Project documentation in subject Design Laboratory

Designers: Wojciech Nodzyński, Witold Klepek

1. **Topic:**

   ‘’Collision reaction system using Zumo Robot in order to fight against another robot”

2. **Assumption:**

   The project was aimed to prepare a suitable collision reaction system depended on place, where the event was detected, which allows to use optimal fight tactics. A robot flees from its opponent or tries to fight it.

3. **Description of code segments:**

3.1 Including libraries and variables initialization

This segment includes libraries needed, among others, to handle Zumo Robot, to initialize objects responsible for motors, buzzer and button. It contains variables such as *Threshold*, which is used in the collision detection, input data to define robot’s speed or time parameters.

```
#include<ZumoShield.h>
#include<Wire.h>
ZumoBuzzer buzzer;
ZumoMotors motors;
Pushbutton button(ZUMO_BUTTON);

int XY_ACCELERATION_THRESHOLD = 16000;
const int motorSpeed = 120;
int RA_SIZE = 2;
unsigned long loop_start_time;
unsigned long contact_made_time;
int MIN_DELAY_BETWEEN_CONTACTS = 3000;
```

3.2 Class template *RunningAverage*

Included to manage a mean of last accelerometer measurements in time. Its purpose is to minimalize the influence of undesirable read-out, which could call the collision detection in a wrong moment.

```
template <typename T>
class RunningAverage
{
  public:
    RunningAverage(void);
    RunningAverage(int);
    ~RunningAverage();
    void clear();
    void addValue(T);
    T getAverage() const;
    void fillValue(T, int);
  protected:
    int _size;
    int _cnt;
    int _idx;
    T _sum;
    T * _ar;
    static T zero;
};
```

```
T RunningAverage<T>::getAverage() const
{
  if (_cnt == 0) return zero; // NaN ?  math.h
  return _sum / _cnt;
}
```

3.3 Accelerometer class and its functions

Task of this segment is to create a comfortable way of reading values of accelerometer's axis X and Y or other parameters such as angle of the resultant vector.

```
class Accelerometer : public ZumoIMU
{
  typedef struct acc_data_xy
  {
    unsigned long timestamp;
    int x;
    int y;
    float dir;
  } acc_data_xy;

  public:
    Accelerometer() : ra_x(RA_SIZE), ra_y(RA_SIZE) {};
    ~Accelerometer() {};
    void getLogHeader(void);
    void readAcceleration(unsigned long timestamp);
    float len_xy() const;
    float dir_xy() const;
    int x_avg(void) const;
    int y_avg(void) const;
    long ss_xy_avg(void) const;
    float dir_xy_avg(void) const;
  private:
    acc_data_xy last;
    RunningAverage<int> ra_x;
    RunningAverage<int> ra_y;
};
Accelerometer acc;
```

```
float Accelerometer::dir_xy_avg(void) const
{
  return atan2(static_cast<float>(x_avg()), static_cast<float>(y_avg())) * 180.0 / M_PI;
}
```

3.4 Functions initializing Robot's start

The function *CountDown()* is responsible for turning on the program after the button is pressed, this action is signalized by buzzer's sound. The function then resets the *contact_made_time* variable.

The function *setup()* initializes I2C magistral, configurates the communication with the accelerometer and calls the function *CountDown().*

```
void CountDown()
{
  button.waitForButton();
  for(int i =0; i<3; i++)
  {
    delay(500);
    buzzer.playNote(NOTE_G(3), 250, 10);
    delay(500);
    buzzer.playNote(NOTE_G(4), 250, 10);
  }
  delay(1000);
  buzzer.playNote(NOTE_G(5), 600, 15);
  delay(500);

  contact_made_time = 0;

}
```

```
void setup()
{
   // Initialize the Wire library and join the I2C bus as a master
  Wire.begin();

  // Initialize accelerometer
  acc.init();
  acc.enableDefault();

#ifdef LOG_SERIAL
  Serial.begin(9600);
  acc.getLogHeader();
#endif
  randomSeed((unsigned int) millis());
  CountDown();

}
```

3.5 Function *loop()* – continuous operation

The function *loop()* provides that the program will work continuously after pressing the button – if the collision is detected, the program will carry out further instructions to handle it. However, it is moving with defined speed which is defaulted. Moreover, the function decrease *Threshold* after 3 sec of the last collision. It prevents calling the collision handler, when the robot starts its movement.

```
void loop()
{
  if(button.isPressed())
  {
    motors.setSpeeds(0, 0);
    button.waitForRelease();
    CountDown();
  }


  loop_start_time = millis();
  if(loop_start_time>contact_made_time+3000)
    XY_ACCELERATION_THRESHOLD = 2400;
  acc.readAcceleration(loop_start_time);

  if(check_collision())
    {
      contact_detected();
    }
  else
  {
  motors.setSpeeds(motorSpeed, motorSpeed);
  }
}
```

3.6 Collision detection

Two functions were used to handle this segment. *Check_collison()* returns bool variable, which informs, if any event occured. It is depended on a measured value of the sum of squares of the acceleration X and Y axis, ($accelerationX^2 + accelerationY^2$) compared to $Threshold^2$.

The function also checks if the collision was detected in certain time period after the last detection.

Void *contact_detected()* process updates the *Threshold* and *contact_made_time* variables (in order to eliminate the collision detection when the robot starts moving again). Moreover, the function calls *contact_response(),* which contains a proper description of behaviours after the collision.

```
bool check_collision()
{
  static long threshold_squared = (long) XY_ACCELERATION_THRESHOLD * (long) XY_ACCELERATION_THRESHOLD;
   return(acc.ss_xy_avg() >  threshold_squared)&& \
    (loop_start_time - contact_made_time > MIN_DELAY_BETWEEN_CONTACTS);

}
void contact_detected()
{

  contact_made_time = loop_start_time;
  contact_response();
  XY_ACCELERATION_THRESHOLD = 16000;

}
```

3.7 Collision handler

This segment is made of:

- *contact_detected_where(),* which returns a number between 1 and 8, depending on which of 8 defined locations the event occurred;
- *contact_response(),* which reads a returned value from the previous function and contains instruction to every case.

Below, in the second picture, there is an example of such instruction. The robot tries to rotate first and then after 200ms begins to move forward to flee.

```
int contact_detected_where()
{
  if(acc.dir_xy()>=-110.0 && acc.dir_xy()<=-70.0)
    return 1;   //front
  else if(acc.dir_xy() > -70.0 && acc.dir_xy()< -20.0)
    return 2;    //front, right
  else if(acc.dir_xy() >= -20.0 && acc.dir_xy()<= 20.0)
    return 3;    //right
  else if(acc.dir_xy() > 20.0 && acc.dir_xy()< 70.0)
    return 4;      //back, right
  else if(acc.dir_xy() >= 70.0 && acc.dir_xy()<= 110.0)
    return 5;      //back
  else if(acc.dir_xy() > 110.0 && acc.dir_xy()< 160.0)
    return 6;      //back, left
  else if(acc.dir_xy() > 160.0 || acc.dir_xy()< -160.0)
    return 7;      //left
  else
    return 8;    //front, left
}
```

```
case 4:  //back, right
  motors.setSpeeds(motorSpeed*1.5 ,0);
  buzzer.playNote(NOTE_C(4), 400, 20);
  delay(200);
  motors.setSpeeds(motorSpeed, motorSpeed);
  buzzer.playNote(NOTE_G(4), 400, 20);
  delay(800);
  break;
```

4. **Sources:**

https://playground.arduino.cc/Main/RunningAverage

https://github.com/pololu/zumo-shield-arduino-library

https://www.pololu.com/docs/0j57/all