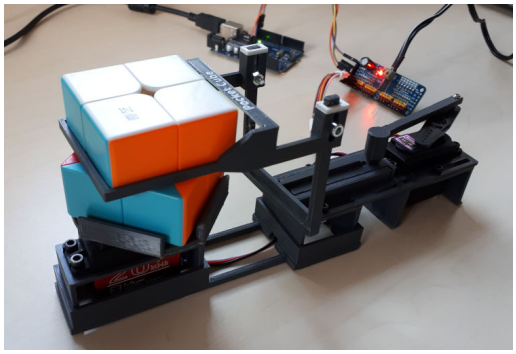
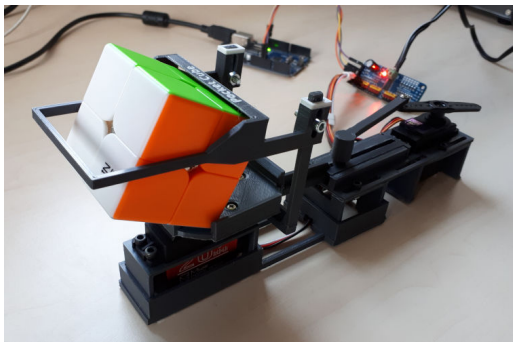
Marc Hensel

Pocket Cube Solver

A Motorized Device for Mini Cubes





HAW
HAMBURG

The author has made effort to ensure the provided information is correct. However, the information is provided without any warranty. Neither the author, nor any other person or organization will be held liable for any damages caused or alleged to have been caused directly or indirectly by this document.

Pocket Cube Solver: A Motorized Device for Mini Cubes
Document version: 31.07.2023

© Prof. Dr. Marc Hensel
http://www.haw-hamburg.de/marc-hensel

Published under Creative Commons license CC BY-NC-ND 3.0
Attribution, non-commercial, no derivatives
https://creativecommons.org/licenses/by-nc-nd/3.0/deed.en

Preface

Dealing with artificial intelligence (AI), I was curious how to let computers learn to solve a scrambled Rubik's cube, and I was lucky that my student Finn Lanz accepted this challenge for his Master thesis[1]. While Finn did a marvelous job—led by his remarkable ambitions, competence, and passion—it showed that the available computing resources would not be sufficient to thoroughly develop appropriate deep learning models, because of the high complexity of Rubik's cube. Thus, he proposed to proceed using the cube's little brother, the *Pocket cube*, also known as *Mini cube*, with size $2 \times 2 \times 2$.

Even this reduced complexity was clearly challenging enough. Still, wouldn't it be even more fun to demonstrate the results by machines that *mechanically* solve physical cubes? Image how cool it would be to scramble a cube manually, put it into the machine to unscramble it for you. Being an engineer, I love this kind of playing around. We already had a high-performance machine available to unscramble Rubik's cube, therefore, I decided to develop a device for the Pocket cube as well. However, I explicitly wanted to design a low-cost device, which is easy to rebuild and use for anyone interested. The design should be based mainly on 3D-printed parts and use two standard servos as motors, only. The device should be controlled by a Laptop, which connects via an Arduino board providing output pins for the servos. Programming applications should be simple by an easy to use Python software interface.

Document and further resources The following chapters document the device, consisting of the hardware, software for the Arduino board controlling the servo motors, and the Python software interface. A project repository and the 3D print files are available on GitHub and Thingiverse, respectively:

`https://github.com/MarcOnTheMoon/cubes`
`https://www.thingiverse.com/thing:5822433`

Stay tuned Naturally, there are things to improve making this project work in progress. Therefore you might want to check for updates. However, Finn Lanz has successfully integrated the developed *Pocket Cube Solver* to a system and we are both very happy that the system works like a charm.

Marc Hensel

[1] Finn Lanz: *Deep Reinforcement Learning zum maschinellen Erlernen von Strategien zur Lösung von Zauberwürfeln.* HAW Hamburg, Master Thesis, 2023

Contents

Chapter 1

Assembly and Setup

This chapter guides you through the hardware setup and calibration of the Pocket cube solver.

1.1 Hardware

Let's begin with the hardware assembly, consisting of the required material, assembly, and wiring of the electronic components.

1.1.1 Required material

The material includes 3D printed parts as well as off-the-shelf components.

3D prints All 3D-printed parts are published as STL files on Thingiverse. This includes the main project[1] and a linear servo actuator[2]. Moreover, assembly is somewhat easier using a mounting tool[3] that I have published on Thingiverse as well.

Off-the-shelf Table 1.1 lists the principal off-the-shelf parts required to build the device and states the concrete models I have actually used for our prototypes. Additionally, jumper cable wires as well as screws and nuts (M3) are required.

Table 1.1: Part list

Type	Part used	Functionality
Controller (μC)	Arduino Uno R3	Control servo motors
PWM driver	PCA9685	Generate signals for servos
Servo (180°)	MG996R	Tilt cube (turn)
Servo (270°)	Miuzei MS24-C-UF	Rotate cube's bottom layer
Power supply (6V)	Standard part (3A)	Supply servos via PWM board
Camera	Jolly Comb W06 (1080P Webcam)	Scan colors
Pocket cube	Standard cube	Have fun!

1.1.2 3D-printed parts and servos

Before you assemble the device, you must print the 3D parts—guess this does not come as a surprise. I do not expect it to make much of a difference, however, in case you are curious: I have prepared the 3D models for printing with PrusaSlicer[4] Version 2.5 using 0.07 mm layers sliced as variable layer height. The parts were printed with PLA by a Prusa i3 MK3S+.

Once, all material is available, Figure 1.1 guides you through the assembly step by step:

1. Put together the linear servo actuator (Fig. 1.1a–b). Make sure to put the small fin into the sparing of the sliding part. The parts will be held in place by the servos' screws.

2. Equip the linear actuator with the 180° servo (Fig. 1.1c–d) and fix it by screws. Use the mounting tool to hold the nuts in place.

3. Attach the actuator's legs using screws and nuts (Fig. 1.1d–e).

[1] `https://www.thingiverse.com/thing:5822433`
[2] `https://www.thingiverse.com/thing:5722550`
[3] `https://www.thingiverse.com/thing:5822224`
[4] `https://www.prusa3d.com/` (visited: 07.02.2023)

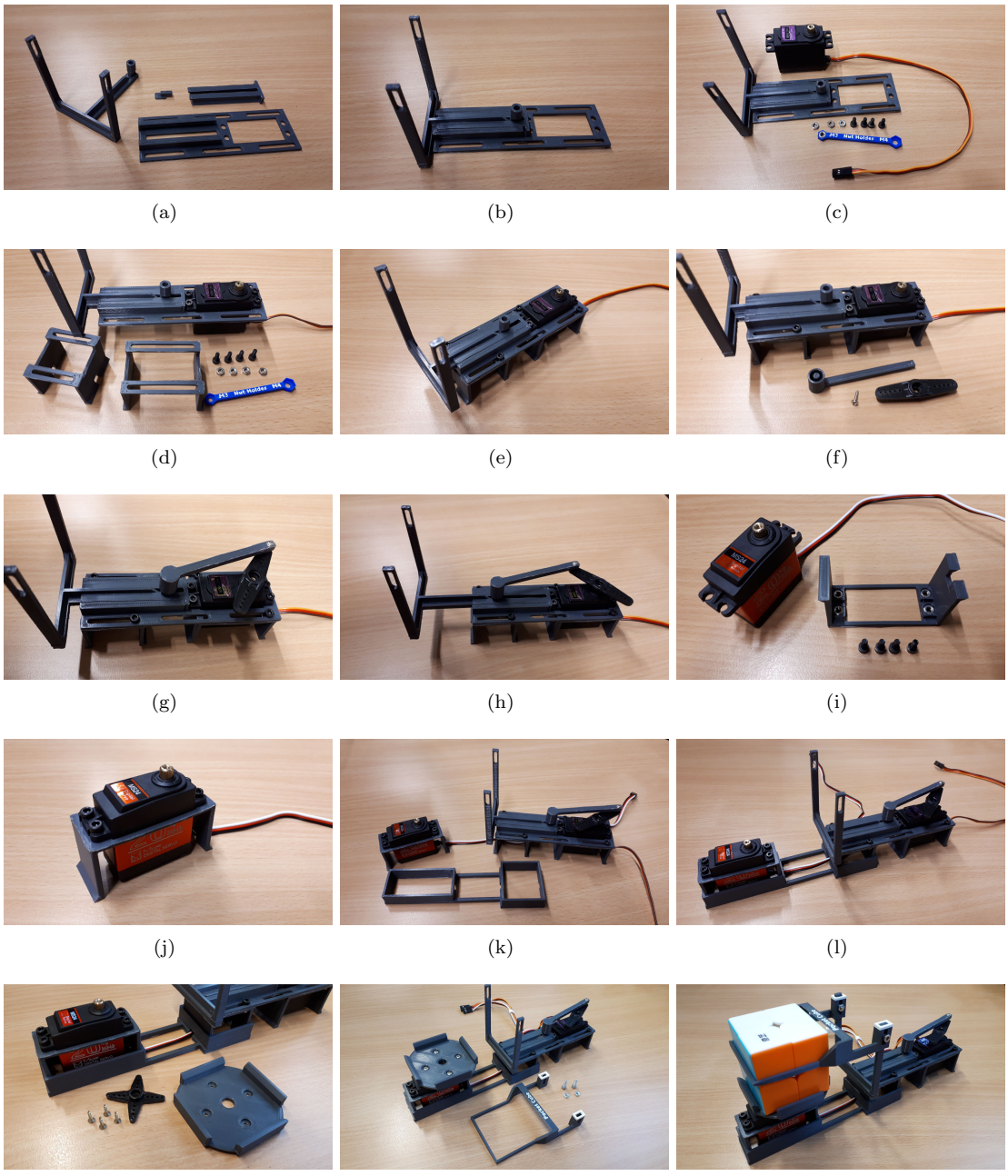


Figure 1.1: Assembly of 3D-printed parts and servos

4. Mount the servo arm and connect it to the slider using the appropriate part (Fig. 1.1f–h). The servo should be rotated to the full right when holding the slider as far to the servo as possible.

5. Attach the legs to the 270° servo (Fig. 1.1i–j).

6. Fix both servo components by clipping them into the common foot stand (Fig. 1.1k–l).

7. Mount the cube holder using a cross-shaped servo arm (Fig. 1.1m–n).

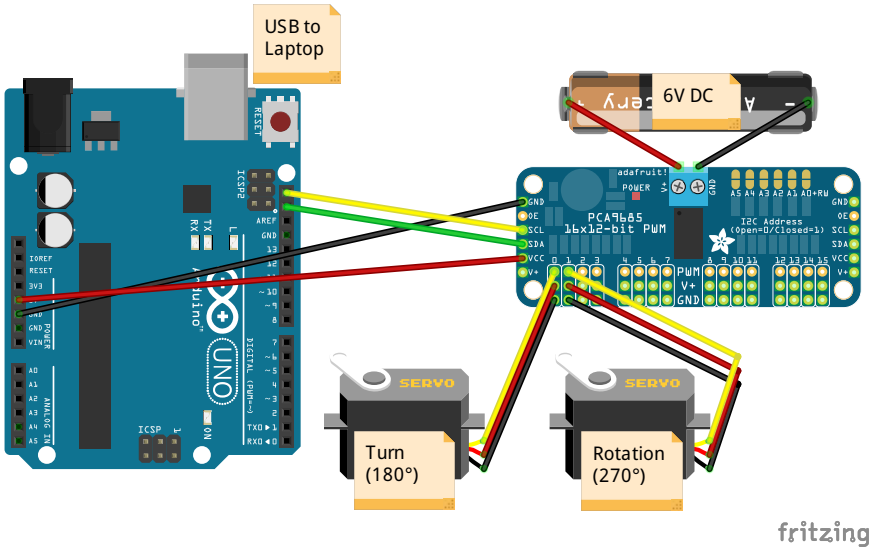


Figure 1.2: Wiring schematics

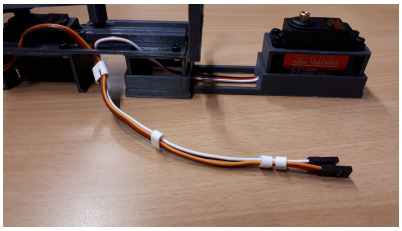


Figure 1.3: Servo cables

8. Finally, mount the cube “pusher” using screws and nuts (Fig. 1.1n–o).

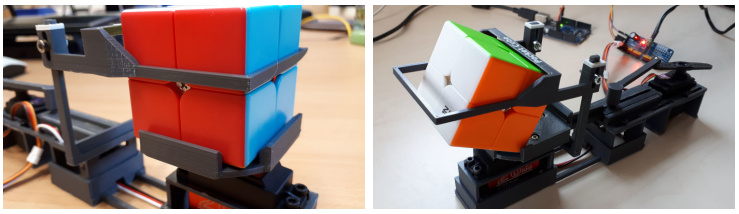
1.1.3 Electronics

Figure 1.2 shows the schematic wiring. As depicted, the Arduino board connects to the PWM driver board by the SDA and SLC ports of the I^2C bus and provides the driver’s electronics with power. I use an Arduino Uno R3. The concrete model is of minor importance, though. Naturally, comparable models like Arduino Nano or Mega will work as well. However, do *not* connect the V+ port of the PCA9685 driver board to the Arduino, because the current drained will most probably destroy the microcontroller. Instead, make sure to connect an appropriate power supply to the terminal headers of the PCA9685.

The servos are plugged into port 0 (180°) and 1 (270°), respectively. I used some 3D-printed clips to have a clean wiring of the servo cables (see Fig. 1.3). The print file is available in the Thingiverse project.

1.2 Software and calibration

Now the device is assembled, we need to load the program (also called *sketch*) to the Arduino board and calibrate the movements. Calibration may include small modification of the hardware assembly as well as adaptation of configuration values in the software. Let’s address these aspects one by one.



(a) Vertical position of "pusher" (b) Maximum turn

Figure 1.4: Calibration of the vertical turn

1.2.1 Arduino

You won't have to understand programming of Arduino boards, however, you need to build and deploy sketches and most probably adapt some values. Hence, you need some software allowing just this. I strongly propose to download and install the official Arduino IDE[5] provided on the website[6] and in the Microsoft store for Windows-based systems.

Connecting to a computer Once installed, connect the Arduino board to your computer using an USB cable and open the main source file *PocketCube.ino* in the Arduino IDE. The IDE should now show the source code in *PocketCube.ino* and, above that, tabs to select the other project files such as *Config.h*.

Next, it is important that the IDE knows which type of board you are using and how to communicate with it. For this, have a look at the *Tools* menu at the top. Set the entry *Board* to the correct Arduino model (e. g. *Arduino Uno*) and the entry *Port* to the COM port the Arduino is connected to. Typically, the selection contains only one port. In case there is more than one port to select, it is just a matter of trying, which one is the correct one. If the selected one does not work (i. e., you cannot load the sketch to the board), try another one.

Loading the sketch Press the button showing a right arrow—the second button from the left, just below the menu bar—to compile the source code and upload it to the Arduino board. In case compilation and upload were successful, the Arduino will automatically starts executing the program. While the Arduino gets its power supply from the computer, please note that the servo motors cannot move, unless you have also switched on the external servo power supply.

1.2.2 Vertical turn

Let's start the calibration by fine-tuning the vertical turn, i.e., the slider movement tilting the cube to its side without rotating any face:

1. Adjust the vertical position of the sliding "pushing unit" so that the bars are at the bottom of the upper cube layer (Fig. 1.4a).

2. Next, let's make sure that the device performs a vertical turn each time the sketch is uploaded or the Arduino's reset button gets pressed. Look for the function *setup()* in the file *PocketCube.ino* and add `servos.turnCube()` at its end:

```
void setup() {
  // Leave other code as it is ...

  servos.turnCube();
}
```

[5] Integrated development environment
[6] `https://www.arduino.cc/en/software` (visited: 21.07.2023)

3. The file *Config.h* defines values for the minimum and maximum position of the turn servo, for instance, 100 and 380 in the code snipplet below:

```
#define TURN_SERVO_MIN 100          // Far crossbar just holding cube
#define TURN_SERVO_MAX 380          // Close crossbar pushing cube over
```

   Adjust the minimum value so that the far crossbar almost touches the Pocket cube (Fig. 1.4a). Upload the sketch after each modification, so that the new value takes effect.

4. In the same way, adjust the maximum value so that the cube is turned flawlessly. Note that pushing the cube as far as possible will not give you good results. Instead, you want the cube to falls to its side, while the crossbar drags it securely back into the tray. Figure 1.4b shows a good maximum position.

5. Remove `servos.turnCube()` from the function *setup()* and upload the sketch.

1.2.3 Horizontal rotation

Finally, we make sure that the Pocket cube's sides are parallel to the pusher's crossbars when rotated in 90° steps. The procedure is similar to the calibration of the vertical turn:

1. Add code to *setup()* to move the servo to the positions 0°, 90°, 180°, and 270° after upload of a sketch or reset of the Arduino board:

```
void setup() {
  // Leave other code as it is ...

  for (int i = 0; i < 4; i++) {
    delay(2000);              // Wait 2 s
    servos.rotateRight();     // Rotate 90 degrees to the right
  }
}
```

2. The respective servo values for 0°, 90°, 180°, and 270° in *Config.h* are as follows:

```
#define ROTATE_SERVO_0   102    //   0 degrees
#define ROTATE_SERVO_90  247    //  90 degrees
#define ROTATE_SERVO_180 397    // 180 degrees
#define ROTATE_SERVO_270 533    // 270 degrees
```

   Adapt these values so that the cube's faces are as parallel as possible to the pusher's crossbars. This is best seen looking directly down from above the cube.

3. Remove the added code from *setup()* and upload the sketch.

Chapter 2

Software Interface (Arduino)

Strictly speaking, you don't need to know how the Arduino communicates with your computer, because I provide an easy to use Python application interface, which is described in the next chapter. Still, if you followed the steps in the last chapter, you have assembled the device and equipped the Arduino with its program and calibrated values. Why not make yourself familiar with the Arduino's software interface used by the Python code—it is quite simple, anyhow.

2.1 Logical movements

Correctly assembled and calibrated, the Pocket cube solver is capable to rotate cubes in two ways: horizontal rotation of the bottom layer and vertical turn. Let's clarify the effect on cubes inserted to the device before looking at implementation details.

2.1.1 Horizontal rotation

Figure 2.1a illustrates a cube, which has not been scrambled. In the following assume the front face (i. e., the orange face) to be at the opposite side of the pushing bar containing the words "Pocket Cube".

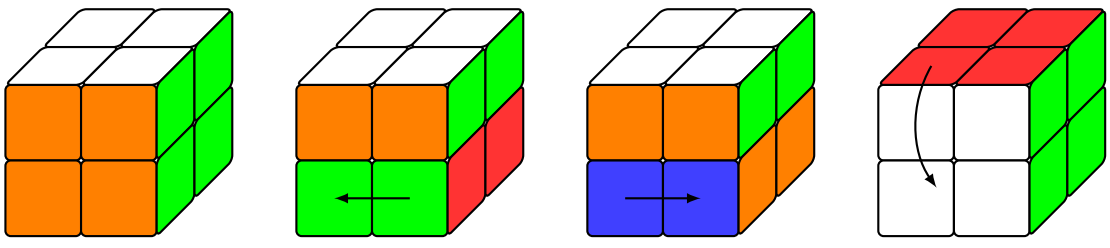
Rotations take place in steps of 90° to the left or right as illustrated in Figure 2.1b and c. As we need to cover all four directions—meaning, all four sides of the bottom layer can be rotated to the front—we use a servo with a range of 270°. For the sake of simplicity and concreteness, let's represent the rotation by an angle $\alpha_r \in \{0°, 90°, 180°, 270°\}$ with $\alpha_r = 0°$ being the initial state. Theoretically, we can keep on turning right (or left) for many times. However, due to the periodic nature of the movement, decreasing $\alpha_r = 0°$ by 90° will result in $\alpha_r = 270°$. Naturally, increasing $\alpha_r = 270°$ by 90° will result in $\alpha_r = 0°$:

$$\cdots \leftrightarrow 270° \leftrightarrow 0° \leftrightarrow 90° \leftrightarrow 180° \leftrightarrow 270° \leftrightarrow 0° \leftrightarrow 90° \leftrightarrow \cdots$$

Be aware, though, that the servo cannot move in full circles. Therefore, 90° rotations between 0° and 270° require a physical movement of 270° (i. e., three steps in the opposite direction) instead of the logical movement of 90° (i. e., one step).

2.1.2 Vertical turn

The vertical turn does not rotate a single face of the cube. Instead, it tips over the cube, rotating the whole cube by 90° as illustrated in Figure 2.1d.



(a) Initial state (b) Rotate left (L) (c) Rotate right (R) (d) Vertical turn (T)

Figure 2.1: Effect of servo movements on the cube. Rotation moves the bottom layer by 90° to the left (b) of right (c). A vertical turn pushes the cube over, rotating the whole cube by 90° towards the observer (d).

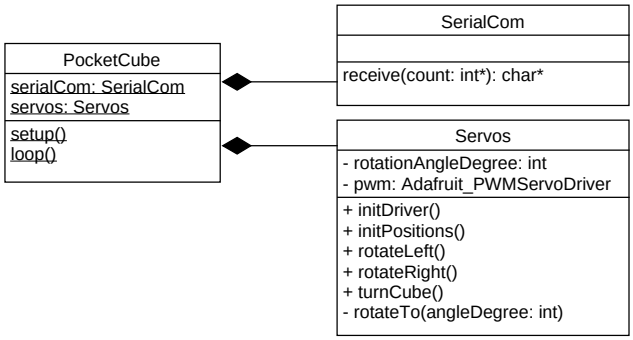


Figure 2.2: Class diagram

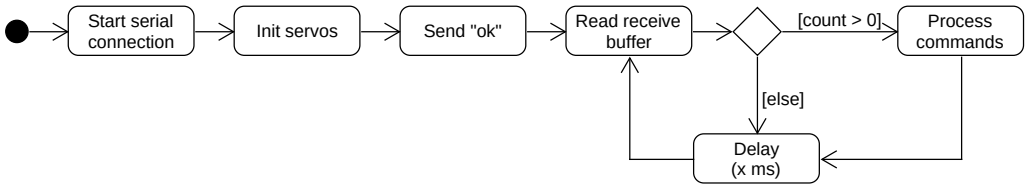


Figure 2.3: Program flow

2.2 Implementation

2.2.1 Software structure and program flow

Arduino programs tend to be comparatively small due to the limited capabilities of the boards. Nonetheless, I have structured the sketch in several files and classes in the hope to increase readability of the source code. The diagram[1] in Figure 2.2 shows the principal structure.

PocketCube takes care to initialize the device and enters an infinite loop. Within the loop it keeps listening for commands sent from the connected computer and reacts to them accordingly (Fig. 2.3). To do this, *PocketCube* contains an object each of the classes *SerialCom* and *Servos* to check for received commands and to move the servo motors, respectively. As described in Chapter 1, servo specific values are located in file *Config.h* and must be adapted for each device individually.

The source files are dependent of *Adafruit_PWMServoDriver.h* to control the PCA9685 board and *Wire.h* to access the PCA9685 board using the I^2C bus. You may need to install these libraries in your Arduino IDE using the library item from the *Tools* menu.

2.2.2 Serial communication

As stated before, the Arduino needs to be connected to a computer by a USB cable. The board establishes a serial connection with a baud rate 9,600 and uses it to receive and send text strings. Received strings are interpreted and processed character by character.

Table 2.1 lists all valid characters. The device interprets these as commands and ignores all other received values. In addition to characters triggering servo movements, a command requesting a reply exists. This allows the sending instance (e. g., a Python script on a connected computer) to synchronize to the time required for the movements. For instance, a script can send several

[1] Yes, I am aware that *PocketCube* is not a class. But it is about readability, isn't it?

Table 2.1: Accepted command characters and their meaning

Command	Functionality	Servos	Reply
I	Initialize servos	$\alpha_r = 0°$, $\alpha_t = \alpha_t^{min}$	
L	Rotate left	$\alpha_r - 90°$	
R	Rotate right	$\alpha_r + 90°$	
T	Turn sequence	$\alpha_t = \alpha_t^{max}$, $\alpha_t = \alpha_t^{min}$	
>	Send reply		ok

commands for servo movements concluded by the character >. As the Arduino processes one command after the other, it will not send the reply before having completed the movements.

Chapter 3

Application Software Interface (Python)

In the last chapters we have seen how to assemble and setup the device and how the cube solver can be controlled from a connected computer. However, you don't want to bother sending low-level commands to turn the cube's bottom layer or tilt the cube. Instead, it is much more comfortable to use a common notation to turn a cube's individual faces clockwise and counter-clockwise.

Algorithms and strategies to solve a cube are typically formulated using such notation as introduced in Section 3.1. However, the physical device does not have enough degrees of freedom to rotate all faces with a single servo motion. Instead, cubes need to be turned, by this changing their orientation (e. g., being upside down or tilted to the left or right). A first set of servo commands will bring the cube back in its original orientation after each logical rotation. It turns out to be much more efficient (i. e., require less servo motions), though, to keep track of the cube's actual orientation and adapt the next rotation accordingly.

This chapter introduces an appropriate notation and translates the turns of each face to the device's servo movements. Building upon this, an easy to use Python application programming interface to control the cube solver is provided.

3.1 Pocket cube notations

We need to distinguish the cube's faces, e. g., to unambiguously notate rotations. According to their position, faces are commonly denoted *up*, *down*, *front*, *back*, *left*, and *right* as illustrated in Figure 3.1.

Rotating a face by 90° is notated by the face's first letter. Capital letters U, D, F, B, L, and R represent clockwise rotation when looking straight at the appropriate face. Counterclockwise rotations are typically denoted by an apostrophe: U', D', F', B', L', and R'. However, depending on the programming language used, an apostrophe may clash with single quote marks used to define text strings or characters in software development. For this reason, we will use the small letters u, d, f, b, l, and r, instead. Finally, rotating a face by 180° corresponds to rotating by 90° twice. This is commonly expressed by adding a 2 to the letter: U2, D2, F2, B2, L2, R2.

3.2 Recurring cube orientation (ReCor)

The physical Pocket cube solver is limited to the movements of both servo motors, being rotations D and D' of the bottom layer (*down*) as well as tilting the cube as a whole towards *front* (see Section 2.1 on page 13). Hence, we need to implement all rotations of the other five faces *up*, *front*, *back*, *left*, and *right* by combinations of these actions.

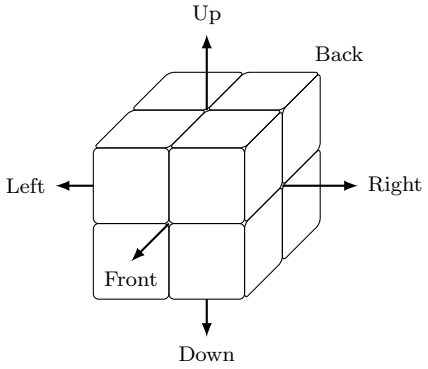


Figure 3.1: Unique identifiers for a cube's faces

Table 3.1: Recurring cube orientation

Move	Servo commands	Move	Servo commands
U	TTRTT	D	R
U', u	TTLTT	D', d	L
U2	TTRRTT	D2	RR
F	TRTTT	B	TTTRT
F', f	TLTTT	B', b	TTTLT
F2	TRRTTT	B2	TTTRRT
L	(TLTTRT) R (TRTTLT)	R	(TRTTLT) R (TLTTRT)
L', l	(TLTTRT) L (TRTTLT)	R', r	(TRTTLT) L (TLTTRT)
L2	(TLTTRT) RR (TRTTLT)	R2	(TRTTLT) RR (TLTTRT)
tl ↶	TLTTRT	tr ↷	TRTTLT

A straight-forward approach is to move the cube such that the face to rotate is at the bottom, perform the desired rotation, and move the cube back into its original orientation. As the cube will establish its original orientation after each and every move, let's introduce the term *Recurring cube orientation* (*ReCor*) for this method.

Table 3.1 summarizes the servo movements required for ±90° and 180° rotations of all faces. In addition, moves to tilt the cube as a whole to the left (tl) and right (tr) are defined in the last row of the table. These moves are required to bring the faces *left* and *right* into the bottom layer and back into their original orientation.

According to Table 2.1 on page 15 the command sequences in Table 3.1 can be send exactly as stated in the table as text strings to the Arduino controlling the servos. Note that parentheses and blanks inserted for better readability will be ignored by the micro-controller.

3.3 Spinning cube orientation (SpiCor)

An obvious weakness of ReCor is the large number of servo movements required to rotate a face, ranging from one (U and U') to 14 (L2 and R2). This is caused by the approach to bring back the cube to its original orientation.

Dropping the requirement of re-establishing the cube's orientation may cut down the number of servo steps significantly. For instance, executing the single servo command R instead of the sequence TTRTT consisting of five commands will result in the same state of the cube, however, the cube as a whole is rotated by 90° to the left. Much the same, not tilting the cube back in rotations of the faces *left* and *right* will save six servo actions (TLTTRT or TRTTLT), each.

In the following, we will investigate this approach, which we will call *Spinning cube orientation* as the cube's orientation changes in contrast to being recurrently restored in ReCor.

3.3.1 Mathematical representation

Let's introduce a mathematical representation which simplifies keeping track of the cube's orientation in space. Additionally, it will allow to transpose desired rotations like R or D2—which are relative to the original orientation in Figure 3.1—to the actual orientation of a cube.

Coordinate system As depicted in Figure 3.2, the origin of the coordinate system is located in the cube's center. The positive x-axis intersects the *front* face at its center. Likewise, the positive y- and z-axes intersect *right* and *up*, respectively. The distance of the origin to all faces is 1 unit.

Faces Cube faces can be represented by vectors $\mathbf{x} \in \mathbb{R}^3$ pointing to the faces' centers. As the faces *front*, *right*, and *up* are in unit distance $|\mathbf{x}| = 1$ on the positive x-, y-, and z-axis, respectively,

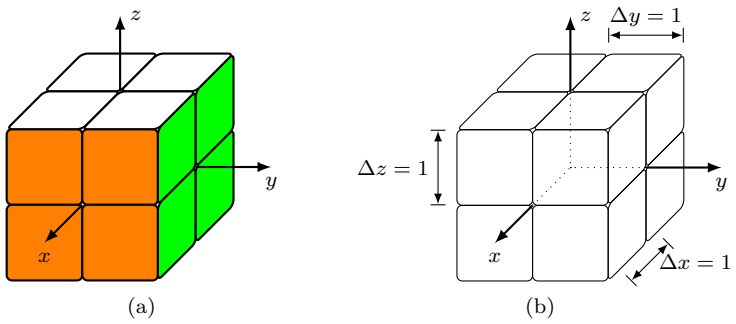


Figure 3.2: Mapping of a cube to a 3D-coordinate system

Table 3.2: Coordinate system locations mapped to faces

Location	$\mathbf{x}$	Face
$+x$	$\mathbf{e}_x$	F
$-x$	$-\mathbf{e}_x$	B
$+y$	$\mathbf{e}_y$	R
$-y$	$-\mathbf{e}_y$	L
$+z$	$\mathbf{e}_z$	U
$-z$	$-\mathbf{e}_z$	D

they are represented by the coordinate system's unit vectors $\mathbf{e}_x$, $\mathbf{e}_y$, and $\mathbf{e}_z$:

$$\mathbf{x}_F = \mathbf{e}_x = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad \mathbf{x}_R = \mathbf{e}_y = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad \mathbf{x}_U = \mathbf{e}_z = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

Likewise, *back*, *left*, and *down* are in unit distance on the respective negative axes:

$$\mathbf{x}_B = -\mathbf{e}_x = \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix}, \quad \mathbf{x}_L = -\mathbf{e}_y = \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix}, \quad \mathbf{x}_D = -\mathbf{e}_z = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$$

It shows that we do not need vector calculation to transpose rotations. It seems helpful, however, to denote a face's location by the direction the appropriate vector is pointing to. Table 3.2 summarizes the notation for all faces.

3.3.2 Cube orientation

We will rotate and tilt the cube by the device's servo motors. Doing so, the cube will change its orientation, for instance, the original face *up* will be at the left after tilting the cube by 90° to the left. Consequently, we will have to keep track of the orientation to apply the next rotations like U or L2 to the correct faces.

Opposite faces remain in opposite locations, for instance, *up* is always on the opposite side of *down*—no matter how you rotate the cube. Hence, it is sufficient to keep track of one face per axes. We choose the faces *front*, *right*, and *up* with the initial locations $+x$, $+y$, and $+z$, respectively.

3.3.3 Cube face rotations

So far we have defined locations of faces as well as cube orientations. Next, let's investigate the minimum number of servo movements required for all face rotations as well as how these change the cube orientation.

Table 3.3: Spinning cube commands and impact on orientation

Move	Servo commands	$+x$	$-x$	$+y$	$-y$	$+z$	$-z$
U	R	$+y$	$-y$	$-x$	$+x$	$+z$	$-z$
U', u	L	$-y$	$+y$	$+x$	$-x$	$+z$	$-z$
U2	RR	$-x$	$+x$	$-y$	$+y$	$+z$	$-z$
D	R	$+x$	$-x$	$+y$	$-y$	$+z$	$-z$
D', d	L	$+x$	$-x$	$+y$	$-y$	$+z$	$-z$
D2	RR	$+x$	$-x$	$+y$	$-y$	$+z$	$-z$
F	TR	$-z$	$+z$	$+y$	$-y$	$+x$	$-x$
F', f	TL	$-z$	$+z$	$+y$	$-y$	$+x$	$-x$
F2	TRR	$-z$	$+z$	$+y$	$-y$	$+x$	$-x$
B	TR	$-z$	$+z$	$+y$	$-y$	$+y$	$-y$
B', b	TL	$-z$	$+z$	$+y$	$-y$	$-y$	$+y$
B2	TRR	$-z$	$+z$	$+y$	$-y$	$-x$	$+x$
L	(TLTTRT) R	$+x$	$-x$	$+z$	$-z$	$-y$	$+y$
L', l	(TLTTRT) L	$+x$	$-x$	$+z$	$-z$	$-y$	$+y$
L2	(TLTTRT) RR	$+x$	$-x$	$+z$	$-z$	$-y$	$+y$
R	(TRTTLT) R	$+x$	$-x$	$-z$	$+z$	$+y$	$-y$
R', r	(TRTTLT) L	$+x$	$-x$	$-z$	$+z$	$+y$	$-y$
R2	(TRTTLT) RR	$+x$	$-x$	$-z$	$+z$	$+y$	$-y$
tl ↶	TLTTRT	$+x$	$-x$	$+z$	$-z$	$-y$	$+y$
tr ↷	TRTTLT	$+x$	$-x$	$-z$	$+z$	$+y$	$-y$

Table 3.4: Spinning cube faces depending on orientation

Move	Mapped face $\mathbf{x}_{SpiCor}$
U, u	$\mathbf{x}_u$
D, d	$-\mathbf{x}_u$
F, f	$\mathbf{x}_f$
B, b	$-\mathbf{x}_f$
L, l	$-\mathbf{x}_r$
R, r	$\mathbf{x}_r$

Orientation To be honest, there is no maths needed at all, but it is just a matter of investigating how rotations of faces actually done by the physical device impact the orientation of the cube's faces. Table 3.3 summarizes the results required for a software implementation of the SpiCor method. For instance, the rotation F is performed by servo movments TR. After these commands, a face that was originally in orientation $+z$ (i. e., at upper side of the cube) will be in orientation $+x$ (i. e., at the front). Applying this mapping to the orientations of the cube's faces *front*, *right*, and *up* for each rotation will keep track of the orientation of these faces.

Rotation By now we know which rotation to perform next, however, our cube may be in a different orientation than in the beginning. We must transform this *logical* rotation to the corresponding rotation in our cube's current orientation (e. g., if the cube is lying on its left face, a rotation U will actually be achived by L). As we have kept track of the orientation of *front*, *right*, and *up*, this is not difficult to achieve. As depicted in Table 3.4, we can use these face's orientations to determine which face to rotate.

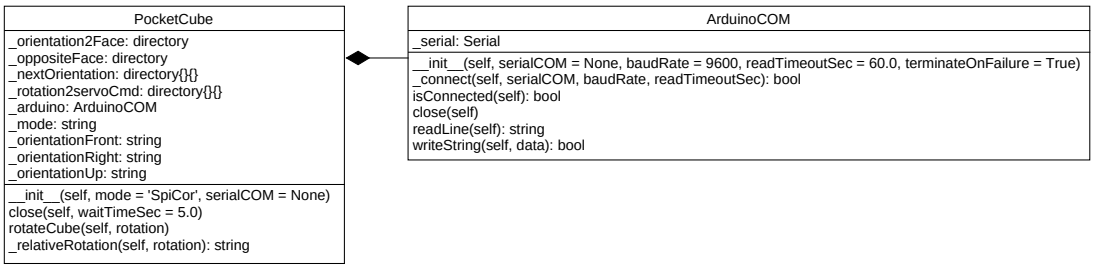


Figure 3.3: Class diagram

3.4 Implementation

As shown in Figure 3.3 the device can be controlled by an object of class *PocketCube*, which uses *ArduinoCOM* to communicate with the Arduino board.

3.4.1 Serial communication

Class *ArduinoCOM* requires the dependency *pyserial* to be installed in your Python environment. The constructor allows trying to connect to a specific COM port corresponding to the value passed to parameter *serialCOM*. In case *serialCOM* is *None*, the object tries ports 0 through 15. Once, a connection has been established, the methods *writeString()* and *readLine()* allow to send and receive character strings to and from the Arduino, for instance:

```
arduino = ArduinoCOM()
reply = arduino.readLine()          # Wait for setup() to complete
print('Device ready: ' + reply)

arduino.writeString('RRLT')
arduino.writeString('I')
arduino.close()
```

3.4.2 Rotations

Class *PocketCube* contains several directories mapping, for instance, orientations, faces, rotations, and servo commands as elaborated in this chapter. The mode can be set to `'ReCor'` or `'SpiCor'`, corresponding to Recurring Cube Orientation or Spinning Cube Orientation. Naturally, the latter is typically the better choice, because it requires less servo movements. Once, the object is created and connected to the device, the method *rotateCube()* allows to rotate all faces by 90° or 180°.

The following sample code rotates the front face by 90° in each direction, meaning, the cube will not be changed. However, as the mode is set to `'SpiCor'` the cube will not be turned back into its original orientation between and after the rotations.

```
cube = PocketCube()
cube.rotateCube('F')
cube.rotateCube('f')
cube.close()
```