Robotkinematik

Skrevet af

Jesper Roager

Afleveret den

20. december 2022

Vejledt af

Søren Præstegaard

Marit Hvalsøe Schou

Resumé

Robotkinematik opdeles i to dele fremadrettet kinematik og invers kinematik. For at regne med fremadrettet kinematik bruges der matricer til at lave homogene transformationer. Homogene transformationer opbevares i en 4 gange 4 matrix, og den indeholder både rotation og translation. For at lave disse transformationsmatricer bruges Denavit-Hartenberg parametrene. Disse beskriver hvert led på robotten med fire værdier. Med seks af disse matricer kan en Dobot Magician Lite's fremadrettede kinematik beskrives. Matricerne ganges samme for at få en matrix, der beskriver positionen af dobottens værktøj ud fra vinklerne på motorerne. Efter dette udledes der en invers kinematik-model ved at betragte de to midterste led på dobotten som en plan robot og de andre to led betragtes gennem et andet plan. På denne måde bliver problemet til to simple analytiske plangeometriproblemer. Dette resulterer i 9 ligninger, som implementeres i et Pythonprogram, så det er muligt at styre dobotten ved at give den et koordinatsæt eller en matrix med position og rotation. For at Pythonprogrammet kan regne med matricer bruges et Pythonbibliotek, der laver en matrix-klasse, som kan lave de fleste standardoperationer med matricer. Både biblioteket og programmet til styring af dobotten analyseres for at forstå deres funktion. Til sidst diskuteres vigtigheden af matematik i programmering af robotter. Hvor meget fokus der er på matematikken skal afhænge af hvad formålet med programmeringens projektet er, så nogle gange er matematikken nødvendig og andre gange kan den med fordel skjules i biblioteker.

Indholdsfortegnelse

Indledning

Robotter er over det hele i verden, og de kommer nærmere på os, så vi møder dem oftere og oftere i vores hverdag. Mest af alt er de uundværlige i produktionen af alle de ting vi bruger til daglig; fra mobiltelefoner til sæbe og alt derimellem. Disse tusindvis af robotter laver atter tusindvis af bevægelser dagen lang, og det er vigtigt, at de udfører bevægelserne korrekt. For at robotter kan lave bevægelserne rigtigt, skal de vide, hvordan leddene skal stå for at nå en given position. For at gøre det lettere at arbejde med position og rotation, bruges der matricer. Hvordan bruges matricer til at udregne position af leddene i dobotten for at nå en given position inden for robottens rækkevidde? Dette problem kan inddeles i to problemer. Det ene et fremadrettet kinematikproblem. Problemet består af at kunne regne, ud hvor robottens værktøj, ender når vinklen på leddene er kendt, dette kaldes også forward kinematik. Det andet problem hedder invers kinematik. Her kendes der en ønsket position og rotation af robottens værktøj, og så skal vinklerne, der opnår den ønskede position findes. For at forstå denne proces med fremadrettet kinematik og invers kinematik, starter opgaven med en redegørelse for matricer og homogene transformationer med matricer. Så beskrives den fremadrettet kinematik for en Dobot Magician Lite og med hjælp fra fremadrettede kinematik, udvikles der en invers kinematik-model for dobotten. Derefter analyseres et Pythonbibliotek til matrixregning, og et program der styrer dobotten med den fremadrettede- og inverse- kinematikmodel. Til sidst diskuteres hvor stor nødvendigheden af matematisk viden er, når der skal programmeres robotter.

Redegørelse for lineære transformationer med matricer

Matricer

En vektor er en liste af tal, som for det meste repræsenterer et punkt i rummet, for eksempel:

$$\vec{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

$\vec{v}$ er en vektor i rummet. Den har tre dimensioner, og mængden af denne vektor er $\mathbb{R}^3$, og hvis vektoren havde n tal ville den tilhøre denne mængde $\mathbb{R}^n$.

En matrix er så en samling af vektorer, som står ved siden af hinanden. Den består af en mængde tal, der er organiseret i rækker og kolonner.

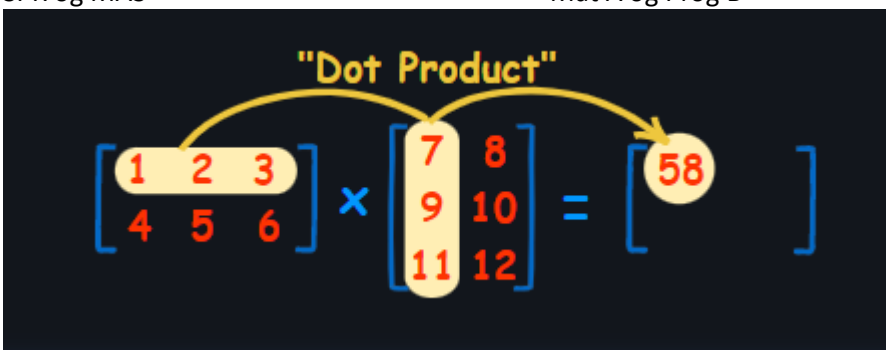
$$A = [\vec{v}_1 \quad \vec{v}_2 \quad \cdots \quad \vec{v}_m] = \left[\begin{pmatrix} v_{1,1} \\ v_{2,1} \\ \vdots \\ v_{n,1} \end{pmatrix} \quad \begin{pmatrix} v_{1,2} \\ v_{2,2} \\ \vdots \\ v_{n,2} \end{pmatrix} \quad \cdots \quad \begin{pmatrix} v_{1,m} \\ v_{2,m} \\ \vdots \\ v_{n,m} \end{pmatrix}\right] = \begin{bmatrix} v_{1,1} & v_{1,2} & \cdots & v_{1,m} \\ v_{2,1} & v_{2,2} & \cdots & v_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ v_{n,1} & v_{n,2} & \cdots & v_{n,m} \end{bmatrix}$$

Mængden af n x m matricer betegnes med $\mathbb{R}^{n\times m}$[1]

Matrixmultiplikation

Der defineres, hvordan man ganger to matricer sammen. For at kunne gange to matricer sammen, skal den første matrix have det samme antal kolonner, som den anden matrix har rækker, ellers er multiplikation ikke defineret. For dette gør det muligt at tage prikproduktet mellem hver række i den første matrix og hver kolonne i den anden matrix, og så sættes det givne produkt ind på rækkenummeret fra den første matrix og kolonnenummeret fra den andet matrix. På Figur 1 ses der et eksempel, hvor det er illustreret, hvilke vektorer der skal prikkes.

[1] (Undervisningsministeriet, 2014)



Figur 1 Illustration af matrixmultiplikation, billede fra (Pierce, 25)

Dette betyder, at hvis $A \in \mathbb{R}^{n\times m}$ og $B \in \mathbb{R}^{m\times k}$ og C er givet ved $C = A \cdot B$, så er $C \in \mathbb{R}^{n\times k}$.

Det betyder, at den resulterende matrix ved multiplikation mellem to matricer giver en matrix med samme antal rækker som den første matrix og det samme antal kolonner som den andet matrix.

Her er et eksempel på matrixmultiplikation:

$$A = \begin{bmatrix} 3 & -2 \\ 2 & 4 \\ 1 & -3 \end{bmatrix}$$
$$B = \begin{bmatrix} -2 & 5 & 3 \\ 4 & 1 & 6 \end{bmatrix}$$
$$C = A \cdot B = \begin{bmatrix} 3 & -2 \\ 2 & 4 \\ 1 & -3 \end{bmatrix} \cdot \begin{bmatrix} -2 & 5 & 3 \\ 4 & 1 & 6 \end{bmatrix}$$
$$C = \begin{bmatrix} \binom{3}{-2} \cdot \binom{-2}{4} & \binom{3}{-2} \cdot \binom{5}{1} & \binom{3}{-2} \cdot \binom{3}{6} \\ \binom{2}{4} \cdot \binom{-2}{4} & \binom{2}{4} \cdot \binom{5}{1} & \binom{2}{4} \cdot \binom{3}{6} \\ \binom{1}{-3} \cdot \binom{-2}{4} & \binom{2}{4} \cdot \binom{5}{1} & \binom{2}{4} \cdot \binom{3}{6} \end{bmatrix}$$
$$C = \begin{bmatrix} 3\cdot(-2) - 2\cdot 4 & 3\cdot 5 - 2\cdot 1 & 3\cdot 3 - 2\cdot 6 \\ 2\cdot(-2) + 4\cdot 4 & 2\cdot 5 + 4\cdot 1 & 2\cdot 3 + 4\cdot 6 \\ 1\cdot(-2) - 3\cdot 4 & 1\cdot 5 - 3\cdot 1 & 1\cdot 3 - 3\cdot 6 \end{bmatrix}$$
$$C = \begin{bmatrix} -14 & 13 & 0 \\ 12 & 14 & 30 \\ -14 & 2 & -15 \end{bmatrix}$$

Identitetsmatricer

Der findes identitetsmatricer, og de har den samme effekt, som når man ganger med et ettal i de reelle tal. Det vil sige, at når man ganger med en identitetsmatrix, giver det den oprindelige matrix tilbage. Identitetsmatricer er kvadratiske, og har ettaller på hoveddiagonalen og resten er nul. Det kaldes I. Et eksempel på en identitetsmatrix:

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

De kan komme i alle størrelser, så længe de er kvadratiske.

At gange med en identitets matrix giver altid den oprindelige matrix, lad $A \in \mathbb{R}^{n\times m}$ og I være et identitet matrix i den rigtige størrelse

$$A \cdot I = A$$
$$I \cdot A = A$$

Her ses at når der ganges med en identitets matrix for man de oprindelige matricer bagefter.

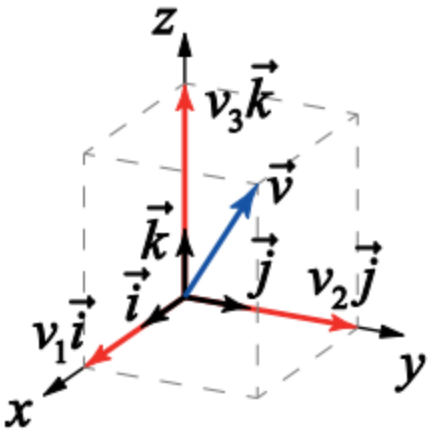
Dette kan også vises med et taleksempel.

$$A = \begin{bmatrix} 3 & -2 \\ 2 & 4 \\ 1 & -3 \end{bmatrix}$$
$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$
$$A \cdot I = \begin{bmatrix} 3 & -2 \\ 2 & 4 \\ 1 & -3 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 3 \cdot 1 - 2 \cdot 0 & 3 \cdot 0 - 2 \cdot 1 \\ 2 \cdot 1 + 4 \cdot 0 & 2 \cdot 0 + 4 \cdot 1 \\ 1 \cdot 1 - 3 \cdot 0 & 1 \cdot 0 - 3 \cdot 1 \end{bmatrix} = \begin{bmatrix} 3 & -2 \\ 2 & 4 \\ 1 & -3 \end{bmatrix}$$

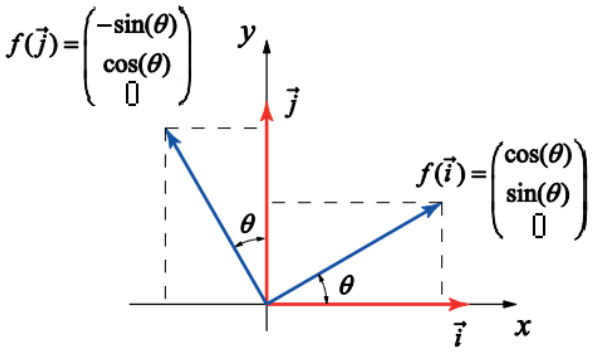
Det er vigtigt, at identitetsmatricen har den rigtige størrelse, så det er muligt at finde matrix-matrixproduktet.

Transformationer

Matricer kan bruges til at rykke rundt på punkter, det kaldes en transformation. Det gøres ved, at en matrix beskriver de nye basisvektorer ud fra de gamle basisvektorer. Enhver vektor kan beskrives ud fra basisvektorerne, de har en længde på 1 og står ortogonalt på hinanden. I rummet kaldes de $\vec{\imath}, \vec{\jmath}$ og $\vec{k}$. De har som standard størrelserne $\vec{\imath} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \vec{\jmath} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$ og $\vec{k} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$. Og en vektor i rummet $\vec{v}$ kan beskrives på denne måde $\vec{v} = v_1 \cdot \vec{\imath} + v_2 \cdot \vec{\jmath} + v_3 \cdot \vec{k}$ ud fra $\vec{\imath}, \vec{\jmath}$ og $\vec{k}$. Dette kan ses på Figur 2.



Figur 2 Illustration af basisvektor figur fra (Undervisningsministeriet, 2014)



Figur 3 Illustration af rotation om z-aksen. Figur fra (Undervisningsministeriet, 2014)

Så hvis man vil transformere et punkt til et andet punkt, kan man nøjes med ændre beskrivelsen af basisvektorerne. De kan både ændre retning og skaleres. På denne måde kan man lave en lineær afbildning af et punkt ved hjælp af en matrix. Så i rummet ser en afbildningsmatrix sådan ud:

$$A = [\overrightarrow{v_i} \quad \overrightarrow{v_j} \quad \overrightarrow{v_k}] = [f(\vec{\imath}) \quad f(\vec{\jmath}) \quad f(\vec{k})]$$

hvor f er en lineær funktion.

Rotation

På denne måde kan en rotation beskrives ved hjælp af en matrix, for eksempel her en rotation om z-aksen.

$$A = [f(\vec{\imath}) \quad f(\vec{\jmath}) \quad f(\vec{k})] = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Det ses, at $\vec{k}$ ikke ændrer sig, da der roteres om z-aksen. Se **Fejl! Henvisningskilde ikke fundet.**. På denne måde kan man beskrive rotation om enhver akse. Senere i opgaven skal der bruges en matrix, som først roterer om x-aksen, så y-aksen og til sidst z-aksen. Denne findes ved at gange hver enkelt af disse rotationsmatricer sammen.

$$R_z(\alpha) = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
$$R_y(\beta) = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{bmatrix}$$
$$R_x(\gamma) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\gamma) & -\sin(\gamma) \\ 0 & \sin(\gamma) & \cos(\gamma) \end{bmatrix}$$

$$
{}^{A}_{B}R_{xyz}(\gamma,\beta,\alpha) = R_z(\alpha)\cdot R_y(\beta)\cdot R_x(\gamma)
$$
$$
= \begin{bmatrix} cos(\alpha)\cos(\beta) & cos(\alpha)sin(\beta)sin(\gamma) - sin(\alpha)\cos(\gamma) & cos(\alpha)\sin(\beta)\cos(\gamma) + \sin(\alpha)\sin(\gamma) \\ \sin(\alpha)\cos(\beta) & sin(\alpha)sin(\beta)sin(\gamma) + cos(\alpha)cos(\gamma) & sin(\alpha)sin(\beta)cos(\gamma) - cos(\alpha)sin(\gamma) \\ -sin(\beta) & cos(\beta)sin(\gamma) & cos(\beta)cos(\gamma) \end{bmatrix}
$$ [2]

Translation

For at kunne lave translationer er det ikke nødvendigt med matricer, da en translation er at flytte en vektor med en anden vektor, så derfor kan to vektorer lægges sammen for at lave en translation. Så hvis punkt P_1 skal flyttes med en vektor $\vec{v}$ lægges disse sammen for at få det nye punkt P_2.

$$P_2 = P_1 + \vec{v}$$

Homogene transformationer

For at forsimple operationer, som både har en rotation og en translation, kan disse sættes sammen til en enkel 4×4 matrix. På denne måde:

$$\left[\begin{array}{ccc|c} & R & & P \\ 0 & 0 & 0 & 1 \end{array}\right]$$

hvor R, den øverste venstre del, er en 3×3 rotationsmatrix, og kolonne 4 er en translation givet ved P. Række 4 i denne matrix er $[0 \quad 0 \quad 0 \quad 1]$ for at gøre det kvadratisk og det er taget fra en 4×4 identitetsmatrix så de påvirker ikke resultatet.

For at kunne gange punkter med denne matrix, tilføjes der et ettal i slutningen, så de har en størrelse, der passer med at gange med et 4×4 matrix, og dette ettal ændrer ikke på resultatet, så det skal der ses bort fra, når resultatet aflæses. Først sker en rotation og så en translation i en og samme operation.

For at forstå det ses først, hvordan en translation fungerer. En transformationsmatrix, der kun laver en translation, ser sådan her ud:

$$D_Q(q) = \begin{bmatrix} 1 & 0 & 0 & q_x \\ 0 & 1 & 0 & q_y \\ 0 & 0 & 1 & q_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Og en transformation med $P = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}$

$$\begin{bmatrix} 1 & 0 & 0 & q_x \\ 0 & 1 & 0 & q_y \\ 0 & 0 & 1 & q_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1\cdot P_x + 0\cdot P_y + 0\cdot P_z + q_x\cdot 1 \\ 0\cdot P_x + 1\cdot P_y + 0\cdot P_z + q_y\cdot 1 \\ 0\cdot P_x + 0\cdot P_y + 1\cdot P_z + q_z\cdot 1 \\ 0\cdot P_x + 0\cdot P_y + 0\cdot P_z + 1\cdot 1 \end{bmatrix} = \begin{bmatrix} P_x + q_x \\ P_z + q_z \\ P_z + q_z \\ 1 \end{bmatrix}$$

Når der udelukkende ses på en translation, viser det sig, at denne måde at gange dem sammen på, giver det samme som at lægge to vektorer sammen. Da der i resultat ses bort fra det nederste ettal.

På samme måde kan rotation undersøges. En transformationsmatrix, der kun laver en rotation om z-aksen, ser sådan her ud:

$$R_z(\theta) = \begin{bmatrix} cos\ \theta & -\sin\theta & 0 & 0 \\ \sin\theta & cos\ \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

[2] (Craig, 2005)

Og en transformation med $P = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}$

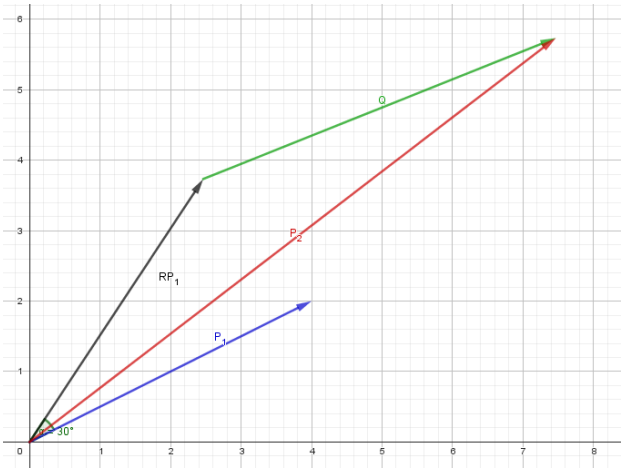
$$\begin{bmatrix} cos\,\theta & -\sin\theta & 0 & 0 \\ \sin\theta & cos\,\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta \cdot P_x - \sin\theta \cdot P_y + 0 \cdot P_z + 0 \cdot 1 \\ \sin\theta \cdot P_x + \cos\theta \cdot P_y + 0 \cdot P_z + 0 \cdot 1 \\ 0 \cdot P_x + 0 \cdot P_y + 1 \cdot P_z + 0 \cdot 1 \\ 0 \cdot P_x + 0 \cdot P_y + 0 \cdot P_z + 1 \cdot 1 \end{bmatrix} = \begin{bmatrix} \cos\theta \cdot P_x - \sin\theta \cdot P_y \\ \sin\theta \cdot P_x + \cos\theta \cdot P_y \\ P_z \\ 1 \end{bmatrix}$$

Dette kan sammenlignes med, hvis der ganges med den rotationsmatrix, der roterer om z-aksen. Som blev fundet i afsnittet Rotation.

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} = \begin{bmatrix} P_x \cdot \cos(\theta) - P_y \cdot \sin(\theta) \\ P_x \cdot \sin(\theta) + P_y \cdot \cos(\theta) \\ P_z \end{bmatrix} = \begin{bmatrix} -P_y \cdot \sin(\theta) + P_x \cdot \cos(\theta) \\ P_x \cdot \sin(\theta) + P_y \cdot \cos(\theta) \\ P_z \end{bmatrix}$$

Så ses det, at det giver det samme. Derfor kan en homogen transformation beskrives på den ovenstående måde.

Eksempel på homogen transformationsmatrix



Figur 4 Eksempel homogentransformation

Her er et eksempel på en transformation med en transformationsmatrix

$$T = \begin{bmatrix} 0{,}866 & -0{,}500 & 0 & 5 \\ 0{,}500 & 0{,}866 & 0 & 2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Denne transformationsmatrix forskyder et punkt med 5 på x-aksen, 2 på y-aksen, og den forskydes ikke på z-aksen, og den roterer 30 grader om z-aksen. Punktet, der transformeres, er $P_1 = \begin{bmatrix} 4 \\ 2 \\ 0 \\ 1 \end{bmatrix}$. Det sidste ettal tilføjes for, at det får den rigtige størrelse, så det er muligt at gange det med transformationsmatricen. Det resulterer i punktet P_2.

$$P_2 = T \cdot P_1 = \begin{bmatrix} 0{,}866 & -0{,}500 & 0 & 5 \\ 0{,}500 & 0{,}866 & 0 & 2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 4 \\ 2 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0{,}866 \cdot 4 - 0{,}5 \cdot 2 + 0 \cdot 0 + 1 \cdot 5 \\ 0{,}5 \cdot 4 + 0{,}866 \cdot 2 + 0 \cdot 0 + 1 \cdot 2 \\ 0 \cdot 4 + 0 \cdot 2 + 1 \cdot 0 + 0 \cdot 1 \\ 0 \cdot 4 + 0 \cdot 2 + 0 \cdot 0 + 1 \cdot 1 \end{bmatrix} = \begin{bmatrix} 7{,}464 \\ 5{,}732 \\ 0 \\ 1 \end{bmatrix}$$

Dette eksempel kan ses på Figur 4, P_1 er blå, P_1 roteret kaldes RP_1 og er sort, forskydningsvektoren kaldes Q og er grøn, og P_2 er rød.

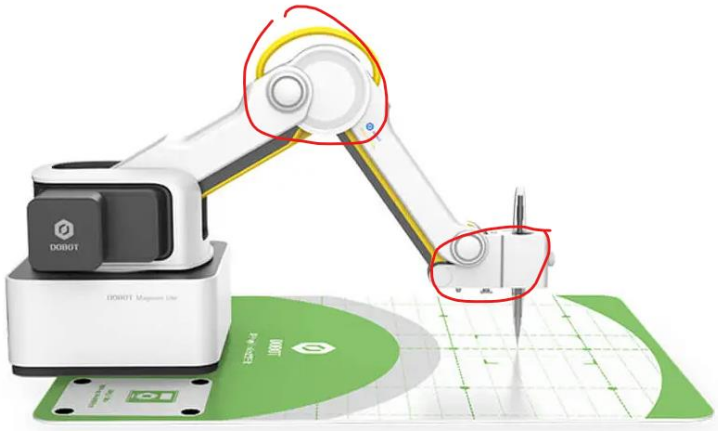
Når der arbejdes med transformationer på robotter, placeres der frames, et andet ord for koordinatsystem, på alle omdrejningspunkter på robotten. Hver enkelt frames position beskrives ud fra den forrige frame ved hjælp af en transformationsmatrix. Så frame B beskrevet ud fra frame A skrives med transformationsmatricen ${}^{A}_{B}T$. Det, at A står øverst, viser, at synspunktet flyttes fra koordinatsystem A til B, der står nederst. På denne måde kan man beskrive værktøjets position ud fra en lang liste af transformationsmatricer.[3]

[3] (Craig, 2005)

Beskrivelse af den kinematiske struktur i en Dobot

Fremadrettet kinematik

Når den kinematiske struktur af en robot skal beskrives, skal leddenes indbyrdes position beskrives. En Dobot, som ses på Figur 6, har 4 led, der er motoriserede, og 2 der er mekanisk styrede. De mekaniske led sørger for, at nogle dele af robotarmen er vandret. På Figur 5**Fejl! Henvisningskilde ikke fundet.** ses de to led der altid er vandret, mærkeret med en rød cirkel. For at beskrive robotten bruges Denavit-Hartenberg notation, som beskriver to frames' indbyrdes position og rotation. Så der startes med en stationær frame 0. Denne frame står stille i bunden af robotten, og den flytter sig aldrig. Efter frame 0 placeres der frames i hvert led, de navngives ved at tælle op fra 0. I starten og slutningen kan det være nødvendigt med en forskydning, fordi leddene ikke starter præcis i bunden af robotten og/eller at værktøjet i robotarmens endepunkt ikke er lige i slutningen af det sidste led. Denavit-Hartenberg notation beskriver hvert led med 4 parametre $a_{i-1}\ \alpha_{i-1}\ d_i\ \theta_i$. Først roteter man med α_{i-1} grader rundt om den oprindelige x-akse, så forskydes med a_{i-1} langs den samme x-akse, derefter roteres denne nye frame med θ_i om den nye z-akse, derefter forskydes med d_i hen langs z-aksen. Normalt bruges disse to sidste parametre til ledvariable, det vil sige, at det er dem, der ændres på, når robotten ændrer position, hvis leddet er et rotationsled, bruges ledvariablen θ_i til at beksrive rotationen, og d_i sættes til nul og bliver ikke brugt. Det er omvendt ved led, som bevæger sig med en lineær forskydning, men dem er der ikke nogen af på dobotten. På denne måde kan to frames' indbyrdes placering beskrives.[4]



Figur 5 Dobot med mekaniske låste led mærket. Billede fra (STEM EDUCATION WORKS, n.d.)

Figur 6 Dobot Magician Lite. Bilede fra (STEM EDUCATION WORKS, n.d.)

Denne proces består af en rotation om den oprindelige x-akse så en translation langs x-aksen og så ny rotation om den nye z-akse og til sidst en translation langs den nye z-akse, dette kan beskrives med disse matricer:

$$T = R_x(\alpha_{i-1})D_x(a_{i-1})R_z(\theta_i)D_z(d_i)$$

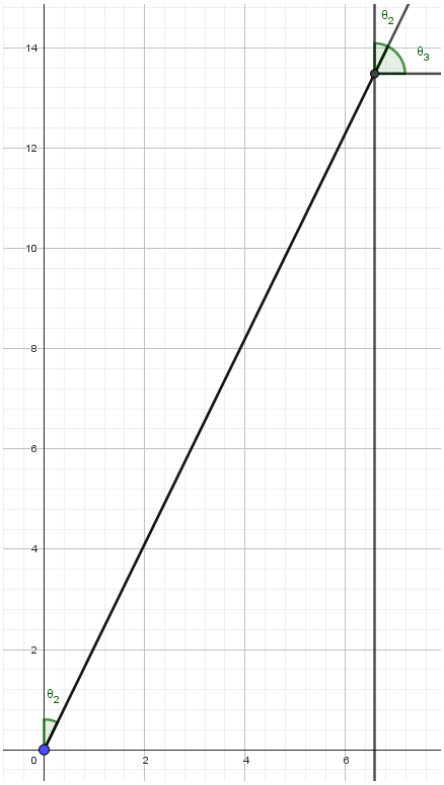
Når disse matricer multipliceres, får vi en transformation fra den tidligere frame til den nye frame i en komplet matrix:

$${}^{i-1}_{i}T = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) & 0 & a_{i-1} \\ \sin(\theta_i)\cdot\cos(\alpha_{i-1}) & \cos(\theta_i)\cdot\cos(\alpha_{i-1}) & -\sin(\alpha_{i-1}) & -\sin(\alpha_{1-1})\cdot d_i \\ \sin(\theta_i)\cdot\sin(\alpha_{i-1}) & \cos(\theta_i)\cdot\sin(\alpha_{i-a}) & \cos(\alpha_{i-1}) & \cos(\alpha_{i-1})\cdot d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

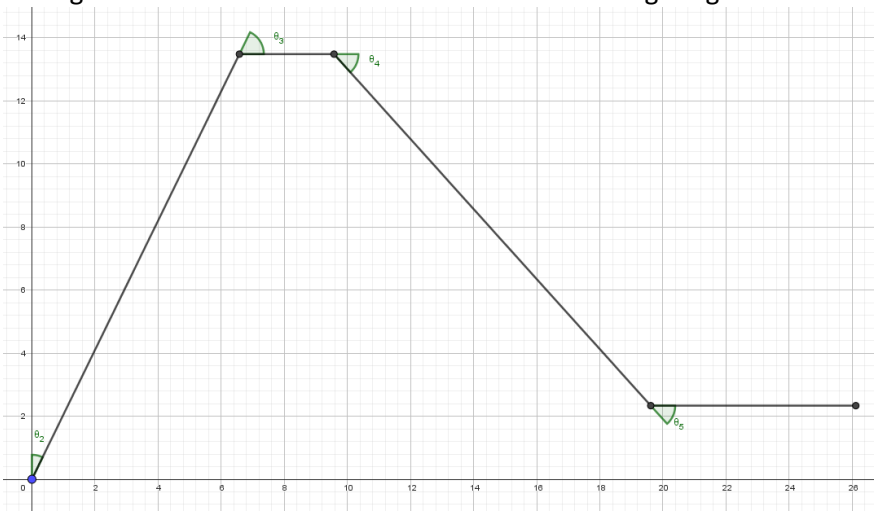
[4] (Craig, 2005)

På denne måde kan der laves en fremadrettet kinematikmodel for en robot, hvis man kender alle Denavit-Hartenberg parametrene. Ved at gange hver matrix med den forrige, fås den komplette fremadrettede kinematikmodel.

For at beskrive dobotten placeres frame 0, der ikke rykker sig. Frame 0 placeres inde i robotten med positiv x lige ud af robotten, og positiv y lige ud mod højre (når robotten ses lige på fra forsiden), og dermed er z-aksen positiv opad på grund af det er et højrehåndet koordinatsystem. Frame 0 placeres i en højde, så den ligger på højde med det første leds rotationspunkt. På denne måde kan den første frame beskrives med θ_1, som beskriver rotation af hele robotten om z-aksen. For at beskrive den næste frame (frame 2) skal der bruges en rotation om frame 1's x-akse. Frame 2 roteres med $-90°$. Det vil sige, at for frame 2 er $\alpha_{2-1} = -90°$. Den er negativ for at sørge for at de næste θ_i ikke kommer til at dreje den forkerte vej, da de altid drejer i positiv omløbsretning om rotationsaksen. Når denne rotation er fundet sted, kan de næste led betragtes som en plan robot, da de kan beskrives fuldstændigt i et plan. Dette plan kan ses på Figur 8. Ved undersøgelse af dobotten fandt jeg ud af, at når det andet led stod i nul stod robotarmen lodret op. Og den vinkel, der forventes af transformationsmatricen, er fra x-aksen i positiv omløbsretning. Derfor beskrives ledvariablen til frame 2 som $90° - \theta_2$. På denne måde giver det $90°$, når θ_2 er 0, så θ_2 er vinklen mellem y-aksen og den første armdel af robotten. Dette kan også ses på Figur 8. Frame 3 er først en forskydning langs x-aksen. Dette er selve længden af leddet så α_{3-1} er 150 mm for dobotten. Rotationen af denne frame er fastlagt mekanisk i robotten, det vil sige, at den kun afhænger af θ_2, og mekanikken sørger for, at den altid er vandret. Derfor skal der findes en vinkel, der simulerer denne opførsel med Denavit-Hartenberg parametrene. For at bestemme hvad θ_3 skal være ud fra θ_2, for at næste del af robotten skal være parallel med x-aksen, tegnes der en linje der er parallel med y-aksen, og da linjen og y-aksen er parallelle, kan vinkel θ_2 overføres til skæring med robotarmen og linjen ifølge Euklid. Og for at den næste del af robotten er parallel med x-aksen, skal den stå $90°$ på y-aksen. Ud fra Figur 7 kan det ses, at $\theta_3 = 90° - \theta_2$. På denne måde kan frame 3 beskrives ud fra frame 2 ved hjælp af θ_2. Frame 4 er helt normal, ved at den kun har en forskydning på $\alpha_{4-1} = 30$ og vinklen θ_4 vender naturligt den rigtige retning. Frame 5 skal forskydes 150 mm på grund af længden af armen, derfor er $\alpha_{5-1} = 150$. Da frame 5 på samme måde som frame 3 skal være parallel med x-aksen, så skal vinklen være modsat den rotation, som blev lavet i frame 4, så derfor $\theta_5 = -\theta_4$. Dette kan ses på Figur 8. Den sidste frame er ikke, lige som de 4 forrige, i planet, så derfor skal den roteres om x-aksen, så den kommer til at stå som den oprindelige frame, det gøres ved at sætte $\alpha_{6-1} = 90°$, det er den modsatte rotation af α_{2-1}. Og så har den en helt normal θ_6, som beskriver rotationen af spidsen af robotten. De komplette Denavit-Hartenberg parametre ses i Tabel 1.



Figur 7 Udsnit af planmodel af dobot



Figur 8 Model af plan del af dobot

i	α_{i-1}	a_{i-1}	d_i	θ_i
1	0	0	0	θ_1
2	$-90°$	0	0	$\theta_2 - 90°$
3	0	150	0	$90° - \theta_2$
4	0	30	0	θ_4
5	0	150	0	$-\theta_4$
6	$90°$	65	0	θ_6

Tabel 1 Denavit-Hartenberg parametre for en dobot

Nu opstilles der en transformationsmatrix for hver frame på dobotten ud fra Denavit-Hartenberg parametrene. Denne matrix beskriver transformationen fra en frame til en anden. Så transformationen fra frame 0 (basen) til frame 1 ser sådan her ud

$$ {}^{0}_{1}T = \begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1) & 0 & 0 \\ \sin(\theta_1) & \cos(\theta_1) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

På samme måde med resten

$$ {}^{1}_{2}T = \begin{bmatrix} \sin(\theta_2) & \cos(\theta_2) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ cos(\theta_2) & -\sin(\theta_2) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ {}^{2}_{3}T = \begin{bmatrix} \sin(\theta_2) & -\cos(\theta_2) & 0 & 150 \\ \cos(\theta_2) & \sin(\theta_2) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ {}^{3}_{4}T = \begin{bmatrix} \cos(\theta_4) & -\sin(\theta_4) & 0 & 30 \\ \sin(\theta_4) & \cos(\theta_4) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ {}^{4}_{5}T = \begin{bmatrix} \cos(\theta_4) & \sin(\theta_4) & 0 & 150 \\ -\sin(\theta_4) & \cos(\theta_4) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ {}^{5}_{6}T = \begin{bmatrix} \cos(\theta_6) & -sin(\theta_6) & 0 & 60 \\ 0 & 0 & -1 & 0 \\ sin(\theta_6) & \cos(\theta_6) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

For at få den komplette transformation fra bunden af robotten til værktøjet på robotten multipliceres disse matricer for at få en matrix, der beskriver hele transformationen

$$ {}^{0}_{6}T = {}^{0}_{1}T\,{}^{1}_{2}T\,{}^{2}_{3}T\,{}^{3}_{4}T\,{}^{4}_{5}T\,{}^{5}_{6}T $$

$$ {}^{0}_{6}T = \begin{bmatrix} -sin(\theta_1)\cdot sin(\theta_6) + cos(\theta_1)\cdot cos(\theta_6) & -cos(\theta_1)\cdot sin(\theta_6) - sin(\theta_1)\cdot cos(\theta_6) & 0 & \left(90 + 150\cdot cos(\theta_4) + 150\cdot sin(\theta_2)\right)\cdot cos(\theta_1) \\ cos(\theta_1)\cdot sin(\theta_6) + sin(\theta_1)\cdot cos(\theta_6) & -sin(\theta_1)\cdot sin(\theta_6) + cos(\theta_1)\cdot cos(\theta_6) & 0 & sin(\theta_1)\cdot(90 + 150\cdot cos(\theta_4) + 150\cdot sin(\theta_2)) \\ 0 & 0 & 1 & -150\cdot sin(\theta_4) + 150\cdot cos(\theta_2) \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

Med denne matrix kan positionen af værktøjet på robotten beregnes, hvis alle vinklerne kendes, det vil sige, at med denne er den fremadrettede kinematik for dobotten beskrevet.

Invers kinematik

Nu kendes den fremadrettede kinematik for dobotten, men for at kunne sige til robotten at den skal bevæge sig et bestem sted hen i rummet ud fra et koordinatsæt og en rotation af værktøjet, er det nødvendigt at kunne gå den anden vej, det vil sige fra et positions matrix til 4 vinkler til de 4 led, dette er også kaldet invers kinematik. Lad os sige at vi gerne vil til position P.

$$ P = \begin{bmatrix} \cos(\phi) & -\sin(\phi) & 0 & P_x \\ \sin(\phi) & \cos(\phi) & 0 & P_y \\ 0 & 0 & 1 & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

Grunden til at denne matrix ikke har særlig mange rotationsmuligheder, fordi dobotten kun kan rotere om z-aksen, da den er mekanisk låst på alle andre akser, rotation af værktøjet er beskrevet med ϕ. For at komme fra P til de 4 vinkler vil jeg bruge geometrisk argumentation. Det er også muligt at løse invers kinematikproblemer med algebra men i dobottens tilfælde er det simplere at gøre det med geometrisk argumentation.

θ_1 er relativ simple da den ikke afhænger af andet en position af slutpunktet. Der kan tegnes en trekant hvor den ene katete er P_x og den anden katete er P_y. Det kan ses på Figur 9. På denne måde kan vi tage den inverse tangens, så derfor kan θ_1 beskrives på denne måde

$$ \theta_1 = \tan^{-1}\left(\frac{P_y}{P_x}\right) $$

θ_6 er kun afhængig af den ønskede vinkel ϕ og θ_1 fordi θ_2 og θ_4 ikke har nogen betydning for vinkellen på værktøjet. For at beregnere θ_6, opstilles denne ligning som beskriver vinklen ϕ ud fra θ_1 og θ_6, hvor ϕ er vinkellen af værktøjet på robotten. Det kan ses på Figur 9 at de to langt sammen giver den endelige rotation af værktøjet på robotten.

$$ \phi = \theta_1 + \theta_6 $$

For at finde θ_6 trækkes θ_1 fra på begge sider.

$$ \theta_6 = \phi - \theta_1 $$

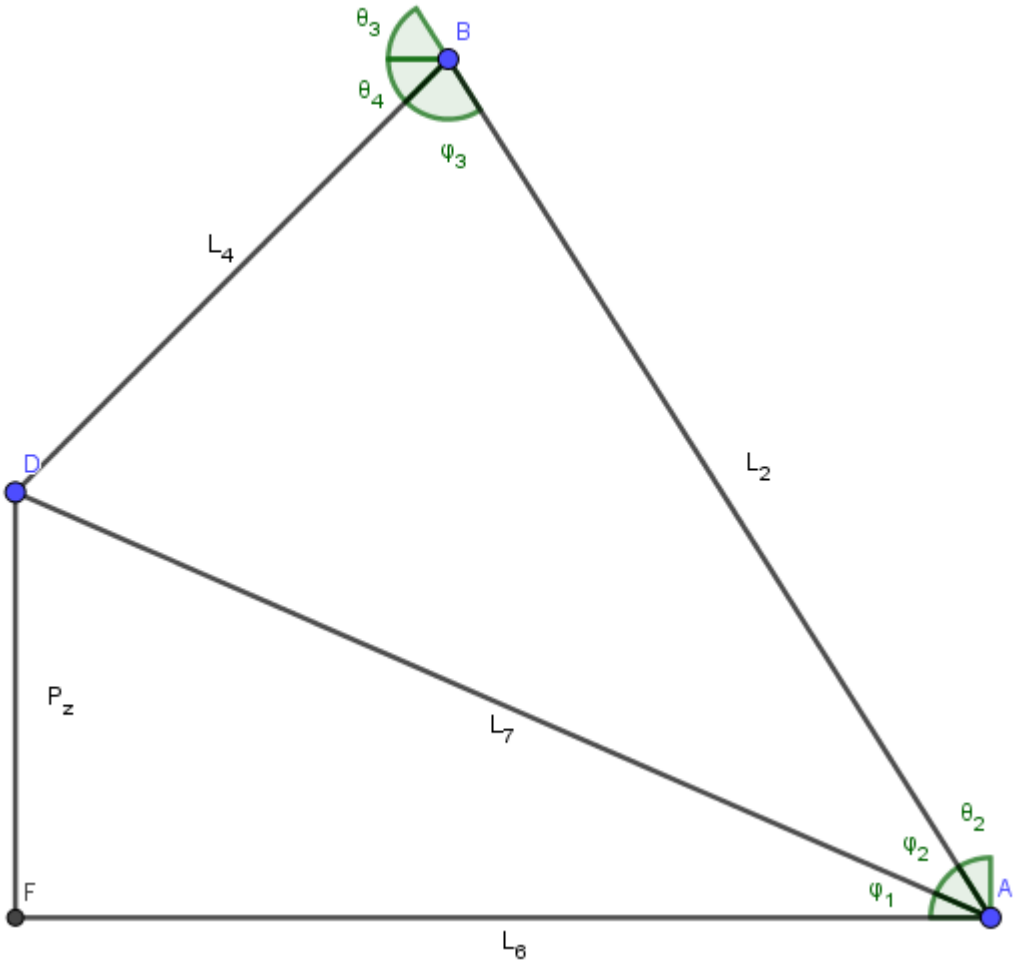
Figur 9 Visualisering af θ1 og θ6

For at finde de sidste to vinkler ud fra den ønskede position, skal vi betragte armen i planet for at forsimple det. Dette kan lade sig gøre fordi θ_2 og θ_4 kun bevæger sig om en akse. På Figur 10 er der sat punkter ind i alle frames' position i planet. L_2 til L_5 er længder der er kendt ud fra udformning af robotten. Længden L_1 er afstanden fra frame 0 til robottens værktøj position, projekteret ned på xy-planet. Det vil sige at hvis P_y er 0 så er $L_1 = P_x$, og for at finde L_1 til alle positioner kan Pythagoras læresætning om retvinklede trekanter bruges, hvor L_1 er hypotenusen og P_x og P_y er kateter, denne trekant ses på Figur 9. Så L_1 findes ved

$$L_1 = \sqrt{P_x^2 + P_y^2}$$

Figur 10 Plan model for dobot

Den form der ses på Figur 10 er ret kompliceret, derfor forsimpler jeg den så det kommer til at ligne en helt almindelige plan robot med 2 rotationsled. For at gøre dette trækker jeg L_3 og L_5 fra modellen, det vil sige at jeg samler det til én trekant. Med andre ord fjerner jeg de flade mekanisk låste vandrette sektioner, mellem punkt B og C, og mellem D og E, når jeg fjerner disse sektioner, samler jeg det hele for at jeg ikke for nogen huller i figuren. Denne ses på Figur 11. Jeg har også tilføjet et linjestykke fra A til D. Den gør at jeg har en lukket trekant ABD.



Figur 11 Forsimplet model for en plan Dobot

L_6 er den tidligere L_1 men jeg har fjernet nogle stykker fra den, så længden af den er nu givet ved:

$$L_6 = L_1 - L_3 - L_5$$

For at jeg kan begynde at regne på ABD trekanten skal jeg først kende tre værdier i trekanten. Jeg kender L_2 og L_4 da de er konstante og begge to givet ud fra robottens fysiske dimensioner, på denne specifikke robot er de begge $150\ mm$, men jeg forsætter med at arbejde med dem generelt. Det eneste ukendte side længde i trekanten ABD er L_7, den er hypotenusen i en vinkelret trekant ADF, hvor begge kateter er kendt, derved kan L_7 findes ved hjælp af Pythagoras.

$$L_7 = \sqrt{P_z^2 + L_6^2}$$

Vinkel ϕ_1 kan findes ved hjælp af den inverse tanges

$$\phi_1 = \tan^{-1}\left(\frac{P_z}{L_6}\right)$$

For at finde ϕ_2 bruger jeg cosinusrelationerne da jeg kender alle side længderne i trekanten.

$$\phi_2 = \cos^{-1}\left(\frac{L_7^2 + L_2^2 - L_4^2}{2 \cdot L_7 \cdot L_2}\right)$$

På samme måde kan jeg finde ϕ_3.

$$\phi_3 = \cos^{-1}\left(\frac{L_4^2 + L_2^2 - L_7^2}{2 \cdot L_4 \cdot L_2}\right)$$

Nu er alle værdierne kendt der skal bruges til at finde θ_2 og θ_4. På Figur 11 ses at det at θ_2 og $\phi_1$og ϕ_2 sammenlagt giver en ret vinkel. Derfor kan θ_2 udtryks sådan:

$$\theta_2 = 90 - \phi_1 - \phi_2$$

Og for at finde θ_4 skal jeg bruge θ_3 og da jeg lavede den fremadrettede kinematik fandt jeg at

$$\theta_3 = 90° - \theta_2$$

Ud fra Figur 11 ses det at

$$180° = \theta_4 + \theta_3 + \phi_3$$

Jeg isolerer θ_4

$$\theta_4 = 180° - \theta_3 - \phi_3$$

Og indsætter definition af θ_3 ud fra θ_2

$$\theta_4 = 180° - (90° - \theta_2) - \phi_3$$

Så hæver jeg minus parentesen, ved at vende alle fortegnene i parentesen

$$\theta_4 = 180° - 90° + \theta_2 - \phi_3$$

Til sidst reducerer jeg

$$\theta_4 = 90° + \theta_2 - \phi_3$$

Ved at trække nogle få af ligninger ovenfor sammen findes disse 9 ligninger som beskriver den inverse kinematik i dobotten:

$$\theta_1 = \tan^{-1}\left(\frac{P_y}{P_x}\right)$$
$$L_6 = \sqrt{P_x^2 + P_y^2} - L_3 - L_5$$
$$L_7 = \sqrt{P_z^2 + L_6^2}$$
$$\phi_1 = \tan^{-1}\left(\frac{P_z}{L_6}\right)$$
$$\phi_2 = \cos^{-1}\left(\frac{L_7^2 + L_2^2 - L_4^2}{2 \cdot L_7 \cdot L_2}\right)$$
$$\phi_3 = \cos^{-1}\left(\frac{L_4^2 + L_2^2 - L_7^2}{2 \cdot L_4 \cdot L_2}\right)$$
$$\theta_2 = 90 - \phi_1 - \phi_2$$
$$\theta_4 = 180° - \theta_3 - \phi_3$$
$$\theta_6 = \phi - \theta_1$$

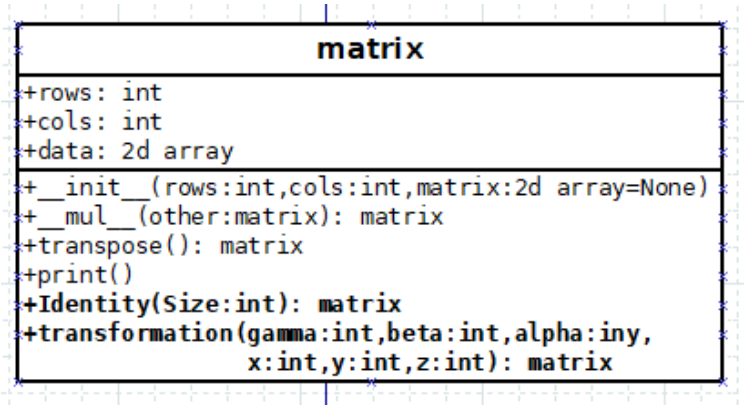
Hvor

$$P = \begin{bmatrix} \cos(\phi) & -\sin(\phi) & 0 & P_x \\ \sin(\phi) & \cos(\phi) & 0 & P_y \\ 0 & 0 & 1 & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Analyse af Python programmer

Analyse af Python bibliotek til regning med matricer

For at lave et bibliotek til at regne med matricer, skal biblioteket have en klasse med noget data om matricen, den skal kende størrelsen af matricen, det vil sige antallet at af rækker og kolonner, den skal også opbevare selve dataen i matricen, det vil sige de tal der er i matricen. Det er alt klassen opbevarer, dette kan ses på klassediagrammet på Figur 12. Data variablen er en todimensional liste, det betyder at det er en liste af lister. Dette stemmer godt overens med at man kan se en matrix som en vektor af vektorer. Ud fra klassediagrammet på Figur 12, kan det ses at klassen har 3 normale funktioner der kun kan kaldes med et instans af klassen nemlig __mul__, transpose og print. Ud over det har den en konstruktør funktion som laver objektet kaldet __init__. Til sidst er der de to funktioner der er mærkeret med fed, identity og transformation, de er klasse metoder, det betyder at det ikke er nødvendigt med en instans af klassen til at kalde disse funktioner, men i stedet skal de kaldes direkte på klassen. Grund til at de er lavet på denne måde er fordi det er fabriks metoder, det betyder at de bruges til at lave nogle specifikke matricer som man ofte har brug for at lave, på denne måde bliver det lettere for dem som skal bruge biblioteket.



Figur 12 Klasse diagram over matrixklassen

```
 4 class matrix:
 5     def __init__(self, rows, cols, matrix=None):
 6         self.rows = rows
 7         self.cols = cols
 8         if matrix is None:
 9             self.data = [[0 for _ in range(cols)]for _ in range(rows)]
10         else:
11             if not isinstance(matrix, list) or not isinstance(matrix[0], list):
12                 raise Exception("Matrix must be a 2d list")
13             self.data = matrix
14             self.rows = len(matrix)
15             self.cols = len(matrix[0])
```

Figur 13 Konstruktør

Lad os starte med at kigge på konstruktøren i klassen, det ses på linje 5 i Figur 13 at init metoden kan tage op til 3 parameter ud over self, den skal have antal af rækker og kolonner, men behøves ikke at have en matrix fordi den har en standard værdi, som er none, det betyder at den er tom. I andre programmeringssprog end Python har man mulighed for at lave flere konstruktør, så man kan lave en konstruktør som tager et antal rækker og søljer, og så laver en tom matrix og så en anden konstruktør der kun tager en 2d liste, også selv regner antal af rækker og kolonner ud, men i Python er det nødvendigt at lave det i en konstruktør, som har nogle standardværdier. På linje 8 tjekkes der om den har fået en matrix, hvis den ikke har, initialiser den bare data til at være en 2d list fyldt med nul. Men hvis den har fået noget tjekker den på linje 11 om det er en 2d liste, hvis ikke rejser den en exception, for at fortælle programmøren at vedkommet har gjort noget forkert ved ikke at sende et 2d liste som argument til matrix argumentet. Men hvis det er en 2d liste så sætter den data variablen til at være den matrix der er blevet sendt med til konstruktøren. Til sidst på linje 14 og 15 sætter den rows og cols til at være højden og bredden af matricen.

```
17     def __mul__(self, other):
18         if self.cols != other.rows:
19             raise Exception(
20                 "The first matrix must have the same numbers of coloms as the other has rows")
21         m = matrix(self.rows, other.cols)
22         for row in range(self.rows):
23             for col in range(other.cols):
24                 vec1 = self.data[row]
25                 vec2 = [other.data[i][col] for i in range(other.rows)]
26                 dotproduct = sum(vec1[i] * vec2[i] for i in range(self.cols))
27                 m.data[row][col] = dotproduct
28 
29         return m
```

Figur 14 Matrix multiplikation metode

For at implementere en funktion til at gange to matricer sammen bruges nøgleordet __mul__ på linje 17 på Figur 14, ved at kalde metoden dette fortæller det Python at denne funktion skal kaldes når der sættes gangetegn mellem to matrix objekter. Det første der bliver tjekket på linje 18, er om det er muligt at gange de to matricer sammen, da matrix-matrixproduktet kun er defineret når antallet af kolonner i den første matrix er det samme som antal rækker i den anden matrix. Hvis sætning spørger om de ikke er det samme, og hvis de ikke er det samme rejser den en exception, for at fortælle programmøren at de matricer vedkommet forsøgte at gange sammen ikke har den rigtige størrelse så det kan ikke lade sig gøre at multiplicere dem. Nu ved programmet at det er muligt at finde et produkt mellem de to matricer, så det forsætter ved at går igennem 2 løkker, og på denne måde regner et resultat ud for hver position i den resulterende matrix. På linje 24 og 25 lægger den de to vektorer der skal prikkes ud i variablerne vec1 og vec2. På linje 26 findes prik produktet ved at går igennem være værdi i vektoren og gange dem sammen, og til sidst tage summen. Dette gøres ved alle tal i den resulterende matrix, og til sidst returner metoden den nye matrix.

```
34     def transpose(self):
35         return matrix(self.cols, self.rows, [[self.data[row][col] for row in
   range(self.rows)] for col in range(self.cols)])
```

Figur 15 Transponer metode

Metoden der transponerer funktion, bruger 2 inline for løkker til at spejle matricen over diagonalen, dette kan ses på Figur 15 på linje 35.

```
31     def invert(self):
32         pass
```

Figur 16 Manglende inverter metode

Inverter metoden på Figur 16 er ikke implementeret i dette bibliotek, da det ikke er helt simpelt at gøre, men det er den første metode der skulle tilføjes, hvis biblioteket skulle udvides. Og da det ikke er nødvendigt i demonstrationsprogrammet der arbejder med dobotten, har jeg valgt at udlade den forholdsvis væsentlige metoden fra biblioteket.

```
37     def print(self):
38         for row in range(self.rows):
39             string = ""
40             for col in range(self.cols):
41                 string += f'{self.data[row][col]} '
42             print(string)
```

Figur 17 Print metode

Matrixklassen har en printfunktion som printer matricen til konsollen, dette gøres for at lettere kunne fejlfinde koden. Metoden fungerer ved at for hver række laver den en string, som består at dataen fra den række, så printer

denne string. For at udvide den string der arbejdes på bruges += operatoren som tilføjer den nye string til enden af variablen string, dette ses på Figur 17 på linje 41. Når der printes til konsollen laver den et linjeskifte. På denne måde kommer hver række på hver sin linje. På Figur 18 er der et eksempel på hvordan det kan se ud når den printer en matrix til konsollen, i dette tilfælde er den en 3×2 matrix.

```
D:\Files\Documents\jesper\skole\GYM\SOP\Dobot>python matrix.py
1 3
1 5
2 -3
```

Figur 18 Eksempel på print af matrix

De sidste to funktioner i klassen er klassemetoder, betydende at de ikke kræver et instans af klassen. Det kan ses ved at der står @classmethod før begge metoder, se Figur 19 på linje 44 og 53. Når det er en klassemetode, skal den altid modtage en reference til selve klassen som det første parameter i funktionen, dette er i modsætning til normale metoder i et objekt i Python som skal have et parameter til en reference til sig selv, dette kan ses på Figur 17 på linje 37. I disse fabriksmetoder bruges klassereferencen ikke, men de skal have den alligevel fordi Python sender den med uanset hvad. Identity funktion laver en identitets matrix i størrelsen den modtager i size parameteret. På linje 46 laver den en tom kvadratisk matrix, da den siger at den skal have det samme antal rækker som antal kolonner. For at lave dette om til en identitets matrix sætter den 1 ind på diagonalen, ved hjælp af for løkken på linje 47 og 48.

```
44      @classmethod
45      def Identity(cls, size):
46          m = matrix(size, size)
47          for i in range(size):
48              m.data[i][i] = 1
49
50          return m
51
52      # fixed angels X-Y-Z in degrees, alpha around z, beta around y, gamma around x
53      @classmethod
54      def tranformation(cls, gamma, beta, alpha, x, y, z):
55          gamma = radians(gamma)
56          beta = radians(beta)
57          alpha = radians(alpha)
58          data = [[cos(alpha)*cos(beta), cos(alpha)*sin(beta)*sin(gamma)-
    sin(alpha)*cos(gamma), cos(alpha)*sin(beta)*cos(gamma)+sin(alpha)*sin(gamma), x],
                    [sin(alpha)*cos(beta),
59  sin(alpha)*sin(beta)*sin(gamma)+cos(alpha)*cos(gamma),
    sin(alpha)*sin(beta)*cos(gamma)-cos(alpha)*sin(gamma), y],
60                  [-sin(beta), cos(beta)*sin(gamma), cos(beta)*cos(gamma), z],
61                  [0, 0, 0, 1]
62                  ]
63          return matrix(4, 4, [[round(data[x][y], 4) for y in range(4)] for x in
    range(4)])
```

Figur 19 Fabriks metoder til at lave identitesmatricer og transformationsmatricer

Transformations metode laver en homogen transformations matrix, ud fra 3 vinkler der beskriver rotationen om 3 fast låste akser, først en rotation om x-aksen så y-aksen til sidst z-aksen, og en translation med x, y, z. Jeg bruger matricen fundet i afsnittet Rotation. På linje 55 til 57 som ses på Figur 19, laves vinklerne fra grader til radianer fordi Pythons math bibliotek regner med radianer i de trigonometriske funktioner. På linje 63 laves denne 2d liste om til en matrix og samtidigt bliver det rundet af til 4 decimaler.

```
66 if __name__ == "__main__":
67     m1 = matrix(3, 2, [[1, 3], [1, 5], [2, -3]])
68     m1.print()
69     print("")
70 
71     m2 = matrix.Identity(2)
72     m2.print()
73     print("")
74 
75     ans = m1 * m2
76     ans.print()
77     matrix.Identity(10)
78     matrix.tranformation(-90, 0, 0, 10, 15, 20).print()
```

Figur 20 Test af Python bibliotek

For at teste et bibliotek i Python, kan der laves en hvis sætning som på linje 66 på Figur 20, til at tjekke om filen bliver kørt selvstændigt eller om det er er blevet inkluderet af en anden Python program, så linjerne 67 til 78 kører kun hvis den bliver kørt selvstændigt, på denne måde kan man test om biblioteket gør som det forventes. I dette bibliotek er er det ikke implementeret så programmet selv ved om testen er rigtig, men der skal programmøren tjekke, men her kunne det udvides med unit testning, hvor man tester være enkelt metode, med noget kendt input, så man også kender resultatet.

Analyse af program til styring af dobot

```
1 import pydobot
2 from serial.tools import list_ports
3 from math import atan2, sqrt, acos, sin, degrees, cos, radians
4 from matrix import matrix
5 
6 ports = [p.device for p in list_ports.comports()]
7 
8 for i, port in enumerate(ports):
9     print(f"{i}: {port}")
10 
11 port = ports[int(input("Vælg en port: "))]
12 
13 try:
14     dobot = pydobot.Dobot(port=port)
15 except:
16     print("Fejl, kunne ikke forbinde til robotten")
17 print("Forbindelse oprettet")
18 
19 L2 = 150
20 L3 = 30
21 L4 = 150
22 L5 = 60
23 mode = pydobot.enums.PTPMode.MOVJ_ANGLE
```

Figur 21 Opsætning af program til styring af dobot

For at kunne styre en robot gennem et Python program skal der først oprettes kontakt til robotten. Computeren og robotten snakker sammen gennem en helt almindelige seriel port. Kontakten oprettes som det første i programmet fra linje 6 til 17 på Figur 21. Inden da importeres alle biblioteker programmet skal bruge, på linje 1 til 4. Læg særlig mærke til at matrix-klassen fra biblioteket importeres på linje 4.

Der laves på linje 14 et dobot objekt som håndterer al kommunikation med dobotten fra computeren. Det forsætter med opsætning af programmet når der på linje 19 til 22 bliver oprettet variabler som beskriver den fysiske størrelse af robotten. Disse variabler har de samme navne som der blev brugt da, den invers kinematikmodel blev fundet. Det gør det let at overføre matematikken til programmet.

Den sidste opsætning der skal til er at vælge hvilken mode dobotten skal bevæge sig med. Dette sker på linje 23. MOVJ_ANGLE betyder at den for koordinaterne til position i jointrummet, det vil sige at de 4 værdier der sendes til dobotten svar til vinklerne til hver af de 4 led på robotten. j'et betyder at den skal bevæge sig fra et punkt til et andet bare ved at gå fra den gamle vinkel til den nye vinkel for hver af motorerne. Det betyder man ikke kender hvilken vej robotten kommer hen til det nye punkt. Modsat kunne man sætte et L i stedet for j'et, også ville den bevæge sig i en lige linje hen til det nye punkt, men for at den kan gøre det skal dobotten lave invers kinematik også giver det ikke mening jeg selv har implementeret det.[5]

[5] (luismesas, et al., 2020)

```
26 def moveToCord(px, py, pz, vinkel):
27     theta1 = atan2(py, px)
28     L6 = sqrt(px*px+py*py)-L3-L5
29     L7 = sqrt(pz*pz+L6*L6)
30     phi1 = atan2(pz, L6)
31     phi2 = acos((L7*L7+L2*L2-L4*L4)/(2*L7*L2))
32     phi3 = acos((L4*L4+L2*L2-L7*L7)/(2*L2*L4))
33     theta1 = degrees(theta1)
34     theta2 = 90 - degrees(phi1) - degrees(phi2)
35     theta4 = 90 + theta2 - degrees(phi3)
36     theta6 = vinkel - theta1
37 
38     dobot._set_ptp_cmd(round(theta1, 5), round(theta2, 5), round(theta4, 5), round(theta6, 5), mode=mode, wait=True)
39 
40 
41 def movetomatrix(m):
42     moveToCord(m.data[0][3], m.data[1][3], m.data[2][3], degrees(acos(m.data[0][0])))
```

Figur 22 Funktion der gør brug af den invers kinematikmodel

På Figur 22 ses funktionen moveToCord som får dobotten til at bevæge sig hen til et givent punkt, parametrene er et koordinatsæt og en vinkel der beskriver værktøjets rotation. Linjerne 27 til 36 på Figur 22 svar en til en med de ligninger jeg fandt i afsnit Invers kinematik, de står i samme rækkefølge som i afslutningen af dette afsnit. Den eneste ting som er tilføjet, er at der bruges funktionen degrees til at omregne fra radianer til grader på linjerne 33 til 36. Her ses det at når formlerne er fundet er det helt simpelt at skrive dem ind et program og kører efter dem. På linje 38 sendes kommandoen til dobotten til at få den til at bevæge sig hen til de udregnet vinkler så værktøjet på robotten kommer til at være på punktet, der blev sendt i parametrene til funktionen, alle vinkelværdierne bliver afrundet til 5 decimaler, da jeg har oplevet at robotten gik i fejl hvis den fik for mange decimaler. Her bruges den mode der blev valgt på linje 23.

```
45 def get_pose(t1, t2, t4, t6):
46     t1 = radians(t1)
47     t2 = radians(t2)
48     t4 = radians(t4)
49     t6 = radians(t6)
50     data = [[-sin(t1)*sin(t6)+cos(t1)*cos(t6), -cos(t1)*sin(t6)-sin(t1)*cos(t6), 0, (90+L4*cos(t4)+L2*cos(t2-radians(90)))*cos(t1)],
51             [cos(t1)*sin(t6)+sin(t1)*cos(t6), -sin(t1)*sin(t6)+cos(t1)*cos(t6), 0, sin(t1)*(90+L3*cos(t4)+L2*cos(t2-radians(90)))],
52             [0, 0, 1, -L3*sin(t4)-L2*sin(t2-radians(90))],
53             [0, 0, 0, 1]]
54     data = [[round(data[x][y], 4) for y in range(4)] for x in range(4)]
55     postioin = matrix(4, 4, data)
56     return postioin
57 
58 
59 def get_dobot_pos():
60     (_, _, _, _, j1, j2, j3, j4) = dobot.pose()
61     return get_pose(j1, j2, j3, j4)
```

Figur 23 Fremadrettet kinematik funktion

Funktionen get_pose der ses på Figur 23, får 4 vinkler fra robotten, og så omregnes det til en matrix der beskriver positionen af robotten. Funktionen bruger den fremadrettet kinematik matrix jeg fandt i afsnittet Fremadrettet kinematik. Denne matrix er skrevet direkte ind på linjerne 50 til 53. For at kunne brug de trigonometriske funktioner omregnes der inden til radianer på linjerne 46 til 49.

Der er en hjælpe funktion til hver af de to ovenstående funktioner, der gør dem lettere at bruge, på Figur 22 på linje 41 og 42 er der en funktion movetomatrix som har en matrix som parameter og den finder de relevante data fra matricen og sender dem videre til moveToCord funktionen. Og på Figur 23 på linje 59 til 61, ses get_dobot_pos funktion, som bruger dobot biblioteket positionsfunktion til at få vinklerne på motorerne lige nu, disse vinkler sender den videre til get_pos funktion som laver fremadrettet kinematik på dem. Dette er alle funktioner der skal bruges til at kommandere dobotten til at gøre hvad man vil have den til.

```
64 moveToCord(300, -100, 0, 50)
65
66 m = get_dobot_pos()
67 m.print()
68 transformation = matrix.tranformation(0, 0, 0, -100, 40, 40)
69
70 reslut = m * transformation
71 print("")
72 reslut.print()
73
74 movetomatrix(reslut)
75
76
77 step_count = 7
78 p1 = (170, 150, 110)
79 p2 = (290, -150, -30)
80 step = ((p2[0]-p1[0])/step_count, (p2[1]-p1[1]) / step_count, (p2[2]-p1[2])/step_count)
81 while(True):
82     for i in list(range(step_count+1))+list(reversed(range(step_count+1))):
83         moveToCord(p1[0]+step[0]*i, p1[1]+step[1]*i, p1[2]+step[2]*i, 90)
```

Figur 24 Test af dobot styrering

For at teste disse funktioner afsluttes programmet med en kort test. På Figur 24 på linjerne 64 til 74 testets matrixregningen ved at på linje 64 fortæller den at den skal kører til 300, -100, 0 og vinkel på $50°$. Derefter aflæses positionen af robotten, dette returnerer en matrix, så på linje 68 laves en transformations matrix med fabriksmetoden. Disse to matricer ganges sammen og så flyttes robotten hen til det nye punkt på linje 74. Det viser at det er muligt af flytte dobotten i forhold til dens nuværende position og rotation. Linjerne 77 til 83 kører dobotten fremad og tilbage i en lige linje mellem to punkter for at vise at invers kinematikmodellen virker.

Hvordan er den matematiske viden nødvendig for programmør?

Som det ses i den ovenstående del af opgaven, er det et stort arbejde der ligger i at lave fremadrettet kinematek og den invers kinematikmodel for dobotten, så nu er det relevant at undersøge hvor stor del af det arbejde der er nødvendigt at programmøren laver. Der er flere måder at løse invers kinematikproblemet på, i denne opgave er der brugt en analytisk metode som finder en lukket form. Med andre ord en funktion som kan regne positionen af leddene ud fra mål positionen. Det er også muligt at bruge numeriske løsninger, hvor man bruger en iterativ metode, hvor man prøver nogle vinkler, og ser hvor tæt robotten er på mål positionen, ud fra denne information ændre man på vinklerne for at komme tættere på. Der findes forskellige metoder til at lave iterativ invers kinematik, nogle er helt simple andre indeholder mere matematik for at effektivisere optimeringsproblemet. Når man skal arbejde med programmering af robotter, kan man de fleste gange lave alt matematikken selv, men det er meget tidskrævende og forholdsvist svært. Men man kan finde biblioteker der er lavet til at løse invers kinematikproblemer, så skal man beskrive den robot der arbejdes med, og så vil den højst sandsynligt kunne løse problemet for programmøren. Dette gøre det meget lettere at komme i gang, dette er især smart hvis det er et hobby projekt hvor det ikke er nødvendigt med den matematiske forståelse som opnås ved at selv at lave det hele.[6] Det er også muligt at finde biblioteker som gør det lettere at arbejde med matematikken, men som ikke er direkte er lavet til at løse invers kinematikproblemer, for eksempel kan man brug numPy til at regne med matricer, og på denne måde ikke fokuserer på hvordan lineær algebra virker men bare på hvordan den bruges.[7] Dobotten er lavet til undervisning så derfor har den indbygget en invers kinematikmodel, så det er normalt ikke nødvendigt at tænke på den invers kinematik del, det gør det muligt at fokuserer på andre problemstillinger, som kan løses med robotten. For eksempel vises på dobottens hjemmeside en masse projekter der indeholder AI, og hvis der fokuseres på denne del, er den invers kinematik ikke interessant, og derfor er det smart dobotten har løst kinematik problemet og så der plads til at fokuserer på andre dele af

[6] (The Robotics Back-End, n.d.)

[7] (numpy.otg, n.d.)

problemløsningen.[8] Hvis der arbejdes i industrien hvor det handler hurtigst muligt at få noget der virker i produktion, er der produkter som robodk leverer som er lavet for at hurtigst muligt få en robot til at virke, de har en masse industri robotter som kan programmers let fra computeren og så kan robotten med det samme gå i gang. De udtaler sig på deres blog at man ikke skal lave matematik hvis det ikke er dig selv der har bygget robotten, for ellers har nogle andre løst problemet og det er ikke nødvendigt at gøre igen.[9]

Konklusion

For at beskrive dobottens kinematiske struktur bruges Denavit-hartenberg notation, og da disse paramenter var blevet fundet, kunne der opstilles matricer for hvert led i robotten, disse ganges sammen for at få den fremadrettet kinematik, for at gange matricer sammen bruges definitionen fra redegørelsen. For at lave den invers kinematikmodel, bruges forskellige trigonometriske formler, på denne måde kunne både den fremadrettet og inverse kinematik beskrives. Den invers kinematikmodel er når den er blevet fundet meget kort og simpel, da den kan beskrive al matematikken med 9 ligninger. Det implementeres i et Python program, med hjælp fra et Python bibliotek der kan regne med matricer, disse giver mulighed for at styr dobotten. De få ligninger der blev fundet med matematikken, gør det let at implementerer det i Python programmet, og gør at det kan kører meget hurtigt da der ikke skal særlig mange beregninger til at udregne positionen af leddene til en given målposition. Til sidst i diskussionen findes det at det ikke altid er nødvendigt at udvikle en komplette kinematikmodel, men det skal vurderes hvad der er relevant for opgaven der skal løses, og hvis det er til læring hvilke læringsmål der for processen.

[8] (Dobot, n.d.)
[9] (Owen-Hill, 2021)

Referencer

Dobot. (u.d.). *Magician Lite*. Hentet 17. December 2022 fra en.dobot.cn: https://en.dobot.cn/products/education/magician-lite.html

luismesas, knutsun, ZdenekM, retospect, ksketo, & remintz. (8. februar 2020). *ptpMode.py*. Hentet 15. December 2022 fra github.com/luismesas/pydobot: https://github.com/luismesas/pydobot/blob/master/pydobot/enums/ptpMode.py

numpy.otg. (u.d.). *Linear algebra (numpy.linalg)*. Hentet 17. December 2022 fra numpy: https://numpy.org/doc/stable/reference/routines.linalg.html

Owen-Hill, A. (14. Juni 2021). *Inverse Kinematics in Robotics: What You Need to Know*. Hentet fra RoboDK: https://robodk.com/blog/inverse-kinematics-in-robotics-what-you-need-to-know/

Pierce, R. (2021. August 25). *How to Multiply Matrices*. Hentet 8. December 2022 fra Math Is Fun: http://www.mathsisfun.com/algebra/matrix-multiplying.htm

STEM EDUCATION WORKS. (u.d.). *DOBOT MAGICIAN LITE*. Hentet 12. December 2022 fra stemeducationworks: https://stemeducationworks.com/product/dobot-magician-lite/

The Robotics Back-End. (u.d.). *Should You Learn Mathematics To Program Robots?* Hentet 17. December 2022 fra The Robotics Back-End: https://roboticsbackend.com/should-you-learn-mathematics-to-program-robots/

Undervisningsministeriet. (27. august 2014). Lineære afbildninger.

Bilag

Kode fra matrix.py

```python
from math import radians, cos, sin


class matrix:
    def __init__(self, rows, cols, matrix=None):
        self.rows = rows
        self.cols = cols
        if matrix is None:
            self.data = [[0 for _ in range(cols)]for _ in range(rows)]
        else:
            if not isinstance(matrix, list) or not isinstance(matrix[0], list):
                raise Exception("Matrix must be a 2d list")
            self.data = matrix
            self.rows = len(matrix)
            self.cols = len(matrix[0])

    def __mul__(self, other):
        if self.cols != other.rows:
            raise Exception(
                "The first matrix must have the same numbers of coloms as the other has rows")
        m = matrix(self.rows, other.cols)
        for row in range(self.rows):
            for col in range(other.cols):
                vec1 = self.data[row]
                vec2 = [other.data[i][col] for i in range(other.rows)]
                dotproduct = sum(vec1[i] * vec2[i] for i in range(self.cols))
                m.data[row][col] = dotproduct

        return m

    def invert(self):
        pass

    def transpose(self):
        return matrix(self.cols, self.rows, [[self.data[row][col] for row in range(self.rows)] for col in range(self.cols)])

    def print(self):
        for row in range(self.rows):
            string = ""
            for col in range(self.cols):
                string += f'{self.data[row][col]} '
            print(string)

    @classmethod
    def Identity(cls, size):
        m = matrix(size, size)
        for i in range(size):
            m.data[i][i] = 1

        return m
```

```
51
52      # fixed angels X-Y-Z in degrees, alpha around z, beta around y, gamma around x
53      @classmethod
54      def tranformation(cls, gamma, beta, alpha, x, y, z):
55          gamma = radians(gamma)
56          beta = radians(beta)
57          alpha = radians(alpha)
58          data = [[cos(alpha)*cos(beta), cos(alpha)*sin(beta)*sin(gamma)-
    sin(alpha)*cos(gamma), cos(alpha)*sin(beta)*cos(gamma)+sin(alpha)*sin(gamma), x],
                    [sin(alpha)*cos(beta),
59  sin(alpha)*sin(beta)*sin(gamma)+cos(alpha)*cos(gamma),
    sin(alpha)*sin(beta)*cos(gamma)-cos(alpha)*sin(gamma), y],
60                  [-sin(beta), cos(beta)*sin(gamma), cos(beta)*cos(gamma), z],
61                  [0, 0, 0, 1]
62                  ]
63          return matrix(4, 4, [[round(data[x][y], 4) for y in range(4)] for x in
    range(4)])
64
65
66  if __name__ == "__main__":
67      m1 = matrix(3, 2, [[1, 3], [1, 5], [2, -3]])
68      m1.print()
69      print("")
70
71      m2 = matrix.Identity(2)
72      m2.print()
73      print("")
74
75      ans = m1 * m2
76      ans.print()
77      matrix.Identity(10)
78      matrix.tranformation(-90, 0, 0, 10, 15, 20).print()
```

Kode fra dobot_controller.py

```
 1  import pydobot
 2  from serial.tools import list_ports
 3  from math import atan2, sqrt, acos, sin, degrees, cos, radians
 4  from matrix import matrix
 5
 6  ports = [p.device for p in list_ports.comports()]
 7
 8  for i, port in enumerate(ports):
 9      print(f"{i}: {port}")
10
11  port = ports[int(input("Vælg en port: "))]
12
13  try:
14      dobot = pydobot.Dobot(port=port)
15  except:
16      print("Fejl, kunne ikke forbinde til robotten")
17  print("Forbindelse oprettet")
18
19  L2 = 150
20  L3 = 30
21  L4 = 150
22  L5 = 60
23  mode = pydobot.enums.PTPMode.MOVJ_ANGLE
```

```
24
25
26 def moveToCord(px, py, pz, vinkel):
27     theta1 = atan2(py, px)
28     L6 = sqrt(px*px+py*py)-L3-L5
29     L7 = sqrt(pz*pz+L6*L6)
30     phi1 = atan2(pz, L6)
31     phi2 = acos((L7*L7+L2*L2-L4*L4)/(2*L7*L2))
32     phi3 = acos((L4*L4+L2*L2-L7*L7)/(2*L2*L4))
33     theta1 = degrees(theta1)
34     theta2 = 90 - degrees(phi1) - degrees(phi2)
35     theta4 = 90 + theta2 - degrees(phi3)
36     theta6 = vinkel - theta1
37     # print(f'j1:{theta1} j2:{theta2} j3:{theta4} j4:{theta6}')
38     dobot._set_ptp_cmd(round(theta1, 5), round(theta2, 5), round(theta4, 5),
   round(theta6, 5), mode=mode, wait=True)
39     (x, y, z, r, j1, j2, j3, j4) = dobot.pose()
40     # print(f'x:{x} y:{y} z:{z} r:{r} j1:{j1} j2:{j2} j3:{j3} j4:{j4}')
41     print(f'Diff: x: {round(px-x,2)} y: {round(py-y,2)} z: {round(pz-z,2)} r:
   {round(vinkel-r,2)}')
42
43
44 def movetomatrix(m):
45     moveToCord(m.data[0][3], m.data[1][3], m.data[2][3],
   degrees(acos(m.data[0][0])))
46
47
48 def get_pose(t1, t2, t4, t6):
49     t1 = radians(t1)
50     t2 = radians(t2)
51     t4 = radians(t4)
52     t6 = radians(t6)
53     data = [[-sin(t1)*sin(t6)+cos(t1)*cos(t6), -cos(t1)*sin(t6)-sin(t1)*cos(t6), 0,
   (90+L4*cos(t4)+L2*cos(t2-radians(90)))*cos(t1)],
54             [cos(t1)*sin(t6)+sin(t1)*cos(t6), -sin(t1)*sin(t6)+cos(t1)*cos(t6), 0,
   sin(t1)*(90+L3*cos(t4)+L2*cos(t2-radians(90)))],
55             [0, 0, 1, -L3*sin(t4)-L2*sin(t2-radians(90))],
56             [0, 0, 0, 1]]
57     data = [[round(data[x][y], 4) for y in range(4)] for x in range(4)]
58     postioin = matrix(4, 4, data)
59     return postioin
60
61
62 def get_dobot_pos():
63     (_, _, _, _, j1, j2, j3, j4) = dobot.pose()
64     return get_pose(j1, j2, j3, j4)
65
66
67 moveToCord(300, -100, 0, 50)
68
69 m = get_dobot_pos()
70 m.print()
71 transformation = matrix.tranformation(0, 0, 0, -100, 40, 40)
72
73 reslut = m * transformation
74 print("")
75 reslut.print()
76
```

```
77 movetomatrix(reslut)
78
79
80 step_count = 7
81 p1 = (170, 150, 110)
82 p2 = (290, -150, -30)
83 step = ((p2[0]-p1[0])/step_count, (p2[1]-p1[1]) / step_count, (p2[2]-
   p1[2])/step_count)
84 while(True):
85     for i in list(range(step_count+1))+list(reversed(range(step_count+1))):
86         moveToCord(p1[0]+step[0]*i, p1[1]+step[1]*i, p1[2]+step[2]*i, 90)
```