Konstruktion av en autonom tävlingsbil

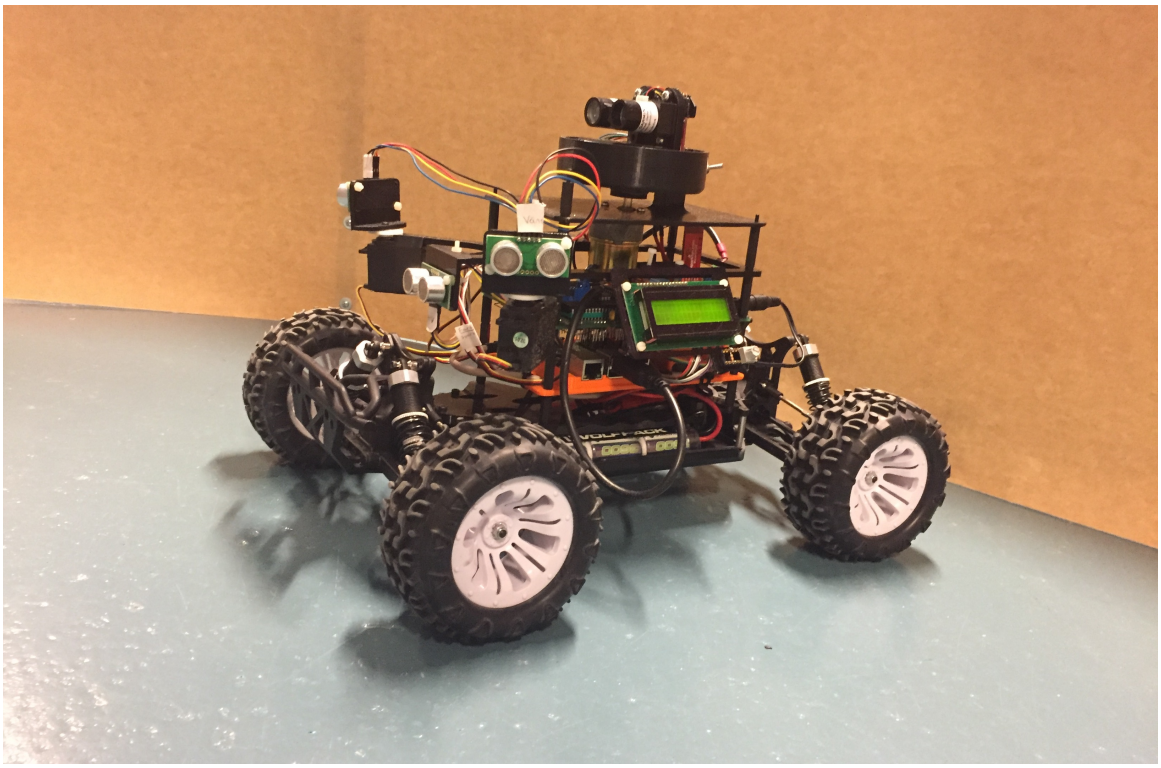
Teknisk dokumentation

Kandidatprojekt i elektronik

Tekniska högskolan vid Linköpings universitet

Christopher Albinsson
Fredrik Almin
Alexander Bärlund
Dennis Edblom
Johan Edstedt
Johannes Grundell
Maria Posluk
Petter Stenhagen

Version 1.1



Status

	Signatur	**Datum**
Granskad		
Godkänd		

Projektidentitet

Projektgrupp 13, 2017/VT
Linköpings tekniska högskola, ISY

Projektmedlemmar			
Namn	**Ansvar**	**Telefon**	**E-post**
Christopher Albinsson	Mjukvara	0705265293	chral920@student.liu.se
Fredrik Almin	Projektledare	0735202300	freal458@student.liu.se
Alexander Bärlund	Bana	0761120772	aleba707@student.liu.se
Dennis Edblom	Hårdvara	0765938131	dened825@student.liu.se
Johan Edstedt	Test	0707496241	johed950@student.liu.se
Johannes Grundell	Kommunikation	0732515123	johgr505@student.liu.se
Maria Posluk	Dokument	0707188861	marpo758@student.liu.se
Petter Stenhagen	Bokning	0722534409	petst908@student.liu.se

Kund: ISY, 581 83 Linköping kundtelefon 013-281000; Fax: 013-139282
Kontaktperson hos kund: Tomas Svensson, 013-281368, Tomas.Svensson@liu.se
Kursansvarig: Tomas Svensson, 013-281368, Tomas.Svensson@liu.se
Handledare: Peter Johansson, 013-28 1345, Peter.A.Johansson@liu.se

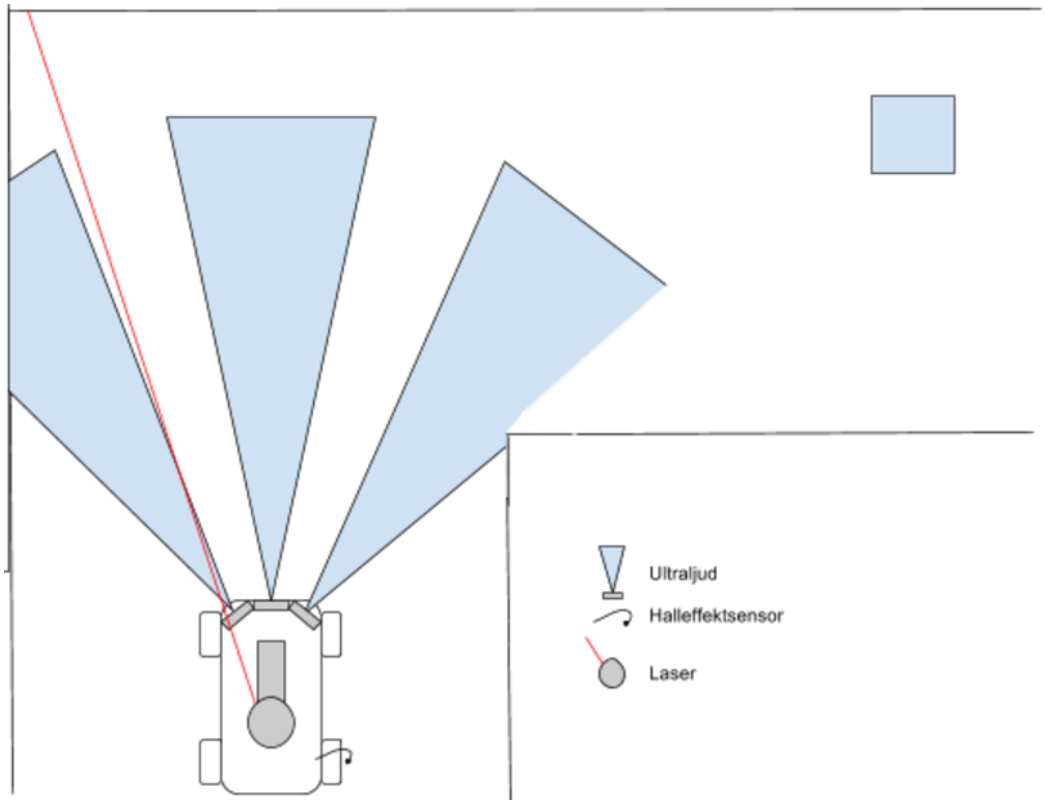
Innehåll

Dokumenthistorik

Version	Datum	Utförda förändringar	Utförda av	Granskad
1.1	12 juli 2019	Små korrigeringar	Grupp 13	
1.0	23 maj 2017	Första versionen	Grupp 13	

1 Inledning

Idag går bilutvecklingen mot att göra bilar autonoma och allt fler företag börjar att utveckla autonoma bilar. I detta kandidatprojekt har åtta studenter på Tekniska högskolan vid Linköpings universitet utvecklat en autonom tävlingsbil i RC-skala. Syftet med denna rapport är att beskriva hur vår tävlingsbil fungerar och är uppbyggd. Tävlingsbilen kan köras autonomt och manuellt via fjärrstyrning från en dator. Bilen består utöver chassi och drivlina av fyra moduler: kommunikationsmodul, styrmodul, sensormodul och lasermodul. Det ingår även mjukvara till dator för visualisering av sensordata och bilens omgivning samt styrning av bilen. I figur 1 visas en exempelbild när bilen kör i en bana med en 90°-sväng och ett hinder efter svängen.



Figur 1: Bilen i sin omgivning.

2 Produkten

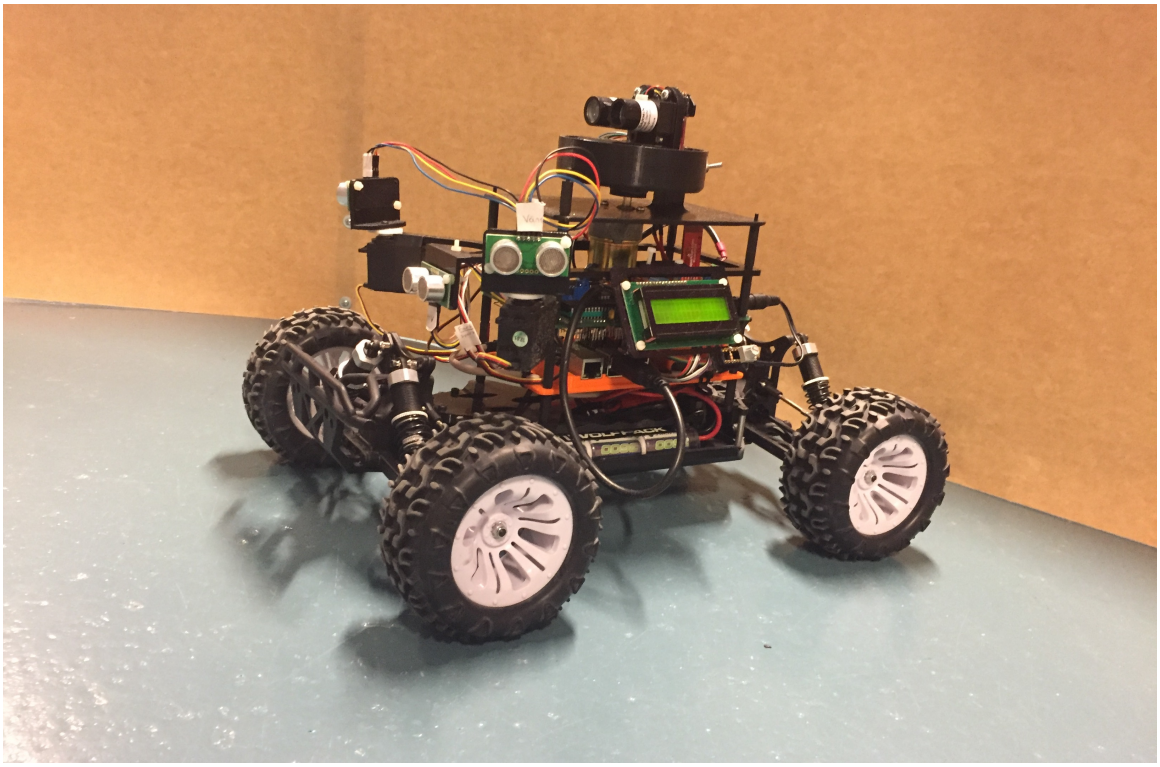
Produkten som denna dokumentation anser är en autonom tävlingsbil i RC-skala. Bilen är en slutprodukt i ett kandidatprojekt och kommer användas för att tävla i en tävlingsbana, se banspecifikation. Målsättningen är att ta sig genom banan på så kort tid som möjligt. Figur 2 visar den slutgiltiga bilen.

Bilen kan köras manuellt via fjärrstyrning från en dator. Användaren kan via datorn styra bilen, ändra regleringsparametrar, se sensorvärden, samt byta bilens körningsläge mellan manuell och autonom. Det visas även kontinuerligt för användaren hur bilens omgivning ser ut.

På bilen har fyra moduler implementerats: styrmodul, kommunikationsmodul, sensormodul och lasermodul. Med hjälp av sensormodulen och lasermodulen kan en uppfattning av omgivningen fås. Informationen om omgivningen, såsom avstånd till olika objekt och bilens hastighet, skickas vidare till kommunikationsmodulen och styrmodulen.

I det autonoma läget analyserar styrmodulen sensorvärdena med hjälp av en maskininlärningsalgoritm och en regleralgoritm och reglerar därefter bilens fart och styrning. Bilens autonoma läge kan därför användas som tävlingsbil. I styrmodulens manuella läge tar styrmodulen emot styrinstruktioner från datorn och styr bilen utifrån dessa. Det går att gasa, stanna, svänga och backa bilen i manuellt läge.

Kommunikationsmodulen skickar vidare alla sensorvärden från bilen till datorn och tar dessutom emot all information som skickas från användaren och skickar dessa vidare till styrmodulen.



Figur 2: En bild på den slutgiltiga bilen.

3 Teori

I detta avsnitt beskrivs teorin som har använts för att implementera regleralgoritmen och maskininlärningsalgoritmen i styrmodulen.

3.1 Reglering

Regleralgoritmen för att hålla bilen på ett visst avstånd från banans väggar består av två delar, en klassisk PD-algoritm samt en term som viktar den vinkel (-90°till 90°) från lasern som gett det längsta avståndet. Algoritmen ser, aningen förenklat, ut som följer:

Om $|e| > 5$: //Om felet är tillräckligt stort

$u[n] = K_P * e[n] + K_D * (e[n] - e[n-1]) + Uc$ //Räkna ut ett styrutslag

Annars:

$u[n] = u[n-1]$ //Använd senaste styrutslaget

Om $|laservinkel| > 20$: //Om vinkeln är mer än 10 grader åt sidan

Returnera $u[n] + K_L * laservinkel$ //Vikta in laservinkel i styrutslaget

Annars:

Returnerna $u[n]$ //Returnera bara utslaget från PD-delen

där e är skillnaden mellan vald referenslinje och bilens position och $u[n]$ är nuvarande styrutslag. Alla konstanter finns att se i programkoden, vissa kan dock ändras från programvaran och är därför inte definitiva. Felet bildas antingen som skillnaden i avstånd från respektive sidosensor alternativt som en konstant minus avståndet till vägg från en sidosensor beroende på vilket referenslinje bilen ska hålla. Att bara räkna ut ett nytt styrutslag när felet är större än 5 samt bara använda laservinkeln när den är större än 10°gör systemet mindre känsligt för störningar och oexakta sensorvärden.

Fartregleringen tar hänsyn till samma fel som ovan och summerar samt viktar beloppet av de 10 senaste felen och subtrahera den från en vald maxfart, se (1).

$$s[n] = maxfart - K_{SR} \sum_{k=n-10}^{n} e[k] \tag{1}$$

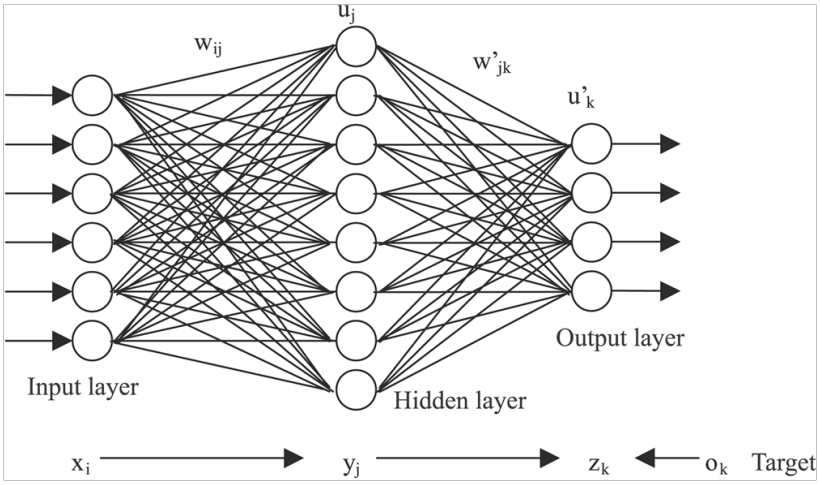
där $s[n]$ är nuvarande fartpåslag. Värdet på maxfart och K_{SR} kan ses i programkod, K_{SR} kan ändras från programvara. För mer info se *Förstudie Reglerteknik*.

3.2 Maskininlärning

Maskinlärning har använts för att bedöma huruvida det finns ett hinder framför bilen på banan, och vilken väg kring hindret som är lämplig. Maskininlärningsalgoritmen fungerar genom att använda ett neuralt nätverk.

Ett neuralt nätverk är ett lärande system som lär sig att klassificera nya datapunkter efter att ha tränats på datapunkter där en klassificering redan är gjord externt. Den kommer således att själv returnera den klasstillhörighet som den finner mest lämplig. Det som skickas in i algoritmen är en vektor av olika input features.

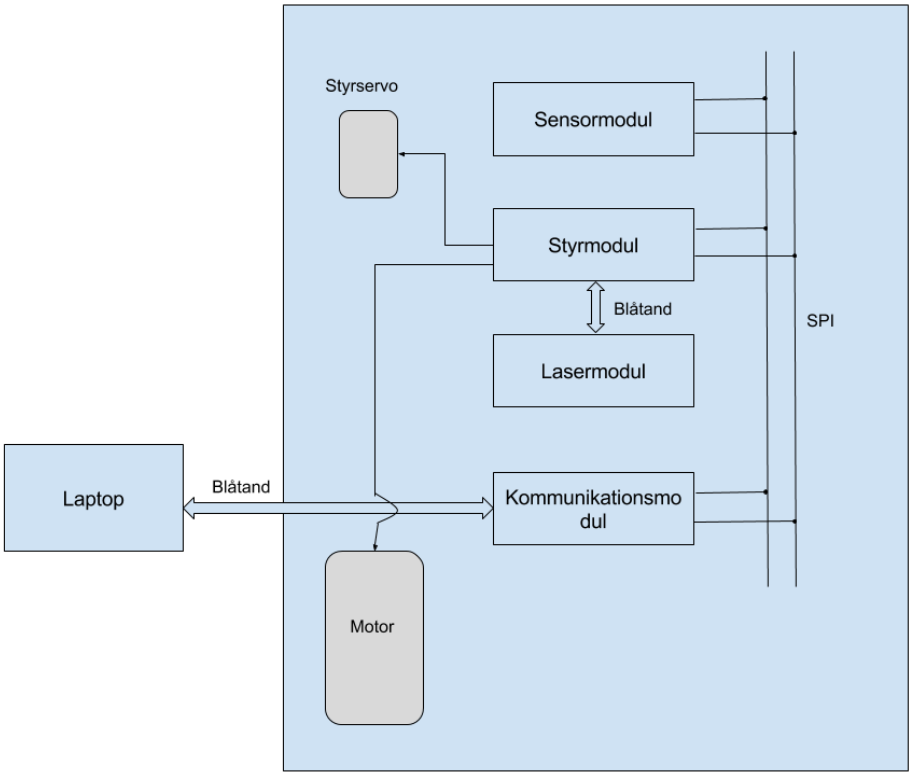
Nätverket består av noder som kallas neuroner och bågar som utgör länkar mellan neuronerna. För en exempelbild se figur 3. Vikterna mellan bågarna uppdateras när algoritmen optimeras och utgör resultatet av träningen. Algoritmen kommer för varje träningsdatapunkt att uppdatera vikterna på bågarna i nätverket och på så sätt kontinuerligt förbättra algoritmens förmåga att i träningsdata diskriminera mellan datapunkter av olika klasser. När algoritmen sedan är färdiglärd är då förhoppningen att den kan klassifiera data utan given klasstillhörighet.



Figur 3: En exempelbild på ett neuralt nätverk.

4 Systemet

Hela systemet består av fyra moduler samt en programvara på en dator. I figur 4 kan man se ett översiktligt blockschema över systemet, de olika modulerna samt hur de kommunicerar. Sensormodulen tar upp information från omgivningen och skickar det via SPI till styrmodulen som i sin tur skickar informationen till kommunikationsmodulen. Även lasermodulen tar upp information från omgivningen som skickas direkt till styrmodulen via blåtand. Denna information skickas sedan vidare till kommunikationsmodulen. Från laptopen kommer olika instruktioner, både vid manuell och autonom körning, såsom start och stopp. Det skickas via blåtand till kommunikationsmodulen som för det vidare till styrmodulen. Användaren kan se information, såsom sensorvärden, från bilen vid körning vilket skickas från sensor- och lasermodulen till kommunikationsmodulen och vidare till datorn via blåtand.



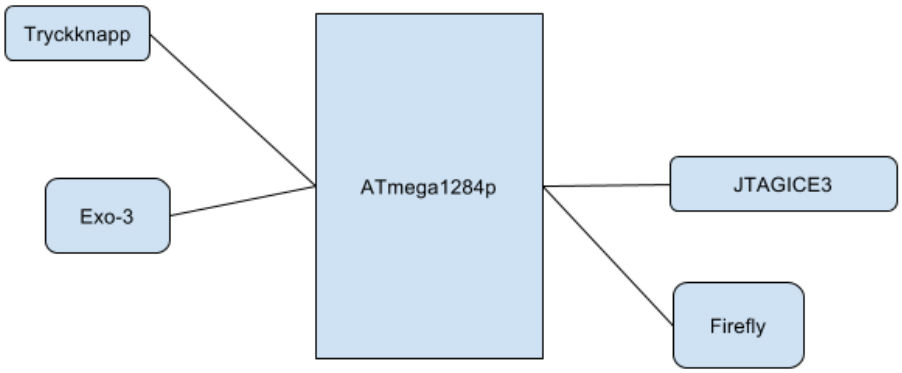
Figur 4: Ett översiktligt blockschema över hela systemet.

5 Modulerna

I detta avsnitt beskrivs varje modul på bilen mer ingående. De moduler som ingår i bilen är kommunikationsmodul, styrmodul, sensormodul, lasermodul samt programvara för att styra bilen.

5.1 Kommunikationsmodul

I detta avsnitt beskrivs kommunikationsmodulens funktion, mjukvara samt hårdvara som ingår. Ett blockschema för kommunikationsmodulen ses i figur 5.



Figur 5: Ett blockschema över kommunikationsmodulen.

5.1.1 Funktionsbeskrivning

Kommunikationsmodulen är den modul som hanterar kommunikation mellan bilen och en dator via blåtand. Modulen tar emot ett datapaket från styrmodulen och skickar vidare det till datorn. Informationen som skickas består av sensorvärden, laservärden, styrinformation, styrinstruktioner samt regleringsparametrar.

5.1.2 Hårdvara

Nedan följer de komponenter som ingår i kommunikationsmodulen. För information om hur komponenterna är kopplade, se kopplingsschemat i appendix A.

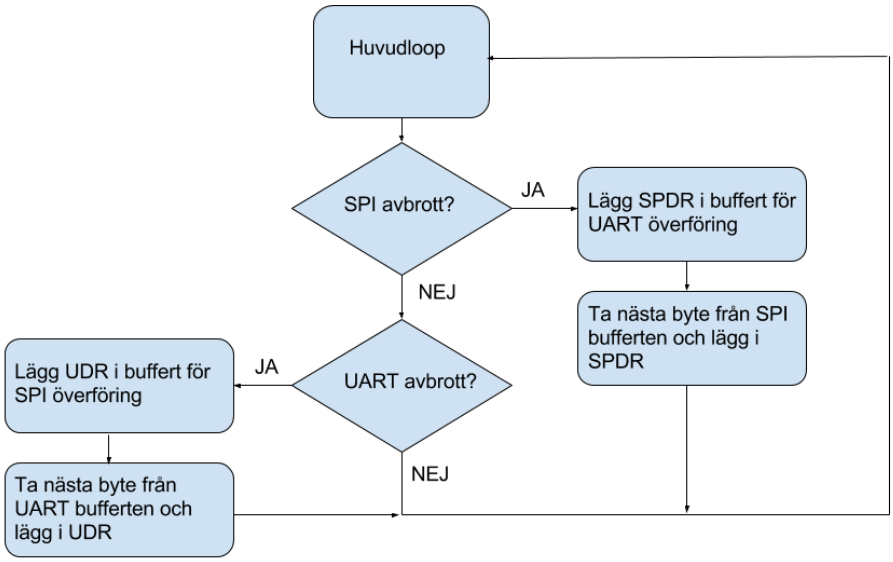
- En Atmel AVR ATmega1284P, mikrokontroller
- Ett blåtandsmodem, FireFly BlueSMiRF Gold. Baud rate på 115,2 kBaud.
- En EXO-3, kristalloscillator i 14,7456 MHz
- En tryckknapp för reset
- Portabel: JTAG ICE 3, emulator

5.1.3 Gränssnitt

Här beskrivs kommunikationsmodulens interna gränssnitt. Gränssnittet mellan kommunikationsmodulen och styrmodulen är en SPI buss och gränssnittet till datorn är först UART som går till ett blåtandsmodem som sedan kommunicerar med datorn. För protokoll se avsnitt 6.

5.1.4 Mjukvara

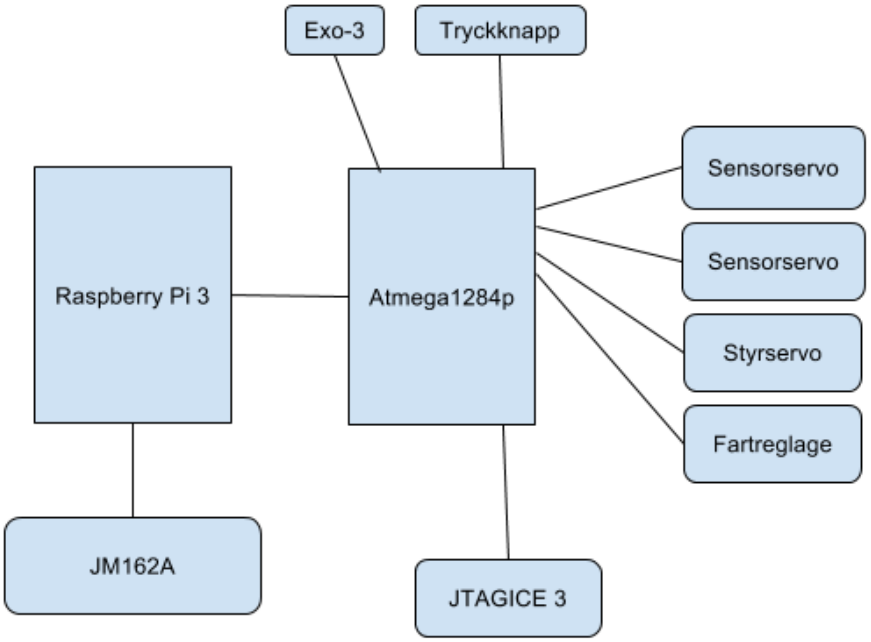
Kommunikationsmodulen har som uppgift att ta emot data och skicka vidare det. Mjukvaran har två buffrar med plats för 16 bytes vardera, en för den data som kommer från styrmodulen och ska till datorn och en för den data som kommer från datorn och ska till styrmodulen. Det finns två riktningar för data, från UART till SPI eller från SPI till UART. När det går från SPI till UART sparas ett datapaket (16 bytes) från SPI bussen i en buffert och när bufferten är full skickas paketet vidare via UART. Och när det går från UART till SPI sker det på samma sätt att först tas ett paket upp och sedan skickas det vidare. Det kräver att SPI bussen inte skickar för snabbt då UART har en långsammare hastighet och det kräver också att datorn inte skickar snabbare än styrmodulen då SPI överföringar bara sker när styrmodulen skickar något. Vi har löst det genom att låta hela kedjan av kommunikation bestämmas av styrmodulen. Först skickar styrmodulen ett datapaket via SPI som sedan skickas vidare till datorn och när datorn har fått paketet kan datorn skicka sitt datapaket till kommunikationsmodulen. Kommunikationsmodulen skickar sedan vidare datorns paket nästa gång styrmodulen startar en överföring. För flödesschema se figur 6.



Figur 6: Flödesschema för kommunikationsmodulen

5.2 Styrmodul

I detta avsnitt beskrivs styrmodulens funktion, mjukvara samt den hårdvara som ingår. Ett blockschema för styrmodulen ses i figur 7.



Figur 7: Ett blockschema över styrmodulen.

5.2.1 Funktionsbeskrivning

Bilens styrmodul reglerar bilens motorsystem som driver hjulen, styrservot som styr framhjulen samt de två sensorservona. Detta görs med hjälp av en maskininlärningsalgoritm samt regleralgoritm och data från de andra modulerna. Mer om mjukvaran under avsnitt 5.2.4.

5.2.2 Hårdvara

Styrmodulen består av komponenterna listade nedan, se även figur 7 och kopplingsschemat i appendix A.

- En Atmel AVR ATmega1284P, mikrokontroller
- En Raspberry Pi 3, enchipsdator
- En EXO-3, kristalloscillator i 14,7456 MHz
- FTDI Chip USB till UART för Raspberry Pi, konverterare
- Spänningsregulator 3.3 V till 5 V
- Tryckknapp för reset
- Potentiometer 10kohm
- En LCD JM162A, LCD-display
- Portabel: JTAG ICE 3, emulator

Enchipsdatorn används för att göra de beräkningar som styrmodulen ska utföra medan mikrokontrollern kommer att fungera som pulsgenerator för styrning av bilens servon och fartreglage.

5.2.3 Gränssnitt

Här beskrivs gränssnittet mellan de interna komponenterna i styrmodulen.

Enchipsdatorn kommunicerar med mikrokontrollern seriellt mellan en USB-port på enchipsdatorn och en UART-port på mikrokontrollern. Mellan portarna finns ett FTDI-chip som konverterar mellan gränssnitten. Informationen som mikrokontrollern tar emot från enchipsdatorn är i form av ett värde mellan 0 och 255 som mikrokontrollern konverterar till en pulslängd. Varje datapaket består av 3 bytes, se tabell 1. Överföringshastigheten är 115200 bps.

Tabell 1: Protokoll för UART

Funktion	Header	Pulsvärde styrservo	Pulsvärde fartreglage
Innehåll	255	0-255	0-255

Styrsignaler till motor och styrservo kommer vara pulsbreddsmodifierade signaler. Tiden mellan den positiva flanken på pulserna kommer konstant att vara 20ms och längden på pulsen bestämmer hur mycket motorsystemet eller styrservot reagerar. Pulsernas längd genereras genom att använda en 16-bitars timer som finns inbyggd i Atmega1284.

Enchipsdatorn kommunicerar med LCD-displayen via en 6-pin GPIO-port där LCD-displayen körs i 4bit-mode.

5.2.4 Mjukvara

Mikrokontrollerns uppgift är att skicka pulser till servon samt ta emot information från Raspberryn. Det har gjorts genom att programmera olika register samt skapa en avbrottsrutin för UART-överföring. Värdet som kommer från Raspberryn via UART görs om till en tid i ms genom följande samband: $1843 + (pulsvärde) * 7$. Värdet 1843 motsvaras av att timern ger ut en puls med längd 1ms och pulsvärdet 255 ger en puls med längd 2ms. Neutral för styrservo och fartreglage är 1,5ms. Pulserna till sensorservona bestäms genom att ta medelvärdet på de 6 senaste pulsvärdena för styrservot och minska dessa genom sambandet: $1843 + (pulsmedelvärde) * 2$, där multiplikation med 2 istället för 7 innebär en minskning av rörelseintervallet.

Raspberryn har hand om regleringen av bilen och är den som tar in data från sensor- och kommunikationsmodul för att sedan ta styrbeslut.

Regleringen för styrningen består i princip av två delar, en PD-algoritm som använder sig av de två ultraljudssensorerna på sidan och bildar ett fel utifrån dessa, samt en extra term som viktar den vinkel från lasern som gett det största avståndet. Resultatet av de tre termerna är ett styrutslag mellan 0 och 255 som därefter skickas till mikrokontrollern. Hastighetsregleringen sker med hjälp av en fartreduktionsterm som viktar en summa av de 10 senaste felen bildade av ultraljudssensorerna och subtraherar bort det från en bestämd maxhastighet.

För att hantera hinder har en maskininlärningsalgoritm implementerats. Det finns två, till hög grad skilda problem som maskininlärningen ska lösa. Det första problemet handlar om att upptäcka hinder. Det andra problemet handlar om att generera en referenslinje som styralgoritmen kan reglera efter. Denna referenslinje beror på ett eventuellt hinders position i banan. De båda problemen har lösts med hjälp av ett och samma neurala nätverk.

Inparametrar till nätverket är 50 laservärden som uppmäts från de 180 graderna under laservarvet som är vinklade framåt i förhållande till bilen, alltså intervallet {-90°, 90°} där 0° är rakt fram i förhållande till bilen. Då lasern har en vinkelskillnad på 3-4 grader mellan två på varandra följande mätningar, erhålls ungefär 50 mätpunkter inom intervallet {-90°, 90°}. Antal uppmätta datapunkter kan dock skilja något. Därför behövs ibland punkter tas bort eller läggas till för att få riktig dimension på vektorn som skickas in till maskininlärningsalgoritmen. Detta görs i vinkelspannets ytterkanter då dessa mätpunkter anses mindre betydelsefulla än de mätvärden när lasern är riktad mer rakt framåt.

Träningsdata har samlats in och strukturerats så som det beskrivs i ovanstående stycke med 50 mätpunkter per datapunkt. Totalt är det ca 6000 datapunkter som samlats in genom att köra bilen, på vilken lasern är fastmonterad, i en bana. Ibland har hinder satts inom laserns synfält. Datapunkterna har sedan märkts med klass 0,1 eller 2, där 0 indikerar hinder till höger, 1 indikerar att det inte finns något hinder och 2 indikerar hinder till vänster. Algoritmen tränades på ca 90% av hela datasetet och resterande 10% av datasetet användes för att utvärdera algoritmen.

När träningsdata samlats in tränades algoritmen på det sätt som redogörs för i avsnitt 3.2. Vilka parametrar i form av antal neuronlager samt antal neuroner per neuronlager som ger mest tillfredställande resultat är svårt att veta på förhand och algoritmen kördes och utvärderades flera gånger med olika kombinationer av parameteruppsättningar. Den uppsättning som gav bäst resultat och som därför i slutändan valdes var ett neuralt nätverket bestående av 4 lager á 800 neuroner. Aktiveringsfunktionen till samtliga neuroner är en Rectified linear unit (Relu). För att optimera det neurala nätverket användes optimeringsfunktionen stochastic gradient descent (SGD).

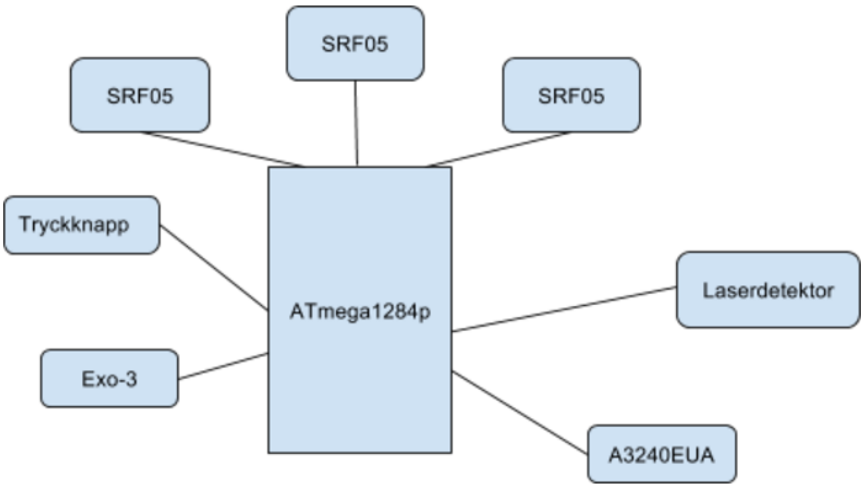
Algoritmen uppnår en precision på ca 90 % på insamlad data. Detta bedömdes inte som tillräckligt tillfredställande och ett filter finns för att få en större tillförlitlighet i de givna referenslinjerna. Detta är utformat så att maskininlärningsalgoritmen måste ge tre identiska prediktioner i rad för att en ny referenslinje ska ges till styralgoritmen.

Genom att analysera de felklassificeringar som algoritmen gör har vi identifierat tre olika typer av fel som algoritmen gör. För det första är en förhållandevis vanligt förekommande typ av felklassificering från algoritmen att den inte detekterar ett hinder när bilen befinner sig långt bort (ca 3 m.). Hindret upptäcks dock nästintill alltid när bilen kommer närmare det. En andra typ av felklassificering som algoritmen gör är att den väljer att köra på den andra sidan av hindret än det var tänkt. Denna typ av felklassificering förekommer ofta i de fall där hindret ställs väldigt nära mitten, vilket gör det föga förvånande att algoritmen vid vissa tillfällen väljer att föreslå att köra förbi hindret på den andra sidan. En tredje typ av felklassificering är att algoritmen upptäcker ett hinder när det inte står ett hinder på banan. När algoritmen hade tränats tillräckligt många gånger förekom denna typ av fel betydligt mer sällan. Det är inverkan av denna feltyp som framför allt minimeras av det ovan nämnda filtret.

För att underlätta hinderdetekteringen stängs maskininlärningsalgoritmen av vid tilfällen då roboten befinner sig i en kurva. Detta beror på att algoritmen har högre träffsäkerhet då den befinner sig på en raksträcka och felklassificeringar i kurvor kan vara förödande för bilens förmåga att undvika att krocka i väggar.

5.3 Sensormodul

I sensormodulen ingår alla sensorer utom lasern. Modulen har som uppgift att samla in data från omgivningen för att det sedan ska kunna fattas styrbeslut i styrmodulen.



Figur 8: Ett blockschema över sensormodulen.

5.3.1 Hårdvara

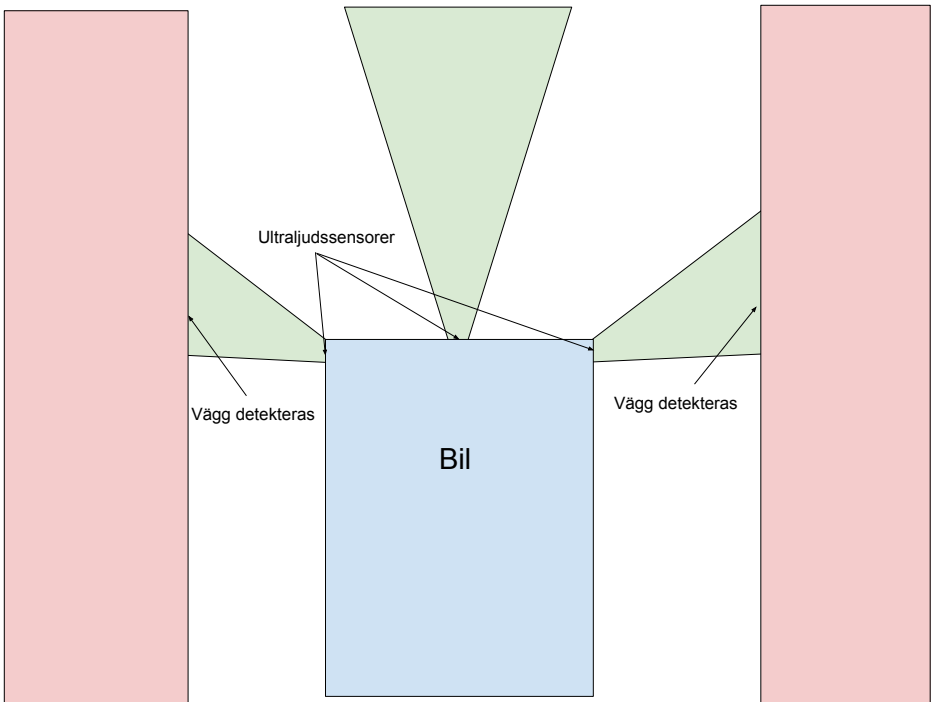
Sensormodulen består av komponenteran listade nedan, se även figur 8 och kopplingsschemat i appendix A.

- En Atmel AVR ATmega1284P, mikrokontroller
- Tre SRF05, ultraljudssensorer
- En laserdetektor
- En EXO-3, kristalloscillator i 14,7456 MHz
- En A3240EUA, halleffektsensor
- En tryckknapp för reset

5.3.2 Ultraljudssensorer

Modulen har tre ultraljudssensorer, SRF05, som förser bilen med avståndsmätning framåt och åt sidorna. Ultraljudssensorerna kan mäta avstånd till väggar och hinder på banan upp till fyra meter bort. Detta används vid reglering av bilen för att hålla den i mitten av banan vid raksträckor. Då mikrokontrollern skickar en puls på minst 10 μs kommer sensorn att skicka 8 ultraljudssmällar. Sensorn returnerar sedan en puls vars pulsbredd är proportionell mot tiden det tar för ultraljudet att komma tillbaka till sensorn. Då ljudhastigheten är konstant kan en uppskattning på avståndet till närmaste objekt fås. För att undvika eko så väntar mikrokontrollern 50 ms innan nästa puls skickas. Detta ger en uppdateringsfrekvens på 20 Hz. Figur 9 visas en förenklad bild på hur ultraljudet fungerar under körning.

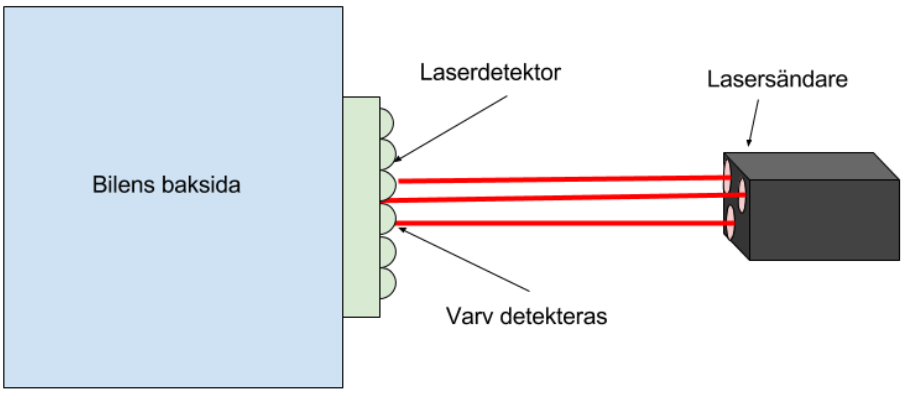
I mikrokontrollern registreras tiden det tar för ultraljudet att komma tillbaka med hjälp av avbrottsvektorer. När en hel puls har tagits emot kollas värdet på en specifik timer och sträckan räknas ut enligt sträcka[cm] = hastighet[cm/s] * tid[s]. Detta värde på sträckan sparas i en variabel, som kontinuerligt uppdateras när modulen får in nya ultraljudsvärden.



Figur 9: En förenklad bild på hur ultraljudssensorerna fungerar på bilen.

5.3.3 Laserdetektor

Laserdetektorn används för att bilen ska kunna avgöra när den kört ett varv. Vid mållinjen finns en lasersändare som träffar laserdetektorn när bilen kör förbi. När laserdetektorn träffas skickas en signal till mikrokontrollern som kan detekteras och vidareförmedlas till styrmodulen. Detektorn kan sedan återaktiveras med en puls från mikrokontrollern. Figur 10 visar en förenklad bild på hur laserdetektorn fungerar under körning. I mikrokontrollern registreras träffen i en avbrottsvektor, och



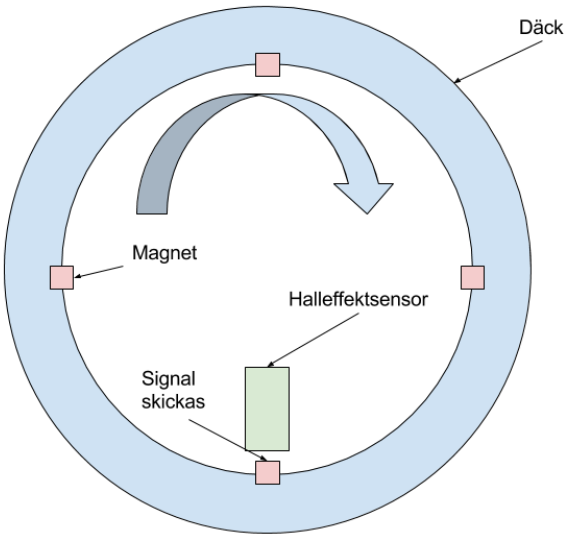
Figur 10: En förenklad bild på hur laserdetektorn fungerar på bilen.

antalet varv räknas upp. Sedan väntar mikrokontrollern ett visst antal sekunder innan en puls skickas till laserdetektorn. Detta för att undvika att detektorn registrerar flera träffar på samma varv.

5.3.4 Halleffektsensor

Halleffektsensorn, A3240EUA, används för att avgöra bilens hastighet. Sensorn detekterar förändringar i magnetfältet runt sig, och ger då en logisk 0:a ut. På bilen är 4 magneter fastsatta på insidan

av hjulet och då hjulet snurrar runt halleffektsensorn kan detta användas för att bestämma hjulets hastighet. Figur 11 visar en förenklad bild på hur halleffektsensorn fungerar under körning.



Figur 11: En förenklad bild på hur halleffektsensorn fungerar på bilen.

I mikrokontrollern räknas hastigheten ut i en avbrottsvektor genom att räkna tiden mellan två avbrott och använda att däckomkretsen är känd. Sedan appliceras ett medianfilter för att få jämna värden.

5.3.5 Seriell överföring till styrmodul

När mikrokontrollern i sensormodulen får ett STC avbrott (SPI Serial Transfer Complete) och det är headern 0xFF som har skickats från styrmodulen börjar de senaste sensorvärdena att skickas. Sensorvärdena läses av enligt ordningen nedan.

1. INT0 - Vänstra ultraljudssensorn
2. PCINT1 - Mittersta ultraljudssensorn
3. PCINT2 - Högra ultraljudssensorn
4. INT2 - Halleffektsensorn
5. INT1 - Laserdetektorn

När ett värde skickats inväntar modulen nästa avbrott och nästa variabel skickas. När alla sensorvärden skickats skickas en checksumma som består av en XOR-operation på alla skickade bytes. Detta för att styrmodulen ska kunna se om något fel uppkommit i överföringen.

5.4 Lasermodul

I lasermodulen ingår en Lidar Lite v3 som roterar ovanför bilen. På så sätt kan avstånd och vinkel till objekt runt bilen detekteras och informationen kan användas till reglering av bilen samt kartuppritning. I detta avsnitt beskrivs hur lasermodulen är uppbyggd och hur den fungerar.

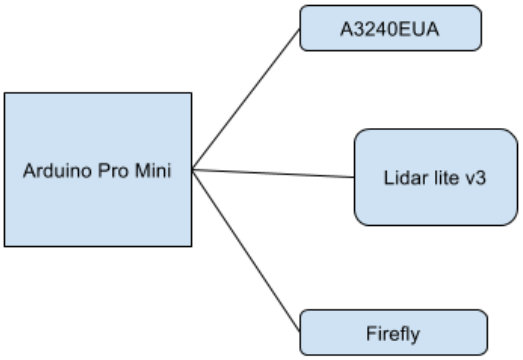
5.4.1 Hårdvara

Nedan listas de komponenter som lasermodulen består av, se även kopplingsschemat i appendix A.

- En Arduino Pro Mini 328, mikrokontrollerkort
- En A3240EUA, halleffektsensor

- En Lidar Lite v3, IR-laser
- Ett blåtandsmodem, FireFly BlueSMiRF Gold. Baud rate på 115,2 kBaud.

Lasermodulen består av en Lidar Lite V3, som är en laser som når upp till 40 meter bort. Modulen innehåller även en Arduino Pro Mini, en firefly och en halleffektsensor, A3240EUA. På så sätt kan den skicka avstånd och vinkel till det objekt som lasern träffar till styrmodulen. Därefter används datan till att undvika hinder med hjälp av maskininlärning. Det används även i kartuppritningen för att placera banans väggar och hinder på rätt avstånd från bilen. Figur 12 visar ett blockschema över modulen och dess hårdvara.



Figur 12: Ett blockschema över lasermodulen.

5.5 Programvara

Till bilen har en programvara skapats i syfte att låta användaren kommunicera med bilen. Programmet skrevs i språket Python.

5.5.1 Funktionsbeskrivning

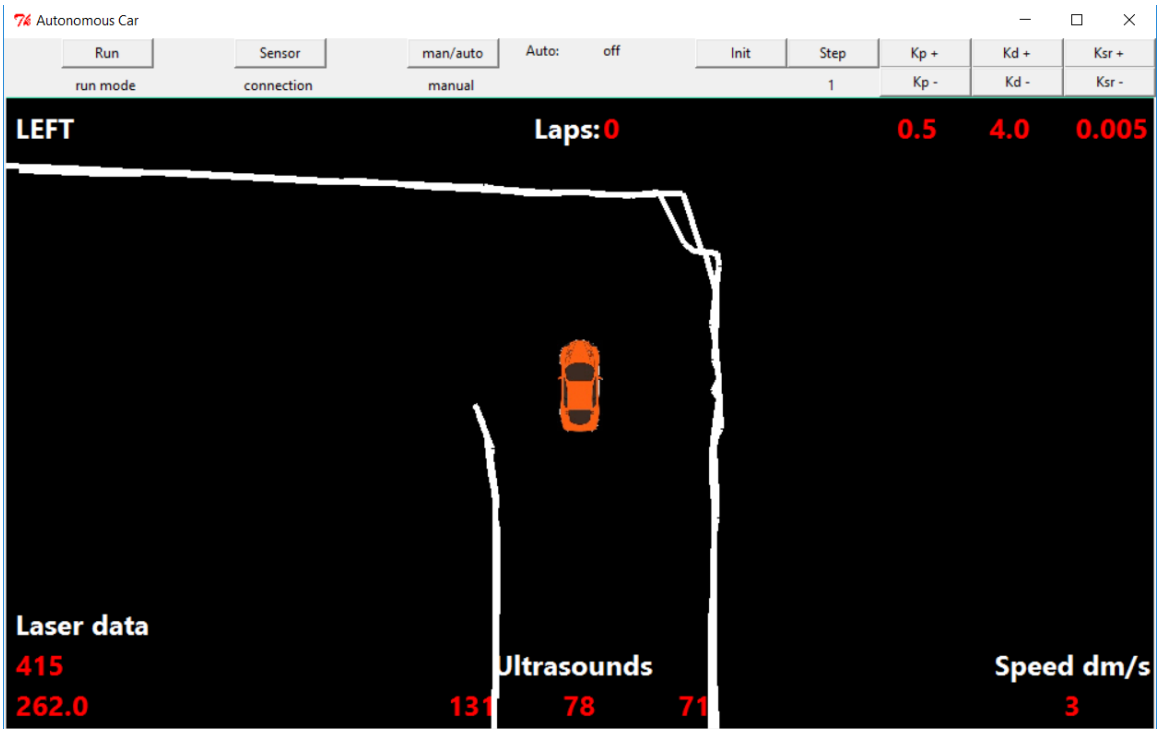
Programmets syfte är att kunna fjärrstyra bilen, visualisera sensordata och styrinformation, växla mellan bilens manuella och autonoma läge, ändra på bilens reglerparametrar samt att momentant rita ut bilen i dens omgivning.

Programmet har en **HUD** (**H**ead **U**p **D**isplay) längst upp i programfönstret där användaren kan ge olika instruktioner såsom att ändra på bilens reglerparametrar, byta mellan bilens autonoma och manuella läge samt att byta mellan programmets två huvudlägen: körläge och sensorläge.

I körläget ritas kartan upp utifrån inkommande laservärden och de senaste värdena laseravstånd och vinkel skrivs ut. Utöver det skriver programmet ut maskininlärningens referenslinje, ultraljudsensorernas värden, nuvarande reglerparametrar, bilens hastighet och antal varv bilen kört.

I sensorläget ritas tre grafer ut på de tre ultraljudsensorernas 30 senaste värden.

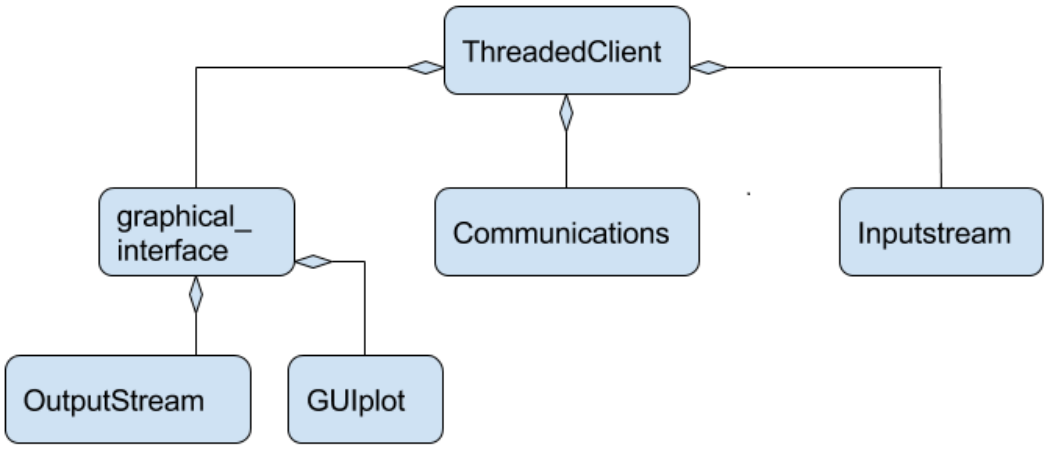
I figur 13 visas ett exempel på programmets interface i körläge medan bilen kör.



Figur 13: Klassdiagram för bilens tillhörande programvara.

5.5.2 Mjukvara

Programmet byggdes i sex olika filer med klasserna: ThreadedClient, Window (i filen graphical_interface), Communication (i filen Communications), InputStream, OutputStream och GUIplot. Se figur 14 för klassdiagram.



Figur 14: Klassdiagram för bilens tillhörande programvara.

Nedan listas klasserna och vad de fyller för funktion samt lite allmänt om programmet.

ThreadedClient

I *ThreadedClient.py* skapas ett *ThreadedClient*-objekt som initieras med ytterligare *Window*-, *Communication*- och *InputStream*-objekt som datamedlemmar. När *ThreadedClient*-objektet konstrueras körs även två processer: *worker_thread_1* och *periodic_call*.

Processen *worker_thread_1* är trådad vilket innebär att den koden exekveras parallellt med den andra. Tråden läser konstant seriell indata från bilen som kommer via blåtand med hjälp av *InputStream*-objektet och hanterar inkommande data. Om en inkommande byte har det hexadecimala värdet 0xFF är det ett tecken på att ett angivet protokoll skickas från bilen. Inkommande protokoll består av 16 byte. Om den sista byten i protokollet har det hexadecimala värdet 0xAA är det ett ”vanligt” protokoll och indatan sparas undan i en kö *queue*. Ifall den sista byten i protokollet har det hexadecimala värdet 0xBB är det ett protokoll med endast värden från bilens lasermodul som sparas undan i en lista *laser_list*.

Processen *perodic_call* är rekursiv och kallar varje iteration på en funktion *process_incoming* i *Window*-objektet för att behandla den indata som ligger först i kön *queue*. Processen kallar även på en funktion *update_lasers* i *Window*-objektet där funktionen tar *laser_list* som invärde och sedan töms *laser_list*. Det rekursiva kallet i *periodic_call* är fördröjt med en tid angiven av programmeraren.

InputStream

Klassen *InputStream* är väldigt simpel och består av en datamedlem. Objektet initieras som sagt i *ThreadedClient* och konstrueras med samma *Communication*-objekt som skapades i *ThreadedClient*. Endast två medlemsfunktioner finns: *get_next_bytes* och *car_ready_to_receive*.

Funktionen *get_next_bytes* returnerar 16 på rad inkommande bytes via *Communication*-objektet.

Funktionen *car_ready_to_receive* är en testfunktion för att kolla om programmet är uppkopplat till bilen.

Communication

Klassen *Communication* har som uppgift att sköta funktionalitet för den seriella blåtandkommunikationen. För att göra detta har biblioteket *Pyserial* använts.

Objektet konstrueras i *ThreadedClient* och initieras med fyra datamedlemmar: ett seriellt kommunikationsobjekt från *Pyserial*, en lista för att spara inkommande värden i, en datamedlem för att spara enskilda värden i och en boolvariabel för att representera huruvida uppkopplingen till bilen är aktiv.

I *Communication* finns främst två relevanta funktioner *send_sequence* och *read_sequence*. Funktionerna använder sig i sin tur av andra medlemsfunktioner *ser_write* och *ser_read* för att skicka och ta emot data i paket om 16 bytes.

Window* i *graphical_interface

Klassen *Window* innehåller all grafisk funktionalitet och omfattar även en del behandling av indata och skickande av utdata. *Window*-klassen är betydligt större än de andra. För att upprätta det grafiska gränssnittet har Pythons standardbibliotek *TkInter* använts.

Objektet *Window* skapas i *ThreadedClient* och tar ett *TkInter*-objekt, en kö och ett *Communication*-objekt som invärde. *Communication*-objektet är samma objekt (ej kopia) som används i de andra klasserna.

Window-objektet initierar då bl.a:

- kön
- *TkInter*-objektet med tillhörande *Frame*
- listor för kartuppritning
- ett *canvas* och en *font* för grafiken
- grafiska objekt såsom linjer, rektanglar, grafiska texter och en bild för att representera bilen
- kö för utdata
- objekt av typen *OutputStream*
- knappar och texter
- många hjälpvariabler

I konstruktörn körs även en del medlemsfunktioner för att initiera fler värden och variabler.

De mest relevanta funktionerna i klassen *Window* är:

- *process_incoming*
- *update_lasers*
- *handle_indata*
- *map_help*
- *map_add*
- *update_ui*

Funktionen *process_incoming* kallas av funktionen *periodic_call* i *ThreadedClient* och behandlat det protokollet som ligger först i kön *queue*. Funktionen överlagrar sedan protokollet i en datamedlem *input_byte* och kallar på *handle_indata*. Sedan kollar funktionen om programmet befinner sig i körläge eller sensorläge. Om programmet befinner sig körläge kallas funktionerna *map_help* och *update_ui*, medan programmet kör funktionen *update_plots* om det är i sensorläge. Funktionen *update_plots* uppdaterar graferna i sensorläget. Efter det kallar funktionen på en funktion *add* i *OutputStream*-objektet med hjälp av en datamedlem *key_press* som innehåller ett värde som representerar den senaste knappen användaren tryckt på, som sedan sätts till 0 (d.v.s inget knapptryck).

Funktionen *update_lasers* kallas också av funktionen *periodic_call* i *ThreadedClient*. Funktionen tar en lista med laservärden som invärde, konverterar dessa till koordinater i programfönstret och sparar undan koordinaterna i listor.

Handle_indata-funktionen behandlar den data som läggs i *input_byte* och uppdaterar relevanta variabler.

Funktionen *map_help* kollar om koordinatlistorna uppnått vad programmeraren anser vara en rimlig storlek och kallar isåfall på funktionen *map_add*.

Map_add-funktionen överför koordinaterna från tidigare nämnda koordinatlistor och sparar de i nya listor för att sedan ritas ut av funktionen *update_ui*.

Funktionen *update_ui* raderar grafiska objekt som ska raderas, skapar nya grafiska objekt utifrån de värden som sparades undan i *map_add*, uppdaterar grafiska texter utifrån data som uppdaterades i *handle_indata* och ritar ut allt på nytt.

Utöver dessa medlemsfunktioner innehåller *Window* även enskilda funktioner för vad som ska hända vid tryck på vardera knapp, vad som händer när användaren byter mellan kör- och sensorläge samt funktioner för vad som händer när användaren trycker på relevanta knappar på tangentbordet.

OutputStream

Objektet *OutputStream* skapas när *Window*-objektet initieras. Objektet menar att hantera den data som ska från datorn till bilen och innehåller endast förbestämda protokoll som skickas till bilen beroende på vad användaren tryckt på för knapp. Denna data skickas med hjälp av *Communication*-objektet.

GUIPlot

Objekt av typen *GUIPlot* skapas när *Window*-objektet initieras. Klassen *GUIPlot* använder sig av biblioteket *MatPlotLib* och ritar med hjälp av det biblioteket ut grafer när programmet är i sensorläge.

Programmet i allmänhet

I filen *ThreadedClient* efter klassdefinitionen skapas ett objekt av typ *TkInter* som sedan skickas ned till *Window*-objektet via *ThreadedClient*-objektet. Programmets huvudloop startas sedan genom att kalla på *TkInter*-objektets *mainloop*-funktion.

6 Gränssnitt

Tre av systemets delsystem kommunicerar med varandra genom en buss av typ Serial Peripheral Interface (SPI). Styrmodulen agerar master på bussen och bestämmer vilka moduler som kommunicerar samt när detta sker. Detta gör att all kommunikation på bussen går genom styrmodulen. Sensormodulen och kommunikationsmodulen är slavar på bussen. SPI är full-duplex vilket innebär att data går i två riktningar och skickas samt tas emot samtidigt. I och med att SPI är full-duplex finns möjlighet att skicka och ta emot information mellan de två modulerna som kommunicerar samtidigt. SPI-bussen använder tre register SPCR (SPI Control Register), SPSR (SPI Status Register) för initiering och SPDR (SPI Data Register) för överföring.

Styrmodulen och lasermodulen kommunicerar genom blåtand som är ansluten på lasermodulen via en UART-port och som finns inbyggd i den Raspberry Pi som styrmodulen har. Även kommunikationsmodulen använder sig av en blåtandsmodul som är ansluten via en UART-port för att kommunicera med en dator.

Dataöverföring på SPI-bussen sker i paket om 16 bytes som skickas i båda datariktningarna samtidigt. Informationen som skickas på varje individuell byte finns specificerad i tabeller 2, 3 och 4 nedan. Det finns två protokoll som båda gäller mellan styrmodulen och kommunikationsmodulen Detta beror på att det är olika information som behöver skickas olika ofta. Vi valde därför att dela upp försändelsen i olika protokoll och skicka ett av dem oftare. Värdet på byte 15 (Informationsbyte sensor/laser i tabellerna) säger om det är protokoll 1 eller 2 som skickas. Kommunikationen mellan lasermodulen och styrmodulen sker via blåtand och överföring genom UART. Avstånd och vinkel skickas från lasermodulen som en ASCII-sträng enligt tabell 5, med 1-4 karaktärer för avstånd, en avskiljare, 1-3 karaktärer för vinkel och slutligen radbrytning.

Kommunikationen mellan dator och kommunikationsmodulen sker via blåtand och överföring genom UART. Även här är informationen uppdelad i två olika protokoll av samma anledning som mellan styrmodulen och kommunikationsmodulen. Protokoll för detta finns specificerat nedan i tabeller 6 och 7.

Tabell 2: Protokoll för SPI-buss mellan sensormodul och styrmodul

Byte	**Sensor till styr**	**Enhet**	**Byte**	**Styr till sensor**	**Enhet**
0	0xFF	-	0	0xFF	-
1	UL fram-vänster	cm	1	-	-
2	UL fram-center	cm	2	-	-
3	UL fram-höger	cm	3	-	-
4	Halleffektsensor	dm/s	4	-	-
5	Varv	-	5	-	-
6	Checksumma	-	6	-	-
7	-	-	7	-	-
8	-	-	8	-	-
9	-	-	9	-	-
10	-	-	10	-	-
11	-	-	11	-	-
12	-	-	12	-	-
13	-	-	13	-	-
14	-	-	14	-	-
15	-	-	15	-	-

Tabell 3: Protokoll 1 för SPI-buss mellan kommunikationsmodul och styrmodul

Byte	**Kom till styr**	**Enhet**	**Byte**	**Styr till kom**	**Enhet**
0	0xFF	-	0	0xFF	-
1	Informationsbyte regler	-	1	UL fram-vänster	cm
2	K_P	-	2	UL fram-center	cm
3	K_D	-	3	UL fram-höger	cm
4	Autonom/manuell	-	4	Halleffektsensor	dm/s
5	Styrutslag	-	5	Varv	-
6	Hastighet	-	6	Checksumma	-
7	Nödstopp	-	7	Styrbeslut	-
8	Start/stopp	-	8	-	
9	K_{SR}	-	9	-	-
10	-	-	10	-	-
11	-	-	11	-	-
12	-	-	12	-	-
13	-	-	13	-	-
14	-	-	14	-	-
15	-	-	15	Informationsbyte sensor/laser	-

Tabell 4: Protokoll 2 för SPI-buss mellan kommunikationsmodul och styrmodul

Byte	**Kom till styr**	**Enhet**	**Byte**	**Styr till kom**	**Enhet**
0	0xFF	-	0	0xFF	-
1	Informationsbyte regler	-	1	Laser-avstånd	cm
2	K_P	-	2	Laser-avstånd	cm
3	K_D	-	3	Laser-vinkel	grader
4	Autonom/manuell	-	4	Laser-vinkel	grader
5	Styrutslag	-	5	Laser-avstånd	cm
6	Hastighet	-	6	Laser-avstånd	cm
7	Nödstopp	-	7	Laser-vinkel	grader
8	Start/stop	-	8	Laser-vinkel	grader
9	K_{SR}	-	9	Laser-avstånd	cm
10	-	-	10	Laser-avstånd	cm
11	-	-	11	Laser-vinkel	grader
12	-	-	12	Laser-vinkel	grader
13	-	-	13	-	-
14	-	-	14	-	-
15	-	-	15	Informationsbyte sensor/laser	-

Tabell 5: Protokoll för överföring mellan lasermodul och styrmodul

Avstånd	**Avskiljare**	**Vinkel**	**Radbrytning**
1234	;	123	\r\n

Tabell 6: Protokoll 1 för UART-buss mellan kommunikationsmodul och dator

Byte	**Dator till kom**	**Enhet**	**Byte**	**Kom till dator**	**Enhet**
0	0xFF	-	0	0xFF	-
1	Informationsbyte regler	-	1	UL fram-vänster	cm
2	K_P	-	2	UL fram-center	cm
3	K_D	-	3	UL fram-höger	cm
4	Autonom/manuell	-	4	Halleffektsensor	dm/s
5	Styrutslag	-	5	Varv	-
6	Hastighet	-	6	Checksumma	-
7	Nödstopp	-	7	Styrbeslut	-
8	Start/stop	-	8	-	
9	K_{SR}	-	9	-	-
10	-	-	10	-	-
11	-	-	11	-	-
12	-	-	12	-	-
13	-	-	13	-	-
14	-	-	14	-	-
15	-	-	15	Informationsbyte sensor/laser	-

Tabell 7: Protokoll 2 för UART-buss mellan kommunikationsmodul och dator

Byte	**Dator till kom**	**Enhet**	**Byte**	**Kom till dator**	**Enhet**
0	0xFF	-	0	0xFF	-
1	Informationsbyte regler	-	1	Laser-avstånd	cm
2	K_P	-	2	Laser-avstånd	cm
3	K_D	-	3	Laser-vinkel	grader
4	Autonom/manuell	-	4	Laser-vinkel	grader
5	Styrutslag	-	5	Laser-avstånd	cm
6	Hastighet	-	6	Laser-avstånd	cm
7	Nödstopp	-	7	Laser-vinkel	grader
8	Start/stop	-	8	Laser-vinkel	grader
9	K_{SR}	-	9	Laser-avstånd	cm
10	-	-	10	Laser-avstånd	cm
11	-	-	11	Laser-vinkel	grader
12	-	-	12	Laser-vinkel	grader
13	-	-	13	-	-
14	-	-	14	-	-
15	-	-	15	Informationsbyte sensor/laser	-

Tabell 8: Protokoll för informationsbyte regler i tabell 3, 4, 6 och 7

Värde	**Beskrivning**
0xAA	Skickar alla parametrar
0xBB	Skickar K_P
0xCC	Skickar K_D
0xDD	Skickar K_{SR}

Tabell 9: UART data-frame enligt tabell 6 och 7

Funktion	Startbit	Data	Paritetsbit	Stoppbit
Antal bitar	1	8	0	1

7 Slutsatser

Det finns ett antal förbättringar som skulle kunna göras på bilen. En är att kartuppritningen skulle kunna visa hela banan efter att bilen har kört ett varv. Därefter skulle man kunna ha en statisk omgivning med en ikon på en bil som visar hur den rör sig i banan.

Både den reglertekniska delen samt maskininlärningen skulle kunna förbättras för att tillåta en stabilare körning utefter en mer optimal racinglinje, exempelvis att bilen ser kurvor och där kan ta ut svängen. En annan relevant förbättring vore att implementera en bättre hastighetsreglering som t.ex. återkopplar värdet från halleffektsensorn för att köra i en bestämd referenshastighet. Det hade även varit intressant att utveckla bilen så att den kan tävla mot en annan bil som kör samtidigt i banan.

En annan förbättring vore att använda en roterande laser med högre uppdateringsfrekvens och snabbare rotationshastighet, exempelvis RPlidar A2 [1] Det skulle innebära att bilen ser en mer högupplöst bild av omgivningen vilket i sin tur skulle tillåta bättre och mer exakta beslut från maskininlärningsalgoritmen samt stabilare reglering. Även ultraljudssensorerna lämnade en del att önska när det kom till uppdateringsfrekvens och pålitlighet och skulle eventuellt kunna bytas ut till någon annan typ av avståndsmätande sensor, ex IR-sensorer.

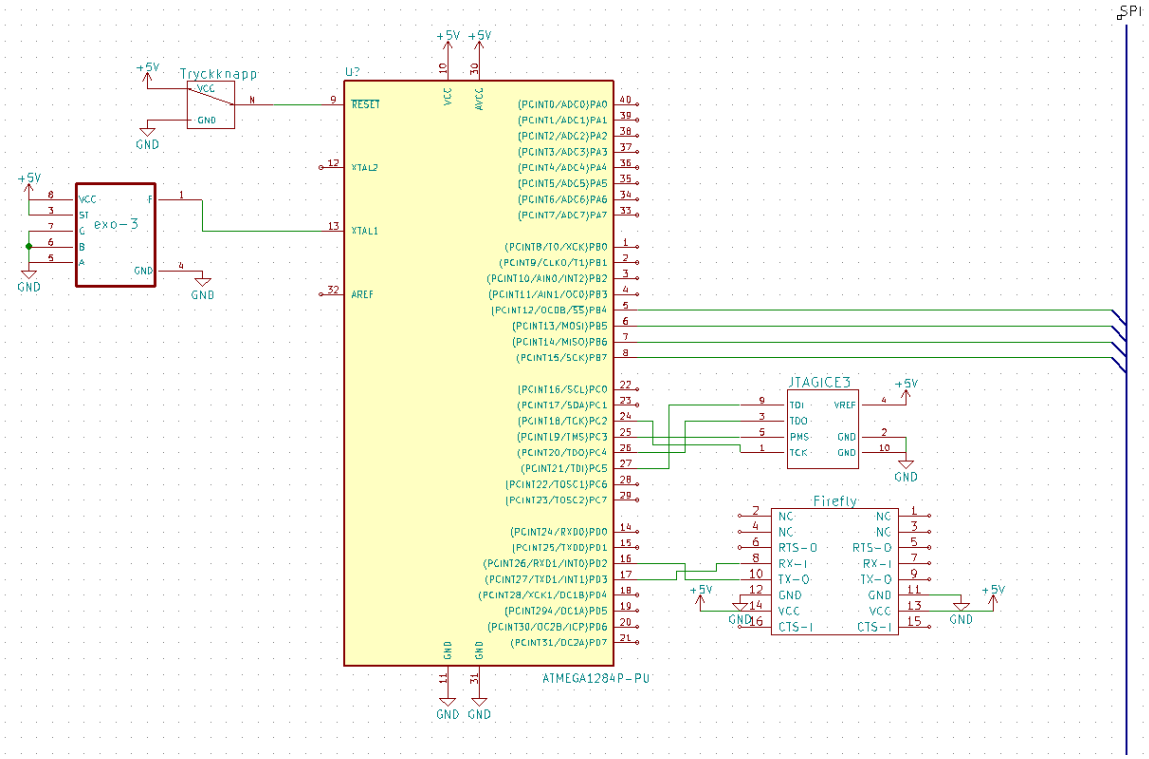
Utöver mjuk- och hårdvaruförbättringar skulle kommunikationsmodulen kunna tas bort helt och ersättas med en USB till UART sladd som kopplar ett FireFlymodem till en av Raspberryns USB-portar. Det skulle innebära lättare implementation, mindre hårdvara och troligtvis snabbare kommunikation.

Referenser

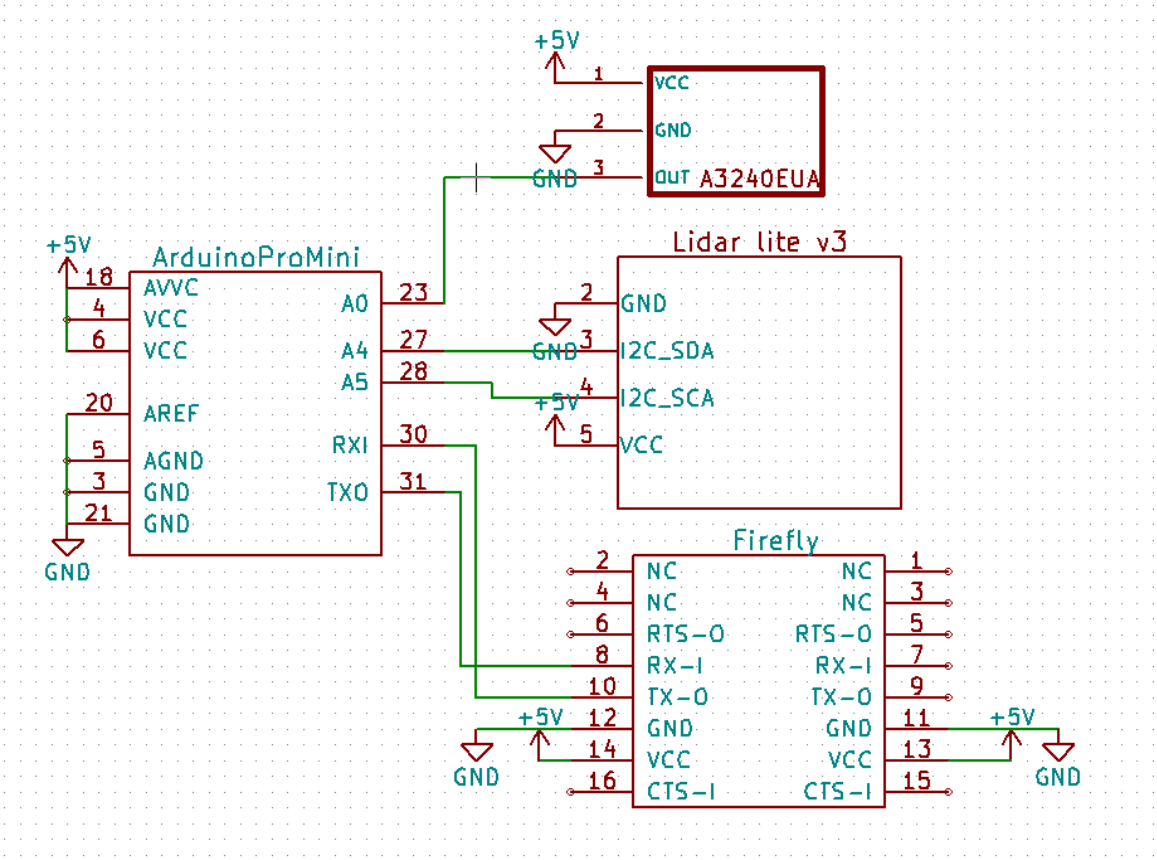
[1] Seeed, “RPLIDAR A2.” [Online]. Available: https://www.seeedstudio.com/RPLIDAR-A2-The-Thinest-LIDAR-p-2687.html

Appendix

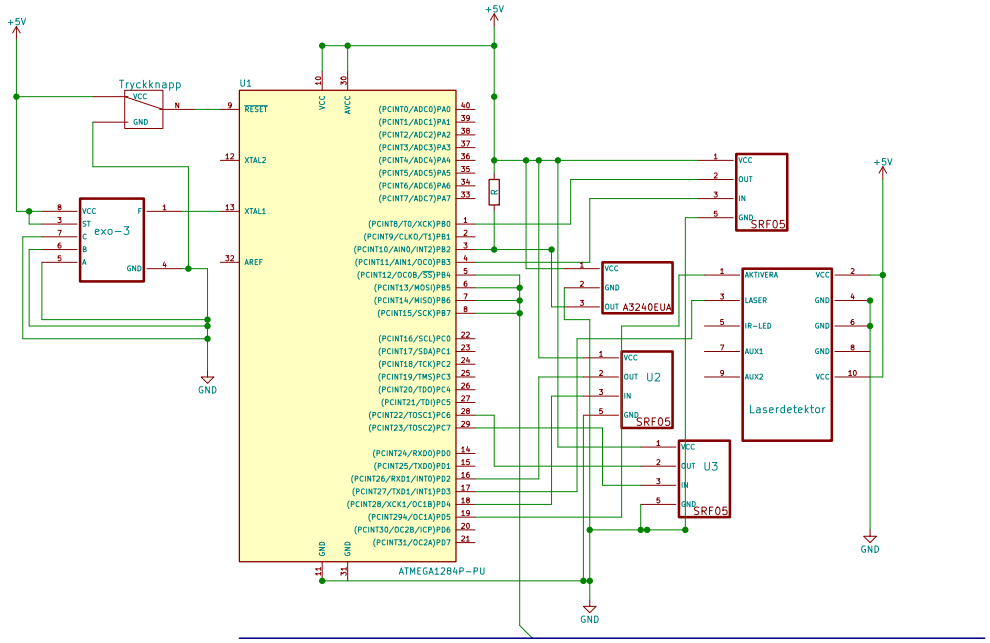
A. Kopplingsscheman



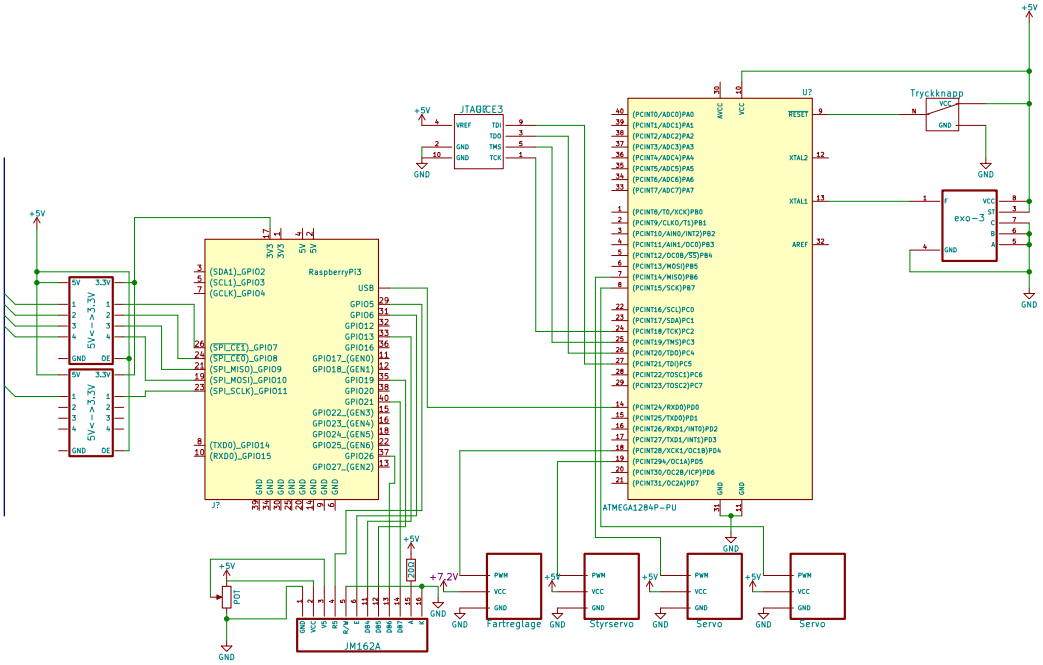
Figur 15: Kopplingsschemat för kommunikationsmodulen.



Figur 16: Kopplingsschemat för lasermodulen.



Figur 17: Kopplingsschemat för sensormodulen.



Figur 18: Kopplingsschemat för styrmodulen.

B. Utdrag ur programlistning

Nedan följer utdrag ur Python-kod från programvaran och styrmodulen, samt C-kod från styrmodulen.

Graphical_interface.py

```
##Authors: Christopher Albinsson, Fredrik Almin, Maria Posluk

from Tkinter import *
from OutputStream import OutputStream
from GUIPlot import GUIPlot

import Queue
import math
#TODO: fixa sa att multiple key press funkar.
import gc
import tkFont
from timeit import default_timer

class Window:
    #Constructor
    def __init__(self,master,queue,comobj,endCommand):

        #Initiate the user interface library with window, frame, serial communication queue and gar
        print master
        gc.enable()
        self.master = master
        self.queue = queue
        frame = Frame(master)
        frame.pack()


        #Initiate class with specified values
        self.key_button_pressed = 0
        self.window_width = 1000
        self.window_height = 600
        self.master.title("Autonomous Car")

        #Initiate important labels
        self.connection_status_text = StringVar()
        self.step_label_text = StringVar()
        self.autonomous_status_text = StringVar()
        self.mode_label_text = StringVar()

        ##--------------------------------------------------Regulation help-----------------------------
        #Initiate regulation start values and stepsize
        self.step = 1
        self.kp = float(16)
        self.kd = float(30)
        self.ksr = float(1)

        ##-----------------------------------GRAPHICS-----------------------------------------
        #Create empty lists for eventual mapping, as well as a help variable for not printing with
        self.map_coords_list_x = []
        self.map_coords_list_y = []
        self.draw_list_x = []
        self.draw_list_y = []
        self.drawlist = []
        self.first_time = 1

        #Create canvas to be drawn upon and graphical font to be used
        self.canvas = Canvas(self.master)
        self.font = tkFont.Font(family="Ebrima", size=18, weight="bold", slant="roman", underline=0

        #Create variables to contain values to be converted into graphical text
        self.current_drive_mode = "CENTER"
```

```
self.current_speed = "0"
self.angle = 0
self.angle_text = "0"
self.distance = 0
self.distance_text = "0"
self.ultra1 = 0
self.ultra2 = 0
self.ultra3 = 0
self.laps = 0

#Create help variable for updating the graphical text for laser values
self.update_lasertext = True

#Create graphical objects on the canvas
self.canvas.create_rectangle(0, 50, 1200, 1200, outline="#40E0B0", fill="black")
self.canvas.create_text(860,525, anchor=NW, font=self.font, text="Speed dm/s",
                        fill="white", activefill="orange")
self.canvas.create_text(10,490, anchor=NW, font=self.font, text="Laser data",
                        fill="white", activefill="orange")
self.canvas.create_text(425, 525, anchor=NW, font=self.font, text="Ultrasounds",
                        fill="white", activefill="orange")
self.canvas.create_text(460, 60, anchor=NW, font=self.font, text="Laps:",
                        fill="white", activefill="orange")

self.driveinfo_graphictext = self.canvas.create_text(10, 60, anchor=NW, font=self.font,
                                                     text=self.current_drive_mode, fill="white", ac
self.speed_graphictext = self.canvas.create_text(900, 560, anchor=NW, font=self.font,
                                                 text=self.current_speed, fill="red", activefill
self.distance_graphictext = self.canvas.create_text(10, 560, anchor=NW, font=self.font,
                                                    text=self.distance_text, fill="red", activefill
self.angle_graphictext = self.canvas.create_text(10, 525, anchor=NW, font=self.font,
                                                 text=self.angle_text, fill="red", activefill="c
self.ultra1_graphictext = self.canvas.create_text(400, 560, anchor=NW, font=self.font,
                                                  text=str(self.ultra1), fill="red", activefill="
self.ultra2_graphictext = self.canvas.create_text(600, 560, anchor=NW, font=self.font,
                                                  text=str(self.ultra2), fill="red", activefill="
self.ultra3_graphictext = self.canvas.create_text(500, 560, anchor=NW, font=self.font,
                                                  text=str(self.ultra3), fill="red", activefill="
self.kp_graphictext = self.canvas.create_text(775, 60, anchor=NW, font=self.font,
                                              text=str(self.kp / 20), fill="red", activefill=
self.kd_graphictext = self.canvas.create_text(855, 60, anchor=NW, font=self.font,
                                              text=str(self.kd / 5), fill="red", activefill="
self.ksr_graphictext = self.canvas.create_text(935, 60, anchor=NW, font=self.font,
                                               text=str(self.ksr / 200), fill="red", activefil
self.lap_graphictext = self.canvas.create_text(520, 60, anchor=NW, font=self.font,
                                               text=str(self.laps), fill="red", activefill="or

#Load car-image from file and draw it on the canvas
self.photo = PhotoImage(file='bil2.gif')
self.photo = self.photo.subsample(2)
self.photo_label = Label(image=self.photo)
self.photo_label.image = self.photo
self.canvas.create_image(self.window_width / 2, self.window_height / 2, image=self.photo)

#Pack everything on the canvas
self.canvas.pack(fill=BOTH, expand=1)

#Initiate the user interface
self.initUI()

##-----------------------------------Communications-----------------------------------
```

```
#Specify that the input communications object is to be used in this class as well. Initiate
self.output_queue = Queue.Queue()
self.com = comobj

#Initiate connection status text with value according to connection activity
if self.com.connection_is_open == True:
    self.connection_status_text.set("connection")
else:
    self.connection_status_text.set("no_connection")

#Initiate output stream with communication object
self.output_stream = OutputStream(self.com)



##-------------------------------------------------------BUTTONS-------------------------------------------------------

#Create UI-buttons with text, font, size and trigger function
self.b1 = Button(master, text = "Run", command=self.run_mode, width = 10,height = 1)
self.b1.place(x=50,y=0)
self.b2 = Button(master, text = "Sensor", command=self.sensor_mode, width = 10, height = 1)
self.b2.place(x=200,y = 0)
self.b3 = Button(master, text = "man/auto",command=self.switch_drive_mode, width = 10, heig
self.b3.place(x=350,y=0)
self.b4 = Button(master, text = "Kp_+", command=self.add_kp, width = 10,height = 1)
self.b4.place(x=self.window_width-240, y=0)
self.b5 = Button(master, text = "Kd_+", command=self.add_kd, width=10, height=1)
self.b5.place(x=self.window_width-160, y = 0)
self.b6 = Button(master, text = "Ksr_+", command=self.add_ksr, width=10, height=1)
self.b6.place(x=self.window_width-80, y=0)
self.b7 = Button(master, text = "Kp_-", command=self.sub_kp, width = 10,height = 1)
self.b7.place(x=self.window_width-240, y=25)
self.b8 = Button(master, text = "Kd_-", command=self.sub_kd, width=10, height=1)
self.b8.place(x=self.window_width-160, y = 25)
self.b9 = Button(master, text = "Ksr_-", command=self.sub_ksr, width=10, height=1)
self.b9.place(x=self.window_width-80, y=25)
self.b10 = Button(master, text = "Step", command=self.change_step, width=10, height=1)
self.b10.place(x=self.window_width-320, y=0)
self.b11 = Button(master, text = "Init", command=self.set_standard_regulation, width=10, he
self.b11.place(x=self.window_width-400, y=0)

#Specify values for textlabels
#Regulation stepsize label
self.step_label = Label(self.master, textvar = self.step_label_text, width=10,height = 1)
self.step_text = "1"
self.step_label_text.set(self.step_text)
self.step_label.place(x=self.window_width-320, y=30)

#Program mode label text (Run mode/Sensor mode)
self.mode_label = Label(self.master, textvar = self.mode_label_text,width = 10,height = 1)
self.mode = "run_mode"
self.mode_label_text.set(self.mode)
self.mode_label.place(x = 50,y = 30)

#Drive mode label text (Autonomous/Manual)
self.drive_mode = "manual"
self.drive_mode_text = StringVar()
self.drive_mode_label = Label(self.master,textvar = self.drive_mode_text, width = 10 , heig
self.drive_mode_text.set(self.drive_mode)
self.drive_mode_label.place(x = 350,y = 30) #distance_between_menu_buttons*4)
```

```
        ##Switch mode twice to initiate both modes
        self.switch_drive_mode()
        self.switch_drive_mode()

        ##--------------------status texts--------------

        #Specify values for status texts (connection on/off, autonomous on/off)
        self.connection_status = Label(self.master,textvar = self.connection_status_text, width = 1
        self.connection_status.place(x = 200, y = 30) #distance_between_menu_buttons*5)

        self.autonomous_label = Label(self.master, text="Auto:", width=10, height=1)
        self.autonomous_label.place(x=430,y=0)

        self.autonomous_status = "off"
        self.autonomous_status_label = Label(self.master, textvar=self.autonomous_status_text, widt
        self.autonomous_status_label.place(x=490,y=0)
        self.autonomous_status_text.set(self.autonomous_status)

        ##-------------------------------------------------WINDOW-------------------------------------------------

        ##set window size to fixed.
        self.master.maxsize(self.window_width,self.window_height)
        self.master.minsize(self.window_width,self.window_height)

        #Create input_byte in bytearray format to handle incoming hexvalues from serial port
        self.input_byte = bytearray()


##-----------------------------------GUI functions-----------------------------------

    #Initiates the user interface, creating and then deleting graphical objects
    def initUI(self):
        for i in range(0, 100):
            self.drawlist.append(self.canvas.create_line(150, 200, 850, 200, fill="white", width=5)
        self.update_ui()

    #Updates the user interface. Deletes graphical objects from earlier iteration and creates new o
    #Also updates graphical text objects.
    def update_ui(self):

        for i in range(0,100):
            #try:
            self.canvas.delete(self.drawlist[i])
            #except:
            #    print "Couldn't delete object in drawlist - because no object?"

        self.update_graphictext()


        if not self.first_time:
            for i in range(0,99):
                #Creates a line between two map coordinates if they satisfy required intervals.
                if abs(self.draw_list_x[i+1] - self.draw_list_x[i]) < 15\
                        or abs(self.draw_list_y[i+1] - self.draw_list_y[i] < 5):

                    self.drawlist[i] = (self.canvas.create_line(
                                        self.draw_list_x[i],
                                        self.draw_list_y[i],
                                        self.draw_list_x[i+1],
                                        self.draw_list_y[i+1],
                                        fill="white", width=5))
                else:
```

```
                print "Skipped line creation because of distance between points"
                pass


    def update_graphictext(self):
        self.canvas.itemconfig(self.driveinfo_graphictext, text=str(self.current_drive_mode))
        self.canvas.itemconfig(self.speed_graphictext, text=self.current_speed)
        self.canvas.itemconfig(self.distance_graphictext, text=self.distance_text)
        self.canvas.itemconfig(self.angle_graphictext, text=self.angle_text)
        self.canvas.itemconfig(self.ultra1_graphictext, text=str(self.ultra1))
        self.canvas.itemconfig(self.ultra2_graphictext, text=str(self.ultra2))
        self.canvas.itemconfig(self.ultra3_graphictext, text=str(self.ultra3))
        self.canvas.itemconfig(self.lap_graphictext, text=str(self.laps))

##-------------------------Mappings functions-------------------------------------

    def map_help(self):
        if len(self.map_coords_list_x) > 100 and len(self.map_coords_list_y) > 100:
            self.update_map()
            self.map_coords_list_x = []
            self.map_coords_list_y = []

    def update_map(self):
        self.draw_list_x = self.map_coords_list_x
        self.draw_list_y = self.map_coords_list_y

        self.update_lasertext = True

        if self.first_time:
            self.first_time = 0

##-------------------------------handle incoming data function-------------------------------

    def handle_indata(self):
        #for value 0xaa in protocol last byte
        if int(self.input_byte[15], 0) == 170:
            #Ultrasound
            if self.mode == "sensor mode":
                self.ultrasound_list = [int(self.input_byte[1], 0),
                                        int(self.input_byte[2], 0),
                                        int(self.input_byte[3], 0)]

            elif self.mode == "run mode":
                self.ultra1 = int(self.input_byte[1], 0)
                self.ultra2 = int(self.input_byte[2], 0)
                self.ultra3 = int(self.input_byte[3], 0)

            #Speed
            if self.mode == "run mode":
                self.current_speed = str(int(self.input_byte[4], 0))

            #Laps
            if self.mode == "run mode":
                self.laps = int(self.input_byte[5], 0)

            #Steering instructions
            if int(self.input_byte[7], 0) == 0:
                self.current_drive_mode = "LEFT"
            elif int(self.input_byte[7], 0) == 1:
                self.current_drive_mode = "CENTER"
            elif int(self.input_byte[7], 0) == 2:
                self.current_drive_mode = "RIGHT"
```

```
##-----------------------------------Run mode functions-----------------------------------------------

    #Changes mode to run_mode
    def run_mode(self):

        if self.mode == "sensor_mode":
            self.clean_up_sensor_mode()
        print "changed_to_run_mode"
        self.draw_runmode()
        self.mode = "run_mode"
        self.mode_label_text.set(self.mode)

    #Draws the run mode interface
    def draw_runmode(self):
        self.canvas.delete("all")
        self.canvas.create_rectangle(0, 50, 1200, 1200, outline="#40E0B0", fill="black")
        self.canvas.create_text(860,525, anchor=NW, font=self.font, text="Speed_dm/s",
                                fill="white", activefill="orange")
        self.canvas.create_text(10,490, anchor=NW, font=self.font, text="Laser_data",
                                fill="white", activefill="orange")
        self.canvas.create_text(450, 525, anchor=NW, font=self.font, text="Ultrasounds",
                                fill="white", activefill="orange")
        self.canvas.create_text(460, 60, anchor=NW, font=self.font, text="Laps:",
                                fill="white", activefill="orange")

        self.driveinfo_graphictext = self.canvas.create_text(10, 60, anchor=NW, font=self.font,
                                                             text=self.current_drive_mode, fill="white", ac
        self.speed_graphictext = self.canvas.create_text(900, 560, anchor=NW, font=self.font,
                                                             text=self.current_speed, fill="red", activefill
        self.distance_graphictext = self.canvas.create_text(10, 560, anchor=NW, font=self.font,
                                                             text=self.distance_text, fill="red", activefill
        self.angle_graphictext = self.canvas.create_text(10, 525, anchor=NW, font=self.font,
                                                             text=self.angle_text, fill="red", activefill="c
        self.ultra1_graphictext = self.canvas.create_text(400, 560, anchor=NW, font=self.font,
                                                             text=str(self.ultra1), fill="red", activefill="
        self.ultra2_graphictext = self.canvas.create_text(600, 560, anchor=NW, font=self.font,
                                                             text=str(self.ultra2), fill="red", activefill="
        self.ultra3_graphictext = self.canvas.create_text(500, 560, anchor=NW, font=self.font,
                                                             text=str(self.ultra3), fill="red", activefill="
        self.kp_graphictext = self.canvas.create_text(775, 60, anchor=NW, font=self.font,
                                                             text=str(self.kp / 20), fill="red", activefill=
        self.kd_graphictext = self.canvas.create_text(855, 60, anchor=NW, font=self.font,
                                                             text=str(self.kd / 5), fill="red", activefill="
        self.ksr_graphictext = self.canvas.create_text(935, 60, anchor=NW, font=self.font,
                                                             text=str(self.ksr / 200), fill="red",activefill
        self.lap_graphictext = self.canvas.create_text(520, 60, anchor=NW, font=self.font,
                                                             text=str(self.laps), fill="red", activefill="or

        self.canvas.create_image(self.window_width / 2, self.window_height / 2, image=self.photo)

        self.update_ui()

##-------------------------Regulation UI functions-----------------------------------

    #Each of the following functions are triggered by presses on the regulation Buttons on the HUD
    #They either add or subtract regulator parameters, or returns them to the standard values.

    def add_kp(self):
        if self.kp + self.step <= 255:
            self.kp = self.kp + self.step
            self.output_stream.regulate("kp", self.kp, 0, 0)
```

```
            self.canvas.itemconfig(self.kp_graphictext, text=str(self.kp / 20))

    def add_kd(self):
        if self.kd + self.step <= 255:
            self.kd = self.kd + self.step
            self.output_stream.regulate("kd", 0, self.kd, 0)
            self.canvas.itemconfig(self.kd_graphictext, text=str(self.kd / 5))

    def add_ksr(self):
        if self.ksr + self.step <= 255:
            self.ksr = self.ksr + self.step
            self.output_stream.regulate("ksr", 0, 0, self.ksr)
            self.canvas.itemconfig(self.ksr_graphictext, text=str(self.ksr / 200))

    def sub_kp(self):
        if self.kp - self.step >= 0:
            self.kp = self.kp - self.step
            self.output_stream.regulate("kp",self.kp,0,0)
            self.canvas.itemconfig(self.kp_graphictext, text=str(self.kp / 20))

    def sub_kd(self):
        if self.kd - self.step >= 0:
            self.kd = self.kd - self.step
            self.output_stream.regulate("kd",0,self.kd,0)
            self.canvas.itemconfig(self.kd_graphictext, text=str(self.kd / 5))

    def sub_ksr(self):
        if self.ksr - self.step >= 0:
            self.ksr = self.ksr - self.step
            self.output_stream.regulate("ksr",0,0,self.ksr)
            self.canvas.itemconfig(self.ksr_graphictext, text=str(self.ksr / 200))

    def change_step(self):
        if self.step_text == "1":
            self.step = 10
            self.step_text = "10"
            self.step_label_text.set(self.step_text)
        elif self.step_text == "10":
            self.step = 1
            self.step_text = "1"
            self.step_label_text.set(self.step_text)

    def set_standard_regulation(self):
        self.kp = float(16)
        self.kd = float(30)
        self.ksr = float(1)
        self.output_stream.regulate("all", self.kp, self.kd, self.ksr)
        self.update_graphictext()

##-------------------------Key press events-------------------------------

    #Each of the following functions changes self.key_button_pressed
    #according to which button was pressed most recently

    def left_key_press(self,event):
        self.key_button_pressed = 1

    def right_key_press(self,event):
        self.key_button_pressed = 2

    def up_key_press(self,event):
        self.key_button_pressed = 3
```

```
    def down_key_press(self,event):
        self.key_button_pressed = 4

    def space_key_press(self,event):
        self.output_stream.add(5)

    def s_key_press(self,event):

        #If autonomous, changes the value of the autonomous status text
        if self.drive_mode == "autonomous":
            if self.autonomous_status == "on":
                self.output_stream.add(7)
                self.autonomous_status = "off"
                self.autonomous_status_text.set(self.autonomous_status)
            elif self.autonomous_status == "off":
                self.output_stream.add(6)
                self.autonomous_status = "on"
                self.autonomous_status_text.set(self.autonomous_status)

##-------------------------Sensor mode functions-------------------------

    #Change to sensor mode
    def sensor_mode(self):

        if self.mode == "sensor mode":
            self.clean_up_sensor_mode()

        if self.first_time == 1:
            self.first_time = 0
        self.create_plots()
        self.update_plots([100,100,100])
        self.switch_drive_mode()
        self.switch_drive_mode()

        self.draw_sensormode()
        self.mode = "sensor mode"
        self.mode_label_text.set(self.mode)

        gc.collect()

    #Draws the sensor mode interface
    def draw_sensormode(self):
        self.canvas.delete("all")
        self.canvas.create_rectangle(0, 50, 1200, 1200, outline="black", fill="white")
        self.canvas.create_text(200,70,anchor=NW,font=self.font, text="ultra left", fill="red")
        self.canvas.create_text(700,70,anchor=NW,font=self.font, text="ultra right", fill="red")
        self.canvas.create_text(450,self.window_height-50,anchor=NW,font=self.font,
                                text="ultra front", fill="red")

    #Update plots in sensor mode
    def update_plots(self, ultrasoundlist):
        self.plot1.update_plot(ultrasoundlist[0])
        self.plot2.update_plot(ultrasoundlist[1])
        self.plot3.update_plot(ultrasoundlist[2])

    #Creates the plots in sensormode
    def create_plots(self):
        ##generate and place plots
        self.plot1 = GUIPlot(0,100,5,2,'plot 1',self.master)
        self.plot2 = GUIPlot(260,340,5,2,'plot 2',self.master)
        self.plot3 = GUIPlot(520,100,5,2,'plot 3',self.master)
```

```
    #Hides plots
    def clean_up_sensor_mode(self):
        self.master.grid_propagate(0)
        self.plot1.hide_plot()
        self.plot2.hide_plot()
        self.plot3.hide_plot()


##-----------------------CHANGE MODE FUNCTIONS-----------------------

    #Switches between autonmous or manual mode on the car.
    def switch_drive_mode(self):

        if self.drive_mode == "autonomous":

            self.master.bind('<Left>', self.left_key_press)
            self.master.bind('<Right>', self.right_key_press)
            self.master.bind('<Up>', self.up_key_press)
            self.master.bind('<Down>', self.down_key_press)
            self.master.bind('<space>', self.space_key_press)
            self.master.unbind('<s>')

            self.autonomous_status_text.set("off")
            self.drive_mode = "manual"
            self.drive_mode_text.set(self.drive_mode)

            self.output_stream.add(8)
            print "switching from autonomous to manual"

        elif self.drive_mode == "manual":
            self.drive_mode = "autonomous"
            self.drive_mode_text.set(self.drive_mode)

            self.master.unbind('<Left>')
            self.master.unbind('<Right>')
            self.master.unbind('<Up>')
            self.master.unbind('<Down>')
            self.master.unbind('<space>')
            self.master.bind('<s>', self.s_key_press)

            self.output_stream.add(9)
            print "switching from manual to autonomous"

##-----------------------HANDLE INCOMING DATA FUNCTIONS-----------------------

    #Handle incoming laser protocol
    #Computes window coordinates in regard to what laser values were retreived from the car
    def update_lasers(self, laser_list):
        if not laser_list == []:
            for i in range(0, len(laser_list)):
                laserlist = laser_list[i]

                dist1 = (int(laserlist[1], 0)*256 + int(laserlist[2], 0))*1.5
                dist2 = (int(laserlist[5], 0)*256 + int(laserlist[6], 0))*1.5
                dist3 = (int(laserlist[9], 0)*256 + int(laserlist[10], 0))*1.5
                rangle1 = int(laserlist[3], 0)*256 + int(laserlist[4], 0) + 78
                rangle2 = int(laserlist[7], 0)*256 + int(laserlist[8], 0) + 78
                rangle3 = int(laserlist[11], 0)*256 + int(laserlist[12], 0) + 78

                angle1 = rangle1 *(math.pi/180)
                angle2 = rangle2 *(math.pi/180)
```

```
            angle3 = rangle3 *(math.pi/180)

            # Laserdata text
            if self.update_lasertext:
                if self.mode == "run_mode":
                    self.distance = (dist1+dist2+dist3)/3
                    self.distance_text = str(self.distance)
                    self.angle = (rangle1+rangle2+rangle3)
                    self.angle_text = str(self.angle)
                    self.update_lasertext = False

            self.map_coords_list_x.append(self.window_width / 2 + dist1 * math.cos(angle1))
            self.map_coords_list_y.append(self.window_height / 2 - dist1 * math.sin(angle1))
            self.map_coords_list_x.append(self.window_width / 2 + dist2 * math.cos(angle2))
            self.map_coords_list_y.append(self.window_height / 2 - dist2 * math.sin(angle2))
            self.map_coords_list_x.append(self.window_width / 2 + dist3 * math.cos(angle3))
            self.map_coords_list_y.append(self.window_height / 2 - dist3 * math.sin(angle3))

#Handle incoming sensor and other protocol
#Grabs object from input queue and handles data accordingly
def process_incoming(self):
        try:
            #print "process incoming?"
            self.input_byte = self.queue.get(0)
            #print self.input_byte
            #print self.com.ser.in_waiting
            self.handle_indata()
            if self.mode == "sensor_mode":
                self.update_plots(self.ultrasound_list)
            if self.mode == "run_mode":
                self.map_help()
                self.update_ui()
            self.output_stream.add(self.key_button_pressed)
            self.key_button_pressed = 0
        except Queue.Empty:
            #print "failed queue.get"
            pass
```

```
##Authors: Johannes Grundell, Alexander Barlund, Dennis Edblom
## Main file, run this to run the program

import sys
import SPIclass
import Enginecontrol
import Laser
import PD
import Display
import Variables
import machine_learning
import List_Maker

from timeit import default_timer as timer
import time

def main():

    #The reference line for the car to follow
    refline = 1

    #Creating instances of classes

    #Object creating lists for machine learning
    listmaker = List_Maker.List_Maker()

    #Instance of our machine learning algorithm
    ml = machine_learning.machine_learning('neural_net_800x4_2017_05_21.pkl')

    #Object handling the communication with the pulsegenerator
    car = Enginecontrol.Enginecontrol()

    #Create an instance of our regulator
    pd_reg = PD.PD()

    #Object representing the display
    display = Display.Display()

    #Object containing flags, stop and autonomous/manual
    variables = Variables.Variables()

    #Object handling the communication with the communicationsmodule
    SPI_kom = SPIclass.SPI_kom(0,car,pd_reg,variables)

    #Object handling the communication with the sensormodule
    SPI_sensor = SPIclass.SPI_sensor(1)
```

```
#Object handling the communication with the lasertower
laser = Laser.Laser()

#Timevariables for keeping the execution frequency somewhat constant
t_sensor = 0
t_regler = 0
t_laser = 0
t_kom_sensor = 0
t_kom_laser = 0
t_display = 0

#flush printbuffer
sys.stdout.flush()

#Local list for holding sensordata
sensor_data = [0]*16

#Initiate local variables
laser_send_list = []

#Sleep for 5 seconds to let connections establish
time.sleep(5)

#MAIN-LOOP
while 1:

    if variables.man_auto_flag == 1: #autonomous

        #Get new sensorvalues
        if(timer()-t_sensor) > 0.05: #Do this 20 times/sec
            SPI_sensor.transfer() #Read data

            if SPI_sensor.checksum(): #If the checksums match
                sensor_data = SPI_sensor.read_data #Save the data
            #If we have gone three laps, stop.
            #Push RESET on the car before staring over
            if sensor_data[5] == 4:
                variables.stop = 1
                car.set_default()
            t_sensor = timer()
```

```
#Update steering and speed
if((timer()-t_regler) > 0.05) and (variables.stop == 0):
    #Steer algorithm
    u = pd_reg.steer_algorithm(sensor_data[1], #Left ultrasound
                               sensor_data[3], #Middle ultrasound
                               sensor_data[2], #Right ultrasound
                               listmaker.largest_distance_angle)
                               #Pair of angle and greatest distan

    #Speed algorithm
    s = pd_reg.speed_algorithm(sensor_data[4],
                               sensor_data[2],
                               listmaker.largest_distance_angle)
    #Update values and send
    car.update_steer(u)
    car.update_speed(s)
    car.send_data()

    t_regler = timer()



#Get laservalues and execute machine learning algorithm
if(timer()-t_laser) > 0.01: #Do this 100 times/sec
    laser_list=laser.update_laser()

    if (laser_list != []):
        laser_send_list += laser_list

    listmaker.add_to_list(laser_list)

    #If there is a finished list of laservalues
    #for machine learning
    if listmaker.new_list_ready and (variables.stop == 0):
        ml_laser_list = listmaker.get_new_list()
        #Calculate a reference line
        refline = ml.predict(ml_laser_list)
        #Send the determined refrenceline to the steeralgorithm
        pd_reg.set_ref(refline)

    t_laser = timer()
```

```
            #Print on display
            if(timer()-t_display) > 0.5: #Do this 2 times/sec
                display.write_sensors("V:" + str(sensor_data[1]),
                                      "␣H:" + str(sensor_data[3]),
                                      "␣E:" + str(pd_reg.e[9]),
                                      "S:" + str(sensor_data[4]))
                t_display = timer()

            #Send laser values and read returned data
            #from the communications module
            if(timer()-t_kom_laser) > 0.01: #Do this 100 times/sec

                while(len(laser_send_list) > 3):
                    for i in range(1,13,4):
                        tmp = laser_send_list.pop(0)
                        SPI_kom.send_data[i] = 255&(tmp[0]/256)
                        SPI_kom.send_data[i+1] = 255&tmp[0]
                        SPI_kom.send_data[i+2] = 255&(tmp[1]/256)
                        SPI_kom.send_data[i+3] = 255&tmp[1]
                    SPI_kom.send_data[0]= 0xFF
                    SPI_kom.send_data[15] = 0xBB
                    SPI_kom.transfer() #Send and recieve data
                    SPI_kom.send_data = [0]*16
                    SPI_kom.decode() #Decode the recieved data

                t_kom_laser = timer()

            #Send sensorvalues and read returned data
            #from the communications module
            if(timer()-t_kom_sensor) > 0.1: #Do this 10 times/sec
                SPI_kom.send_data = sensor_data
                SPI_kom.send_data[15] = 0xAA
                SPI_kom.send_data[7] = refline
                SPI_kom.transfer() #Send and recieve data
                SPI_kom.send_data = [0]*16
                SPI_kom.decode() #Decode the recieved data

                t_kom_sensor = timer()
#-------------------------------------------------------------------------------
        else: #manual
            #Get new sensorvalues
            if(timer()-t_sensor) > 0.05: #Do this 20 times/sec
               SPI_sensor.transfer() #Read data
               if SPI_sensor.checksum(): #If the checksums match
                   sensor_data = SPI_sensor.read_data #Save the data
               t_sensor = timer()
```

```
#Send sensorvalues and read returned data
#from the communications module
if(timer()-t_kom_sensor) > 0.1:
    SPI_kom.send_data = sensor_data
    SPI_kom.send_data[15] = 0xAA
    SPI_kom.transfer() #Send and recieve data
    SPI_kom.send_data = [0]*16
    SPI_kom.decode() #Decode the recieved data

    t_kom_sensor = timer()

#Print on display
if(timer()-t_display) > 0.5: #Do this 2 times/sec
    display.write_sensors("V:" + str(sensor_data[1]),
                          "␣H:" + str(sensor_data[3]),
                          "␣E:" + str(pd_reg.e[9]),
                          "S:" + str(sensor_data[4]))
    t_display = timer()

#Get laservalues
if(timer()-t_laser) > 0.01: #Do this 100 times/sec
    laser_list=laser.update_laser()

    if (laser_list != []):
        laser_send_list += laser_list
    t_laser = timer()

#Send laser values and read returned data
#from the communications module
if(timer()-t_kom_laser) > 0.01: #Do this 100 times/sec

    while(len(laser_send_list) > 3):
        for i in range(1,13,4):
            tmp = laser_send_list.pop(0)
            SPI_kom.send_data[i] = 255&(tmp[0]/256)
            SPI_kom.send_data[i+1] = 255&tmp[0]
            SPI_kom.send_data[i+2] = 255&(tmp[1]/256)
            SPI_kom.send_data[i+3] = 255&tmp[1]
        SPI_kom.send_data[0]= 0xFF
        SPI_kom.send_data[15] = 0xBB
        SPI_kom.transfer()
        SPI_kom.send_data = [0]*16
        SPI_kom.decode()
    t_kom_laser = timer()
```

```
## Kor main
if __name__ == "__main__":
    main()
```

```
/*
 * styrmodul.c
 *
 * Created: 3/22/2017 8:27:20 AM
 *  Author: Johannes Grundell, Alexander Barlund
 */


#include <avr/io.h>
#include <avr/interrupt.h>

//Initializes the PWM pulse values

//styrpuls is a value between 0-255 and is later converted
//to a pulse width between 0-1ms
uint8_t styrpuls = 128;

//fartpuls is a value between 0-255 and is later converted
//to a pulse width between 0-1ms
uint8_t fartpuls = 128;

//sensorservo is an array containing values between 0-255 that
//is later used to calculate the mean
uint8_t sensorservo[6] = {128};

//sensorservopuls is a value between 0-255 and
//is later calculated as the mean of sensorservo
uint8_t sensorservopuls = 128;

int UARTcounter = 0; //Initializes the UART counter to 0


//Interrupt method for UART0
ISR(USART0_RX_vect)
{
        int k = 0;

    //Check if we recieve a header (0xFF)
        if((UARTcounter == 0) && (UDR0 == 0xFF))
        {
                UARTcounter = 1;
        }
```

```
        //The second recieved byte is the value for the steering pulse
            else if(UARTcounter == 1)
            {
                    styrpuls = UDR0;
                    OCR1A = 1843+styrpuls*7;

                    for(k=0; k < 5; k++)
                    {
                            sensorservo[k] = sensorservo[k+1];
                    }
                    sensorservo[5] = styrpuls;

                    //sensorservopuls is the pulse that determines the position
                    //of the servos that move the ultrasound sensors on the car
                    //sensorservopuls is the mean of the six values
                    //in sensorservo
                    sensorservopuls = (sensorservo[0] + sensorservo[1]
                                    sensorservo[2] + sensorservo[3] +
                                    sensorservo[4] + sensorservo[5])/6;

                    if (sensorservopuls >= 128)
                    {
                            OCR3A = 1843+(sensorservopuls)*2;
                            OCR3B = 1843+(128)*2;
                    }
                    else
                    {
                            OCR3B = 1843+(sensorservopuls)*2;
                            OCR3A = 1843+(128)*2;
                    }

                    UARTcounter = 2;
            }
        //The third recieved byte is the value for the speed pulse
            else if(UARTcounter == 2)
            {
                    fartpuls = UDR0;
                    OCR1B = 1843+fartpuls*7;
                    UARTcounter = 0;
            }

}
```

```
int main(void)
{
        //Timer1 handles PWM for the steering and speed
        //Timer3 handles PWM for the ultrasound servos

        //Sets the needed ports as output or input
        DDRD |= (1 << DDD5) | (1 << DDD4) | (0 << DDD3) |
                (0 << DDD2) | (0 << DDD0);
        DDRB |= (1 << DDB6) | (1 << DDB7);

        //Sets up TIMER1 16-bit for PWM
        TCCR1A |= (1 << COM1A1) | (0 << COM1A0) |(1 << COM1B1) |
                  (0 << COM1B0) | (1 << WGM11) | (0 << WGM10);
        TCCR1B |= (0 << ICNC1) | (0 << ICES1) | (1 << WGM13) |
                  (1 << WGM12) |  (0 << CS12) | (1 << CS11) | (0 << CS10);

        //Sets up TIMER3 16-bit for PWM
        TCCR3A |= (1 << COM3A1) | (0 << COM3A0) |(1 << COM3B1) |
                  (0 << COM3B0) | (1 << WGM31) | (0 << WGM30);
        TCCR3B |= (0 << ICNC3) | (0 << ICES3) | (1 << WGM33) |
                  (1 << WGM32) | (0 << CS32) | (1 << CS31) | (0 << CS30);

        //Sets ICR1 to 20ms and the output compares to the initial values
        ICR1 = 36864; //36864 is 20ms in clock cycles for the processor
        TCNT1 = 0;
        //1843 is 36864/20=1ms in clock cycles.
        //1843+styrpuls*7 is between 1-2ms
        OCR1A = 1843+styrpuls*7;
        OCR1B = 1843+fartpuls*7;

        //Sets ICR3 to 20ms and the output compares to the initial values
        ICR3 = 36864;
        TCNT3 = 0;
        OCR3A = 1843+sensorservopuls*2;
        OCR3B = 1843+sensorservopuls*2;

        //Sets up UART0 for UART transfers
        UCSR0A |= (0 << U2X0);
        UCSR0B |= (1 << RXCIE0) | (1 << RXEN0)| (0 << UCSZ02);
        UCSR0C |= (0 << UMSEL01) | (0 << UMSEL00) | (1 << UCSZ01) |
                  (1 << UCSZ00) | (0 << UCPOL0);
        UBRR0 = 7; //Sets the BAUD rate to 115200


        //Global interrupt enable
        sei();
```

```
    while(1)
    {

    }

return 0;
}
```