**The Particle Detector Project**

**Giorgia Mazzini**

**Matilde Mazzini**

**University of Hawaii at Manoa, 2017**

The goal of this project is to build a detector prototype capable of detecting particles. We are using inexpensive, easy to use and simple components, in order to actively involve students from high and middle school.

We used a Raspberry Pi 3 Model B, a small single-board computer developed to promote the teaching of basic computer science in schools. It is very easy to use and very powerful in terms of performances and image processing. But most of all it has an open source Operating System, so it’s possible to find millions of tutorials, examples, projects, questions, advices and pieces of code on the net.

The sensor that we used to detect particles is a webcam. Why?

Every camera has a CMOS sensor (complementary metal-oxide semiconductor) that is sensitive to ionizing particles.

When a cosmic ray enters in the CMOS sensor, it deposits a charge onto the sensor itself and leaves a track. Because of the CMOS's inability to distinguish between photons and other charged particles, we can say that if the sensor is not exposed to any sources of light, it is possible to see the track of cosmic rays in the resulting image.

We covered with black tape and aluminium foil the camera lens, in order to eliminate visible light (photons). This allowed us to observe penetrating cosmic rays such as beta/gamma rays and/or muons.

We choose the Webcam Logitech c270 because compared to other webcams with same resolution (3MP) it has a bigger CMOS sensor (3,6 mm\*2mm),and bigger size of pixels(2.8 µm).

This webcam is very cheap (20 $) and easy to use, so it meets the design specifications.



*This is what you see when a particle hits the sensor of the camera (ZOOM PARTICULAR)*

When you take pictures with the obscured webcam you obtain a completely dark frame.  
When a particle hits the camera sensor, it leaves a track of lighter pixels. To see this we should be able to simulate a sort of “long exposure”, in this way the particle tracks are not erased at every acquisition cycle but accumulated frame by frame.

To take multiple frames and not to lose information when you detect a particle, we decided to sum all the frames we are taking.

Every image is like a matrix of **pixels** (a two dimensional array of pixels, picture element) and each pixel stores the colour of the image at a certain entry of the matrix with a RGB value.   
E.g.: if you have a completely black frame it means that all the RGB values of all the pixels are set to (0,0,0). A white pixel is a pixel with the RGB level set to (255,255,255). When you put black tape on the top of your camera to block photons, you will not have a completely black frame with all the pixel with RGB level (0,0,0), but a very dark grey frame with the RGB levels around (10,11,9) for example.

What our **program** does in an infinite loop:

it takes the first frame, then it applies a threshold(val) function to it and it saves that frame in memory. The camera then takes the next picture, the program applies a threshold(val) function to it and sums it with the previous frame it stored. Therefore, when a particle hits the sensor, we will see a black frame with a spot or a stripe of white pixels.

The program we wrote uses the SimpleCV python library, which is a simple version of a bigger library used for image processing like OpenCV. In this library, there are many useful functions.

This is the part of the code in which we actually simulate the long exposure:

…

tresh\_val=20 #from 0 to 255, it is the threshold value

…

cam = Camera()

…

first = cam.getImage()

first\_thresh = first.threshold(thresh\_val)

while True:

…

tmp = cam.getImage()

tmp\_thresh = tmp.threshold(thresh\_val)

first\_thresh = first\_thresh + tmp\_thresh

first\_thresh.show()

…

blobs = first\_thresh.findBlobs(minsize=1) #set the lower value of the size of the blob

The threshold() function analyses the RGB value of all the pixels of an image. It converts to black (RGB=0,0,0) all the pixels that have the RGB value **lower** than a certain threshold value. If the RGB value is **higher** than this threshold value, the pixel is converted to white: RGB (255, 255, 255). Thanks to that, we can immediately discriminate the black background from the lighter spots that represent the tracks of the particles that hit the sensor.

The threshold value is a number between 0 and 255.

After some tests, we found that 20 is the best balance between high sensitivity and low noise.

Try to change that value and see how the frame changes.

For example if the value set is too low, the picture will be all white.

The sum “+” function does the addition between two images. It means that it sums every pixel’s RGB value of the first image with the RGB value of the corresponding pixel of the second image: like a matrix addition.  
E.g.: If you sum a pixel that has RGB value =(1, 1, 3) to the corresponding pixel of the second frame with RGB value =(3, 4, 5), the resulting RGB value is (4, 5, 8).

We apply that threshold function to every captured frame before doing the addiction, because the black background RGB value is not completely (0, 0, 0), but it is a dark grey colour. If we would not apply the threshold function, we would obtain an undesirable all white picture after few loops. In fact, every time we sum two not completely black frames together, the RGB level increase until it reaches the maximum value 255,255,255(white).

The threshold function is also useful because the track left from a particle in the frame is highlighted converting the lighter pixels (light grey) to a completely white spot.

The findblobs() function analyses the final frame and detects the blobs that are bigger that a certain value. A Blob is an area of connected pixels that share some common property like the colour (in our case the pixels have to be white because we want to count how many white spots there are in the final frame).

At the end, you obtain a completely black frame with all the detected particles represented by white spots.  
The final frame is saved in a jpg format.

**

*THIS IS AN EXAMPLE OF BEFORE AND AFTER THRESHOLD(zoom particular)*

The show() function is used to show in real time every frame that the camera is taking in the SimpleCV displayer, so you can see immediately when a particle hits the sensor. Our program has a framerate of ̴14fps. If you want to have a different framerate, comment the show() function (you lose the real-time functionality) but add this line inside the while: time.sleep(0,01) .

With the time.sleep() function you can decide how many seconds the program should pause between a frame and another, if you do this you modify the framerate.

You have to add this function because the show() itself slows the program: if you remove both the time.sleep() and the show() lines the Raspberry will NOT have enough time to process the frames properly!

In the code we provide we call the show() function and the line with the time.sleep(0.01) function is commented.

When you want to stop the program you can press the button on the breadboard: the program exit the taking frames loop and saves the final image in the current directory with a name format like this: starting time, stopping time, date and number of particles detected.   
After that it attaches to the frame the GPS data that the GPS received modifying the EXIF tags of the picture.

**

*This is one of the final frame we took with a radioactive source of Cobalt 60 on top of the camera sensor.*

**GPS MODULE:**

When you connect the GPS to the Raspberry, it automatically starts looking for satellites information. To read the GPS lines in output we created a variable that listens to the serial port with an infinite loop. Every time the serial port outputs something, the program prints the line and eventually saves that information in an array of values in order to use them later in other ways.

This simple code reads all the GPS lines in real time:

import serial

gps = serial.Serial("/dev/ttyUSB0", baudrate = 9600)

while True:

line = gps.readline()

print line

There are several GPS sentences that you can see in output, the ones in which we are interested in are: $GPRMC and $GPGGA. (in red: the lines we use in our code).

**$GPGGA**Global Positioning System Fix Data.

$GPGGA,194530.000,3051.8007,N,10035.9989,W,1,4,2.18,746.4,M,-22.2,M,,\*6B

0 = sentence identifier

**1 = UTC of Position(time)**

**2 = Latitude**

**3 = N or S**

**4 = Longitude**

**5 = E or W**

**6 = GPS quality indicator (0=invalid; 1=GPS fix; 2=Diff. GPS fix)**

**7 = Number of satellites in use [not those in view]**

8 = Horizontal dilution of position

**9 = Antenna altitude above/below mean sea level (geoid)**

**10 = Meters (Antenna height unit)**

11 = Geoidal separation (Diff. between WGS-84 earth ellipsoid and

mean sea level. -=geoid is below WGS-84 ellipsoid)

12 = Meters (Units of geoidal separation)

13 = Age in seconds since last update from diff. reference station

14 = Diff. reference station ID#

15 = Checksum

## $GPRMC Recommended minimum specific GPS/Transit data.

eg. $GPRMC,220516,A,5133.82,N,00042.24,W,173.8,231.8,130694,004.2,W\*70

1 220516 Time Stamp

2 A validity - A-ok, V-invalid

3 5133.82 current Latitude

4 N North/South

5 00042.24 current Longitude

6 W East/West

7 173.8 Speed in knots

8 231.8 True course

**9 130694 Date Stamp**

10 004.2 Variation

11 W East/West

12 \*70 checksum

If you need to better understand the GPS NMEA sentences here is the link: <http://aprs.gids.nl/nmea/>

**SAVING THE FRAME EVERY 30 MINUTES:**

We will give you two different programs: **particledetectorGM.py** and **particledetectorGM30MINUTES.py**

The difference between these two programs is that the particledetectorGM30MINUTES.py saves the detection image every 30 minutes.

This is useful because even if you are not seeing what is going on in real time, you can check it every 30 minutes looking at the image that the program saves.

Besides, with this, you can collect different GPS location if you are moving around with your experiment.

The program uses a counter that keeps track of the number of acquisition cycles.

Instead of writing 30 minutes, it is necessary to write 25200 cycles.

25200 = 14 cycles per second \*60 seconds per minute \* 30 minutes.

If you want to change the number of minutes, you should change the number of cycles.

What we do then is to show the image, save it, attach the EXIF tag to it and updating the log file. We do this same thing at the end of the program when we press the button to save the final frame at the end.

This is the part of the code we are talking about:

…

cicles = cicles + 1

…

if cicles==72000: #every 30 minutes

first\_thresh.show()

stop = time.strftime('%H:%M:%S%p')

date = time.strftime(' on %b %d %Y')

image\_path="%s-%s%s.jpg" % (start , stop , date)

first\_thresh.save(image\_path)

cont = str(0)

gps\_info = GPSdata(image\_path, ledrosso)

GPIO.output(ledrosso, GPIO.HIGH) # led on

GPIO.output(ledrosso, GPIO.LOW) # led off

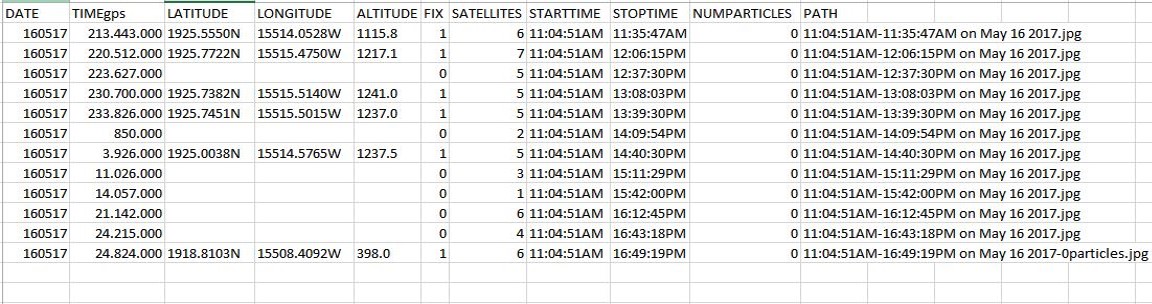
updateLogfile(gps\_info, start, stop, cont, image\_path)

cicles=0

…

**LOG FILE:**

At the end of the program, we call a function that opens and modifies a file (if it is not existing it creates a new one) with the .csv extension. This file can be easily imported in a software like excel to obtain a table of values like this:



**EXIF TAGS:**  
Images can be geotagged. With the software called Exiftool you can modify EXIF tags of a picture and add GPS information to know when and where you took the picture.

For example in the LXTerminal write:

exiftool “imagepath.jpg” to see EXIF data of an image

exiftool –GPSLatitude=”50.860361” “imagepath.jpg” to modify latitude of the image

**GPIO: CONTROL THE LEDS AND THE SWITCH BUTTON**

To control the GPIO you have to import the Rpi.GPIO library.

There are two ways of numbering the IO pins on a Raspberry Pi within RPi.GPIO. The first is using the BOARD numbering system. The second numbering system is the BCM numbers. This is a lower level way of working - it refers to the channel numbers on the Broadcom SOC. Always refer to a graphical Raspberry Pi GPIO Pinout when you work with GPIO pins. (You can find it in the GPIO section of the documentation).

To specify which you are using (mandatory):

**GPIO.setmode(GPIO.BOARD)**

*# or*

**GPIO.setmode(GPIO.BCM)**

You need to set up every channel you are using as an input or an output. To configure a channel as an input:  
**GPIO.setup(channel, GPIO.IN)**

For the button, you have to declare if you are using the pull up or pull down configuration. We use the pull up: when the button is pressed the value of the pin is false:

**GPIO.setup(button,GPIO.IN, pull\_up\_down=GPIO.PUD\_UP)**

To set up a channel as an output:  
**GPIO.setup(channel, GPIO.OUT)**

To read the value of a GPIO pin:  
**GPIO.input(channel)**

To set the output state of a GPIO pin:  
**GPIO.output(channel, state)**

To clean up at the end of your script:  
**GPIO.cleanup()**

This is the part of the code in which the LED starts to blink when the button is pressed:

import RPi.GPIO as GPIO

import time

GPIO.setmode(GPIO.BCM) #we are using the bcm nomenclature

button = 22 #this number is refferred to the bcm number of the gpio

#pin, you can see it in the cobbler

GPIO.setup(button,GPIO.IN, pull\_up\_down=GPIO.PUD\_UP) #we are using the pull up setting

ledverde = 17 #the green led is connected to the pin 17(bcm)

GPIO.setup(ledverde,GPIO.OUT) #the green led is an output

While True:

…

input = GPIO.input(button) #the button is an input, the value of this input is saved in this variable

if (input == False): #if the value of the pin 22 is false:the button is PRESSED

#blink the green led 3 times/ each is half second

for i in range(3)

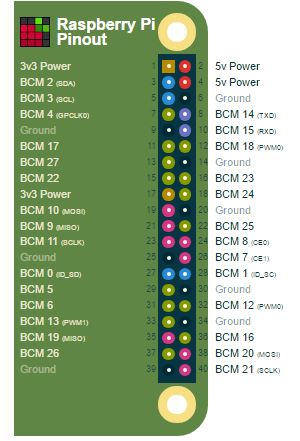
GPIO.output(ledverde, GPIO.LOW) # led of

time.sleep(0.5)

GPIO.output(ledverde, GPIO.HIGH) # led on

time.sleep(0.5)

…



GPIO stands for **General Purpose Input Output**. The GPIO are ports of Raspberry Pi that -through jump wires- let it communicate with the outside world. For instance, it is possible to control electrical components connected with a breadboard to Raspberry Pi.

The name refers to the fact that you can use them for all sorts of purposes; they can both send and receive information. For examples, LEDs receive outputs from a GPIO pin or a switch button sends inputs to our pin. We will see both of the configurations.

Every port has an associated number or name. To deeply understand their meaning you can visit the website <https://pinout.xyz/>

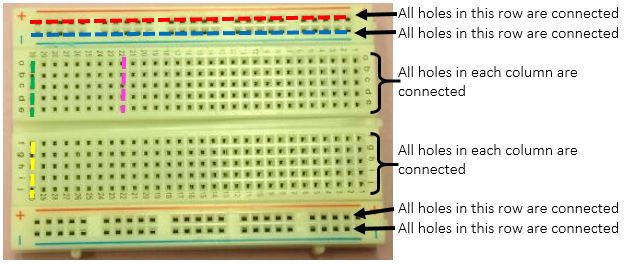
But we are going to use just the numbers that refer to BCM nomenclature.

[image taken from <https://pinout.xyz/> ]

**THE BREADBOARD**

The breadboard is a board that let you connect electronic components without having to solder. It is used to test a circuit design before creating a Printed Circuit Board (PCB).

The holes on the breadboard are connected in a pattern as you can see:

The top row of holes are all connected together – marked with res dots. And so are the second row of holes – marked with blue dots. The same goes for the two rows of holes at the very bottom of the breadboard. In the middle, the columns of wires are connected together with a break in the middle. So, for example, all the green holes marked are connected together, but they are not connected to the yellow holes, nor the pink ones.

You will use the breadboard to set up all components of our Particle Detector.

**THE LED**

A Red LED

LEDs, being diodes, will only allow current to flow in one direction.

So it is very important to connect them in the right way!!

You will notice that one leg of the LED is longer than the other one.

The longer leg (known as the ‘anode’), is always connected to the positive supply of the circuit.

The shorter leg (known as the ‘cathode’) is connected to the negative side of the power supply, known as ‘ground’.

You will connect the anode of the LED up to one of the GPIO pin in order to light it up or down = blinking!!

**THE RESISTOR**

330 Ohm Resistor

Resistor is a component that opposes to the flow of current, we ALWAYS use it in series with a LED because the resistor will ensure that only a small current will flow in the circuit and both Pi and Led will not be damaged.

In the Adafruit Parts Pal the minimum value of resistance we can find is 560 Ω. Actually it is a little too big and that’s why the red LED will be a little bit dim, but we will be sure to not damage anything.

The value of a resistor can be calculated by watching the coloured stripes on top of it. The 560 Ω resistor will be: Green Blue Red Gold

It will not matter which way round you connect the resistors. Current flows in both ways through them.

**CALCULATING EFFICIENCY OF THE SENSOR.**

To do our first experiments we used a source of Cobalt 60. We just put the source in the top of the camera and started measuring with our particle detector.

The source contained 9.730 µCi in 1997. We choose to use an half-life value =5.3 (The real value is almost 5.27…). It means that now (2017) our source has 26 millions beta-decays per second.

Because of the cmos sensor in sensitive in only one direction and because of the geometry of the source we decided to use half the value of beta-decays(13 millions). So 13x10^6x60 seconds= 780millions is the number of beta particles that the source emits in one minute and so that we expect to measure in one minute.

Using our program detection with the source on the top of our camera we detected just 11 particles in one minute (at 30 fps) so the efficiency of our sensor, defined as detected events on expected events is:

ɛ=11/780000000= 1.41\*10^-8=0.0000000141

(Yes, the efficiency is very low, but also the size of our sensor is very small!!)

**MUONS RATE RELATED TO OUR SENSOR.**

We know that at sea level a 1 m2 sensor can detect 100 muons per second. (<http://www2.fisica.unlp.edu.ar/~veiga/experiments.html>)

Our sensor is 7.2 mm2 this means we can detect 0.00072 muons per second.

This is equal to 2.6 muons per hour.