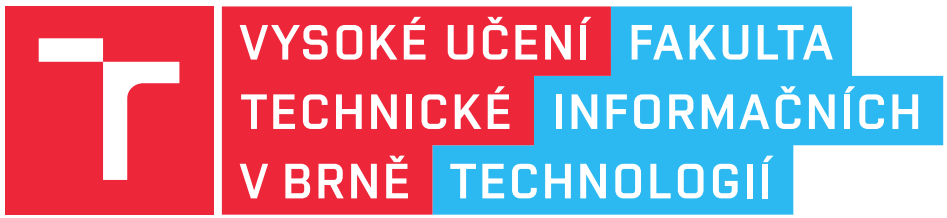
Educational tutorial for Arduino and Trilobot

Jan Beran

Faculty of information technology Brno University of Technology
Božetěchova 1/2. 612 66 Brno – Královo Pole
xberan43@stud.fit.vutbr.cz



May 25, 2020

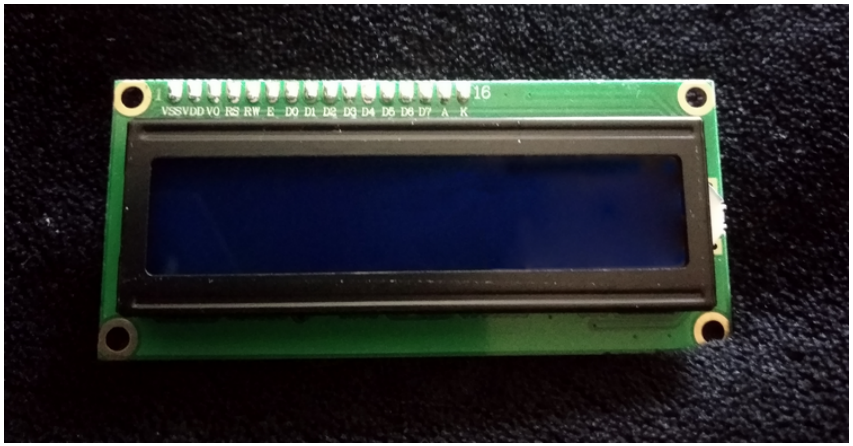
Overview

Introduction

- Originally: robot for academical purposes
- Rebuilt several–times
- Original look in the picture
- Current version based on Arduino is in front of you
- It is fully–assembled and there is no need to connect anything physically
- Fun fact: it used to ride in the opposite direction :)

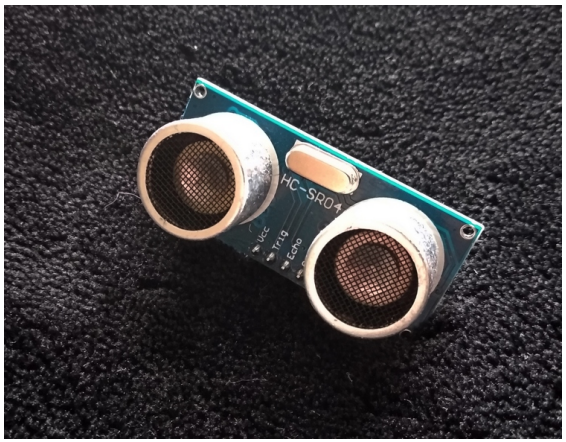
Trilobot components

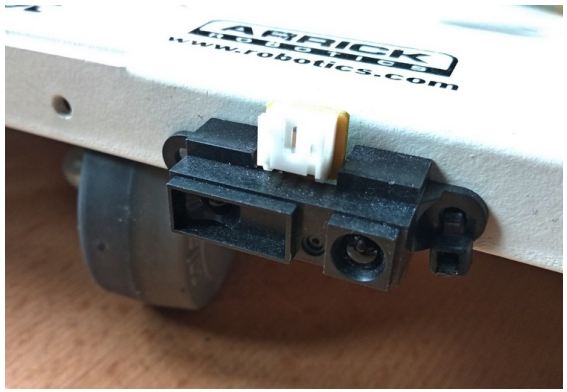
- Standard 16x2 LCD display connected via I^2C
- Can be controlled via `display` object or via `LiquidCrystalI2C.h` library



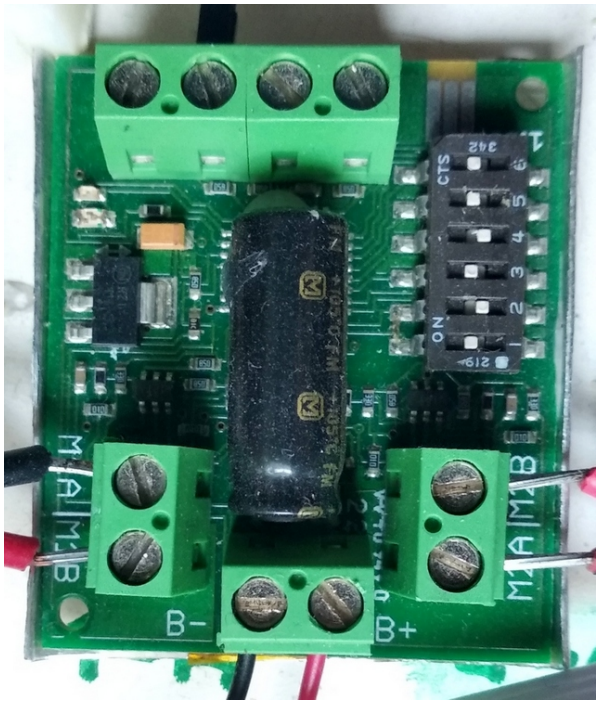
Distance sensors

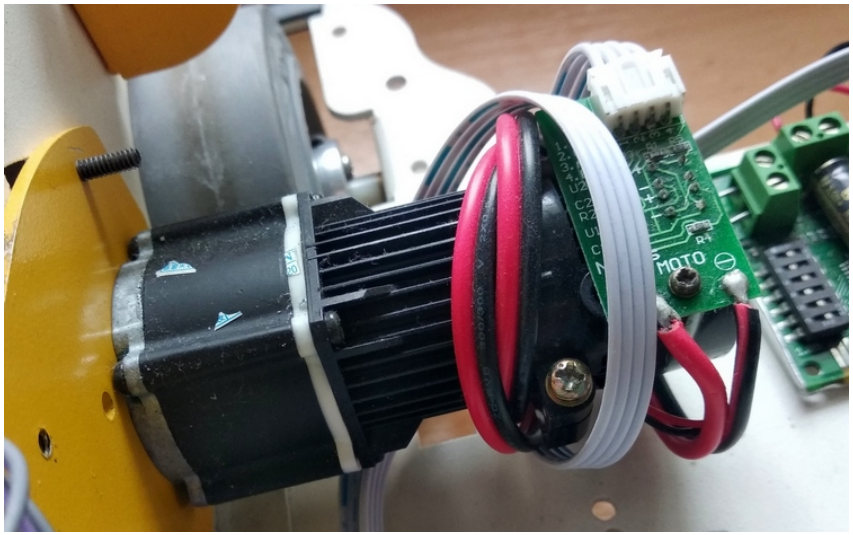
- Ultrasound HC–SR04 a SRF–08, infrared Sharp
- Every sensor fits on something else
- They can be controlled via `hcsr04`, `srf08` and `sharp`





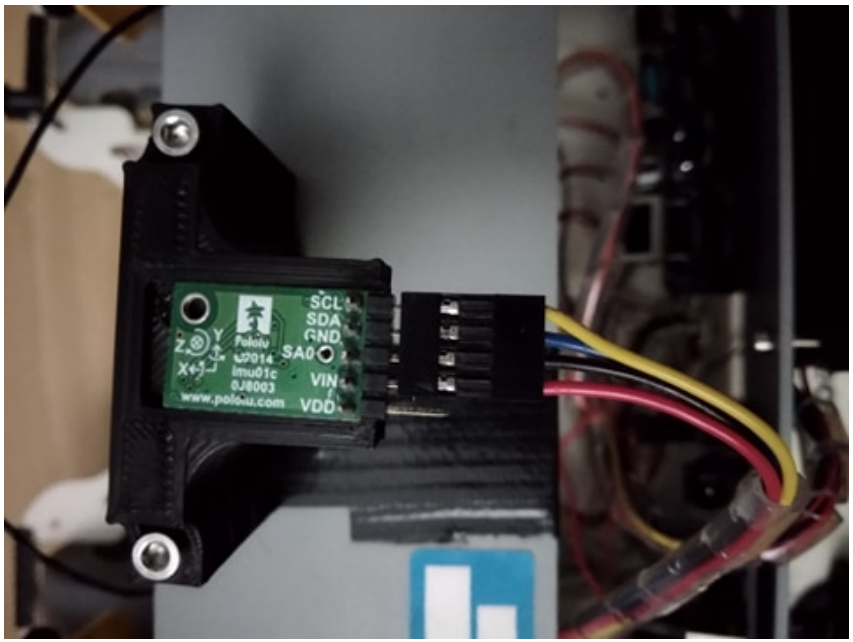
- Two Faulhaber motors controlled by the Sabertooth board are used to drive
- Communication between Arduino and Sabertooth takes place via Serial line `Serial1`
- Feedback is realized via two rotary encoders
- Motors are controlled via `motors` object in the code





IMU – gyroscope, accelerometer, magnetometer

- 3–in–1 sensor pack:
  - Gyroscope for angular velocity
  - Accelerometer for acceleration
  - Magnetometer for measuring of magnetic field intensity
- The data from IMU serves for positioning via AHRS (Attitude and heading reference system)
- IMU has to be above Trilobot to prevent magnetometer from interfering with Trilobot metal body
- There are three object to control the sensors: `gyro`, `accel` and `mag`



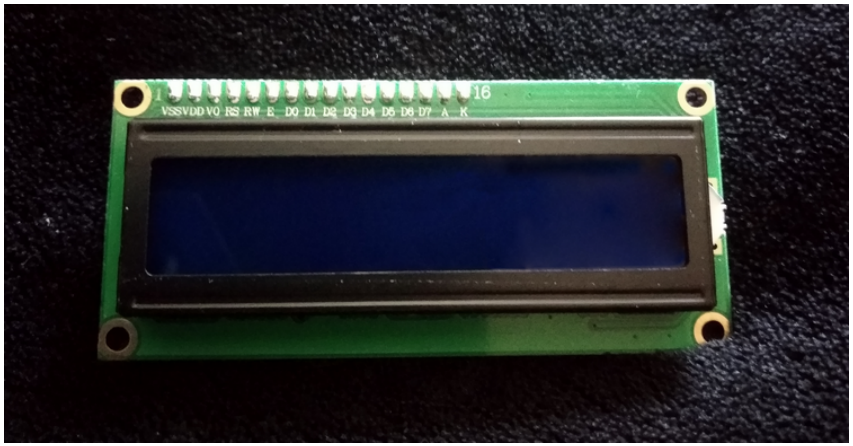
Before start

- Number in **`0x00`** format:
  - Number in hexadecimal system
  - It is used widely in IT, but it is not mandatory
  - If you want to use decimal numbers, you surely can! But don't forget to convert them (converters on Google)!
- Functions to complete has **czech** names to help czech speaking people to recognize them.

Display

Display 16x2

- 16 chars, 2 rows
- Connected via I^2C
- LCD is relatively slow – it cannot be redrawn 1000x in a second
- Controls:
  - `display.h` library and `Display` class
  - `LiquidCrystal_I2C.h` library



- Very simple wrapper* to use the display and print data output from sensors etc.
- Instance of `Display` class (default name is `display` in the code) has three methods:

```
    #include <display.h>
    ...
    /** Methods clear display and print data*/
    display->print_first_line(to_print);
    display->print_second_line(to_print);
    display->clear(); /* Clears the whole display */
   }
```

*wrapper = class, which wraps another class, In this case, it make it easier to use display and wraps `LiquidCrystal_I2C.h`.

- Open–source library from Github
- *De facto* standard for 16x2 and 20x4 displays connected via I^2C
- It uses OOP – so as we
- Briefly (examples on the next slide):
  - We make and object with address set by macro `DISPLAY_ADDRESS`, 16 chars per row and 2 rows
  - We call the methods on it
  - Cursor: current place where the next chr will be printed

```
#include<LiquidCrystal\_I2C.h>
/* Global variable for the display */
LiquidCrystal_I2C *display;
void setup() {
 /* Display initialization */
 display = new LiquidCrystal_I2C(DISPLAY_ADDRESS, 16, 2);
 /* Basic setting */
 display->init();
 /* Enable backlight. */
 display->backlight();
 /* set the cursor to the start
  * First value is for column, second for row,
  * indexed from 0 */
 display->setCursor(0,0);
}
```

```
void loop() {
    /* Clear the display */
    display->clear();
    /* Prints something to the display*/
    display->print(something);
}
```

Speaker

Speaker

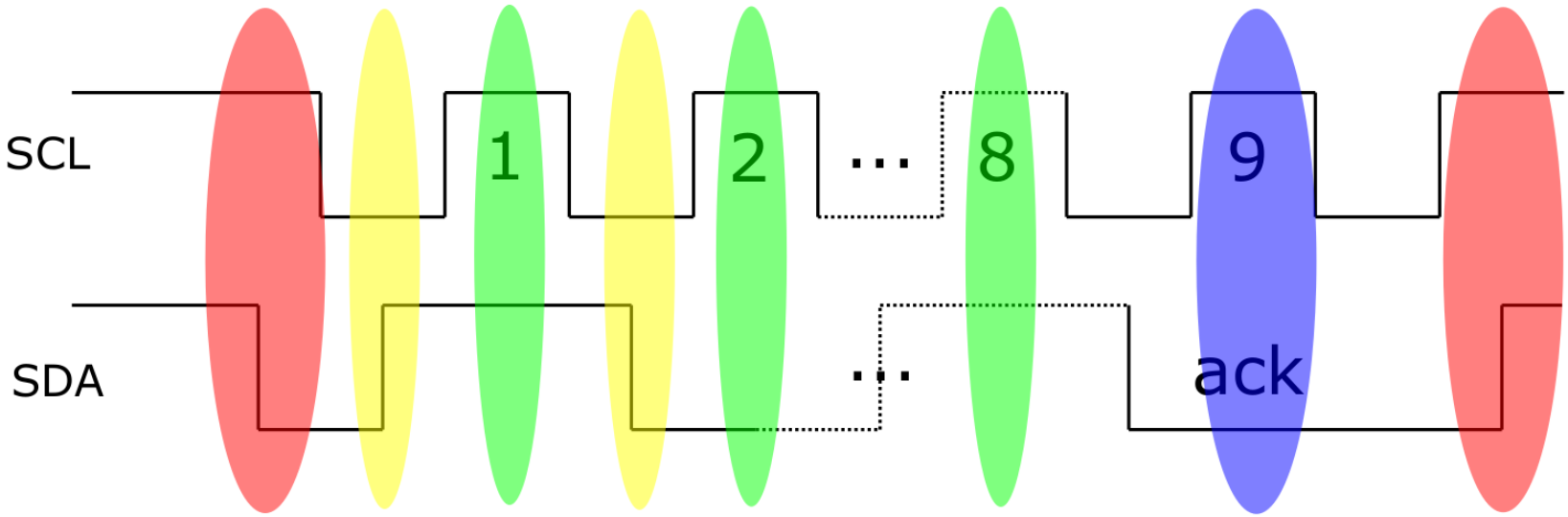
- Trilobot has builtin speaker
- Simpl econtrolling is done by `Speaker` object with default name `speaker`
- Examples in the next slide

Speaker: Usage

```
#include <speaker.h>
/* Enable speaker before use */
speaker->enable();

/* Note: note name - look into speaker.h for list
 * Duration: note duration in  ms
 * */
speaker->beep(<note>, <duration>);
/* Plays Imperial March from Star Wars :) */
speaker->imperial_march();

/* Disable after use (it will whistle instead)*/
speaker->disable();
```

I²C

- I²C /TWI is one of the simpliest and most common buses
- I uses 4 wires 2 for power (not an actual part of I²C), SSDA for data (blue wire) and SCL for clock (yellow wire)
- Master–Slave bus – one device – master – controls multiple slave devices
- For interest: I²C principle below:
  - Red: START and STOP conditions
  - Yellow: Master writes new bit
  - Green: Slave reads new bit
  - Blue: ACK, Slave acknowledges transmission



```
#include <Wire.h> /* Library for IIC*/
... /* in setup(), Wire.begin() is called*/
/*Function activates IIC - called in setup()*/
Wire.begin();
... /*loop()*/
/*Write to a device with address <address> into
 * register <reg>*/
/*Communication start*/
Wire.beginTransmission(<address>);
/*Specify register for writing*/
Wire.write(<reg>);
/* Send data (byte type)*/
Wire.write(data);
/*Communication end*/
Wire.endTransmission();
```

```
#include <Wire.h>  /* Library for IIC*/
... /*loop() */
/*Read x bytes from device <address> from register <reg>*/
Wire.beginTransmission(SRF08_ADDRESS);
/*Write from what register reading starts*/
Wire.write(<reg>);
Wire.endTransmission();

/*From device with address <address> read x bytes*/
Wire.requestFrom(<address>, x);
/* Wait until data ready */
while(!Wire.available());
/* Usually, we dont throw out the data we just read :).
 * Wire.read() will be called x-times here to read x bytes*/
Wire.read();
```

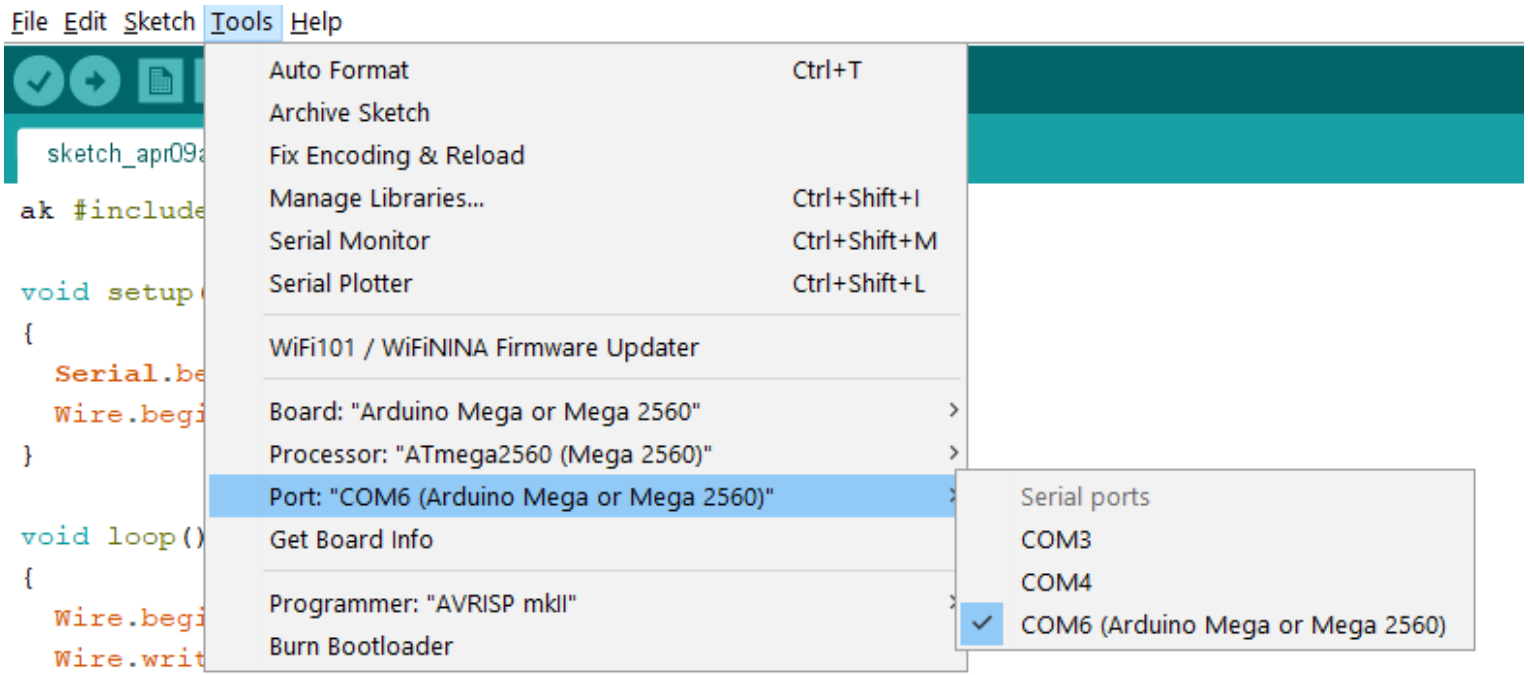
Arduino IDE and Trilobot connection

- Development environment for writing and loading code into Arduino
- Simple – no code completition etc.
- For programming Arduino, it is needed to select a proper port:
  - Go to *Tools/Port* and select right COM port
  - If we don't know which one, we open COM port list, disconnect and reconnect the Arduino. The port which disappears and appears again is the right one.
- Arduino IDE description (picture in the next slide):
  - (1): Main part, code goes here
  - (2): Console, where useful information is shown (during board programming etc.)
  - (3): Buttons for (from left): Check, programming into board, New File, Open and Save

Arduino IDE


Blink | Arduino 1.8.9 (Windows Store 1.8.21.0)
File Edit Sketch Tools Help
(3)
Blink
(1)
Blink
Turns an LED on for one second, then off for one second, repeatedly.
Most Arduinos have an on-board LED you can control. On the UNO, MEGA and ZERO
it is attached to digital pin 13, on MKR1000 on pin 6. LED_BUILTIN is set to
the correct LED pin independent of which board is used.
If you want to know what pin the on-board LED is connected to on your Arduino
model, check the Technical Specs of your board at:
https://www.arduino.cc/en/Main/Products
modified 8 May 2014
by Scott Fitzgerald
modified 2 Sep 2016
by Arturo Guadalupi
modified 8 Sep 2016
by Colby Newman
This example code is in the public domain.
http://www.arduino.cc/en/Tutorial/Blink
*/
// the setup function runs once when you press reset or power the board
void setup() {
// initialize digital pin LED_BUILTIN as an output.
pinMode(LED_BUILTIN, OUTPUT);
}
// the loop function runs over and over again forever
void loop() {
digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage level)
delay(1000); // wait for a second
digitalWrite(LED_BUILTIN, LOW); // turn the LED off by making the voltage LOW
delay(1000); // wait for a second
}
(2)
18
Arduino/Genuino Mega or Mega 2560, ATmega2560 (Mega 2560) on COM6

- Via USB cable
- After successful connection, Trilobot's display should turn on.
- Go to *Tools/Port* and select proper port – if Arduino is the original one, there should be its type name next to the COM port



Connection test

- Open new sketch using *File/New* or keyboard shortcut ctrl + N. Sketch is a Arduino–specific file with .ino suffix
- After Trilobot is connected, copy the following code into `loop()` and load it to Arduino using the green arrow button in the left top part of Arduino IDE:

```
void loop(){
    /*pin 13 into HIGH - LED turns on*/
    digitalWrite(13, HIGH);
    /*500 ms wait */
    delay(500);
    /*Change the LED state */
    digitalWrite(13, LOW);
    /*Wait again*/
    delay(500);
}
```

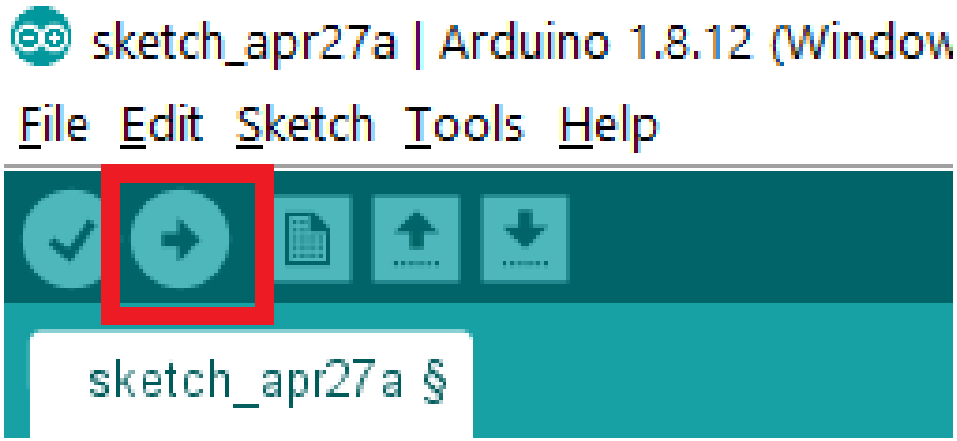
- If everything is all right, LED next to the display should start blinking

Introduction to Arduino programming and OOP

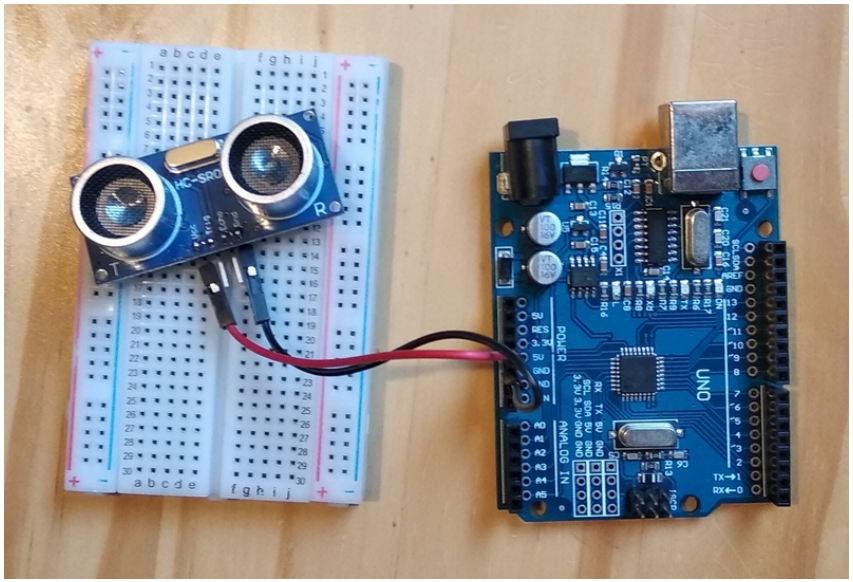
- Language used to write programs for Arduino is named Wiring – it is fully compatible with C++
- Always, there are two functions in the code:
  - `setup()`: This code is called only once; after Arduino starts or after reset
  - `loop()`: This code is executed in an infinite loop

```
void setup()
{
/* Put your setup code here, to run once */
}
void loop()
{
/* Put your main code here, to run repeatedly */
}
```

- By clicking the green arrow, the code will translate and load into Arduino or Arduino IDE shows an error message in the console
- Arduino IDE wants to save your sketch before uploading. It is not mandatory, but recommended.
- If syntax error appears, Arduino IDE shows, where



- Using breadboard and/or DuPont cables



- OOP is a way how to write your code based on *objects*
- *Objects* in the code correspond to the real world objects in two ways:
  - Characteristics – atributes: cat's color or I^2C address of the specific sensor
  - Abilities – methods: cat can meow and sensor can collect data
- *Objects* have its templates – classes
  - For example: The `Cat` class is a template for an instance if a cat named `rose`

- Everything will be shown on `rose` object (imaginary cat) and `display` object – really implemented in the code and can be used.
- If we want to call a methods on an object, we use following syntax:

```
object_name->methos_name(parameters);
```

- If we want our cat `rose` to meow, we just type:

```
rose->meow();
```

- And for printing a line on the display we type:

```
display->print_first_line("Hello world!");
```

- Briefly: Method call is similar to a function call. We only type object name and arrow–> before the method name.

Tutorial

Photosensor

Physical principle:

- Photosensor is unenriched semiconductor –> it conducts the current very poorly (almost not at all)
- However, light/photons gives the electrons enough energy in the valence layer –> pair electron–hole are created
- These pairs can conduct current very well
- The more the light shines, the more pairs appears and the more current can be conducted

Photosensor

- Mostly a little sensor, distinguishable by its characteristics curves
- In our case, it is a part of SRF–08 sensor

- Test if Trilobot and Arduino IDE are connected correctly
- Complete following into the function template `SRF08::zmer_svetlo()` (placed in `srf08.cpp`):
  - Register number, in which SRF–08 stores value from the photosensor (see documentation, page 4, address is in *Location* column)
  - Read the values via I^2C and return it (see I^2C)
  - Print the value on on the display (using `display->print_first_line()`)
  - (optional) Turn the LED on when it gets too dark

Photosensor: Template from srf08.cpp

```
byte SRF08::zmer_svetlo()
{   this->set_measurement();
    /*We request the data from register PHOTOSENSOR REGISTER*/
    Wire.beginTransmission(SRF08_ADDRESS);
    /* 0x00 is only placeholder to make code compilable */
    Wire.write(0x00/*TODO PHOTOSENSOR REGISTER*/);
    Wire.endTransmission();

    /*Request for one byte...*/
    Wire.requestFrom(SRF08_ADDRESS, 1);
    /*...and wait until is available*/
    while(Wire.available() < 0);
    /*TODO: read the value and return it.
     * Following lines only make the code syntax error free */
    byte light_intensity  = 0;
    return light_intensity ;
}
```

Photosensor: Instructions for `SRF08::zmer_svetlo()`

- Address of the register with light intensity information from photosensor is **`0x01`**
- I^2C reading is shown below:

```
Wire.read();
/*Note: returns byte*/
```

- Displaying the values is done via:

```
display->print_first_line(data);
```

- Now, you just need to retype this to the function itself.

```
byte SRF08::zmer_svetlo()
{   this->set_measurement();
    /*We request the data from register 0x01*/
    Wire.beginTransmission(SRF08_ADDRESS);
    Wire.write(0x01);
    Wire.endTransmission();

    /*Request for one byte...*/
    Wire.requestFrom(SRF08_ADDRESS, 1);
    /*...and wait until is available*/
    while(Wire.available() < 0);

    byte light_intensity = Wire.read();
    return light_intensity;
}
/* V loop()*/
display->print_first_line(srf08->photosensor());
```

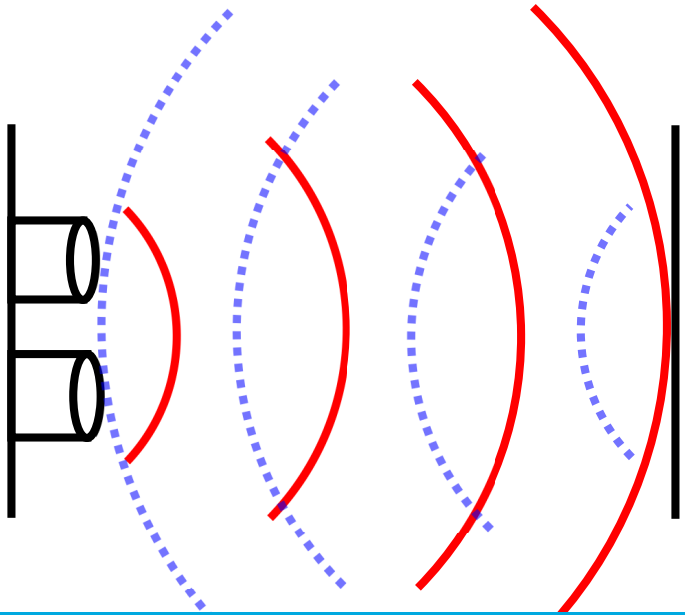
Distance measurement

In this chapter we will explain:

- How can we measure the distance
- Measurement using HC–SR04, SRF–08 and Sharp–GP2D120 sensors

How can we measure distance

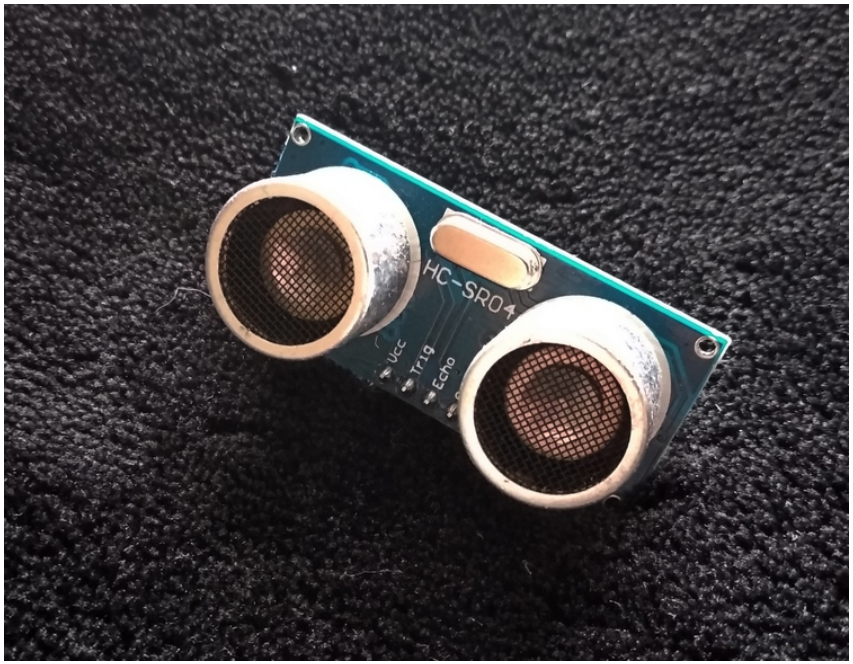
- With ruler :) – robots cannot do that
- If the robot is movable, we can use motor/wheel spin. Not very useful or accurate – wheels can slip etc.
- Using movement sensors (accelerometer…) – measurement of distance traveled, ot the distance from a barrier
- Wirelessly: using some kind of wave (light, sound…)
  - This is what we will try with all three sensors
  - The principle can be seen in the picture: red waves are sent by the sensor, blue waves are received. From the time between transmitting and receiving we can compute the distance (if we know the speed of the wave).



HC_SR04

HC_SR04: Introduction

- Ultrasound distance measurement sensor
- 4 wires: 2x power, `ECHO_PIN` and `TRIGGER_PIN` (macros)
- Documentation



HC_SR04: Principle

- When the `TRIGEER_PIN` is in HIGH state for at least 10 microseconds, sensor start measurement
- `ECHO_PIN` is in HIGH state until the wave comes back
- After the reflected wave comes back, `ECHO_PIN` goes back to LOW
- Value in microseconds has to be converted to distance using the speed of sound: **340 m/s or 0.0343cm/us**

- Complete the function template named `HCSR04::zmer_vzdalenost()`:
  - Set the `TRIGGER_PIN` to HIGH for `10` mikrosekund
    Sensor will send the wave and wait for it to come back
  - Listen on `ECHO_PIN` and get the duration of the pulse in microseconds
    Once the wave comes back, sensor ends the pulse (HIGH state on `ECHO_PIN`)
  - Convert microseconds to centimeters using the speed of sound 340 m/s (or `0.0343`cm/us, which should be more useful)

```
long HCSR04::zmer_vzdalenost()
{/* Set LOW to be sure */
    digitalWrite(TRIGGER_PIN, LOW);
    delayMicroseconds(2);
    /* TODO: Set TRIGGER_PIN to HIGH HIGH for 10
     * microseconds, then return it to LOW*/

     /*TODO: using pulseIn(pin, state) find out, how
      * long will be ECHO_PIN in HIGH  a convert
      * time to distance */
    long return_value = 0;
    return return_value;
}
```

- We set the `TRIGGER_PIN` to HIGH for 10 microseconds using

```
digitalWrite(TRIGGER_PIN, HIGH);
delayMicroseconds(10);
digitalWrite(TRIGGER_PIN, LOW);
```

- Listening on `ECHO_PIN` is done by `pulseIn`:

```
long echo = pulseIn(ECHO_PIN, HIGH);
```

- Conversion between time and distance when speed is known is simple :) (distance = speed * time)

```
long HCSR04::zmer_vzdalenost()
{/* Set to LOW for a while to be sure */
    digitalWrite(TRIGGER_PIN, LOW);
    delayMicroseconds(2);

    digitalWrite(TRIGGER_PIN, HIGH);
    delayMicroseconds(10);
    digitalWrite(TRIGGER_PIN, LOW);

    long echo = pulseIn(ECHO_PIN, HIGH);
    long return_value = echo * 0.0343;
    return return_value;
}
```

SRF-08

- The same principle as with HC_SR04
- Communication via I^2C
- Documentation
- In addition to distance measurement, is can also measure light intensity
- It has way more complicated setting, however, we will use only the basics

- Send the measurement command to the register with address `0x00` with proper value:
  - `0x50`: inches
  - `0x51`: centimetres
  - `0x52`: microseconds
- Wait until the measurement is completed. According to the documentation, 60 ms should be enough, but sometimes, it is not true.
- Request the measured value via I^2C :
  - The values is divided into two registers (addresses `0x02` and `0x03`)
  - It is divided into the high byte and the low byte – we have to join them

- Complete the `SRF08::zmer_vzdalenost()` function:
  - Send the right value to register `0x00`. First, we have to send the register address, then the value (look at the examples of I^2C)
  - Wait at least `60` ms, but `100` is safer (we don't mind time efficiency)
  - Read the values from registers `0x02` and `0x03`
  - Join values from registers `0x02` a `0x03` (high and low byte)

- Complete the `SRF08::zmer_vzdalenost()` function:
  - Send the right value to register `0x00`. First, we have to send the register address, then the value (look at the examples of I^2C)
  - Wait at least 60 ms, but 100 is safer (we don't mind time efficiency)
  - Read the values from registers `0x02` and `0x03`
    - Note: SRF-08 Always send either all registers (starting with `0x01`) or we have to explicitly say which starting register we want. However, this is solved by the template, so everything you have to do is to read two bytes. :)
  - Join values from registers `0x02` a `0x03` (high and low byte)

- Complete the `SRF08::zmer_vzdalenost()` function:
  - Send the right value to register `0x00`. First, we have to send the register address, then the value (look at the examples of I^2C)
  - Wait at least 60 ms, but 100 is safer (we don't mind time efficiency)
  - Read the values from registers `0x02` and `0x03`
    - Note: SRF-08 Always send either all registers (starting with `0x01`) or we have to explicitly say which starting register we want. However, this is solved by the template, so everything you have to do is to read two bytes. :)
  - Join values from registers `0x02` a `0x03` (high and low byte)
    - Hint #1: We have to use bit shift or multiplication. We have to use bigger datatype then byte – `uint16_t` is enough

- Complete the `SRF08::zmer_vzdalenost()` function:
  - Send the right value to register `0x00`. First, we have to send the register address, then the value (look at the examples of I^2C)
  - Wait at least 60 ms, but 100 is safer (we don't mind time efficiency)
  - Read the values from registers `0x02` and `0x03`
    - Note: SRF-08 Always send either all registers (starting with `0x01`) or we have to explicitly say which starting register we want. However, this is solved by the template, so everything you have to do is to read two bytes. :)
  - Join values from registers `0x02` a `0x03` (high and low byte)
    - Hint #1: We have to use bit shift or multiplication. We have to use bigger datatype then byte – `uint16_t` is enough
    - Hint #2: `uint16_t value = (uint16_t)(high<<8)+low;`

```
int SRF08::zmer_vzdalenost(){
  /*Communication start*/
  Wire.beginTransmission(SRF08_ADDRESS);
  /*Where we want to write...*/
  Wire.write(REG_CMD); /* expanduje do 0x00 */
  /*...and what we want to write.*/
  Wire.write(unit);
  /*End the communication in the end */
  Wire.endTransmission();

  /* TODO: wait 100 ms*/

  /* We need 2 bytes starting at 0x02*/
  Wire.beginTransmission(SRF08_ADDRESS);
  Wire.write(0x02);
  Wire.endTransmission();
  /* TODO read the data, join them and return*/

  uint16_t return_value = 0;
  return return_value;
}
```

- Waiting is a piece of cake – delay is what we need
- Reading values from registers is nothing difficult at all – the cycle just awaits the data

```
Wire.requestFrom(SRF08_ADDRESS, 2);
while(Wire.available() < 0);
byte high =  Wire.read();
byte low =  Wire.read();
```

- Joining two bytes into one 16–bit number is a little bit more difficult:

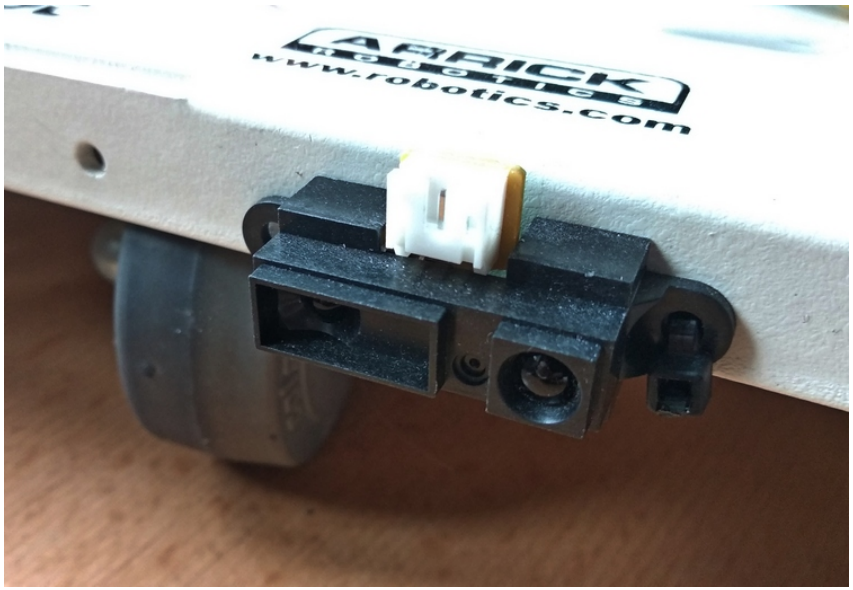
```
uint16_t number = (uint16_t)(high << 8)+low;
```

Note: << operator is something like a *shifter*, which shifts the number into the left by x positions. So 111 << 3 will be 111000 (empty positions are filled by 0). And 101 << 3 + 101 = 101000+101 = 101101 – absolutely same principle as we have used above.

```
int SRF08::zmer_vzdalenost()
{
  Wire.beginTransmission(SRF08_ADDRESS);
  Wire.write(REG_CMD); /* expands into 0x00 */
  Wire.write(0x51); /* cm */
  Wire.endTransmission();
  delay(100); /* wait */
  Wire.beginTransmission(SRF08_ADDRESS);
  Wire.write(0x02);
  Wire.endTransmission();

  Wire.requestFrom(SRF08_ADDRESS, 2);
  while(Wire.available() < 0);
  byte high =  Wire.read();
  byte low =  Wire.read();

  uint16_t return_value = (uint16_t)(high << 8)+low;
  return return_value;
}
```

Sharp-GP2D120

- This sensor uses infrared light instead of sound
- Works only for 3 cm and more
- Very simple: the distance is derived from the voltage on the data pin
- There is no linear relationship between the voltage and distance (see documentation) – we have to either approximate or compute quite a difficult function.
- Documentation



- Very shortly after start the sensor starts measuring
- On the data pin (`SHARP_PIN` macro) can be measured voltage
- We can compute the distance from the graph in documentation.
  - Analytically: We find a function which suits the graph the most
  - Numerically: the original graph from the documentation will be divided into small intervals, which can be approximated by linear function. This is way more faster, but not as accurate as the previous option.
  - For our needs, there is function called `aproximate()`, using numerical approximation

- Complete the function `Sharp::zmer_vzdalenost()`, located in `sharp1994v0.cpp`:
  - Read the value on the data pin (`SHARP_PIN` macro)
  - Convert the value from interval `0`–`1024` to interval `0`–`5`
    - Because function `analogRead()` returns value in interval `0`–`1024` and the graph in documentation is for values `0`–`5`, we have to convert one value to another. And converting the measured value to voltage is much simpler.
  - using function `aproximate()`, get the distance in cm

```
float Sharp::zmer_vzdalenost()
{
/*TODO:
 * read the value on SHARP_PIN
 * convert the value from 0-1024 to 0-5
 * using this->aproximate() get the value in cm
  * and return it
  */
  float aproximated_value = 0;
  return aproximated_value;
}
```

- Read the value on SHARP_PIN is simple:

```
int value = analogRead(SHARP_PIN);
```

- When converting from range 0–1024 to 0–5, we have to think a bit, but it is not difficult in general:

```
/*ANALOG_MAX = 1024. DEFAULT_VOLTAGE = 5*/
float normalized_value = 0;
normalized_value = (value/ANALOG_MAX)*DEFAULT_VOLTAGE;
```

- And calling a function is just... Calling a function :)

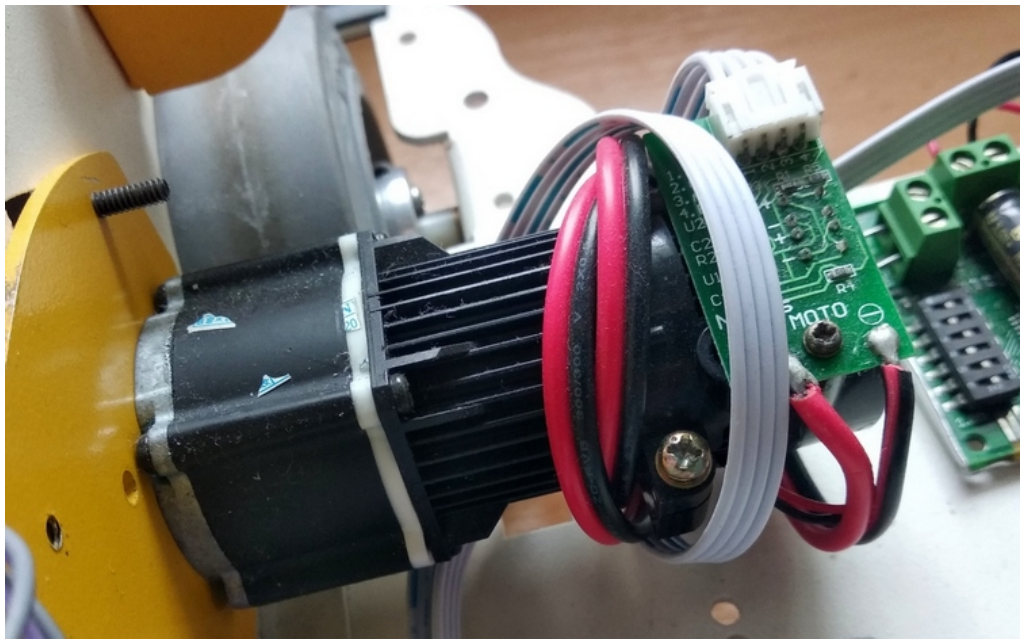
```
aproximated_value = this->aproximate(normalized_value);
```

```
float Sharp::zmer_vzdalenost()
{
  int value = analogRead(SHARP_PIN);

  /*ANALOG_MAX = 1024. DEFAULT_VOLTAGE = 5*/
  float normalized_value = 0;
  normalized_value = (value/ANALOG_MAX)*DEFAULT_VOLTAGE;

  float aproximated_value = 0;
  aproximated_value = this->aproximate(normalized_value);
  return aproximated_value;
}
```

We’re moving

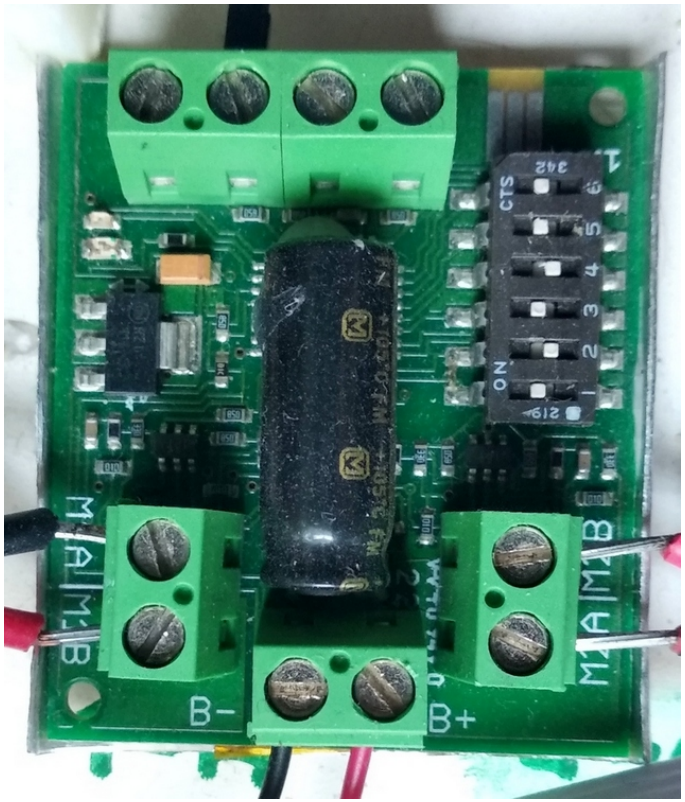
We’re moving

- Two Faulhaber 16002 motors with encoders (we will tell what is encoder later)
- Motors are controlled by Sabertooth 2x5 (board specially designed for controlling motors)
- Communication with Sabertooth is via Serial line, but only in one direction (Arduino –> Sabertooth) only one wire needed
- Double–channel encoders provides feedback on movement. We use only one channel of each encoder.



Motor controlling with Sabertooth 2x5

- Sabertooth works in Simplified Serial mode
  - Principle: two bytes (one for each wheel) are transferred via Serial line. The number provides the information what motor is controlled and how fast (and in hat direction) it should move.
- Specific values on the next slide
- Communication is extremely simple and fully sufficient for Trilobot.



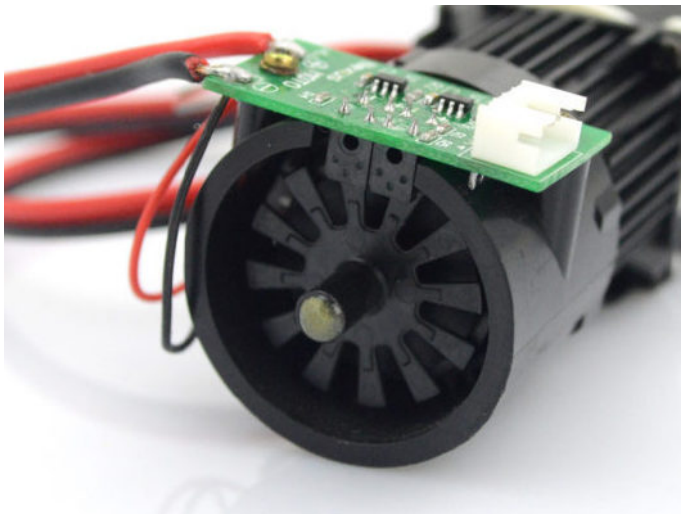
Basic values to control the motors

- 0 is both motor stop
- 1–127 controls left motor:
  - 1 is full backward
  - 64 is left motor stop
  - 127 is full forward
- 128–255 controls right motor (these are only number for left motor + 127):
  - 128 is full backward
  - 192 is right motor stop
  - 255 is full forward

```
/* Example: */
Serial1.write(0x0); /* Both motors stop */
...
/* Both motors full forward - we can use dec as well*/
Serial1.write(0x7F); /* = 127 dec*/
Serial1.write(0xFF); /* = 255 dec*/
```

Encoders

- Encoders periodically change a signal* depending on the movement of the motors. One change in the signal = one step.
- Signal changes its value from 0 to 1 and back. When there is a barrier in the sensor (one of the black lamellas in the picture) encoder sends 0, 1 in other cases
- If we know the number of signal changes per one full wheel rotation and wheel diameter, we can compute the distance travelled
- In this tutorial, only one channel will be used – only one wire with the signal



- Movement straight
- Turning

- One advantage, compared to turning: the value sent for both motors is the same (only shifted by 127 for one of them to actually control both motors).
- Principle*:
  - **Inputs**:
    - Desired distance travelled in cm
    - Speed in interval <−100,100>, negative values is for backward movement
  - We compute, how much steps we have to read on encoder to move a given distance
  - We compute what number we have to send to Sabertooth (`driving_byte` variable). It is done by `Motors->get_speed_from_percentage()` function
  - We send the bytes via Serial line `Serial1`
  - We wait until Trilobot ends its move. you can see an infinite loop in the function template, which is not completely infinite
  - We stop the motors

Note: We are going to use interrupts. If you are interested, look at Google or, if you can speak Czech, look at the tutorial text to the advanced section of this chapter. However, you don't need to know it to pass the tutorial.

*Probably, you can think out a better way to implement function like this, but this way is clear and compatible to a reference function in the code.

Movement straight: Template

```
void Motors::jed_rovne(int distance, int speed)
{
  int steps_to_go = 0/*TODO steps needed*/;
  byte driving_byte;
  driving_byte = Motors::get_speed_from_percentage(speed);
  attach_interrupts();
  /*TODO send driving byte to both motors*/
  /*This cycle will be explained*/
  while(1)
  {
    if(steps_left == steps_to_go ||
    steps_right == steps_to_go)
      break;
  }
  /*TODO stop motors*/
  detach_interrupts();
}
```

How can we compute the steps?

- We need two values: the number of steps per one full rotation of the wheel and the wheel diameter
- There are macros for both the values: `STEPS_ONE_CHANNEL` and `WHEEL_CIRCUIT`
- The exact computation is only a simple mathematics (if x steps correspond to one distance, then how many steps correspond to another distance?) :)

How can we compute the steps?

- We need two values: the number of steps per one full rotation of the wheel and the wheel diameter
- There are macros for both the values: `STEPS_ONE_CHANNEL` a `WHEEL_CIRCUIT`
- The exact computation is only a simple mathematics (if x steps correspond to one distance, then how many steps correspond to another distance?) :)

```
(int)(distance/WHEEL_CIRCUIT*STEPS_ONE_CHANNEL)
```

Infinite loop? No, interrupts!

- There is something like an infinite loop in the code
- Actually, we can break the cycle and jump out thanks to *the interrupts*
- What is an interrupt? Where can be used?
  - An Interrupt is used in that situation, when you need to stop currently executed code, jump to another place, make some simple action – add to a counter, set a flag etc. This action is called interrupt handler.
  - Imagine it as a teacher lecturing a class. Once in a while, some of the student raises their hand and asks a question. The teacher interrupts (see the word? :)) their lecture, answers the question and continues.
  - In our case, interrupts are activated by `attach_interrupts()` and deactivated by `detach_interrupts()`. Our interrupt handlers only adds one to the step counters`steps_left` or `steps_right`. That means that the condition inside the cycle can be changed and eventually, the cycle will end.

- We have the formula for steps needed::

```
int steps_to_go = (int)(distance/
                  WHEEL_CIRCUIT*STEPS_ONE_CHANNEL);
```

- Sending via Serial line is simple (stopping the motors means sending a 0x00):

```
Serial1.write(driving_byte);
Serial1.write(driving_byte+127);
...
Serial1.write(STOP_BYTE); /* = 0x00*/
```

```
void Motors::jed_rovne(int distance, int speed)
{
 int steps_to_go = (int)(distance/
                   WHEEL_CIRCUIT*STEPS_ONE_CHANNEL);
  byte driving_byte;
  driving_byte = Motors::get_speed_from_percentage(speed);
  attach_interrupts();
  Serial1.write(driving_byte);
  Serial1.write(driving_byte+127);
  while(1)
  {
    if(steps_left == steps_to_go ||
    steps_right == steps_to_go)
      break;
  }
  Serial1.write(STOP_BYTE); /* = 0x00*/
  detach_interrupts();
}
```

Turning

Turning: Introduction

- Turning is not very different from moving straight – the functions will be similar
- Turning can be done in many ways:
  - The simplest: the motor on the side we turn stops, the second one moves forward
  - In place: the motor on the side we turn moves backward and the other one forward
  - Like a car: robot makes a circuit – both motors are moving forward, but with different speeds...
- We will choose the first one – the simplest one

- Turning function principle:
  - **Inputs**:
    - The angle in degrees from interval $< -180, 180 >$, where the positive angle is for turning right and the negative one is for turning left
    - Speed: this time only from `0` to `100`
    - Again the same as with `Motors::jed_rovne()` function: the inputs can be chosen differently and the function will do the job as well. This is only a one of the ways.
  - We compute how many steps must the outer motor travel
  - We send the speed information to Sabertooth – we choos right or left motor depending on the direction of the turn
  - We stop the motor

```
void Motors:zatoc(int angle, int speed)
{
  int steps_to_go = 0/*TODO steps needed */;
  attach_interrupts();
  byte driving_byte;
  driving_byte = Motors::get_speed_from_percentage(speed);
 /*TODO turning depending on the sign of
  * the angle parameter*/
  while(1)
  {
    if(steps_left == steps_to_go ||
      steps_right == steps_to_go)
      break;
  }
  /*TODO stop the motor */
  detach_interrupts();
}
```

How can we compute the steps needed?

- It is a little more difficult this time – the wheel will describe part of the circle
- We have to solve this question: How much of a circle of radius `r` belongs to an angle of size α?
- Then, once we know the distance, we can use the formula from `jed_rovne()`
- Solution in the next slide

How can we compute the steps needed?

```
/*
r = 20, we can use TURN_RADIUS macro
General principle: The ratio of the part of a circle
to the circumference of the circle is the same as
the ratio of a given angle to an angle full (360 degrees).
This brings us the distance that we can substitute into
the same formula we used for driving straight.
Example: circumference is 125 cm, angle is 60 degrees,
part of the circle is x
60 is 1/6 of 360 -> x is 1/6 of 125 -> x is +-21 cm
*/
int steps_to_go = (int) (((2 * TURN_RADIUS * PI * angle)
/ 360) / WHEEL_CIRCUIT * STEPS_ONE_CHANNEL);
```

- The right number of steps needed we already know from the previous slide – just copy that
- We can send data via Serial line as well:

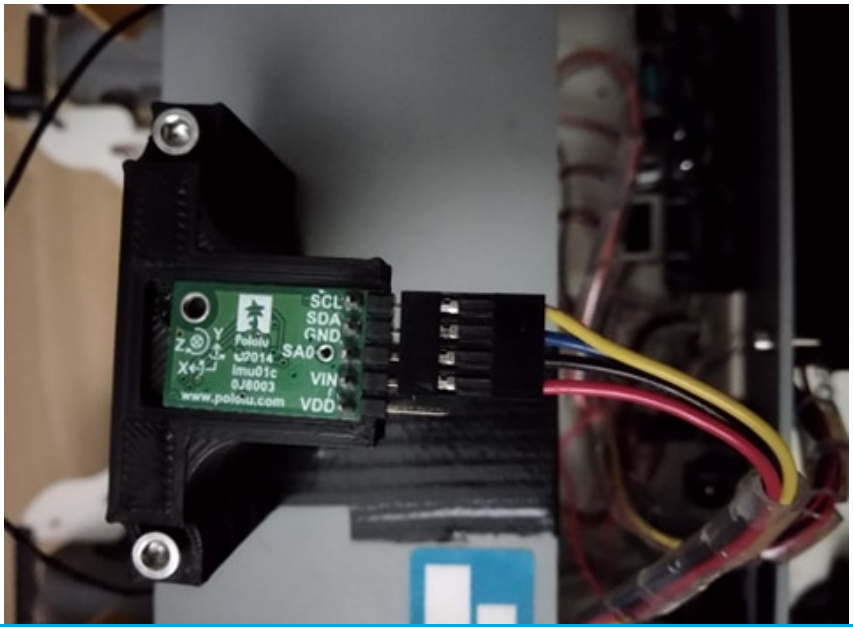
```
Serial1.write(driving_byte);
...
Serial1.write(STOP_BYTE);
```

```
void Motors::zatoc(int angle, int speed)
{
  int steps_to_go = (int) (((2 * TURN_RADIUS * PI * angle)
                / 360) / WHEEL_CIRCUIT * STEPS_ONE_CHANNEL);
  attach_interrupts();
  byte driving_byte;
  driving_byte = Motors::get_speed_from_percentage(speed);
  Serial1.write(driving_byte);

  while(1)
  {
    if(steps_left == steps_to_go ||
      steps_right == steps_to_go)
      break;
  }
  Serial1.write(STOP_BYTE);
  detach_interrupts();
}
```

IMU – inertial measurement unit

What is IMU?

- IMU is the combination of three sensors: gyroscope, magnetometer and accelerometer
- We can use the data from any of the three sensor separatedly, but there is also something called AHRS*, which is the best for computing the position
- Because of the magnetometer, IMU has to be above the Trilobot. Trilobot has metal body and magnetometer could interfere
- We can make a compass from magnetometer
- Gyroscope can be used as shake detector
- Accelerometer will be used for impact detector



What is IMU?

- IMU is the combination of three sensors: gyroscope, magnetometer and accelerometer
- We can use the data from any of the three sensor separatedly, but there is also something called AHRS*, which is the best for computing the position
- Because of the magnetometer, IMU has to be above the Trilobot. Trilobot has metal body and magnetometer could interfere
- We can make a compass from magnetometer
- Gyroscope can be used as shake detector
- Accelerometer will be used for impact detector

*AHRS – Attitude and heading reference system – serves for computing the position and heading. It provides position information in all three axes and some additional pieces of information. The implementation is, however, quite a difficult task and even Trilobot's AHRS does not work as intended.

Compass #1

Compass #1: Introduction

- Compass will return the angle between the front of the Trilobot and north *
- This method is very simple, but not as accurate as can be
- We neglect 3D space and using only 2D

Compass #1: Introduction

- Compass will return the angle between the front of the Trilobot and north *
- This method is very simple, but not as accurate as can be
- We neglect 3D space and using only 2D

* For simplicity, we will be satisfied with the angle from 0 to 180 °

- Complete template `float Magnetometer::Compass_1()`:
  - Read the magnetometer data
  - Throw away the z–part of the data
  - We compute the angle between the data vector from magnetometer and the `front` vector

```cpp
float Magnetometer::Compass_1()
{
  /*TODO get the magnetometer data into the variable below*/
  vector<int16_t> mag_values = {0,0,0};
  /* It says where is the front.
   * Sensor is mounted backwards - it is the reason
   * for -1 */
  vector<int> front = {-1,0,0};

  mag_values.x -= (mag_min.x + mag_max.x) / 2;
  mag_values.y -= (mag_min.y + mag_max.y) / 2;
  /*TODO set z coordinate to 0*/
  vector<float> norm_mag_values;
  norm_mag_values = vector_normalize(mag_values);

  /*TODO compute the angle between two vector.
   * Guide will help*/
  float angle = 0;
  return angle;
```

- We read the data by:

```
this->get_intensity();
```

- Zeroing the coordinate is just... zeroing the coordinate :)

```
mag_values.z = 0;
```

- We know the formula for the angle between two vectors from the high school: $\alpha = acos(\frac{mag\_values * front}{|mag\_values| * |front|})$
- In the code, you can use `vector_dot(u,v)` for scalar product and `vector_abs(v)` for absolute value of the vector. Arcus cosine is just `acos()`, however, it returns the value in radians – just multiply the result by $\frac{180}{\Pi}$

Compass #1: Solution

```
float Magnetometer::Compass_1()
{
 vector<int16_t> mag_values = this->get_intensity();
 /* It says where is the front.
   * Sensor is mounted backwards - it is the reason
   * for -1 */
  vector<int> front = {-1,0,0};

  mag_values.x -= (mag_min.x + mag_max.x) / 2;
  mag_values.y -= (mag_min.y + mag_max.y) / 2;
  mag_values.z = 0;
  vector<float> norm_mag_values;
  norm_mag_values = vector_normalize(mag_values);

  float angle = acos(vector_dot(norm_mag_values, front)
              /
              (vector_abs(norm_mag_values)*vector_abs(front))
            )* 180 / PI;
  return angle;
```

Compass #2

- Compass will return the angle between the front of the Trilobot and north *
- This method is not quite simple, but works very well
- It uses vector arithmetic

- Compass will return the angle between the front of the Trilobot and north *
- This method is not quite simple, but works very well
- It uses vector arithmetic

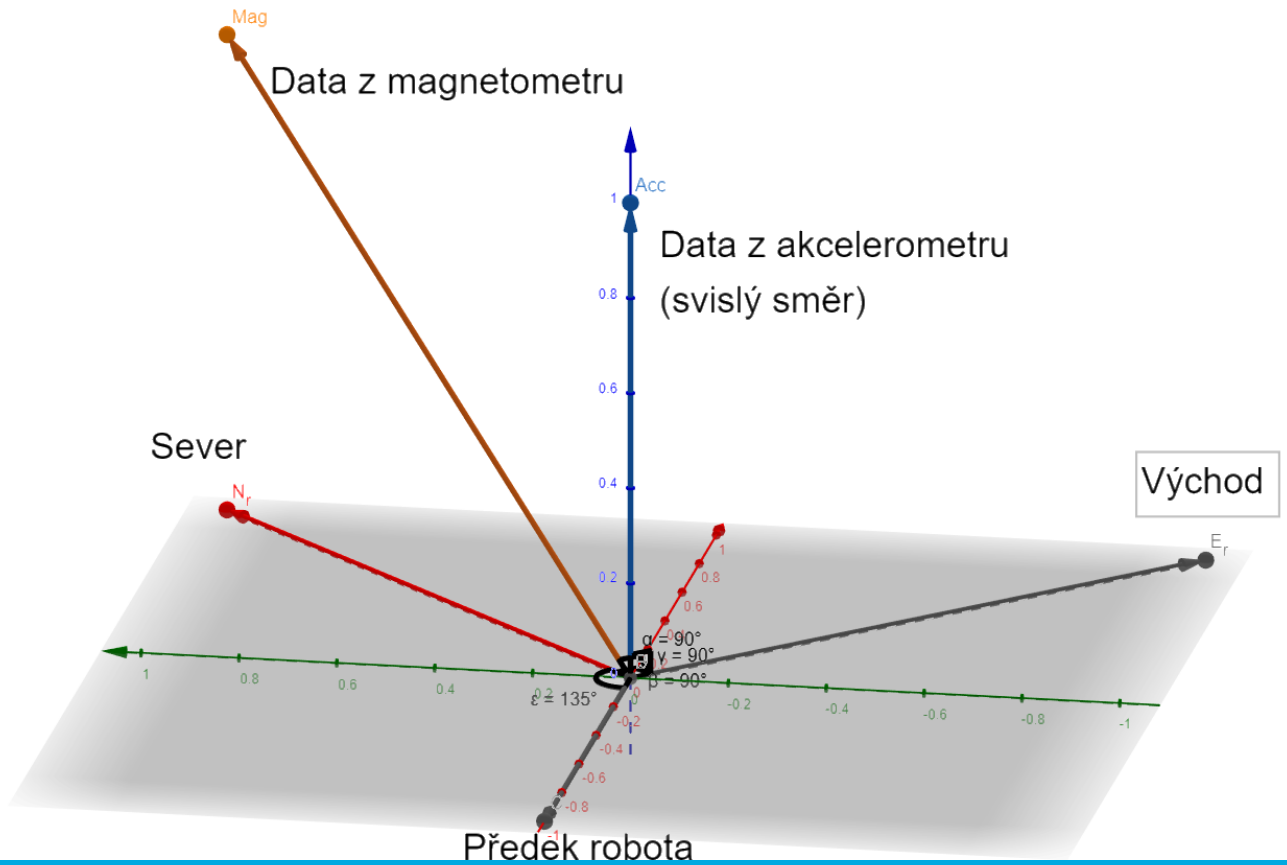
* For simplicity, we will be satisfied with the angle from 0 to 180 °

Compass #2: Tasks

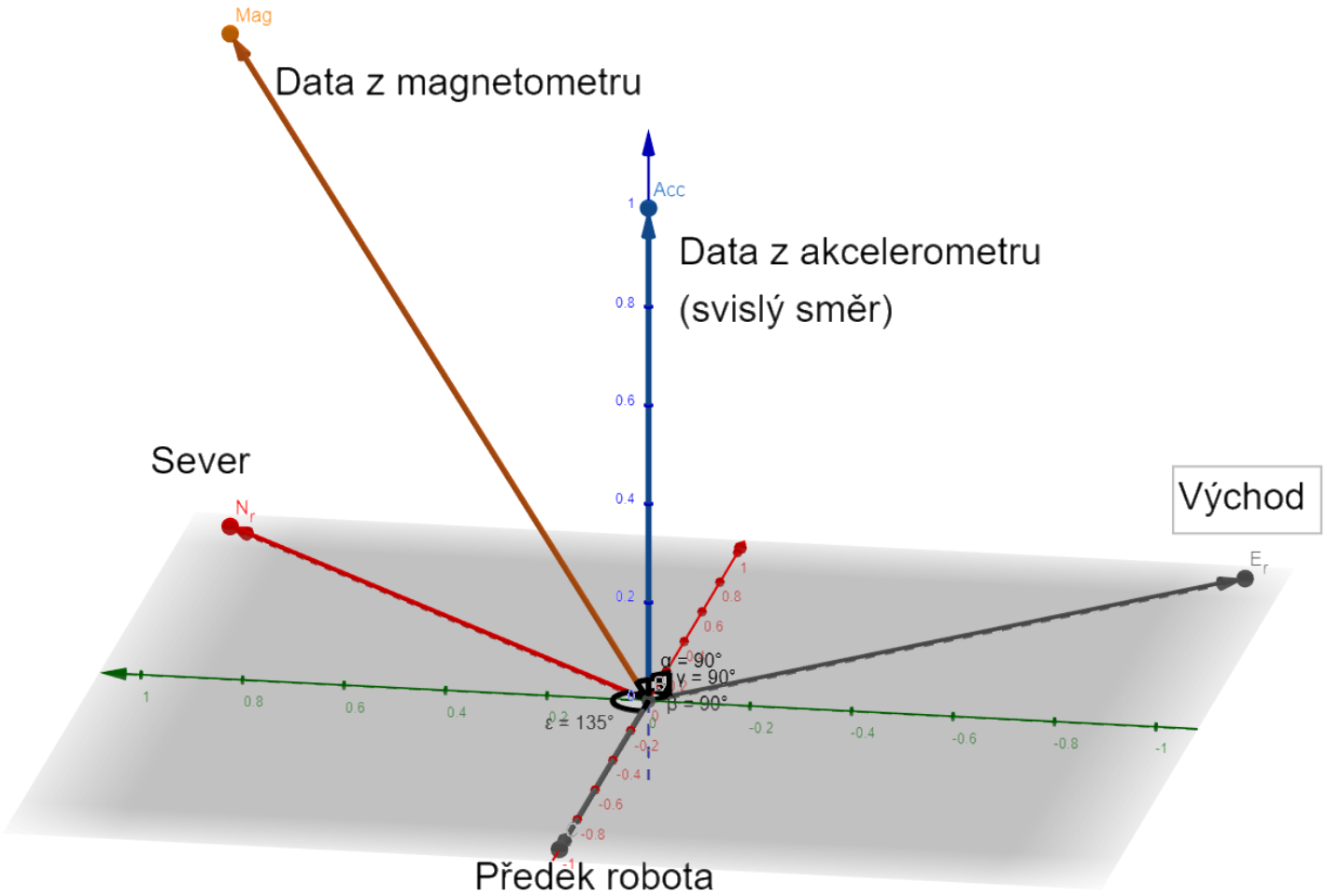
- We complete the template
  `Magnetometer::Compass_2(Accelerometer accel)`:
  - From the magnetic field intensity vector, vertical direction vector and Trilobot's front vector, we will compute the angle between the north and the Trilobot's front side
  - The method will be described in the next slides

```
float Magnetometer::Compass_2(Accelerometer *accel)
{   /* It says where is the front.
    * Sensor is mounted backwards - it is the reason
    * for -1 */
  vector<int> front = {-1,0,0};
  vector<int16_t> mag_values = this->get_intensity();
  vector<float> accel_values = accel->get_acceleration();
  /** offset substraction */
  mag_values.x -=(mag_min.x+mag_max.x)/2;
  mag_values.y -=(mag_min.y+mag_max.y)/2;
  mag_values.z -=(mag_min.z+mag_max.z)/2;

  vector<float> E; /* east */
  vector<float> N; /* north*/
  /* TODO: compute the north using guide */

  float angle = 0; /*TODO angle between N and front*/
  return angle; }
```

- Vector product of the vertical direction and the magnetometer data gives us the vector of East
- How can we know it is East? East is perpendicular to the verical direction (East is horizontal) and it is also perpendicular to the magnetometer data (it points to north, but in 3D. And we need the north in horizontal plane)



- The vector product of the vertical direction and east gives us north in horizontal plane. And that is what we want!
- How can we know it is North? It is perpendicular both to east and vertical direction



- Computing the angle between two vectors is jsut implementation of the formula $\alpha = acos(\frac{N*front}{|N|*|front|})$:

```
/*vector_dot je skalarni soucin*/
float angle = acos(vector_dot(N, front)
                   /
                   (vector_abs(N)*vector_abs(front))*180/PI;
```

```
float Magnetometer::Compass_2(Accelerometer *accel)
{
  /***Deleted lines which did not change***/

  vector<float> E; /* East */
  vector<float> N; /* North*/
  E = vector_cross(mag_values, accel_values);
  E = vector_normalize(E);
  N = vector_cross(accel_values, E);
  N = vector_normalize(N);

  float angle = acos(vector_dot(N, front)
                /(vector_abs(N)*vector_abs(front))
                * 180/PI;
  return angle;
}
```

Shock/vibration detector

Vibration detector: Introduction

- It is used in many places:
  - earthquake detection – seismograph
  - Checking the machine's state – if the machine vibrates a more than usual, something can be wrong
  - Fragile packages control together with impact detectors
- We will make such a sensor from the gyroscope

Vibration detector: Tasks

- Using two measurements from the gyroscope and their difference, we will determine if Trilobot is being shaken or not
- If so, we willl turn the LED on in the `loop()` function

- In gyroscope.cpp

```
bool Gyroscope::detektor_otresu(float treshold)
{
  /* TODO get two datasets from gyroscope with some delay
   * between them- 20 ms should be enough*/
  vector<float> first = {0,0,0};
  vector<float> second = {0,0,0};
  vector<float> shake = {0,0,0};

  /* We will get their difference */
  shake.x = abs(first.x-second.x);
  shake.y = abs(first.y-second.y);
  shake.z = abs(first.z-second.z);

  /* TODO if some difference in any of the three axes will be
   * greater than treshold, we turn the LED_BUILTIN on*/
 return false;
}
```

- Data can be read by `this->get_angular_velocity()`:

```
vector<float> data = this->get_angular_velocity();
```

- Zpoždění už známe – použijeme funkcí `delay()`
- Porovnání dat z vektoru také zvládneme:

```
if(data.x > treshold || data.y > treshold...)
{/* Do...*/}
```

Vibration detector: Solution in gyroscope.cpp

```
 bool Gyroscope::detektor_otresu(float treshold)
{
  vector<float> first = this->get_angular_velocity();
  delay(20);
  vector<float> second = this->get_angular_velocity();
  vector<float> shake = {0,0,0};
  shake.x = abs(first.x-second.x);
  shake.y = abs(first.y-second.y);
  shake.z = abs(first.z-second.z);
  if(shake.x > treshold || shake.y > treshold
    || shake.z > treshold)
    return true;
  else
    return false;
}
```

Vibration detector: Solution in `main.ino`

```
void loop()
{
...
/* 10 is a good starting value, but you can experiment
 * a little bit :) */
/* PLEASE! DO NOT SHAKE TOO MUCH! THANKS! :)*/
if(gyro->shake_detector(treshold=10) == true)
{
    digitalWrite(LED_BUILTIN, HIGH);
}
else
{
    digitalWrite(LED_BUILTIN, LOW);
}
...
}
```

Impact detector

Impact detector: Introduction

- We can see impact detectors for example when we buy something – they can be seen on the package and looks like a sticker. These stickers works as a check that everyone treated the package well. However, there is no MCU and these detectors work on a mechanical principle.
- The principle of any impact detector is simple: Acceleration is measured (even indirectly) and if the value reaches given threshold, detector gives us a message: LED turns on or ampule with paint bursts.
- We will try to make similar detector with accelerometer data.

Impact detector on a package of one unnamed 3D printer :)

Impact detector: Tasks

- It will be really easy this time
- Everything we need is to read the values from accelerometer a and if one of them reaches treshold, we will return true from then

- In `accelerometer.cpp`

```
bool Accelerometer::detektor_narazu(float treshold,
                                    bool reset)
{
  /* static variable preserves its value to the next
   * function call*/
  static bool impact = false;
  if(reset) /* we reset the memory if it is needed */
  {
    impact = false;
  }
  /* TODO read the data*/
  vector<float> a = {0,0,0};
  /* TODO if any of the three axes register exceed threshold,
   * change the impact variable to true */
  /* We return the current value of impact variable */
  return impact;
}
```

- Reading the value is simple:

```
vector<float> a = this->get_acceleration();
```

- Comparing the components of the vector (vector.x, vector.y and vector.z) is simple, too. However, we need to add 10 to z component, to compensate gravity. `abs()` function is also useful. the impact will be registered even when it is in the opposite direction and the acceleration is negative

```
/* 10 +-= g*/
if(... data.y > treshold || data.z > treshold+10)
  {/* Do...*/}
```

```
bool Accelerometer::detektor_narazu(float treshold,
                                    bool reset)
{
  static bool impact = false;
  if(reset)
  {
    impact = false;
  }
  vector<float> a = this->get_acceleration();
  /* +10 u osy z je kompenzace gravitace */
  if(abs(a.x) > treshold || abs(a.y) > treshold
     || abs(a.z) > treshold+10)
  {
    impact = true;
  }
  return impact;
}
```

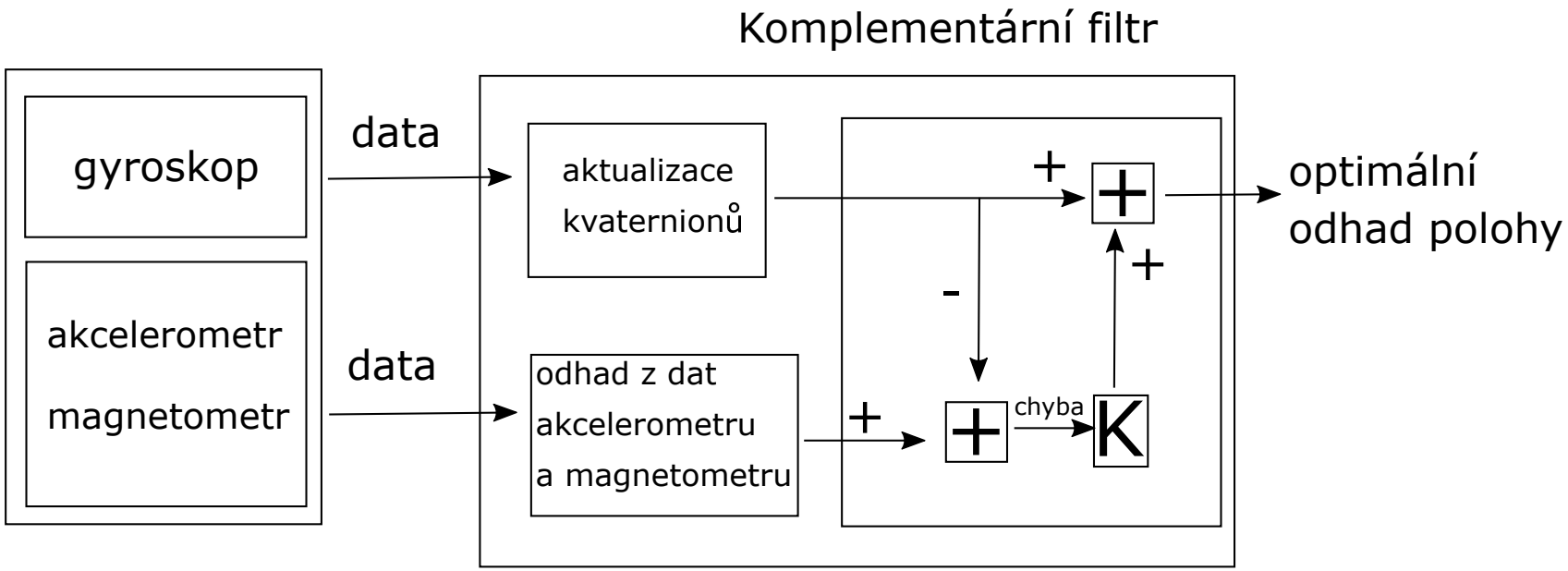
```
void loop()
{
...
/*1.2 good starting value, but you can make experiments :)*/
if(accel->impact_detector(treshold=1.2) == true)
{
    digitalWrite(LED_BUILTIN, HIGH);
}
else
{
    digitalWrite(LED_BUILTIN, LOW);
}

...
}
```

AHRS

How does AHRS work?

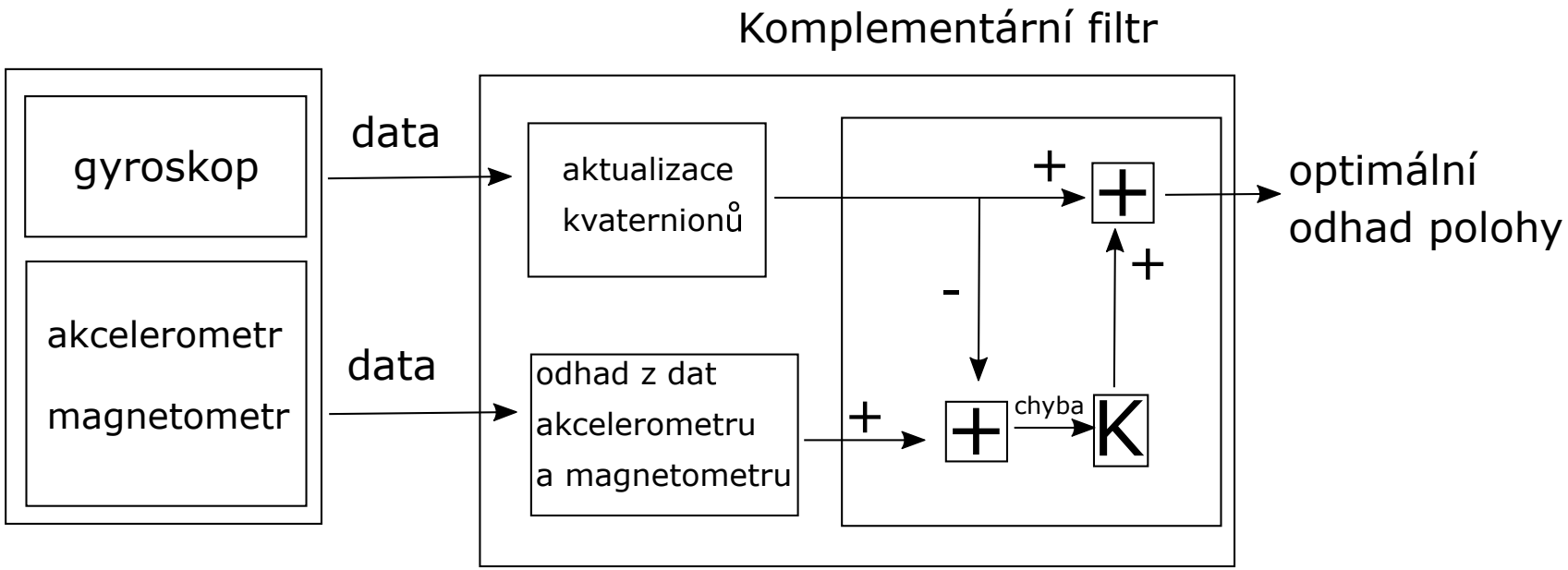
- AHRS in Trilobot is based on Complementary filter (picture and description in the next slide)
- Very briefly:
  - Gyroscope data are used for short–term prediction
  - In long–term horizon, the gyroscope data are modified by the data from accelerometer and magnetometer. The reason is following: gyroscopes has a tendency to drift – they measure some movement even when robot is still
  - Magnetometer serves for better heading – it points to magnetic north, which should not move (however, there is plenty of metal here, so it is not entirely true)
- Functionality depends on first calibration and many other things
- Current implementation of AHRS does not work very well

- The gyroscope data are converted to *quaternions* – 4 numbers. Mathematically it is extension of complex numbers, in our case, the meaning is following: first three numbers sets the axis of rotation and the last one is the angle of rotation around that axis.
- Quaternions are converted to the position estimation.



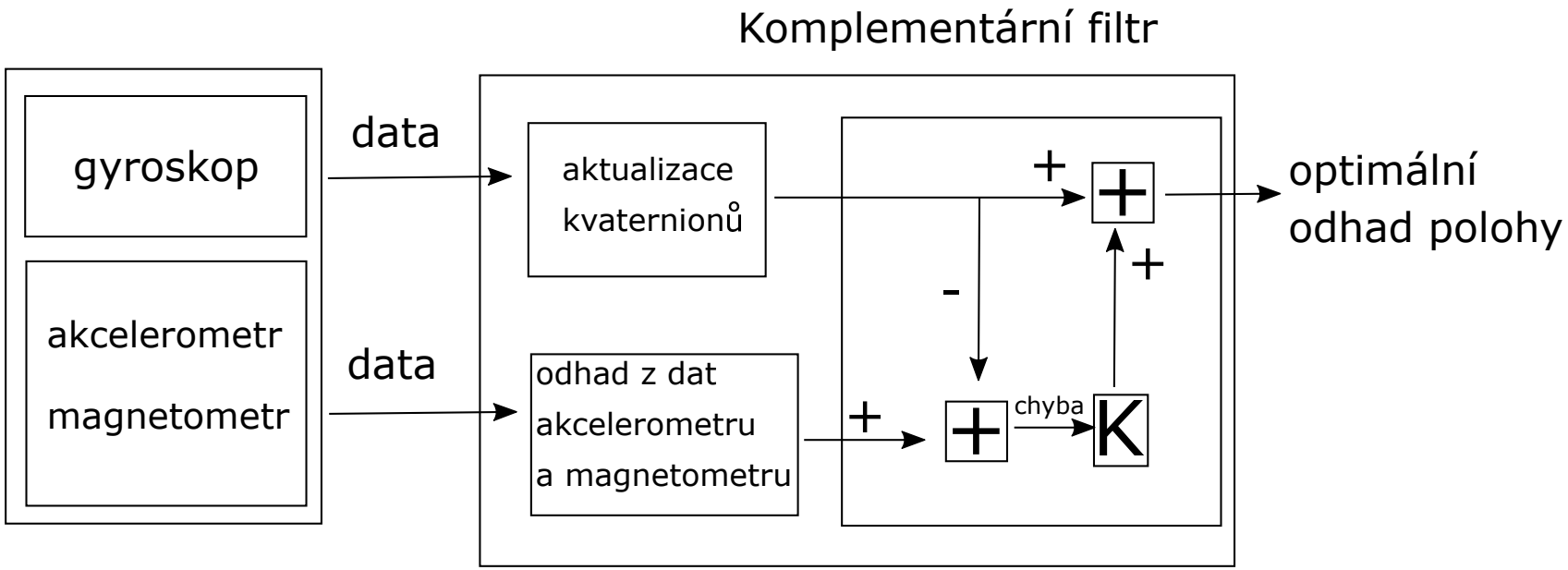
How does AHRS work: Accelerometer and magnetometer

- Accelerometer and magnetometer data are converted to the position estimation by using quite a difficult goniometry.
- Low pass is used – very fast changes are filtered out



How does AHRS work: Complementary filter

- Error is computed – the difference between the prediction from gyroscope and the one from accelerometer and magnetometer
- The K parameter indicates, how much the correction will take effect. The current value is K = 0.02
- If error∗K is not too big, it will be added to gyroscope data and the result will be considered the optimal prediction
  - However, there are some issues around the angle of 0 °. One method can indicate the angle slightly above 0 and the other one can indicate something around 360. When the error∗K is too high, it will be thrown away.



The end