**Teknisk dokumentation**

G07

Version 1.0

Status

| Granskad | FR, FN, JV, CN, ML | 2023-12-12 |
| --- | --- | --- |
| Godkänd | AN | 2023-12-12 |

**S.A.N.T.A M.A.R.I.A**

Grupp 07, 2023HT  
Linköpings tekniska högskola, ISY

| **Namn** | **Ansvar** | **Telefon** | **E-post** |
| --- | --- | --- | --- |
| Felix Ramnelöv | Projektledare (PL) och Dokumentansvarig (DOK) | 073-510 14 07 | [felra653@student.liu.se](mailto:felra653@student.liu.se) |
| Filip Nygren | Versionshantering (GIT) och Systemarkitekt (SA) | 076-059 66 76 | [filny841@student.liu.se](mailto:filny841@student.liu.se) |
| Hannes Lindström | UX-designer (UX) och Implementationsansvarig (IMP) | 070-847 82 39 | [hanli001@student.liu.se](mailto:hanli001@student.liu.se) |
| Jacob Volz | Leveransansvarig (LEV) och Ekonom (EKO) | 076-306 07 27 | [jacvo343@student.liu.se](mailto:jacvo343@student.liu.se) |
| Christoffer Näs | Integrationsansvarig (ITG) | 072-568 45 61 | [chrna581@student.liu.se](mailto:chrna581@student.liu.se) |
| Mikael Lundgren | Testansvarig (QA) | 070-754 76 70 | [miklu523@student.liu.se](mailto:miklu523@student.liu.se) |

**E-postlista för hela gruppen**: [TSEA29\_2023HT\_XX-Grupp7@groups.liu.se](mailto:TSEA29_2023HT_XX-Grupp7@groups.liu.se)

**Kund:** Anders Nilsson VALLA, B-Huset, Ingång 27, Rum 3B.512,  
 +4613282635, [anders.p.nilsson@liu.se](mailto:anders.p.nilsson@liu.se)

**Kontaktperson hos kund:** Anders Nilsson VALLA, B-Huset, Ingång 27, Rum 3B.512,  
 +4613282635, [anders.p.nilsson@liu.se](mailto:anders.p.nilsson@liu.se)

**Kursansvarig**: Anders Nilsson VALLA, B-Huset, Ingång 27, Rum 3B.512,  
 +4613282635, [anders.p.nilsson@liu.se](mailto:anders.p.nilsson@liu.se)

**Handledare:** Theodor Lindberg, VALLA, B-huset, Ingång 25,  
 Rum 3B:551A, [theodor.lindberg@liu.se](mailto:theodor.lindberg@liu.se)

**Innehåll**

[**1 Inledning 1**](#_heading=h.tyjcwt)

[**2 Produkten 1**](#_heading=h.17vucsfvd1jw)

[**3 Systemet 2**](#_heading=h.91322tx0p5yn)

[**4 Modulerna 3**](#_heading=h.st5obfqvvsmz)

[4.1 Sensormodul 3](#_heading=h.lescigkpb370)

[4.2 Styrmodul 4](#_heading=h.6n07op3lfku9)

[4.3 Huvudmodul 5](#_heading=h.98a5kct31nfo)

[4.4 Kommunikationsmodul 6](#_heading=h.9b6ofhn5g580)

[4.5 Användarmodul (extern dator) 7](#_heading=h.q4hmj6xocfs2)

[**5 Teori 9**](#_heading=h.q80ek5ypqunr)

[5.1 Algoritmer 9](#_heading=h.ghaxfcpfhpjy)

[5.1.1 Sökning 9](#_heading=h.8qm7gcdm3lu5)

[5.1.2 Pathfinding 9](#_heading=h.h737nvgjgqkx)

[5.1.2.1 Breadth-first-search (BFS) 9](#_heading=h.5zfgfyo7qmgy)

[5.1.2.2 A-star search med heuristik (A\*) 9](#_heading=h.vd6ub23wrf91)

[5.2 Regleralgoritmer 11](#_heading=h.i1zc39vd8y55)

[5.2.1 IR-reglering 11](#_heading=h.acwmn7h6zq7q)

[5.2.2 Gyro-reglering 11](#_heading=h.3rrulfrnchky)

[**6 Slutsatser 12**](#_heading=h.ekwhhzfd5qvy)

[**7 Referenser 12**](#_heading=h.rw1h5s3pk74q)

[**A. Kopplingsschema Raspberry Pi 13**](#_heading=h.z337ya)

[**B. Kopplingsschema AVR (Sensormodul) 13**](#_heading=h.w6ll2vm5yoh9)

[**C. Kopplingsschema AVR (Styrmodul) 14**](#_heading=h.jfut70556sx1)

[**D. Pseudokod sökningsalgoritm 15**](#_heading=h.337y1erodact)

[**E. Pseudokod BFS 16**](#_heading=h.3pfv2z3mm0f3)

[**F. Pseudokod A\* 17**](#_heading=h.9vqh7zc1puk1)

**Dokumenthistorik**

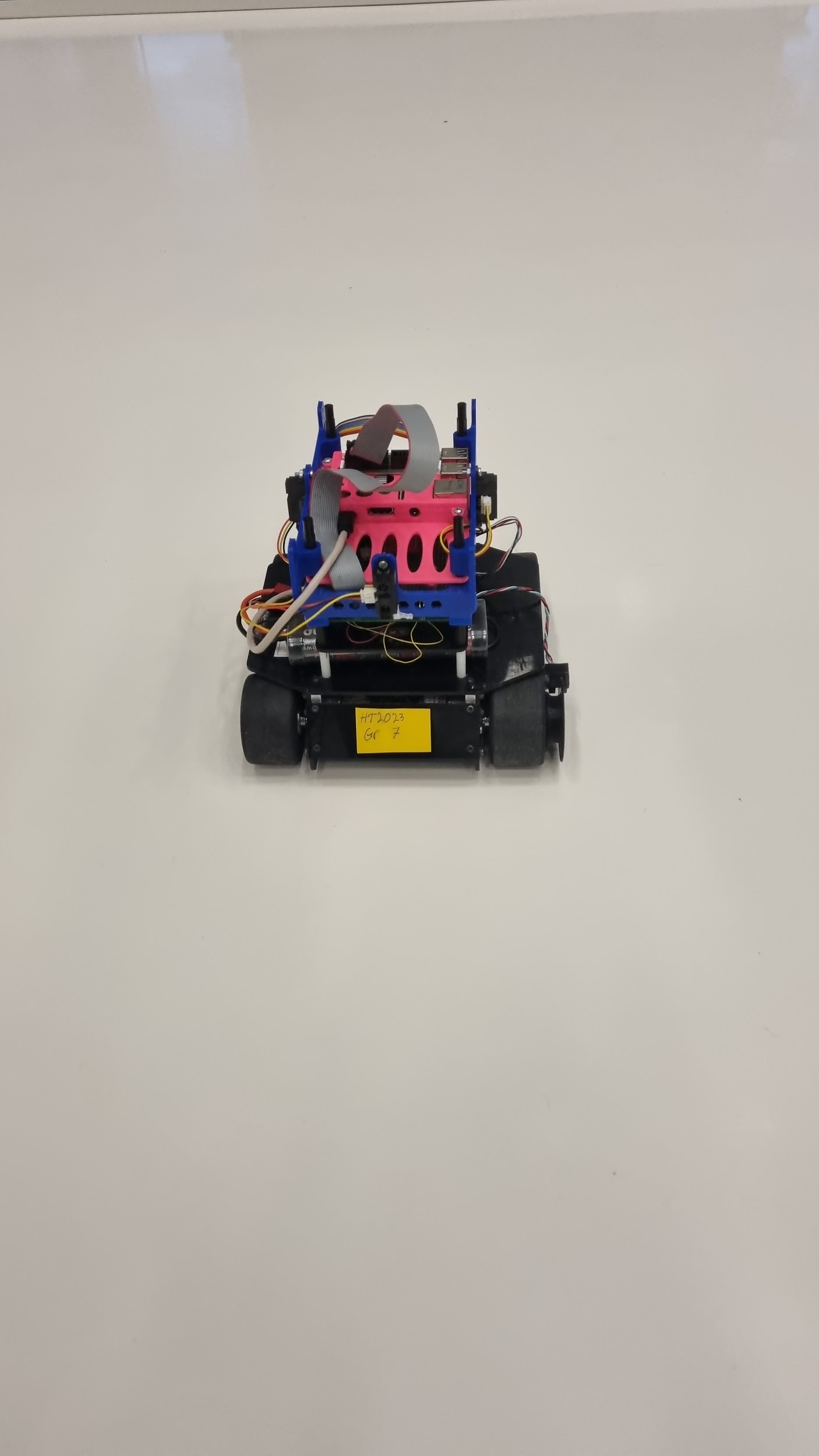
| **Version** | **Datum** | **Utförda förändringar** | **Utförda av** | **Granskad** |
| --- | --- | --- | --- | --- |
| 0.1 | 2023-12-06 | Första utkastet | G07 | FR |
| 0.2 | 2023-12-08 | Reviderat enligt återmatning | G07 | FR |
| 1.0 | 2023-12-12 | Första versionen, reviderat enligt återmatning | G07 | AN, FR |

# Inledning

Detta är en teknisk dokumentation för Kartrobot för projektet i Konstruktion med mikrodatorer (TSEA29). Syftet är att kunna användas som konstruktionsunderlag vid återskapande och dokumentation för underhåll och felsökning av både hård- och mjukvara.

# Produkten

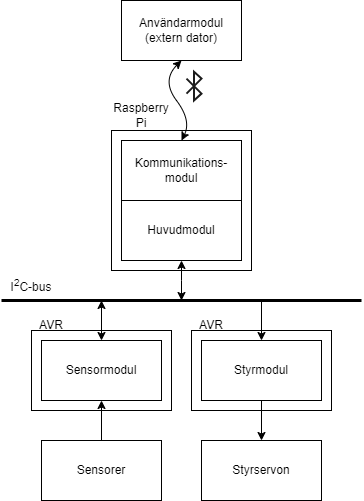
Kartroboten används för att kartlägga en rutnätbaserad bana, detta görs genom att söka av banan och samla in data som sedan behandlas för att besluta om nästa steg. Efter hela banan är kartlagd återvänder roboten till startpositionen. Det går att använda roboten med ett användargränssnitt på en extern dator, då data skickas genom bluetooth och visualiseras eller helt själv oberoende av extern dator.

**

1. Kartroboten

# Systemet

Nedan följer en översiktlig beskrivning av hela systemet.



1. Översiktligt blockschema av hela systemet.

Hela systemet är modulbyggt enligt Figur 2 ovan, uppdelat på fyra separata komponenter, en RPi (Raspberry Pi 3 Model B), två stycken AVR (ATmega 1284p) samt en extern dator.

Under RPi:n ligger både huvudmodulen och kommunikationsmodulen som båda delar utrymme. Detta gör att kommunikationen mellan dessa är väldigt enkel, speciellt eftersom modulerna båda är skrivna i Python. Det är också användarmodulen, vilket gör att bluetooth-kommunikationen mellan RPi och extern dator kan göras enkelt med redan färdiga Python-bibliotek.

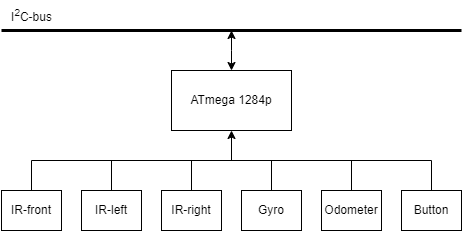
Både sensormodulen och styrmodulen ligger på varsin AVR, vilket passar väl för kommunikation med primitiva komponenter som sensorer och styrservon. Dessa två moduler är sedan kopplad till hvudmodulen genom Inter-Integrated Circuit (I2C) där RPi agerar master och AVR slavar. För detta används på RPi-sidan ett bibliotek *SMbus*[1] där speciellt två funktioner används, *write\_byte* som tar in en sju bitars slavadress och en byte data som ska skickas, och *read\_byte* som också tar in en sju bitars slavadress och returnerar data från en slav. Hur dessa används är beskrivet nedan.

# Modulerna

Nedan följer en mer detaljerad beskrivning av varje modul.

## Sensormodul

Nedan följer en mer detaljerad beskrivning av sensormodulen.



1. Översiktligt blockschema av sensormodulen.

Sensormodulen är byggd på en AVR (ATmega 1284p) och är kopplad till ett antal sensorer (3st IR-avståndssensor, 1st odometer, 1st gyroskop, 1st tryckknapp). Arbetsgången är baserat på en huvudlopp och ett antal avbrott. När microcontrollern startas initieras ett antal variabler och register och sedan startar huvudloopen. Därefter hanteras sensor efter sensor och senaste värde sparas i en *Struct*.

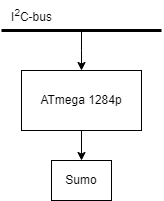
Vissa sensorer (IR-avståndssensor och gyroskop) är analoga och behöver AD-omvandlas. När en analog sensor ska läsas av väljs först vilken port som ska omvandlas och sedan startas omvandlingen. Därefter kommer ett avbrott när omvandlingen är klar och värdet sparas. AD-omvandlarens inspänning delas med referensspänningen (i vårt fall 5V) för att få ett procentuellt värde och därefter multipliceras värdet med 1024 för att få ett tio-bitarsvärde digitalt, detta visas i formeln nedan.

Resterande sensorer (odometer och tryckknapp) kan ses som digitala och är direkt kopplade till avbrott. Exempelvis för odometern finns en global variabel som agerar räknare för hur långt roboten åkt. När roboten kör och odometer blir hög triggas avbrottet och räknaren räknar upp ett steg. Det fungerar liknande för tryckknappen, bara att det istället finns två globala variabler som skiftar mellan noll och ett. Detta sker alltså hela tiden och när det är någon av dessas tur i huvudloopen tas endast värdet från den globala variabeln och stoppas in i den *Struct* som tidigare nämnts.

Kommunikationen med huvudmodulen sker som tidigare nämnt över I2C (även känt som TWI på en AVR). Över I2C skickas enbart en byte åt gången vilket innebär att all data inte kan skickas på en gång. Istället för att hålla koll på vilken data som ska till bussen skickar huvudmodulen först en byte med ett nummer som representerar vilken data som ska skickas. Därefter ber huvudmodulen om data och då vet sensormodulen vad som önskas. Detta ger även möjlighet för effektivisering, om man ska ha en typ av data ofta räcker det med att skicka vilken datatyp en gång och sedan kan huvudmodulen skicka flera read för att få samma typ av data. Den AVR som sensormodulen är belägen på har slavadress *0x24*.

## Styrmodul

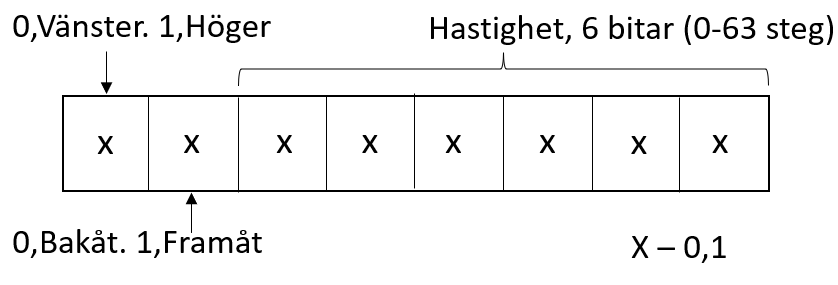
Nedan följer en mer detaljerad beskrivning av styrmodulen.



1. Översiktligt blockschema av styrmodulen.

Styrmodulen är byggd på en AVR (ATmega 1284P) och är kopplad till styrservon (Sumo) som innehåller robotens motorer. Styrmodulen kommunicerar med huvudmodulen via en I2C buss. Styrmodulen styr robotens motorer utifrån 8-bitars signaler från huvudmodulen. En 16-bitars Pulse Width Modulation (PWM) timer används för att hantera hastigheten på hjulen. En puls med längre *‘hög*’ period skickad till Sumo-modulen ger en snabbare rotationshastighet på hjulen. AVR och Sumo är ihopkopplade med fyra portar där ström och data kan skickas för att styra motorerna.

Styrmodulen arbetar baserat på en huvudloop samt två avbrottsrutiner. En för I2C (TWI) och en för PWM-timern. Den inskickade signalen hanteras och därefter modifieras pulsbredden så att korrekt data skickas vidare till sumo från styrmodulen via de fyra portarna. Insignalen till styrmodulen är uppbyggd enligt figur 5. Den mest signifikanta biten styr om motor på vänster eller höger sida ska styras, biten därefter styr riktningen på valda motor, framåt eller bakåt och resterande sex bitarna styr hastigheten. Den AVR som styrmodulen är belägen på har slavadress *0x74*.

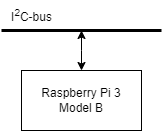


1. Representation av insignal till styrmodulen.

## 

## Huvudmodul

Nedan följer en mer detaljerad beskrivning av huvudmodulen.



1. Översiktligt blockschema av huvudmodulen.

Huvudmodulen är själva hjärnan som kontrollerar roboten och kör huvudloopen på RPi. När huvudmodulen startar initieras ett antal variabler och den I2C-bus som används. Ifall bluetooth skall användas startas en server och väntar på att en användare ska ansluta (roboten går att köra helt utan användarmodulen). Därefter startar huvudloopen som börjar med att läsa av sensorerna, det för att från början veta om roboten ska köra på automatiskt läge, något som kommer från tryckknappen i sensormodulen.

Efter sensordatan hämtas och om bluetooth ska användas går vi vidare till att hantera data som skickas till kommunikationsmodulen. Utöver sensordatan skickas det även med ifall roboten håller på med kartläggning för tillfället, och om robotens kartläggning är pausad. Om något fel skulle uppstå och en exception kastas kommer detta att hanteras genom att stänga servern och starta om den igen för att inte behöva starta om hela programmet.

Därefter hanteras autonomt körande, ifall automatiskt läge är inställt i sensormodulen (svarta tryckknappen) kontrolleras ifall kartläggning är aktiv och ifall kartläggningen inte är pausat. Detta styrs från användargränssnitt eller den röda tryckknappen beroende på om bluetooth används. Om så är fallet startas cykeln som hanteras autonomt körande.

I autopilot-cykeln kontrolleras om det finns några instruktioner i instruktionskön, om så är fallet genomförs instruktionen och returnerar till huvudloopen. Detta för att externa datorn ska ha den senaste informationen och autopiloten den senaste datan från IR-avståndssensorer. Instruktionsuppsättningen består av *drive forward* och *turn north/west/south/east*, som vardera ger indata till funktionerna *drive\_forward*, *turn\_left* och *turn\_right*. Dessa funktioner behandlar sensordata och skickar data till styrmodulen. Funktionen *drive\_forward* tar in odometer-data för att stanna efter rätt avstånd samt IR-avståndsdata och gyro-data för reglering medan *turn\_left* och *turn\_right* enbart tar in gyro-data för att stanna efter rätt vinkelavstånd.

För att stanna efter rätt vinkelavstånd konstrueras en Riemannsumma där *y* ges av läst gyro-data och *Δx* av tid sedan senaste avläsning, den totala summan är vinkelavståndet. Hur datan ser ut som skickas till styrmodulen beskrivs nedan men huvudmodulen har en funktion *set\_movement* som tar in en riktning, en hastighet och översätter till bytes som skickas via I2C.

Om det inte finns några instruktioner kvar i kön skannas de intilliggande rutorna för att spara öppna rutor som inte redan har sökts av. Dessa läggs till i den stack som beskrivs i [Sökning](#_heading=h.8qm7gcdm3lu5), och sätter den nuvarande rutan som besökt. När grannar skannas uppdateras ett rutnät som representerar banan till *empty* eller *wall*, för detta används en *Enum*.

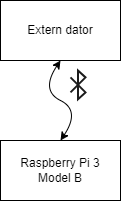
Därefter kontrolleras om stacken är tom, det vill säga allt är avsökt. Då får roboten instruktioner om att åka tillbaka till startpositionen med algoritmer beskrivet i [Pathfinding](#_heading=h.h737nvgjgqkx). Det kan vara så att hela startpositionen inte har sökts av (då det saknas IR-avståndssensor bak) så innan roboten vänder till ursprungsriktning söks startrutan av. Om det inte är något mer som ska sökas av svänger roboten till ursprungsriktning och uppdaterar att den för tillfället inte kartlägger.

Om stacken inte är tom går cykeln vidare till en while-loop för att hantera stacken. Det första elementet poppas och kontrolleras ifall den verkligen behöver besökas. Om en ruta är *empty* (vilket den måste vara för att vara med i stacken) och alla rutor omkring är kända behöver den inte sökas av, utan kan sättas som besökt på direkten. Om rutan behöver besökas läggs bästa väg enligt algoritm till i instruktionskön.

Under hela tiden skickas även data till en datakö i kommunikationsmodulen, det gäller rutor som bestämts, robotens position och tänkta vägar för roboten att åka. Detta för att användaren ska få förståelse över hur roboten tänker.

## Kommunikationsmodul

Nedan följer en mer detaljerad beskrivning av kommunikationsmodulen.



1. Översiktligt blockschema av kommunikationsmodulen.

För att extern dator och robot ska kunna kommunicera med varandra används bluetooth. Servern för bluetooth är belägen på den RPi som delas med huvudmodulen och klienten sedan via extern dator ansluta sig till med hjälp av ett Python-bibliotek, *socket*[2]. När servern hittat en enhet som söker efter rätt MAC-adress parkopplas enheterna.

När knappen *“Start mapping”* trycks ned på extern dator skickas meddelandet *“send data”* till kommunikationsmodulen. Kommunikationsmodulen tar emot data från sensormodulen om vad dess sensorer har för värde, men också från algoritmen i huvudmodulen som räknar ut vart roboten befinner sig samt vart det finns eller inte finns väggar. Datan från huvudmodulen sparas i en kö för att data som kommer när användarmodulen inte är redo inte ska förloras. När kommunikationsmodulen då får meddelandet “send data” via bluetooth, skickar den en sträng bestående först av sensormodulens nuvarande sensordata följt av all utritningsdata som ligger i kön. Exempel på sträng till användarmodulen: *"auto\_drive: 1 ir\_front: 20 ir\_left 32 ir\_right: 45 odometer: 200 gyro: 345 is\_mapping: 1is\_paused: 0 r 1 2 w 1 0 e 0 1 u 1 2 p 1 1”*. Här kan vi se att förutom sensordata i början av strängen har vi också, *auto\_drive* som säger om roboten kan styras manuellt eller inte, *is\_mapping* som säger om roboten kör sökalgoritmen och *is\_paused* som säger ifall algoritmen är pausad eller inte. Efter dessa data samt sensordata kommer utritningsdatan som består av tre karaktärer styck med mellanslag mellan varje karaktär och data. Första karaktären säger vad som ska ritas ut följt av en x-koordinat, därefter y-koordinat.

Typerna som kan ritas ut är, *‘r’* för robot, *‘w*’ för vägg, *‘e’* för tom, *‘u’* okänd och *‘p’* för sökväg. I exempel-strängen kan vi till exempel se att roboten ska ritas ut på koordinat (1,2).

Vid användarstyring skickar extern dator instruktioner om vilken riktning roboten ska förflytta sig till kommunikationsmodulen. Kommunikationsmodulen skickar vidare instruktionerna på I2C-bussen till styrmodulen.

Sist så sköter även kommunikationsmodulen annan data från använderamodulen som om när kartläggning ska startas, stoppas, pausas, osv. Denna data skickas vidare till huvudmodulen för processering.

## Användarmodul (extern dator)

Nedan följer en mer detaljerad beskrivning av användarmodulen.

Användarmodulen består ett användargränssnitt gjort i QT Creator’s open-source python-bibliotek, *PySide6*[3]. Användargränssnittet är uppbyggt på följande sätt. I toppen finns en QApplication klassen som hanterar GUI’s huvudinställningar. För att sedan få upp ett fönster för visualisering använder vi QMainWindow, detta objekt kan sedan fyllas med olika QWidgets som kan vara allt från en grafisk rityta till en knapp.

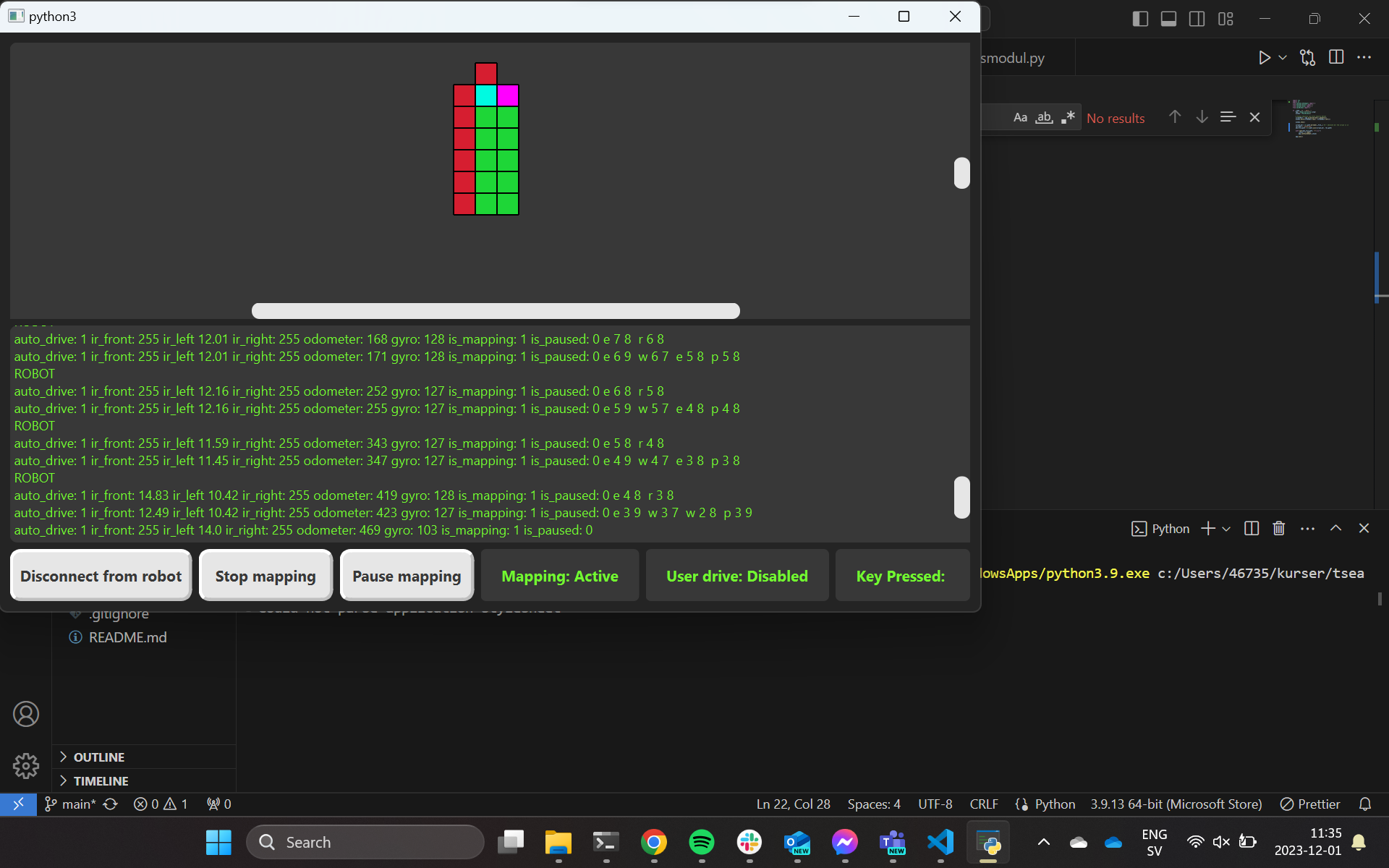
Längst upp i fönstret finns en QGraphicsView som renderar det som finns i en QGraphicsScene. Viewen har en mindre storlek än scenen vilket gör att scenen kan flyttas med hjälp av två “scroll bar’s” eller bara scrolla normalt. I scenen kan olika kvadrater ritas i olika färger beroende på vad som ritar upp. Detta görs utifrån datan som skickas från kommunikationsmodulen.

Under området för kartuppritningen finns en konsol där all data skrivs ut som tas emot från kommunikationsmodulen, här skrivs också resultat av knapptryck ut för att hjälpa felsökning.

Under detta finns tre knappar. Första knappen är för att ansluta och avbryta kopplingen till och från kommunikationsmodulen. Andra knappen används för att efter anslutning kunna starta eller stoppa sökalgoritmen, detta genom att skicka *“start mapping”* eller *“stop mapping”* via bluetooth till kommunikationsmodulen. Sista knappen är till för att pausa eller återuppta sökningen, detta på samma sätt som när kartläggning ska startas men istället skickas *“pause mapping”* eller *“unpause mapping”*. Knapparna uppdateras vid tryck men enbart efter bekräftelse från roboten.

Användarmodulen ritar upp området roboten åker i när den kör autonomt. När roboten börjar köra startas en timer. Den används för att varje halv sekund läsa datan som kommit in från kommunikationsmodulen. För varje avsökt koordinat ritas det upp rött om det är en vägg, grönt om det är en tom yta. En turkos ruta representerar robotens nuvarande position och rosa rutor representerar den tänkta vägen roboten ska ta.

Vid användarstyrning finns ett *KeyEvent* som konstant väntar på nedtryckningar från någon av de tangenter som styr roboten. Trycks de ned detekteras det vilken och skickas över till kommunikationsmodulen.



1. Användargränssnittet. Röd ruta representerar vägg, grön öppen yta, blå robotens position och rosa vilken väg den är på väg att köra.

# Teori

Nedan följer beskrivningar av de olika algoritmer som används i produkten.

## Algoritmer

### Sökning

För att söka av banan och bestämma vart roboten ska ta sig i nästa steg används en typ av depth-first-search. När roboten kommer till en ny position söks omgivningen av genom IR-avståndssensorerna och nyupptäckta rutor läggs till i en stack, där rutan framför roboten läggs in sist. Sedan tas beslut om vart roboten ska näst i stackens ordning, det vill säga man börjar med de senaste tillagda. Det kan hända att roboten upptäcker en ruta som redan finns med i stacken, då skrivs den gamla över med den nya.

Anledningen till varför en DFS används för sökning är delvis för att det är lätt att implementera men också för att det passar väl för uppgiften. Att använda exempelvis en breadth-first-search-algoritm för avsökningen hade gjort att roboten hade behövt åka fram och tillbaka hela tiden vilket inte hade varit effektivt.

Det finns ett specialfall då roboten startar i slutet av en korridor då IR-avståndssensorer fram, vänster och höger alla läser av väggar. Då kommer roboten att svänga 180° och fortsätta därifrån.

### Pathfinding

För att bestämma vilken väg roboten ska ta för kända rutor används två algoritmer beskrivet nedan.

#### Breadth-first-search (BFS)

BFS är en algoritm som fungerar väldigt likt DFS som beskrivits ovan, med skillnaden att en kö används istället för en stack. I detta fallet är det mer effektivt att använda BFS eftersom vägen redan är känd och roboten behöver inte röra sig då den hittar den bästa vägen eftersom endast kända rutor betraktas. BFS används alltså inte för kartläggningen utan för att hitta bästa väg mellan kända rutor.

Efter att en väg hittats, returneras den som en lista av positioner i rutnätet. Detta översätts sedan till instruktioner över hur roboten ska åka.

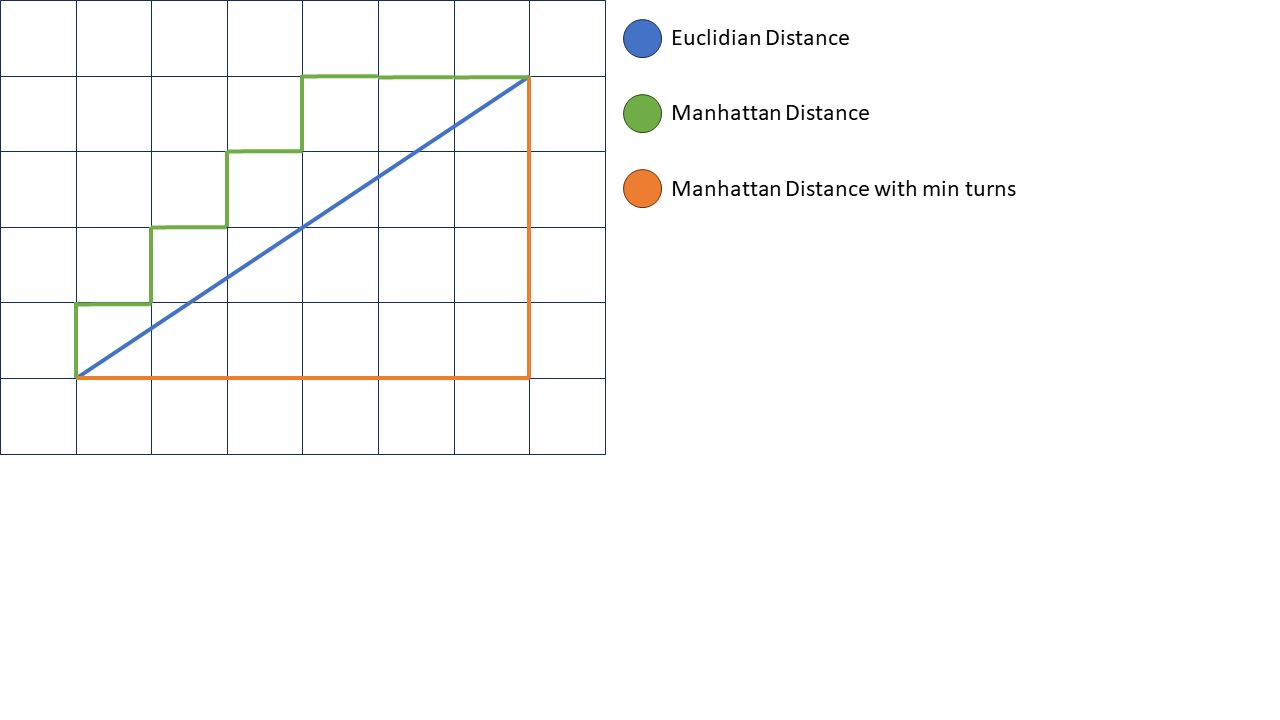
Roboten är gjord på ett så generellt sätt som möjligt, alltså även om roboten enbart ska röra sig en ruta används en sökalgoritm för att hitta bästa vägen.

#### A-star search med heuristik (A\*)

Eftersom vägen översätts till instruktioner betyder det inte att den kortaste vägen är den snabbaste, se figuren nedan som visualiserar Manhattan Distance. Det har att göra med att det tar tid för roboten att svänga. Därav används en A-star search med heuristik för att hitta den kortaste vägen med så få svängar som möjligt. A\* fungerar likt BFS med en prioritetskö där vägar kostar extra om det finns svängar.

Däremot eftersom detta är en heuristik kan det ibland bli lite fel och därav användes både BFS och A\* när snabbaste vägen bestäms genom att räkna Manhattan Distance och ta hänsyn till svängar.

Att ta en väg som har så få svängar som möjligt har ytterligare fördelar, att roboten ska svänga exakt 90° hela tiden går inte att lita på och färre svängar betyder mindre felmarginal.



1. Visualisering av Manhattan Distance

## ees(k=sdfRegleralgoritmer

Kartroboten använder en PD-regulator i två olika lägen som beskrivs nedan. Hjulhastigheten på vardera sida ändras beroende på resultat från reglering för att roboten ska svänga och minska felet. Anledningen till behovet av reglering är att hantera fel i exempelvis svängar och att styrservot ej kan röra rakt.

### IR-reglering

När någon utav kartrobotens sidomonterade IR-avståndssensorer har en vägg inom räckhåll är det första regleringsläget aktivt. Kartroboten använder då värdet från en IR-avståndssensor subtraherat med en konstant lika med det önskade avståndet från väggen som insignal för att centrera roboten på aktuell ruta.

Där är utsignalens styrka, *K* är proportionalitetskonstanten, *e* är insignalen, och är samplingstiden. *k* används som index för iterationer.

### Gyro-reglering

Då kartrobotens IR-avståndssensorer inte har väggar inom räckhåll är det andra regleringsläget aktivt. Kartroboten använder då vinkelhastigheten hämtad från gyro-sensorn subtraherat med önskat värde som insignal. Felet skalas sedan med en konstant för att kunna stoppas in i samma formel som vid första läget.

Där är en konstant som justerar felkonstanten från gyro-sensorn för att felutslaget ska vara av samma amplitud som för IR-avståndssensorerna och är det justerade felet. Därefter används samma formel som för första läget.

# Slutsatser

Det finns två huvudkategorier av områden där saker kan gå fel. Det är i mjukvaran där implementationer kan vara felaktiga eller dåligt gjorda och hårdvaran som kan vara trasig eller inte bete sig som tänkt.

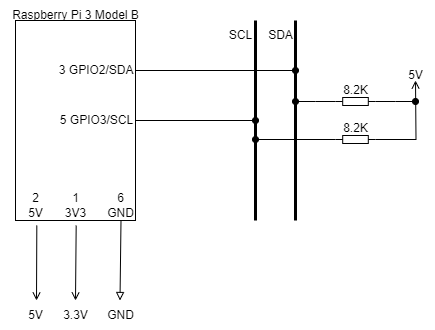
De förbättringsområden som finns gällande mjukvara handlar för oss om att algoritmer skulle kunna göras bättre eller helt enkelt bytas ut. Exempelvis är roboten för tillfället blind för framtiden och kan inte tänka i flera steg. Skulle den göra det skulle kartläggningen kunna göras mer effektivt i färre steg.

För hårdvaran däremot handlar förbättringsområden om att göra roboten mer felsäker och bättre hantera problem med sensorer och styrning. För tillfället är roboten inte helt konsekvent vad det gäller att svänga exakt 90° eller köra i 40 cm. Vi har redan hanterat detta genom att titta på konsekvenserna och exempelvis reglera efter väggar och stanna roboten då den håller på att åka in i en vägg. Däremot skulle vi vilja se över orsaken istället och lösa problemet innan det blir en konsekvens.

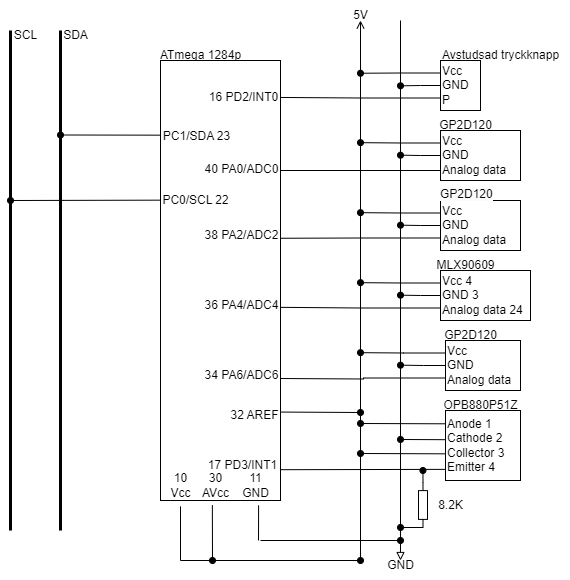
# Referenser

1. SMBUS2. PyPI. (n.d.). https://pypi.org/project/smbus2/
2. Socket - low-level networking interface. Python documentation. (n.d.). https://docs.python.org/3/library/socket.html
3. Pyside6. PyPI. (n.d.-a). https://pypi.org/project/PySide6/
4. Wikimedia Foundation. (2023, November 23). Depth-first search. Wikipedia. https://en.wikipedia.org/wiki/Depth-first\_search
5. Wikimedia Foundation. (2023, October 24). Breadth-first search. Wikipedia. https://en.wikipedia.org/wiki/Breadth-first\_search

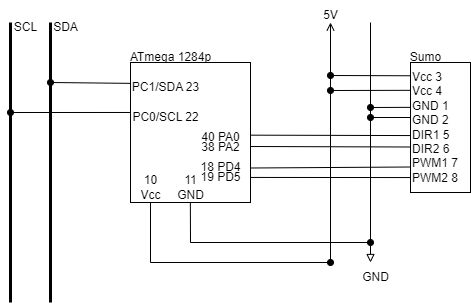
# Kopplingsschema Raspberry Pi



# Kopplingsschema AVR (Sensormodul)



# Kopplingsschema AVR (Styrmodul)



# Pseudokod sökningsalgoritm

Variabler *stack*, *pos*, *instr\_queue* och *visited* är globala.

function cycle\_autopilot(ir\_data)

if not instr\_queue.empty()

return

open\_neightbours = scan\_neighbours(ir\_data)

for n in open\_neighbours:

if n not in visited

stack.append(n)

visited.add(pos)

if not len(stack):

if pos != START\_POS:

path = find\_path(START\_POS)

make\_instructions\_from\_path(path=

elif heading != NORTH

instr\_queue.put(ROTATE\_NORTH)

else:

mapping = False

return

while len(stack):

nextpos = stack.pop()

if nextpos not in visited:

if is\_visit\_needed(nextpos):

path = find\_path(nextpos)

make\_instructions\_from\_path(path)

if nextpos in stack:

stack.remove(nextpos)

break

else:

visited.add(nextpos)

# Pseudokod BFS

function BFS(grid, start, end):

queue = queue.Queue()

queue.put([start])

visited = set([start])

width = len(grid[0])

height = len(grid)

while not queue.empty():

path = queue.get()

x,y = path[-1]

if (x,y) == end:

return path

for adjacent\_x, adjacent\_y in [(x+1,y), (x-1,y), \

(x,y+1), (x,y-1)]:

if 0 <= adjacent\_x < width and 0<= adjacent\_y < height \

and grid[adjacent\_y][adjacent\_x] == EMPTY \

and (adjacent\_x, adjacent\_y) not in visited:

queue.put(path + [(adjacent\_x, adjacent\_y)])

visited.add((adjacent\_x, adjacent\_y))

# Pseudokod A\*

function a\_star(grid, start, end):

prio\_queue[(0, start, heading, [])]

visited = set()

width = len(grid[0])

height = len(grid)

while len(prio\_queue):

total\_cost, current\_node, current\_dir, path = prio\_queue.pop()

if current\_node == end:

return path + [current\_node]

if current\_node in visited:

continue

visited.add(current\_node)

x, y = current\_node

for adjacent\_x, adjacent\_y in [(x + 1, y), (x - 1, y), \

(x, y + 1), (x, y - 1)]:

if (0 <= adjacent\_x < width \

and 0 <= adjacent\_y < height \

and grid[adjacent\_y][adjacent\_x] == EMPTY

and (adjacent\_x, adjacent\_y) not in visited):

new\_dir = get\_compass\_dir\_from\_value(

y - adjacent\_y, x - adjacent\_x)

new\_path = path + [current\_node]

cost = len(new\_path) + heuristic((adjacent\_x,\ adjacent\_y), end, currect\_direction, new\_direction)

prio\_queue.push(cost, (adjacent\_x, adjacent\_y),\ new\_dir, new\_path)