Measuring the Gravitational Acceleration using Raspberry Pi

# DOCUMENTATION AND REPORT

### Motivation

I have chosen to do this project for my internship at ELI-NP, not only because it fits my passion for Physics and Programming, but because this represented a good opportunity for me to apply in real-life all the subjects that I have been studying at the Automatics and Computer Science Faculty of Politehnica University of Bucharest.

For this project, I needed knowledge in Programming, Electronics, Operating Systems, Data Acquisition, Data Handling and Processing and Automation, perfectly fitting the scientific areas that I have been studying at the University.

Besides gaining a lot of knowledge in the above-mentioned areas, I have also improved other skills, such as: overcoming bugs, problems and challenges and tackling a project more complex that anything I had done before.

### Experimental Setup

The experiment uses both hardware and software prerequisites.

For the setup, we have used the following hardware:

1. 1x Raspberry Pi Model 3 B+
2. 1x Raspberry Pi V2 Camera
3. 1x 4GB SD Card
4. 3x 400W Halogen Spotlight
5. 1x Ethernet Cable
6. 3x Jumper Wire, M/F
7. 1x Jumper Wire, M/M
8. 1x IN4007 Diode Rectifier
9. 1x BC547 Bipolar Transistor (NPN)
10. 1x Resistor (1k Ohm)
11. 1x Breadboard
12. 1x Wooden Plank
13. 1x Electromagnetic Coil
14. 1x Metal Ball
15. Hardware
16. Ruler
17. Scissors
18. Duct Tape
19. Paint

Then, to measure the Gravitational Acceleration, we have used the following software:

1. Raspbian Stretch with Desktop OS
2. RaspiRaw Suite 0.2
3. dcraw
4. Python 3.7
5. OpenCV 3.4.2
6. Thonny IDE
7. GIMP
8. GNU Plot
9. SD Card Formatter
10. Etcher
11. PuTTY
12. WinSCP

### Setting up the Raspberry Pi

* 1. **Installing Raspbian Stretch**

Rasbpian Stretch is an Operating System developed by the Raspberry Pi Foundation specifically for the ARM Chipset on the Raspberry Pi. It is a variant of the Debian Jessie Linux distribution. The *.img* file can be downloaded from [here](https://www.raspberrypi.org/downloads/raspbian/).

Although the Raspberry Pi does have 1GB RAM, it does not have any storage memory. Instead, there is an SD Card slot installed on the board, where any Micro SD Card can be inserted, for loading the Operating System and storing data.

I have formatted the SD Card using a freeware, SD Card Formatter, downloaded from [here](https://www.sdcard.org/downloads/formatter_4/) and flashed the *.img* file using another freeware, Etcher, downloaded from [this](https://etcher.io/) website.

One particular challenge I have encountered is that I could not connect straight away to the Raspberry Pi’s SSH Interface. This is because, by default, the SSH Interface is disabled. I have tried to connect to the Raspberry Pi via SSH because, when connecting a mouse, a keyboard and an LCD to it, I was prompted with *Under-Voltage* warnings at the POST screen. I fixed this problem by creating an empty file, called *ssh* in the *boot* mounted device of the SD Card, that Windows can view.

Afterwards, I connected to the Raspberry Pi via local SSH (using PuTTY, on Windows 10), through an Ethernet cable, to turn on its Wi-Fi module and find out its IP. At this point, I needed no more wires to connect to the Raspberry Pi, though, for the most of the developing stage, I have connected directly to the Raspberry Pi (mouse, keyboard and LCD screen), because of the need of display modules.

* 1. **Setting up the working environment**

I have updated and upgraded the Raspbian and enabled the Camera, I2C and VNC interfaces that I would have been using later. Afterwards, I have expanded the filesystem, in order to be able make use of the whole storage capacity of the SD Card, and allocated 2 MB of Virtual RAM.

Another challenge was enabling the Desktop Environment. While Raspbian boots in Desktop Mode by default, if connected to an LCD, it also prompts on the terminal that there are some libraries that are not currently used, including some X libraries. As suggested by the OS, I removed the library pack, including the LX Terminal pack and X Pack. Thus, Raspbian was unable to boot in Desktop Mode anymore. I have fixed this by reinstalling the X and the LX Terminal library packs and by re-updating and re-upgrading the Raspbian OS.

### Installing OpenCV for Python

I have updated and upgraded any pre-installed packages:



Afterwards, I have installed developer tools, to compile OpenCV 3.4.2:



I have installed libraries and packages used to read various image formats from disk:



I have also installed libraries used to read video formats from disk:



Afterwards, I installed GTK to access OpenCV’s GUI features:



Then, I installed packages that are used to optimize various functions inside OpenCV:



At this point, I have proceeded to install *pip*, a Python package manager:



Then, I set up the virtual environments for Python development:



And then, I updated the *~/.bashrc* file:



I installed the Python headers and development files:



And then NumPy, because OpenCV represents images as NumPy arrays:



I downloaded OpenCV from GitHub:



And then, I set up the build:



Afterwards, I compiled and installed it:



With this, the OpenCV was set up for Python development.

### Setting up RaspiRaw

RaspiRaw is a program developed by the Raspberry Pi community, which aims to expand the functionality of the classic Raspberry Pi V2 Camera. It has multiple modes of filming and can film with up to 1000 frames per second. The output format is *.raw*, which can be converted to the more convenient format *.ppm* using a tool within the RaspiRaw suite.

I have downloaded and installed RaspiRaw using GitHub:



Once set up, I have altered the *raw2ogg2anim* tool, which can convert the *.raw* output files into multiple formats, to only convert files to *.ppm* format, and saved it into a new file named *raw2ppm*. Here is the final script:

#!/bin/bash

# needs: dcraw, double, netpbm tools, gstreamer, gifenc.sh

if [ "$4" = "" ]; then echo "format: `basename $0` vname first last fps [d[d]]"; exit; fi

echo "removing old auxiliary files"

rm -f out.\*.raw out.\*.ppm out.\*.ppm.[dDT] out.\*.ppm.d.png

echo "copying /dev/shm/out.????.raw files"

for((f=$2; f<=$3; ++f))

do

# cp /dev/shm/out.$(printf "%04d" $f).raw .

cat hd0.32k /dev/shm/out.$(printf "%04d" $f).raw >out.$(printf "%04d" $f).raw

echo -en "$f \r"

done

echo

echo "dcraw each .raw file (to .ppm)"

for f in out.\*.raw

do

dcraw $f

echo -en "$f \r"

done

echo

After altering the script, it still required the same number of arguments as before, even if I would have only needed 2 out of 4 arguments to run it. So, I ran it by giving it, as input, both useful and dummy arguments. For example:

$ raw2ppm d 0 600 d

The *d* in the first and the fourth argument are dummy arguments, while *0* and *600* describe that the tool should convert files from 0 to 600 to the *.ppm* file format.

The *raw2ppm* tool uses *dcraw*, so I needed to get it installed, using GitHub:

$ sudo git clone <https://github.com/6by9/dcraw>

$ cd dcraw-master

$ chmod 755 buildme

$ ./buildme

One challenge that I had to overcome was that these scripts were not recognized by the Raspbian OS, so I needed to set their path variable:

$ PATH=$PATH:/home/pi/raspiraw-master/tools:/home/pi/raspiraw-master:/home/pi/dcraw-master

Another challenge was setting up the I2C Interface. Despite the fact that I had enabled it previously, the GPIO and the hardware needs to be reset every time a new terminal instance is turned on, or every time *raspistill* or *raspivid* (default apps for accessing the Raspberry Pi V2 Camera) are used, because these scripts, being the default for the Raspberry Pi V2 Camera, overwrite the Camera Interface settings. So, the solution was calling the *camera\_i2c* script every time I wanted to use *raspiraw*, which sets up the interface specifically for this program.

For the filming of the metal ball, I used the *640x64* tool (for 1000 ms), which films in an 640x64 format, at approximately 660 frames per second, outputs the frames that were not skipped (the maximum frame skipping error I have noticed was of 2% of the total frames that were output) and their specific timestamps, in a file called *timestamps.csv*.

I wrote a simple script that calculates the differences between the timestamps of successfully-captured frames and noticed that the frame-capture period was of 1.5 ms (with an error of 0.1 ms).

### Developing the Object-Detection program

The main purpose of the Object-Detection program is to convert the captured frames into HSV image, mask all the pixels that have the Hue value in a specific range, then convert the masked image into a Binary image, apply a threshold (using Otsu’s Binarization Method), calculate the center of the object and output a matrix of timestamps (in seconds) and abscise (in meters).

Here is the full code of the Object-Detection program I have developed and used throughout the experiment, with comments along the code:

#importing the OpenCV(cv2), NumPy(numpy) and glob libraries

import cv2

import numpy as np

import glob

#creating an array out of the output .ppm files and sorting it in ascending order

#for some reason, I couldn't just call glob(something), even if I had imported the glob library

#instead, I call glob.glob(something) and it works

filenames = sorted(glob.glob('out.\*.ppm'))

#using a variable to count the frames that are processed

i = 0

#creating a for loop to cycle through all the files in the filename array

for a in filenames:

#creating a variable to store the information of each file of the array

#the first argument is the file

#the second argument is the storing mode, where '1' means BGR, '0' means GrayScale and '-1' means Unchanged

img = cv2.imread(a, 1)

#creating a variable that stores each pixel of the matrix of the original file (except the last 10 of each line and column)

#but converted to Hue/Saturation/Value (HSV) mode

hsv = cv2.cvtColor(img[:-10,:-10], cv2.COLOR\_BGR2HSV)

#setting lower and upper ranges for the Hue

#the first arguments in the square brackets are variables and should be changed

#according to the color of the object that should be detected in the image files

#the specific values of these arguments can be calculated by opening the image file

#using GIMP or other Image Manipulation Program, selecting the Colour Picker Tool,

#selecting a pixel of the specific object, and inputting the BGR values (in this order)

#into the converter.py program. The output of converter.py should represent the

#square brackets. The Saturation and Value values should be changed according to the

#lighting system that is used in the experimental setup.

#The second argument of the np.array function is the data type. The images' data type is uint8.

#This means that it is an 8 bit integer, because the maximum value for Hue/Saturation/Value is 255.

lower\_range = np.array([52, 50, 75], dtype = np.uint8)

upper\_range = np.array([72, 255, 255], dtype = np.uint8)

#creating a variable that represents the mask of the pixels from the hsv image, that

#have the Hue value between the lower\_range and upper\_range values.

#The program will perform operations only using this mask from now on.

mask = cv2.inRange(hsv, lower\_range, upper\_range)

#converting the HSV image to binary using Otsu's Binarization Method

#firstly, I use the Gaussian Blur to make the image appear smoother

#the first parameter is the source, the second are the distance (amount) of blur on X and Y

#these parameters should be positive and odd

#the third parameter is the standard deviation

blur = cv2.GaussianBlur(mask,(5,5),0)

#secondly, I convert the HSV image (with the blur applied) to binary and apply the threshold

#using Otsu's Binarization Method.

#first parameter is the source image, the second is the treshold value which is used to classify pixel values

#the third is the maxVal which represents the value to be given if pixel value is more than

#(or sometimes less than) the threshold value. The fourth is the used thresholding method.

#This function has two outputs: a retValue and the image with the threshold applied.

ret,thresh = cv2.threshold(blur,0,255,cv2.THRESH\_BINARY+cv2.THRESH\_OTSU)

#Calculating the moments of the thresholded image

M = cv2.moments(thresh)

#Calculating the centre of the object using moments

if M['m00'] != 0:

cX = int(M['m10']/M['m00'])

cY = int(M['m01']/M['m00'])

else:

cX, cY = 0, 0

#Marking the centre of the object with a small black circle

cv2.circle(mask, (cX, cY), 5, (0, 0, 0), -1)

#Displaying the thresholded image, with the centre of the object found

#cv2.imshow('final', mask)

#cv2.waitKey(0)

#cv2.destroyAllWindows()

#Writing the matrix of the timestamp (in seconds) and the abscise of the centre of the object (in meters)

#the 0.012/27 constant is the meters to pixels ratio, in this case. It should be updated for every different object

#used in the experiment.

f = open('centre\_matrix.txt', 'a')

if cY != 0 or cY < 551:

f.write(str(float(i)\*0.0015) + ' ' + str(float(cX)\*0.012/27) + '\n')

#f.write(str(cX) + ',' + str(cY) + '\n')

i = i + 1

f.close()

To determine the exact range of the object’s Hue, I have used another Python program, that converts the BGR values into HSV. Here is the full code:

import sys

import numpy as np

import cv2

#using sys library for command-line arguments

blue = sys.argv[1]

green = sys.argv[2]

red = sys.argv[3]

#declaring the colored pixel in 8-bit format and converting it into HSV format

color = np.uint8([[[blue, green, red]]])

hsv\_color = cv2.cvtColor(color, cv2.COLOR\_BGR2HSV)

#grabbing the HSV values for the specific pixel

hue = hsv\_color[0][0][0]

#output

print("Lower bound is :"),

print("[" + str(hue-10) + ", 100, 100]\n")

print("Upper bound is :"),

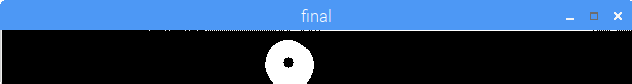
print("[" + str(hue + 10) + ", 255, 255]")

**Results**

Original frame:



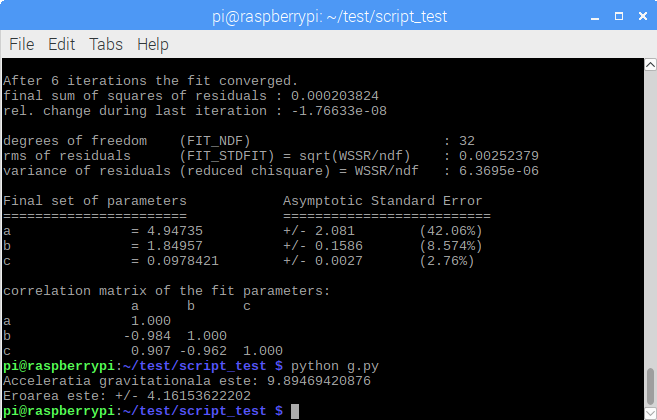
Resulting image:



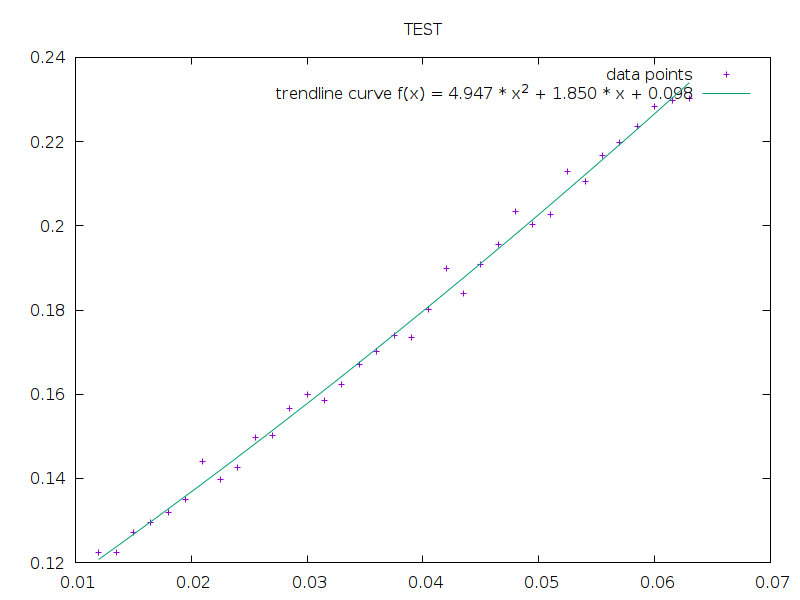
### Plotting, Final Results and Error Sources

I have used *GNU Plot* to plot a Scatter Chart and a trendline, using the input from the output matrix of the Object-Detection program, and that outputs the coefficient of the x2 and its error. I have developed a small Python program that acquires these values and outputs the final result on the terminal screen, which is the measured gravitational acceleration.

The final results look like this:

**

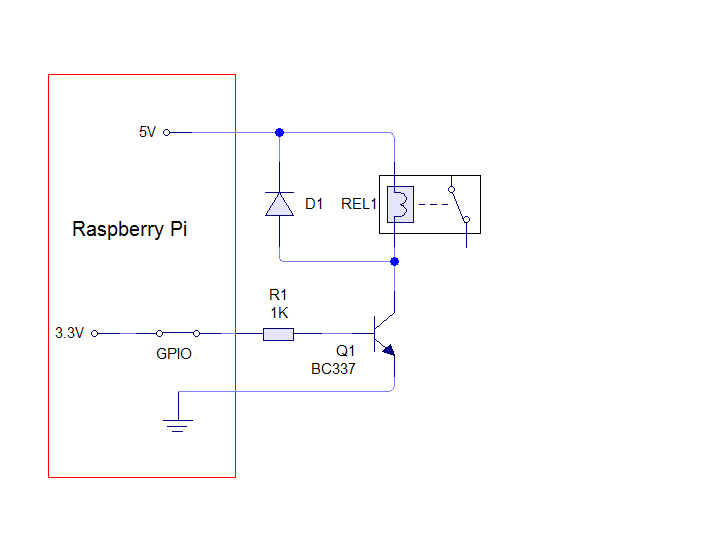
And this is how a plot looks like:



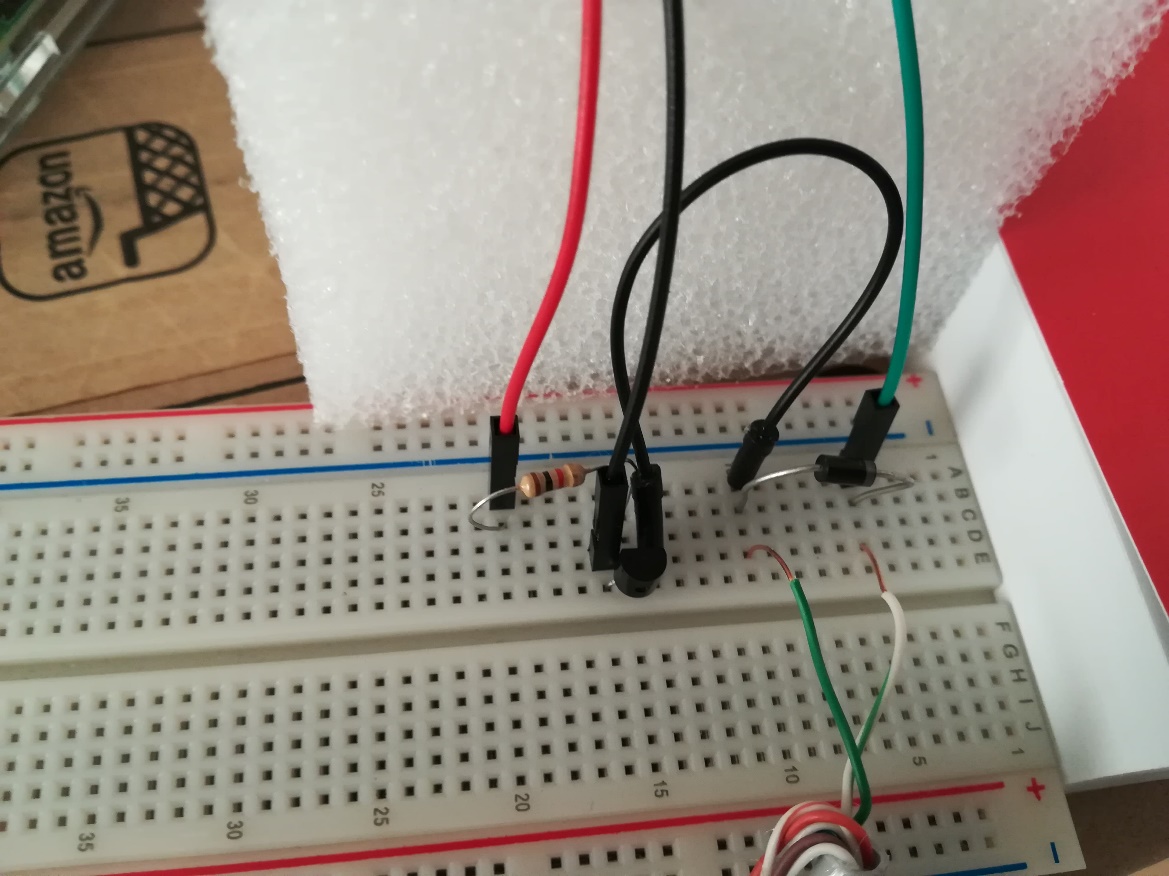
Some error sources might be:

* Lighting – the metal ball reflects some light, so there are some pixels that cannot be detected, thus the program miscalculates the center of the ball.
* The Operating System – Raspbian Stretch is not a Real-Time Operating System, so the processes are not executed at their right time at every iteration or every try of the experiment.
* Raspberry Pi’s CPU Clock – the *tstamps.cvs* file shows that a frame is captured every 1.5 ms, which I have discovered to be false, because the OS cannot synchronize that well all the processes.
* The Ball’s Color – some colors are detected easier than others. For instance, red is detected easier than blue, due to the lighting method I have used.
* Random pixels – the camera captures a thin, multi-colored line of pixels that the program cannot ignore and are automatically considered as part of the object, which leads to the miscalculation of the object’s geometrical center.

### A Relay out of an Ethernet Cable and an Electromagnetic Coil

I have used the following schematic to connect the Relay to the Raspberry Pi’s GPIO

The final result looks like this:



I have developed two scripts to control the current flow from the Raspberry Pi’s GPIO to the Electromagnetic Coil. The first script charges the Electromagnetic Coil with enough current to create an electromagnetic field strong enough to keep the metal ball suspended, while second script just releases it, after the frame capturing script starts.

### Automating the experiment and final setup

The last step was doing all the experiment at the push of a single button. So, what I did was putting together all the scripts and programs that I had previously used in a Bash script. When executing it, it sets the PATH variable, runs the setup for the camera, powers on the Electromagnetic Coil, in order for it to be able to create that magnetic field to sustain the ball, gives the user a 15 seconds window to put the ball in place, starts the frame-capture program and releases the ball, then converts all the frame into the .ppm format, runs the Object-Detection program, plots the Scatter Plot with the Trendline and outputs the final results.

The only thing the user needs to do *manually* is to find out the Hue range for their object. They can do that by capturing some frames with the camera mode they want to use furtherly, find out the BGR values for the object using GIMP’s Color Picker Tool, convert them to HSV using the *converter.py* program and, finally, changing these values in the Object-Detection program. This operation only needs to be done when changing the object the user wants to do the experiment with.

The final experimental setup looks like this, with the lights off: