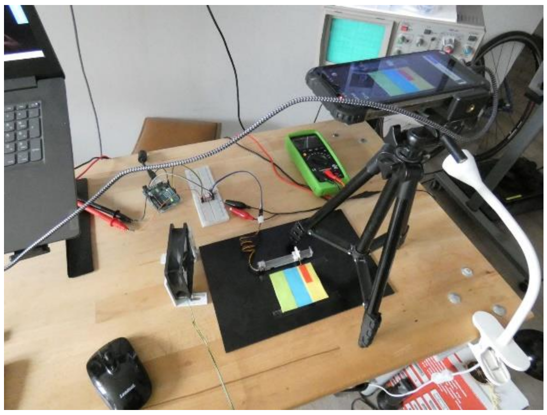
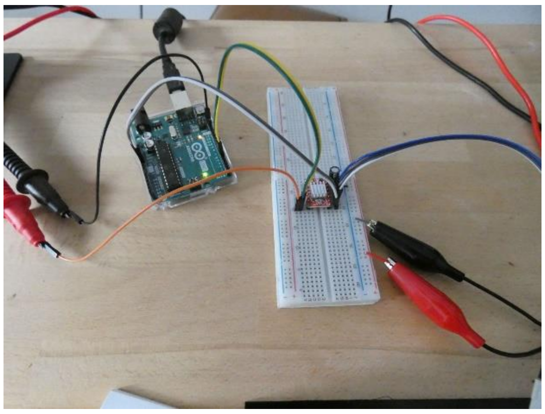
Documentation for Local Hardware Setup

created to solve Task 2 of the Raspberry Pi Project in WS21/22





Some key-attributes:

- Arduino UNO as hardware interface, controlled from the python script by using PyMata4

- Phone camera embedded using 'Droidcam' App/Client over USB connection

- Motor incl. linear rail (https://www.amazon.de/gp/product/B07X6J8C3T/ref=ppx_yo_dt_b_asin_title_o04_s01?ie=UTF8&psc=1)

- A4988 motor-driver (https://cdn.shopify.com/s/files/1/1509/1638/files/A4988_Stepper_Motor_Driver_Datenblatt_AZ-Delivery_Vertriebs_GmbH.pdf?v=1608626085)

- The Control of the motor works the same way as the for the one in the RaspPi '2-Pin'-Setup (PIN_DIR, PIN_STEP), single/half/micro-step operation possible

- Number of steps per rotation is far smaller, but as horizontal movement per rotation/step is far smaller as well, the step-numbers from the original can be used to maintain well observable motion

- Due to the non-cyclic pattern of movement, I implemented a 'turn around'-mechanism (in code) that makes the marker travel the remaining distance in the opposite direction when reaching the edges of the setup

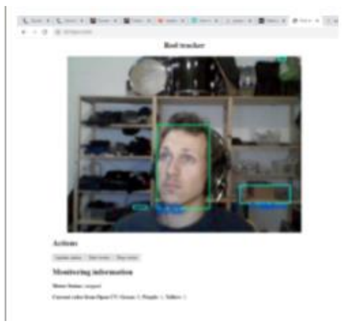
-As the motor tends to quickly run hot even though the current is far below the given limit of 0.8A, I installed a fan and also use only 5 Volt of supply voltage as I don't trust the quality of the cheap motor really.

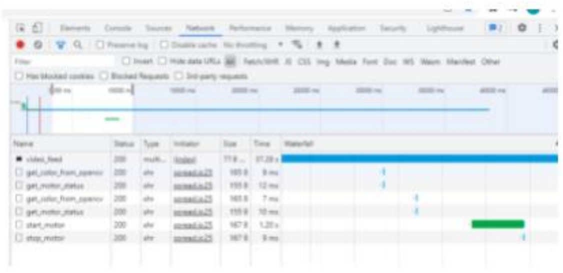
- Unfortunately, the operating speed of the setup therefore is slow, transitioning from one field to another takes ~10 sec.

- Apart from the 'turn-around'-mechanism, the usage of a cyan instead of a purple field (due to a lack of usable purple material) and the addressing of PyMata where GPIO was addressed originally everything is identical. I also update everything within an own branch within the repository.

Some explanation on the code:

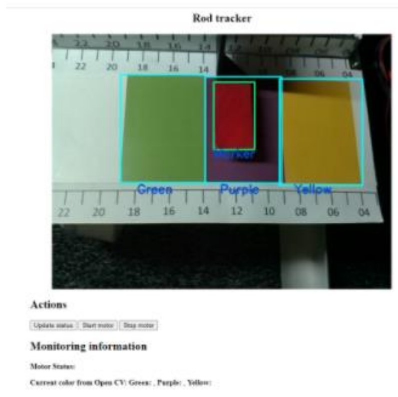
- As the video stream was a critical aspect within the debugging sessions on the remote hardware setup due to 'blocked' http requests, a robust, simple approach based on 'picture refreshing' was chosen.
-

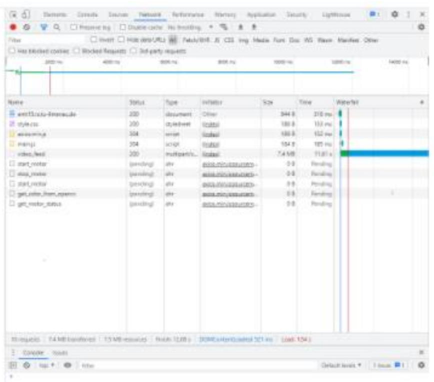




Name	Status	Type	Initiator	Size	Time
video_feed	200	multi...	(index)	77.8 ...	37.28 s
get_color_from_opencv	200	xhr	[illegible]	165 B	9 ms
get_motor_status	200	xhr	[illegible]	155 B	12 ms
get_color_from_opencv	200	xhr	[illegible]	165 B	7 ms
get_motor_status	200	xhr	[illegible]	155 B	10 ms
start_motor	200	xhr	[illegible]	167 B	1.20 s
stop_motor	200	xhr	[illegible]	167 B	9 ms

(working video stream in local debugging -> no requests get stuck in state 'pending')





(Problematic video stream in remote debugging -> requests get stuck in state 'pending')

- The image recording/processing/actualization has therefore been integrated into the 'main motor loop'

- A new image is therefore recorded in each cycle of the active motor loop (detailed explanation later)
  -> new image each predefined 'angle' (number of steps), no image update when motor is resting
  -> image 'buffer.jpg' (part of the 'static' folder) is overwritten each cycle
- The image update on the webpage is executed in an asynchronous loop within javascript
- A common 'trick' is applied: image source is being updated as the the image source is manipulated periodically by adding a 'timestamp' -> change is registered, and image is then reloaded without changing the 'actual' source file

```
async function updateFrame() {
  try {
    // Make request to server
    setTimeout(function() {
      img=document.getElementById('background')
      img.src = img.src.split("?")[0] + "?" + new Date().getTime()
      console.log('frame update');
    }, 100);

  } catch (e) {
    console.log('Error getting the status', e)
    updateStatus('Error getting the status')
  }
}
```

Few cases of missing pictures can be observed (~1-2 per run) – probably due to bad timing (updating image while image is currently overwritten). Yet still no failure (caught errors)

'Motor-loop' – detailed explanation:

Function 'approx_step' contains the 'main loop' for running the motor interval-wise and for triggering the image processing/update.

As mentioned previously, a new image and a new update on the next action to be executed is generated each 'angle' (number of steps).

The two main operational variables are 'check_arr' and 'compare_arr'.

```
 def approx_step(self,compare_array):
        #GPIO.setmode(GPIO.BCM)
        #GPIO.setup(self.PIN_DIR, GPIO.OUT)
        #GPIO.setup(self.PIN_STEP, GPIO.OUT)

        board.set_pin_mode_digital_output(self.PIN_DIR)
        board.set_pin_mode_digital_output(self.PIN_STEP)

        angle =30 #set as desired
        #total_steps=1600 #for RaspPi setup

        counter=0

        self.check_arr = opencv_controller.get_current_color()
```

- ***'compare_arr'*** represents ***the values to be reached***
- ***'compare_arr'*** is either set to a 'realistic' value (all except for *(1,1,1),(1,0,1),(1,1,1),(0,0,0)*) or one of the 'non-realistic' ones (*(1,1,1),(1,0,1),(1,1,1),(0,0,0)*).
- In the current version (0,1,0) is used to trigger the 'auto-cyan' mode ('aiming' for cyan).
- The 'non-realistic' *(1,1,1)* is used to state a case where one of the 'random movements' will be executed (-> 'not aiming')
- ***'check_arr'*** represents the output of the processing within ***opencv_controller.get_current_color()***.
- In case of 'aiming' '***check_arr'*** is used to determine the direction the motor must be turning and when to stop (aim reached)

```
-           if compare_array == (1,1,1): #case for 'non'-aiming
-
-               angle_bit = random.getrandbits(1)
-               dir_bit = random.getrandbits(1)
-
-               if angle_bit == 1:
-                   angle_total = self.strot_90
-                   self.amt= '90'
-               else:
-                   angle_total = self.strot_270
-                   self.amt= '270'
-
-               if dir_bit == 1:
-                   dir = self.CW
-                   self.out_dir= 'clockwise direction'
-               else:
-                   dir = self.CCW
-                   self.out_dir= 'counter-clockwise direction'
-
-               self.out_message = 'rotating ' + str(angle_total)  + ' steps in
   '  + str(self.out_dir)
-
-               max= int(angle_total/angle)
-               board.digital_write(self.PIN_DIR, dir)
-               #GPIO.output(self.PIN_DIR, dir)
-           else:
-               max=3000
```

- As previously mentioned,*''**compare_arr'**=* =*(1,1,1)'* is used to trigger the 'random' mode.
- Python's *random*-function is used to randomly assign the direction and range of the following movement
- Variable ***'max'*** is then the number of intervals/cycles of range ***'angle'*** that is then executed
- PyMata's ***'digital_write'*** then sets the direction the motor is rotating
- In case of ***'aiming'*** a random value of 3000 is assigned to ***'max'*** as the motor will be stopped if ***'compare_arr'*** == ***'check_arr'.*** Max has no influence on here.
- y is the incremental variable for the main loop. There are several 'break' cases included. The comments inside the code do explain each of these
- as explained, the new frame is processed every cycle

```
        for y in range(max): # main loop controlling all actions/checks

            if self.working == False:
                print('out of motor loop')
                break

            #call opencv_controller for processing frame once per cycle ->
every 'angle' (number of) steps
```

```
            opencv_controller.process_frame()
            self.check_arr = opencv_controller.get_current_color()
```

As the Arduino/Linear-rail setup of course can't exceed the edges of the rail, turn-around mechanisms had to be implemented. These are recognized by *opencv_controller* output ***'check_arr' ==(1,0,1)*** (green limit) /***'check_arr' ==((1,1,1)*** (yellow limit).

The direction if then changed accordingly in order to avoid damage. The remaining number of intervals of size 'angle' is then executed in the opposite direction. The webpage output is modified accordingly.

This is another example for how the non-/realistic values are used to ***either state operational commands or actual color-value feedback.***

```
            #turn around if border reached - check_arr -> (1,0,1) for green limit
            if self.check_arr== (1,0,1) and counter > 3:
                dir=self.CCW
                board.digital_write(self.PIN_DIR, dir)
                #GPIO.output(self.PIN_DIR, dir)
                print('turned around')
                self.out_message = 'turned - remaining '+str(angle_total -y*angle)
+ ' steps in opposite direction'
                sleep(.5)
                counter = 0

            #turn around if border reached - check_arr -> (1,1,1) for yellow limit
            if self.check_arr== (1,1,1) and counter > 3:
                dir =self.CW
                board.digital_write(self.PIN_DIR, dir)
                #GPIO.output(self.PIN_DIR, dir)
                print('turned around')
                self.out_message = 'turned - remaining '+str(angle_total -y*angle)
+ ' steps in opposite direction'
                sleep(.5)
                counter = 0
```

The '>3' condition is included in order to 'give the marker time' to move out of the critical zone and avoid another unnecessary 'adjustment break'.

```
            #execute angle steps
            for x in range(angle):

                if self.working == True:

                    #GPIO.output(self.PIN_STEP, GPIO.HIGH)
                    board.digital_write(self.PIN_STEP, 1)
                    sleep(self.delay)
                    #GPIO.output(self.PIN_STEP, GPIO.LOW)
                    board.digital_write(self.PIN_STEP, 0)
                    sleep(self.delay)
                else:
                    break
```

This is the sub-loop which then let's the motor execute the ***'angle'***-interval before image processing/update and position check is executed again.

```python
            #cascade of possible 'break' (out of loop) scenarios
            if self.working == False:
                self.out_message = 'stopped'
                break

            if compare_array != (1,1,1) and self.check_arr == compare_array :
                self.out_message = 'finished'
                print('reached field',self.check_arr)
                break
            else:
                print('checked!')

            if y == max-1 :
                self.out_message = 'finished'
                print('reached field',self.check_arr)
                break

        #set to default state when loop is left
        self.working = False
```

The are different outputs for the different break-cases that either indicate an interrupt of the current process or a *'finsihed'* state when either cyan is reached or the random movement has been executed.

```python
    def is_working(self):
        self.out_arr = [bool(i) for i in self.check_arr]
        print('out_arr',self.out_arr)
        return self.out_message,self.out_arr[0],self.out_arr[1] ,self.out_arr[2]
```

Function 'is working' returns the color-vectors as well as the motor-state within a shared tuple. This method has been chosen as the JavaScript-side of the app returned an error when I tried to employ multiple parallel jsonify-returns from the flask app.