially read the 6 faces, starting from the face with the red central cell, followed by blue, orange, green, white and yellow. At the end of the reading sequence, the cube is taken back to the original position. This is possible thanks to the instantiated states which sequencially changes every 2 sec, allowing the `microcode` to change accordingly. Moreover, once the next face is aligned with the webcam, the output `read_face_trig` is set to 1, triggering the acquisition of the colors of one face. Afterwards the user can decide to accept the acquired colors, retake a picture or manually load the face configuration. See 2.3.2 for more details.

- **solve_cube**: this is the interface between the execution and control states. The `microcode` is set again to 1100 while the output `read_done`, if the user does not press any button, is set to 1.

This lets the `Rubik's Cube Model block` to update the `current_move_idx`, in order to process the next move in the execution states. Before moving to the execution states, `read_done` is set again to 0. See section 2.4.

- **waiting**: the user can decide to pause the cube's solving by going inside this state and restart the execution at any time.
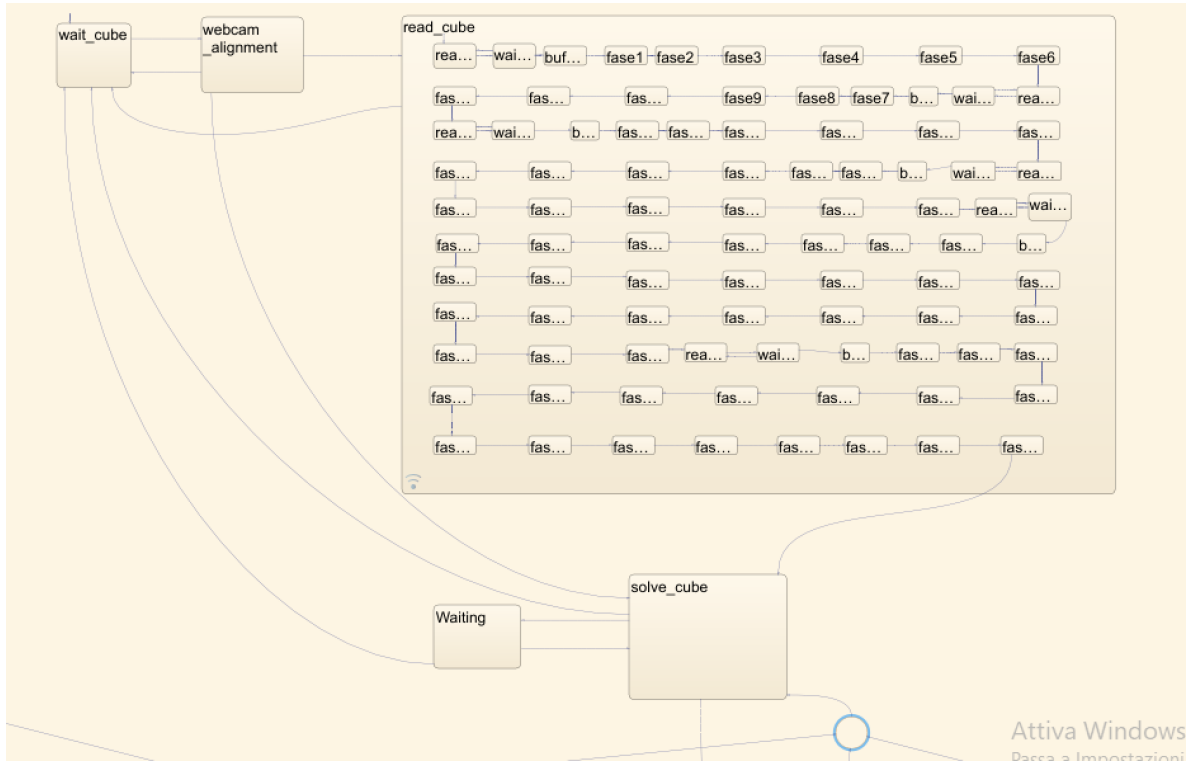

Figure 2.4: Control section of the Stateflow

State connections

In order to move from one state to the other, some specific conditions need to be verified. Before explaining how state connections work, it is important to highlight that the stateflow transitions that depend on SW2 or SW3 button inputs are **sensitive to only the rising edge** of the associated input signal.

Starting from the `wait_cube` state, the user can close the pliers and go to `webcam_alignment` by pressing the SW2 button. After this initial condition, SW2 will be the **reset** button for all the other states, allowing to come back to `wait_cube` and opening one of the pliers. The reset cannot happen during the execution of a single move.

Once the user has completed the webcam alignment, by pressing SW3 the cube's reading starts, thus going into the associated state. If, however, the input constant `skip_cube_read` is set to 1, it means that the user does not want to scan the cube, inserting it manually instead, see 2.3.2. In this case by pressing SW3, the next state will be directly `solve_cube`.

`read_cube` state goes to `solve_cube` once the last face has been read. In order to accept a cube's face scan and continue with the acquisition, the user has to press SW3. If the face was manually loaded, SW3 is still necessary to proceed with the acquisition.

The connection between the execution states and `solve_cube` state is possible thanks to the input `execute_move`, which is set to 1 by the *Rubik's Cube Model*, see 2.4.2.

When the *Rubik's Cube Model* sets to 1 the global variable `execute_move`, the system goes into one of the execution state, doing a move of the cube's solving. The move to be executed is specified by the `next_move` input, which consists of an array of two characters. Moreover, from `solve_cube` it is possible to switch to the `waiting` state if SW3 button is pressed or if the local variable `pause` is equal to 1.

From `waiting` if the user presses again SW3, the execution is resumed, going back to `solve_cube` state.

Execution Section

Each of the 18 states in this section corresponds to a move to be performed.

When the global variable `execute_move` is set to 1, the stateflow moves from `solve_cube` to the execution state specified by `next_move`. Each execution state is composed of multiple sub-states, which change the output `microcode` in order to rotate or modify the cube.

At the end of the last cube's operation of one execution state, the stateflow goes back to `solve_cube`. This process is repeated until the end of the solution moves.

Besides, every state contains a pause and reset condition. If the user presess SW3, the local variable `pause` is set to 1; similarly pressing SW2 the local variable `reset_status` is set to 1. In this way it is possible to go to `waiting` or `wait_cube` states directly from `solve_cube` checking these local variables, as soon as the current move is finished.

2.2.2   Cube Operation Decoder and Actuate Servos

To convert the `microcode` in 4 distinct inputs, a MATLAB function with the following body has been used:

Listing 2.2: Cube_operation_decoder

```
function [move_bl, move_tl, move_br, move_tr] =
    cube_operation_decoder(cube_operation)
x1 = int32(mod(cube_operation/1000, 10));
x2 = int32(mod(cube_operation/100, 10));
x3 = int32(mod(cube_operation/10, 10));
x4 = int32(mod(cube_operation, 10));

move_tl = x1;
move_tr = x2;
move_bl = x3;
move_br = x4;
```

After that, the outputs are converted into the corresponding duty-cycle values with another MAT-LAB function:

Listing 2.3: Actuate_Servos

```
function [BR, TR, BL, TL] = actuate_servos(move_bl, move_tl, move_br, move_tr)

min_duty = 0.025;
max_duty = 0.125;

duty_0_deg = min_duty;
duty_grip_open = min_duty + (max_duty-min_duty)/3;
duty_grip_closed = min_duty + (max_duty-min_duty)/14;
```

```matlab
 9  duty_90_deg = min_duty + (max_duty-min_duty)/2;
10  duty_180_deg = max_duty;
11
12
13  duty_offset_left_arm = (max_duty-min_duty)/30;
14  duty_offset_right_arm = 0;
15
16  switch (move_tl)
17      case 0                    % Open left grip
18          TL = duty_grip_open;
19      case 1                    % Close left grip
20          TL = duty_grip_closed;
21      otherwise                 % Open left grip
22          TL = duty_grip_open;
23  end
24
25  switch (move_tr)
26      case 0                    % Open right grip
27          TR = duty_grip_open;
28      case 1                    % Close right grip
29          TR = duty_grip_closed;
30      otherwise                 % Open right grip
31          TR = duty_grip_open;
32  end
33
34  switch (move_bl)
35      case 0                    % Left arm straight
36          BL = duty_90_deg + duty_offset_left_arm;
37      case 1                    % Left arm at 90   clockwise
38          BL = duty_0_deg + duty_offset_left_arm;
39      case 2                    % Left arm at 90   counter-clockwise
40          BL = duty_180_deg + duty_offset_left_arm;
41      otherwise
42          BL = duty_90_deg + duty_offset_left_arm;
43  end
44
45  switch (move_br)
46      case 0                    % Right arm straight
47          BR = duty_90_deg + duty_offset_right_arm;
48      case 1                    % Right arm at 90   clockwise
49          BR = duty_0_deg + duty_offset_right_arm;
50      case 2                    % Right arm at 90   counter-clockwise
51          BR = duty_180_deg + duty_offset_right_arm;
52      otherwise
53          BR = duty_90_deg + duty_offset_right_arm;
54  end
```

It is possible to notice that, in order to compensate for alignment errors between the two motors and therefore to be able to fine-tune the motors positions, two **offset** values were added.

2.3   Webcam Manager

This block is responsible for the webcam management, thanks to *MATLAB Support Package for USB Webcam*. The system block is presented in figure 2.5.

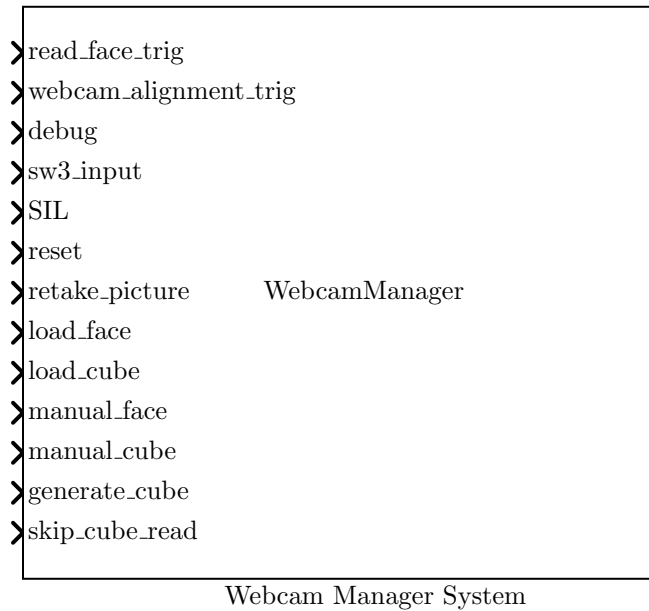The tasks performed by this system are the following:

Webcam Manager System

Figure 2.5: Webcam Manager System Block

- **Webcam alignment**

- **Face colors acquisition**

These tasks are triggered by the rising edge of signals coming from the *Cube Actuator*. The rising edge detection is implemented using state variables that store the previous value of the respective input. For example, the state variable `webcam_alignment_trig_status` is updated every tick with the current value of the corresponding input, `webcam_alignment_trig`: at the next simulation tick, if `webcam_alignment_trig == 1` and `webcam_alignment_trig_status == 0`, this means that a rising edge on that signal has occurred.

The `reset` input is likewise controlled by the *Cube Actuator* and triggers an immediate reset of all the state variables.

2.3.1 Webcam Alignment

When a rising edge on the signal `webcam_alignment_trig` is detected, the MATLAB script `webcam_alignment` is started: this script opens the connected USB webcam, displays a video preview of what the webcam sees and prints over the frames nine equally spaced red circles which form a square, so that the user can adjust the position of the webcam, in order to align the circles with the nine tiles of a single Rubik's cube face (see figure 2.6). This way the face colors acquisition will correctly read the colors in the middle of each tile.

Since the script continuously outputs the preview from the webcam, to make it easier for the user to perform the alignment, in order not to pause the whole simulation, it has to be executed on a different background thread. This is accomplished my means of the MATLAB function `parfeval`.

Listing 2.4: Background thread execution

```
if webcam_alignment_trig == 1 && obj.webcam_alignment_trig_status == 0
    obj.webcam_alignment_process = parfeval(@webcam_alignment, 0);  % Enable
        webcam alignment in a separate thread
    obj.alignment_in_progress = true;
```
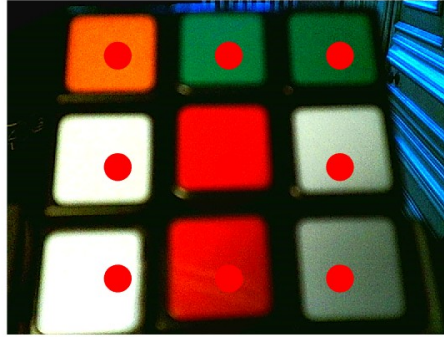
Figure 2.6: Webcam alignment view

As we can see in 2.4, `parfeval` is called with second argument set to 0, which tells the interpreter that no output is expected from the new thread. `parfeval` returns a *future object*, which is assigned to the state variable `webcam_alignment_process`. It can be later used to retrieve the thread outputs, if any, and to terminate the background thread using the function `cancel`, as soon as the alignment is completed and approved by the user by pressing the SW3 button.

2.3.2 Face Colors Acquisition

When a rising edge on the signal `read_face_trig` is detected, the `get_face_colors` script is triggered. This time there is no need of a background thread, since the script is non-blocking. It opens the webcam, takes a picture and computes nine RGB color codes by averaging the RGB values inside each of the nine circle areas used during the webcam alignment phase. These nine RGB color codes are compared with hardcoded RGB ranges, each of which corresponds to one of the six possible colors on the cube. Each color correspond to a different integer: red is 1, blue is 2, orange is 3, green is 4, white is 5 and yellow is 6. `get_face_colors` returns a 3x3 matrix containing the color integer codes detected for each tile. If no color is recognized, the matrix element corresponding to that tile is set to 0. If the `debug` input port of the system is set to 1, this script will also return the RGB average colors that have been acquired for debug and calibration purposes.

Once the acquisition is completed, the webcam must be closed, since it is not possible to store its handler in any state variable of a MATLAB System Block, and the face colors matrix is analyzed: if any of the elements is 0 (undefined color), the user is notified through the simulation log and it is possible to re-take the picture by pressing the *Re-take picture* button; otherwise, the user can approve the read colors (errors during acquisition can occur) by clicking the SW3 button and the face is loaded onto the global 3x3x6 `cube` matrix, that holds the color configuration of the whole cube during the simulation. When a face is approved, the global variable `read_done` is set to 1 (and reset to 0 later on): this flag is read by the *Cube Actuator*, which now can proceed and rotate the cube, in order to present to the webcam the following face to be read.

Due to bad light environments, it can happen that some colors of a face are hard to read. In this case, the user can manually load the color configuration of the current face through the user interface (see figure 2.7) and add it to the cube configuration using the *Load face* button.

Not only can the user manually load the color configuration of a single face, but it is also possible to manually load the whole cube configuration as a 3x3x6 matrix of integers from 1 to 6, assigned to the `manual cube` constant value block. The faces must be inserted with increasing number of the middle tile color code (i.e. the first face is the red one and the last is the yellow one), as showed in equation 2.1.
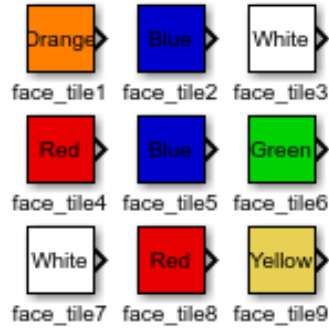
Figure 2.7: UI to manually load face colors

$$[[1, 2, 3; 2, 1, 4; 5, 6, 3], ..., [2, 5, 6; 3, 6, 3; 4, 1, 2]] \tag{2.1}$$

2.4 Rubik's Cube Model

This block is responsible for the real-time animations of the virtual cube and for the computation of the sequence of solving moves given the cube configuration, thanks to the *MATLAB Rubik's Cube Simulator and Solver* toolbox. The system block is presented in figure 2.8.
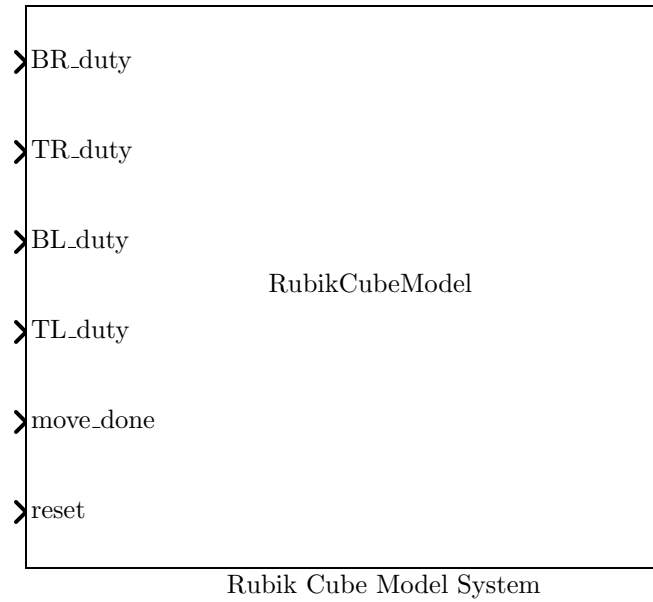

Rubik Cube Model System

Figure 2.8: Rubik's Cube Model block

The tasks performed by this system are the following:

- **Digital twin** of the physical cube

- **Rubik's cube solver**

Similarly to the *Webcam Manager*, some of the actions of this block are triggered by the edge of signal flags set by the *Cube Actuator*. Also here the `reset` input resets all the state variables of the system.

2.4.1 Digital Twin

The system takes as inputs the duty-cycles set by the *Cube Actuator* to the PWM generators, which determine the absolute positions of the four servo motors (BR is bottom-right, BL is bottom-left, TR is top-right and TL is top-left). At each simulation tick, the current values of these duty-cycles is stored in state variable, so that the system can detect changes in the positions of the motors. In this system are also defined some constant variables that contain the duty-cycles values corresponding to fixed positions in which the motors can be found in (e.g. 180°, 90° or 0° for the bottom motors or open/closed for the grip motors), see 2.5. These values are set after the analogous parameters that can be found in the *Motor Positions Decoder* (see subsection 2.2.2), and they must not differ, in order to allow the *Rubik's Cube Model* to correctly interpret the positions.

Listing 2.5: Constant duty-cycles parameters

```
obj.min_duty = 0.025;
obj.max_duty = 0.125;
obj.duty_0_deg = obj.min_duty;
obj.duty_90_deg = obj.min_duty + (obj.max_duty-obj.min_duty)/2;
obj.duty_180_deg = obj.max_duty;
obj.duty_grip_open = obj.min_duty + (obj.max_duty-obj.min_duty)/3;
obj.duty_grip_closed = obj.min_duty + (obj.max_duty-obj.min_duty)/14;
obj.offset_duty_left_arm = (obj.max_duty-obj.min_duty)/30;
obj.offset_duty_right_arm = 0;

% Truncate values to the 6th decimal number
obj.duty_90_deg = floor(obj.duty_90_deg * 10^6) / 10^6;
obj.duty_180_deg = floor(obj.duty_180_deg * 10^6) / 10^6;
obj.duty_grip_closed = floor(obj.duty_grip_closed * 10^6) / 10^6;
obj.duty_grip_open = floor(obj.duty_grip_open * 10^6) / 10^6;
obj.offset_duty_left_arm = floor(obj.offset_duty_left_arm * 10^6) / 10^6;
obj.offset_duty_right_arm = floor(obj.offset_duty_right_arm * 10^6) / 10^6;
```

The reason why these values and the input duty-cycle values are truncated to the sixth decimal number is because, due to resolution errors in computations with floating point numbers typical of scripting languages, such as Python and MATLAB, the input values differed from these constants with an error of magnitude of 10^{-10}. This fact implied that the motors were never found in any of these fixed absolute positions. Of course, an error of that order, which is only a numerical fluctuation, can be easily ignored.

As soon as the cube colors are fully acquired, a visual representation of the cube is drawn (see figure 2.9), thanks to the toolbox function `rubplot`, and, given the sequence of absolute positions of the motors, the system can detect movements of the physical cube and animate them on the cube drawing, using the provided toolbox functions.

This feature is very helpful for debugging purposes, because it can be used to test the functionality of the *Cube Actuator* without the need of the actual mechanical structure and of deploying the model on the microcontroller (this will be better explained in section 3.1). Moreover, it is also helpful along with the physical cube being solved, since it can be used to check for inequalities between the real and virtual implementations: given that the digital twin model has been thoroughly tested together with the actuator model, in case of discrepancies it can be easily affirmed that the problem is in the physical motors, resulting in a considerable help during the debug process.
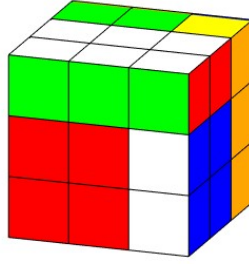
Figure 2.9: Digital twin of the physical cube

2.4.2 Solution Moves Management

The algorithm used to solve the cube is the ***Thistlethwaite 45***, which solves the cube in less than 45 moves, averaging at 31. This algorithm is already available in the MATLAB Rubik's Cube toolbox and it is implemented in the function `Solve45` which takes as input a 3x3x6 matrix, containing the cube colors configuration with integers from 1 to 6, as described before (see 2.3.2). If the given configuration is impossible, i.e. it cannot be realized with a standard Rubik's cube, the model will notify the user and it will allow to load a new configuration; otherwise, it will return an array of strings of variable length (between two and three), containing the sequence of moves that solve the cube, expressed in the standard Rubik's notation: for example "L" stands for a 90° clockwise rotation of the left face, "U'" stands for a 90° counter-clockwise rotation of the upper face or "B'2" stands for a 180° counter-clockwise rotation of the back face. The standard convention for the naming of the faces is presented in figure 2.10.
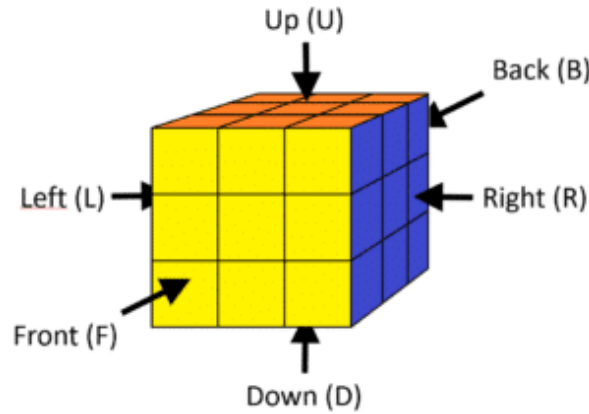

Figure 2.10: Standard face names for a 3x3 Rubik's cube.

The sequence of moves is stored in the global *Data Store Memory* called `moves`: given that Simulink cannot implement global variables as arrays of strings and to make it easier the exchange of information with the *Cube Actuator* model (remember that it will be converted into C code), `moves` is an array of 90 character vectors of constant length two. The custom script `update_algorithm` has the purpose of converting the output of `Solve45` from an array of strings (actually an array of MATLAB cell strings)

into an array of character vectors, as showed in 2.6. This is done by means of the MATLAB method `arrayfun`, which applies a function to all the elements of an array.

Listing 2.6: Array of cells strings into array of character vectors

```
1  moves_str = Solve45(varargin{2}); % Solve cube. Returns an array of cell
        strings
2
3  % Update the object moves sequence, converting moves_str to
4  % an array of chars vectors
5  moves = arrayfun(@(s) uint8(char(s)), moves_str, 'UniformOutput', false);
6
7  new_moves = uint8((zeros(90, 2)));
8
9  % Convert the moves from a single row cell array of chars vectors back to a
10 % matrix of uint8, where each row is a move
11 for i = 1:length(moves)
12     new_moves(i, :) = uint8([cell2mat(moves(:, i)), zeros(1, 2 -
            length(cell2mat(moves(:, i))))]); % Consider a zero-padding at the
            end of each vector (only for moves of length 1)
13 end
14
15 % Update moves
16 moves = new_moves;
```

Since some moves are of length three, the solution array is first parsed to look for moves like "F'2", which are then split into two subsequent "F'" moves. That is why the final array `moves` must have length higher than the theoretical maximum expected with the *Thistlethwaite 45* algorithm: in the worst case there are 45 double counter-clockwise moves, leading to a final number of moves of 90.

The point of view of the given solution moves is always considering the red face as the front face and the white face as the up face: this is why the cube must be inserted in the robotic arms with the red side facing the webcam and, after the read sequence, the cube will be automatically reset in the original position, with the red face in front of the webcam. Of course, during the execution of the sequence of moves, the whole cube can be rotated and the point of view from the webcam, which is always our reference, changes. Thankfully, the Rubik's cube toolbox provides also a function that updates the algorithm, given the applied rotation: `algrot`. This function is called inside `update_algorithm` only when the cube undergoes a rotation around one of its three axes. This function updates the point of view of all the moves in `moves`.

Here in 2.7 is showed a snippet of code showing the detection of a cube rotation, thanks to the temporal sequence of duty-cycles as explained in 2.4.1, followed by the update of the cube animation and the update of the moves' sequence.

Listing 2.7: Detection of a cube rotation, its animation and update of the moves' sequence

```
1  if TR_duty_truncated == obj.duty_grip_closed && TL_duty_truncated ==
        obj.duty_grip_open && BR_duty_truncated > obj.BR_duty_old
2      % Counter-clockwise rotation of the right arm with the left
3      % grip open (x1)
4      cube = rubrot2(cube, 'x1', 'animate', 1);
5      drawnow;
6
7      % Rotate algorithm
8      moves = update_algorithm(moves, false, true, 'rotation', 'x3');
9
10     % If the previous motor angle was at 0 degrees and now
11     % is set to 180 degrees, a 180 degrees turn has
```

```
12        % happened: rotate again
13        if obj.BR_duty_old == obj.duty_0_deg && BR_duty_truncated ==
             obj.duty_180_deg
14            cube = rubrot2(cube, 'x1', 'animate', 1);
15            drawnow;
16            moves = update_algorithm(moves, false, true, 'rotation', 'x3');
17        end
```

The moves are fed one by one to the *Cube Actuator*, by means of a *Data Store Read* of `moves`, which allows to read the move at the position specified by the pointer `current_move_idx`, as shown in figure 2.11.

Figure 2.11: Current move pointer usage

Every time the *Cube Actuator* completes one move, it generates a positive edge on the signal `move_done`, which is detected by the *Rubik's Cube Model*: then it increments the pointer to the array of moves and triggers the execution of the currently selected move by rising the signal `execute_move`, read by the *Cube Actuator*. This sequence is presented in the snippet 2.8:

Listing 2.8: Update of the current move and move execution trigger

```
1  % After applying rotations (rising edge of move_done), update current move
      index if the
2  % move is completed and start move execution
3  if move_done == 1 && obj.move_done_old == 0
4      current_move_idx = uint16(current_move_idx + 1);
5  elseif move_done == 0
6      execute_move = 0;
7  elseif move_done == 1
8      execute_move = 1;
9  end
```

Once the end of the `moves` array is reached, the model does not execute any move and waits for the user input to reset the system by clicking SW2.

Chapter 3

Execution

The execution is performed inside the Simulink **Test Harness** environment. To open it, starting from the Simulink window of the project, you need to follow these steps:

1. Click on APPS in the Menu bar, at the top of the window, and select *Simulink Test*.

2. In the associated toolbar click on *Manage Test Harnesses* and then select the only available Test Harness, i.e. *rubik_cube_solver_testbench*.

3. Once the Test Harness window is opened, select in the menu bar the APP *SIL/PIL Manager*.

4. Select in the toolbar from the dropdown menu on the left whether to perform a SIL mode (*Simulation Only*) or a PIL mode (*SIL/PIL Simulation Only*).

5. From the toolbar, insert a desired stop time and click on *Run Simulation*.

3.1   SIL Execution

Thanks to the digital twin, see 2.4.1, it was possible not only to use the PIL mode in order to solve the cube as already described in 2.1, but also to run a **SIL** (*Software-in-the-Loop*) execution.

Using SIL, webcam acquisition is disabled and the user can choose whether to manually load a cube configuration or generate one with a random configuration. The cube is then solved and displayed through an animation.

The *Cube Actuator* code is executed in simulation on Simulink. In this way it is possible to simulate a virtual cube's solving, with the same resolution times and the same possible movements of the arms and pliers. Moreover, as for the PIL execution, user can use the `pause` function, by pressing SW3, or the `reset` function, by pressing SW2. In the latter case, the cube's animated figure is closed and it is possible to generate a new one by pressing SW2 again.

3.1.1   Test Harness Configuration for SIL

The Test Harness screen must be properly configured prior to execution.

To start with, looking at figure 2.1, constant inputs on the left of *Webcam Manager* block have to be configured as follows:

- `SIL` has to be set at 1.

- if the user wants to generate a random cube, also `Generate Cube` has to be set at 1.

Going to the User Interface, if the user does not generate the cube, he can manually load it, by typing the configuration as explained in 2.1, and then press the *Load cube* button.

Moreover, without the need to scan the cube, the user can set `skip_cube_read` input constant to 1. This way, the `read_cube` state is skipped. It can be useful to not skip this state, for example when the user wants to emulate the moves done on the cube during the acquisition phase on the virtual cube animation (in this case, it is necessary to generate a random cube). As a matter of fact, this mode was used to debug this state.

Once the configuration is completed, the user can start the execution and leave the reset state by pressing SW2. As soon as the cube configuration is fully loaded, the digital twin animation will appear on the screen, and, by pressing SW3, the resolution of the cube will start.

3.2 PIL Execution

The *Processor-in-the-Loop* execution is the mode used when a resolution of the physical cube is desired, by inserting it into the robot.

As widely seen in chapter 2.1, the user can decide whether to acquire the cube colors via the webcam or upload them manually. Once the configuration is available, the cube solving begins. The set-points of the angular positions of the motors are read by the *Rubik's Cube Model*, which generates a digital twin of the physical cube and replicates its movements.

Once the PIL simulation is launched, the *Cube Actuator* code is compiled and downloaded to the microcontroller, and a UART communication interface is started between the PC and the microcontroller.

3.2.1 Test Harness Configuration for PIL

As already done for the SIL, the instructions that must be added to the test harness for PIL are reported below.

Constant inputs on the left of *Webcam Manager* block have to be configured as follows:

- `SIL` has to be set at 0.

- `Generate Cube` does not care in this mode.

- `debug` can be set to 1, in which case the debug RGB values acquired are reported in the output log.

If the user wants to acquire the cube's faces via webcam, `skip_cube_read` on the left of the *Cube Actuator*, has to be set to 0. Now, looking at the User Interface, by pressing SW2 the webcam alignment is activated, and then, using the SW3 button, it is possible to start the cube's acquisition. The user can check the acquired colors on the output log, or, as already described in chapter 2.3.2, they can manually change the color configuration of the face and upload it by pressing `Load face`. When the user is satisfied with the acquired colors, by pressing SW3 again it is possibile to proceed to the acquisition of the next face. Once all the faces are uploaded and the cube is set back to the original position, the resolution of the cube starts automatically.

It is always possible (except during a move) in both kinds of executions to reset the system using the SW2 button and reconfigure the Test Harness. However, if the user wants to switch from a SIL to a PIL execution or vice versa, he must interrupt the current mode and start a new one, making sure that the NXP board and the webcam are properly connected to the PC.

Chapter 4

Conclusions

4.1 Problems

The problems that we mainly had to cope with are related to the servo motors and their actuation:

- **Grip on the cube surface**: due to the smooth surface of the standard Rubik's cube, the grip of the PLA printed robotic hands was not optimal. For this reason we had to find some buffer material to put between the cube and the hand. The best choice resulted in a soft foamy material overlayered with a rubber on top of it, which had the perfect friction coefficient.

  Unfortunately, there is a subtle threshold of duty-cycle value: if the duty-cycle of the grip motors is below this threshold, the grip is not strong enough to hold the cube; if it is higher, the motor stalls and it becomes irresponsive of any further set point values. When this happens the motor generates a considerable amount of heat and it independently tries to go to a 0° position, which corresponds to a fully closed grip hand.
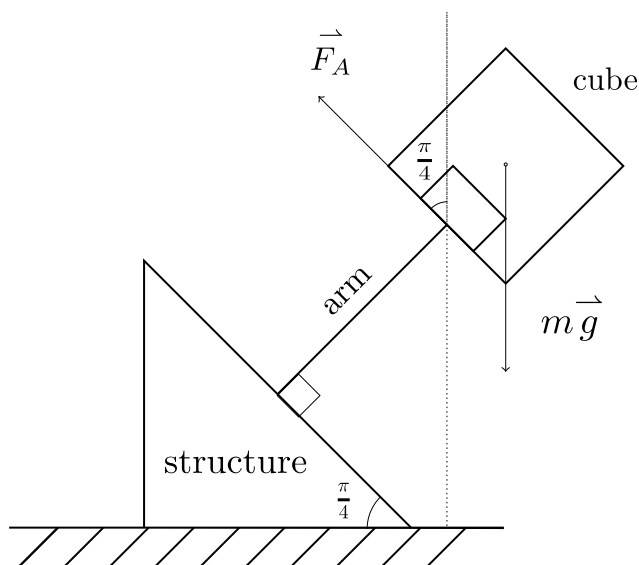

Figure 4.1: Rubik's Cube Solver structure physics

  Now, we could think that the problem lies in the fact that the maximum stall torque is not enough to hold the cube, but we can see that this is not the case. In figure 4.1 we can see the

model for the mechanical structure, in the worst case when only one arm is holding the cube, where $m\vec{g}$ is the weight force, with m the mass of the cube (equal to $140g$) and $\vec{F}_A$ is the static friction force between the hand and the cube. As we can see from figure 4.2, we can find the magnitude of the static friction force as function of the force impressed by one servo motor through the grip hand.

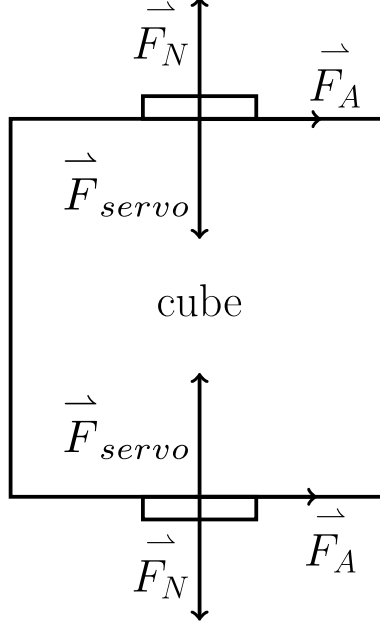$$|\vec{F}_A| = \mu_s|\vec{F}_{servo}| \tag{4.1}$$

Figure 4.2: Top view of the Rubik's cube with the robotic hand.

By imposing the static equilibrium along the vertical direction, we obtain:

$$-mg + 2|\vec{F}_A|\sin\left(\frac{\pi}{4}\right) = 0 \tag{4.2}$$

$$-mg + 2\mu_s|\vec{F}_{servo}|\frac{\sqrt{2}}{2} = 0 \tag{4.3}$$

$$|\vec{F}_{servo,tot}| = 2|\vec{F}_{servo}| = mg\frac{2}{\sqrt{2}}\frac{1}{\mu_s} = 3.238N \tag{4.4}$$

Where we took $\mu_s = 0.6$, which is the worst friction coefficient that can be generally found between rubber-like materials and plastic. Now, considering that the lenght of the grip hand is about $5cm$, we can find that the maximum stall torque required by one motor to hold the cube is $T_s = |\vec{F}_{servo,tot}| \cdot 0.05m = 0.1619Nm = 1.65kgf \cdot cm$, which results far less than the maximum available stall torque declared in the datasheet ($T_{max} = 9.4kgf \cdot cm$) at a power supply voltage of 4.8V (in our case it is 5V). This could mean that the maximum torque may be overcome during transients or the motor is defective.

One possible solution could be increasing the operating voltage of the motors up to 7.2V, which is the maximum allowed for this model: this way, also the maximum stall torque will increase and, reasonably, also the dynamic torque (it is not stated in the datasheet).

Another possible solution could be reducing the strength of grip of the motors and replace the hands with elastic pliers, as shown in figure 4.3.
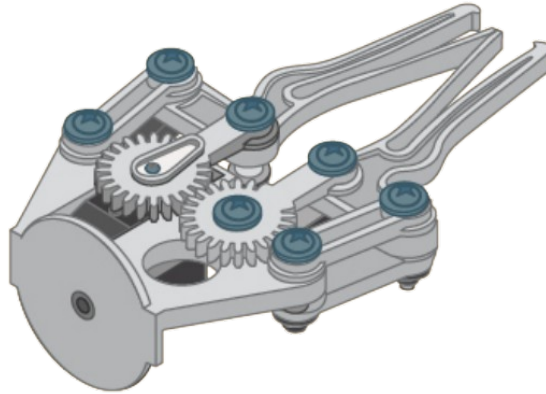

Figure 4.3: Elastic pliers.

This way, all the cube load will not be only on the servo motors, but the elastic force of the pliers would help to increase the force on the cube without overloading the motors.

This solutions are not implemented yet.

- **Motors instability** During PIL simulation, there were sudden and random small oscillations of the motor shafts around their set point angular position (defined by the PWM duty-cycle).

This effect can be caused by noise on the PWM wave generated by the microcontroller: however, this is not the case, because when the microcontroller flash memory is loaded with a test program (analog control of motor angular position through the built-in potentiometer) and executed in stand-alone mode, this effect is not present. Therefore it must be due to the co-execution between the microcontroller and the PC. The solution to this problem is then converting the whole project to an embedded system, without the need of a Simulink PIL simulation.

4.2 Future Improvements

In addition to the solutions proposed in the previous section 4.1, other improvements can be made to the current project:

- **Re-design of the current 3D structure**, in order to fix small misalignments, due to the servo motors used, which are different compared to the ones used in the original project.

- **Webcam with a larger FOV**: the current webcam has a quite small FOV and for this reason it could not be mounted on the original holder, since it was too close to the cube face (it was designed for a Raspberry Pi Camera Module).

- **Custom UART protocol interface**: many limitations in the exchange of data between the PC environment and the microcontroller were due to the UART interface (*openSDA*) not being optimized for our objectives, because it was automatically taken care by the *NXP's Model Based*

Design toolbox. One way to reduce the instability of the motors due to the PIL simulation and to gain more control on what data is actually exchanged, limiting it to the bare minimum, could be to implement a custom UART interface. This means that the *Webcam Manager* and *Rubik's Cube Model* blocks would be deployed from the Simulink testing enviroment to a custom application (e.g. written in Python), which also takes care of the graphical user interface and of the UART communication, for example using the library pySerial.