Testplan Ipass

By: Stijn van Wijk, stijn.vanwijk@student.hu.nl

Introduction

This document is a testplan for the libraries and decorators that I have written for my Ipass. I have made two hardware libraries, one for the MAX7219 led driver, and one for the MPU6050 sensor. The third library I have written implements a 2D physic simulator. The decorators are for hwlib windows. This testplan tests all the interface (public) functions of the libraries and decorators.

MAX7219 library

The MAX7219 library implements the MAX7219 as a hwlib window. This means that there are three functions we need to test:

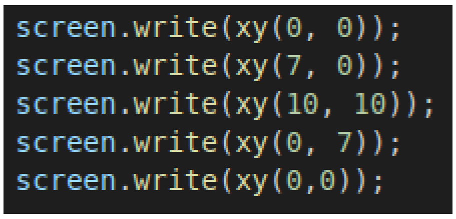
- write(xy pos, color col);
- clear()
- flush()

What behaviour do we expect from the funtions?

- **write(xy pos, color col)**: This function updates the buffer. After a write nothing will be shown on the window. Point (0,0) is in the topleft corner of the screen. The col variable will not change the color of the led. This is because the ledmatrix uses one color leds. If you write outside the window borders, the buffer will NOT be updated. If you write to the same spot nothing will change. The leds will turn on after a flush.
- **flush():** The flush should put all the write commands that are given before the current flush call, and the last flush call on screen. This means if someone did write(xy(0,0)) after flush we will see led on position (0,0) turn on.
- **clear():** This function clears the screen. This means that NO leds are turned on.

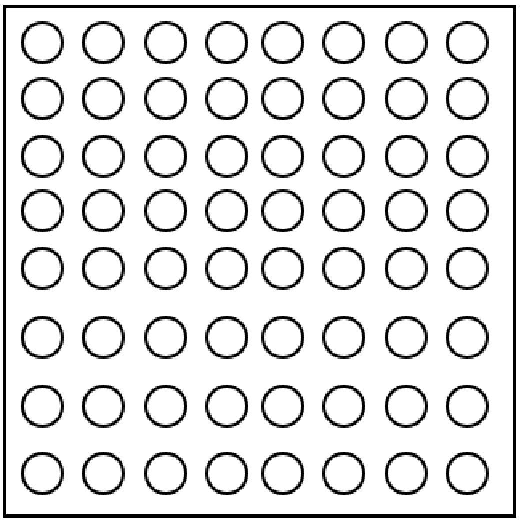
What is the result of the functions?

- **write(xy pos, color col):** I call the write functions multiple time, so we can test all edge cases. The code looks as followed:

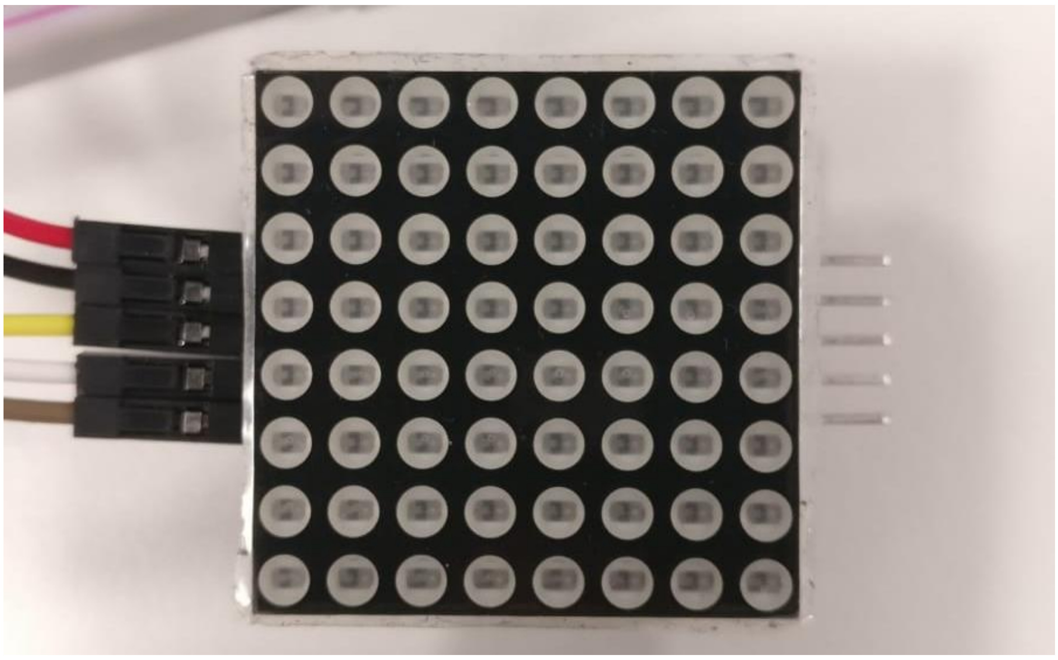


```
screen.write(xy(0, 0));
screen.write(xy(7, 0));
screen.write(xy(10, 10));
screen.write(xy(0, 7));
screen.write(xy(0,0));
```

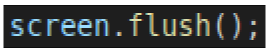
The expected result is:



The result we get:

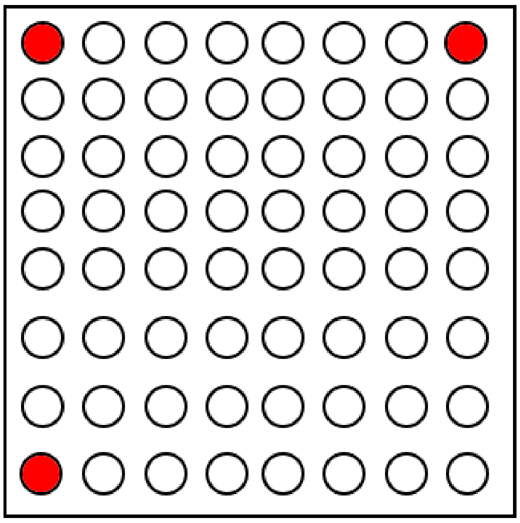


- **flush():** The code looks as followed:

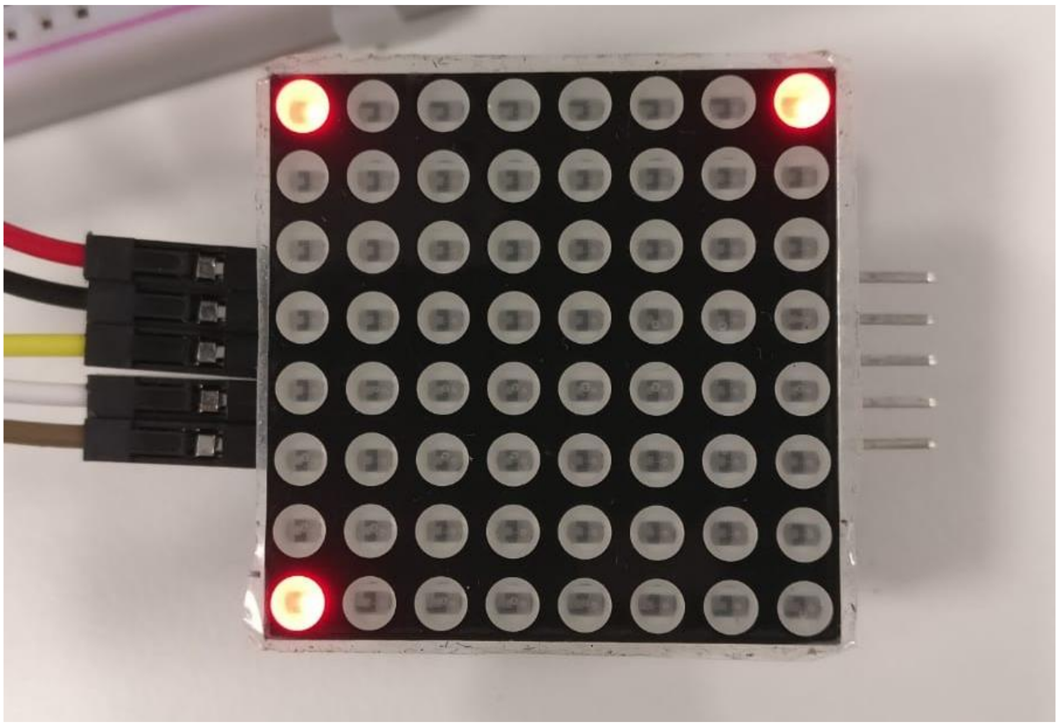


```
screen.flush();
```

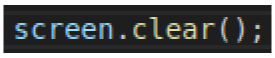
The expected result is:



The result we get:

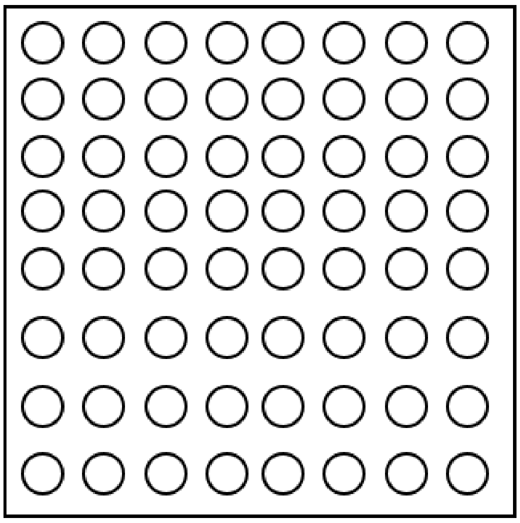


- **clear():** The code looks as followed:

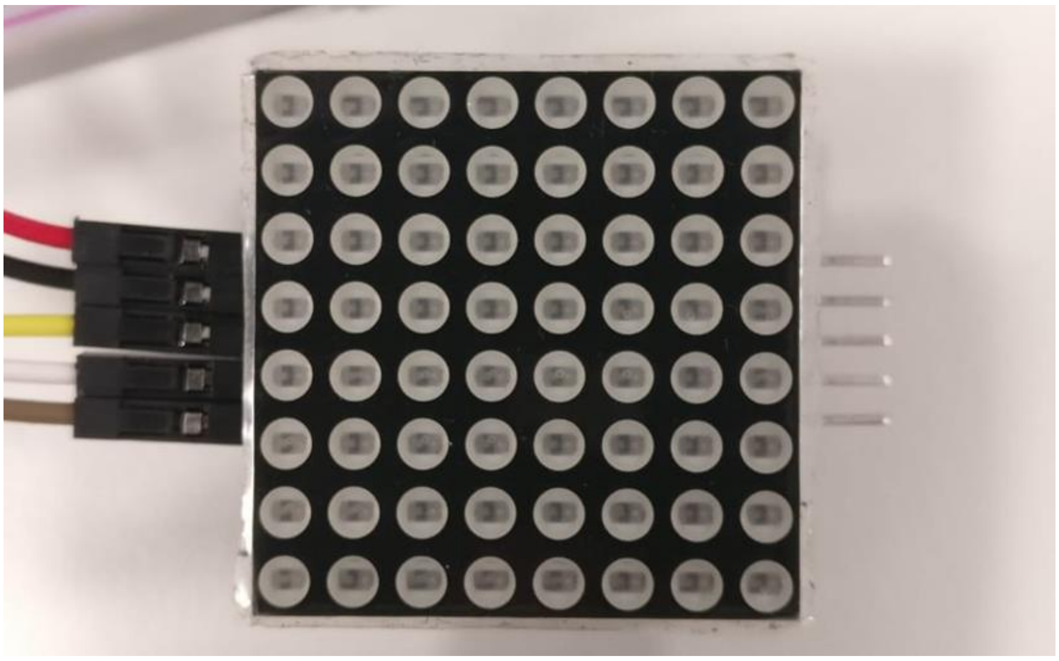


```
screen.clear();
```

The expected result is:



The result we get:



Conclusion

All the results are as expected, this means that the library works correctly. This means all the private functions in the library work correctly, that is because the private functions enables the use of a the public functions.

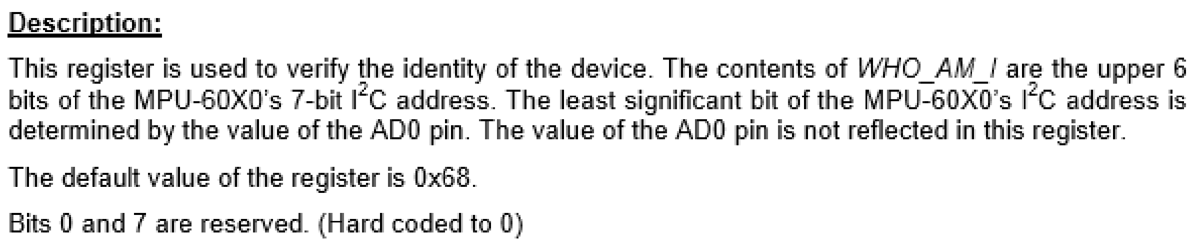
MPU6050 library

The MPU6050 library implements a custom interface. The interface contains of 14 functions. The following 14 functions we need to check are:

- whoAmI()
- calibrate(sample_rate)
- setFullRange(range_acc, range_gyro)
- getAccX()
- getAccY()
- getAccZ()
- getAccAll()
- getGyroX()
- getGyroY()
- getGyroZ()
- getGyroAll()
- getAngleX()
- getAngleY()
- getTemperature()

What behaviour do we expect from the funtions?

- **whoAmI():** This function reads the byte that is found in Register 117. According to the MPU6050 register map the register contains the following:



Description:

This register is used to verify the identity of the device. The contents of *WHO_AM_I* are the upper 6 bits of the MPU-60X0's 7-bit I^{2}C address. The least significant bit of the MPU-60X0's I^{2}C address is determined by the value of the AD0 pin. The value of the AD0 pin is not reflected in this register.

The default value of the register is 0x68.

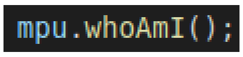
Bits 0 and 7 are reserved. (Hard coded to 0)

- **calibrate(sample_rate):** This function calculates the error/ offset for the acc_x, acc_y, gyro_x, gyro_y and gyro_z sensors. When the sensors is level we expect that the 5 mentioned are all equal to zero. If not, the difference between the value and zero is the offset. It is diffecult to predict the offset, because this sensor dependend.
- **setFullRange(range_acc, range_gyro):** this function changes the full scale range of the accelerometer and gyroscope.
- **getAccX():** This function reads the bytes found in the registers 59 and 60. The resulting two bytes in a signed integer (16-bit). The function should return a value between: -32,768 and 32,767.
- **getAccY():** I will not test this function because it is the same as getAccX(), only the register addresses are different.
- **getAccZ():** I will not test this function because it is the same as getAccX(), only the register address are different.
- **getAccAll():** This function returns a struct with contains three int16_t, all the three values should be between -32,768 and 32,767. We expect that on the Z-axis the number is the biggest.
- **getGyroX():** I will not test this function because it is the same as getAccX(), only the register address are different.

- **getGyroY():** I will not test this function because it is the same as getAccX(), only the register address are different.
- **getGyroZ():** I will not test this function because it is the same as getAccX(), only the register address are different.
- **getGyroAll():** This function returns a struct with contains three int16_t, all the three values should be between -32,768 and 32,767.
- **getAngleX():** This funtion returns a signed integer. This is an angle that should be between -90 and 90 degrees. The angle should change when the sensor is rotated around its x-axis. If this function returns a expected value, we can say that the getAccX-Y-Z functions work correctly. This is because this function uses the data from those three functions to calculate the angle.
- **getAngleY():** This funtion returns a signed integer. This is an angle that should be between -90 and 90 degrees. The angle should change when the sensor is rotated around its y-axis. If this function returns a expected value, we can say that the getAccX-Y-Z functions work correctly. This is because this function uses the data from those three functions to calculate the angle.
- **getTemperature():** This function returns the temperature measured by the MPU6050. This is in celcius. The temperature should be around the same temperature as the room the sensor is in.

What is the result of the functions?

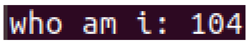
- **whoAmI():** The function prints the byte read from the register. The code looks as followed:



```
mpu.whoAmI();
```

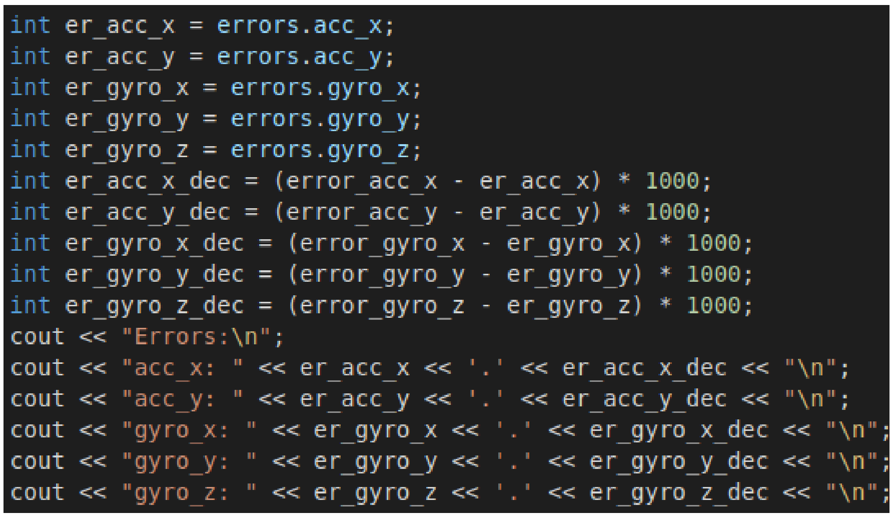
The result we expect is to see is: “who am i: 104”, when AD0 is not connected, and when AD0 is connected to Vcc: “who am i: 105”.

The result we get:

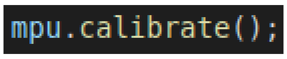


```
who am i: 104
```

- **calibrate(sample_rate):** The function does not output when in use, but for testing I added a hwlib::cout to get the error_values on the terminal. The code looks as followed:



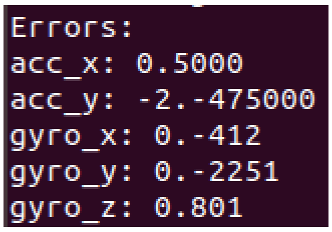
```
int er_acc_x = errors.acc_x;
int er_acc_y = errors.acc_y;
int er_gyro_x = errors.gyro_x;
int er_gyro_y = errors.gyro_y;
int er_gyro_z = errors.gyro_z;
int er_acc_x_dec = (error_acc_x - er_acc_x) * 1000;
int er_acc_y_dec = (error_acc_y - er_acc_y) * 1000;
int er_gyro_x_dec = (error_gyro_x - er_gyro_x) * 1000;
int er_gyro_y_dec = (error_gyro_y - er_gyro_y) * 1000;
int er_gyro_z_dec = (error_gyro_z - er_gyro_z) * 1000;
cout << "Errors:\n";
cout << "acc_x: " << er_acc_x << '.' << er_acc_x_dec << "\n";
cout << "acc_y: " << er_acc_y << '.' << er_acc_y_dec << "\n";
cout << "gyro_x: " << er_gyro_x << '.' << er_gyro_x_dec << "\n";
cout << "gyro_y: " << er_gyro_y << '.' << er_gyro_y_dec << "\n";
cout << "gyro_z: " << er_gyro_z << '.' << er_gyro_z_dec << "\n";
```



```
mpu.calibrate();
```

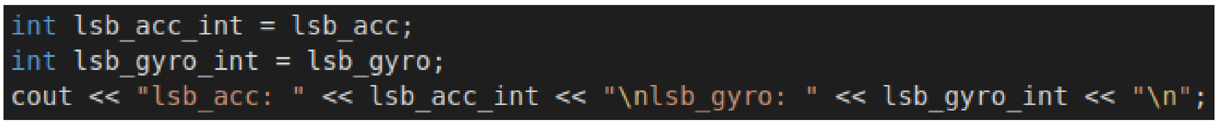
The exact result we expect is unknown. But the errors should be around the 0.

The result we get:

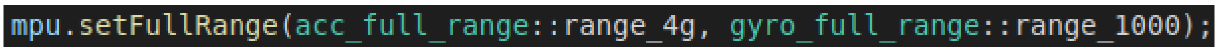


```
Errors:
acc_x: 0.5000
acc_y: -2.-475000
gyro_x: 0.-412
gyro_y: 0.-2251
gyro_z: 0.801
```

- **setFullRange(range_acc, range_gyro):** This function changes the lsb_acc and lsb_gyro, I added a cout to see the current values of lsb_acc and lsb_gyro. The code is as followed:



```
int lsb_acc_int = lsb_acc;
int lsb_gyro_int = lsb_gyro;
cout << "lsb_acc: " << lsb_acc_int << "\nlsb_gyro: " << lsb_gyro_int << "\n";
```



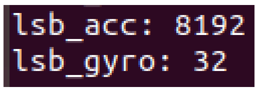
```
mpu.setFullRange(acc_full_range::range_4g, gyro_full_range::range_1000);
```

The expected result is:

lsb_acc: 8192

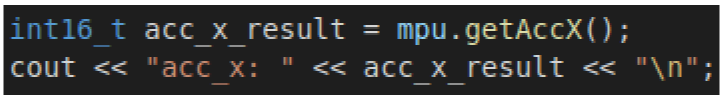
lsb_gyro: 32

The result we get:



```
lsb_acc: 8192
lsb_gyro: 32
```

- **getAccX():** The code is as followed:

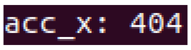


```
int16_t acc_x_result = mpu.getAccX();
cout << "acc_x: " << acc_x_result << "\n";
```

The expected result is:

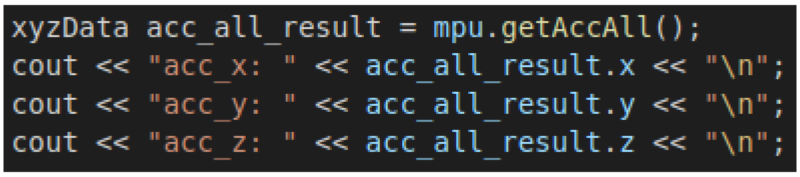
acc_x: -32,768 > ? < 32,767

The result we get:



```
acc_x: 404
```

- **getAccAll():** The code is as followed:



```
xyzData acc_all_result = mpu.getAccAll();
cout << "acc_x: " << acc_all_result.x << "\n";
cout << "acc_y: " << acc_all_result.y << "\n";
cout << "acc_z: " << acc_all_result.z << "\n";
```

The expected result is:

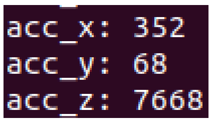
acc_x: -32,768 > ? < 32,767

acc_y: -32,768 > ? < 32,767

acc_z: -32,768 > ? < 32,767

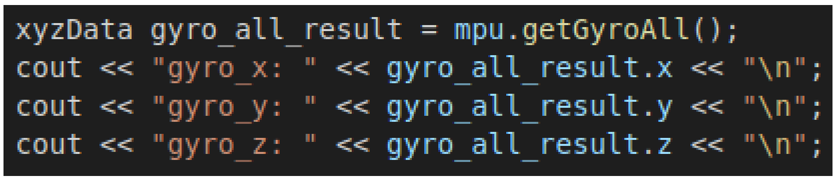
The z-axis should contain the biggest value because that is the axis opposite of the gravity direction.

The result we get:



```
acc_x: 352
acc_y: 68
acc_z: 7668
```

- **getGyroAll():** The code is as followed:



```
xyzData gyro_all_result = mpu.getGyroAll();
cout << "gyro_x: " << gyro_all_result.x << "\n";
cout << "gyro_y: " << gyro_all_result.y << "\n";
cout << "gyro_z: " << gyro_all_result.z << "\n";
```

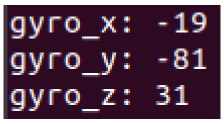
The expected result is:

gyro_x: -32,768 > ? < 32,767, if the mpu doesn't move the number should be 0.

gyro_y: -32,768 > ? < 32,767, if the mpu doesn't move the number should be 0.

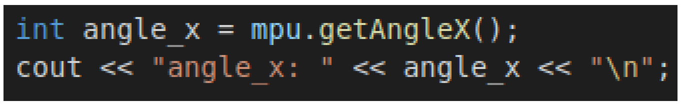
gyro_z: -32,768 > ? < 32,767, if the mpu doesn't move the number should be 0.

The result we get:



```
gyro_x: -19
gyro_y: -81
gyro_z: 31
```

- **getAngleX():** The code is as followed:

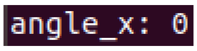


```
int angle_x = mpu.getAngleX();
cout << "angle_x: " << angle_x << "\n";
```

The expected result is:

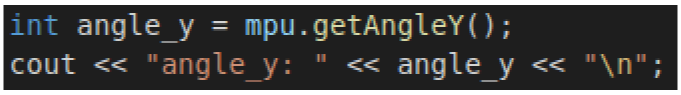
angle_x: -90 > ? < 90, with te expectation if the mpu6050 is level that the angle would be 0.

The result we get:



```
angle_x: 0
```

- **getAngleY():** The code is as followed:

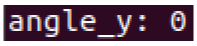


```
int angle_y = mpu.getAngleY();
cout << "angle_y: " << angle_y << "\n";
```

The expected result is:

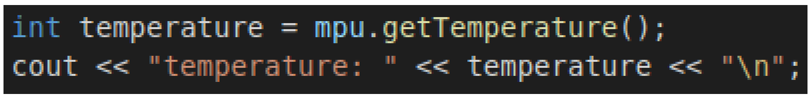
angle_y: -90 > ? < 90, with te expectation if the mpu6050 is level that the angle would be 0.

The result we get:



```
angle_y: 0
```

- **getTemperature():** The code is as followed:

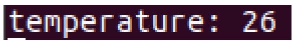


```
int temperature = mpu.getTemperature();
cout << "temperature: " << temperature << "\n";
```

The expected result is:

This test is taken on my room. This is a room in the attic. So the temperature will be around 25 degrees celcius.

The result we get:



```
temperature: 26
```

Conclusion

The results are about the same as we expected. I can't make an exact guess of what the rawData should look like from the gyro. But thanks to the getAngleX-Y, and a good result from those functions we can say that we read out the registers correctly. This means that all the private/ supporting functions work as expected.

Window decorators

I have made four different window decorators, it are the following decorators:

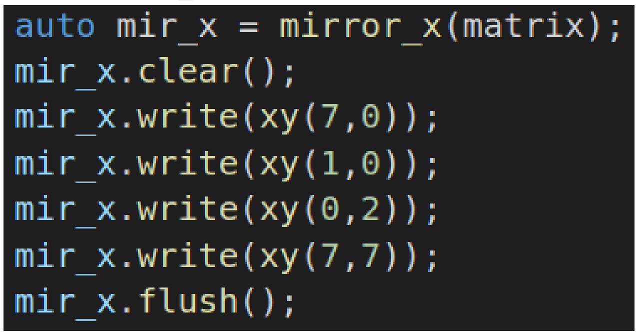
- mirror_x
- mirror_y
- duplicate
- combine_windows

What behaviour do we expect from the decorators?

- **mirror_x:** This decorator should flip the pos on the x-axis. The y-axis stays the same.
- **mirror_y:** This decorator should flip the pos on the y-axis. The x-axis stays the same.
- **duplicate:** This decorator flushes the write command to two screens add the same time. We expect to see the same on both windows.
- **combine_windows**: This decorator combines multiple windows to one window. The created window will translate the given pos in write to the correct window and pos.

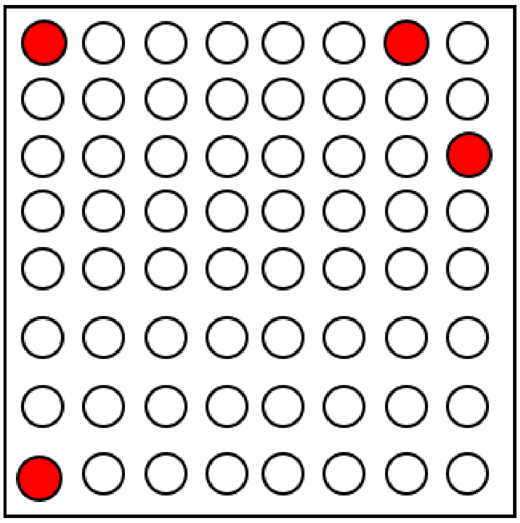
What is the result of the decorators?

- **mirror_x:** The code looks as followed:

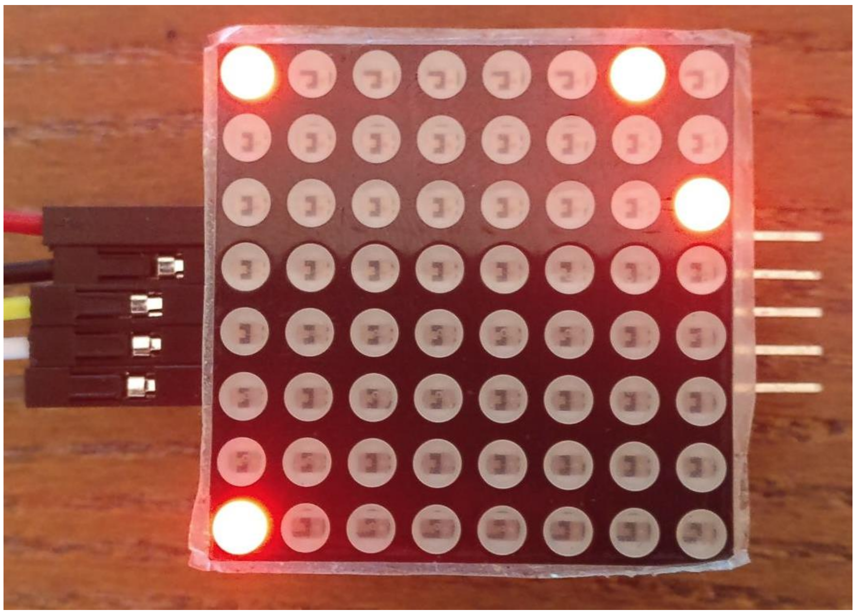


```
auto mir_x = mirror_x(matrix);
mir_x.clear();
mir_x.write(xy(7,0));
mir_x.write(xy(1,0));
mir_x.write(xy(0,2));
mir_x.write(xy(7,7));
mir_x.flush();
```

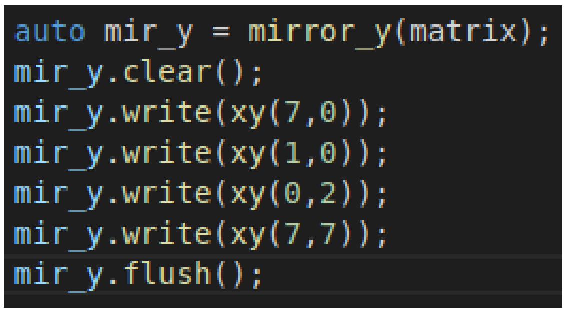
The expected result is:



The result we get:

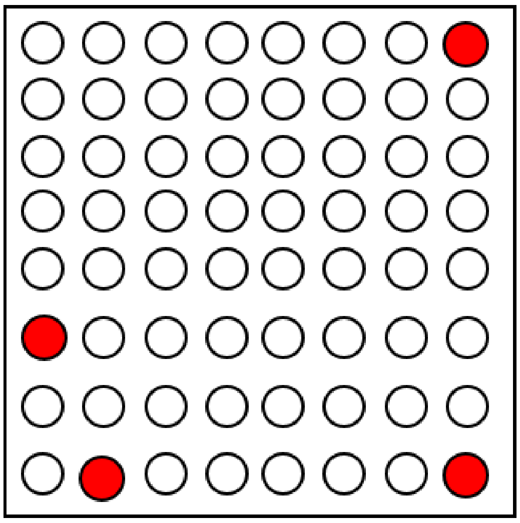


- **mirror_y:** The code looks as followed:

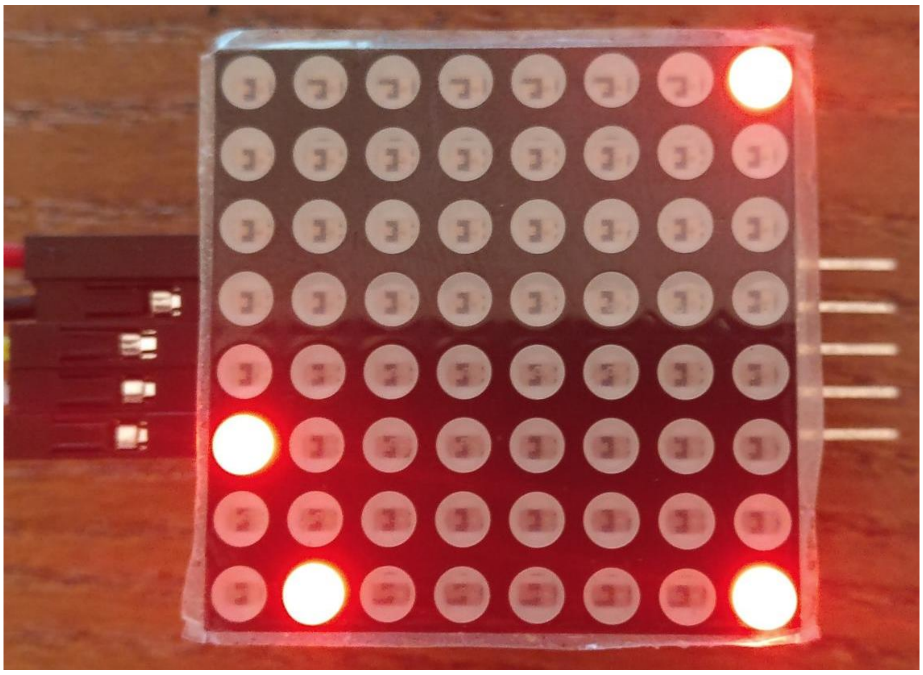


```
auto mir_y = mirror_y(matrix);
mir_y.clear();
mir_y.write(xy(7,0));
mir_y.write(xy(1,0));
mir_y.write(xy(0,2));
mir_y.write(xy(7,7));
mir_y.flush();
```

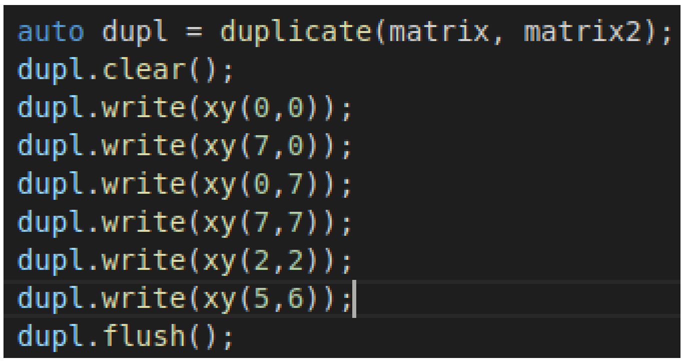
The expected result is:



The result we get:

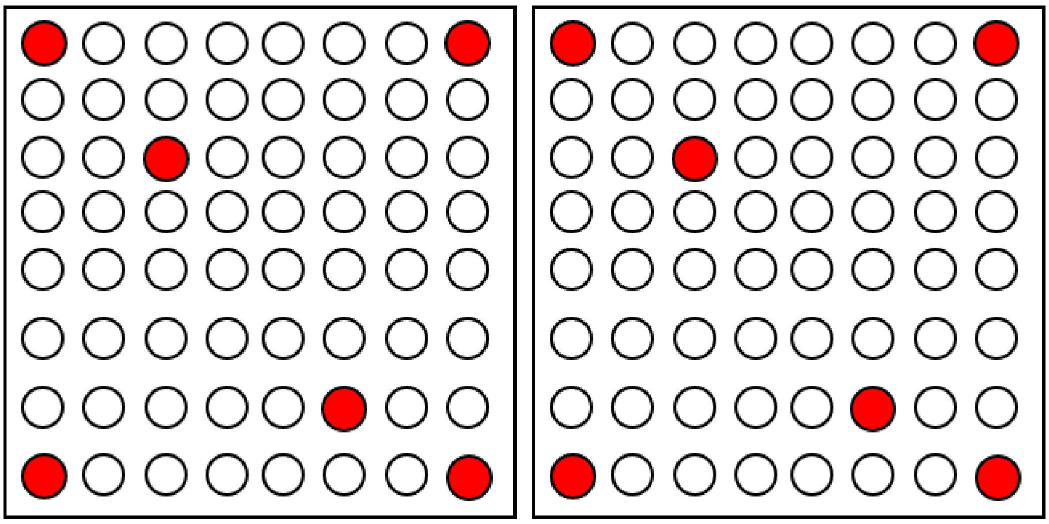


- **duplicate:** The code looks as followed:

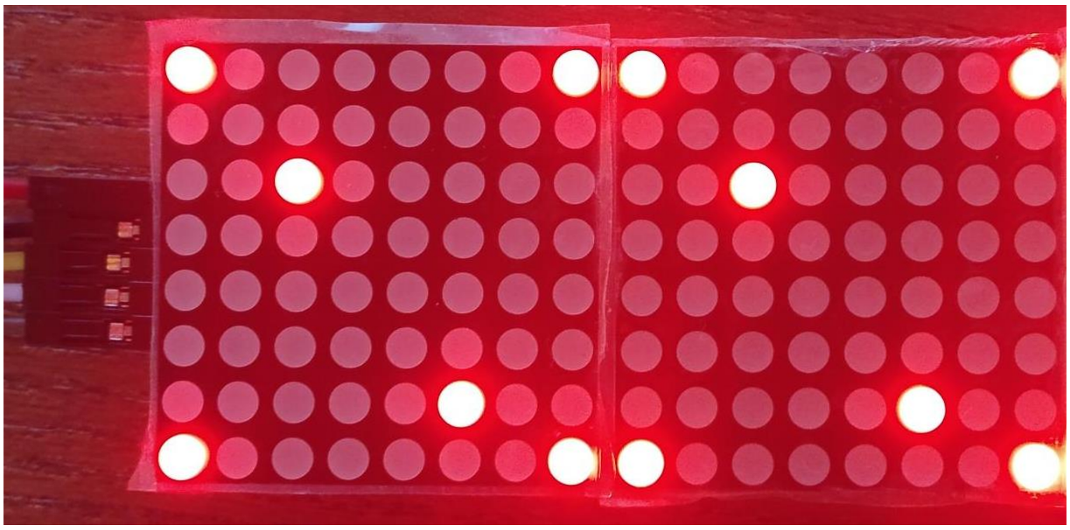


```
auto dupl = duplicate(matrix, matrix2);
dupl.clear();
dupl.write(xy(0,0));
dupl.write(xy(7,0));
dupl.write(xy(0,7));
dupl.write(xy(7,7));
dupl.write(xy(2,2));
dupl.write(xy(5,6));
dupl.flush();
```

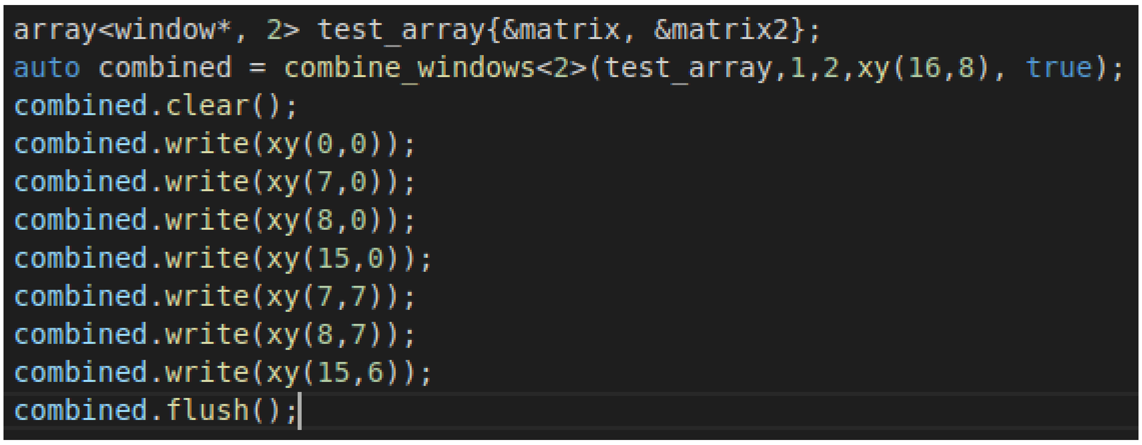
The result we expect is:



The result we get:

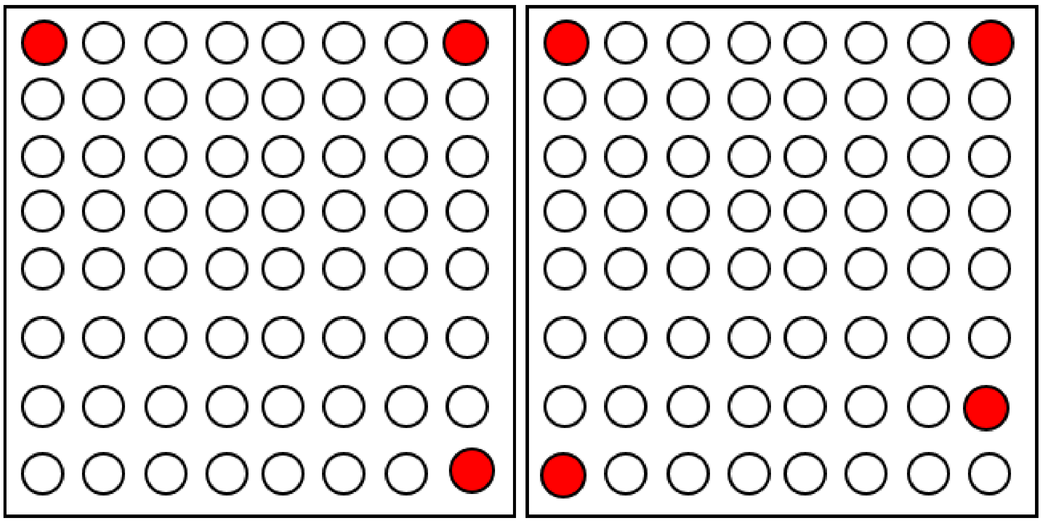


- **combine_windows:** The code is as followed:

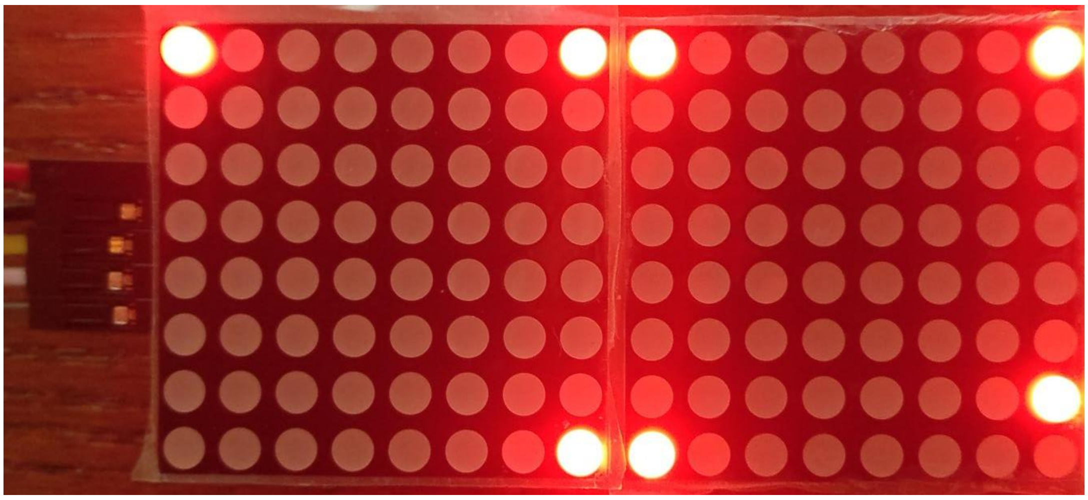


```
array<window*, 2> test_array{&matrix, &matrix2};
auto combined = combine_windows<2>(test_array,1,2,xy(16,8), true);
combined.clear();
combined.write(xy(0,0));
combined.write(xy(7,0));
combined.write(xy(8,0));
combined.write(xy(15,0));
combined.write(xy(7,7));
combined.write(xy(8,7));
combined.write(xy(15,6));
combined.flush();
```

Te reesult we expect is:



The result we get:



Conclusion

All the decorators show the behaviour we expect from them. This means that the implementation of the decorators is correct.

Physics simulator

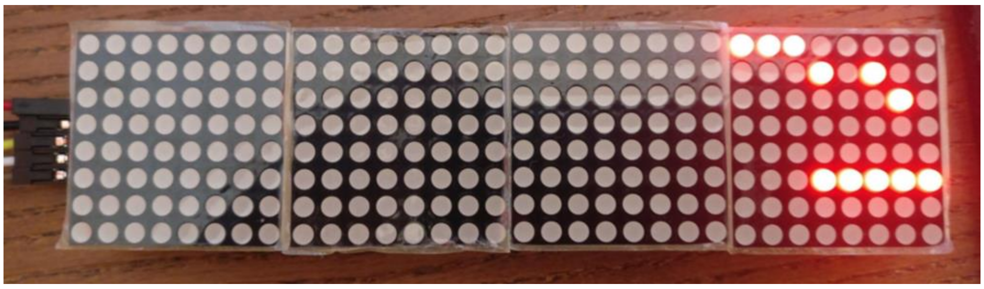
The simulator is made of two classes, a particle class and a simulation class. The easiest way to test all the functionality of both libraries, is by running the simulationloop.

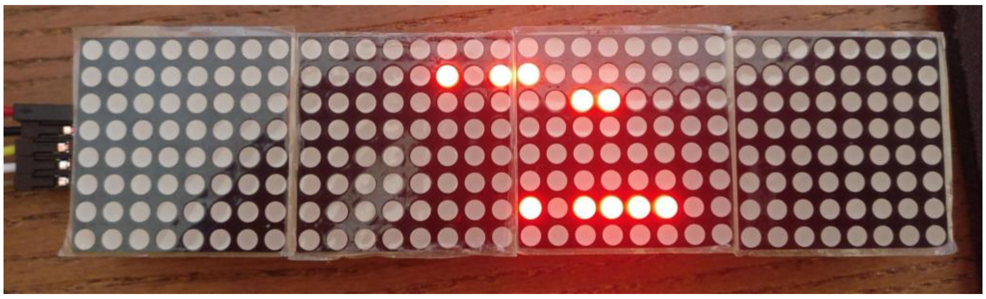
What behaviour do we expect from the functions?

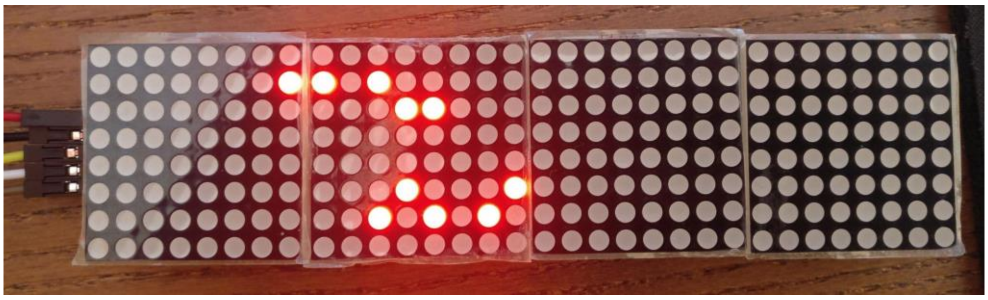
The behaviour we expect is that the particles react to the angle of the mpu6050 sensor. When the sensor is tilted to the left, the particles go left. This is exactly the same for every direction.

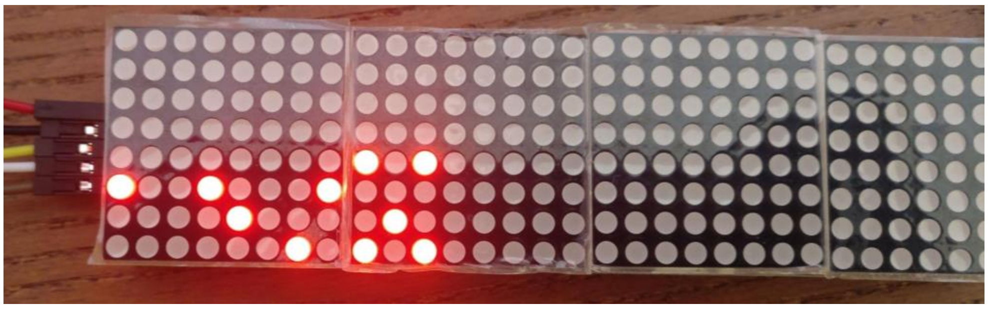
What is the result of the functions?

The best way to show the result is in a video, this is not possible so I took some photos of the state of the simulation.









Conclusion

The simulation feels responsive. It reacts correctly on the input of the user. The simulation works as expected.