31.01.2022

RaspberryPi Project WiSe 21/22 – Documentation for Task 3

Submission based on local hardware setup – Group H

Implementing a functional environment for the HC-SR04 distance sensor based on a Windows-machine turned out to be a rather complicated task due to the **missing realtime-ability of Windows.**

The Arduino UNO that is controlled by using PyMata4 itself has the typical high frequency timing-capabilities of a microcontroller. As the Arduino is only used as a hardware-interface these resources can't be exploited straight away.

PyMata offers different template functions such as 'sonar' which is a special function to detect object distance using the HC-SR04.

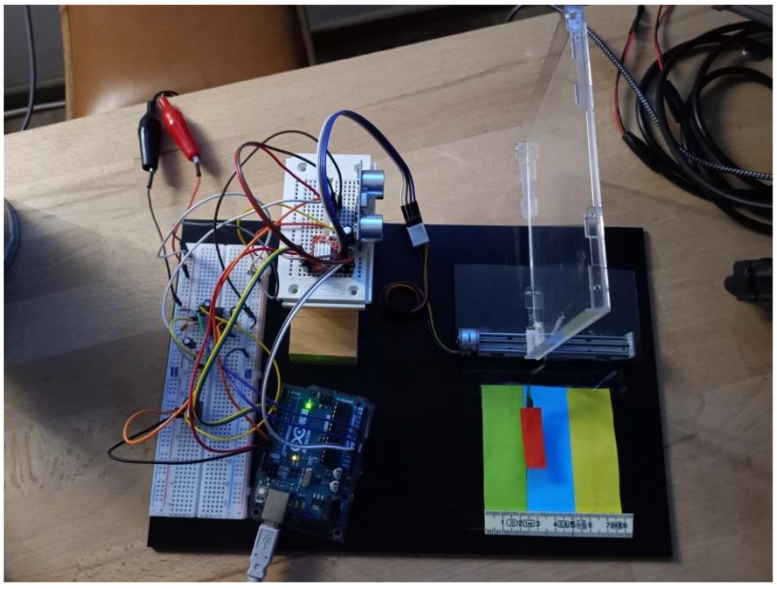
(https://github.com/MrYsLab/pymata4/blob/master/examples/hc-sr04_distance_sensor.py)

This default function unfortunately does not serve the actual requirements for two main reasons:

- The 'built-in' function only returns integer centimeter values which does not represent a sufficiently high resolution for the only 8cm large test image.
- Also, the built-in function does not leave space for any self-created concept until the point where the user decides to create an PyMata extension.

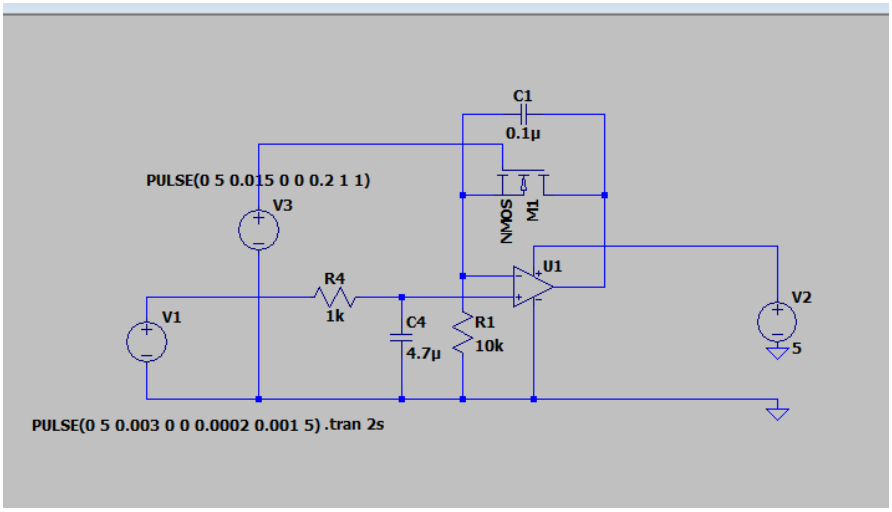
As a conclusion to the matter, one could say that the user would either need to quickly gain a non-sustainable amount of extra skill that heavily exceeds the original task or miss any chance to show proper understanding of the concept that's presented in the related tutorials.

To solve this issue a simple helper circuit based on a standard non-inverting, active RC-integrator including a reset function was introduced.

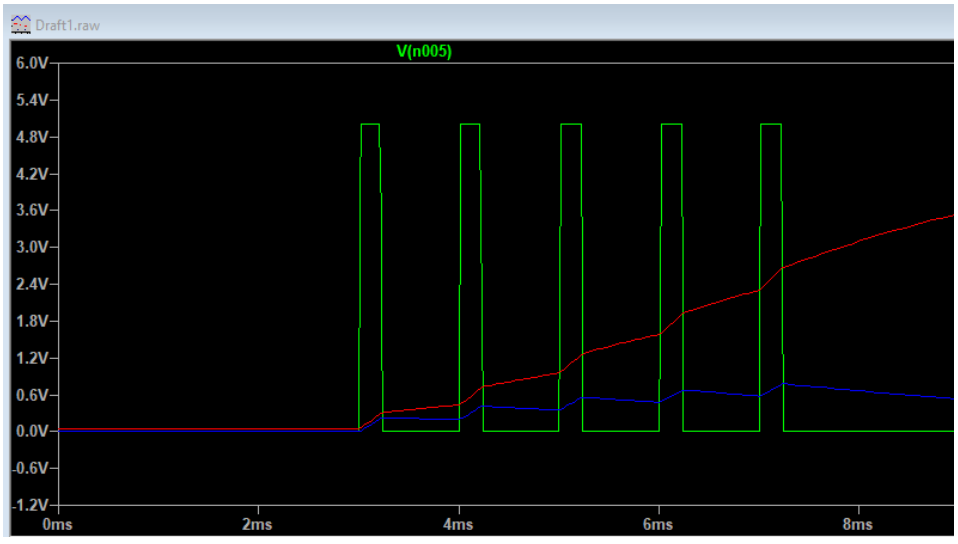


Instead of measuring the time between rising and falling edge, the integration of 20 pulses and the measurement of the resulting (analog) capacitor voltage was implemented.

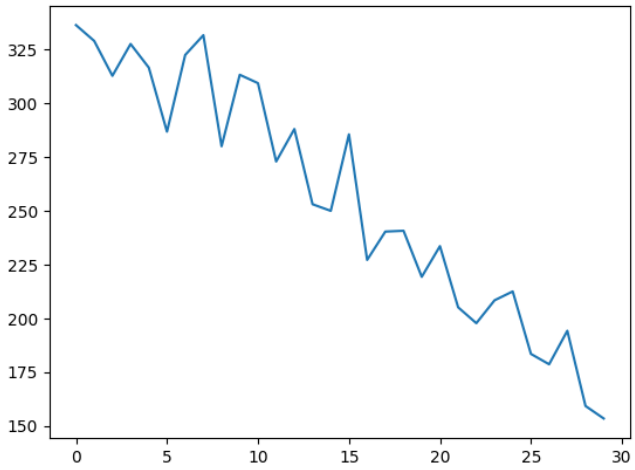
The great attribute of this solution is its simplicity as it only requires a standard LM324 OpAmp, a single N-channel Mosfet for the reset and fitting RC-components for the implemented pulse frequency.



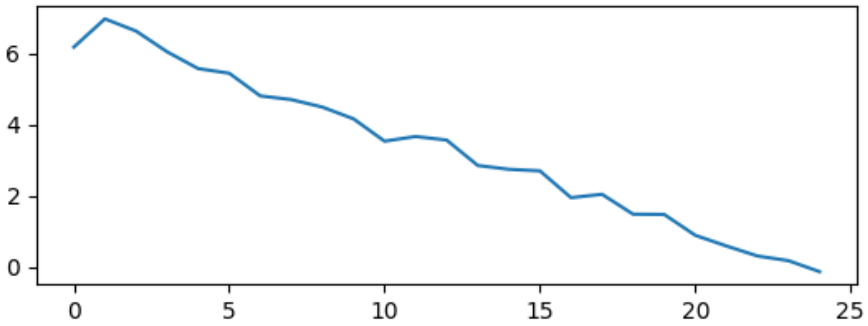
The sensor output that results from periodical triggering can be described as a pulsewidth-modulated signal. The measured distance is directly proportional to the duty-cycle of the signal. As the resulting capacitor voltage is then also directly proportional to the duty cycle, the detection of the distance based on the analog voltage over the RC-integrator is well plausible in theory.



At this point I'm already anticipating by pointing out to the main trade-off of this concept:The simplicity of the circuit comes with a low degree of reliability . This is mostly related noise sensitivity and unstable reference/ground-design. This issue was fixed by including several prediction/correction methods. The following graphs show a sequence of measured values while the distance of the sensor to counter part was continuously decreasing.



This first graph shows the linear relation of voltage value (corresponding value on y-axis) and distance. Unfortunately, the heavy distortion of the measurement values seems even more significant than the expected relation.



The second graph shows how including a simple prediction algorithm improves the result significantly (note to avoid confusion: the two runs did differ in scaling and number of steps)

The prediction algorithm here simply rejects all values that were larger than the previous one and initiates a new measurement if that is the case.

Approximately one third of all measurements was made up by those ‘error’-cases. The error cases required between 1-3 extra measurements until the result did fit the prediction.

For the repeated measurements a small ‘push into place’ function as implemented as well.

This makes good sense as the mechanical construction of the test-setup struggles with a lot of play Two small ‘pushes’ back and forth helped to align the counter-part for the measurement perpendicular to the sensor axis.

At this point it became obvious that the simple integrator-based setup was nor really usable without correction algorithms on the software side.

A reliable hardware based-solution should be implemented by using a digital counter/timer circuit instead of the analogue integrator. Apart form this an extension of the usual PyMata 'sonar'-function became an interesting project-idea as well.

In general, the combination of test-setup size (8cm), sensor accuracy (>3mm), mechanical build-quality of the setup (immense play of the sled ~20° movable when motor position fixed) and the poor reliability the analogue integrator-based solution does represent a rather band starting point to build a satisfyingly functioning system.

In the following section, I'd like to describe how the software environment handles the sensor control and the input processing based on mutable prediction algorithms.

As the circuit can't be directly translated to a corresponding cm-value, a calibration mechanism needed to be implemented. This procedure makes use of the green field as a reference of 2.7cm in width. The marker is first driven to the 0cm position (recognized by image processing), then 5 measurements are executed. The standard deviation of all measured values is being calculated and all values above or beyond this range around the average are discarded the 'normalized' average value is then set as the lower reference value. This is repeated for the upper (2.7 cm) value.

```
    def calibrate_sensor(self):
        #turn to left border
        dir = self.CCW
        self.board.digital_write(self.PIN_DIR, dir)
        self.working = True
        self.out_message = 'rotating to 0 cm'
        self.cm_val_out='calibrating'
        self.board.digital_write(self.PIN_SLEEP, 1)

        for n in range(3000):
            #execute angle steps
            for x in range(20):
                if self.working == True:
                    #GPIO.output(self.PIN_STEP, GPIO.HIGH)
                    self.board.digital_write(self.PIN_STEP, 1)
                    sleep(self.delay)
                    #GPIO.output(self.PIN_STEP, GPIO.LOW)
                    self.board.digital_write(self.PIN_STEP, 0)
                    sleep(self.delay)
                else:
                    self.out_message = 'stopped'
                    break
            opencv_controller.process_frame()
            self.check_arr = opencv_controller.get_current_color()
            #border reached - check_arr -> (0,0,0) for leftmost limit
            if self.check_arr== (0,0,0):
                self.out_message = 'paused'
                self.cm_val_out = 'measuring first reference'
                break
```

```
        self.board.digital_write(self.PIN_SLEEP, 0)
        for n in range(5):
            sensor_controller.cal_val(self.board,0)
            dir = n%3
            self.re_adjust(dir,10)
        print('read lower limit')
```

As this procedure also requires motor-control, the functions are spread over the Task2/Task3 files

```
def cal_val(self,board,border):
   if border== 0:
     if len(self.buff_arr_low)>4:
       self.buff_arr_low = np.append(np.array([]),self.track_rod(board,0,False)[1])
     else:
       self.buff_arr_low = np.append(self.buff_arr_low,self.track_rod(board,0,False)[1])
     av_low =np.average(self.buff_arr_low)
     std_dev=np.std(self.buff_arr_low)
     norm_arr_low=[n for n in self.buff_arr_low if n<av_low+std_dev and n>av_low-std_dev]
     self.lower =np.average(norm_arr_low)
     self.lower =np.average(self.buff_arr_low)
```

```
    def rep_handling(self,set_count,pred_enable):
        self.working = False
        self.out_message = 'distance measurement'
        self.dist_status='pending'
        self.re_adjust(self.CW,10)
        time.sleep(.5)
        self.re_adjust(self.CW,10)
        time.sleep(.5)
        self.re_adjust(self.CCW,25)
        rep_count,res_val,cm_val =sensor_controller.track_rod(self.board,set_count,pred_enable)
        if rep_count ==0:
            self.dist_status='distance finished'
            self.cm_val_out=str(cm_val) + ' cm'
            return
        else:
            self.dist_status='pending'
            self.cm_val_out=str(cm_val) + 'cm (correction pending)'
            dir=self.CCW#rep_count%2
            self.re_adjust(dir,10)
            return self.rep_handling(rep_count,pred_enable)
```

The **recursive** function ***rep_handling*** is used to handle the desired sequence of measurement/re-adjusting/prediction/correction. Parameters:

- pred_enable : inclusion of prediction based on opencv: on/off
- set_count : maximum number of allowed corrections

In case image-based prediction is applied, only values that represent an improvement compared to the previous value are considered.

```
def track_rod(self,board,set_count,pred_enable):
  rep_count =set_count
  res_val=self.control_sensor(board,5)
  cm_val_buff= self.get_distance(res_val)
  if rep_count != 0 and pred_enable == True:
    if cm_val_buff < self.col_low or cm_val_buff > self.col_up :
      rep_count -=1
      print('rep count',rep_count,'  ',cm_val_buff,'cm')
      comp = abs(cm_val_buff-self.col_low) + abs(cm_val_buff-self.col_up)
      if comp< abs(self.cm_val-self.col_low) + abs(self.cm_val-self.col_up):
        self.cm_val=cm_val_buff
        return rep_count,res_val,self.cm_val
      else:
        return rep_count,res_val,self.cm_val
    else:
      self.cm_val=cm_val_buff
      return 0,res_val,self.cm_val
  else:
    self.cm_val=cm_val_buff
    return 0,res_val,self.cm_val
```

track_rod is the counter-part for the recursive rep_handling functions. It is characterized by a cascade of optional return cases.

Function ***control_sensor*** executes the basic sensor control.

```
def control_sensor(self,board,rep):
  val_arr=np.zeros(rep)
  for k in range(rep):
    board.digital_write(8, 0)
    for i in range(20):
        board.digital_write(11, 1)
        time.sleep(0.001)
        board.digital_write(11, 0)
        time.sleep(0.001)
    value, time_stamp = board.analog_read(0)
    board.digital_write(8, 1)
    val_arr[k]= value

  av =np.average(val_arr)
  std_dev=np.std(val_arr)
  norm_arr=[n for n in val_arr if n<av+std_dev and n>av-std_dev]
  res_val=np.average(norm_arr)
  return res_val
```

Key-steps:

- ***board.digital_write(11, 1/0)*** represents the triggering of the sensor
- ***value, time_stamp = board.analog_read(0)*** reads the anlog voltage the capacitor has been charged with
- ***board.digital_write(8, 1)*** performs the reset of the integrator by switching the MosFet to conducting mode.

Web App:

The consideration of predicted values can be activated using the tick box in the web app. In case a color detected based on the sensor differs from the actual image, the wrong Boolean values are marked orange.

Actions

Stop Motor | Auto Random | Set Cyan | Calibrate | MeasDistance | ☑ consider image-based prediction

Monitoring information

Motor Status: stopped

Current color from Open CV: Green: true, **Cyan:** true, **Yellow:** false

Detected color from Sensor: Green: true, **Cyan:** true, **Yellow:** false

Distance: 2.0 cm

The scale of an old ruler has been added and a marker is drawn using opencv.line

Rod tracker