FFI-rapport 2015/00713

Autokalibrering av kamera ved bruk av rotasjon og vinkelmåling

Marius Halsør og Jørgen Rennemo

FFI Forsvarets forskningsinstitutt

Autokalibrering av kamera ved bruk av rotasjon og vinkelmåling

Marius Halsør og Jørgen Rennemo

Forsvarets forskningsinstitutt (FFI)

5. juni 2015

FFI-rapport 2015/00713

1263

P: ISBN 978-82-464-2526-9
E: ISBN 978-82-464-2527-6

Emneord

Kalibrering

Kamera

Bildebehandling

Bilder

MATLAB

Godkjent av

Halvor Ajer	Prosjektleder
Jon E. Skjervold	Avdelingssjef

Sammendrag

Dette arbeidet er utført med tanke på kameraer som skal brukes for Augmented Reality (AR). AR innebærer at man viser virtuell informasjon plassert på korrekt sted i en reell scene. For å få til dette, må man vite hvor kameraet befinner seg, og hvilken retning det peker i. Men man må også kjenne kameraets egenskaper, dvs hvordan det overfører en 3-dimensjonal scene til et 2-dimensjonalt bildeplan. For et "vanlig" digitalt kamera er aktuelle parametere fokallengde, beskrivelse av kameraets forvrengning og hvilket pixel i bildet som tilsvarer det optiske sentrum.

Vi ønsket en metode for å finne kameraets parametere, og metoden må ikke være for omfattende eller ta for lang tid. Den skulle heller ikke kreve at man tok bilde av spesielle mønstre. Metoden vi kom frem til, baserer seg på at man holder kameraet i samme posisjon og tar bilder mens man roterer kameraet om to akser normalt på optisk akse. Bildene med tilhørende vinkelverdier brukes så i et dataprogram.

Programmet gjenkjenner automatisk punkter i scenen i de forskjellige bildene. Med et bestemt parametersett skal ett punkt i ett bilde avbildes til ett bestemt punkt i et annet bilde. Avviket mellom dette beregnede punktet og det faktisk, målte punktet i det nye bildet er en "feil". Gitt den matematiske kameramodellen, finner algoritmen det parametersettet som gir den minste totale feilen for alle gjenkjente punkter i alle bildene, og angir dette som det beste estimatet for kameraets parametere.

English summary

We have conducted this work on camera calibration with Augmented Reality (AR) in mind. AR means displaying virtual information placed in correct positions in a real world, usually viewed through a camera. In order to achieve this, one must know the camera's position and orientation. In addition, one needs to know the basic camera properties. For "ordinary" digital cameras, such parameters are focal length, a description of the camera's distortion and which pixel that corresponds to the camera's optical center.

We looked for a method for finding these camera parameters that was reasonably user friendly, did not require too long processing time, and did not require taking pictures of specific patterns. The method we decided on, is based on a camera in a fixed position, taking pictures while it is being rotated around two axes normal to the optical axis. The images, along with their corresponding recorded viewing angles, are then used as input to a computer program.

The program automatically recognizes points in the scene in the various images. With a certain set of parameters, one such point should be mapped to a specific location in another image. The difference between this location and the actually observed location, is an "error". Given the mathematical model, the algorithm finds the set of parameters that yields the smallest total error for all recognized points in all the images, and outputs this as the best estimate for the camera parameters.

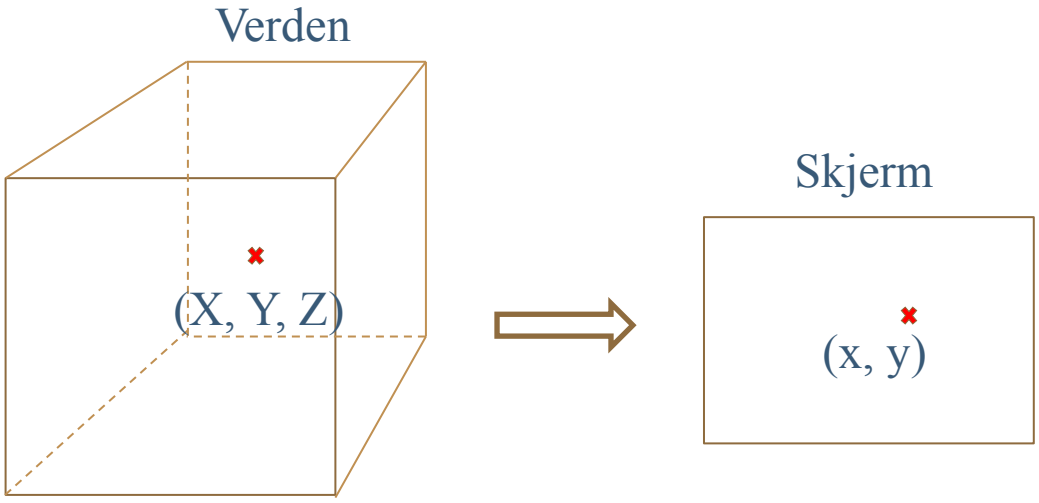
Innhold

1 Innledning

Et kamera plassert i et landskap er representert ved en funksjon

$$P : \mathbb{R}^3 \rightarrow \mathbb{R}^2$$

Her er $\mathbb{R}^3$ punkter i den tredimensjonale, virkelige verden, mens $\mathbb{R}^2$ er punkter i det todimensjonale bildeplanet. I et AR-system vil det være nødvendig å bestemme denne funksjonen for å kunne plassere interessepunkter på riktig sted i skjermbildet. For å gjøre dette, må man for det første kjenne kameraets posisjon og synsretning, og for det andre må man kjenne de indre kameraparametrene, som f.eks. zoom (brennvidde) og prinsipalpunktet (avbildningen av optisk akse). For å oppnå større nøyaktighet i plasseringen av punkter, kan det også være nødvendig å kjenne eventuell forvrengning i linsen. Vi har undersøkt en metode som finner disse indre parameterne uten bruk av spesielle kalibreringsmotiver, men med tilgang på nøyaktig måling av vinkler. Et kalibreringsmotiv er en bestemt gjenstand eller et bestemt mønster man avbilder i kalibreringsøyemed, for eksempel et rutenett.



Figur 1.1 Et kamera er representert ved en funksjon som overfører punkter fra en tredimensjonal virkelighet til en todimensjonal skjerm

Problemet vi studerer er altså å finne de indre kalibreringsparametrene til et kamera, gitt et oppsett hvor kameraets posisjon er fiksert, men hvor det kan roteres (pan og tilt), og hvor vi kan lese av rotasjonsvinkler med stor nøyaktighet. En variant av metoden er tidligere studert av G. Stein i [5] og [6].

Det finnes en rekke andre måter å kalibrere et kamera på. Fordelene med denne metoden er at 1) det kreves ikke spesielle kalibreringsmotiver, 2) kameraets posisjon trenger ikke å endres.[1] En kalibrering med vår metode kan gjennomføres med bare to bilder, men det er ikke undersøkt hva slags effekt dette vil ha på sikkerheten i estimatet for kameraparametrene.

2 Kameramodellen

Vi vil her beskrive den matematiske modellen vi bruker for å representere et kamera.

2.1 Pin hole-modellen

Den enkleste modellen for et kamera fås ved å se bort fra forvrengninger i linsen, og kalles pin hole-modellen. En slik modell er bestemt av følgende data:

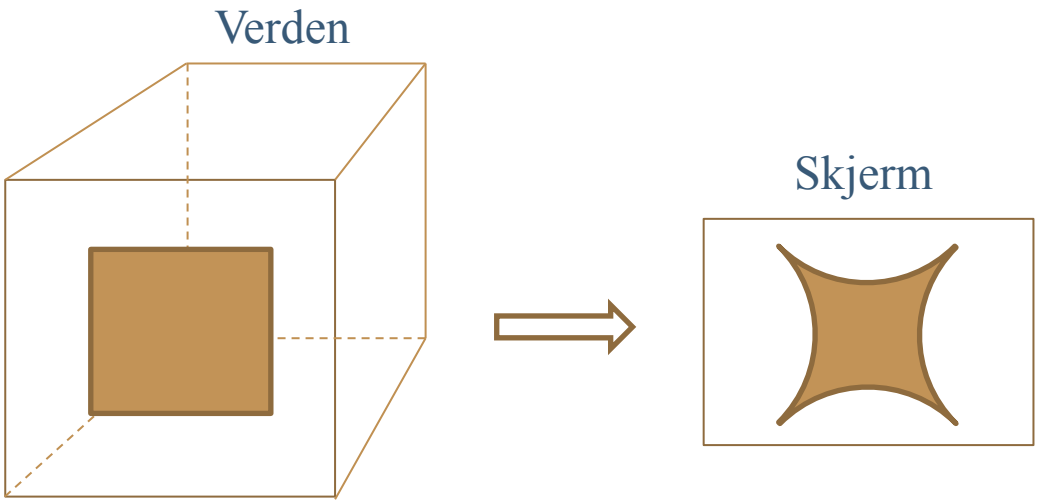
- Posisjon til optisk senter i rommet.
- Retningen kameraet peker i (optisk akse).
- Kameraets brennvidde, kalt *f*.
- Prinsipalpunktet, dvs. punktet hvor optisk akse møter bildeplanet. Dette angis med pikselkoordinater (c_x, c_y).

De to første punktene i denne listen kalles ytre kalibreringsparametre, og de to siste kalles *indre* parametre.

Hvis brennvidden er *f*, er bildeplanet det planet som ligger i avstand *f* fra optisk senter, og som står normalt på prinsipalaksen (optisk akse). Transformasjonen som representerer fotografering av scenen, vil da være projeksjonen av rommet ned i bildeplanet gjennom optisk senter, etterfulgt av en translasjon som plasserer prinsipalpunktet i (c_x, c_y).

[1] Faktisk er det i flere kalibreringsalgoritmer et krav at kameraet eller motivet må forandre posisjon mellom bildene.

2.2 Distorsjon



Figur 2.1 På grunn av forvrengning i kameraet kan linjer som er rette i virkeligheten, fremstå som buer på skjermen

En ekte linse vil som regel ha noen grad av forvrengning, som kan modelleres på følgende måte:

For et vilkårlig punkt i rommet lar vi (x_u, y_u) være kamerakoordinatene (med origo i prinsipalpunktet) til projeksjonen av punktet i henhold til pin hole-modellen beskrevet ovenfor. For å finne de faktiske koordinatene der punktet vil vises, beregner vi da

$$(x, y) = K(r) \cdot (x_u, y_u)$$

Her er $r = \sqrt{x_u^2 + y_u^2}$ avstanden fra (x_u, y_u) til prinsipalpunktet, og K er en funksjon som avhenger av linsens egenskaper. Et vanlig valg for K er

$$K(r) = 1 + k_1 r^2 + k_2 r^4$$

Med denne antakelsen er det altså de to parametrene k_1 og k_2 som skal bestemmes for å finne distorsjonen til kameraet.

Dette er det vanligste valget av K. Det finnes andre kandidater for valg av K som tar hensyn til mer eller mindre kompleks forvrengning i linsen. Hvilken funksjon som er passende, vil avhenge av det enkelte kamerasystemet..

2.3 Fullstendig modell for bildedannelse

Vi lar X, Y, Z være koordinater i rommet og velger koordinatsystem slik at kameraets optiske senter er i origo, og kameraet (prinsipalaksen) peker i positiv Z-retning. Videre

antar vi at X- og Y -aksene samsvarer med kameraets x- og y-koordinater. Et punkt som befinner seg i (X, Y, Z), vil da avbildes etter følgende kjede av funksjoner:

$$\begin{aligned}&(X,Y,Z)\rightarrow(f\cdot X/Z,\, f\cdot Y/Z)\rightarrow K(r)\cdot(f\cdot X/Z,\, f\cdot Y/Z)\\&\rightarrow(K(r)\cdot f\cdot X/Z+c_x, K(r)\cdot f\cdot Y/Z+c_y)\end{aligned} \tag{1}$$

Her er første steg projeksjonen, andre steg distorsjonen, og siste steg er translasjonen som plasserer origo i øvre, venstre hjørne av bildet, som er måten MatLab representerer bilder på.

3 Metoden

De vesentlige trekkene ved metoden er hentet fra artiklene [6] og [5]. Den grunnleggende ideen er at hvis et kamera tar to bilder forbundet av ren rotasjon med kjente (målte) vinkler og hvis de indre kameraparametrene er kjente, vil vi kunne beregne hvor hvert punkt i bilde 1 vil gjenfinnes i bilde 2. Hvis vi velger ut landemerker i bilde 1, kan man i bilde 2 sammenligne beregnet posisjon med målt posisjon for hvert landemerke. På grunn av støy eller fordi man bruker gale verdier for de indre kameraparametrene, vil beregningen ikke stemme nøyaktig overens med målingen, men et godt estimat for de indre kameraparametrene vil da være de verdiene som minimerer feilen i denne beregningen.

Oppskriften på en kalibrering er som følger:

1. Ta bilder og mål kameraets retning for hvert bilde.
2. Finn potensielle landemerker i hvert bilde.
3. Match alle landemerker i ulike bilder med hverandre.
4. Minimer avviket mellom målte og beregnede landemerker.

Steg 1 er selvforklarende. For steg 2 og 3 har vi brukt ferdigskrevne Matlab-rutiner som finnes i Computer Vision Toolbox[2], se kapittel 5. Steg 4 er mer innviklet og er beskrevet i kapittel 6.

4 Matematisk beskrivelse av kostfunksjonen

For å beregne effekten av en gitt rotasjon på bildepunktet p =(x, y), gjøres følgende:

1. Inverter (1) i Seksjon 2.3 for å finne et punkt (X, Y, Z) som vil avbildes til p.
2. Beregn effekten av rotasjon på (X, Y, Z), som er

[2] Denne er ikke installert som standard i MatLab på FFI

$$\begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

3. Bruk (1) til å beregne projeksjonen av punktet (X', Y', Z').

Hvis rotasjonsmatrisen er

$$R = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}$$

kan transformasjonen altså beskrives eksplisitt som

$$x' = K \cdot f \frac{r_{11}(x - c_x) + r_{12}(y - c_y) + r_{13}Kf}{r_{31}(x - c_x) + r_{32}(y - c_y) + r_{33}Kf} + c_x$$

$$y' = K \cdot f \frac{r_{21}(x - c_x) + r_{22}(y - c_y) + r_{23}Kf}{r_{31}(x - c_x) + r_{32}(y - c_y) + r_{33}Kf} + c_y$$

En praktisk metode for å finne rotasjonsmatrisen er å bruke Rodrigues' formel. Hvis W = ($w1$, $w2$, $w3$) er en vektor av lengde 1, og R representerer en rotasjon av størrelse θ med W som rotasjonsakse, har vi

$$R = \cos\theta \cdot I + \sin\theta \cdot Q + (1 - \cos\theta) W^T W,$$

hvor I er identitetsmatrisen og

$$Q = \begin{pmatrix} 0 & -w_3 & w_2 \\ w_3 & 0 & -w_1 \\ -w_2 & w_1 & 0 \end{pmatrix}.$$

I vårt eksperimentelle oppsett vil rotasjon om x-aksen være direkte beregnbar fra denne formelen, med W = (1, 0, 0). Rotasjon om y-aksen med vinkel θ vil også være direkte beregnbar slik, dersom kameraet peker i retning xz-planet. Hvis dette ikke er tilfellet, kan en transformasjon fra (θ_x,θ_y) til (θ_x',θ_y') modelleres som komposisjonen av transformasjoner

$$(\theta_x, \theta_y) \to (\theta_x, 0) \to (\theta_x', 0) \to (\theta_x', \theta_y')$$

Hver enkelt av disse transformasjonene svarer til en dreining langs en av våre vinkelmåleres akser, og den endelige rotasjonsmatrisen vil være produktet av matrisene for slike elementære rotasjoner.

I tillegg kan det tenkes at kameraet har en roll, hvilket tilsvarer en rotasjon om z-aksen. Denne kan være forskjellig for bilde 1 og bilde 2, og vi får i så fall følgende transformasjoner for å gå fra ($\theta_x, \theta_y, \theta_z$) til ($\theta_x', \theta_y', \theta_z'$):

$$(\theta_x, \theta_y, \theta_z) \to (\theta_x, \theta_y, 0) \to (\theta_x, 0, 0) \to (\theta_x', 0, 0) \to (\theta_x', \theta_y', 0) \to (\theta_x', \theta_y', \theta_z')$$

Metoden håndterer roll dersom man oppgir slike vinkler, men i resten av dokumentet ser vi bort fra roll, da det å inkludere roll ikke bidrar til å forklare hvordan metoden fungerer.

4.1 Kostfunksjon og minimering

Hvis vi har tatt to bilder med ulike vinkelutslag og gjenfinner samme landemerke i begge bildene, får vi det vi kaller en observasjon. I vårt oppsett (hvis vi ser bort fra roll i kameraet) er en observasjon definert ved 8 tallverdier, p_x, p_y, θ_x, θ_y ,q_x, q_y, $\varnothing_x$ og $\varnothing_y$. Her er (p_x, p_y) pikselverdiene til landemerket observert i bilde 1, mens (q_x, q_y) er pikselverdiene til samme landemerke i bilde 2. Vinkelavlesningene for henholdsvis bilde 1 og 2 er gitt ved (θ_x, θ_y) og ($\varnothing_x$, $\varnothing_y$).

Fastsetter vi kameraparametre (f, c_x, c_y, k_1, k_2) og bruker metoden over, kan vi beregne en forventet verdi for posisjonen til landemerket i bilde 2. Vi kaller denne posisjonen (q_x', q_y'). Et mål på feilen i kameraparametrene er da avstanden mellom forventet og faktisk posisjon til landemerket i bilde 2, det vil si

$$\sqrt{\left(q_x - q_x'\right)^2 + \left(q_y - q_y'\right)^2}\,.$$

En optimal verdi for kameraparametrene er da den som minimerer kostfunksjonen

$$\mathrm{C}\left(f, c_x, c_y, k_1, k_2\right) = \sum \left(q_x - q_x'\right)^2 + \left(q_y - q_y'\right)^2$$

hvor summen er over alle «observasjoner».

Det finnes alternative kostfunksjoner (se for eksempel [6], side 93-102), og selv om dette ikke er den ideelle kostfunksjonen, er den relativt enkel å regne med, og trolig "god nok". Man bør imidlertid inkludere feilen "begge veier", altså både avstanden mellom beregnet landemerke med utgangspunkt i bilde 1 og målt landemerke i bilde to, og mellom beregnet landemerke med utgangspunkt i bilde 2 og målt landemerke i bilde 1.

5 Landemerkegjenkjenning

5.1 Beskrivelse

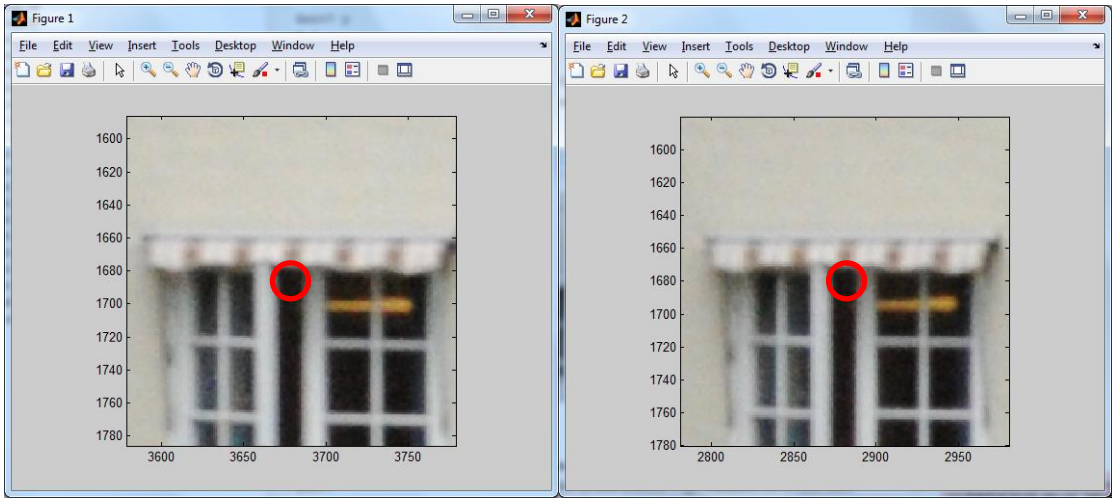
Til gjenkjenning og matching av landemerker er SURF-algoritmen [2] brukt, slik denne er implementert i Computer Vision System Toolbox i Matlab. For hvert par av bilder som skal sammenlignes leser vi først av alle potensielle landemerker i hvert bilde ved hjelp av `detectSURFFeatures`, for så å finne potensielle par ved hjelp av `matchFeatures`-rutinen.

Her er det to parametre som kan stilles inn. For det første tar `detectSURFFeatures` innstillingen `MetricThreshold`, som justerer hvor sterkt et punkt må skille seg ut fra omgivelsene før vi regner det som interessant. For det andre tar `matchFeatures` innstillingen `MatchThreshold`, som justerer hvor like to interessante punkter må være for at de regnes som en match.

5.2 Problemer

Det er to problemer som kan oppstå ved bruk av automatisk landemerkedeteksjon. For det første kan det være for få landemerker som oppdages, eller at landemerkene befinner seg i en liten region av bildeflaten. Ideelt sett skal landemerkene være spredt utover hele bildeflaten. Til en viss grad kan dette kompenseres for ved å redusere `MetricThreshold`-verdien, men bare hvis bildet i utgangspunktet inneholder gjenkjennelige punkter. (En gressplen eller en himmel vil sjelden inneholde SURF-punkter.) Særlig for å oppnå pålitelige verdier for distorsjonskoeffisientene vil det være gunstig at det finnes gjenkjente landemerker i nærheten av bildekanten, da effekten av distorsjon er mest merkbar her. Det kan være et poeng å utvide metoden med en rutine som fjerner punktpar i en region der det er mange landemerker, slik at punktparene som brukes i beregningen, er spredt jevnest mulig utover bildeflaten, eventuelt kan man vekte punktparene, slik at punktpar i «øde» områder av bildet blir tillagt mer vekt. Dette er foreløpig ikke gjort.

Det andre problemet er at man kan finne falske punktpar hvis det finnes flere punkter i bildet med lignende omegner. Dette problemet vil reduseres ved å senke verdien for `MatchThreshold`, og ved å bruke RANSAC-algoritmen i stedet for direkte least squares-løsning i beregningen av parametre. I bildesettene vi har brukt har dette problemet ikke spilt noen stor rolle. Vi har testet MatLabs gjenkjenningsalgoritme på mange bilder, og i våre tester har vi så langt ikke fått et eneste falskt punktpar.



Figur 5.1 Det samme landemerket gjenkjennes i bilde 1 og bilde 2, og det er liten tvil om at det faktisk er samme punkt.

5.3 Tidsbruk

Tiden det tar å trekke ut SURF-landemerker fra et bilde ser etter noen enkle tester ut til å vokse lineært med antall piksler, og tar på vårt system mellom 4 og 5 sekunder for et bilde med oppløsning 4128x2832.

5.4 Alternative landemerkedetektorer

Å trekke ut landemerker og matche disse har ikke vært hovedfokus, og metoden som er brukt, er valgt på bakgrunn av enkelhet heller enn ytelse. Det kan undersøkes om det finnes andre algoritmer som løser denne oppgaven bedre enn SURF-metoden. Det er for eksempel veldig sannsynlig at det kan utnyttes at bildene vi sammenligner verken er rotert eller skalert i forhold til hverandre, men tilnærmet bare translatert. Det er også sannsynlig at vi ved å "gjette" på hvor vi skal finne et matchende landemerke kan redusere søkeområdet, og dermed tidsbruken. For vår bruk har imidlertid ikke tidsbruken til denne metoden vært begrensende på noen måte.

6 Parametre fra observasjoner

Vi vil nå beskrive hvordan parameterne kan estimeres fra observasjonene (steg 4 i Seksjon 3).

6.1 Least squares

Vi regner den beste løsningen for å være den som minimerer kostfunksjonen

$$\mathrm{C}\left(f, c_x, c_y, k_1, k_2\right) = \sum \left(x'_{1ud} - x_{2ud}\right)^2 + \left(y'_{1ud} - y_{2ud}\right)^2 .$$

En måte å beregne kameraparameterne finnes da ved å bruke en «nonlinear least squares»-algoritme til å beregne de verdier av kameraparameterne som minimerer C. Opprinnelig brukte vi algoritmen «NL2SOL» [3], med en Matlab-front til denne i «OPTI Toolbox» [1]. Her må det oppgis en initiell verdi for kameraparametrene, og algoritmen vil så iterativt finne fram til et lokalt minimum for kost-funksjonen. Senere har vi også benyttet Matlab-funksjonen «fminsearch», som fungerer på omtrent samme måten, men ikke krever «OPTI Toolbox».

I de datasettene vi har kjørt ser metoden NL2SOL ut til å håndtere dårlige initielle gjetninger godt, og konvergerer mot samme løsning over et stort spenn av initialverdier. Metoden fminsearch ser ut til å være noe mer avhengig av en brukbar initiell gjetning. Metodene vil kunne påvirkes av outliers[3], som f.eks. kan oppstå hvis det finnes feilidentifiserte par av landemerker. For å bøte på dette har vi også implementert en RANSAC-beregning, beskrevet i Seksjon 6.2. I våre tester virker dette imidlertid unødvendig, ettersom alle landemerkene har blitt korrekt gjenkjent.

6.1.1 Beregning av gradient

Least squares-solveren vi har brukt (og de fleste andre så vidt vi vet) bruker gradienten til kostfunksjonen til iterativt å komme med nye gjetninger på parametre som minimerer kostfunksjonen. Det finnes numeriske algoritmer for beregning av gradienten, men for best ytelse har det vært nødvendig å beregne denne for hånd. Den totale transformasjonen som gir kostfunksjonen er nokså komplisert, men den er sammensatt av enklere funksjoner som kan deriveres for hånd. Disse kan så settes sammen ved å multiplisere matriser, noe som er en enkel operasjon i Matlab.

Funksjonen fminsearch er ikke avhengig av slike gradienter, og bruker Nelder-Meads metode.

6.2 RANSAC

RANSAC (Random Sample Consensus) er en type algoritmer designet for å løse problemet med å tilpasse en modell til et datasett med (muligens svært mange) outliers. Dette kan potensielt forekomme i vår kalibreringsprosedyre, hvis det er mange falske positive par i landemerke-matchingen.

Under følger en grunnleggende beskrivelse av RANSAC slik den er implementert i vår kode. Vi er gitt et datasett med N datapunkter, hvorav mange kan være outliers.

```
numTrials :=0, bestNumInliers := 0, maxTrials = inf
while numTrials < maxTrials
```

[3] Vi bruker begrepene “outliers” og “inliers” i mangel av gode norske alternativer

```
        Velg tilfeldig n datapunkter, hvor n er det minst
        antall datapunkter som trengs for å bestemme en
        modell
        Beregn en modell M ut fra de valgte n punktene
        inliers := de punkter som stemmer med M opp til gitt
        nøyaktighet
        k := antall punkter i inliers
        if k > bestNumInliers
                bestNumInliers := k, besteModell := M,
        besteInliers := inliers
        end
        numTrials := numTrials + 1
        maxTrials := antall trials vi må gjøre for at det
        skal være 99% sikkert at vi har funnet minst en
        sample som bare inneholder inliers, gitt en
        antakelse om at antall inliers > bestNumInliers
end
print besteModell og besteInliers
```

Hvis vi i vårt problem skal bestemme *k* kameraparametre, vil vi i algoritmen bruke *n* = *k/2* (rundet opp), ettersom hvert datapunkt legger to begrensninger ($q_x^{'} = q_x$ og $q_y^{'} = q_y$) på de ukjente.

Vi har brukt en ferdig implementasjon av RANSAC funnet i [4]. Hvorvidt et punkt er en inlier eller ikke, avgjøres av avstanden mellom forventet og faktisk posisjon av andre pikselpunkt. Grensen for hvor stor avstand man tillater for inliers, er en egen innstilling i algoritmen. RANSAC produserer en modell, men er hovedsakelig egnet til å forkaste outliers, så etter at RANSAC-algoritmen har kjørt ferdig, bør en least squares-solver kjøres på det reduserte datasettet bestående av beregnede inliers.

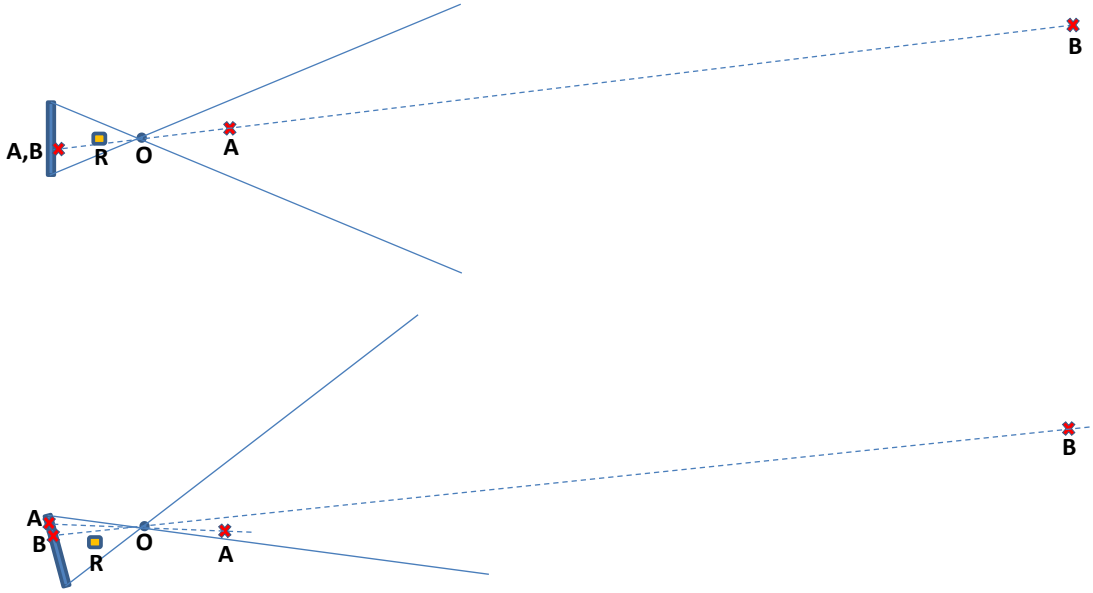
7 Feilkilder

Det er flere mulige feilkilder for metoden.

7.1 Uren rotasjon

Hvis det optiske senteret til kameraet ikke ligger på rotasjonsaksen, vil vi få systematiske feil i målingene. En beregning i [6] viser at om optisk senter er plassert i en avstand *D* direkte foran rotasjonsaksen, vil dette medføre en feil i estimatet for *f* på størrelse med *fD/Z*, hvor *Z* er avstanden til landemerket som brukes for estimatet. Av dette ser vi at stor avstand til landemerkene i noen grad kan kompensere for at kameraet ikke opplever nøyaktig ren rotasjon.

En annen effekt av uren rotasjon er at transformasjonen et pikselpunkt opplever under rotasjon fra bilde 1 til bilde 2 avhenger av avstanden fra landemerket til kameraet (parallakse-feil). Da vil det være vanskelig å finne gode parametere som passer med alle observasjoner dersom det finnes landemerker på svært ulike avstander i bildet (spesielt dersom det er landemerker på korte avstander, se Figur 7.1). Med andre ord vil minimumsverdien til kostfunksjonen *C* være relativt høy. Dette problemet oppstår ikke om alle landemerker befinner seg på samme avstand, men som nevnt ovenfor, vil man likevel kunne få feil i estimatene for kameraparameterne.



Figur 7.1 Her er kameraets optiske senter O, mens kameraet roteres om punktet R. I den øverste figuren ligger A og B på en rett linje gjennom O, og avbildes dermed på samme punkt i skjermen. Nederst er kameraet rotert om R. Det flytter O, slik at A og B nå avbildes på forskjellige steder i skjermen.

7.2 Nullnivå for y (pitch)

For at metoden skal fungere, er det nødvendig å kunne beregne rotasjonsmatrisen som transformerer kameraet fra en posisjon til den neste.

Gitt to par av vinkelverdier (θ_x, θ_y) og ($\emptyset_x$, $\emptyset_y$), bør transformasjonen fra første til annen posisjon beregnes som et produkt av transformasjoner man kjenner. I vårt eksperimentelle oppsett blir dette som tidligere nevnt:

$$(\theta_x, \theta_y) \rightarrow (\theta_x, 0) \rightarrow (\phi_x, 0) \rightarrow (\phi_x, \phi_y)$$

Legg merke til at det her er nødvendig å kjenne de faktiske vinklene i *y*-retning, mens det holder å kjenne forskjellen mellom vinklene i *x*-retning. På apparatet vi bruker er det ingen intrinsik måte å finne nullverdien i *y*-retning på, så denne må kalibreres separat. En

potensiell (ikke utprøvd) metode er å legge inn *y-0*-nivået på linje med de interne kameraparametrene i kalibreringen.

Et annet alternativ er å bruke bare rotasjon i *y*-retning. Dette vil medføre at man kun trenger å bruke vinkel*forskjeller*. På den annen side vil man da begrense det totale vinkelutslaget, noe som medfører mer usikkerhet, og det virker mekanisk sett vanskeligere å produsere ren rotasjon i denne retningen enn i *x*-retningen.

7.3 Avvik mellom optisk akse og oppmålt retning

Når man måler retningen på kameraet, bruker man normalt et eksternt måleinstrument for å måle kameraets retning. I vårt testoppsett var dette et mekanisk vippebord med to sammenkoplede vinkelmålere. I andre sammenhenger kan det være en navigasjonsenhet eller en annen form for vinkelmåler.

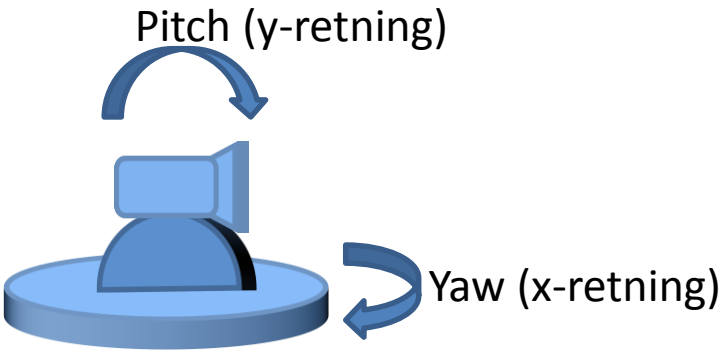
Det som er viktig å tenke over, er hvilke vinkler man faktisk måler. I vårt tilfelle måler vi ikke absolutte vinkler i forhold til nord, kun relative. Vi forsøker imidlertid å måle absolutte vinkler opp fra horisontalplanet (pitch) så godt det lar seg gjøre. Og det vi måler, er da vinklene til vippebordet. Vi forsøker selvsagt å plassere kameraet slik at det er parallelt med bordet, men her kan det være små avvik. Det kan også være avvik mellom kameraets faktiske optiske akse og retningen på selve kamerahuset.

For optimaliseringsformål har disse avvikene trolig lite å si (selv om det KAN gi små utslag). Når man skal bruke kameraet til AR, har det imidlertid stor betydning. Når man bruker AR, bruker man retningen kameraet peker i til å beregne hvor virtuelle objekter skal plasseres i bildet for at de skal stå på "riktig plass" (her kreves det absolutte vinkler i forhold til nord og i forhold til horisontalplanet). Dersom det her er et avvik mellom målt vinkel og kameraets faktiske optiske vinkel, vil det gi en feil.

Skal man bruke kameraet og vinkelmålerne for AR-formål, må man kjenne avviket mellom retningsgivere/vinkelmålere og kameraets optiske akse. En metode for beregning av dette avviket er ikke diskutert i dette dokumentet.

8 Om utstyret/det eksperimentelle oppsett

Kameraet er montert på toppen av to sammenkoblede vinkelmålere (se Figur 8.1). Vinkelmåleren i *y*-retning er fastmontert til vinkelmåleren i *x*-retning, og dette er årsaken til at det er nødvendig å kjenne 0-nivået i *y*-retning. Om *x*-vinkelmåleren var montert på *y*-vinkelmåleren, ville det omvendt være nødvendig å finne 0-nivået i *x*-retning. (I dette tilfellet ville det kanskje være aller mest fornuftig å begrense seg til rotasjon i *x*-retning.)



Figur 8.1 Illustrasjon av det eksperimentelle oppsettet

9 Beskrivelse av kode

Her følger en beskrivelse av hva de ulike funksjonene i Matlab-mappen gjør. Beskrivelsene er delt inn etter hvilken mappe kodefilene finnes i.

9.1 SolveFunctions

centreandundistort: Tar en (2 x *N*)-matrise med målte pikselkoordinater, parameterne c_x, c_y, k_1, k_2 og beregner sentrerte og ikke-forvrengte koordinater. Output er en (2x*N*)-matrise med beregnede koordinater. Hvis antall output-argumenter er 2, vil det andre argumentet være en (7 x 7 x *N*)-matrise `grad` slik at `grad(:,:,k)` er Jacobi-matrisen til transformasjonen

$$\left(x_k, y_k, f, c_x, c_y, k_1, k_2\right) \rightarrow \left(x_{ku}, y_{ku}, f, c_x, c_y, k_1, k_2\right)$$

Her er (x_k, y_k) målte pikselverdier, og (x_{ku}, y_{ku}) er sentrerte og ikke-forvrengte verdier.

I den vanlige modellen for forvrengning vil det her beregnes røtter av en femtegradsligning for å invertere forvrengningen. Vi bruker Matlabs `roots`-rutine, som er forholdsvis treg, og dette kan muligens gjøres raskere med en oppslags-metode, ettersom vi kjenner området roten skal finnes i nokså godt.

Jacobi-matrisen til transformasjonen over beregnes ved å først beregne Jacobi-matrisen til den inverse funksjonen

$$\left(x_{ku}, y_{ku}, f, c_x, c_y, k_1, k_2\right) \rightarrow \left(x_{ku}, y_{ku}, f, c_x, c_y, k_1, k_2\right)$$

og så invertere denne. Jacobi-matrisen for denne transformasjonen fra ikke-forvrengte til forvrengte koordinater er lett å finne, ettersom funksjonen essensielt er gitt ved polynomer. Å så invertere matriser er en enkel operasjon i Matlab.

Hvis en annen modell for forvrengning skal testes, er det her koden for dette endres.

Det er laget en alternativ, enklere algoritme basert på MatLabs "fminsearch"-funksjon, som ikke bruker "centreandundistort". Den forventes å være en anelse mindre presis i sin nåværende form, men det vil bli forbedret i en kommende versjon

costjacobian: Gitt N datapar og verdier for f, c_x, c_y, k_1, k_2 , beregner denne en ($2N$ x 5)-matrise som er Jacobi-matrisen til `vectorcostfun`. Selve beregningene utføres i `centreandundistort` og `rotatepixels`, og settes sammen ved å multiplisere sammen Jacobi-matrisene disse produserer på en passende måte.

rotatepixels: Tar en (2 x N)-matrise med ikke-forvrengte pikselverdier (beregnet i `centreandundistort`, og to (2xN)-matriser med vinkelverdier i radianer. Elementene i søyle N av vinkelmatrisene svarer til vinkelmålingene for bilde 1 og 2. Metoden beregner forventet posisjon (uten forvrengning) for hver pikselverdi, og returnerer en (2 x N)-matrise med disse.

Hvis metoden har to output-argumenter, vil andre argument være en (7 x7 x N)-matrise, hvor den k-te (7x7)-matrisen gir Jacobi-matrisen til funksjonen

$$\left(x_{ku}, y_{ku}, f, c_x, c_y, k_1, k_2\right) \rightarrow \left(x_{ku}^{'}, y_{ku}^{'}, f, c_x, c_y, k_1, k_2\right),$$

hvor $(x_{ku}^{'}, y_{ku}^{'})$ er pikselverdien til punktet etter rotasjon.

De roterte posisjonene finnes ved først å beregne

$$\left(x, y\right) \rightarrow \left(x, y, f\right) \rightarrow R\begin{pmatrix} x \\ y \\ f \end{pmatrix},$$

og så ta projeksjonen ned i bildeplanet:

$$\left(X, Y, Z\right) \rightarrow \left(fX / Z, fY / Z\right).$$

Rotasjonsmatrisen R beregnes ut fra vinkeldataene ved hjelp av `rodriguesmatrix`. Jacobimatrisen finnes ved å beregne matrisene for de enkelte funksjonene, og så multiplisere sammen disse.

setsolveropts: Returnerer en struct med alternativer til bruk i *solvecalibration*. Brukes ved å skrive kode

```
sOpt = setsolveropts('Option1', 'Value1', ...
'Option2', 'Value2', ...);
```

for så å bruke `sOpt` som et argument i `solvecalibration`. Alternativene er (med defaultverdier i kursiv):

display: Skal NL2SOL-solveren gi output under kjøring? Verdier: «*iter*» = kontinuerlig utskrift, «off» = ingen utskrift, «final» = utskrift av endelig resultat.

initCameraParams: En 5-elements radvektor med initielle gjetninger for [f, c_x, c_y, k_1, k_2]. Må stilles inn, har default-verdi [0, 0, 0, 0, 0].

plot: Skal solveren plotte verdier for f, k_1 og k_2 underveis i kjøringen? Verdier: «*off*» = ikke plott, «cont» = kontinuerlig plot, «end» = ett plot til slutt.

timer: «*off*» = ikke ta tiden på kjøringen, «on» = ta tiden.

grad: «*provided*» = bruk egenberegnet gradientfunksjon, «computed» = beregn gradientfunksjonen numerisk.

Innstillingen plot er tenkt brukt til debugging hvis solveren sliter med å finne en god løsning.

solvecalibration: Kjører en least squares-solver på et datasett. Krever som input `sOpt`, som er en struct med alternativer til solveren, pluss 4 (2 x `numPairs`)-matriser med pikseldata bilde 1, pikseldata bilde 2, vinkeldata bilde 1 og vinkeldata bilde 2. Returnerer en vektor med estimerte verdier for kameraparametre.

vectorcostfun: Beregner en vektor av lengde 2 x `numPairs` slik at de første `numPairs`−elementene er feil i *x*-retning for hvert bildepar, mens de neste `numPairs` elementene er feil i *y*-retning. Kalles av `solvecalibration`. Kan også for debuggingsformål kalles direkte fra Matlab-prompten, da med argumenter `cameraParameter` (en 5-vektor), `pData1`, `pData2`, `aData1` og `aData2` (disse er (2x`numPairs`)-matriser.

9.2 Alternative funksjoner

Dette er alternativ til den metoden som blant annet baserer seg på “centreandundistort”.
costfun: Dette er kostfunksjonen (summen av kvadratisk avvik mellom beregnede og målte punkter for alle landemerker) som funksjon av kameraparametre. Metoden forventer at dataene fra “measuredData” (eventuelt etter bruk av “ransac”) finnes i globale variable.

costfunprep: Funksjon som genererer rotasjonsmatrise og gjør dataene fra measuredData tilgjengelige i globale variable.

costfunoptim: Funksjon som bruker “fminsearch” for å optimalisere “costfun”.

9.3 Ransac

calibrateransac: Hovedfunksjonen for RANSAC-kalibrering. Tar 3 argumenter:

initCamParams: Initiell gjetning for kameraparametrene.

measuredData: En (8 x `numPairs`)-matrise hvor rad 1-2 er pikseldata for første bilde i hvert par, rad 3-4 er pikseldata for andre bilde, rad 5-6 er vinkeldata for første bilde, og rad 7-8 er vinkeldata for andre bilde.

distanceThreshold: Hvor mange piksler avvik et punktpar kan ha fra modellen før det regnes som en outlier.

Returnerer som første argument estimerte verdier for kameraparametere, og som (et eventuelt) andre argument, en vektor med indeksene til de beregnede inliers.

distancefunction, dummyfalsefunction, fittingfunction: Brukes internt av `ransac`. Deres roller er forklart i kommentarene i denne filen.

randomsample, ransac: Ferdigskrevne metoder hentet fra [4], hvor `ransac` er en implementasjon av RANSAC, og `randomsample` er kode for å ta en tilfeldig sample fra en mengde datapunkter.

9.4 Datakonstruksjon

Konstruersyntetiskdata: Produserer syntetiske punktkorrespondanser. Tar 2 argumenter, det første er en 5-vektor med parameterne til et kamera som skal simuleres, det andre er en struct med innstillinger som produseres av `setdataopts`. Output er 4 matriser med dimensjon (2 x `numPairs`), hvor matrise 1 er pikseldata i bilde 1, matrise 2 er pikseldata i bilde 2, matrise 3 er vinkeldata i bilde 1, matrise 4 er vinkeldata i bilde 2.

Setdataopts: Produserer en struct med innstillinger som kan tolkes av `konstruersyntetiskdata`. Kalles ved å skrive

```
dOpt = setdataopts('Option1', 'Value1',...
'Option2', 'Value2', ...);
```

Alternativene som kan stilles inn, er beskrevet i kommentarer i koden.

9.5 Hjelpefunksjoner

rodriguesmatrix: Beregner en (3 x 3)-rotasjonsmatrise. Input er en retningsvektor og en vinkel målt i radianer, output er matrisen for rotasjonen med gitt vinkel om aksen gitt av retningsvektoren.

9.6 Pairextraction

loadimages, chooseimagepairs, setangledata: Funksjoner for å laste inn bilder og målt informasjon på en måte som er forståelig for `computefeaturepairs`. Se eksempel på bruk i Seksjon 10.1.

computefeaturepairs: Tar 3 argumenter, en 3D-matrise `images` hvor `images`(:,:,k) er matrisen til bilde *k*, en (`numPairs` x 2)-matrise med hvilke par av bilder som skal sammenlignes, og en (2 x `numPhotos`)-matrise med målte vinkelverdier. Output er en matrise med 8 rader, hvor hver søyle i matrisen tilsvarer en forekomst av samme landemerke i to bilder. I en slik søyle er rad 1-2 og 3-4 pikselverdiene til landemerket i hhv. bilde 1 og 2, mens rad 5-6 og 7-8 er vinkelverdiene til hhv. bilde 1 og 2. Funksjonen tar også options for innstilling av sensitivitet i SURF-deteksjon og matching, se Seksjon 10.2.

10 Eksempelkjøringer

Her er noen eksempler på bruk av koden.

10.1 Datalasting

Vi antar at alle bilde-filene ligger et sted Matlab kan finne dem, enten i arbeidsmappen eller i en mappe i PATH-variablen til Matlab. Vi får lastet bildene til variabelen `images` ved å skrive

```
images = loadimages();
```

Her kommer det opp en dialogboks, hvor vi skal fylle inn antall bilder, filnavn, og filutvidelse. Det er antatt at bildefilene har navn på formen

```
filnavn01.ext, filnavn02.ext, ..., filnavnN.ext
```

Vi kan for eksempel bruke de 18 bildene som er tatt fra biblioteksvinduet og som har filnavn `img1708-01.tif, ..., img1708-18.tif`, i så fall skriver vi inn «18», «img1708-» og «tif».

Neste punkt er å legge inn vinkelverdiene; dette gjøres ved

```
angles = setangledata();
```

Her kommer det opp en dialogboks for input av *x*-vinkelverdier i steps, *y*-vinkelverdier i steps og antall steps i en omdreining (65536 i vårt testoppsett). Vi har 18 bilder, så vi skriver inn 18 *x*-verdier og 18 *y*-verdier (her bør *y*-verdiene være slik at 0-nivået legges

inn som 0). Nå vil angles være en (2 x 18)-matrise med vinklene vi har målt, konvertert til radianer.

Denne metoden å angi vinkler på kan være litt knotete. Det er mulig, og ofte ønskelig, å lage "angles"-variabelen på en annen måte, for eksempel ved å lese en tekstfil der vinklene finnes, eller å lage et lite script der man angir vinklene direkte.

Etter dette må vi oppgi hvilke par av bilder vi ønsker å sammenligne, det vil si for hvilke par av bilder vi skal lete etter matchende landemerker. Dette gjøres ved

```
pairs = chooseimagepairs(18);
```

Merk at vi må oppgi det totale antallet bilder som argument. Det kommer opp en dialogboks med en liste hvor vi kan velge ut hvilke bildepar som skal sammenlignes. Som regel vil man kanskje velge alle bildepar, men av effektivitetshensyn eller for testing kan det være greit å velge ut bare noen av bildene. Resultatet til slutt er at `pairs` er en (N x 2)-matrise, hvor N er antall bildepar som skal sammenlignes, og hver rad angir et bildepar som skal sammenlignes, det vil si at en rad med verdier `[4 15]` angir at vi skal lete etter matchende punkter i bilde 4 og 15.

10.2 Landemerkeekstraksjon

Landemerkeekstraksjon gjøres nå ved

```
measuredData = computefeaturepairs(images, pairs, ...
angles, 'StrengthThreshold', 3000, 'MatchThreshold', 0.02);
```

Verdiene for `StrengthThreshold` og `MatchThreshold` kan forandres på. En større verdi for `StrengthThreshold` medfører at færre SURF-punkter finnes i hvert bilde, mens en lavere verdi for `MatchThreshold` medfører at færre identifiseringer mellom SURFpunkter i ulike bilder tas med. Hvis et bildesett produserer veldig mange falske punktpar, kan det for eksempel hjelpe å stille `MatchThreshold` lavere eller `StrengthThreshold` høyere. Hvis et bildesett produserer for få punktpar, kan parametrene stilles inn andre veien for å få ut flere matchede punkter.

Variabelen `measuredData` er nå en matrise med 8 rader og *N* søyler, hvor *N* er antall matchede punkter. I hver søyle er rad 1-2 pikselverdien i bilde 1, rad 3-4 er pikselverdien i bilde 2, rad 5-6 er vinkelverdien til bilde 1, og rad 7-8 er vinkelverdien til bilde 2.

10.3 Estimering av kameraparametre

Vi har nå forberedt dataene nok til å prøve å finne parametre. Vi skriver inn

```
[X,I] = calibrateransac([5900, 2128, 1416, 0],...
measuredData, 10);
```

Her er første argument en initiell gjetning for kameraparametrene, altså f = 5900 piksler, (c_x, c_y) = (2128, 1416) og k_1 = 0. (Koden er for øyeblikket slik at den forventer k_2 = 0, så denne parameteren tas ikke med). Resultatet etter endt kjøring er at `X` er en 4-vektor med beregnede kameraparametre, mens `I` er en vektor med indeksene til alle inliers.

Vi anbefaler ikke å bruke `X` direkte, og en mer nøyaktig beregning fås ved å kjøre en least squares-rutine på datasettet som består av inliers. Først må vi klargjøre noen alternativer til solveren:

```
sOpt = setsolveroptions('initCameraParams', X);
```

Her setter vi verdiene i `X` som initiell gjetning, og lar alle andre alternativer stå på defaultverdier. Vi kjører nå least squares-rutinen med

```
A = solvecalibration(sOpt, measuredData(1:2,I),...
measuredData(3:4,I), measuredData(5:6,I),...
measuredData(7:8,I));
```

Det endelige estimatet for kameraparametre finnes nå i A. Et alternativ til bruk av "solvecalibration" er å kjøre "costfunoptim:

```
costfunoptim();
```

Nå finnes estimatet for kameraparameter i en parameter "x". Denne bruker hele "measuredData", så dersom man bare vil bruke inliers, må man først skrive

```
measuredData=measuredData(:,I);
```

10.4 Konstruksjon av syntetiske data

Dette avsnittet beskriver hvordan man kan generere syntetiske data fra et tenkt, "perfekt", kamera. Dette er gunstig for å teste om metoden gir riktig svar.

For å konstruere syntetiske data, setter vi først opp alternativene for dataene

```
dOpts = setdataopts();
```

Her kan det eventuelt gjøres ulike valg i argumentene til `setdataopts`. Vi får så konstruert syntetiske data ved å kjøre

```
[pixelData1, pixelData2, angleData1, angleData2] = ...
konstruersyntetiskdata([5000, 2128, 1416, 0, 0], dOpts);
```

Her simulerer vi et kamera med parametre (5000, 2128, 1416, 0, 0). Verdiene for pikseldata og vinkeldata kan nå plugges inn i `solvecalibration`, og hvis alt er vel, skal metoden raskt regne seg fram til parametrene vi har angitt. I våre tester har dette så langt gitt riktige resultater.

11 Spørsmål/mulige forandringer

11.1 Kvantitativ test av metoden

Det er ikke gjort skikkelige tester på hvor nøyaktig metoden er. Ett mål på dette kan være å finne standardavviket i estimert verdi for ulike uavhengige datasett. Med andre ord: Er metoden enig med seg selv? (Denne typen test er også gjort av Stein i [6].)

Selv om metoden er enig med seg selv, kan f.eks. uren rotasjon gi systematiske feil i parameterne metoden finner. Det hadde vært interessant å finne ut hvor godt samsvar det er mellom vår metodes svar og en «fasit» som beregnes av en annen algoritme, f.eks. Camera Calibration Toolbox i Matlab (http://www.vision.caltech.edu/bouguetj/calib_doc/) eller PhotoModelers kalibreringssystem (som Trym Haavardsholm har).

Generelt er det kanskje også interessant å se hvor mange bilder som behøves for å få en rimelig god verdi, og hva slags vinkelutslag som er gunstigst for å få nøyaktige verdier. Metoden vil kunne beregne kameraparametre ut fra bare 2 bilder, men i hvilken grad dette øker usikkerheten i de beregnede parametrene er uvisst (den vil opplagt øke noe).

11.2 Beregning av 0-nivå for vinkelmåling

Om det gjøres en dreining i *x*-retning, er det nødvendig å kjenne hvilket vinkelutslag i y-retning som representerer 0-nivået. Det er antakelig mulig å finne dette 0-nivået ved å legge det inn som en ekstra parameter i beregningen på linje med kameraparameterne, men dette er ikke utprøvd.

På ett datasett har vi, ved å prøve oss fram. funnet at en mulig verdi for 0-nivået i *y*-retning i vårt opprinnelige oppsett er på step 61167. Dette tilsvarer verdien som gjør at den beste modellen har minimalt med feil. (Merk at koordinatsystemet til vinkelmålerne kan endres av brukeren, så dette tallet kan endre seg om noen stiller inn disse.)

11.3 Andre landemerkedetektorer

Sannsynligvis er SURF-metoden mer generell enn det som behøves for vårt formål, da den er ment å kunne gjenkjenne landemerker etter både rotasjon og forstørrelse. I vårt

oppsett vil landemerker stort sett oppleve kun translasjon, og dette kan utnyttes for å få mer effektiv landemerke-deteksjon og -matching.

11.4 Kontroll på posisjon av landemerker i bilde

Slik metoden er, vektes alle observasjoner likt, og det tas spesielt ikke hensyn til om punktene er klumpet sammen i en region av bildet. Det kan undersøkes om dette har betydning for parameterestimatet.

I så fall kan dette kanskje kompenseres for ved å vekte observasjoner etter hvor landemerket er observert, eller ved å sørge for at man velger landemerker spredt jevnt ut over bildene.

11.5 Avvik mellom målt retning og retning på optisk akse

Som nevnt i kapittel 7.3, kan det være et avvik mellom retningen man måler med en retningsgiver og den faktiske retningen til den optiske aksen. Dette har trolig lite å si for selve kamerakalibreringen, men er av stor betydning dersom man skal bruke kameraet til AR-formål.

En metode for å finne dette avviket, slik at det kan tas hensyn til når man legger på AR-symboler, vil trolig være nødvendig for å kunne bruke et kamera til AR-formål.

11.6 Kameramodell

Metoden som er utviklet, antar en bestemt kameramodell, som vanligvis gjelder for "normale" kameraer. For kameraer som fungerer helt annerledes, for eksempel fisheye-kameraer, vil metoden ikke kunne anvendes. For slike kameraer må en egen modell utvikles, selv om store deler av koden vi har utviklet trolig vil kunne gjenbrukes.

11.7 Ting å gjøre i koden

Koden er nokså upolert, og om den skal brukes videre, er det et par ting det er naturlig å endre på.

- Tillate å låse parametre ((c_x, c_y) kan settes til midt i bildet, k_2 kan settes til 0). Slik koden er nå, må dette gjøres ved å redigere *m*-filene.
- I centreandundistort brukes Matlabs roots-algoritme til å løse en femtegradsligning. Dette er tidsmessig ineffektivt, så om ytelsen skal forbedres, kan det være aktuelt å bytte ut denne kodebiten med en lookup-metode eller noe slikt, ettersom man kjenner verdiene røttene skal befinne seg innenfor ganske godt.
  Denne rutinen har for øvrig vist seg å ikke alltid fungere. En enklere algoritme er derfor også implementert. Denne rutinen benytter MatLabs "fminsearch", som finner minimum til en funksjon. Funksjonen som minimeres, er avstanden mellom målt posisjon til en "feature" i bilde 2 og beregnet posisjon for samme feature, basert på

dennes posisjon, i bilde 1. Dette er litt mindre presist, men raskere, og det virker stort sett hver gang.

12 Oppsummering

Med bakgrunn i et behov som stammer fra arbeidet med Augmented Reality, har vi laget en algoritme med tilhørende programmer for kamerakalibrering. Metoden forutsetter at man bruker et "normalt" kamera (ikke for eksempel et "fisheye-kamera").

Koden er skrevet i MatLab, og krever enkelte toolbokser for å fungere.

Metoden er laget slik at man ikke trenger å ta bilde av bestemte mønstre, men man må kjenne retningen kameraet peker i for hvert bilde.

Metoden går i korthet ut på at man tar flere bilder av omtrent den samme scenen, der kameraet står i samme posisjon, men roteres mellom hvert bilde. Så finner vi landemerker i alle bildene, og matcher disse mot hverandre. Vi bestemmer så det settet med kameraparametere som gir minst totalt avvik mellom beregnede og målte posisjoner for par av landemerker.

Andre metoder for kamerakalibrering eksisterer. Flere av disse baserer seg på at man tar bilde av et kjent mønster, som regel et rutenett ("sjakkbrett"), fra flere posisjoner. Hvilken metode som er best egnet, vil avhenge av den aktuelle problemstillingen.

13 Referanser

[1] OPTI toolbox. http://www.i2c2.aut.ac.nz/Wiki/OPTI/index.php/Main/HomePage.

[2] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-Up Robust Features (SURF). *Computer Vision and Image Understanding*, 110(3):346–359, June 2008.

[3] John E. Dennis Jr., David M. Gay, and Roy E. Welsch. Algorithm 573: NL2SOL – An adaptive nonlinear Least-Squares algorithm [E4]. *ACM Trans. Math. Softw.*, 7(3):369–383, September 1981.

[4] P. D. Kovesi. MATLAB and Octave functions for computer vision and image processing. Centre for Exploration Targeting, School of Earth and Environment, The University of Western Australia. Available from: <http://www.csse.uwa.edu.au/~pk/research/matlabfns/>.

[5] G.P. Stein. Internal camera calibration using rotation and geometric shapes. *Masteroppgave MIT*, 94:21167, February 1993. Finnes på http://hdl.handle.net/1721.1/7052.

[6] G.P. Stein. Accurate internal camera calibration using rotation, with analysis of sources of error. In *Fifth International Conference on Computer Vision, 1995. Proceedings*, pages 230 –236, June 1995.

[7] R. Hartley and A. Zisserman. *Multiple View Geometry in computer vision*, Cambridge University press, ISBN 978-0-521-54051-3.