CHALMERS

Lidar rörelsedistorsionsfiltrering

Punktmolnsfiltreringssytem för bilar i realtid

Examensarbete inom högskoleingenjörsprogrammet Elektroteknik

SIMON KRANTZ

PONTUS ÖSTBERG

INSTITUTIONEN FÖR ELEKTROTEKNIK

CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2021
www.chalmers.se

Examensarbete inom högskoleingenjörsprogrammet Elektroteknik

Lidar rörelsedistorsionsfiltrering

Punktmolnsfiltreringssytem för bilar i realtid

Simon Krantz

Pontus Östberg

Institutionen för Elektroteknik
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2021

Lidar rörelsedistorsionsfiltrering
Punktmolnsfiltreringssytem för bilar i realtid

Simon Krantz
Pontus Östberg

© *Simon Krantz, 2021*
© *Pontus Östberg, 2021*

Institutionen för Elektroteknik
Chalmers Tekniska Högskola
SE-412 96 Göteborg Sverige
Telefon + 46 (0)31-772 1000

Förstasida: Bilen som har Lidarn och IMU:n monterad på sig, vilket också algoritmen testades med. Bilden är publicerad med tillstånd från Volvo Cars.
Göteborg, Sverige 2021

Förord

Detta projekt har utförts som ett examensarbete värderat 15 HP, för institutionen för elektroteknik på Chalmers Tekniska Högskola. Examensarbetet är utfört åt Volvo Cars under vårterminen år 2021.

Vi vill först och främst tacka vår handledare Arvid Pearson på Volvo Cars för att ha gett oss möjligheten till att utföra examensarbetet. Vi vill även tacka honom för hjälpen med samtliga delar under examensarbetets gång.

Vi vill även tacka vår examinator och handledare Bertil Thomas på Chalmers Tekniska Högskola för hjälp med examensarbetets rapport.

Sedan skulle vi vilja tacka Erik Frick och Oscar Johansson på AstaZero för hjälp med avlyssningen på IMU. Vi tackar Torbjörn Persson på Provinn för hjälp med IMU:n, samt Paris Austin på OxTS för hjälp med båda sensorerna och själva algoritmen.

--Simon Krantz, Pontus Östberg, Göteborg, Maj 2021

Sammanfattning

En av utmaningarna inom bilindustrin när det gäller aktiv säkerhet och automatisering är bilens perception av omgivningen. Det finns många olika metoder för att tolka omgivningen, varav en metod är att montera en ”*Light Detection and Ranging*”, Lidar, på bilen. Lidarn tolkar omgivningen genom att sprida ut laserpulser kring bilen och sedan ta emot pulsernas reflektion. För varje reflekterad puls kan Lidarn, bland annat, utgöra det reflekterande objektets position i det 3-dimensionella rummet, som en punkt med X, Y och Z koordinater.

Att ha Lidarn monterad på en bil medför dock att Lidarn utsätts för samma störningar som bilen gör, i form av accelerationer och rotationer. Störningarna leder till att Lidarns tolkning av omvärlden blir förvrängd.

Volvo Cars föreslog en metod där Lidarns störningar mäts med en ”*Inertial Measurements Unit*”, IMU, monterad i bilen. Genom det kan en algoritm konstrueras som filtrerar bort störningarna i Lidarns tolkning med IMU mätningar, vilket leder till att förvrängningen elimineras.

Detta projekt utförs tillsammans med Volvo Cars på deras provbana i Hällered. Projektet innefattar design av filtreringsalgoritmen och implementering av algoritmen i Python 3 kod. Algoritmen och implementeringen är konstruerad med avseende på realtid, vilket innebär för denna implementering att förvarningen ska filtreras innan Lidarns nästa perception är färdigt.

Den resulterande algoritmen uppfyller samtliga delmål och krav, men inte syftet i sin helhet. Algoritmen klarar av att filtrera punktmoln men filtrerar punktmolnen fel ibland, där ingen säker orsak till den felaktiga filtreringen har fastställts. Möjliga felkällor och vidareutveckling diskuteras. Utökade användningsområden och hållbarhet diskuteras även.

Slutsatsen av projektet är att det är möjligt att realtidsfiltrera bort rörelsedistorsioner från Lidar punktmoln, men algoritmen behövs förbättras vidare.

Sökord: Lidar, IMU, Punktmoln, Rörelsedistorsion och Filtrering.

Abstract

One of the challenges in the car industry when it comes to active security and automatization is the cars perception of its surroundings. There are several different methods that allows perception of the cars surrounding, one of them is to attach a Lidar on the car. The Lidar achieves perception of its surroundings via scattering laser pulses around the car and retrieving the reflection of the pulses. The reflected pulses allow the Lidar to determine properties of the pulse, including the position of the reflecting object as a point in the 3-dimensionall space with X, Y and Z coordinates.

By having the Lidar mounted on a car causes the Lidar to be subjected to the same disturbances that the car is. These disturbances come in the form of accelerations and rotations, usually referred to as motion distortion. The motion distortion causes the Lidars perception to become skewered and wrongly perceive the world.

Volvo Cars suggested a method where the motion distortions of the car are measured with an IMU mounted in the car. With that an algorithm can be constructed which filters out the distortion of the Lidar perception via the use of the IMU measurements.

This project is done together with Volvo Cars at their proving ground in Hällered. The project included designing a filtering algorithm and then implementing the algorithm as Python 3 code. The algorithm and implementation were constructed with the intent of being used in real-time, which means that the distortion of the Lidar perception is filtered out before the Lidar has finished the next perception.

The resulting algorithm fulfills every sub-goal and requirements but fails to fulfill the overarching goal of the project. The algorithm can sometimes filter point clouds correctly but sometimes it filters the point clouds incorrectly. There has been no cause determined for the wrongly behavior. Potential error sources and further development is discussed, as well has future uses and sustainability.

The conclusion of the project is that it’s possible to filter out motion distortion in Lidar point clouds, but further development is needed for the algorithm.

Search Words: Lidar, IMU, Point Cloud, Distortion removal, Filtering.

Innehållsförteckning

TERMINOLOGI/FÖRKORTNINGAR 1

1. INLEDNING 2
   1.1. Bakgrund 2
   1.2. Syfte 2
   1.3. Avgränsningar 2
   1.4. Precisering av frågeställningen 3

2. TEORI/TEKNISK BAKGRUND 4
   2.1. Lidar 4
   2.2. IMU 6
   2.3. Accelerometer 7
   2.4. Gyroskop 7
   2.5. Rörelseekvation 7
   2.6. Partiell derivata och jacobian 8
   2.7. Koordinattransformation 8
   2.8. Kalmanfilter 9
   2.9. Interpolering 10

3. METOD 11
   3.1. Hårdvara 11
   3.2. Design av algoritm 11
   3.3. Test och verifiering 11

4. ALGORITM 12
   4.1. Dataavlyssning och timing 12
   4.2. Ekvationer 14
   4.3. Rekursivitet och initiering 17
   4.4. Flödesschema 18

5. RESULTAT 20

6. SLUTSATS OCH DISKUSSION 22
   6.1. Slutsats 22
   6.2. Vidareutveckling och förbättringar 22
   6.3. Hållbarhet 23

REFERENSER 25

BILAGOR i
   Bilaga A: Jacobians av bilens stadie i
   Bilaga B: EKF brus och kovariansmatriser ii
   Bilaga C: Avlyssningsskript och algoritmhuvudkropp iii
   Bilaga D: Algoritm i kod v
   Bilaga E: EKF i kod vi
   Bilaga F: EKF modeller och jacobians i kod vii
   Bilaga G: EKF konstanter i kod viii
   Bilaga H: HTM i kod ix

TERMINOLOGI/FÖRKORTNINGAR

Lidar = *Light detection and ranging.*
Punktmoln = Samling av kartesiska koordinater i relation till Lidarn.
ETAVEP = *Enablers for Testing Autonomous Vehicles on Existing Proving Grounds*
POC = *Proof-Of-Concept*
IMU = *Inertial Measurements Unit.*
GPS = *Global Positioning System.*
TOF = *Time Of Flight.*
FOV = *Field Of View.*
HFOV = *Horizontal Field Of View.*
VFOV = *Vertical Field Of View.*
GNSS = *Global Navigation Satellite System.*
HTM = *Homogeneous Transformation Matrix.*
EKF = *Extended Kalmanfilter*

1. INLEDNING

Detta kapitel innefattar bakgrunden och syftet med projektet, samt vad projektets mål och avgränsningar är.

1.1. Bakgrund

När bilar ska övergå från att vara manuellt körda till att vara fullständigt självkörande, måste bilen kunna uppfatta världen runt omkring fordonet med stor säkerhet. För att realisera detta har en Lidar sensor monterats på fordonet blivit av stort intresse för biltillverkare. Lidarn bidrar med en snabb och stabil metod för bilens system att tolka sin omgivning genom att nyttja ljusemission från Lidarn. Lidarn kommer skicka ut ljus vilket sedan reflekteras och tas upp av Lidarn igen, och det reflekterade objektets koordinater kan framställas. Genom att skicka ut och ta upp ljus i en area kan samtliga reflekterande objekt koordinater, kallad punkter, sammanställas till ett punktmoln.

Med detta uppstår ett problem när bilen uppnår höga hastigheter eller när bilen blir utsatt för störningar, i form av rotationer och accelerationer av fordonet. Detta orsakar distorsioner i omgivningstolkningen av Lidarn, där bilen sedan tolkar sin omgivning fel och även riskerar att agera fel.

Volvo Cars och andra företag samarbetar i projektet ”*Enablers for Testing Autonomous Vehicles on Existing Proving Grounds*”, ETAVEP. Målet med ETAVEP är att ta fram en ”*Proof-Of-Concept*”, POC, på att automatisera testningen av fordon på företagens provbanor.

För att motverka problemet med rörelsedistorsionerna förslog Volvo Cars en lösning där en IMU sitter i bilen. IMU:n består av accelerometrar och gyroskopsensorer. Dessa sensorer mäter bilens hastigheter, accelerationer och rotationer i det 3-dimensionella rummet med tre axlar, X, Y och Z. Med IMU-data kan sedan rörelsedistorsionerna korrekteras så att bilens system tolkar sin omgivning rätt igen.

1.2. Syfte

Syftet med projektet är att skapa en algoritm programmerad i Python-språket. Algoritmen ska kunna korrektera rörelsedistorsionerna vilket uppstår när Lidarn är i rörelse och är utsatt för störningarna en bil upplever i bruk. Algoritmen ska även arbeta i realtid genom att kontinuerligt hämta data från Lidarn och IMU:n och sedan göra filtreringen innan nästa Lidar-datapaket hämtas.

1.3. Avgränsningar

Projektet kommer inte nyttja den inbyggda ”*Global Positioning System*”, GPS, i IMU:n för att mäta positioner, detta estimeras istället i algoritmen. Detta är dock något algoritmen kan tänkas vidareutvecklas framåt för att nyttja. Projektet kommer ej innefatta en sammansättning av en karta av de filtrerade punktmolnen. För test och verifiering av algoritmen kommer då individuella punktmoln jämföras före och efter filtreringen, istället för att bedöma en karta bestående av flera punktmoln. Även detta är ett användningsområde för algoritmen längre fram, vilket möjliggör bättre test och verifiering av algoritmen.

På grund av hastighetsbegränsningar mellan IMU och Lidarns sensordata kommer viss distorsion kvarstå, detta beror på att varje Lidar punkt inte kan ha ett eget motsvarande IMU-

data. Dock ska den kvarstående distorsionen vara tillräckligt liten att den ej påverkar bil systemens uppfattning. Algoritmen kommer skrivas i Python 3 och kommer då eventuellt inte fungera i äldre versioner som Python 2.

1.4. Precisering av frågeställning

Delmål för projektet var följande:

- Konstruera två avlyssnings skript vilket hämtar data kontinuerligt från IMU respektive Lidar.
- Designa en algoritm vilket filtrerar punktmolns distorsionen med komponent data.
- Implementera algoritmen som skriven Python 3 kod.
- Test och verifiering av algoritmen.

Kraven för algoritmen är följande:

- Algoritmen ska vara programmerad i Python 3-språket.
- Algoritmen ska kontinuerligt hämta data från både Lidarn och IMU:n.
- Algoritmen ska arbeta i realtid, vilket innebär att algoritmen ska göra filtreringen innan nytt Lidar-data finns tillgängligt.
- Algoritmen ska testas och verifieras på Volvo Cars Hällered testbana.

2. TEORI/TEKNISK BAKGRUND

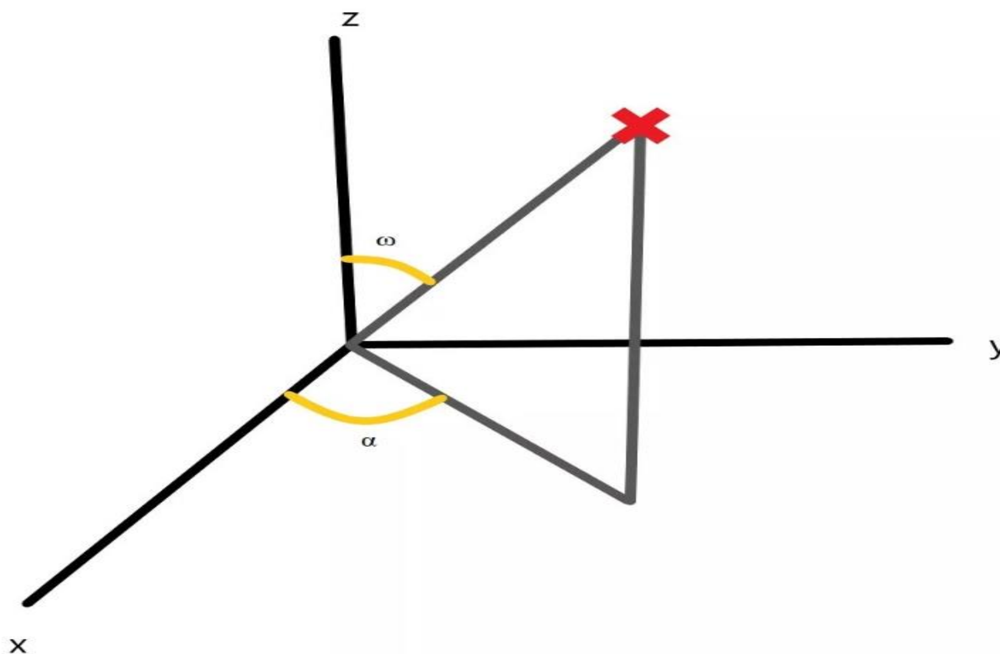
Följande kapitel innefattar viktig teori för algoritmen samt förklaring av använda komponenter.

2.1. Lidar

Lidarn är en aktiv sensor, vilket innebär att den sänder ut en form av signal och sedan detekterar reflektionen av den utsända signalen [1]. En Lidar sänder ut elektromagnetiska vågor, vanligen refererad som laser för Lidarn, av olika våglängder och intensiteter [1]. Genom att detektera den reflekterade signalen kan Lidarn detektera objekt framför sin lins [1]. Lidarn skickar ut laserpulser med korta intervaller, vanligen inom storleksordningen ett par nanosekunder beroende på modell [1]. Laserpulsen reflekteras sedan av ett objekt och återupptas av Lidarn [1]. När Lidarn tar emot den reflekterade laserpulsen, kan Lidarn från lasern utgöra spatiala data av det reflekterande objektet [1]. Den spatiala data som är relevant för projektet är de kartesiska koordinaterna, X, Y och Z, vilket det reflekterande objektet har i relation till Lidarn.

I det kartesiska koordinatrummet kommer Lidarn vara origo och det reflekterande objektet kommer ha en X, Y och Z koordinat vilket motsvarar objektets relation till origo [1]. Lidarn skapar en digital tolkning av sin 3-dimensionella omgivning genom att svepa en area med laserpulser [1]. Varje återupptagen laserpuls skickad i svepningen motsvarar en diskret punkt i koordinatsystemet, där sammanställningen av punkterna vilket resulteras av svepningen kallas ett punktmoln [1]. När en reflekterad laserpuls detekteras så tolkar Lidarn den i det sfäriska koordinatrummet och kalkylerar om till det kartesiska [2].

En sfärisk koordinat har följande information: d är distansen mellan Lidarn och det reflekterande objektet, α är longitudvinkeln och ω är polvinkeln [2]. Longitudvinkeln är rotationen i XY-planet mellan Lidar och punkten, och polvinkeln är rotationen i ZY-planet [2]. Följande figur påvisar hur det sfäriska koordinatsystemet ser ut:



Figur 2.1: Sfäriska koordinatsystemet.

Distansen beräknas av Lidarn via formeln för rörelse, relationen mellan sträcka, hastighet och tid [2]. Hastigheten (c) är ljusets hastighet, och tiden är "*Time Of Flight*", TOF, vilket är tiden det tog för laserpulsen från att bli avsänd till att bli detekterad igen [1]. Följande ekvation påvisar hur Lidarn beräknar distansen, där TOF är dividerat med 2 då Lidarn antar att tiden till och tiden från det reflekterande objekt är samma [2]:

$$d = c * \frac{TOF}{2} \tag{2.1}$$

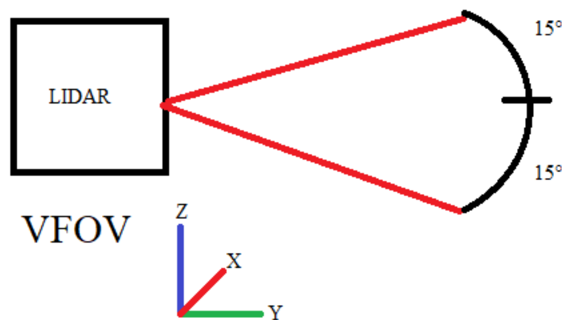
Genom att använda de sfäriska koordinaterna kan Lidarn konvertera dem till kartesiska koordinater enligt följande ekvation [2]:

$$x = d * cos(\omega) * sin\ (\alpha) \tag{2.2}$$

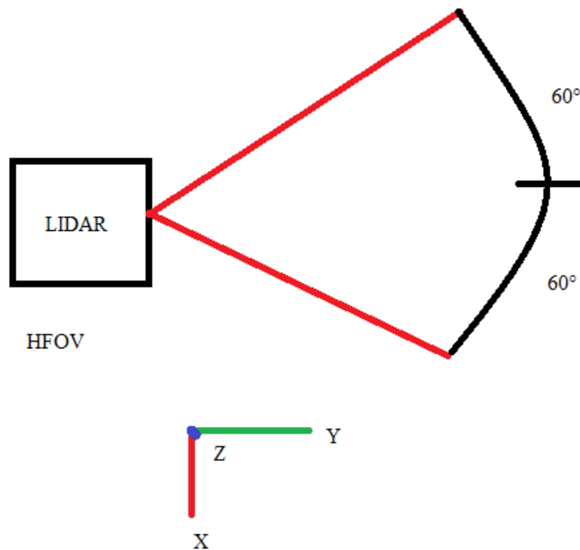
$$y = d * cos(\omega) * cos\ (\alpha) \tag{2.3}$$

$$z = d * sin\ (\omega) \tag{2.4}$$

När en kartesisk koordinatpunkt är beräknad kommer Lidarn vidare beräkna nästa punkt tills Lidarn har svept sin satta area med laserpulser och fått tillbaka motsvarande pulser [3]. En fullständig svepning refereras vanligen som en skanningscykel [3]. Skanningscykelens hastighet och area varierar på Lidar modell och konfigurering [3]. Lidar modellen använd i denna implementering är en Luminar Hydra [4]. Hydran har en inställbar skanningscykel och för denna implementering användes en skanningscykeltid på 0,1 sekunder, vilket motsvarar 10 Hz [4]. Lidarns skanningsarea refereras vanligen som "*Field Of View*", FOV [3]. Hydran i denna implementering har en FOV i form av en pyramid, vilket Lidarn skannar rakt framför sin lins [4]. FOV:en kan delas upp i Vertikal FOV, VFOV, och Horisontell FOV, HFOV [3]. Fig. 2.2 och 2.3 påvisar den använda Hydrans VFOV och HFOV, där VFOV är konfigurerbar men inställd enligt figur [4]:



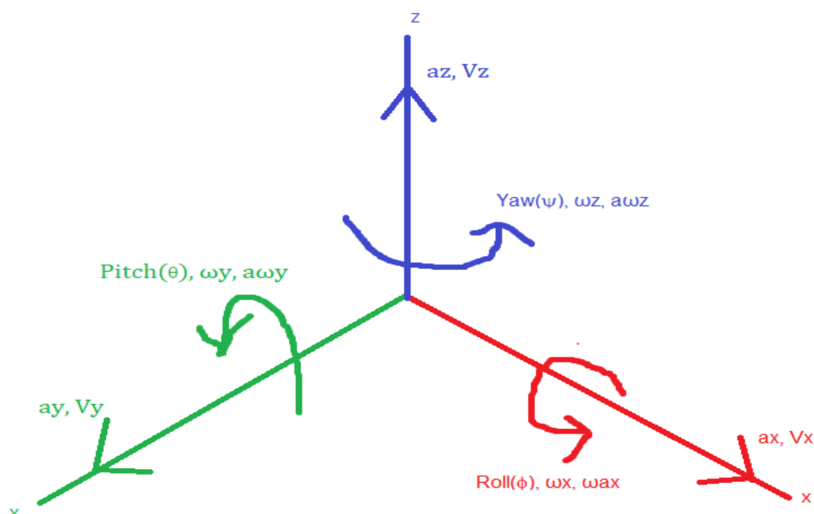
Figur 2.2: Illustration av den använda Lidar modellens VFOV.



Figur 2.3: Illustration av den använda Lidar modellens HFOV.

2.2. IMU

En IMU är en samling av tröghetssensorer som är konstruerade ihop till en komponent [5]. Dessa sensorer är accelerometer, gyroskop, magnetometer och ”*Global Navigation Satellite System*”, GNSS [6]. IMU:n använder dessa sensorer för att mäta olika enheter i det 3-dimensionella rummet, där trögheterna mäts kring XYZ-axlarna [6]. Varje individuell tröghetssensor mäter endast längs en av axlarna, därför består IMU:n av tre av samma tröghetssensorer, kallad triad, vilket möjliggör mätning för de tre axlarna [5]. Genom att kombinera en triad av accelerometrar och en triad av gyroskop uppnår IMU:n ett 6-axels system [5]. Detta system möjliggör att IMU:n kan ta två olika mätningar på vardera av de 3 kartesiska axlarna [5]. Följande figur illustrerar hur IMU:ns 6-axels system ser ut, där *yaw* (ψ) är rotationen runt Z-axeln, *roll* (ϕ) runt X-axeln och *pitch* (θ) runt Y-axeln:



Figur 2.4: IMU:ns 6-axels system.

Den använda IMU modellen i projektet är en OxTS 3003, där IMU:ns gyroskop- och accelerometermätningar är använda i algoritmen [6]. Mätningarna från IMU:n som är relevanta

till algoritmen är linjära- och vinkelaccelerationer och hastigheter, samt rotations vinklar. Modellen ger mätdata med 0,01 sekunders intervaller (100 Hz) [6].

2.3. Accelerometer

En accelerometer är en sensor vilket mäter störningarna på en kropp med massa [7]. Kroppen med massa använd i accelerometern består av ett piezoelektriskt material, vilket orsakar elektrisk laddning när den utsätts för krafter [7]. Störningar som mäts kan vara vibrationer eller accelerationer [7]. Accelerationerna orsakar en kraft på kroppen, vilket orsakar en proportionerlig elektrisk laddning [7]. Givet den proportionella relationen mellan elektrisk laddning samt acceleration, samt att kroppens massa är konstant, kan accelerationen beräknas av accelerometern [7].

2.4. Gyroskop

Ett gyroskop är en sensor som mäter orientering och vinkelhastighet av en kropp [8]. Gyroskopet mäter detta genom att mäta förflyttningen av en intern kropp i sensorn, där förflyttningen orsakas av rotationer [8]. När kroppen blir utsatt för förflyttningar kommer gyrot producera en motsvarande proportionerlig elektrisk signal [8]. Givet den proportionella relationen mellan elektrisk signal och rotation, kan orientering och vinkelhastighet beräknas av gyrot [8].

2.5. Rörelseekvationer

Ett objekt kan befinna sig i rörelse i det 3-dimensionella rummet, och rörelsen kan vara både linjär och i rotation [9]. Rörelsen består av hastigheter, accelerationer, tider och positioner både linjärt och i rotation [9]. Genom att mäta objektets rörelser kan man framställa ekvationer vilket beräknar kroppens utveckling av rörelse [9]. Följande två ekvationer är de primära rörelseekvationerna, hastighet och förflyttning [9]:

$$V_{n+1} = V_n + a * t \tag{2.5}$$

$$s = V_n * t + \frac{a*t^2}{2} \tag{2.6}$$

V är hastighet, där *n* indikerar det tidigare tidstegets hastighet och *n+1* indikerar nästa tidsstegs hastighet, *t* är tiden för rörelsen, *a* är accelerationen och *s* är förflyttningen [9]. Ekvationerna gäller både för linjär rörelse och för rotation, skillnaden i ekvationen blir då enheterna på variablerna [9]. Fig. 2.4 påvisar kroppens rörelse, vilket är den samma som IMU:ns 6-axelssytem. Med ekv. 2.5 och 2.6 och 6-axelssystemet i Fig. 2.4 kan, enligt [10], följande ekvationer för ändring av position och orientering i det 3-dimensionella rummet användas:

$$x_n = x_{n-1} + (V_x * t + \frac{(a_x*t^2)}{2}) * cos(\theta_{n-1}) * cos\ (\psi_{n-1}) \tag{2.7}$$

$$y_n = y_{n-1} + (V_x * t + \frac{(a_x*t^2)}{2}) * cos(\theta_{n-1}) * sin\ (\psi_{n-1}) \tag{2.8}$$

$$z_n = z_{n-1} - (V_x * t + \frac{(a_x*t^2)}{2}) * sin(\theta_{n-1}) \tag{2.9}$$

$$\phi_n = \phi_{n-1} + tan(\theta_n) * \left\{\left(\omega_y * t + \frac{a_{\omega y}*t^2}{2}\right) * sin\left(\phi_{n-1}\right) + \left(\omega_z * t + \frac{a_{\omega z}*t^2}{2}\right) * cos\ (\phi_{n-1})\right\} \tag{2.10}$$

$$\theta_{n-1} = \theta_{n-1} + \left(\omega_y * t + \frac{a_{\omega y} * t^2}{2}\right) * cos(\phi_{n-1}) - \left(\omega_z * t + \frac{a_{\omega z} * t^2}{2}\right) * sin(\phi_{n-1}) \quad (2.11)$$

$$\psi_n = \psi_{n-1} + \frac{1}{cos\,(\theta_{n-1})} * \left\{\left(\omega_y * t + \frac{a_{\omega y} * t^2}{2}\right) * sin(\phi_{n-1}) + \left(\omega_z * t + \frac{a_{\omega z} * t^2}{2}\right) * cos\,(\phi_{n-1})\right\} \quad (2.12)$$

2.6. Partiell derivata och jacobian

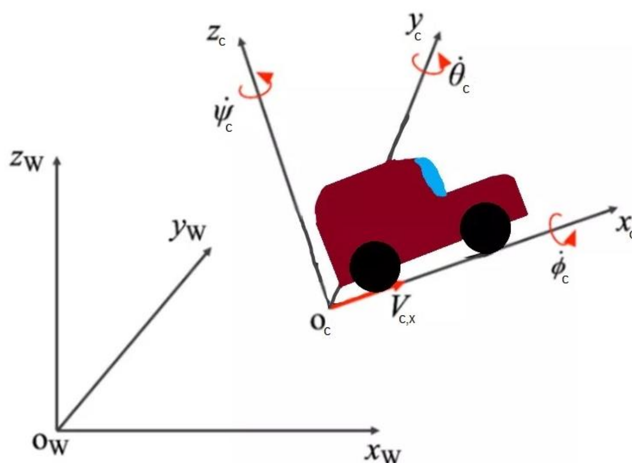
En funktion betecknas vanligen som *f(x)* om funktionen är beroende av en variabel, där då derivatan av funktionen betecknas som *df(x)/dx* [11]. Är funktionen beroende av flera oberoende variabler betecknas funktionen som, till exempel *f(x,y,z)* [11]. För att derivera en funktion vilket är beroende av flera variabler kan partiell derivering användas, vilket betecknas med till exempel *∂f(x,y,z)/∂x* [11]. Partiell derivering innebär att funktionen deriveras med avseende av en av variablerna medan resterande variabler anses som konstanter [11].

I fallet där en vektor, vars element består av olika flervariabels funktioner, deriveras av en annan vektor, vars element består av olika variabler vilket funktionerna är beroende av, uppstår en jacobian matris [12]. I jacobian matrisen är varje element en partiell derivata av ett element i en vektor med avseende av ett element i den andra vektorn [12]. Följande ekvation påvisar jacobian ekvationen, där $f=\{f_1(x_1, x_2...x_n),\ f_2(x_1, x_2...x_n)...\ f_n(x_1, x_2...x_n)\ \}$ och $x=\{x_1, x_2...x_n\}$ [12]:

$$J = \frac{\partial f}{\partial x} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_2}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_1} \\ \frac{\partial f_1}{\partial x_2} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_1}{\partial x_n} & \frac{\partial f_2}{\partial x_n} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix} \quad (2.13)$$

2.7. Koordinattransformation

Ett kartesiskt koordinatsystem byggs upp en av koordinatram vilket indikerar punkten av origo och positiv riktning för samtliga X, Y och Z-axlar [13]. En punkt i ett givet koordinatsystem kan transformeras och förflyttas till ett annat koordinatsystem, alltså en ny koordinatram [13]. Ett exempel på detta är att man transformerar från en lokal koordinatram i ett fordon (*b*) till en referensram vilket har en fast plats (*w*) [13]. Följande figur illustrerar detta exempel:



Figur 2.5: Lokal koordinatram och referensram.

Transformationen kan ske både i rotation och linjärt, vilket vanligen benämns ”*Homogeneous Transformation*” [13]. Transformationen beräknas genom att nyttja en ”*Homogeneous Transformation Matrix*” (HTM) mellan de två ramarna [13]. Följande ekvation påvisar HTM från *b*-ramen till *w*-ramen, där *R* är en 3x3 rotations matris och *d* är en 3x1 linjär förflyttnings matris med X, Y och Z värden [13]:

$$H_b^w = \begin{pmatrix} R_b^w & d_b^w \\ 0_{1X3} & 1 \end{pmatrix} \tag{2.14}$$

Följande ekvation påvisar hur en punkt (*p*), vilket består av X, Y och Z koordinater, transformeras från ett koordinatrum till ett annat [13]:

$$\begin{pmatrix} p_w \\ 1 \end{pmatrix} = H_b^w * \begin{pmatrix} p_b \\ 1 \end{pmatrix} \tag{2.15}$$

Enligt [13] så kan man beskriva de basala rotations matriserna för respektive X, Y och Z-axel med följande ekvationer:

$$R_{X,\phi} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos\,(\phi) & -sin\,(\phi) \\ 0 & sin\,(\phi) & cos\,(\phi) \end{pmatrix} \tag{2.16}$$

$$R_{y,\theta} = \begin{pmatrix} cos\,(\theta) & 0 & sin\,(\theta) \\ 0 & 1 & 0 \\ -sin\,(\theta) & 0 & cos\,(\theta) \end{pmatrix} \tag{2.17}$$

$$R_{z,\psi} = \begin{pmatrix} cos\,(\psi) & -sin\,(\psi) & 1 \\ sin\,(\psi) & cos\,(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.18}$$

Den totala basala rotations matrisen för samtliga axlar tas fram med följande ekvation [13], där *cos* och *sin* förkortas till *c* och *s* respektive:

$$R_{\psi\theta\phi} = R_{z,\psi} * R_{y,\theta} * R_{X,\phi} = \begin{pmatrix} c\theta c\psi & s\phi s\theta c\psi - c\phi s\psi & c\phi s\theta c\psi + s\phi s\psi \\ c\theta s\psi & s\phi s\theta s\psi + c\phi c\psi & c\phi s\theta s\psi - s\phi c\psi \\ -s\theta & s\phi c\theta & c\phi c\theta \end{pmatrix} \tag{2.19}$$

2.8. Kalmanfilter

Ett Kalmanfilter är en form av matematisk modell som används för att estimera okända linjära variabler, oftast kallad ett stadie, via kända mätningar [14]. Filtret består av två steg, prediktering och estimering [14]. Predikteringssteget innefattar att göra en prediktering på vad stadiet kommer vara i nästa tidssteg, med avseende på det tidigare tidstegets stadieestimering [14]. Estimeringssteget innefattar att göra en estimering av stadiet, givet predikteringen och de kända mätningarna [14]. Filtret kommer även beräkna felkovariansen i både prediktering och estimeringssteget, felkovariansen indikerar vad filtret beräknar att felet på estimeringen är [14]. En variant av det vanliga kalmanfiltret är *extended* kalmanfilter (EKF) vilket möjliggör att filtret också fungerar på icke linjära stadier [14].

Följande ekvationer utgör predikteringssteget för ett EKF [14]:

$$\widetilde{A_{k|k-1}} = f(\widehat{A_{k-1}}, u_{k-1}) \tag{2.20}$$

$$F_{k-1} = \frac{\partial f(\widehat{A_{k-1}}, u_{k-1})}{\partial \widehat{A_{k-1}}} \tag{2.21}$$

$$\widetilde{\Sigma_{k|k-1}} = F_{k-1} * \Sigma_{k-1} * (F_{k-1})^T + Q \tag{2.22}$$

I ekv. 2.20 och 2.21 så predikteras stadiet *A* och felkovariansen *Σ* i tidssteget *k* givet stadiet och felkovariansen i tidsteget *k-1* [14]. Prediktering noteras med ~ och estimering noteras med ^. I ekv. 2.20 sätts stadieprediktenigen till resultatet av den icke linjära funktionen *f ()* vilket är beräknad med avseende på tidigare stadie och kontroll inputen *u* [14]. Felkovariansen beräknas med tidigare felkovarians samt *F,* vilket är partiell derivatan av *f* med avseende på stadiet, och *Q*, vilket är en kovariansmatris av bruset för stadiet [14]. *F* beskriver hur stadiet ändrar sig givet att ingen kontroll input ges, och *Q* beskriver kovarians mellan alla variabler i stadiet [14].

Följande ekvationer utgör estimeringssteget för ett EKF [14]:

$$h(\widetilde{A_{k|k-1}}) = H * \widetilde{A_{k|k-1}} + v \tag{2.23}$$

$$\widetilde{y_k} = z_k - h(\widetilde{A_{k|k-1}}) \tag{2.24}$$

$$S_k = H * \widetilde{\Sigma_{k|k-1}} * H^T + R \tag{2.25}$$

$$K_k = \widetilde{\Sigma_{k|k-1}} * H^T * (S_k)^{-1} \tag{2.26}$$

$$\widehat{A_k} = \widetilde{A_{k|k-1}} + K_k * \widetilde{y_k} \tag{2.27}$$

$$\widehat{\Sigma_k} = (I - K_k * H) * \widetilde{\Sigma_{k|k-1}} \tag{2.28}$$

I ekv. 2.23 beräknas *h()* med det tidigare predikterade stadiet *A*, *H* vilket är mätningsmatrisen som hämtar predikterade sensormätningar ur stadie matrisen, samt *v* vilket är sensors brus [14]. Ekv. 2.24 predikterar differensen mellan den faktiska sensormätningen *z* och den predikterade sensormätningen *h()* [14]. *S,* vilket beräknas i ekv. 2.25, är mätnings prediktions kovarians, och beräknas med felkovariansen, mätningsmatrisen samt *R* vilket är kovariansen för *v* [14]. *K* är kalmanförstärkningen och indikerar hur mycket estimeringen ska bero på sensormätningen, kontra predikteringen [14]. Ekv. 2.27 och 2.28 beräknar estimeringen av stadiet och felkovariansen med tidigare uträknade värden [14]. Estimeringarna används sedan rekursivt i nästa iteration [14].

2.9. Interpolering

Interpolering är en matematisk metod vilket möjliggör att okända värden mellan två kända kan estimeras [15]. Den enklaste formen av interpolering är linjär interpolering, vilket estimerar en okänd punkt mellan två kända via följande ekvation [15]:

$$y = y_1 + \frac{(y_2 - y_1)}{(x_2 - x_1)} * (x - x_1) \tag{2.29}$$

I ekv. 2.29 så är den interpolerade punkten (x, y) och de två kända före och efter punkterna är (x_1, y_1) och (x_2, y_2) respektive [15].

3. METOD

Följande kapitel innefattar metoden som används under projektets gång. Projektet kan delas upp tre huvud delmoment: Hårdvara, design av algoritm samt test och verifiering.

3.1. Hårdvara

En del av projektet var att få igång använd hårdvara. Den använda hårdvaran var Lidar, IMU och beräkningsdator. Utöver att få igång hårdvaran, behövdes även två avlyssnings skripts kodas i Python 3, för att kontinuerligt inhämta data från Lidar och IMU. Dessa skripts skrevs i samarbete med Volvo Cars för Lidarn, och med AstaZero för IMU:n.

Arbetsgången för skripten bestod av att AstaZero och Volvo Cars bidrog bas-skript för respektive komponent, vilket sedan omskrevs för att passa de använda modellerna av hårdvaran och för att samverka med algoritmen.

3.2. Design av algoritm

Arbetsgången för designen bestod av tre delar, skrivbordsundersökning, design samt kodning i Python 3. Skrivbordsundersökningen bestod av att identifiera tidigare gjorda lösningar och om de kan användas för denna implementering. Utifrån det kunde tidigare lösningar kombineras ihop till en design för en algoritm som funkar för denna implementering. Efter en designad algoritm, i matematiska uttryck och teori, kodas den i Python 3-språket.

3.3. Test och verifiering

Algoritmen testades och verifierades i ett praktiskt scenario. Volvo Cars hade en bil, med nödvändig hårdvara implementerad i, vilket algoritmen kunde testas i. Utrustningen bestod primärt av IMU, Lidar samt beräkningsdator vilket behövdes för projektet. Bilen kördes på Volvo Cars Hällered testbana med algoritmen och avlyssningarna på hårdvaran aktivt. Genom att köra fordonet och testa algoritmen på provbanan kan miljön kontrolleras, då fordonet ej kan anses vara säkert i en vanlig bilmiljö. I en kontrollerad miljö på provbanan kan således en högre grad av säkerhet garanteras för samtliga inblandade. På testbanan utsattes fordonet för möjliga störningar som kan orsaka Lidarns distorsion, vilket var olika rotationer, accelerationer samt höga hastigheter.

Hårdvaran verifierades genom att spara mätningar under körning från både Lidar och IMU, vilket sedan kontrollerades att stämma med vad bilen utsatts för. För att verifiera algoritmens funktionalitet jämfördes punktmolns bilder, där tydliga rörelsedistorsioner uppstått, före och efter algoritmen. Beroende på det filtrerade punktmolnets förändring i förvrängning och skärpa, kan algoritmens funktionalitet verifieras.

4. ALGORITM

Följande kapitel innefattar filtreringsalgoritmens olika delar, vilket innefattar struktur, timing samt matematiska ekvationer. Algoritmen är modellerad efter [10] förslagna algoritm. Detta görs då resultatet på den förslagna algoritmen gav bra resultat, bra möjlighet för vidareutveckling samt att den är gjord i avseende på tidsoptimering och realtid.

Denna algoritm kommer avskilja sig från algoritmen i [10] i avseende att denna kommer expandera visa delar och förenkla andra, så att den förslagna algoritmen passar just denna implementering.

4.1. Dataavlyssning och timing

Lidarn har en skanningscykel på 0,1 sekunder. Denna tid anges som nt där n är antalet skanningscykel (n=0,1,2,3…). Det aktuella Lidar punktmolnet anges som P_{nt} och iterationen innan anges som $P_{(n-1)t}$. Punktmolnet P kommer bestå av punkter p enligt följande:

$$P = (p_1, p_2, p_3 \dots) \tag{4.1}$$

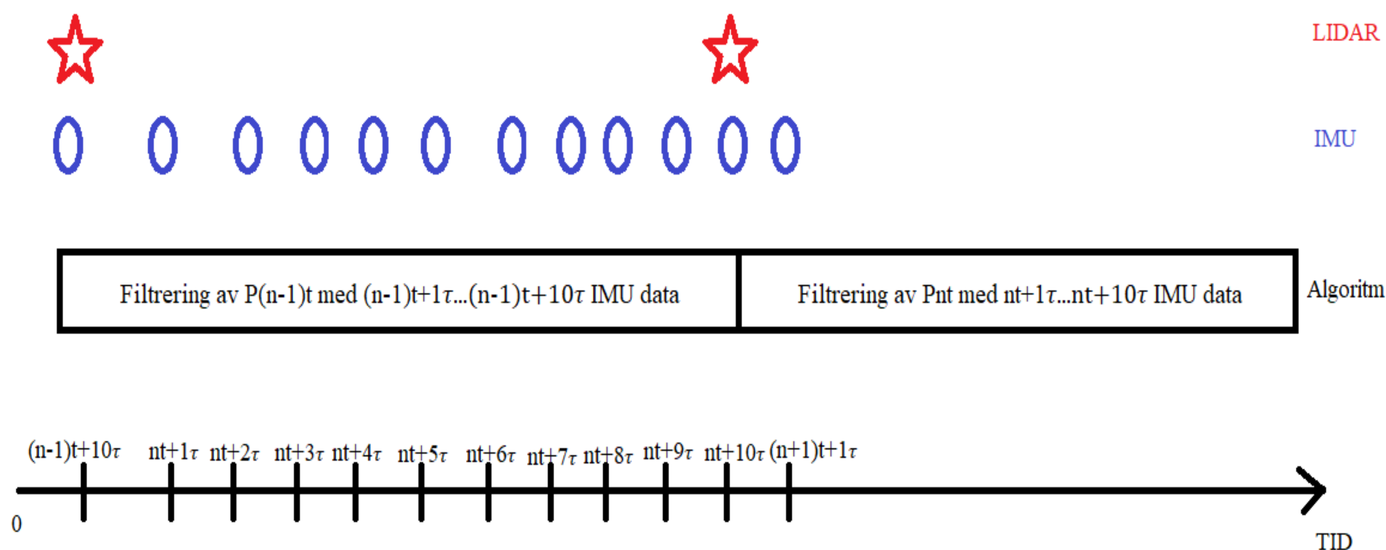
För varje skanningscykel av Lidarn kommer olika antal reflektionspunkter identifieras. Detta beror på hur många laserpulser som hinner återupptas för en period. Från mätningar med Lidarn så kan man utgöra att antalet punkter varierar mellan 55 000 och 57 000 punkter.

IMU:n har en dataperiod tid på 0,01 sekunder. Detta anges som $k\tau$. Under en Lidar skanning kommer IMU:n ideellt ge 10 data mätningar enligt följande ekvation:

$$\frac{t}{\tau} = \frac{0{,}1}{0{,}01} = 10 \tag{4.2}$$

Fullständig synkronisering mellan IMU och Lidar kan ej uppnås, då varken av komponenterna har exakt den angivna periodtiden. Periodtiderna för komponenterna kan komma att variera ±1ms för IMU och ±5ms för Lidarn, vilket leder att fullständig synkronisering inte kan antas. Variationen kan vara både positiv och negativ, vilket förhindrar komponenterna från att bli förskjutna. Påföljande av icke-idealiteten är att Lidarns punktmoln kan bli filtrerad med både 9 och 10 IMU mätningar.

Ekv. 4.2 leder till att $k \in [0, 9]$, men icke-idealiteten kan leda till $k \in [0, 8]$, som repeteras för varje skanningscykel. Följande figur illustrera datatimingen för sensorerna, samt algoritmens arbetstiming, i det ideella fallet:



Figur 4.1: Timing för sensordata och algoritm i ideellt fall. Ett skannat punktmoln tillgängligt från Lidar indikeras med röd stjärna, och tillgänglig IMU-mätning indikeras med blå cirkel.

Fordonets position och orientering estimeras via interpolering, vilket sedan används för filtrering av delar av punktmolnet. Detta görs för att estimera bilens position och orientering för tidpunkterna mellan två IMU skannings perioder. Periodtiden för interpoleringen är 0,000625s och betecknas $j\Delta\tau$. Periodtiden för interpoleringen kommer variera mellan implementeringar, då periodtiden bestäms via hur mycket exekveringstid algoritmen har tillgängligt i relation till Lidar periodtiden. På grund av detta kan interpoleringsperiodtiden variera beroende på hårdvaran som beräkningsdatorn har.

För denna implementering så har periodtiden $\Delta\tau$ bestäms via mätningar av exekveringstiden för algoritmen, där 0,000625s passade för denna beräkningsdator. Följande ekvation påvisar hur många interpoleringar som kan göras mellan två IMU data:

$$\frac{\tau}{\Delta\tau} = 16 \tag{4.3}$$

Ekv. 4.3 leder till att $j \in [0, 15]$, vilket upprepas för varje k iteration**.** Utifrån ekv. 4.2 och 4.3 kan algoritmens timing-ekvation sammanställas som följande:

$$nt + k\tau + j\Delta\tau \tag{4.4}$$

På grund av att sensorerna ej har fullständig synkroniserat kommer vissa punktmolnsiterationer sakna det sista tidssteget, när k är 8. För att motverka icke-idealiteten används följande ekvation, vilket nyttjar ekv. 4.1 till 4.4 för att dela upp vilka punkter i punktmolnet som har tagits för varje tidssteg:

$$\frac{L(P)}{17 * L(IMU)} * j + \frac{L(P)}{L(IMU)} * (k - 1) \tag{4.5}$$

I ekv. 4.5 så är *L(P)* antalet punkter i punktmolnet, och *L(IMU)* är antalet IMU mätningar för punktmolnet. Ekvationen resulterar i ett värde vilket agerar index till punkterna i punktmolnet. Punkterna med ett index mellan resultatet och det tidigare indexet kommer då behandlas för

detta tidssteg. I nästa tidssteg kommer sedan det tidigare indexet uppdateras till resultatet från ekv. 4.5 och ekvationen räknar ut slutindex igen. Detta upprepas tills slutindexet når *L(P)*.

IMU modellen OxTS 3003 ger möjlighet till att ta olika rörelsedata mätningar. Följande ekvation påvisar mätningarna som tas från IMU:n i denna implementering, där antalet prickar på indexet indikerar antalet derivator på variabeln, till exempel en prick på rotation indikerar rotationshastighet och två indikerar rotationsacceleration:

$$z_k = \left(\dot{V}_x, \ddot{\phi}, \ddot{\theta}, \ddot{\psi}, \phi, \theta, \psi, V_x, \dot{\phi}, \dot{\theta}, \dot{\psi}\right)^T \tag{4.6}$$

4.2. Ekvationer

Algoritmen är uppbyggd i följande block: datainsamling, kalmanfiltrering, interpolering och slutligen punktmolnsfiltrering. Datainsamlingsblocket innefattar att hämta punkter från Lidarn och rörelsedata från IMU:n. Kalmanfiltreringsblocket innefattar att filtrera mätningarna som togs av IMU:n. Behovet av blocket i detta avseende är för att varken den olinjära stadieprediktering, eller IMU mätningarna kan anses som det faktiskt stadiet. Anledningen till detta är predikteringen och mätningarna har brus och störningar. Genom att nyttja kalmanfiltret där stadieprediktering och IMU mätningarna slås ihop, kan man estimera ett värde närmre det sanna stadiet. Blocket används även för att estimera värden som används i stadiet och algoritmen, men inte mäts av IMU:n. I denna implementering är detta bilens X, Y och Z koordinater under en skanningscykel.

Interpoleringsblocket används för att vidare estimera bilens stadie under Lidars skanningscykel. Genom att estimera bilens stadie i mindre tidssteg än IMU:ns avläsningstid τ, vilket i denna implementering är tidssteget $\Delta\tau$, kan en högre precision av filtreringen uppnås. Detta då punktmolnets punkter kan delas upp i mindre grupper, där interpoleringsestimeringen bättre återspeglar bilens faktiska stadie under tidssteget. Interpoleringen används också för att motverka icke-idealiteten i synkroniseringen mellan IMU och Lidar. Blocket nyttjar ekvationen för linjär interpolering, trots att stadiet inte är linjärt. Ett antagande har gjorts att tidsteget för interpoleringen är så litet att ändringen av stadiet under den tiden kan anses vara linjärt.

Punktmolnsfiltreringensblocket nyttjar en HTM av bilens stadie, specifikt bilens linjära position och bilens orientering. Algoritmen behandlar alltid under tidssteget *nt* Lidar punktmolnet vilket skannades under *(n-1)t* tidssteget, därför är indexeringen *(n-1)* exkluderad ur ekvationerna i detta kapitel. Kapitel 4.3 behandlar flödesschemat för algoritmen där det framgår hur algoritmen nyttjar detta kapitels ekvationer, och hur algoritmen itereras.

Följande ekvationer påvisar stadiematrisen *A* vilket bilen har i denna implementering:

$$A = \left(x, y, z, \phi, \theta, \psi, V_x, \dot{\phi}, \dot{\theta}, \dot{\psi}\right)^T \tag{4.7}$$

I ekv. 4.6 påvisas vilka mätningar som tas från IMU:n, där orienteringsaccelerationen samt den linjär accelerationen i X-led är inkluderat. Matrisen *a* är accelerationsmatrisen som är en sammanställning av accelerationsmätningarna från IMU:n, vilket följande ekvation påvisar:

$$a = \left(\dot{V}_x, \ddot{\phi}, \ddot{\theta}, \ddot{\psi}\right)^T \tag{4.8}$$

Följande ekvationer är bilens förflyttningsekvationer som är baserad på ekv. 2.6, där förflyttningen beräknas under en IMU periodtid i linjära X-led, samt rotationshastigheten runt Y och Z-axeln:

$$b_1(a) = V_x * \tau + \frac{\dot{V_x} * \tau^2}{2} \tag{4.9}$$

$$b_2(a) = \ddot{\theta} * \tau + \frac{\ddot{\theta} * \tau^2}{2} \tag{4.10}$$

$$b_3(a) = \ddot{\psi} * \tau + \frac{\ddot{\psi} * \tau^2}{2} \tag{4.11}$$

Bilens kontrollinput *u* är bilens accelerationer samt IMU:ns periodtid, vilket följande ekvation påvisar:

$$u = \begin{pmatrix} a \\ \tau \end{pmatrix} \tag{4.12}$$

Med ekv. 4.7 till 4.12 kan modellen för bilens olinjära stadie, samt rörelseekvationerna, tas fram. Ekv 4.13 och 4.14 påvisar modellen och ekvationerna i matrisform respektive vektorform. I ekv.4.13 och 4.14 är *C*, *S* och *T* en förkortning för *Cos*, *Sin* och *Tan*. Rörelseekvationerna beräknar hur bilens stadie har ändrats i nästa tidssteg för IMU:n, givet IMU mätningar och bilens stadie i det tidigare IMU tidssteget.

$$f(A_k, u_k) = \begin{pmatrix} x_k + b_{1_k} * C\theta_k * C\psi_k \\ y_k + b_{1_k} * C\theta_k * S\psi_k \\ z_k - b_{1_k} * S\theta_k \\ \phi_k + T\theta_k\left(b_{2_k} * S\phi_k + b_{3_k} * C\phi_k\right) \\ \theta_k + b_{2_k} * C\phi_k - b_{3_k} * S\phi_k \\ \psi_k + \frac{1}{C\phi_k}\left(b_{2_k} * S\phi_k + b_{3_k} * C\phi_k\right) \\ V_{x_k} + \dot{V}_{x_k} * \tau \\ \dot{\phi}_k + \ddot{\phi}_k * \tau \\ \dot{\theta}_k + \ddot{\theta}_k * \tau \\ \dot{\psi}_k + \ddot{\psi}_k * \tau \end{pmatrix} = \begin{pmatrix} x_{k+1} \\ y_{k+1} \\ z_{k+1} \\ \phi_{k+1} \\ \theta_{k+1} \\ V_{x\,k+1} \\ \dot{\phi}_{k+1} \\ \dot{\theta}_{k+1} \\ \dot{\psi}_{k+1} \end{pmatrix} \tag{4.13}$$

$$A_{k+1} = f(A_k, u_k) \tag{4.14}$$

Med bilens olinjära stadie beskrivet så kan algoritmens EKF modelleras. Algoritmen beskriven i [10] har en förslagen EKF modell, vilket skiljer sig från det traditionella EKF modellen i kap 2.8. Skillnaden mellan den traditionella EKF modellen och den förslagna i [10] är ekvationen för den predikterade felkovariansen. Genom jämförelse av resultat mellan den traditionella ekvationen och den förslagna i [10], kunde det konstateras att den förslagna ekvationen gav bättre resultat. Anledningen till det bättre resultatet beror på att den förslagna ekvationen är anpassad utifrån den använda olinjära stadiemodellen, medan den traditionella fungerar för alla stadie modeller.

I en jämförelse av exekveringstid mellan [10] förslagna modell och den traditionella, kunde det konstateras att den traditionella hade bättre exekveringstid. I denna implementering valdes den förslagna modellen, vilket leder till att EKF modellen ger bättre estimeringar av de sanna värden men sämre exekveringstid.

Följande ekvation påvisar stadiepredikteringsekvationen, som predikterar bilens stadie givet bilens stadie och accelerationer i det tidigare tidssteget:

$$\widetilde{A_{k|k-1}} = f\left(\widetilde{A_{k-1}}, u_{k-1}\right), där\ u_{k-1} = \begin{pmatrix} a \\ \tau \end{pmatrix} \tag{4.15}$$

Felkovarianspredikteringen beräknas med stadiemodellen A:s jacobian matriser F och G, samt Q matrisen. F är partiell derivatan av stadiemodellen med avseende på stadiet och G är partiell derivatan av stadiemodellen med avseende på accelerationerna. Q matrisen är i denna implementering kovariansen av accelerationerna istället för kovarians av stadie bruset. Det är inkluderingen av G samt ändringen av vad Q är som urskiljer den förslagna modellen i [10].

I denna implementering så antas det att accelerationerna inte har kovarians, vilket leder till att Q matrisen endast blir en diagonal matris med variansen för varje acceleration. Värden utanför diagonalen i Q blir därför 0. Antagandet togs då det inte fanns tid att mäta kovariansen av mätningarna. Accelerationsvariansen bestämdes genom mätning av IMU i stillastående läge. Se Bilaga A för jacobian matriserna F och G, samt Bilaga B för Q matrisen. Följande ekvation påvisar felkovarians predikteringen:

$$\widetilde{\Sigma_{k|k-1}} = F_{k-1} * \Sigma_{k-1} * (F_{k-1})^T + G_{k-1} * Q * (G_{k-1})^T \tag{4.16}$$

I estimering steget av EKF så används sensorbrus matrisen v, kovarians matrisen av sensor bruset R samt mätningsmatrisen H_{IMU}. Likt Q matrisen antas kovariansen mellan sensormätningarna vara 0, vilket leder till att R blir en diagonalmatris med variansen för varje sensormätning. Sensor bruset och variansen mäts genom att ta mätningarna från IMU i ett stillastående läge. Se Bilaga B för v och R matriserna.

H_{IMU} matrisen används primärt för att hämta de predikterade sensormätningarna som är inkluderade i stadiematrisen. Detta leder till att matrisen har samma antal kolumner som rader i stadiematrisen och samma antal rader som predikterade sensormätningar. Följande ekvationer påvisar H_{IMU} matrisen:

$$H_{IMU} = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.17}$$

Följande ekvationer är de traditionella EKF ekvationerna tidigare beskrivet i kap 2.8, ekv.2.23 till 2.28, vilket används i algoritmen:

$$h(\widetilde{A_{k|k-1}}) = H_{IMU} * \widetilde{A_{k|k-1}} + v \tag{4.18}$$

$$\widetilde{y_k} = z_k - h(\widetilde{A_{k|k-1}}) \tag{4.19}$$

$$S_k = H_{IMU} * \widetilde{\Sigma_{k|k-1}} * H_{IMU}^T + R \tag{4.20}$$

$$K_k = \widetilde{\Sigma_{k|k-1}} * H_{IMU}^T * (S_k)^{-1} \tag{4.21}$$

$$\widehat{A_k} = \widetilde{A_{k|k-1}} + K_k * \widetilde{y_k} \tag{4.22}$$

$$\widehat{\Sigma_k} = (I - K_k * H_{IMU}) * \widetilde{\Sigma_{k|k-1}} \tag{4.23}$$

För interpoleringsblocket och filtreringsblocket så används endast bilens linjära position och orientering, vilket hämtas i algoritmen ur bilens estimerade stadie från ekv. 4.22. Följande ekvation påvisar vektorn *X* som består bilens linjära position och orientering:

$$X_k = (x, y, z, \phi, \theta, \psi) \in \widehat{A_k} \tag{4.24}$$

I interpoleringsblocket nyttjas *X* vektorn för nuvarande IMU tidssteg k och det tidigare k-1. Den linjära interpoleringsekvationen beskriven i ekv. 2.29 nyttjas för att estimera *X* vektorn mellan X_{k-1} och X_k. Interpoleringen används på en IMU periodtid τ och varje interpoleringspunkt görs med en interpoleringsperiodtid $\Delta\tau$. Följande ekvation påvisar interpoleringsekvation för vektorn *X*:

$$X_{k-1,j} = X_{k-1} + \frac{X_k - X_{k-1}}{\tau} * j * \Delta\tau \tag{4.25}$$

För varje interpoleringspunkt transformeras motsvarande del av punktmolnet, vilket beräknas med ekv. 4.5, från bilkoordinatramen till den fasta världskoordinatramen, illustrerad i Fig. 2.5. Punkterna transformeras med en HTM av den interpolerade *X* vektorn, vilket leder till punkterna filtreras. Filtreringen sker då HTM är baserad på den linjära förflyttningen och orienteringen som bilen blir utsatt för vid det tidssteget. Genom det justerar HTM punkterna till sin sanna plats om de ej var utsatta för förflyttningen och vridningen. Följande ekvation påvisar transformeringsekvationen:

$$\begin{pmatrix} p_{w,x,k-1,j} \\ p_{w,y,k-1,j} \\ p_{w,z,k-1,j} \\ 1 \end{pmatrix} = H_c^w(X_{k-1,j}) * \begin{pmatrix} p_{c,x,k-1,j} \\ p_{c,y,k-1,j} \\ p_{c,z,k-1,j} \\ 1 \end{pmatrix} \tag{4.26}$$

Slutligen transformeras hela punktmolnet med en invers HTM av *X* vektorn för den slutliga stadieestimeringen, vilket leder att hela punktmolnet är i bilens koordinatrum och i relation till bilen vid slutet av Lidarns skanningscykel. Följande ekvation påvisar transformeringen

$$\begin{pmatrix} {p_{c,x}}^* \\ {p_{c,y}}^* \\ {p_{c,z}}^* \\ 1 \end{pmatrix} = H_w^b(X_{10}) * \begin{pmatrix} p_{w,x} \\ p_{w,y} \\ p_{w,z} \\ 1 \end{pmatrix} = (\, H_b^w(X_{10})\,)^{-1} * \begin{pmatrix} p_{w,x} \\ p_{w,y} \\ p_{w,z} \\ 1 \end{pmatrix} \tag{4.27}$$

4.3. Initiering och rekursivitet

Vid användningen av kalmanfiltret är det viktigt hur det ska initieras för sin första iteration och hur estimeringen återanvänds. Kalmanfiltret använder sitt tidigare tidsstegs stadie- och felkovarians estimering för att göra nuvarande tidsstegs prediktering. I den första iterationen av filtret finns det ingen tidigare estimering, vilket måste initieras till ett fast värde. För denna implementering kan den första stadieestimeringen anses vara 0. För den första felkovariansen kan inget antagande göras, vilket leder till att felkovariansen bestäms genom att testa olika värden och jämföra resultatet. Följande två ekvationer påvisar initieringsmatriserna för EKF som gav bäst resultat:

$$\widehat{A_{init}} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{4.28}$$

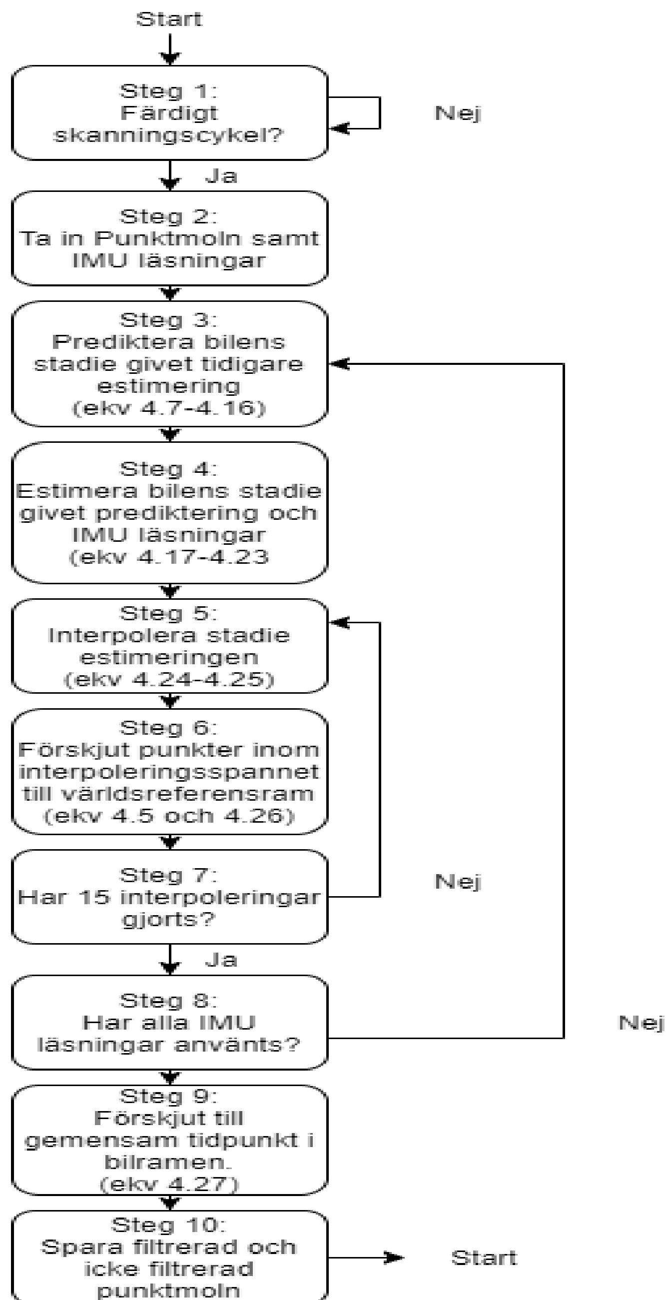
$$\widehat{\Sigma_{init}} = \begin{pmatrix} 0.1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.1 \end{pmatrix} \tag{4.29}$$

I denna implementering hanteras rekursiviteten av estimeringen annorlunda från det traditionella EKF. För varje IMU mätning inom en skanningscykel hanteras rekursiviteten enligt den traditionella metoden, vilket är att estimeringen i tidigare tidssteg används i predikteringen i nästa steg. Mellan olika skanningscykel, från n till $n+1$ återställs estimeringen till motsvarande initieringsvärden beskrivna i ekv. 4.28 och 4.29.

Detta görs så bilen omjusterar vart den fasta världsreferensramen är för varje skanningscykel, vilket leder till att varje punktmoln blir filtrerat lokalt för den iterationen. Påföljden av att återställa estimeringarna är att kalmanfiltret får sämre precision i estimeringen, men att punktmolnet inte påverkas av tidigare iterationens estimeringar.

4.4. Flödesschema

Algoritmen arbetar genom att dela upp punkterna som motsvarar varje IMU läsning och sedan vidare dela upp punkterna till motsvarande interpoleringsestimering. Algoritmen behöver iterera igenom varje IMU läsning, vilket blir 10 gånger. Vidare behöver algoritmen för varje IMU läsning iterera igenom varje interpoleringspunkt, vilket blir 15 gånger. Följande figur påvisar arbetsgången för algoritmen med motsvarande ekvationer från kap 4.2, samt iterationspunkter:



Figur 4.2: Algoritmens flödesschema.

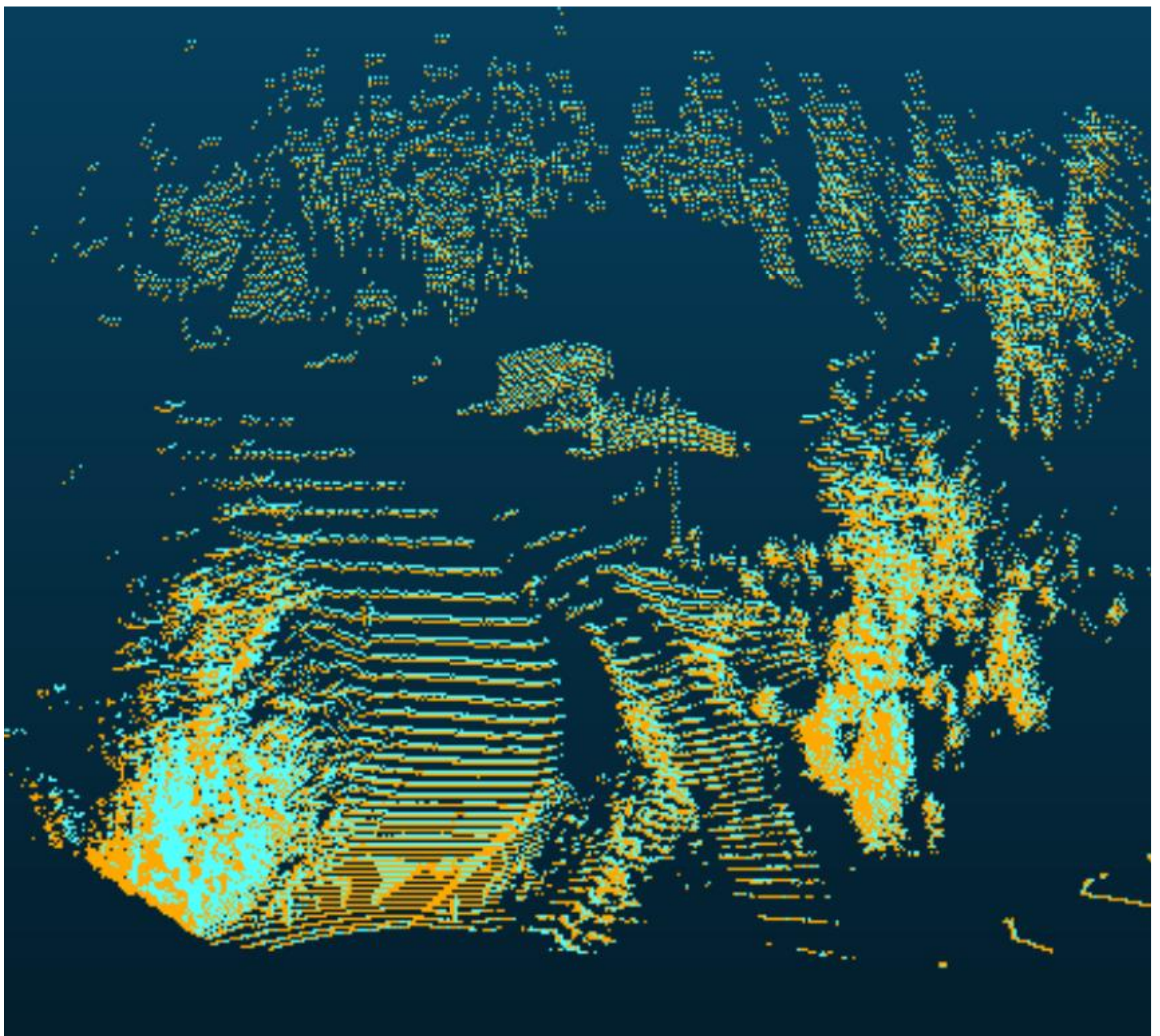
5. RESULTAT

Följande kapitel innefattar resultatet av den förslagna algoritmen.

Sensorfusionen mellan IMU och Lidar är uppfylld enligt kraven, och båda sensorer kan avlyssnas kontinuerligt och parallellt. Algoritmens krav på realtid uppfylls, då exekveringstiden varierar mellan 5-10ms. För att realtids kravet skulle uppfyllas behövde exekveringstiden vara under 100ms, vilket är Lidarns skanningscykeltid.

Algoritmens funktion testades genom att utsatta bilen för olika störningar på Volvo Cars Hällered testbana, där det icke filtrerade och det filtrerade punktmolnet sparades. Sedan togs stickprov för att jämförda de sparade punktmolnen för att verifiera algoritmens funktion.

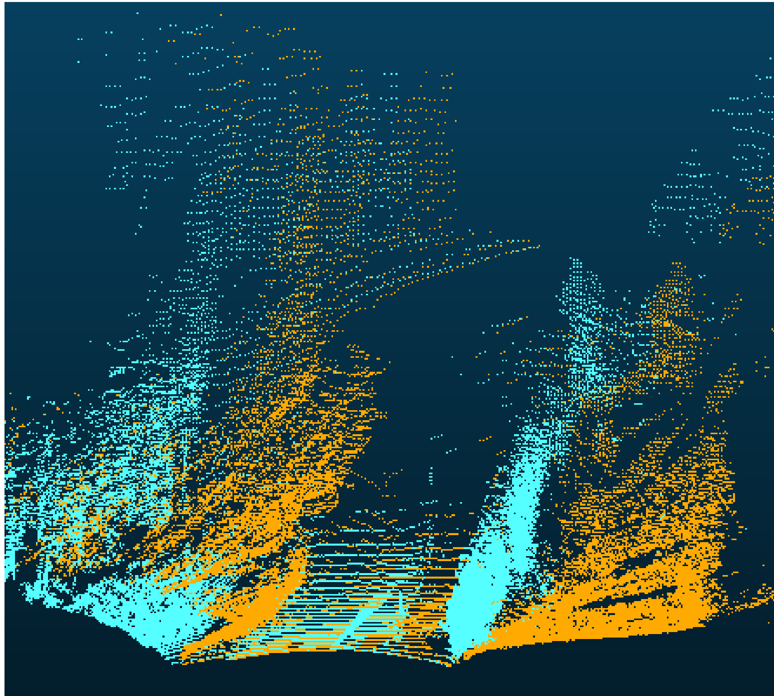
Följande figurer illustrerar punktmoln där algoritmen fungerat enligt tänkt teori:



Figur 5.1: Icke filtrerad (blå) och filtrerad(orange) punktmoln som överlappar. Det filtrerade punktmolnet korrigerar ner till marknivå samt till bilens perspektiv i slutet av sitt skanningscykel.

I fig. 5.1 syns det tydligt att fordonet svänger åt höger och kör framåt, men även att bilen har en pitchrotation i negativ riktning. Pitch-rotationen framgår då de filtrerade punkterna är förskjutna nedåt. Att bilen kör åt höger framgår då samtliga av de filtrerade punkterna förskjuts till vänster.

Följande figur illustrerar punktmoln där algoritmens resultat inte stämmer enligt teori:



Figur 5.2: Icke filtrerad (blå) och filtrerad(orange) punktmoln som överlappar. Det filtrerade punktmolnet korrigerar fel och skapar en onaturlig kurvatur i punktmolnet.

I fig. 5.2 framgår det att algoritmen inte fungerar helt fullständigt enligt teori. Det filtrerade punktmolnet korrigeras fel, till punkten att det inte går att avgöra bilens riktning under skanningscyklet. Det filtrerade punktmolnet får en exponentiell liknade form, vilket inte stämmer överens med den icke filtrerade punktmolnet.

Utifrån test och verifiering kan det konstateras att filtreringen kan resultera både som i fig. 5.1 där den fungerar, eller som fig. 5.2 där den inte fungerar. Båda figurer togs under samma körning, och är tagna vid likt väglag och störningar. Genom test och verifiering kunde inget fastställas som orsaken till problemet.

6. SLUTSATS OCH DISKUSSION

Följande kapitel innefattar dragna slutsatser från projektet, diskussioner om resultatet och möjliga förbättringar samt algoritmens påverkan från ett hållbarhetsperspektiv.

6.1. Slutsats

Syftet med projektet var att designa och implementera en algoritm i Python som filtrerar och korrigerar Lidar rörelsedistorsioner. Algoritmen ska kontinuerligt hämta data från Lidar och IMU, samt arbeta i realtid.

Projektets delmål och krav har uppfyllts. Algoritmen är designad och implementerad som Python 3 kod. Algoritmen avlyssnar kontinuerligt från IMU och Lidar, och exekveringstiden för filtreringen är snabbt nog för att arbeta i realtid.

Projektets syfte är dock inte fullständigt uppfyllt, då algoritmen inte alltid fungerar enligt teori. Samtliga delmål och krav är uppfyllda, men syftet i helhet är inte uppfyllt. Utifrån test och verifiering kunde ingen slutsats dras till orsaken, då det saknades tid, men kap. 6.2 diskuterar potentiella orsaker och hur lösningarna kan verkställas. Resultatet påvisar dock tydligt att det är möjligt att realtidsfiltrera rörelsedistorsioner i Lidar punktmoln, men algoritmen behöver utvecklas vidare

6.2. Vidareutveckling och förbättringar

Från resultatet kan det konstateras att algoritmen inte fungerar helt enligt teori. Vissa punktmoln filtreras enligt tänkt teori medan andra punktmoln får den felaktiga exponentiella formen. Genom att vissa punktmoln filtreras rätt medan andra filtreras fel kan man med stor sannolikhet säga att felet inte ligger i själva kodimplementeringen.

Dock kan inte felaktig kod försummas helt som potentiell felkälla. En potentiell felkälla med koden kan vara när något värde av bilens stadie, till exempel hastigheter, accelerationer eller rotationer, närmar sig 0 så uppstår oönskade beteenden. Då den misslyckade filtreringen har en exponentiell liknande form, kan ekvationer med division med värden nära 0 vara en möjlig orsak till detta beteende. I koden sker detta i bilens stadiemodell samt jacobians, vilket kan testas vidare, genom att lägga in en hantering eller blockering. En annan möjlig felkälla till att koden inte fungerar enligt teori är vid tillfällen där rotationsspill uppstår. Rotationsspill uppstår när en rotation överskrider sin maxgräns och hoppar till sin min gräns, till exempel att yaw går från 180° till -180°. Kodimplementeringen har skydd implementerat för rotationsspill situationer, vilket har testats och verifierats, men det går ej att eliminera som en möjlig felkälla på grund av felaktig verifiering och mänskliga felet.

Den mer troliga orsaken är att felet ligger i själva teorin för algoritmen. En potentiell felkälla med teorin kan vara att kalmanfiltret inte kan stabilt ge rimliga estimeringar av bilens faktiska stadie. Orsaken till detta kan eventuellt bero på både att kalmanfiltret är fel kalibrerat eller fel modellerat. Q, R och v matriserna är modellerade utifrån brus och variansmätningar av IMU:n, men kan även hanteras som kalibreringsparametrar för filtret. Genom att ändra värdet av parametrarna i matriserna kan EKF eventuellt ge bättre estimeringar vilket bättre återspeglar bilens faktiska stadie. Bilens stadiemodell och predikteringsekvation kan eventuellt vara fel modellerade och inte vara gynnsam för denna implementering. Genom att ändra stadiemodellen samt predikteringsekvationerna kan EKF ge rimliga estimeringar mer konsekvent.

Algoritmen är designad med avseende för vidareutveckling. Den ger bra möjligheter till att implementera in GNSS mätningar, vilket kan inkluderas i det använda EKF som sensormätningar för bilens X, Y och Z position. Även andra kända GNSS algoritms metoder kan inkluderas med viss kodomskrivning. Användningen av transformeringen till det fasta världskoordinatrummet ger goda möjligheter till konstruktion av en världskarta med de filtrerade punktmolnen. I denna implementering omjusteras världsreferensramen för varje punktmoln, vilket leder till att varje punktmoln har en lokal världsreferensram. För vidareutveckling med kartläggning kan omjusteringen av världsreferensramen tas bort, vilket leder till att varje punktmoln får en global gemensam referensram. Detta leder till att varje punktmoln filtreras i avseende på samma fasta punkt, vilket möjliggör kartkonstruktion.

Algoritmen i [10] ger ett förslag på punktmolnskartläggning, baserat på ”*NDT-scan matching”*. Algoritmen använd i denna implementering är modellerad efter [10], vilket leder till att källans förslagna kartläggningsmetod bör även passa denna implementering. Något som stärker detta är att [10] förslagna algoritm är designad med avseende på realtid, vilket även denna implementering är. Skulle denna metod användas behöver även rekursiviteten av stadie- och felkovarians estimeringen uppdateras, då [10] förslagna kartläggningsmetod påverkar dessa.

I brist på tid gjordes flera antaganden för Q och R matrisen i algoritmens EKF, vilket förenklade den drastiskt. En förbättring för algoritmen är att ta bort antagandena och även modellera med kovariansen, vilket eventuellt skulle leda till bättre resultat av EKF. För ännu bättre resultat kan även bruset och kovariansen uppdateras kontinuerligt under algoritmens gång. Detta då brus och kovarians inte är statiska utan kan dynamiskt ändras beroende på olika faktorer. Vidare kan även startinitieringen av felkovariansmatrisen förbättras, då i denna implementering är initieringen endast antagna värden som fungerat. Genom att nyttja en metod som tar fram bättre initieringsvärden kan resultatet av EKF eventuellt förbättras.

Slutligen kan en eventuell förbättring vara att ersätta den linjära interpoleringen med en punktestimering som passar bilens olinjära stadie bättre, vilket kan vara till exempel B-splines eller Bézier kurvor. Dock kan detta påverka exekveringstiden, vilket leder att en övervägning mellan resultat och tid måste göras.

6.3. Hållbarhet

Utvecklingen av denna algoritm möjliggör att bilen kan bättre tolka sin omgivning under sin färd, och utefter tolkningen välja hur den ska agera. Detta kan användas för att förstärka säkerheten både i manuellt körda bilar och självkörande bilar. Algoritmen har en tydlig roll framåt för att främja säkerheten i trafiken. Algoritmen kan också användas inom andra områden där det finns behov av en tydlig perception av omvärlden, vilket kan vara till exempel robotar.

Dock så finns även detta behov inom det militära och inom övervakning, vilket kan ha visa moraliska och etiska problem. På grund av att algoritmen ger så tydlig perception kan det anses som en form av övervakning, vilket kan skapa viss oro på en individ och samhällsnivå. Vidare kan den tydliga perceptionen nyttjas inom det militära för olika former av vapen, med avsikt för kontroll och eventuellt skada. Dessa kan både vara moraliskt rätt eller fel på en individ- och samhällsnivå, beroende på perspektiv. Det kan vara moraliskt rätt då bättre övervakning och militär bidrar till en bättre säkerhet för individ och samhälle. Det kan vara moraliskt fel då övervakningen kan anses som frihetsberövande och intrång, samt att stärka det militära med avsikt att kontrollera och skada kan anses ett missbrukande av algoritmen.

Algoritmen kan även nyttjas för att främja miljön, då algoritmen ger möjlighet till att svepa förbi och avläsa stora ytor i en störningspräglad miljö. Genom detta kan till exempel stora ytor av skog eller lantbruk avläsas för att detektera ändringar i miljön, vilket är gynnsamt för bevarandet av miljön.

Vidare så har även utrustningen som nyttjats i projektet en indirekt negativ påverkan på miljön. Utrustningen, vilket är beräkningsdator, Lidar och IMU, ökar energikonsumtionen i bilen. För att driva utrustningen behöver bilen genera mer energi vilket leder till att bilen antingen måste vara uppkopplad till laddare vilket konsumerar mer ström, eller stå på tomgång vilket konsumerar mer bränsle. I båda fallen resulterade det i viss negativ miljöpåverkan via ökat utsläpp och energikonsumtion.

REFERENSER

[1] "Introduction to LiDAR", Spie.org. [Online]. Tillgänglig: https://spie.org/samples/PM300.pdf. [Acc: 22- Feb- 2021]

[2] "3D Coordinate Systems for Lidar Sensor Explained", Lidarnews, 2020. [Online]. Tillgänglig: https://blog.lidarnews.com/3d-coordinate-systems-explained/. [Acc: 22- Feb- 2021]

[3] T. Raj, F. Hashim, A. Huddin, M. Ibrahim and A. Hussain, A Survey on LiDAR Scanning Mechanisms. Electronics, 2020, sid. 3-6 [Online]. Tillgänglig: https://www.researchgate.net/publication/341126299_A_Survey_on_LiDAR_Scanning_Mechanisms. [Acc: 22- Feb- 2021]

[4] *Luminar Hydra Specs*. Luminar, 2021, pp. 1-2 [Online]. Tillgänglig: https://www.luminartech.com/thank-you-hydra/. [Acc: 19- May- 2021]

[5] "What is an inertial measurement unit", *Vectornav.com*. [Online]. Tillgänglig: https://www.vectornav.com/resources/what-is-an-imu. [Acc: 03- Apr- 2021]

[6] OxTS, ”oxts.com,” Oxford Technical Solutions, 2020. [Online]. Tillgänglig: https://www.oxts.com/wp-content/uploads/2020/03/rtman-200302.pdf. [Acc: 03- Apr- 2021]

[7]"Accelerometer", *Omega.co.uk*. [Online]. Tillgänglig: https://www.omega.co.uk/prodinfo/accelerometers.html. [Acc: 04- Apr- 2021]

[8] J. Watson, "MEMS Gyroscope Provides Precision Inertial Sensing in Harsh, High Temperature Environments | Analog Devices", *Analog.com*. [Online]. Tillgänglig: https://www.analog.com/en/technical-articles/mems-gyroscope-provides-precision-inertial-sensing.html. [Acc: 04- Apr- 2021]

[9] "Kinematic equations", *Pasco*, 2021. [Online]. Tillgänglig: https://www.pasco.com/products/guides/kinematic-equations. [Acc: 09- Apr- 2021]

[10] K. Tokorodani, M. Hashimoto, Y. Aihara and K. Takahashi, *Point-cloud Mapping using Lidar Mounted on Two-wheeled Vehicle based on NDT Scan Matching*, 1st ed. Kyoto: scitepress, 2019, sid. 3-5.

[11] G. Strang and E. Herman, "14.3: Partial Derivatives", Mathematics LibreTexts, 2021. [Online]. Tillgänglig: https://math.libretexts.org/Bookshelves/Calculus/Book%3A_Calculus_(OpenStax)/14%3A_Differentiation_of_Functions_of_Several_Variables/14.3%3A_Partial_Derivatives. [Acc: 10- Apr- 2021].

[12] G. Flake, *The Calculus of Jacobian Adaptation*, vol.1. Researchgate, 2014, sid. 1.

[13] E. Dean, *EEN095 – L7: Forward Kinematics*, vol.1. Gothenburg: Chalmers, 2020, sid. 5-11.

[14] Y. Kim and H. Bang, *Introduction to Kalman Filter and Its Applications*, vol.1. Researchgate, 2018, sid. 1-10.

[15] "Linear Interpolation Formula", *toppr*. [Online]. Tillgänglig: https://www.toppr.com/guides/maths-formulas/linear-interpolation-formula/. [Acc: 12- Apr-2021].

BILAGOR

Bilaga A: Jacobians av bilens stadie

$$F_{k-1} = \frac{\partial f\left(\widehat{A_{k-1}}, u_{k-1}\right)}{\partial \widehat{A_{k-1}}}$$

$$= \begin{pmatrix}
1 & 0 & 0 & 0 & -Sq * Cy * b_1 & -Sy * Cq * b_1 & \tau * Cq * Cy & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & -Sq * Sy * b_1 & Cy * Cq * b_1 & \tau * Sy * Cq & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & -Cq * b_1 & 0 & -Sq * \tau & 0 & 0 & 0 \\
0 & 0 & 0 & 1 + Tq(b_2 * Cf - b_3 * Sf) & (Tq^2 + 1)(b_2 * Sf + b_3 * Cf) & 0 & 0 & 0 & \tau * Tq * Sf & \tau * Tq * Cf \\
0 & 0 & 0 & -Sf * b_2 - Cf * b_3 & 1 & 0 & 0 & 0 & \tau * Cf & -Sf * \tau \\
0 & 0 & 0 & \frac{1}{Cq}(b_2 * Cf - Sf * b_3) & \frac{Sq}{Cq^2}(b_2 * Sf + b_3 * Cf) & 1 & 0 & 0 & \frac{\tau * Sf}{Cq} & \frac{\tau * Cf}{Cq} \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}$$

$$G_{k-1} = \frac{\partial f\left(\widehat{A_{k-1}}, u_{k-1}\right)}{\partial a_{k-1}} =$$

$$\begin{pmatrix}
\frac{t^2}{2} * Cq * Cy & 0 & 0 & 0 \\
\frac{t^2}{2} * Cq * Sy & 0 & 0 & 0 \\
-(\frac{t^2}{2} * Sy) & 0 & 0 & 0 \\
0 & 0 & \frac{t^2}{2} * Tq * Sf & \frac{t^2}{2} * Tq * Cf \\
0 & 0 & \frac{t^2}{2} * Cf & -(\frac{t^2}{2} * Sf) \\
0 & 0 & \frac{t^2}{2} * \frac{Sf}{Cq} & \frac{t^2}{2} * \frac{Cf}{Cq} \\
t & 0 & 0 & 0 \\
0 & t & 0 & 0 \\
0 & 0 & t & 0 \\
0 & 0 & 0 & t
\end{pmatrix}$$

Bilaga B: EKF brus och kovariansmatriser

$$Q = \begin{pmatrix} Var(\dot{V}_x) & 0 & 0 & 0 \\ 0 & Var(\ddot{\mathrm{f}}) & 0 & 0 \\ 0 & 0 & Var(\ddot{\mathrm{q}}) & 0 \\ 0 & 0 & 0 & Var(\ddot{\mathrm{y}}) \end{pmatrix} = \begin{pmatrix} 0.027 & 0 & 0 & 0 \\ 0 & 0.00002 & 0 & 0 \\ 0 & 0 & 0.038 & 0 \\ 0 & 0 & 0 & 0.033 \end{pmatrix}$$

$$R = \begin{pmatrix} Var(\mathrm{f}) & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & Var(\mathrm{q}) & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & Var(\mathrm{y}) & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & Var(V_x) & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & Var(\dot{\mathrm{f}}) & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & Var(\dot{\mathrm{q}}) & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & Var(\dot{\mathrm{y}}) \end{pmatrix}$$

$$= \begin{pmatrix} 0.00008 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.00008 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.00009 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.00000032 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2.396 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2.474 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2.305 \end{pmatrix}$$

$$v = \begin{pmatrix} 0.009 \\ 0.006 \\ 0.0074 \\ 0.004 \\ 0.006 \\ 0.013 \\ 0.018 \end{pmatrix}$$

Bilaga C: Avlyssningsskript och algoritmhuvudkropp

```
7
8    import threading
9    import numpy as np
10   import rospy
11   from sensor_msgs.msg import PointCloud2
12   import ros_numpy.point_cloud2 as rn_pc2
13   import time
14   import ncom_wrapper
15   import Algorithm as Alg
16
17   def listener_imu():
18       #Retrieve global variables
19       global IMU_flag
20       global Lidar_flag
21       global IMU_data
22       global Stopper
23       global times_imu
24       global IMU_ready
25       #Choose point to listen to and wrap
26
27       NCOMlistener = ncom_wrapper.UDPHandler(3000, "", 1, "Server")
28       myNcom = ncom_wrapper.NCOMWrapper(NCOMlistener)
29
30       #Init reading list
31       IMU_list = []
32
33       while IMU_ready == False: #Check if the IMU has the needed valid flags, i.e warm up is done.
34           myNcom.updateDataStruct() #Sample IMU data
35           if( (myNcom.nrx.mIsFiltAxValid) & (myNcom.nrx.mIsFiltYxValid) & (myNcom.nrx.mIsIsoVoXValid) & (myNcom.nrx.mIsIsoYawValid)):
36               IMU_ready = True #Set ready flag.
37               print("IMU warm up is finished and is ready to be used!\n")
38
39       while True: #Inits done, inf loop
40           if Stopper == True: #Main has stopped end listener
41               print("IMU listener is closed!\n")
42               break
43           tStart = time.time() #Used for timing
44           myNcom.updateDataStruct() #Sample IMU data
45
46           #Take IMU sample
47           IMU_sample =[myNcom.nrx.mFiltAx, #V.F Filtered linear acceleration x-axis, 0
48                        myNcom.nrx.mFiltYx, #V.F Filtered angular acceleration x-axis, 1
49                        myNcom.nrx.mFiltYy, #V.F Filtered angular acceleration y-axis, 2
50                        myNcom.nrx.mFiltYz, #V.F Filtered angular acceleration z-axis, 3
51                        myNcom.nrx.mIsoRoll, #ISO v.s Roll, 4
52                        myNcom.nrx.mIsoPitch, #ISO v.s Pitch, 5
53                        myNcom.nrx.mIsoYaw, #ISO v.s Yaw, 6
54                        myNcom.nrx.mIsoVoX, #ISO v.s Linear velocity x-axis, 7
55                        myNcom.nrx.mWx, #V.F Angular rate x-axis, 8
56                        myNcom.nrx.mWy, #V.F Angular rate y-axis, 9
57                        myNcom.nrx.mWz] #V.F Angular rate z-axis, 10
58           IMU_list.append(IMU_sample) #Add list of measurment as a row
59           IMU_sample = [] #Empty list to prevent overwrite
60
61           if Lidar_flag == True: #Check if the Lidar callback has been called, i.e time to retreive IMU data.
62               IMU_flag = True # Flag to indicate that IMU data has been received.
63               IMU_data = IMU_list #Store to global variable
64               IMU_list = [] #Clear local list
65
66           times_imu = times_imu + 1 #Increment imu data counter, used to check for sync with IMU.
67
68           if (0.01-(time.time()-tStart)) > 0:   #Check if there is a need for sleep
69               time.sleep(0.01-(time.time()-tStart)) # ish 100Hz
70
71   def callback(data):
72       #Init global variables
73       global Lidar_flag
74       global Point_cloud
75       global Filt_cloud
76       global IMU_flag
77       global IMU_data
78       global IMU_ready
79       global State_est
80       global Error_cov_est
81       global X_prev
82       global times_imu
83       global times_lidar
84       global previous_time
85
86       Lidar_flag = True #Set Lidar flag True to indicate that the callback has been called, i.e time to retrieve IMU data.
87
88       #Retrieve Lidar scan
89       points = rn_pc2.pointcloud2_to_array(data)
90       ones = np.ones((1,len(points['x'])))
91       Lidar_scan = np.append([points['x'],points['y'],points['z']],ones, axis = 0)
92
93       if IMU_ready == True: #Check if IMU warm up done
94           while True: #Start algorithm
95               if IMU_flag == True: #Wait until IMU data is retrieved.
96                   #Reset flags
97                   Lidar_flag = False
98                   IMU_flag = False
99
100                  IMU_scan = np.array(IMU_data)#Retrive IMU data.
101                  #Start algorithm
102                  State_est, Error_cov_est, X_prev, Cloud = Alg.algorithm(Lidar_scan, IMU_scan, State_est, Error_cov_est, X_prev)
103                  #Extend point cloud
104                  Point_cloud.extend(Lidar_scan[0:3])
105                  Filt_cloud.extend(Cloud)
106                  break#Algoritm finished
107      else: #Warm up not done
108          print("IMU warm up not done yet\n")
109
110  if __name__ == '__main__':
111      rospy.init_node('listener', anonymous=True) #Init the Lidar node.
```

```
112
113     #Init the global variables
114     IMU_data = [] #Array to store IMU measurments
115     Lidar_flag = False #Flag to indicate that the Lidar scan is finsihed, and that IMU data should be recieved.
116     IMU_flag = False #Flag to indicate that the IMU data is recieved and the algorithm should start.
117     Stopper = False #Flag to indicate that the algorithm is stopped and the IMU listener should close.
118     IMU_ready = False #Flag to indicate IMU has the needed valid flags, i.e warm up is done.
119     Point_cloud = []
120     Filt_cloud = []
121     X_prev = np.zeros((6,1)) #Array to store linear and rotational pose of the car, is init as zero for the first iteration, 6x1
122     State_est = np.zeros((10,1)) # No previous state estimate upon start up, thus start at zero, 10x1
123     #No previous error covariance upon start up, thus values are guessed via trial and error, 10x10
124     Error_cov_est = np.array([ [0.1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
125                                [0, 0.1, 0, 0, 0, 0, 0, 0, 0, 0],
126                                [0, 0, 0.1, 0, 0, 0, 0, 0, 0, 0],
127                                [0, 0, 0, 0.1, 0, 0, 0, 0, 0, 0],
128                                [0, 0, 0, 0, 0.1, 0, 0, 0, 0, 0],
129                                [0, 0, 0, 0, 0, 0.1, 0, 0, 0, 0],
130                                [0, 0, 0, 0, 0, 0, 0.1, 0, 0, 0],
131                                [0, 0, 0, 0, 0, 0, 0, 0.1, 0, 0],
132                                [0, 0, 0, 0, 0, 0, 0, 0, 0.1, 0],
133                                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0.1] ])
134
135     Lidar = rospy.Subscriber('luminar_pointcloud', PointCloud2, callback) #Start the Lidar listener
136     x = threading.Thread(target=listener_imu) #Set the IMU listener as a thread which runs parallel to the Lidar and algorithm.
137     x.daemon = True #Set the IMU listener as daemon to ensure it is killed when main exits
138     x.start() #Start the IMU listener.
139
140     print("Listeners started\n")
141     rospy.spin() #Prevent the code from exiting until the user has pressed ctrl-c in the terminal.
142
143     print("\nAlgorithm finished, closing listeners\n")
144     Stopper = True #Set stopper flag true to end the IMU listener
145
146     print("Saving clouds")
147     g = 0
148     for x in range(0,len(Point_cloud)/3):
149         T_P = np.transpose(np.array(Point_cloud[g:g+3]))
150         T_F = np.transpose(np.array(Filt_cloud[g:g+3]))
151         np.savetxt('Raw_data1/data' + '%s.csv' % x, T_P, delimiter=',')
152         np.savetxt('Filtered_data2/Data' + '%s.csv' % x, T_F, delimiter=',')
153         g = g+3
154     print("Finished saving clouds, closing")
155
```

Bilaga D: Algoritm i kod

```python
import numpy as np
import EKF
import Hom_Trans as HT
import Constants as Const

def algorithm(Lidar_scan, IMU_scan, State_est, Error_cov_est, X_prev):
    Spill = 0 #Int flag to indicate yaw spill
    X_ref = X_prev #Store the previous final pose as the reference pose for this scan, ie world frame is this value now.
    k = 1 #Init IMU data counter, k=1... k=len(IMU_scan)+1
    Filt_Points_w = np.empty((4,len(Lidar_scan[0]))) #Init a matrix for the filtered points in world frame, 4xlen(points)
    index_start = 0 #Init vector start index as 0
    while (k < len(IMU_scan) + 1 ): #Perform the filtering for each IMU measurment

        #EKF
        #w is the acceleration disturbances which is read from the IMU in pos 0 to 3, len(IMU_scan)x4
        w = IMU_scan[:,0:4]
        State_est, Error_cov_est = EKF.run(State_est, Error_cov_est, w[k-1], Const.v, Const.Q, Const.R, IMU_scan[k-1,4:11], Const.H_IMU, Const.H_IMU_T, X_ref[3:6,0])
        X_now = np.dot(Const.H_X, State_est) #Retrieve state, 6x10*10x1=6x1

        #Check for yaw spill
        if (X_prev[5,0] >= 130) & (X_now[5,0] <= -130): #THE yaw has passed the 180 limit and jumped down to -180
            X_now[5,0] = X_now[5,0] + 360
            Spill = 1 #Indicate this spill with 1

        if (X_prev[5,0] <= -130) & (X_now[5,0] >= 130): #THE Yaw has passed the -180 limit and jumped up to 180
            X_now[5,0] = X_now[5,0] - 360
            Spill = 2 #Indicate this spill with 2

        j = 1 #Init interpolation counter, j= 1... j=17
        while (j < 16): #Interpolate 15 times

            X_intpol = X_prev + ( (X_now-X_prev)/0.01 )* j * 0.000625 #Linear interpolation eq, 1x6
            if X_intpol[5,0] > 180: #Interpol limit passed
                X_intpol[5,0] = X_intpol[5,0] - 360

            else:
                if X_intpol[5,0] < -180: #Interpol limit passed
                 X_intpol[5,0] = X_intpol[5,0] + 360

            #Calculate vector indexes:
            index_end = (len(Lidar_scan[0]) * j ) /(15*len(IMU_scan)) + ( len(Lidar_scan[0]) * (k-1) ) / len(IMU_scan)
            if index_end == len(Lidar_scan[0])-1:
                index_end = index_end +1


            #Transform from car frame to world frame:
            T = HT.Hom_Trans(X_intpol.T) #Compute HTM of interpolated state, 4x4
            Filt_Points_w[:, index_start:index_end ] = np.dot(T, Lidar_scan[:,index_start:index_end]) #4xlen(points)

            #Recursive:
            j = j + 1 #Increment interpolation counter
            index_start = index_end #Recursive for next iteration, takes next batch of points.

        if Spill == 1: # 180 to -180 yaw jump
            X_now[5,0] = X_now[5,0] - 360
        if Spill == 2: #-180 to 180 yaw jump
            X_now[5,0] = X_now[5,0] + 360
        X_prev = X_now #Current state estimate is stored and used in next iteration, 6x1
        k = k + 1 #Increment IMU counter

    #Transform back to car frame at the final IMU measurment
    T = np.linalg.pinv( HT.Hom_Trans(X_now.T) ) #Inverse HTM of final IMU measurment, 4x4
    Filt_Points_c = np.dot(T, Filt_Points_w) #Transform the filtred point cloud from world frame to car frame, 4xlen(points)

    #THIS PART CAN BE FURTHER IMPROVED WITH A MAPPING ALGORITM, SEE NDT SCAN MATCHING REPORT
    X_prev[0:3,0] = 0 #Reset XYZ pos, keep roll, pitch, yaw.
    State_est[0:len(State_est)] = 0 #Reset state estimate, this can be improved with GNSS and/or mapping algorithm
    Error_cov_est = np.array([  [0.1, 0, 0, 0, 0, 0, 0, 0, 0, 0],    #Reset error cov, same improvment as above.
                                [0, 0.1, 0, 0, 0, 0, 0, 0, 0, 0],
                                [0, 0, 0.1, 0, 0, 0, 0, 0, 0, 0],
                                [0, 0, 0, 0.1, 0, 0, 0, 0, 0, 0],
                                [0, 0, 0, 0, 0.1, 0, 0, 0, 0, 0],
                                [0, 0, 0, 0, 0, 0.1, 0, 0, 0, 0],
                                [0, 0, 0, 0, 0, 0, 0.1, 0, 0, 0],
                                [0, 0, 0, 0, 0, 0, 0, 0.1, 0, 0],
                                [0, 0, 0, 0, 0, 0, 0, 0, 0.1, 0],
                                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0.1] ])

    return(State_est, Error_cov_est, X_prev, Filt_Points_c[0:3])
```

Bilaga E: EKF i kod

```
import motion_model as MM
import numpy as np


def run(State_est_prev, Error_cov_prev, w, v, Q, R, IMU_scan, H_IMU, H_IMU_T, Ori_ref):
    IMU_scan = IMU_scan.reshape((7,1))

    #Get jacobians:
    F = MM.df_dstate(State_est_prev, w, 0.01, Ori_ref) #10x10
    G = MM.df_dw(State_est_prev, w, 0.01, Ori_ref) #10x4

    #Prediction:
    State_pred = MM.model(State_est_prev, w, 0.01, Ori_ref) #10x1

    #10x10 =(10x10*10x10)*10x10 + (10x4*4x4)*4x10
    Error_cov_pred = np.dot(np.dot(F, Error_cov_prev), F.T) + np.dot(np.dot(G,Q),G.T)

    #Estimation:
    h = np.dot(H_IMU, State_pred) + v #7x1 = 7x10*10x1+7x1
    y = IMU_scan - h #7x1 = 7x1 - 7x1
    measurment_residual = np.dot(np.dot(H_IMU, Error_cov_pred), H_IMU_T) - R #7x7=(7x10*10x10)*10x7 -7x7
    K = np.dot(np.dot(Error_cov_pred, H_IMU_T), np.linalg.pinv(measurment_residual)) # 10x7=(10x10*10x7)*7x7
    State_est = State_pred + np.dot(K, y) #10x1=10x1 + 10x7*7x1
    Error_cov_est = Error_cov_pred - np.dot(np.dot(K,H_IMU), Error_cov_pred) #10x10=10x10-(10x7*7x10)*10x10
    return(State_est, Error_cov_est)
```

Bilaga F: EKF modell och Jacobians i kod

```
from math import radians, cos, sin, tan
import numpy as np

def yaw(f, Ori_ref):
    #Prevent spill when jumping between 180 and -180
    Spill = 0
    if (Ori_ref <=  -130) & (f >= 130): #Yaw spill from -180 to 180
        f = f - 360
        Yaw = Ori_ref-f
        Spill = 1

    if (Ori_ref >=  130) & (f <= -130): #Yaw spill from 180 to -180
        f = f  + 360
        Yaw = Ori_ref-f
        Spill = 1

    if Spill == 0:
        Yaw = f- Ori_ref

    return(Yaw)

def angles(f, Ori_ref):
    #Calculate world frame orientation change
    Roll = f[3,0]- Ori_ref[0]
    Pitch = f[4,0]-Ori_ref[1]
    Yaw = yaw(f[5,0],Ori_ref[2])

    #Cosine shortcut, Retrieve alltitude angles from IMU paket. Compare with fixed frame.
    CR = cos(radians(Roll))
    CP = cos(radians(Pitch))
    CY = cos(radians(Yaw))


    #Sinus shortcut
    SR = sin(radians(Roll))
    SP = sin(radians(Pitch))
    SY = sin(radians(Yaw))

    #Tan shortcut
    TP = tan(radians(Pitch))

    return(CR, CP, CY, SR, SP, SY, TP)
```

```

def model(f, w, t, Ori_ref):
    #Compute orientations, displacements and caluclate motion model
    a1 = f[6,0]*t+(w[0]*t**(2))/2
    a2 = f[8,0]*t+(w[2]*t**(2))/2
    a3 = f[9,0]*t+(w[3]*t**(2))/2
    CR, CP, CY, SR, SP, SY, TP = angles(f, Ori_ref)
    Model = np.array([  [f[0,0] + a1*CP*CY],
                        [f[1,0] + a1*CP*SY],
                        [f[2,0] - a1*SP],
                        [f[3,0] + (a2*SR+a3*CR)*TP],
                        [f[4,0] + (a2*CR-a3*SR)],
                        [f[5,0] + (a2*SR+a3*CR)*(1/CP)],
                        [f[6,0] + w[0]*t],
                        [f[7,0] + w[1]*t],
                        [f[8,0] + w[2]*t],
                        [f[9,0] + w[3]*t] ])
    return(Model)

def df_dw(f, w, t, Ori_ref):
    #Compute the jacobian, the partial derivate of the motion model in respect to the acceleration distrubances, 10x4
    CR, CP, CY, SR, SP, SY, TP = angles(f, Ori_ref)
    G = np.array([ [CP*CY*t**(2)/2, 0, 0, 0],
                   [CP*SY*t**(2)/2, 0, 0, 0],
                   [-SP*t**(2)/2,   0, 0, 0],
                   [0, 0, (SR*t**(2)/2)*TP, (CR*t**(2)/2)*TP],
                   [0, 0, CR*t**(2)/2, -SR*t**(2)/2],
                   [0, 0, (SR*t**(2)/2)*(1/CP), (CR*t**(2)/2)*(1/CP)],
                   [t, 0, 0, 0],
                   [0, t, 0, 0],
                   [0, 0, t, 0],
                   [0, 0, 0, t]])
    return (G)
```

```

def df_dstate(f, w, t, Ori_ref):
    #Compute the jacobian, the partial derivate of the motion model in respect of the motion model, 10x10
    a1 = f[6,0]*t+(w[0]*t**(2))/2
    a2 = f[8,0]*t+(w[2]*t**(2))/2
    a3 = f[9,0]*t+(w[3]*t**(2))/2
    CR, CP, CY, SR, SP, SY, TP = angles(f, Ori_ref)
    dTP_dP = 1/(CP**(2)) #Partial derivative of tan(pitch) in respect to pitch
    d1_CP_dP= SP/(CP**(2)) #Partial derivative of 1/cos(pitch) in respect to pitch
    F = np.array ([[1.0, 0, 0, 0, a1*CY*(-SP), a1*CP*(-SY), t*CP*CY, 0, 0, 0],
                   [0, 1.0, 0, 0, a1*SY*(-SP), a1*CP*CY, t*CP*SY, 0, 0, 0],
                   [0, 0, 1.0, 0, -a1*CP, 0, -t*SP, 0, 0, 0],
                   [0, 0, 0, 1.0+TP*(a2*CR-a3*SR), (a2*SR+a3*CR)*dTP_dP, 0, 0, 0, t*SR*TP, t*CR*TP],
                   [0, 0, 0, a2*(-SR)-a3*CR, 1.0, 0, 0, 0, t*CR, -t*SR],
                   [0, 0, 0, (a2*CR-a3*SR)*(1/CP),(a2*SR+a3*CR)*d1_CP_dP, 1.0, 0, 0, t*SR*(1/CP), t*CR*(1/CP)],
                   [0, 0, 0, 0, 0, 0, 1.0, 0, 0, 0],
                   [0, 0, 0, 0, 0, 0, 0, 1.0, 0, 0],
                   [0, 0, 0, 0, 0, 0, 0, 0, 1.0, 0],
                   [0, 0, 0, 0, 0, 0, 0, 0, 0, 1.0]])
    return (F)

```

Bilaga G: EKF konstanter i kod

```
7   import numpy as np
8
9   #Used to retrieve linear and rotation values, the X matrix,6x10
10  H_X =np.array ([ [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],      #Used to retrieve x
11                   [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],      #Used to retrieve y
12                   [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],      #Used to retrieve z
13                   [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],      #Used to retrieve roll
14                   [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],      #Used to retrieve pitch
15                   [0, 0, 0, 0, 0, 1, 0, 0, 0, 0] ])    #Used to retrieve yaw
16
17  #Used to retrieve sensor measurments, 7x10
18  H_IMU = np.array([ [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],     #Used to retrieve roll
19                     [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],     #Used to retrieve pitch
20                     [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],     #Used to retrieve yaw
21                     [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],     #Used to retrieve linear x velocity
22                     [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],     #Used to retrieve roll rate
23                     [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],     #Used to retrieve pitch rate
24                     [0, 0, 0, 0, 0, 0, 0, 0, 0, 1] ])  #Used to retrieve yaw rate
25  H_IMU_T = np.transpose(H_IMU) #Transpose of measurments matrix
26
27  #Sensor noise, len(IMU_scan[0,4:11])x1=7x1 (acceleration disturbance noise removed)
28  v = np.array([ [0.009],
29                 [0.006],
30                 [0.0074],
31                 [-0.04],
32                 [0.006],
33                 [0.013],
34                 [0.018] ])
35
36  #Covariance matrix of process noise(acc disturbances), assume there is only variance, 4x4
37  Q = np.array([ [0.02700, 0, 0, 0],
38                 [0, 0.00002, 0, 0],
39                 [0, 0, 0.03800, 0],
40                 [0, 0, 0, 0.03300] ])
41  #Covariance matrix of sensor noise, assume that there aren't any covariance only variance 7x7
42  R = np.array ([ [0.000080000, 0, 0, 0, 0, 0, 0],
43                  [0, 0.000080000, 0, 0, 0, 0, 0],
44                  [0, 0, 0.000090000, 0, 0, 0, 0],
45                  [0, 0, 0, 0.000000032, 0, 0, 0],
46                  [0, 0, 0, 0, 2.396000000, 0, 0],
47                  [0, 0, 0, 0, 0, 2.474000000, 0],
48                  [0, 0, 0, 0, 0, 0, 2.305000000] ])
```

Bilaga H: HTM i kod

```python
import numpy as np
from math import cos, sin, radians

def Hom_Trans(X):
    #Cosine shortcut, Retrieve alltitude angles from IMU paket.
    CR = cos(radians(X[0,3]))
    CP = cos(radians(X[0,4]))
    CY = cos(radians(X[0,5]))

    #Sinus shortcut
    SR = sin(radians(X[0,3]))
    SP = sin(radians(X[0,4]))
    SY = sin(radians(X[0,5]))

    #Compute homogenious tranformation matrix
    T = np.array([[CP*CY, SR*SP*CY-CR*CY,  CR*SP*CY+SR*SY, X[0,0]],\
                  [CP*SY,  SR*SP*SY+CR*CY,  CR*SP*SY-SR*CY, X[0,1]],\
                  [-SP,    SR*CP,           CR*CP,          X[0,2]],\
                  [0,      0,               0,              1]])
    return(T)
```

INSTITUTIONEN FÖR ELEKTROTEKNIK
LIDAR RÖRELSEDISTORSIONSFILTRERING
SIMON KRANTZ, PONTUS ÖSTBERG
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2021

CHALMERS