Robotkinematik

Skrevet af

Jesper Roager

Afleveret den

17. december 2022

Vejledet af

Søren Præstegaard

Marit Hvalsøe Schou

Resume

Robot kinematik opdeles i to del fremadrettet kinematik og inverse kinematik, for at regne med fremadrettet kinematik bruges der matricer til at lave homogene transformationer. Homogene transformationer opbevare i et 4 gange 4 matrix, og den indeholder både rotation og translation, for at lave disse transformationsmatricer bruges Denavit-Hartenberg paramenterne disse beskriver hvert led på robotten med 4 værdier. Med seks af disse matricer kan en Dobot Magician Lite fremadrettet kinematik beskrives, disse matricer ganges samme får at for en matrix der beskriver position af dobottens værktøj ud fra vinklerne på motorerne. Efter findes der en invers kinematik model ved at betragte de midterste to led på dobotten som en plan robot, og på denne måde bliver problemet til et simpelt i analytisk plangeometri problem. Dette resulteret er i 9 ligning som implementeres i et Python program så det er muligt at styrer dobot ved at give den et koordinat sæt eller et en matrix med position og rotation. For at Python programmet kan regne med matricer bruges et Python bibliotek der laver en matrix klasse som kan lave de flest standardoperationer med matricer, både bibliotek og programmet til styring af dobotten analyser for at forstår deres funktion. Til sidst diskuteres vigtighed af matematik i programmering af robotter, her findes det at hvis det glæder om at komme hurtigt videre er det ikke nødvendig med matricematematikken men det giver en god forståelse, så det kommer an på hvad for målet med programmet er.

Indholdsfortegnelse

Indledning

Robotter er over det hele i verden og de kommer nærmere på i os, så vi møder dem oftere og oftere i vores hverdag. De er uundværlige i produktion af alle de ting vi bruger til daglig, alt fra mobiltelefoner til sæbe og alt der ind imellem. Disse tusindvis af robotter laver massevis af bevægelser dagen lang, og det er vigtigt de udfører bevægelserne korrekt, og for at robotter kan lave bevægelserne rigtigt, skal den vide hvordan leddene skal stå for at nå en given position. For at gøre det lettere at regne med position og rotation, bruges der matricer, så hvordan brugers matricer til at udregne position af leddene i dobot for at nå en given position inden for rækkevidden af robotten? Dette er problem kan inddeles i to problemer. Det ene fremadrettet kinematik problem, som betyder at man skal finde ud af hvor robottens værktøj ender ved et specifik sæt, vinkler på leddene, dette kaldes også forward kinematik. Det andet problem hedder inverse kinematik, hvor der kendes en ønsket position og rotation af værktøjet, også skal vinklerne der opnår den ønsket position findes. For at forstå denne proces med fremadrettet kinematik og inverse kinematik, starter opgaven med en redegørelse for matricer og homogene transformation med matricer. Så beskrives den fremadrettet kinematik for en Dobot Magician Lite og med hjælp fra fremadrettede kinematik, udvikles der et inverse kinematik model for dobotten. Derefter analyseres et Python bibliotek til matrix regning, et program der styr dobotten med den fremadrettet- og inverse- kinematik model. Til sidst diskuteres hvor stor nødvendighed af matematisk viden når der skal programmers robotter.

Redegørelse for lineære transformationer med matricer

Matricer

En vektor er en list af tal som for det meste repræsenter et punkt i et rummet, for eksempel repræsenter

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

$\vec{v}$ en vektor i rummet det vil sige at den har tre dimension, og mængden af denne vektor er $\mathbb{R}^3$, og hvis den så havde n tal i vektoren ville den til høre denne mængde $\mathbb{R}^n$.

En matrix er så en samling af vektor, som står ved siden af hinanden. De vil sige at den består af en mængde tal de er organiseret i rækker og søjler.

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

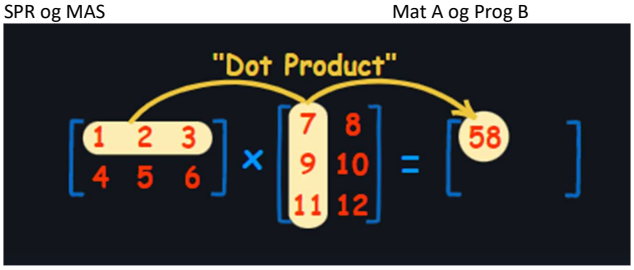
Mængden af n x m matricer betegnes med $\mathbb{R}^{n\times m}$[1]

Matrix multiplikation

Kommenteret [JR1]: Kumkationv og assoativ

Nu skal 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 prik produktet mellem hver række i den første matrix og hver kolonne i den anden matrix, også sættes det givne produkt ind på række nummeret fra det første matrix og kolonne nummeret fra den andet matrix. På Figur 1 ses der et eksempel hvor det er illustreret hvilke vektorer der skal prikkes sammen.

[1] (Undervisningsministeriet, 2014)



Figur 1 Illustration af matrice multiplikation, 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 $C = A \cdot B$ er $C \in \mathbb{R}^{n\times k}$.

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

Her er et eksempel på matrix multiplikation

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

Identitets matrix

Der findes identitet matricer, og de har den samme effekt som når man ganger med et ettal, i de reelle tal, det vil sige at når man gange med den giver de sig selv tilbage. Identitets matricer er kvadratiske, og har ettaller på diagonalen resten er nul. Det kaldes I. Et eksempel på en identitets matrix

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

De kan komme i alle størrelser så længde 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 mange gange med en identitets matrix for man de oprindelige matricer bagefter.

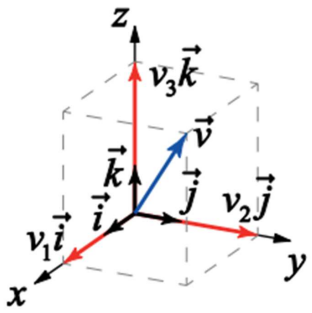
Dette kan også vises med tal eksempel.

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

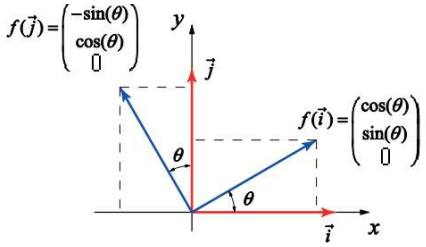
Det er vigtigt at identitetsmatricen har den rigtige størrelse så det er muligt at finde matrix-matrixproduktet.

Transformationer

Matricer kan bruges til at rykke rundt på punkter med, det kaldes en transformation. Det gøres ved at en matrix beskriver de nye basis vektorer ud fra de gamle de gamle basis vektor. En 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 $\vec{k}$. Og 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 $\vec{v} = v_1 \cdot \vec{\imath} + v_2 \cdot \vec{\jmath} + v_3 \cdot \vec{k}$



Figur 2 Illustration af basis vektor figur fra (Undervisningsministeriet, 2014)



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

Så hvis man vil transformer et punkt til et andet punkt kan man nøjes med ændre beskrive de nye basis vektor. De kan både ændre retning og skalares hvis dette findes nødvendigt. På måde kan man lave en lineær afbildning af et punkt ved hjælp af en matrix. Så i rummet ser sådan en af billedning matrix sådan her ud

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

Hvor f er en linær funktion.

Rotation

På denne måde kan rotations 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 ændre sig da vi roteter om z aksen. Se Figur 3. På denne måde kan man beskrive rotation om enhver akse. Senere i opgaven skal bruges en matrix som først rotere om x akse, så y aksen og til sidst z aksen. Dette findes ved at gange være enkelte af disse rotationer sammen.

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

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

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

Translation

For at kunne lave translationer er det ikke nødvendigt med matricer. Da en translation er bare at flytte en vektor med en anden vektor så derfor kan to vektor 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 transformation

For at forsimple operationer som både har en rotation og en translation, kan disse sættes sammen til et enkelt 4 gange 4 matrix. På denne:

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

Hvor R den øverste venstre del er et 3 gange 3 rotation matrix, og kolonner 4 er translation givet ved P, for at kunne gange punkter med denne matrix så tilføjes der et ettal i slutning så de har en størrelse der passer med at gange med et 4 gange 4 matrix, og dette ettal ændre ikke på resultat et, så det skal bare ses bort fra når resultat aflæses. Først sker en rotation så sker translation, men i en og samme operation.

For at forstå det, ses først hvordan en translation fungere. Et transformations matrice der kun laver en translation, ser så da herud:

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

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 uden lukkende ses på en translation, er det tydeligt at denne måde at gange dem sammen på giver det samme som at lægge to vektor normalt sammen.

På samme måde kan rotation undersøges. En transformations matrix der kun laver en rotation om z aksen, ser så da herud:

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

[2] (Craig, 2005)

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

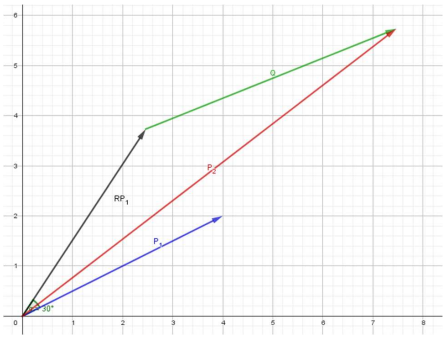
$$\begin{bmatrix} cos\,\theta & -\sin\theta & 0 & 0 \\ \sin\theta & cos\,\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta \cdot P_x - \sin\theta \cdot P_y + 0 \cdot P_z + 0 \cdot 1 \\ \sin\theta \cdot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

Det kan samlingens med hvis der bare ganges med det rotations matrix der tidligere blev fundet

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

Så ses det at det giver det samme, derfor kan en homogen transformation beskrive på ovenstående måde.

Eksempel på homogentransformations matrix



Figur 4 Eksempel homogentransformations

Her er et eksempel en transformation med en transformation matrice

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

Dette transformation matrice forskyder et punkt med 5 på x aksen og 2 på y aksen, og den forskydes ikke på z aksen, og den roterer 30 grader om z, aksen. Punktet som der transformeres

er $P_1 = \begin{bmatrix} 4 \\ 2 \\ 0 \\ 1 \end{bmatrix}$ det sidste ettal til føjes for at de for de rigtige størrelser og det resulterer punkt kalder vi 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, forskyndnings vektoren kaldes Q er grøn, den resultere punkt betegnes P_2 og er rød.

Når der arbejdes men transformation er på robotter placer der frames, et andet ord for koordinatsystem, på alle omdrejningspunkter på robot. Hvert af disse frames position beskrives ud fra den forrige, ved hjælp af en transformations matrix. På denne måde bruges det til at skifte hvilket koordinatsystem der ses på robotten udfra. Så en frame B er beskrevet ud fra frame A, skrives sådan her ${}^{A}_{B}T$, så det der står øverst, viser hvilket koordinatsystem den kommer fra, og er det resulterende koordinatsystem

Beskrivelse af den kinematiske struktur i en Dobot

Forward kinematik

Når den kinematiske struktur af en robot skal beskrive, skal leddenes indbyrdes position beskrives. En Dobot som ses på Figur 5 har 4 led der er drevet og 2 der mekanisk styret, som mekanisk holder en bestemt vinkel. For at beskrive robotten bruges Denavit-Hartenberg notation, som beskriver to frames indbyrdes position og rotation. Til at beskrive den næste frame bruges den nye frame forhold til den forrige frame. Så der startes med en stationer frame 0, denne frame står stille i bunden af robotten og den flytter sig aldrig, efter denne frame tælles der op ad så frame 1, 2 og 3 ind til alle led er beskrevet. I starten og slutning kan det være nødvendigt med en forskydning for der hvor leddene starter ikke er i bunden af robotten og eller at værkøjet i robot armens ende punkt ikke er lige i slutning af det sidste led. Denavit-Hartenberg notation beskrives hvert led med 4 parameterne a_{i-1} α_{i-1} d_i θ_i. Først roteter man med α_{i-1} grader rundt om den er oprindelige x akse, så forskydes med a_{i-1} langs den samme x akse, derefter roteres denne nye frame med θ_i og den nye z akse, det efter en forskydning med d_i hen langs z aksen. Normalt bruges disse to sidste parameter til led variable, det vil sige at det dem der ændres på når robotten ændrer position, hvis ledet er et rotations led bruges led variablen θ_i til at beksrive rotation, d_i sættes til nul og bliver ikke brugt. Det er omvendt ved led som bevæger sig med lineær forskydning. På denne måde kan to frames indbyrdes placering beskrives.

Figur 5 Dobot Magician Lite, billed fra (STEM EDUCATION WORKS, n.d.)

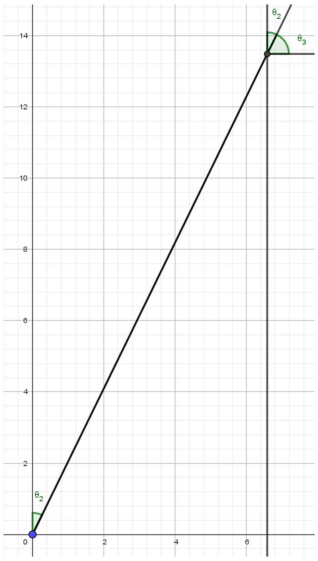
Denne proces består af en rotation om den oprindelige x akse så en translation langs x aksen og så ny rotation om den nye z akse og til sidst en translation langs den nye z akse

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

Og når disse matricer ganges sammen, får vi en transformation fra den tidligere frame til den nye frame i en komplet matrix. Denne transformation kan beskrives med denne matrix

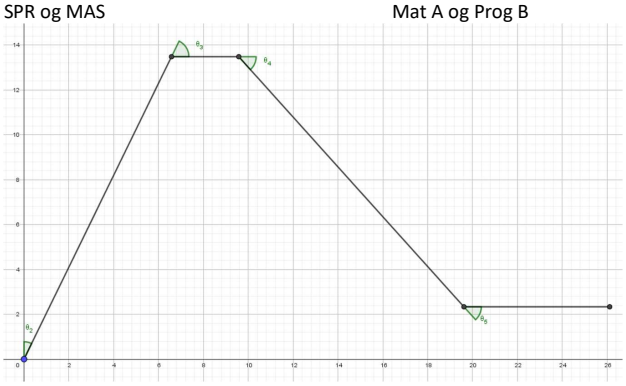
$$^{i-1}_{i}T = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) & 0 & a_{i-1} \\ \sin(\theta_i)\cdot\cos(\alpha_{i-1}) & \cos(\theta_i)\cdot\cos(\alpha_{i-1}) & -\sin(\alpha_{i-1}) & -\sin(\alpha_{1-1})\cdot d_i \\ \sin(\theta_i)\cdot\sin(\alpha_{i-1}) & \cos(\theta_i)\cdot\sin(\alpha_{i-a}) & \cos(\alpha_{i-1})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

På denne måde kan der laves en fremadrettet kinematik model for en robot hvis man kender alle Denavit-hartenberg paramenterne, ved at gange hvert matrix med det forrige fås den komplette fremadrettede kinematik model. For at beskrive dobotten, startes der med at placeres frame 0 den 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 op, og den placere i en højde så det ligger på højde med det første leds rotation punkt. På denne måde kan den første frame bare beskrives med θ_1 som beskriver rotation af hele robotten om z-aksen. For at beskrive den næste frame (frame 2), skal deres bruges en rotation om frames 1's x-akse. Den roteres med $-90°$ det vil sige at for frame 2 er $\alpha_{2-1} = -90°$, det er negativ for at sørger 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 rotations er fundet afsted, kan den næste led betragtes som en plan robot da de kan beskrives fuldstændigt i et plan. Dette plan kan ses på Figur 7. Ved undersøgelse af dobotten fandt jeg ud af at når det andet led stod i nul stod den lodret op. Og den vinkel der forventes af transformations matrix, er fra x aksen i positiv omløbsretning. Der for beskrives led variable til den 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 først armdel af robotten, dette kan også ses på Figur 7. Frame 3 er først en forskydning langs x aksen dette er selve længden af ledet så af α_{3-1} er 150 mm. Rotation af denne frame er fast langt mekanisk i robotten, det vil sige at den kun afhænger af θ_2, og den sørger for at den altid er vandret. Dette opnået med at lave et parallelogram af 2 pinde op til dette punkt i robotten. Derfor skal der findes en vinkel der simulerer denne opførelse med dennavit-hartberg parameterne. For at bestemme hvad vinkel 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 de er parraller kan vinkel θ_2 overføreres til skæring med robot arm og linjen ifølge Euklid. Og for at den nye linje er parralle med x aksen, skal den stå 90 grader på y aksen. Ud fra Figur 6 kan det ses det 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 det 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, og da denne frame på samme mådes som frame 3 skal være parallelt med x-aksen, men her skal den bare være modsat den rotation som blev lavet i frame 4, så derfor $\theta_5 = -\theta_4$. Den sidste frame er ikke, lige som de 4 forrige, en plan så derfor skal roters den så den kommer til at stå som den oprindlige frame, det gøres ved at sætte $\alpha_{6-1} = 90°$. Og så har den en helt normal θ_6 som beskriver rotation af helt spidsen af roboten. De komplete dennavit-hartberg ses i Tabel 1



Figur 6 Udsnit af planmodel af dobot

Kommenteret [JR2]: https://en.wikipedia.org/wiki/Transversal_(geometry)



Figur 7 Model af planar 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 parameter for en dobot

Nu opstilles der en transformationsmatrix fra hver frame til dobotten udfra Denavit-Hartenberg parameterne og den matrix som beskriver transformation fra et frame til en anden. Så transformation 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 kan man få 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, ganges disse matricer samme for at for en matrix der beskriver

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

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

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

Inverse 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 koordinater sæt og en rotation af værktøjet. Er det nødvendigt at kunne gå den anden vej, det vil sige fra et positions matrix til 4 vinkler de 4 led, dette er også kaldet inverse 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 er ikke særlig mange rotationsmuligheder, fordi dobotten kun kan rotere om z aksen, da den 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 inverse kinematik problemer med algebra men i dobottens tilfælde er der det simplere at gøre det ved med geometrisk argumentation. Vinkel θ_1 er relativ simple da den ikke afhænger af andet en position af slutpunktet. Der kan tegnes en trekant hvor den en katete er P_x og den anden katete er P_y. Det kan ses på Figur 8**Fejl! Henvisningskilde ikke fundet.**. 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 den ønskede vinkel ϕ og θ_1 fordi θ_2 og θ_3 ikke har nogle betydning for vinkel på værktøjet. Da der er mekanisk lavet så dele kommer til at være vandret ret. For at beregnere θ_6, opstilles denne ligning som beskriver vinklen ϕ ud fra θ_1 og θ_6, hvor ϕ er vinkel af værktøjet på robotten. Det kan ses på Figur 8 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 8 Visualisering af θ_1 og θ_6

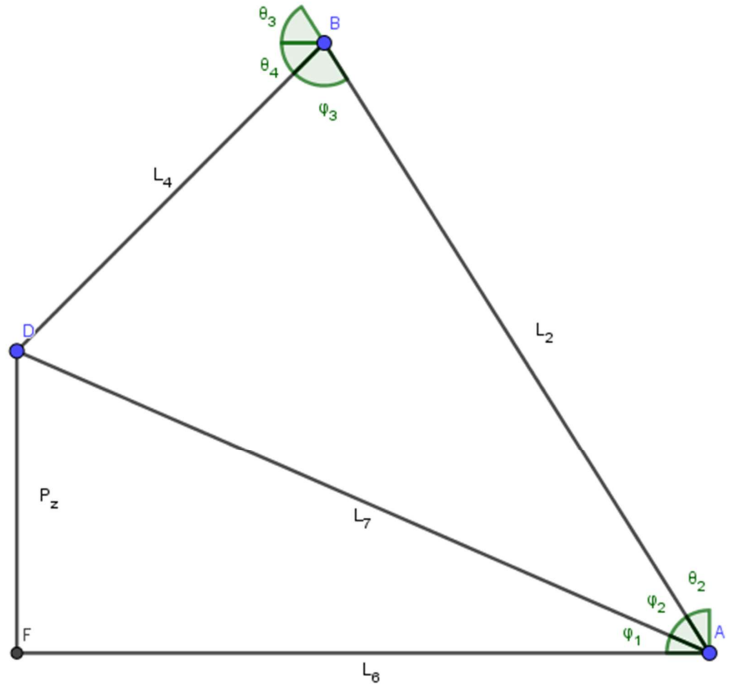
For at finde de sidste to vinkler ud fra den ønskede position, skal vi betragte den arm i planet for at forsimple det. Dette kan lade sig gøre fordi θ_2 og θ_4 kun bevæger sig om en akse. På Figur 9 er der sat punkter ind i alle frames position i planet. L_2 til L_5 er længder der er kendt ud fra udformning 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 8. Så L_1 findes ved

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

Dette kan ses på Figur 8.

Figur 9 Plan model for dobot

Den form der ses på Figur 9 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 trekanten, det vil sige at jeg samler det til én trekant. Med andre ord fjerner jeg den flade runde sektion, mellem punkt B og C, og sektion mellem D og E, når jeg fjerne disse sektioner, samler det hele sig for at jeg ikke for nogen huler i figuren. Denne ses på Figur 10. Jeg har også tilføjet et linjestykke fra A til D. Den gør at jeg har en lukket trekant ABD.



Figur 10 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 værdi i trekanten. Jeg kender L_2 og L_4 da de er konstante begge to givet ud fra robottens fysiske dimensioner, denne spisefikse 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 ret vinkelret trekant, hvor begge kateter er kendt så L_7 kan 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, og jeg kender alle side længderne i trekanten så på denne måde kan jeg finde.

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

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 værdi kendt der skal bruges til at finde θ_2 og θ_4.

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

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

$$\theta_3 = 90° - \theta_2$$

Ude fra Figur 10 ses det at

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

Jeg isoler 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 oven for sammen får man disse 9 ligninger som beskriver den inverse kinematik 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

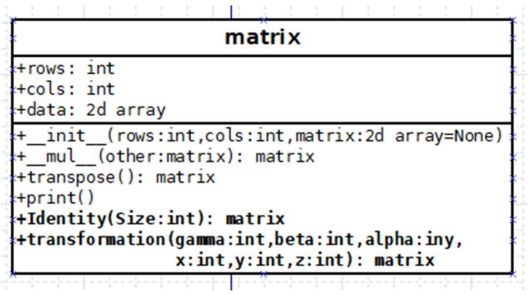
$$P = \begin{bmatrix} \cos(\phi) & -\sin(\phi) & 0 & P_x \\ \sin(\phi) & \cos(\phi) & 0 & P_y \\ 0 & 0 & 1 & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Analyse af Python programmer

Kommenteret [JR3]: Tjek alle linje nummer

Analyse af Python bibliotek til regning med matricer

For at lave et bibliotek til at regne med matricer, skal klassen have noget data om matricen, den skal kende størrelsen af matricen, det vil sige antallet at af rækker og kolonner, så skal den også opbevare selve data i matricen, det vil sige de tal der er i matrixen. Det er alt klassen opbevarer, dette kan ses på klassediagrammet på Figur 11. Data variable er et todimensionale liste, det betyder at det er en liste af lister. Dette stemmer godt overens med at man kan set en matrix som en vektor af vektor. Ud fra klassediagrammet på Figur 11, kan det ses at klassen 3 normale funktioner der kun kan kaldes med et instans af klassen nemlig __mul__, transpose og print. Ud over det er det en konstruktør funktion som laver objektet __init__. Til sidst er der de to der er mærket med fed, identity og transformation, fordi det 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 11 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 12 Konstruktør

Lad os starte med kigge på konstruktøren i klassen, det ses på linje 5 i Figur 12 at init metoden kan tage op til 3 parameter, den skal have antal af rækker og søjler, men behøves ikke at have en matrix fordi den har standard værdi, som er none, det betyder at den er tom. I andre sprog end Python har man mulighed for at lave flere konstruktør, så man kan lave en som tager et antal rækker og søljer, og så laver en tom matrix og så en anden der kun tager en 2d liste, også selv regner antal af rækker og søljer ud, men i Python er det nødvendigt at lave det i en konstruktør, som har nogle standard værdier. På linje 8 tjekkes der om den har fået en matrix, hvis den ikke har, initialiser den bare data til at være et 2d list fyldt med 0. Men hvis den har fået noget tjekker den på linje 11 om det er et 2d liste, hvis ikke raise den en exception, for at fortælle at programmøren at vedkommet har gjort noget forkert ved ikke at sende et 2d liste som argument til matrix. Men hvis det er en 2d liste så sætter den data variablen til at være det 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 13 Matrix multiplikation metode

For at implementere en funktion til at gange to matricer sammen bruges nøgleordet __mul__ på linje 17 på Figur 13, ved at kalde den 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 at det er muligt 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. I hvis sætning spørger dem om de ikke er det samme, og hvis de ikke er det sammen raiser den en exception, for at fortælle programmøren at de matricer vedkommet forsøgte at gange sammen ikke kan lade sig gøre. Nu ved programmet at det er muligt at finde produkt mellem de to matricer, så det forsætter ved at går igennem 2 løkker, og på denne måde regner 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 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 resultaterne matrix, og til sidst return funktion 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 14 Tranpose metode

Funktion der transponerer funktion, bruger 2 inline for løkker til at spejle matricen over diagonalen, dette kan ses på Figur 14 på linje 35.

Kommenteret [JR4]: Kan uddybes

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

Figur 15 Manglende invert metode

Invert funktion på Figur 15 er ikke implemtert i dette biblioteket, da det ikke helt simpelt at gøre, men det er den første funktion der skulle tilføjes, hvis biblioteket skulle udvides. Og da det ikke er nødvendigt i demonstrations programmet der arbejder med dobotten, har jeg valgt ud lade den forholdsvis væsentlige funktion fra biblioteket.

Kommenteret [JR5]: Genlæses muligvis omskrives

```
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 16 Print metode

Matrixklassen har en printfunktion som printe matricen til konsollen, dette bruges for at lettere kunne debugge koden. Funktionen fungerer ved at for hver rækker laver den string, som består at dataen fra den række, så printer denne string. For at udvide den string der arbejdes på bruges += operatoren som bare tilføjer den nye string til enden af variablen string, dette ses på Figur 16 op linje 41. Når der printes til konsollen laver den eet linjeskifte på

denne måde kommer være række på hver sin linje. På Figur 17 er der et eksempel på hvordan det kan ses 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 17 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 18 på linje 44 og 53. Når det er klasse metode, skal den alt modtage en reference til selve klassen som det første parameter i funktion, 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 16 på linje 37. I disse fabriksmetoder bruger ikke denne klasse reference, 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 havde det samme antal rækker som søjler. 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 18 Fabriks metoder til at lave identites matricer og transformations matricer

Transformations metode laver en homogen transformations matrix, ud fra 3 vinker der beskriver rotationen om 3 fast låse akser, først en rotation om x-aksen så y-aksen til sidst z-aksen, og translation med x, y, z. Jeg bruger matricen fundet i afsnit Rotation. På linje 55 til 57 som ses Figur 18, 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 19 Test af Python bibliotek

For at teste et bibliotek i Python, kan lave en hvis sætning som på linje 66 på Figur 19, til at tjekke om filen bliver kørt selvstædigt 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 den selv ved om testen er rigtig, men der skal programmøren tjekke, men her kunne det udvides med unit testning, hvor man tester være enkelt funktion, 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 20 Opsætning af program til styring af dobot

For at kunne styre robot gennem et Python program skal der først oprettes kontant til robot. Computeren og robotten snakker sammen gennem en helt almindelige seriel port. Det gøres som det første i programmet fra linje 6 til 17 på **Fejl! Henvisningskilde ikke fundet.**. Inden da importers alle biblioteker programmet skal bruges, 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 har beskriver den fysiske størrelse af robotten. Disse variabler har de samme navne som der blev brugt da, den inverse kinematik blev lavet. Det gør let at det bliver 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 værdi der sendes til den svar til vinkel til hver af det 4 led på robotten og j'et betyder at den skal bevæger sig fra et punkt til et andet bare ved at gå fra den ene vinkel til den for hver af motorerne. Det betyder man ikke kender hvilken vej robotten kommer hen til det punkt man ønsker. Mod sat kunne man sætte et L i stedet for j'et, men for at den kan gøre det skal dobotten lave inverse kinematik også giver det ikke mening jeg selv har lavet det.[3]

[3] (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 21 Funktion der gør brug af den inversekinematik

På Figur 21 ses funktionen moveToCord som får dobotten til at bevæge sig hen til et givent punkt, et koordinat sæt og en vinkel af værktøjets rotation. Linjerne 27 til 36 på Figur 21 svar en til en med de ligninger jeg fandt i afsnit Inverse kinematik, de står i samme rækkefølge som i afslutning af dette afsnit. Den eneste ting som er ændret, er at der bruges funktion 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 den ind et program og kører efter dem. På linje 38 sendes kommandoen til dobot til at få den til at bevæge sig hen til de udregnet vinkler så værktøjet på robotten kommer til at være på pointet 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 22 Forward kinematik funktion

Funktion get_pose der ses på Figur 22, får 4 vinkler fra robotten, også regner det om til en matrix der beskriver position, af robotten. For at gøre det bruger funktion den forward kinematik matrix jeg fandt i afsnittet Forward kinematik. Denne matrix er skrevet direkte inden på linjerne 50 til 53. For at kunne brug de trigonometriske funktioner om regnes der inden til radianer på linjerne 46 til 49.

Der er en hjælpe funktion til begge ovenstående funktioner, der gør dem lettere at bruge, på Figur 21 på linje 41 og 42 er der en funktion movetomatrix som har en matrix som parameter og finder den de relevante data fra matricen og sender dem videre til moveToCord funktion. Og på Figur 22 på linje 59 til 61, ses get_dobot_pos funktion, som bruger dobot biblioteket positions funktion til at for 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 dobot til at gøre hvad man vil havde 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 23 Test af dobot styrering

For at teste disse funktioner afsluttes programmet med en kort test. På Figur 23 på linjerne 64 til 74 testets matrixregningen ved at på linje 64 fortæller at den skal kører til 300, -100, 0 og vinkel på 50°. Derefter aflæser position af robot, dette returneres som en matrix, så på linje 68 laves en transformations matrix med fabriksmetoden. Disse to matricer ganges sammen og så flytter robotten hen til det nye punkt på linje 74. Det viser at der muligt af flytte dobotten i forhold til den nuværende position og rotation. Linjerne 77 til 83 kører dobotten fremad og tilbage i en lige linje mellem to punkter for at vise at inverse kinematik model 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 alt lave fremadrettet kinematek og den inverse kinematik model for dobotten, så nu er det relevant at undersøge hvor stor del af det arbejde er nødvendigt at programmøren laver. Der er flere måder at løse inverse kinematik problemet på, i denne opgave er der bruget en analytisk metode som finder en lukket form, så en 1 til 1 funktion som kan regne position af leddene ud fra mål positionen. Det er også muligt at bruge numeriske løsninger, hvor man bruger en iterativ metode, hvor man prøve nogle vinkler, og ser hvor tættet robotten er på mål position, ud fra denne information ændre man på vinklerne for at komme tættere på. De findes forskellige metoder til lave iterativ inverse kinematik, nogle er helt simple andre indeholder mere matematik for at forsimple optimeringsproblemet. Når man skal arbejde med at programmerer robotter er der flere til gange man kan lave alt matematikken selv, men det er meget tidskrævende og forholdsvist svært. Men man kan finde et biblioteker der er lavet til at løser inverse kinematik problemer, så skal man beskrive den robot der arbejdes med også vil den højst sandsynligt kunne løse det problemet for programmøren. Dette gøre det meget lettere at komme i gang, dette er især smart hvis det er hobby projekter hvor det ikke er nødvendigt med den matematiske forståelse som opnås ved at selv at lave det hele.[4] Det er også muligt at finde biblioteker som gør det letter at arbejde med matematikken, men som ikke er direkte er lavet til at løse inverse kinematik problemer, for eksempel kan man brug numPy til at regner med matricer med, og på denne måde ikke fokuser på hvordan lineær algebra virker men bare op hvordan den bruges.[5] Dobotten er lavet til undervisning så derfor har den indbygget en inverse kinematik model, så det er normalt ikke nødvendigt at tænke på den inverse kinematik del, det gør det muligt at fokuserer på andre problem stillinger, 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 inverse kinematik ikke interessant, og derfor er det smart den har løst kinematik problemet og så der plades

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

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

til at fokuserer på andre dele af problemløsningen.[6] 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 for en robot til at virke, de har en masse industri robotter som kan programmers let fra computeren også kan robotten med det samme gå i gang. De udtaler sig på deres blog at man ikke skal lave matematik hvis ikke det ikke er dig selv der har bygget robotten, for ellers har nogle andre lavet løst problemet og det er ikke nødvendigt at gøre igen.[7]

Konklusion

For at beskrive dobottens kinematiske struktur bruges denavit hartenberg notation, og da disse blev fundet kunne der opstille matricer for hvert led i robotten, disse ganges sammen for at få den fremadrettet kinematik, og for at lave inverse kinematik model, bruges forskellige trigonometriske formler, på denne måde kunne både den fremadrettet og inverse kinematik beskrives. Den inverse kinematik model er når den færdig meget kort og simpel, da det kan beskrive al matematikken med 9 ligninger. Det implieters i et Python program, med hjælp fra et Python bibliotek der kan regner med matricer, disse giver mulighed for at styr dobot, de få ligneringer der fundet matematikken gør det let at implmerntere det i Python programmet, og gør at det kan kører meget hurtigt da der ikke skal særlig mange beregninger til at uddragen position af leddene til en given mål position. Til sidst i diskussionen findes det ikke altid er nødvendigt at udvikle en komplette kinematik model, men det skal vurderes hvad er relevant for opgaven der skal løses, og hvis det er til læring hvilke lærings mål der for processen.

[6] (Dobot, n.d.)
[7] (Owen-Hill, 2021)

Referencer

Craig, J. J. (2005). *Introduction to Robotics mechanics and control* (3 ed.). Upper Saddle River, United States of America: Pearson Prentice Hall.

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

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

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

STEM EDUCATION WORKS. (n.d.). *DOBOT MAGICIAN LITE*. Retrieved December 12, 2022, from stemeducationworks: https://stemeducationworks.com/product/dobot-magician-lite/

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

Undervisningsministeriet. (2014, august 27). 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

Kommenteret [JR6]: Opdater bilaig

```
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)
```