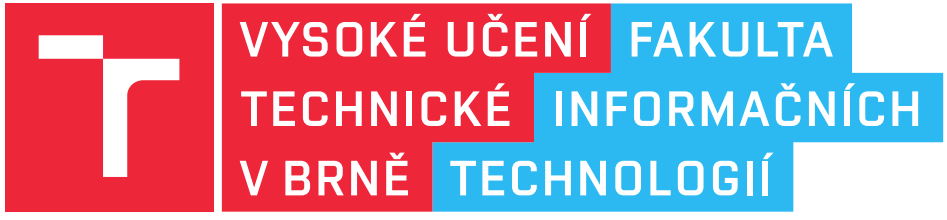
Tutoriál pro robota Trilobot

Jan Beran

Fakulta informačních technologií Vysokého učení technického v Brně
Božetěchova 1/2. 612 66 Brno – Královo Pole

xberan43@stud.fit.vutbr.cz



May 25, 2020

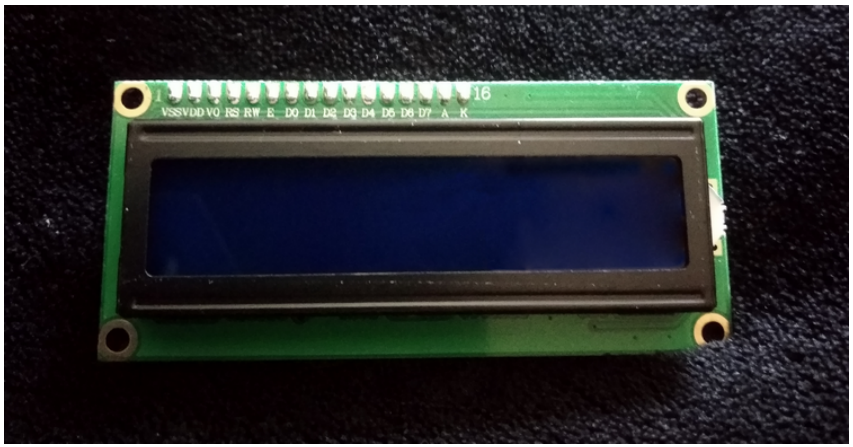
Overview

Úvod

- Původně robot pro akademický vývoj
- Několikrát přestavován
- Na obrázku jeho původní podoba
- Současná verze založená na Arduinu leží před vámi
- Pro potřeby tutoriálu je sestavený a není nutné cokoli fyzicky zapojovat
- Fun fact: původně jezdil opačně :)

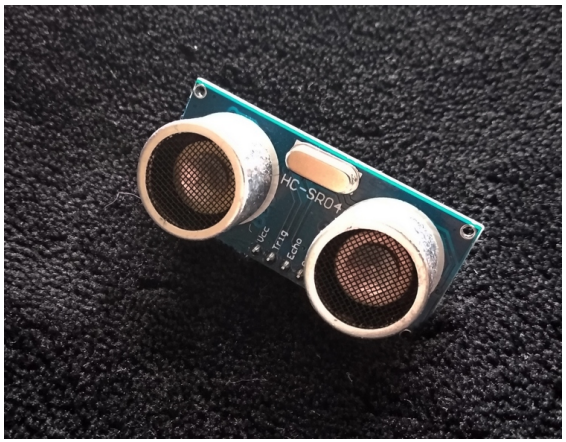
Komponenty Trilobota

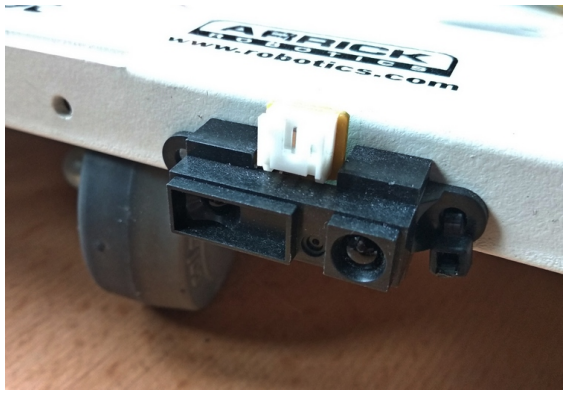
- Klasický 16x2 LCD displej připojený přes I^2C kvůli zjednodušení
- Možno ovládat přes objekt `display` nebo přes knihovnu `LiquidCrystalI2C.h`



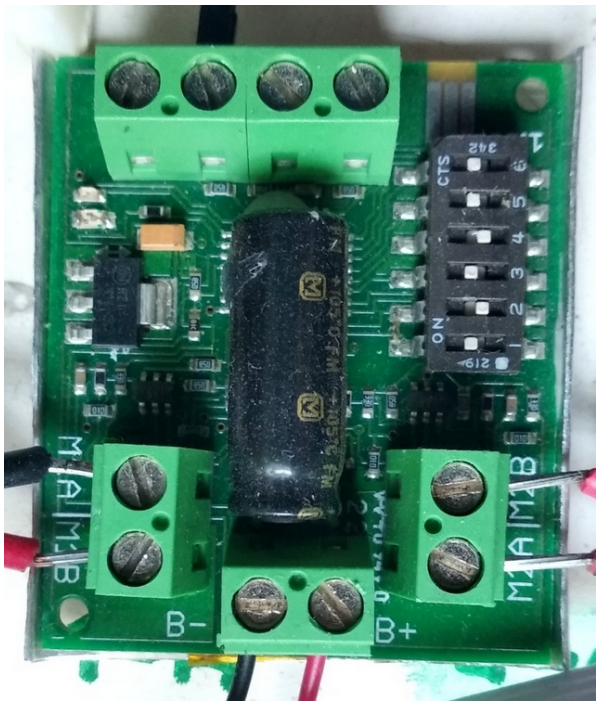
Senzory vzdálenosti

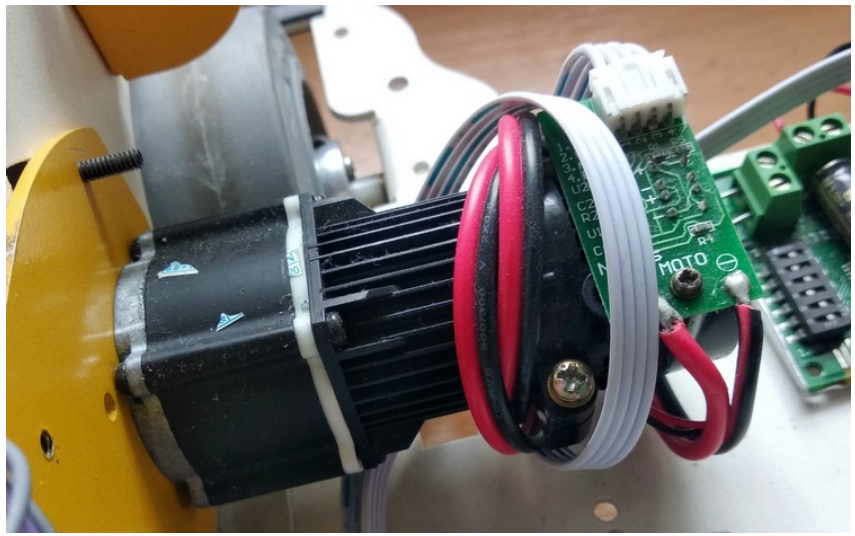
- Ultrazvukové HC–SR04 a SRF–08, infračervený Sharp
- Každý se hodí pro něco trochu jiného
- Ovládají se přes objekty `hcsr04`, `srf08` a `sharp`





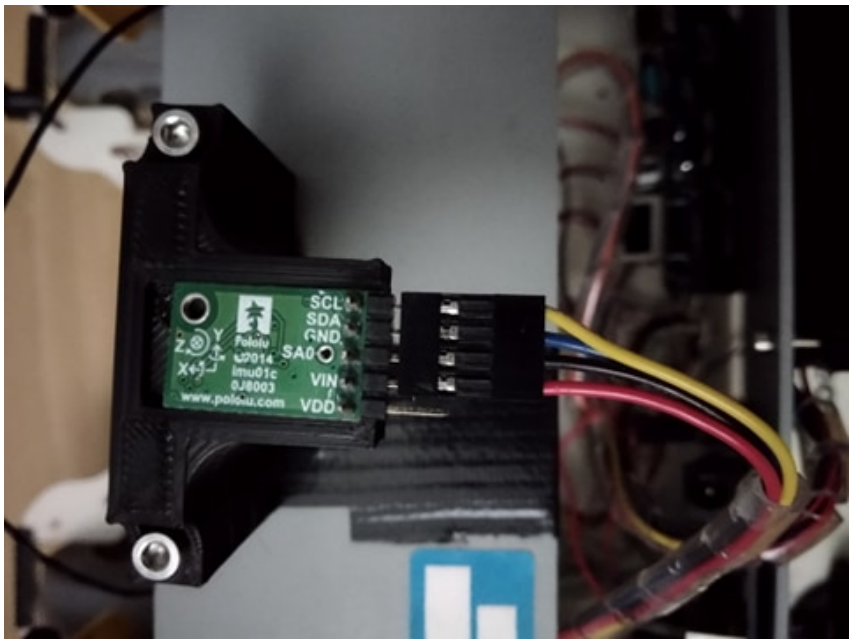
- K pohonu slouží dva motory Faulhaber řízené deskou Sabertooth
- Komunikace mezi Arduinem a Sabertooth probíhá přes seriovou linku `Serial1`
- Zpětná vazba o ujeté vzdálenosti přes rotační enkodéry
- Pro jejich ovládání slouží objekt `motors`





IMU – gyroskop, akcelerometr, magnetometr

- Tři senzory v jednom:
  - Gyroskop: měří úhlovou rychlost
  - Akcelerometr: Měří zrychlení
  - Magnetometr: Měří intenzitu magnetického pole (kompas)
- Data z IMU slouží pro zjišťování polohy – tu počítá AHRS (poziční systém)
- Senzor musí být dál od Trilobota, jinak je jeho magnetometr nepoužitelný (Trilobot je z kovu)
- Každý ze senzorů lze ovládat svým objektem: `gyro`, `accel` nebo `mag`



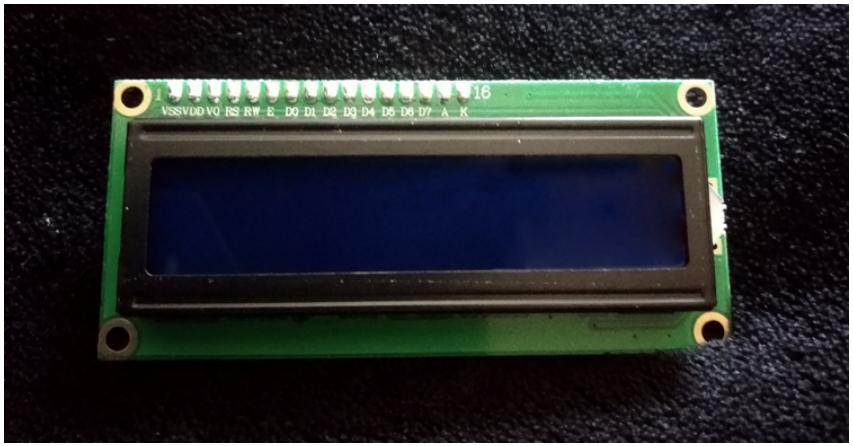
Před začátkem tutoriálu

- Zkratka **TODO** označuje části kódu, které bude třeba doplnit (z EN *To Do*)
- Číslo ve formátu **`0x00`**:
  - Číslo v hexadecimální soustavě
  - V IT se používá běžně, ale není to povinné
  - Chcete–li raději používat čísla v desítkové soustavě, bez problémů je můžete zaměnit, jen nezapomínejte na převod (převodníky na Google)!
- Funkce pro doplnění mají **český** název pro snadnou identifikaci, ostatní kód je psaný **anglicky** – pokud si nejste jistí angličtinou, své názvy proměnných pište česky bez diakritiky, počítač bude rozumět :)

Displej

Displej 16x2

- 16 znaků, 2 řádky
- Připojen přes I^2C (zatím nejsou nutné detaily, bude rozebráno později)
- LCD je relativně pomalá součástka – nezvládne se překreslovat 1000x za vteřinu! – musí se podle toho uzpůsobit kód
- Ovládání:
  - Vlastní knihovna `display.h` a třída `Display`
  - Knihovna `LiquidCrystal_I2C.h`



- Velmi jednoduchý wrapper* pro výpis např. dat ze senzorů bez znalosti práce s displejem
- Objekt třídy `Display` (defaultně pojmenovaný `display`) má tři funkce:

```
  #include <display.h> /* Knihovna pro praci s displejem*/
  ...
  /** Funkce smazou dany radek a vytisknou data*/
  display->print_first_line(to_print);
  display->print_second_line(to_print);
  display->clear(); /* Smaze cely displej */
 }
```

*wrapper = třída, která obaluje jinou třídu (v našem případě `LiquidCrystal_I2C`) a umožňuje ji používat jinak. V našem případě jednodušeji.

- Open–source knihovna dostupná na Githubu
- *De facto* standard pro práci s LCD displeji 16x2 a 20x4 přes I^2C – proto je používaná i zde
- Pracuje objektově – stejně jako my
- Stručně (na dalším slajdu příklad):
  - Vytvoříme si objekt displej s adresou danou makrem `DISPLAY_ADDRESS`, 16 znaky na řádek a 2 řádky
  - Nad vytvořeným displejem voláme funkce
  - Kurzor: aktuální místo, kam se vykreslí další znak

```
#include<LiquidCrystal\_I2C.h>
/* Deklarujeme si globalni promennou pro nas displej */
LiquidCrystal_I2C *display;
void setup() {
 /* Inicializace displeje - pokud bychom pouzivali
  * jiny displej, zmenime udaje */
 display = new LiquidCrystal_I2C(DISPLAY_ADDRESS, 16, 2);
 /* Zakladni nastaveni */
 display->init();
 /* Zapnuti podsviceni, ktere je v zakladu vypnute.
  * Muze setrit energii. */
 display->backlight();
 /* Nastaveni kurzoru na zacatek displeje.
  * Muzeme pouziv i v loop(), prvni je hodnota sloupce,
  * druha hodnota radku. Pozor na indexovani od 0 */
 display->setCursor(0,0);
}
```

```
void loop() {
    /* Vymaze displej */
    display->clear();
    /* Vytiskne libovolnou hodnotu na displej.
     * Pro cisla ma druhy parametr - pro cela soustavu,
     * pro desetinna pocet mist*/
    display->print(something);
}
```

Reproduktor

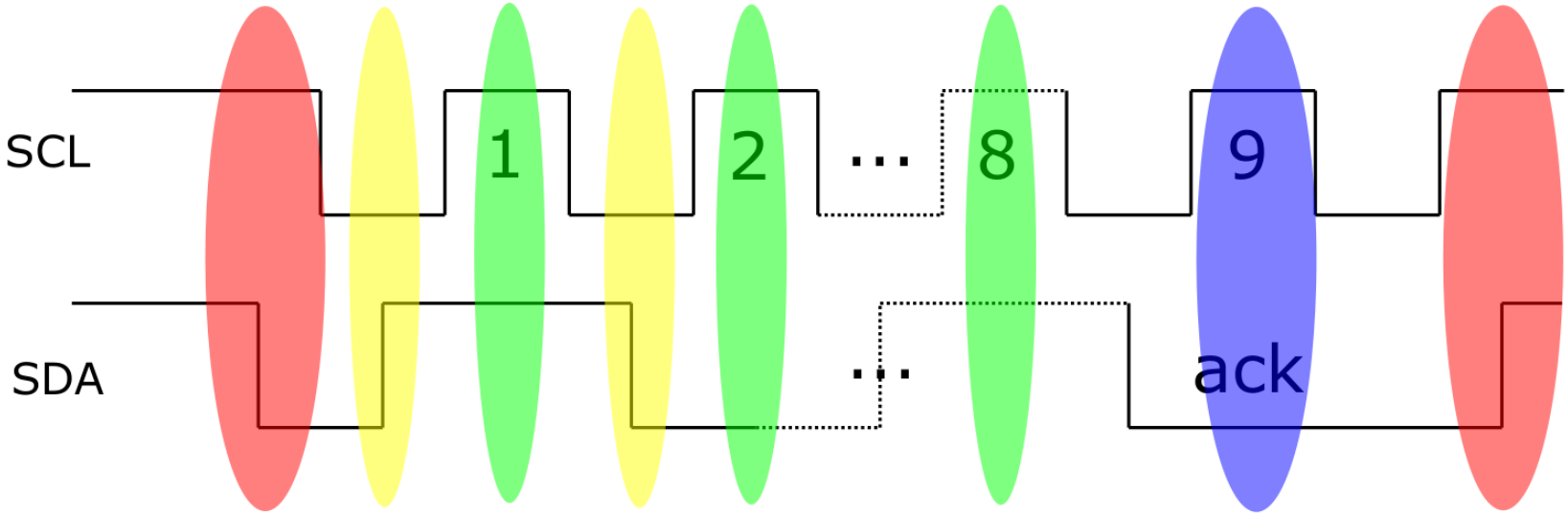
Reproduktor

- Trilobot má zabudovaný i reproduktor
- Jednoduché ovládání zajišťuje objekt typu `Speaker`, defaultně pojmenovaný `speaker`
- Ukázka na dalším slajdu

```
#include <speaker.h>
/* Pred ppouzitim zapneme reproduktor */
speaker->enable();

/* Note: nazev noty - podivejte se do speaker.h pro seznam
 * Duration: cas trvani noty v ms
 * */
speaker->beep(<note>, <duration>);
/* Zahraje Imperial March ze Star Wars :) */
speaker->imperial_march();

/* Po pouziti ho vypneme, jinak bude vydavat piskani */
speaker->disable();
```

I²C

- I²C /TWI je jednou z nejjednodušších a nejrozšířenějších sběrnic
- Používá 4 vodiče: 2 pro napájení, SDA pro data (modrý kabel) a SCL pro synchronizaci (žlutý kabel)
- Master–Slave sběrnice – jedno zařízení Master řídí několik zařízení Slave
- Pro zajímavost: princip I²C komunikace na obrázku:
  - Červeně: START a STOP podmínky
  - Žlutě: Master zapisuje nový bit
  - Zeleně: Slave čte nový bit
  - Modře: ACK, Slave potvrzuje přenos



```
#include <Wire.h> /* Knihovna pro praci s IIC*/
... /* v setup() se vola jen Wire.begin()*/
/*Funkce aktivuje IIC sbernici - volana v setup()*/
Wire.begin();
... /*loop()*/
/*Zapis na zarizeni s adresou <address> do registru <reg>*/
/*Zahajeni komunikace*/
Wire.beginTransmission(<address>);
/*Specifikujeme registr pro zapis*/
Wire.write(<reg>);
/*Posleme data (typu byte)*/
Wire.write(data);
/*Ukoncime komunikaci*/
Wire.endTransmission();
```

```
#include <Wire.h> /* Knihovna pro praci s IIC*/
... /*loop() */
/*Cteni x bajtu ze zarizeni <address> od registru <reg>*/
Wire.beginTransmission(SRF08_ADDRESS);
/*Zapiseme, od ktereho registru chceme cist*/
Wire.write(<reg>);
Wire.endTransmission();

/*Od zarizeni s adresou <address>*/
Wire.requestFrom(<address>, x);
/* Dokud data nejsou pripravena, cekame */
while(!Wire.available());
/* Bezne ovsem data nezahodime :). Wire.read() se tu bude
 * volat ve smycce xkrat*/
Wire.read();
```

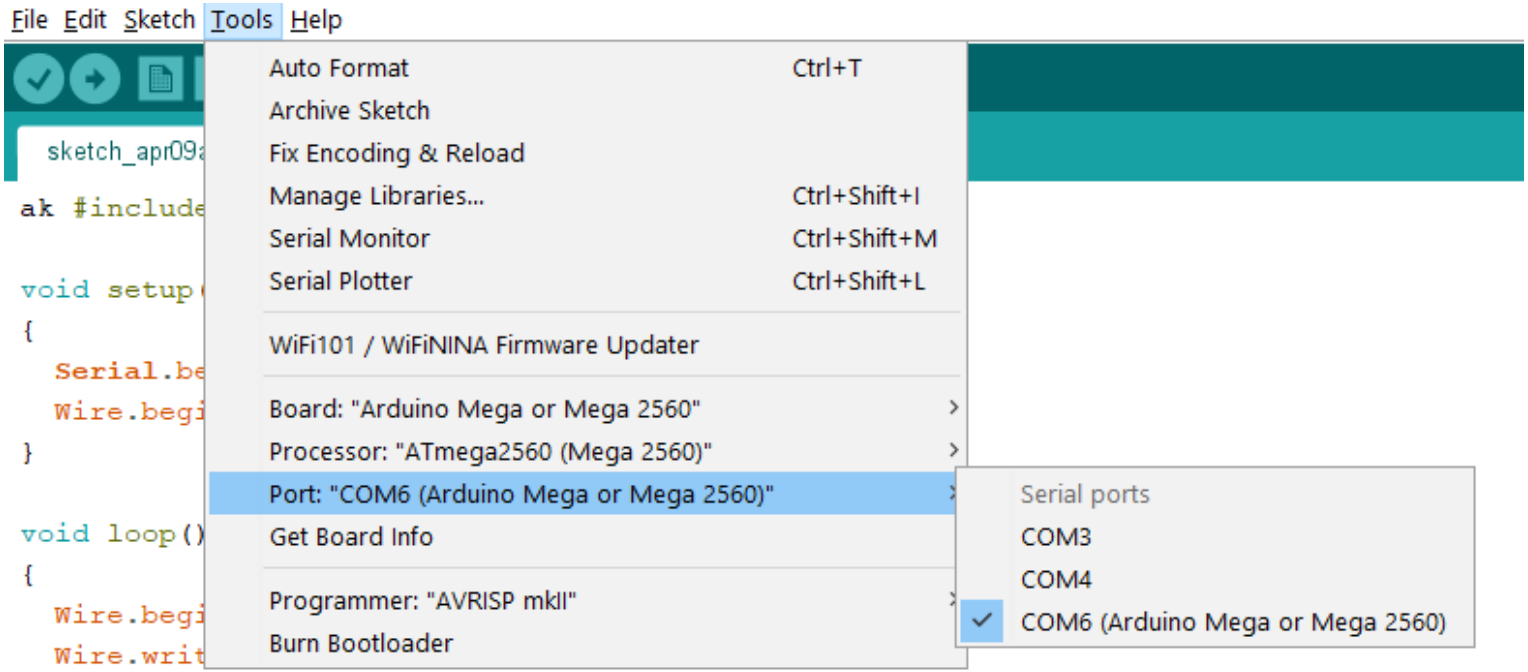
Arduino IDE a připojení Trilobota

- Vývojové prostředí pro psaní a nahrávání kódu do desek Arduino
- Jednoduché – nepodporuje našeptávání apod.
- Pro naprogramování Arduina je třeba vybrat port, na kterém je připojené:
  - Přes *Tools/Port* vybereme správný COM port
  - Pokud nevíme, který to je, otevřeme si nabídku COM portů, odpojíme a znovu připojíme Trilobota. Port, který zmizí a zase se objeví bude ten správný.
- Popis Arduino IDE (obrázek na dalším slajdu):
  - (1): Hlavní část, kam se píše kód
  - (2): Konzole, kde se objevují informace při nahrávání kódu
  - (3): Tlačítka pro (zleva) kontrolu, nahrání, nový soubor, otevření a uložení

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

- Pomocí USB kabelu
- Po úspěšném zapojení by se měl rozsvítit displej a v Arduino IDE (viz dále) se objeví nový COM port
- Přes *Tools/Port* vybereme požadovaný port – je u něj napsáno, že se jedná o Arduino Mega



Test spojení

- Pomocí *File/New* nebo klávesové zkratky ctrl + N si otevřeme nový sketch (zdrojový soubor s koncovkou .ino)
- Po připojení Trilobota k počítači podle předchozího slajdu zkopírujte do funkce `loop()` následující kód a nahrajte ho kliknutím na zelenou šipku vlevo nahoře:

```
void loop(){
    /*pin 13 do logicke jednicky - rozsviti se dioda*/
    digitalWrite(13, HIGH);
    /*Pul vteriny/500 ms pockame */
    delay(500);
    /*Zmenime stav na logickou nulu - dioda zhasne*/
    digitalWrite(13, LOW);
    /*Opet pockame 500 ms*/
    delay(500);
}
```

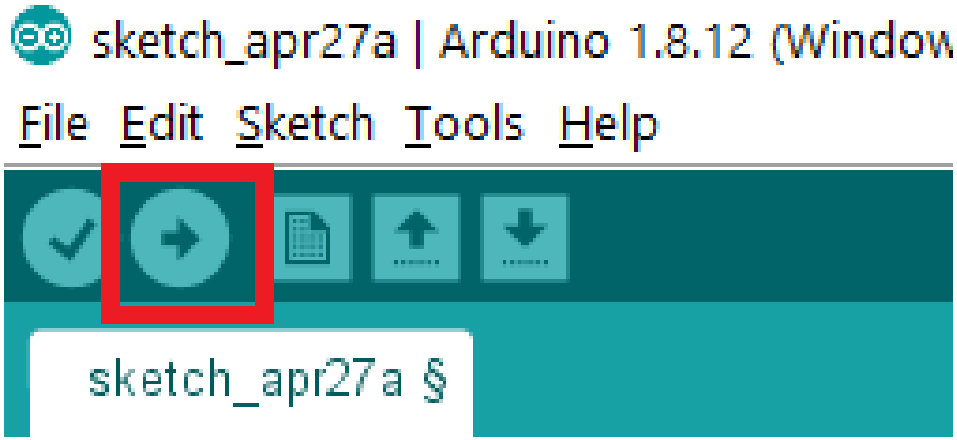
- Pokud je vše v pořádku, měla by začít blikat dioda vedle displeje

Úvod do programování Arduina a OOP

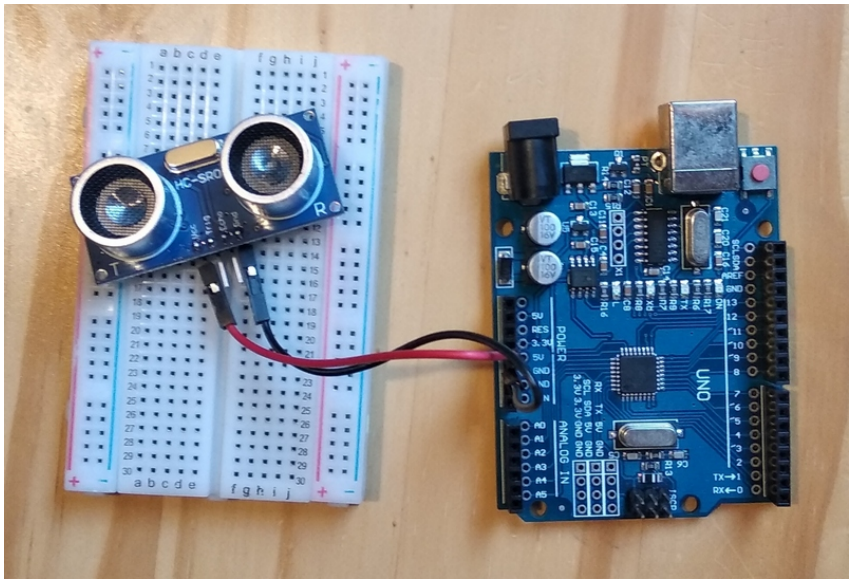
- Arduino se programuje v jazyce Wiring – plně kompatibilní s C++
- V programu se vždy nachází dvě funkce:
  - `setup()`: Tento kód se volá jen jednou, a to při spuštění nebo resetu Arduina
  - `loop()`: Kód v této funkci se volá opakovaně stále dokola – je to nekonečná smyčka

```
void setup()
{
/* Tento kod se provede jen jednou - treba nastaveni */
}
void loop()
{
/* Tento kod se bude volat opakovane
 * - treba cteni ze senzoru */
}
```

- Kliknutím na zelenou šipku (viz obrázek) se kód přeloží a nahraje do Arduina nebo Arduino IDE oznámí, že nastala chyba
- Arduino IDE pravděpodobně bude chtít před přeložením kód uložit – není to nutné, ale doporučené
- Pokud nastala syntaktická chyba, Arduino IDE oznámí, kde



- Pomocí nepájivého pole a DuPont kabelů
- Kabel se nasadí na pin na senzoru a poté se přes nepájivé pole nebo přímo připojí do Arduina



- OOP je způsob programování, kdy se vytvářejí *objekty*
- *Objekty* v kódu pak odpovídají objektům reálného světa ve dvou věcech:
  - Charakteristikou – atributy: barva psa nebo I^2C adresa konkrétního senzoru
  - Vlastnostmi – metodami: pes může štěkat nebo senzor měřit
- *Objekty* mají šablony, podle kterých se tvoří – třídy
  - Například třída `Pes` je šablonou pro konkrétní objekt psa `rex`

- Vše si ukážeme na objektu `rex` a `display`. Ten druhý je i v kódu a dá se použít
- Pokud chceme zavolat metodu na konkrétním objektu, používáme následující syntaxi:

```
jmeno_objektu->nazev_metody(parametry);
```

- Kdybychom chtěli, aby náš `rex` zaštěkal, napíšeme:

```
rex->zastekej();
```

- A pro vytisknutí řádku na displej napíšeme:

```
display->print_first_line("Ahoj svete!");
```

- Ve zkratce: Volání metody se podobá volání funkce, jen před ni připíšeme název objektu a šipku `->`

Tutoriál

Fotosenzor

Fyzikální princip:

- Fotosenzor je neobohacený polovodič –> proud vede velmi špatně (téměř vůbec)
- Světlo/fotony ale dodají energii elektronům ve valenční vrstvě –> vznikají páry elektron–díra
- Tyto páry již vedou proud
- Čím více světla, tím více párů a tím více proudu

Fotosenzor

- Většinou maličký senzor, rozpoznatelný podle charakteristických kliček
- V našem případě součást senzoru SRF–08

- Otestovat, zda jsou Trilobot a Arduino IDE správně připojené
- Do kostry funkce `SRF08::zmer_svetlo()` (k nalezení v souboru `srf08.cpp` doplnit:
  - Číslo registru, ve kterém SRF–08 ukládá hodnotu z fotočidla (viz dokumentaci, strana 4, adresa je ve sloupci *Location*)
  - Přečíst hodnotu přes sběrnici I^2C a vrátit ji (viz I^2C)
  - Vypsat hodnotu na displej (pomocí `display->print_first_line()`)
  - (nepovinné) Rozsvítit LED při nízké hladině světla

```
byte SRF08::zmer_svetlo()
{   this->set_measurement(); /* Zahajeni mereni */
    /*Pozadame SRF-08 o informaci
    pocinajici REGISTREM FOTOCIDLA*/
    Wire.beginTransmission(SRF08_ADDRESS);
    /* 0x00 je jen zastupny znak, aby sel kod prelozit */
    Wire.write(0x00/*TODO REGISTR FOTOCIDLA*/);
    Wire.endTransmission();

    /*Pozadame pouze o jeden bajt...*/
    Wire.requestFrom(SRF08_ADDRESS, 1);
    /*... a pockame, dokud nebude dostupny*/
    while(Wire.available() < 0);
    /*TODO: precist hodnotu a vratit ji.
     * Nasledujici radky pouze zajisti, ze se kod
     * zkompiluje, nakonec vracejte namerenou hodnotu.*/
    byte light_intensity  = 0;
    return light_intensity ;
}
```

- Adresa registru s informací z fotočidla je **0x01**
- Čtení z I^2C probíhá pomocí

```
Wire.read();
/*Funkce vraci datovy typ byte*/
```

- Na displej vypíšeme data přes

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

- Teď už stačí pouze napsat vše do funkce.

```
byte SRF08::zmer_svetlo()
{   this->set_measurement(); /* Zahajeni mereni */
    /*Pozadame SRF-08 o informaci
    pocinajici REGISTREM FOTOCIDLA*/
    Wire.beginTransmission(SRF08_ADDRESS);
    Wire.write(0x01);
    Wire.endTransmission();

    /*Pozadame pouze o jeden bajt...*/
    Wire.requestFrom(SRF08_ADDRESS, 1);
    /*... a pockame, dokud nebude dostupny*/
    while(Wire.available() < 0);

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

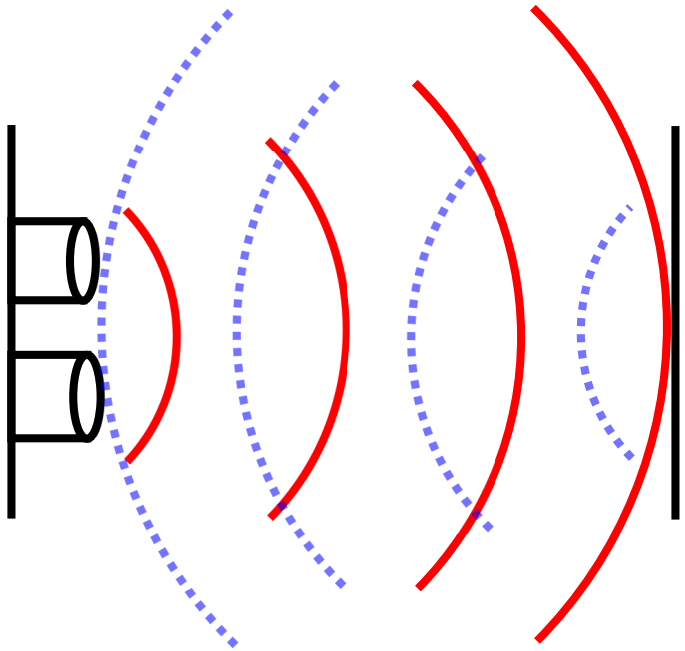
Měříme vzdálenost

V této kapitole si vysvětlíme:

- Jak můžeme měřit vzdálenost
- Měření pomocí senzorů HC–SR04, SRF–08 a Sharp–GP2D120

Jak měříme vzdálenost

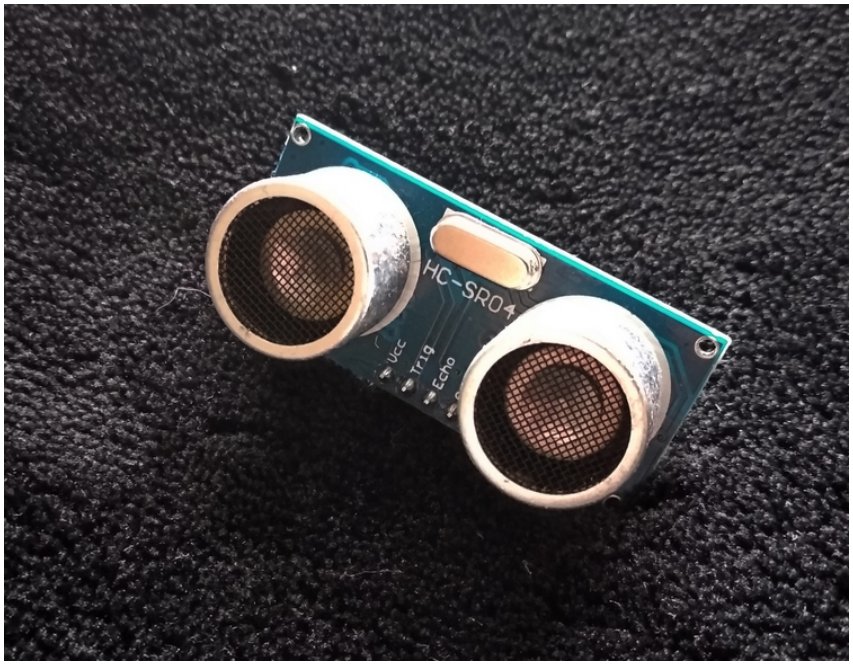
- Pravítkem :) – to roboti nemohou
- Pokud se robot pohybuje, tak pomocí otáček kol. To je ale vcelku nepřesné – prokluzování a podobně
- Pomocí pohybových senzorů – měří ujetou vzdálenost, ne vzdálenost od překážky
- Bezdrátově – odrazem nějaké vlny (zvuk, světlo…)
  - A to si vyzkoušíme i my na všech třech senzorech
  - Princip vidíme na obrázku: červené vlny vysílá senzor, modré (odražené) zachytává. Z uplynulého času mezi vysláním a přijetím a rychlosti vlny lze spočítat vzdálenost



HC_SR04

HC_SR04: Úvod

- Ultrazvukový senzor pro měření vzdálenosti
- 4 vodiče: 2x napájení, `ECHO_PIN` a `TRIGGER_PIN` (makra)
- Dokumentace



HC_SR04: Princip

- Když je na `TRIGEER_PIN` alespoň `10` mikrosekund stav HIGH, senzor začne měřit
- Dokud se nevrátí odražená vlna, drží `ECHO_PIN` ve stavu HIGH
- Sotva dorazí ozvěna, pin se vrátí do stavu LOW
- Hodnota v mikrosekundách se musí převést na vzdálenost pomocí rychlosti zvuku: **`340 m/s neboli 0.0343cm/us`**

- Doplnit kostru funkce `HCSR04::zmer_vzdalenost()`:
  - Na `TRIGGER_PIN` nastavit hodnotu HIGH na 10 mikrosekund
    Senzor vyšle zvukovou vlnu a čeká na její odraz
  - Naslouchat na `ECHO_PIN` a získat délku pulzu v mikrosekundách
    Sotva se vlna vrátí, senzor ukončí pulz
  - Převést mikrosekundy na centimetry či jinou vhodnou jednotku (pomocí rychlosti zvuku)
    Spokojíme se s hodnotou 340 m/s (nebo 0.0343cm/us, což se nám hodí víc)

```
long HCSR04::zmer\_vzdalenost()
{/* Pro jistotu na chvili nastavime stav LOW */
    digitalWrite(TRIGGER_PIN, LOW);
    delayMicroseconds(2);
    /* TODO: Nastavit TRIGGER_PIN do stavu HIGH na 10
     * mikrosekund, pote ho vratit do LOW*/

     /*TODO: pomoci funkce pulseIn(pin, stav) zjistit, jak
      * dlouho bude na ECHO_PIN stav HIGH  a prevest
      * cas na vzdalenost*/
    long return_value = 0;
    return return_value;
}
```

- Na `TRIGGER_PIN` nastavíme HIGH na 10 mikrosekund pomocí

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

- Na `ECHO_PIN` naslouchám pomocí funkce `pulseIn`:

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

- Převod z času na vdálenost, známe–li rychlost, je už hračka :) (s = v*t)

```
long HCSR04::zmer_vzdalenost()
{/* Pro jistotu na chvili nastavime stav LOW */
    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

- Stejný princip jako u HC_SR04
- Komunikace přes I^2C
- Dokumentace
- Kromě měření vzdálenosti i senzor světla
- Má mnohem podrobnější nastavení, nám ale stačí základní

- Do registru s adresou `0x00` odeslat příkaz měření v dané jednotce
  - `0x50`: palce
  - `0x51`: centimetry
  - `0x52`: mikrosekundy
- Počkat, než se měření provede. V dokumentaci se píše, že stačí 60 ms, občas je ale potřeba více.
- Vyžádat si první hodnotu přes I^2C :
  - První hodnota není v prvním registru (ale v registrech 2 a 3)
  - Je rozdělena na horní a spodní bajt, horní je v registru 2, spodní v registru 3
  - Musíme je spojit

- Doplnit funkci `SRF08::zmer_vzdalenost()`:
  - Odeslat do registru `0` správnou jednotku z předešlého slajdu. Nejdříve se přes I^2C odešle adresa registru, poté jeho obsah
  - Počkat `100` milisekund (`60` ne vždycky stačí)
  - Přečíst hodnotu z registrů 2 a 3

  - Spojit hodnoty z registrů 2 a 3 (horní a dolní bajt)

- Doplnit funkci `SRF08::zmer_vzdalenost()`:
  - Odeslat do registru `0` správnou jednotku z předešlého slajdu. Nejdříve se přes I^2C odešle adresa registru, poté jeho obsah
  - Počkat `100` milisekund (`60` ne vždycky stačí)
  - Přečíst hodnotu z registrů 2 a 3
    - Poznámka: SRF–08 Vždy posílá buď všechny registry, počínaje prvním (ne nultým!) nebo mu přes I^2C musíte dát vědět, od kterého registru vás data zajímají. Tohle ale řeší už kostra funkce, takže stačí přečíst dva bajty dat. :)
  - Spojit hodnoty z registrů 2 a 3 (horní a dolní bajt)

- Doplnit funkci `SRF08::zmer_vzdalenost()`:
  - Odeslat do registru `0` správnou jednotku z předešlého slajdu. Nejdříve se přes I^2C odešle adresa registru, poté jeho obsah
  - Počkat 100 milisekund (60 ne vždycky stačí)
  - Přečíst hodnotu z registrů 2 a 3
    - Poznámka: SRF–08 Vždy posílá buď všechny registry, počínaje prvním (ne nultým!) nebo mu přes I^2C musíte dát vědět, od kterého registru vás data zajímají. Tohle ale řeší už kostra funkce, takže stačí přečíst dva bajty dat. :)
  - Spojit hodnoty z registrů 2 a 3 (horní a dolní bajt)
    - Nápověda 1: Je třeba využít bitového posuvu nebo násobení. Také je potřeba větší datový typ (třeba `uint16_t`)

- Doplnit funkci `SRF08::zmer_vzdalenost()`:
  - Odeslat do registru `0` správnou jednotku z předešlého slajdu. Nejdříve se přes I^2C odešle adresa registru, poté jeho obsah
  - Počkat `100` milisekund (`60` ne vždycky stačí)
  - Přečíst hodnotu z registrů 2 a 3
    - Poznámka: SRF–08 Vždy posílá buď všechny registry, počínaje prvním (ne nultým!) nebo mu přes I^2C musíte dát vědět, od kterého registru vás data zajímají. Tohle ale řeší už kostra funkce, takže stačí přečíst dva bajty dat. :)
  - Spojit hodnoty z registrů 2 a 3 (horní a dolní bajt)
    - Nápověda 1: Je třeba využít bitového posuvu nebo násobení. Také je potřeba větší datový typ (třeba `uint16_t`)
    - Nápověda 2: `uint16_t value = (uint16_t)(high<<8)+low;`

```
int SRF08::zmer_vzdalenost(){
  /*Navazeme komunikaci*/
  Wire.beginTransmission(SRF08_ADDRESS);
  /*Napiseme, kam chceme zapisovat...*/
  Wire.write(REG_CMD); /* expanduje do 0x00 */
  /*...a co chceme zapisovat.*/
  Wire.write(0x00/*TODO jednotka*/);
  /*Nakonec ukoncime prenos*/
  Wire.endTransmission();

  /* TODO: pockat 100 ms*/

  /* Zajimaji nas data od registru 0x02*/
  Wire.beginTransmission(SRF08_ADDRESS);
  Wire.write(0x02);
  Wire.endTransmission();
  /* TODO precist data, spojit je a vratit*/

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

- Čekání je brnkačka – stačí delay
- Přečtení hodnoty z registrů také není nic těžkého. (cyklus jen čeká,až budou data připravená)

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

- Spojování dvou bajtů do jednoho 16bit čísla už je trochu složitější

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

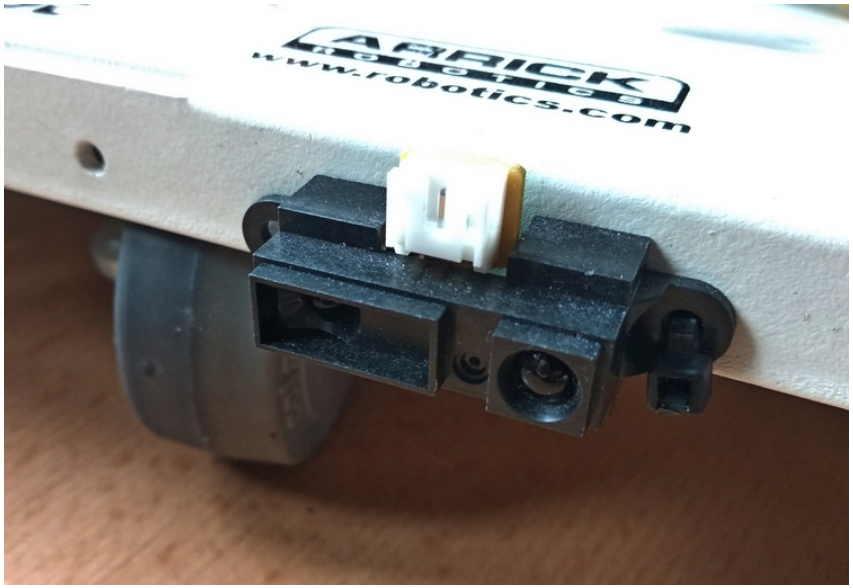
Poznámka: operátor << si představte jako posuvník, který posune číslo vlevo od něj o x řádových míst doleva – takže třeba výsledek 111 << 3 bude 111000 (na volné místo dosadí nuly). A například 101 << 3 + 101 = 101000+101 = 101101 – úplně stejný princip používáme i my.

```
int SRF08::zmer_vzdalenost()
{
  Wire.beginTransmission(SRF08_ADDRESS);
  Wire.write(REG_CMD); /* expanduje do 0x00 */
  Wire.write(0x51); /* centimetry*/
  Wire.endTransmission();
  delay(100); /* cekani */
  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

Sharp–GP2D120: Úvod

- Místo zvuku používá odrazu světla
- Funguje až od 3 centimetrů
- Velmi jednoduchý: Vzdálenost se odvodí z hodnoty napětí na datovém vodiči
- Neexistuje lineární vztah mezi napětím a naměřenou vzdáleností (vizte dokumentaci) – musíme buď aproximovat nebo počítat funkcí
- Dokumentace



- Krátkou chvíli (na poměry Arduina prakticky okamžitě) po přivedení napětí začíná senzor měřit
- Na datovém vodiči (makro `SHARP_PIN`) se objevuje napětí
- Pomocí vztahu z dokumentace lze dopočítat vzdálenost
  - Analyticky: Najdeme funkci, která se co nejvíc blíží vztahu z dokumentace
  - Pro naši potřebu existuje funkce `aproximate()`, používající numerickou aproximaci

- Doplňte funkci `Sharp::zmer_vzdalenost()`:
  - Přečtěte hodnotu na datovém pinu (makro `SHARP_PIN`)
  - Převeďte hodnotu z intervalu `0`–`1024` na interval `0`–`5`
    - Jelikož funkce vrací hodnotu v rozmezí `0`–`1024` a vztah pro výpočet v dokumentaci je z rozmezí `0`–`5`, je nutné jeden z těchto dvou rozsahů převést. A naměřená hodnota je mnohem snazší.
  - Pomocí funkce `aproximate()` získejte vzdálenost v cm

```
float Sharp::zmer_vzdalenost()
{
/*TODO:
 * precist hodnotu na SHARP_PIN
 * ziskanou hodnotu prevest do rozsahu 0-5
 * Pomoci funkce this->aproximate ziskat hodnotu v cm
 * a vratit ji.
  */
  float aproximated_value = 0;
  return aproximated_value;
}
```

- Přečíst hodnotu na SHARP_PIN je snadné:

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

- Při převodu z intervalu 0–1024 do 0–5 už musíme trochu zapřemýšlet, ale v konečném důsledku je to jenom trojčlenka:

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

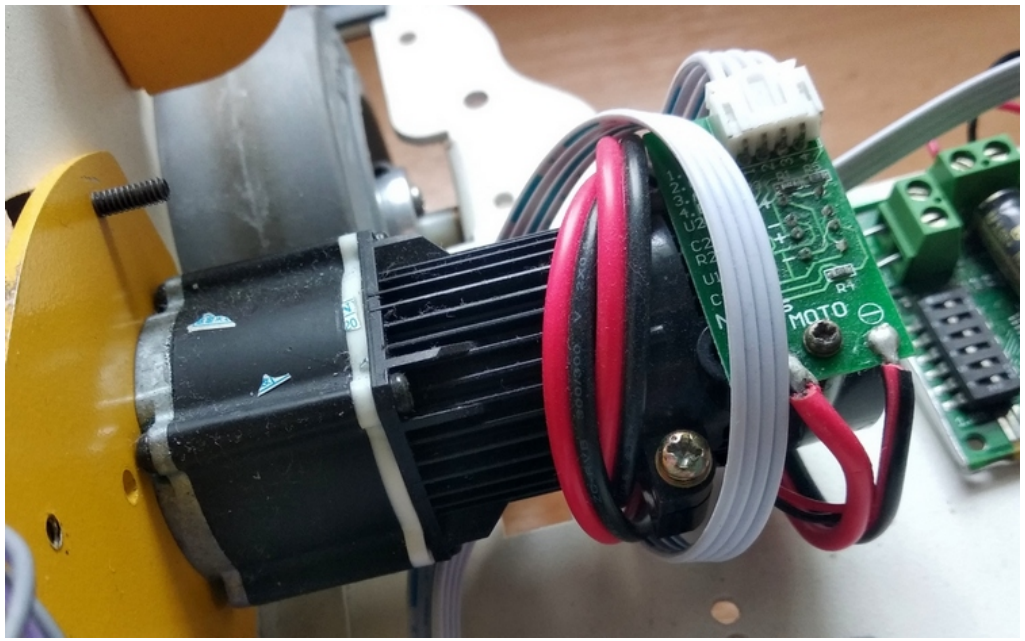
- A zavolání funkce zvládnou všichni :)

```
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;
}
```

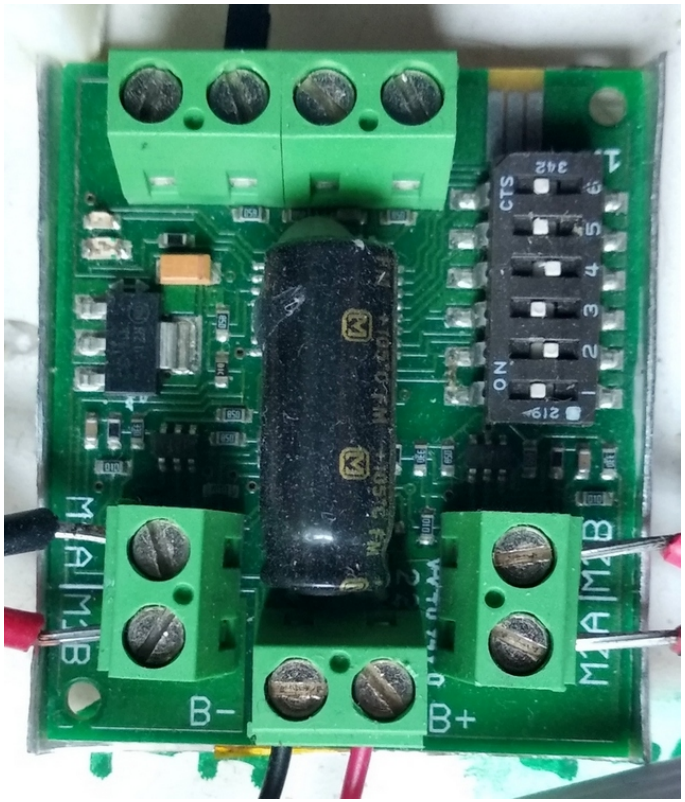
Jezdíme

- Dva motory Faulhaber 16002 s enkodéry (co je to enkodér si řekneme dále)
- Motory řízeny pomocí Sabertooth 2x5 (deska speciálně určená k ovládání motorů)
- Komunikace se Sabertooth přes seriovou linku, ale pouze v jednom směru – stačí jediný vodič
- Dvoukanálové enkodéry zajišťují zpětnou vazbu o pohybu, každý enkodér má jeden kanál připojený na pinu, který podporuje přerušení (co je to přerušení si také povíme dále)



Řízení motorů přes Sabertooth 2x5

- Sabertooth pracuje v tzv. Simplified Serial módu
  - Princip: přes seriovou linku `Serial1` se odešle jeden bajt s danou hodnotou pro pravý a druhý bajt pro levý motor. Tím se levý, resp. pravý motor uvedou do pohybu
- Konkrétní hodnoty na dalším slajdu
- Komunikace je extrémně jednoduchá a pro Trilobota plně dostatečná.



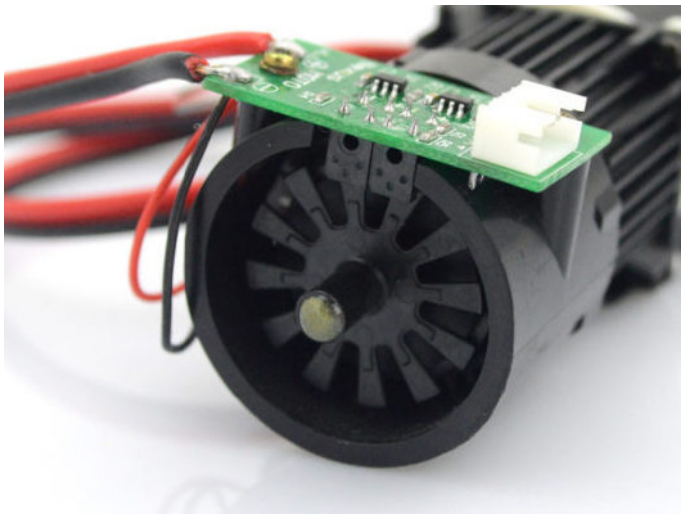
Základní hodnoty pro ovládání motorů

- 0 znamená oba motory stop
- 1–127 ovládá levý motor motor následovně:
  - 1 je naplno vzad
  - 64 je levý motor stop
  - 127 je naplno vpřed
- 128–255 ovládá pravý motor (jsou to čísla pro levý motor, ke kterým je přičteno 127):
  - 128 je naplno vzad
  - 192 je pravý motor stop
  - 255 je naplno vpřed

```
/* Priklad: */
Serial1.write(0x0); /* zastavi oba motory */
...
/* Oba motory naplno vpred - muzeme pouzit i desitkovou
 * soustavu, cislo je v komentarich */
Serial1.write(0x7F); /* = 127 */
Serial1.write(0xFF); /* = 255 */
```

Enkodéry

- Enkodéry periodicky mění signál* v závislosti na tom, jak se jednotlivé motory pohybují. Jedna změna signálu = jeden *krok*
- Signál = změna mezi `0` a 1. Když je před senzorem (na obrázku dva černé výstupky pod zelenou destičkou) lamela, vysílá senzor na daném kanále `0`. Pokud je před senzorem prázdno, vysílá jedničku.
- Pokud známe počet změn signálu za otočku a obvod kola, můžeme spočítat ujetou vzdálenost
- V tutoriálu využit jeden kanál – jeden vodič, na kterém se mění signál



- Jízda rovně
- Zatáčení

Jízda rovně: Úkoly

- Oproti zatáčení výhoda: hodnota pro oba motory bude vždy posunutá o 127
- Princip funkce pro jízdu rovně*:
  - **Vstupy**:
    - Vzdálenost v centimetrech
    - Rychlost v intervalu <−100,100>, kde záporná rychlost znamená jízdu dozadu
  - Vypočítáme, kolik kroků musíme odečíst na enkodéru, aby se kola posunula o danou vzdálenost
  - Hodnotu kontrolujících bajtů nám vrátí funkce `Motors->get_speed_from_percentage()`
  - Odešleme kontrolující bajty přes seriovou linku
  - Počkáme, až robot ujede danou vzdálenost – v kódu je to nekonečný cyklus, který tak úplně nekonečný není
  - Zastavíme motory

Poznámka: v kódu se bude využívat tzv. přerušení/interrupts. Pokud vás zajímá, co to je, podívejte se do pokročilé části učebního textu. Ale pro úspěšné rozjetí Trilobota nic toho nutné není.

*Možná byste vymysleli lepší způsob, jak takovou funkci navrhnout, ale kvůli zachování kompatibility s referenční funkcí budeme uvažovat výše popsané vstupy.

```
void Motors::jed_rovne(int distance, int speed)
{
  int steps_to_go = 0/*TODO potrebne kroky*/;
  byte driving_byte;
  driving_byte = Motors::get_speed_from_percentage(speed);
  attach_interrupts();
  /*TODO odeslat kontrolní bajt do obou motoru*/
  /*Toto si vysvetlime mimo kod*/
  while(1)
  {
    if(steps_left == steps_to_go ||
    steps_right == steps_to_go)
      break;
  }
  /*TODO zastavit motory*/
  detach_interrupts();
}
```

Jak spočítáme potřebné kroky, pokud známe vzdálenost?

- Potřebujeme k tomu dvě hodnoty: počet kroků enkodéru za jednu otočku motoru/kola (jsou v převodovém poměru 1:1) a obvod kola.
- Obě hodnoty jsou v kódu dostupné přes makra `STEPS_ONE_CHANNEL` a `WHEEL_CIRCUIT`
- Přesný výpočet je již otázkou jednoduché úvahy :)

Jak spočítáme potřebné kroky, pokud známe vzdálenost?

- Potřebujeme k tomu dvě hodnoty: počet kroků enkodéru za jednu otočku motoru/kola (jsou v převodovém poměru 1:1) a obvod kola.
- Obě hodnoty jsou v kódu dostupné přes makra `STEPS_ONE_CHANNEL` a `WHEEL_CIRCUIT`
- Přesný výpočet je již otázkou jednoduché úvahy :)

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

Nekonečný cyklus? Ne, přerušení!

- V kódu se vyskytuje něco, co vypadá jako nekonečný cyklus
- Ve skutečnosti ale jde z cyklu vyskočit díky *přerušení*
- Co je to přerušení? Kdy se používá?
  - Přerušení se používá v situaci, kdy je nutné standardní běh kódu *přerušit* (odtud název) a vykonat nějakou jednoduchou činnost, třeba přičíst do čítače, nastavit flag a podobně. Této činnosti se říká obslužná rutina přerušení.
  - Představte si to jako učitele, který přednáší před třídou a jednou za čas se během přednášky přihlásí student a položí otázku bez toho, aniž by se učitel ptal, zda někdo nějaké otázky má. Učitel na ni odpoví a pokračuje tam, kde skončil.
  - V našem případě přerušení aktivujeme funkcí `attach_interrupts()` a deaktivujeme funkcí `detach_interrupts()`. Naše obslužné rutiny pouze přičítají jedničku k proměnným `steps_left` nebo `steps_right`. Časem tedy podmínka pro vyskočení z cyklu začne platit a cyklus skončí.

- Vzorec pro potřebný počet kroků už máme:

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

- Odeslání přes seriovou linku je také jednoduché, stejně jako zastavení:

```
Serial1.write(driving_byte);
Serial1.write(driving_byte+127);
...
Serial1.write(STOP_BYTE);
```

```
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);
  detach_interrupts();
}
```

Zatáčíme

Zatáčíme: Úvod

- Zatáčení se od jízdy vpřed v principu neliší – funkce budou velmi podobné
- Lze implementovat mnoha různými způsoby:
  - Nejjednodušší: motor ve směru zatáčení stojí, druhý motor se pohybuje dopředu
  - Na místě: motor ve směru zatáčení se pohybuje opačně než druhý motor
  - Jako u auta: robot opíše oblouk…
- My zvolíme první – nejjednodušší – variantu

- Princip funkce pro zatáčení:
  - **Vstupy**:
    - Úhel ve stupních z intervalu $< -180, 180 >$, kde kladný úhel bude znamenat zatáčení doprava, záporný zatáčení doleva
    - Rychlost: tentokrát jen od `0` do `100`, dozadu zatáčet nebudeme
    - Opět platí to co u funkce `Motors::jed_rovne()`: vstupy se dají zvolit naprosto jinak a stále dojdeme k funkčnímu výsledku. Ukázaná je jen jedna z mnoha cest. která je zároveň kompatibilní s referenční funkcí.
  - Vypočítáme, kolik kroků musí urazit vnější motor
  - Odešleme informaci o rychlosti motoru (podle směru zatáčení zvolíme pravý nebo levý motor)
  - Motor zastavíme

```
void Motors:zatoc(int angle, int speed)
{
  int steps_to_go = 0/*TODO pocet kroku */;
  attach_interrupts();
  byte driving_byte;
  driving_byte = Motors::get_speed_from_percentage(speed);
 /*TODO zatoceni podle znamenka u parametru angle*/

  while(1)
  {
    if(steps_left == steps_to_go ||
      steps_right == steps_to_go)
      break;
  }
  /*TODO zastavit dany motor */
  detach_interrupts();
}
```

Jak vypočítáme kroky?

- Tentokrát je to trochu složitější – kolo se bude pohybovat po části kružnice
- Pro zjištění, po jak dlouhé části kružnice se bude pohybovat, musíme vyřešit tuto úlohu: Jak velká část kružnice o poloměru `r` náleží úhlu o velikosti α?
- Následně, když budeme znát vzdálenost, budeme moci použít vzorec z funkce `jed_rovne()`
- Řešení na dalším slajdu

Jak vypočítáme kroky?

```
/*
    r = 20, lze pouzit makro TURN_RADIUS
    Obecny princip: Pomer casti kruznice prislusne k uhlu
    k obvodu kruhu je stejny, jako pomer daneho uhlu k uhlu
    plnemu (360 stupnu). Tim dostavame vzdalenost, kterou
    muzeme dosadit do stejneho vzorce, ktery jsme pouzili
    pro jizdu rovne.
    Priklad: obvod = 125 cm, uhel = 60 stupnu
    x je cast kruznice
    60 je 1/6 360 -> x je 1/6 125 -> x += 21
*/
int steps_to_go = (int) (((2 * TURN_RADIUS * PI * angle)
/ 360) / WHEEL_CIRCUIT * STEPS_ONE_CHANNEL);
```

- Správný počet kroků již známe z minulého slajdu – stačí zkopírovat
- Odeslání po seriové lince také již umíme:

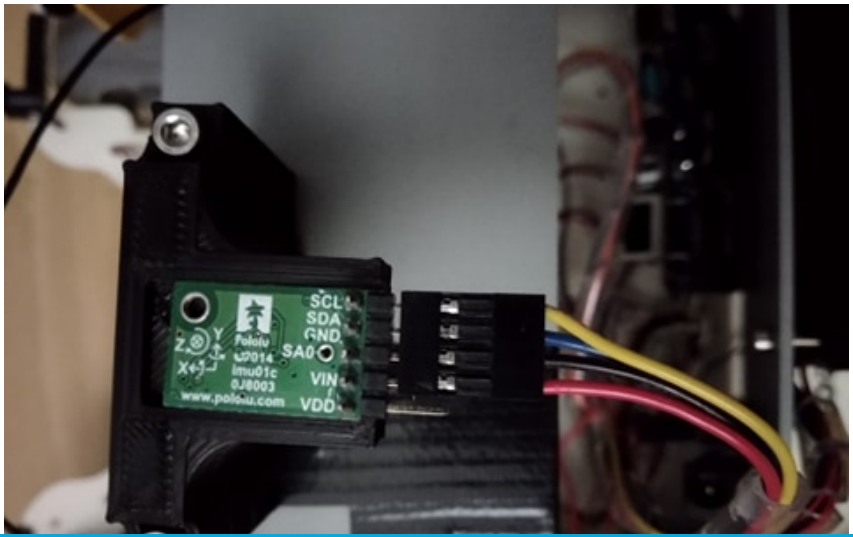
```
Serial1.write(driving_byte);
...
Serial1.write(STOP_BYTE);
```

```
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

Co je to IMU?

- IMU je kombinací tří tříosých senzorů: gyroskopu, magnetometru a akcelerometru
- Můžeme používat data z každého samostatně, ale nejlépe fungují dohromady jako tzv. AHRS*
- Kvůli magnetometru musí být celá IMU nad robotem, aby nebyl tolik rušený
- Z magnetometru si můžeme udělat kompas
- Gyroskop může fungovat jako detektor otřesů
- Akcelerometr použijeme jako detektor nárazu



Co je to IMU?

- IMU je kombinací tří tříosých senzorů: gyroskopu, magnetometru a akcelerometru
- Můžeme používat data z každého samostatně, ale nejlépe fungují dohromady jako tzv. AHRS*
- Kvůli magnetometru musí být celá IMU nad robotem, aby nebyl tolik rušený
- Z magnetometru si můžeme udělat kompas
- Gyroskop může fungovat jako detektor otřesů
- Akcelerometr použijeme jako detektor nárazu

*AHRS – Attitude and heading reference system – slouží k určování polohy a směru robota. Dodává informace o poloze (natočení ve všech třech osách) a další informace. Implementace je ale poměrně složitá a ani Trilobotovův AHRS nefunguje úplně přesně.

Kompas #1

Kompas #1: Úvod

- Kompas vypíše úhel, jaký úhel svírá Trilobot se severem*
- Tento postup je velmi jednoduchý, ale nedává moc přesně výsledky
- Naprosto zanedbáme polohu ve 3D

Kompas #1: Úvod

- Kompas vypíše úhel, jaký úhel svírá Trilobot se severem*
- Tento postup je velmi jednoduchý, ale nedává moc přesně výsledky
- Naprosto zanedbáme polohu ve 3D

* Pro jednoduchost se spokojíme s úhlem od 0 do 180 °

- Doplníme funkci `float Magnetometer::kompas_1()`:
  - Přečteme data z magnetometru
  - Zahodíme složku Z (použijeme jen data pro osu x a y – vodorovnou plochu)
  - Vypočítáme úhel mezi vektorem dat z magnetometru a vektorem `front`

```cpp
float Magnetometer::kompas_1()
{
  /*TODO ziskat data intenzity z magnetometru do prom. nize*/
  vector<int16_t> mag_values = {0,0,0};
  /*Udava, jaky smer je "predek" (osa X - prvni pozice).
   * Jelikoz je pro lepsi pristup senzor "opacne",
   * je hodnota zaporna. */
  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 vynulovat souradnici Z*/
  vector<float> norm_mag_values;
  norm_mag_values = vector_normalize(mag_values);

  float angle = 0;/*TODO spocitat uhel dle navodu*/
  return angle;
}
```

- Data přečteme pomocí:

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

- Vynulování souřadnice je prostě... vynulování souřadnice :)

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

- Úhel mezi vektory známe ze střední školy:
  $\alpha = acos(\frac{mag\_values * front}{|mag\_values| * |front|})$
- V kódu stačí použít funkce `vector_dot(u,v)` pro skalární součin a `vector_abs(v)` pro velikost vektoru. Arkus kosinus je jednoduše `acos()`, ovšem hodnotu vrací v radiánech – stačí vynásobit výsledek zlomkem $\frac{180}{\Pi}$

```
float Magnetometer::kompas_1()
{
 vector<int16_t> mag_values = this->get_intensity();
  /*Udava, jaky smer je "predek" (osa X - prvni pozice).
   * Jelikoz je pro lepsi pristup senzor "opacne",
   *  je hodnota zaporna. */
  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;
```

Kompas #2

- Kompas vypíše úhel, jaký úhel svírá Trilobot se severem*
- Tento postup není nejjednodušší, ale funguje velmi dobře
- Využívá vektorovou aritmetiku

* Pro jednoduchost se spokojíme s úhlem od 0 do 180 °

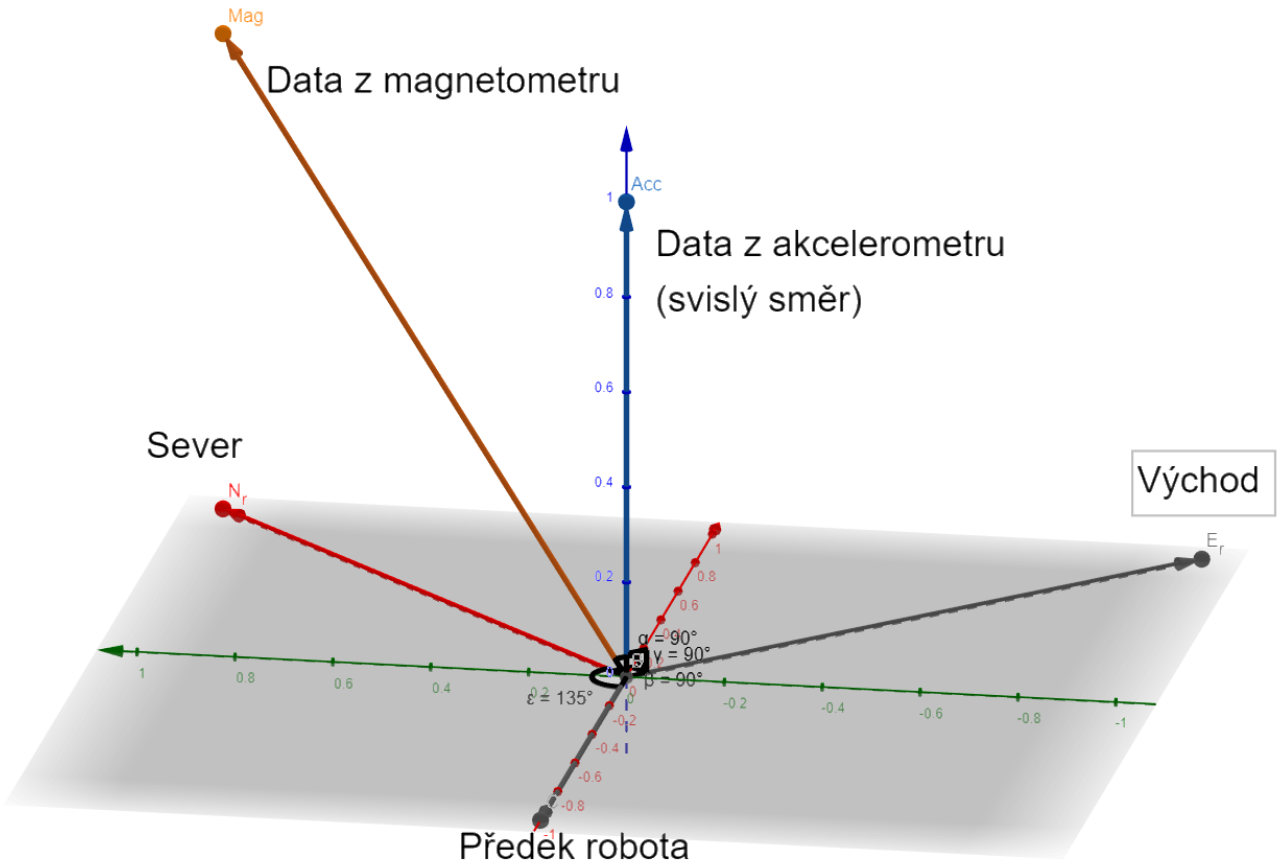
- Kompas vypíše úhel, jaký úhel svírá Trilobot se severem*
- Tento postup není nejjednodušší, ale funguje velmi dobře
- Využívá vektorovou aritmetiku

* Pro jednoduchost se spokojíme s úhlem od 0 do 180 °

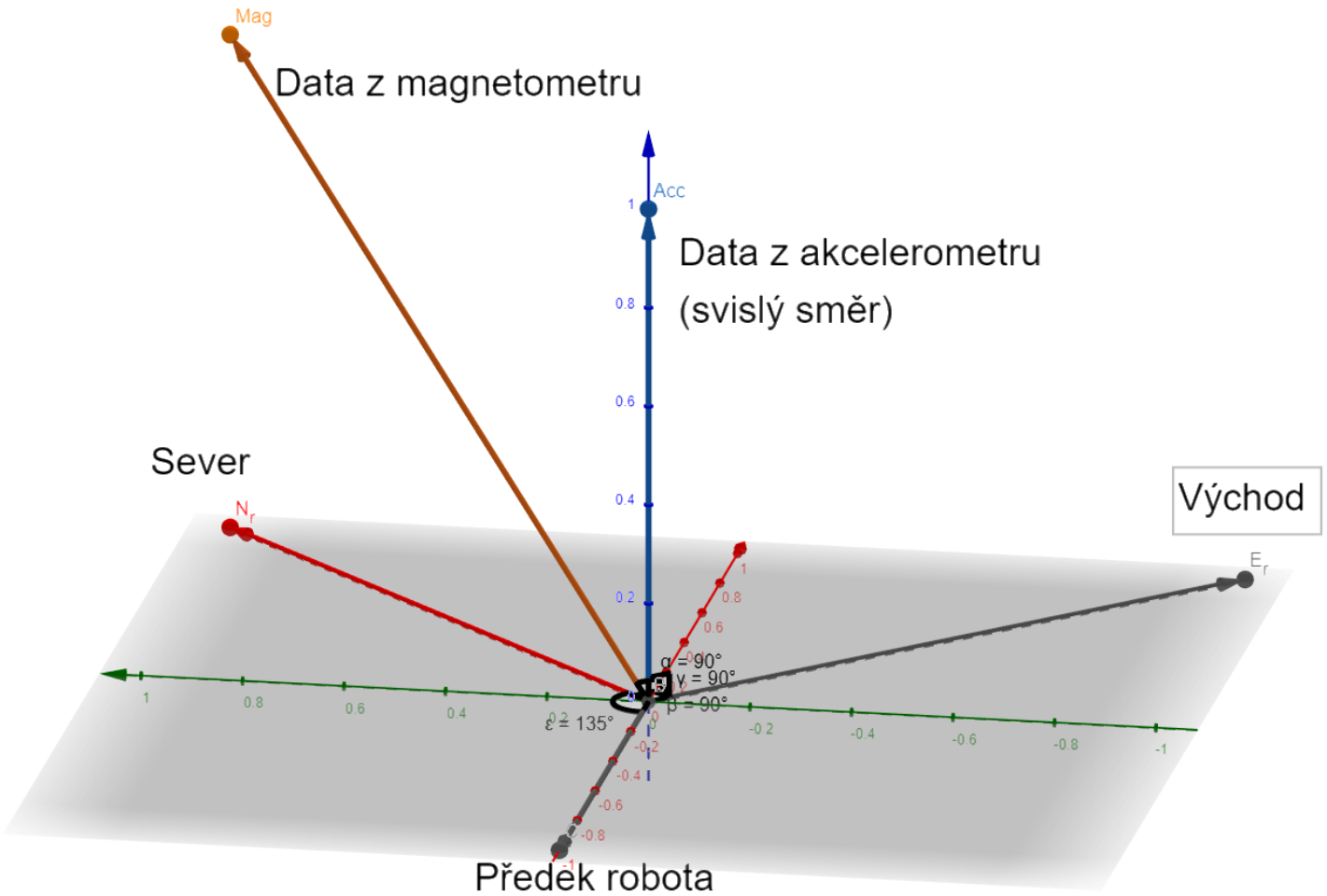
- Doplníme funkci `Magnetometer::kompas_2(Accelerometer accel)`:
  - Z vektoru intenzity magnetického pole, svislého směru a orientace předku Trilobota vypočítáme úhel mezi severem a předkem Trilobota
  - Podrobnější postup si ukážeme na dalších slajdech

```
float Magnetometer::kompas_2(Accelerometer *accel)
{  /*Udava, jaky smer je "predek" (osa X - prvni pozice).
   * Jelikoz je pro lepsi pristup senzor "opacne",
   * je hodnota zaporna. */
  vector<int> front = {-1,0,0};
  vector<int16_t> mag_values = this->get_intensity();
  vector<float> accel_values = accel->get_acceleration();
  /** odecteni offsetu */
  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; /* Vychod */
  vector<float> N; /* Sever*/
  /* TODO: spocitat sever podle navodu */

  float angle = 0; /*TODO uhel mezi vektory N a front*/
  return angle; }
```

- Vektorový součin svislého směru (data z akcelerometru) a dat z magnetometru nám dá vektor východu v rovině.
- Jak víme, že je to východ? Je kolmý na svislý směr a na směr intenzity magnetického pole (který stále ukazuje na sever, jen v prostoru).



- Vektorový součin svislého směru (data z akcelerometru) a východu nám dá vektor severu v rovině = to, co chceme.
- Jak víme, že je to sever v rovině? Je kolmý na východ a na svislý směr.



- Vypočítáme úhel mezi vektory – jedná se jen o přepsání vzorce do kódu, tím se zatěžovat nemusíme :)

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

```
float Magnetometer::kompas_2(Accelerometer *accel)
{
  /***Vypustene radky, co se nezmenily***/

  vector<float> E; /* Vychod */
  vector<float> N; /* Sever*/
  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;
}
```

Detektor otřesů

Detektor otřesů: Úvod

- Používá se na mnoha místech:
  - Detekce zemětřesení – seismometr
  - Kontrola stavu strojů – když se stroj moc třese, pravděpodobně je někde chyba
  - Kontrola křehkých zásilek spolu s detektory nárazu
- My si jeden vytvoříme pomocí dat z gyroskopu

Detektor otřesů: Úkoly

- Pomocí dvou měření z gyroskopu a jejich rozdílu určíme, zda se s Trilobotem třese
- Pokud ano, v `loop()` rozsvítíme diodu

- V souboru gyroscope.cpp

```
bool Gyroscope::detektor_otresu(float treshold)
{
  /* TODO ziskat dvoje data s urcitou casovou prodlevou
  - treba 20 ms */
  vector<float> first = {0,0,0};
  vector<float> second = {0,0,0};
  vector<float> shake = {0,0,0};

  /*Ziskame jejich rozdil */
  shake.x = abs(first.x-second.x);
  shake.y = abs(first.y-second.y);
  shake.z = abs(first.z-second.z);

  /* TODO pokud bude nektery z rozdilu v ose x,y,z vetsi nez
  treshold - hranice "klidu", vratime true, jinak false */
 return false;
}
```

- Data přečteme jednoduše – metodou `get_angular_velocity()` objektu `this` – tohoto objektu:

```
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...)
{/* Udelej...*/}
```

```
 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;
}
```

```
void loop()
{
...
/* 10 je vyzkousena hodnota, ale nebojte
 * se experimentovat :) */
if(gyro->shake_detector(treshold=10) == true)
{
    digitalWrite(LED_BUILTIN, HIGH);
}
else
{
    digitalWrite(LED_BUILTIN, LOW);
}
...
}
```

Detektor nárazu

Detektor nárazu: Úvod

- S detektory nárazu se setkáváme například u přepravních služeb – kontrola, zda se k balíku všichni chovali dobře – zde většinou mechanické detektory, které se poškodí po nárazu určité síly
- Princip je jednoduchý: měří se zrychlení – byť i nepřímo – a pokud překročí nějakou hranici, senzor o tom dá vědět a tento stav si již ponechá
- My si vyzkoušíme podobný detektor vyrobit z akcelerometru

Senzor nárazu na balení jedné nejmenované 3D tiskárny :)

Detektor nárazu: Úkoly

- Tentokrát to bude opravdu jednoduché
- Stačí přečíst data a pokud některá hodnota přesáhne `treshold`, podáme o tom zprávu – třeba rozsvítíme LEDku

- K nalezení v `accelerometer.cpp`

```
bool Accelerometer::detektor_narazu(float treshold,
                                    bool reset)
{
  /* static promenna si zachova ulozenou hodnotu
   * i pri dalsim volani funkce */
  static bool impact = false;
  if(reset) /* pri resetu vyresetujeme i pamet dopadu */
  {
    impact = false;
  }
  /* TODO precteme data*/
  vector<float> a = {0,0,0};
  /* TODO pokud nektera z os prekroci treshold, nastavime
  impact na true */
  return impact; /* vracime aktualni hodnotu impact */
}
```

- Čtení hodnoty je jednoduché:

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

- No a porovnávat jednotlivé složky vektoru zvládneme taky – jen nesmíme zapomenout u osy Z přičíst k tresholdu 10 (neboli gravitační zrychlení). Také se nám vyplatí použít funkci `abs()` – zajistí, že se náraz zaznamená i pokud přijd ez druhé strany

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

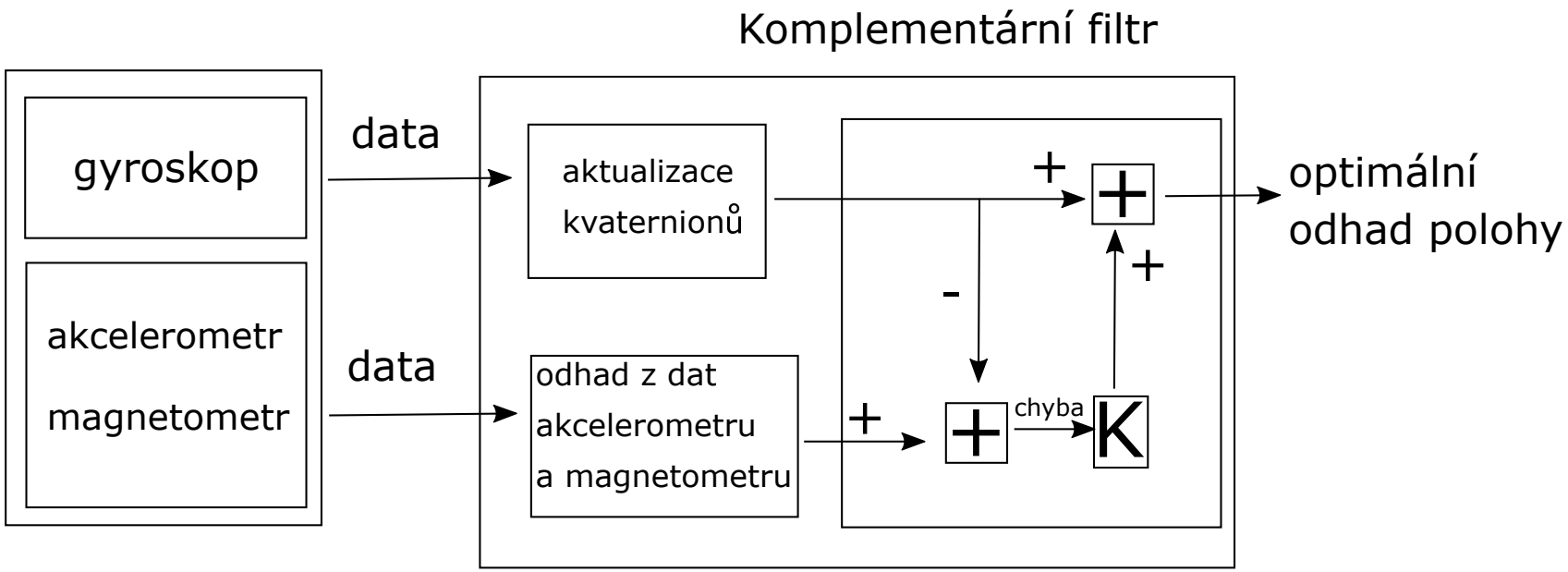
```cpp
bool Accelerometer::detektor_narazu(float treshold,
                                    bool reset)
{
  static bool impact = false;
  if(reset)
  {
    impact = false;
  }
  vector<float> a = this->get_acceleration();
  /* +10 u osy z je kompenzace gravitace */
  if(abs(a.x) > treshold || abs(a.y)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

```
void loop()
{
...
/*1.2 je vyzkousena hodnota, ale muzete experimentovat :)*/
if(accel->impact_detector(treshold=1.2) == true)
{
    digitalWrite(LED_BUILTIN, HIGH);
}
else
{
    digitalWrite(LED_BUILTIN, LOW);
}

...
}
```

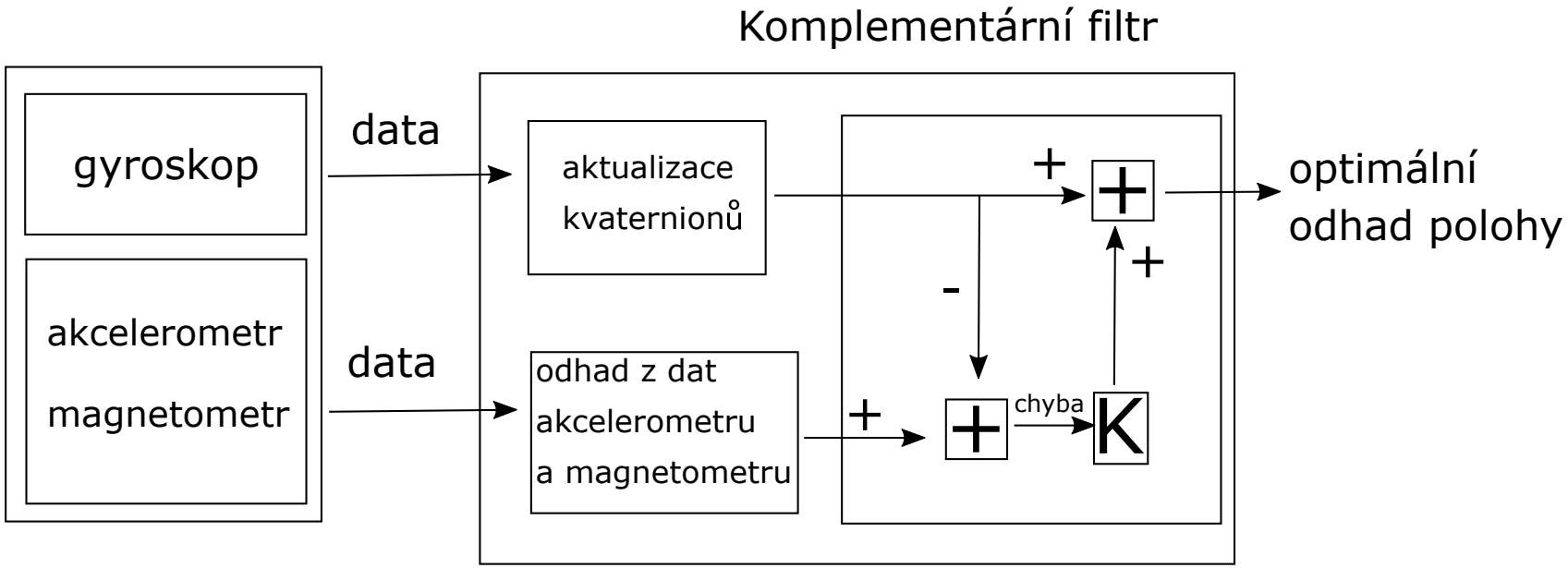
AHRS

- AHRS v Trilobotovi funguje na principu Komplementárního filtru (obrázek a popis na dalším slajdu)
- Velmi jednoduše:
  - Data z gyroskopu se používají na krátkodobou predikci
  - V dlouhodobém horizontu jsou upravována daty z akcelerometru (gyroskopy mají tendenci driftovat – i v klidu vykazují chybně pohybová data)
  - Magnetometr slouží pro zpřesnění – odkazuje na (relativně) neměnný sever (pokud není poblíž kov nebo magnet)
- Funkčnost závisí na prvotním nakalibrování a mnoha dalších věcech

- Data z gyroskopu jsou převedená na tzv. *kvaterniony* – 4 čísla. Matematicky je to rozšíření komplexních čísel, v našem případě udávají tři z nich souřadnice osy otáčení a čtvrté otočení kolem nich.
- Kvaterniony se následně převedou na odhad polohy

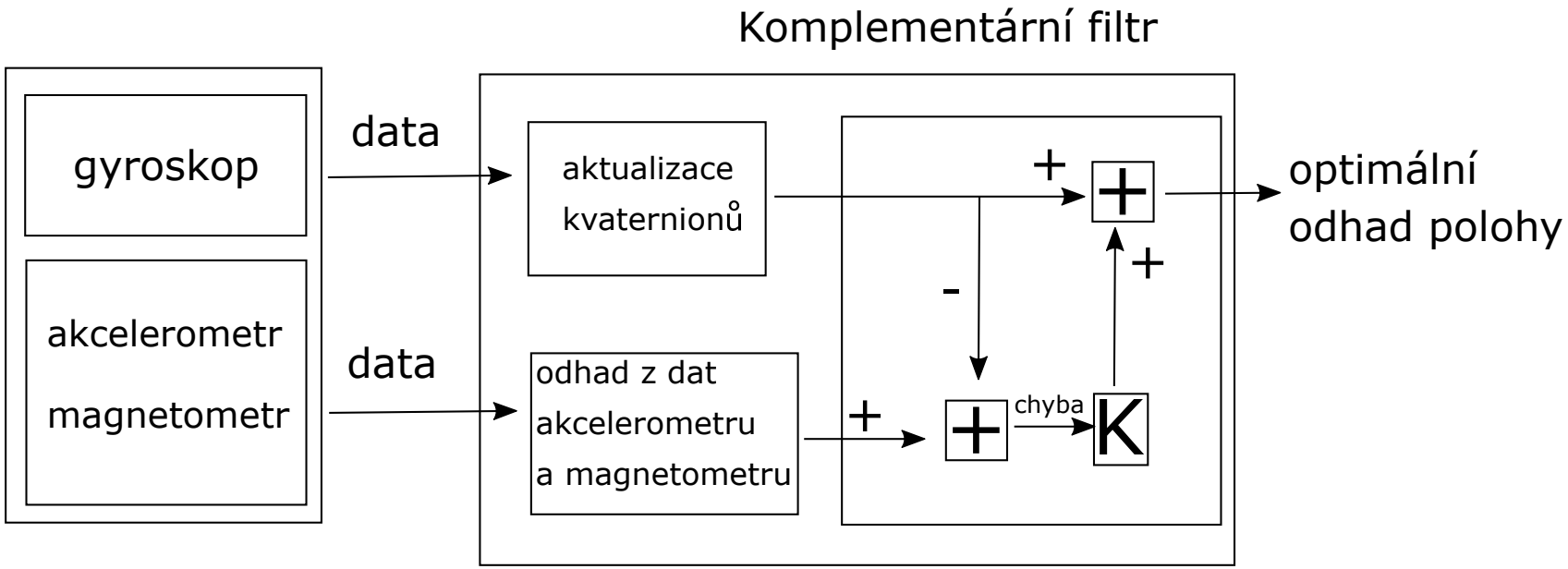


- Data z akcelerometru a magnetometru jsou převedená na odhad polohy pomocí poměrně složité goniometrie.
- Využívá se dolní propusti – filtrují se velmi rychlé změny



Jak funguje AHRS: Komplementární filtr

- vypočítá se chyba – rozdíl mezi daty z gyroskopu a akcelerometru
- Parametr K udává, jak výrazně se projeví korekce. V našem případě je K = 0.02
- Pokud chyba*K není příliš velká, bude přičtená k datům z gyroskopu a výsledek pokládán za optimální odhad
  - Kolem úhlu 0 vznikají problémy, kdy jedna metoda odhaduje úhel například 1 stupeň, druhá ale třeba 359. Pokud je tedy chyba příliš vysoká, předpokládá se, že je to tento případ a nebude započítaná do optimálního odhadu



Konec