TDAT3024 Prosjekt

Kristian Gulaker, Ole Jonas Liahagen, Max T. Schau, Martin J. Nilsen

Høst 2020

Innholdsfortegnelse

1 Forord

Vi ønsker å starte med å rette en stor takk til vår foreleser Hans Jakob Rivertz, førsteamanuensis ved Institutt for datateknologi og informatikk (IDI), Norges teknisk-naturvitenskapelige universitet. Han har bidratt med sin kunnskap om feltet, og alltid vært tilgjengelig dersom vi hadde spørsmål.
I tillegg ønsker vi å takke våre medstudenter for gode diskusjoner rundt de ulike oppgavene gitt i prosjektet.

2 Introduksjon

I denne rapporten følger det teori og resultater i vårt arbeid med oppgaver knyttet til rotasjon av et stivt legeme. Oppgavesettet bygger på fremstillinger i fysikken av Marion og Hornyak[1], samt matematikk fra Sauer[2], Helgasson[3], Munthe-Kaas[4] og Sophus Lie.

Vi startet med å se på fysikken bak et stivt legeme som roterer, noe Isaac Newton la grunnlaget for i 1666-67. Videre var det viktig å se på valg av koordinatsystem, i tillegg til å se på viktige konsepter som treghetsmoment, dreiemoment, massesenter, bevegelseslikninger for roterende koordinatsystem og energien til et roterende legeme. Vi har også arbeidet med ulike numeriske metoder som Eulers metode og Runge-Kutta RK4 på matriseform.

Innledningsvis starter vi med å beskrive teorien som ligger til grunn for arbeidet vi har gjort. Videre følger det et kapittel med resultater av de 6 oppgavene som ble gitt, før vi tolker disse resultene i det påfølgende kapittelet. Avslutningsvis har vi en referanseliste, vedlegg med kode vi har skrevet i Python og en erklæring om arbeidsinnsats fra alle gruppemedlemmene.

[1] Marion, J. B. & Hornyak, W. F. (1982). “Physics for Science and Engineering”.
[2] Sauer, T. (2014). ”Numerical Analysis”.
[3] Helgason, S. (1978). ”Differential Geometry, Lie Groups, and Symmetric Spaces” & ”Pure and Applied Mathematics”
[4] Munthe-Kaas, H. Z. & Zanna, A. (1993). ”Numerical integration of differential equations on homogeneous manifolds”.

3 Teori

3.1 Lie-Gruppe

Lie-Gruppe er en samling av gruppestrukturer som er oppkalt etter nordmannen Sophus Lie. Det spesielle med Lie-grupper er at elementene er organisert kontinuerlig slik at en Lie gruppe er en deriverbar mangfoldighet [7].

3.2 Ortogonale matriser

En ortogonal matrise er en kvadratisk matrise, altså der alle rader og kolonner har samme lengde, hvor dens transponerte er lik dens inverse.

$$X^T = X^{-1}$$

Med dette følger at matrisen multiplisert med sin transponerte er lik identitetsmatrisen

$$X \cdot X^T = I$$

3.3 Den spesielle ortogonale gruppen

Den spesielle ortogonale gruppen er en Lie-Gruppe, og blir ofte gjengitt som SO(3). Dette er en mengde med 3 x 3 matriser hvor alle matrisene er ortogonale, noe som videre gir at alle disse matrisene multiplisert med sin transponerte vil være lik identitetsmatrisen

$$SO(3) = \text{Alle 3 x 3 matriser X der } X^T X = Id_{3x3} \text{ og } \det X = 1$$

3.4 Global trunkeringsfeil

Dersom vi har en tilnærming w_i og en eksakt løsning på initialverdiproblemet $y_1 = y(t_i)$ vil den globale trunkeringsfeilen være gitt som differansen mellom de på formen

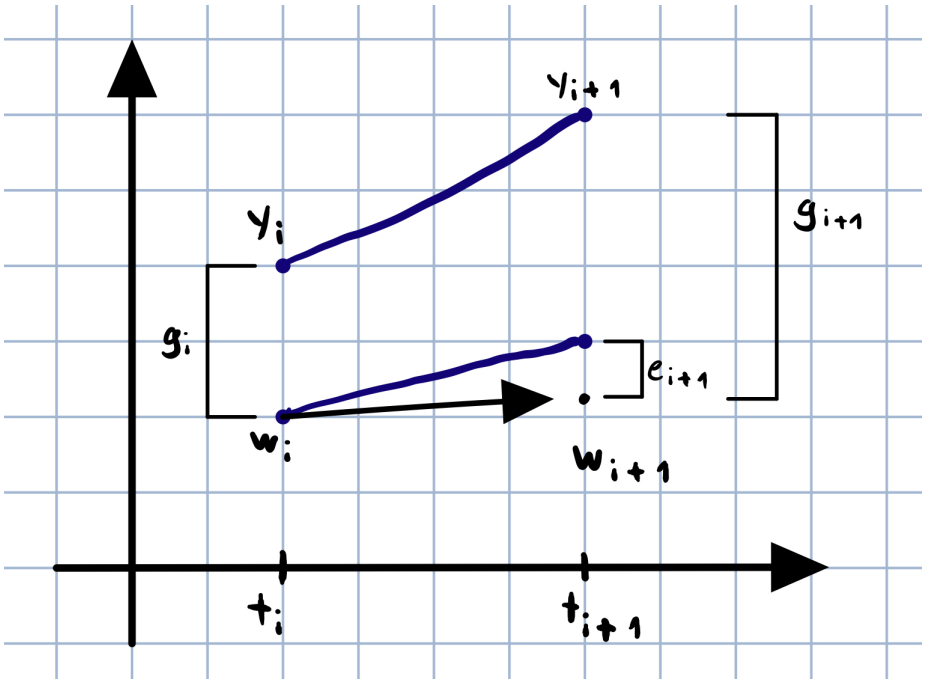
$$|g_i| = |w_i - y_i|$$

3.5 Lokal trunkeringsfeil

Den lokale trunkeringfeilen er den feilen som man får av ett steg i iterasjonen når man tilnærmer løsninger på differensiallikninger. Her vil man bruke den forrige tilnærmede løsningen som startpunkt. Den lokale trunkeringsfeilen er gitt ved:

$$|e_{i+1}] = |w_{i+1} - z(t_{i+1})|$$

der z(t) er den eksakte løsningen på initialverdiproblemet [1](s.294).



Figur 1: Illustrasjon av lokal og global trunkeringsfeil.

3.6 Numeriske metoder: Runge-Kutta

Runge-Kutta er en betegnelse på en familie av numeriske metoder som tilnærmer en løsning på differensiallikninger. Metodene inkluderer Euler og Trapesmetoden, i tilegg til andre metoder for å tilnærme differensiallikninger av høyere orden. Av disse finner vi blant annet Midtpunktsmetoden, Runge Kutta av fjerde orden (RK4) og Runge-Kutta-Fehlberg (RKF45). I løsningen av oppgavesettet har vi brukt Euler, RK4 og RKF45, så videre følger en beskrivelse av disse tre metodene.

3.6.1 Eulers metode

Eulers metode er en velkjent metode som brukes innen matematikken til å løse ordinære differensiallikinger numerisk. Metoden ble utviklet av Leonhard Euler på 1700-tallet, og brukes for å iterativt beregne en tilnærming til den eksakte

løsningen [6]. En starter med en oppgitt funksjon $y' = f(x, y)$ og tilhørende initialverdi $y(x_0) = y_0$. Deretter vil man iterativt tilnærme en løsning $w_i \approx y_i$ ved steg i på formen

$$\begin{aligned} w_0 &= y_0 \\ w_1 &= w_0 + hf(x_0, w_0) \\ w_2 &= w_1 + hf(x_1, w_1) \\ &\vdots \\ w_{i+1} &= w_i + hf(x_i, w_i) \end{aligned}$$

Når man nå skal anvende dette på matriseform settes $W_0 = X_0$ og regner videre ut $W_{i+1} = W_i + hW_i\Omega_i$ der Ω_i er på formen

$$\Omega = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}$$

Ω_i vil i hvert ledd bli dannet fra $\vec{\omega_i} = I^{-1}W_i^T\vec{L}$ ettersom vi ser at dens komponenter er å finne i $\vec{\omega_b} = [\omega_x \quad \omega_y \quad \omega_z]^T$. Denne formen for Euler utgjør (24) i prosjektheftet [4].

En av flere svakheter ved Eulers metode er at W_i ikke er ortogonal. Dette fikses ved å ta i bruk en annen versjon av Euler gitt i oppgaveheftet som likning 25. Forskjellen på denne varianten er at neste steg finnes ved $W_{i+1} = W_i\exp(h\Omega_i)$ isteden. Man må fortsatt regne ut $\vec{\omega_i}$ på samme måte som i likning 24, og for små h er det svært lite som skiller disse metodene. Det er altså ved høyere steglengder h man kan se forskjeller i nøyaktighet.

3.6.2 Runge-Kutta RK4

Man deler opp utregningen av et trinn i Runge-Kutta i fem steg, der fire av disse er mellomregninger før man regner ut W_i i femte steg. RK4 settes opp på

formen

$$\vec{\sigma_1} = I^{-1} W_i^T \vec{L}$$
$$\vec{\sigma_2} = I^{-1} \exp(-(h/2)\Sigma_1) W_i^T \vec{L}$$
$$\vec{\sigma_3} = I^{-1} \exp(-(h/2)\Sigma_2) W_i^T \vec{L}$$
$$\vec{\sigma_4} = I^{-1} \exp(-h\Sigma_3) W_i^T \vec{L}$$
$$W_{i+1} = W_i \exp(\frac{h}{6}(\Sigma_1 + 2\Sigma_2 + 2\Sigma_3 + \Sigma_4))$$

der Σ_i utregnes for hvert ledd fra $\vec{\sigma_i}$ på samme måte som Ω_i ble utregnet fra $\vec{\omega_i}$ i Euler. Dette gir Σ_i og $\vec{\sigma_i}$ på formen

$$\vec{\sigma} = [\sigma_x \quad \sigma_y \quad \sigma_z]^T$$

$$\Sigma = \begin{bmatrix} 0 & -\sigma_z & \sigma_y \\ \sigma_z & 0 & -\sigma_x \\ -\sigma_y & \sigma_x & 0 \end{bmatrix}$$

3.6.3 Runge-Kutta-Fehlberg

Runge-Kutta-Fehlberg, eller RKF45, er en variant som skal gi en bedre nøyaktighet enn RK4. Dette oppnår den ved blant annet å ta i bruk et "butcher table" (skjema) definert av Fehlberg og variabel steglengde.

Skjemaet er på formen $\begin{array}{c|c} \mathbf{c} & \mathrm{A} \\ \hline & \mathrm{B} \end{array}$ og er gitt ved

$$\begin{array}{c|cccccc}
0 & & & & & & \\
\frac{1}{4} & \frac{1}{4} & & & & & \\
\frac{3}{8} & \frac{3}{32} & \frac{9}{32} & & & & \\
\frac{12}{13} & \frac{1932}{2197} & -\frac{7200}{2197} & \frac{7296}{2197} & & & \\
1 & \frac{439}{216} & -8 & \frac{3680}{513} & -\frac{845}{4104} & & \\
\frac{1}{2} & -\frac{8}{27} & 2 & -\frac{3544}{2565} & \frac{1859}{4104} & -\frac{11}{40} & \\
\hline
 & \frac{25}{216} & 0 & \frac{1408}{2565} & \frac{2197}{4104} & -\frac{1}{5} & \\
 & \frac{16}{135} & 0 & \frac{6656}{12825} & \frac{28561}{56430} & -\frac{9}{50} & \frac{2}{55}
\end{array}$$

der den første raden av B (4. kvadrant) gir den 4.ordens metoden, og den andre raden gir den 5.ordens metoden.

Videre beregnes et steg av RKF45 på formen

$$\vec{\sigma_1} = I^{-1}W_i^T\vec{L}$$
$$\vec{\sigma_2} = I^{-1}\exp(-ha_{21}\Sigma_1)W_i^T\vec{L}$$
$$\vec{\sigma_3} = I^{-1}\exp(-h(a_{31}\Sigma_1 + a_{32}\Sigma_2))W_i^T\vec{L}$$
$$\vec{\sigma_4} = I^{-1}\exp(-h(a_{41}\Sigma_1 + a_{42}\Sigma_2 + a_{43}\Sigma_3))W_i^T\vec{L}$$
$$\vec{\sigma_5} = I^{-1}\exp(-h(a_{51}\Sigma_1 + a_{52}\Sigma_2 + a_{53}\Sigma_3 + a_{54}\Sigma_4))W_i^T\vec{L}$$
$$\vec{\sigma_6} = I^{-1}\exp(-h(a_{61}\Sigma_1 + a_{62}\Sigma_2 + a_{63}\Sigma_3 + a_{64}\Sigma_4 + a_{65}\Sigma_5))W_i^T\vec{L}$$
$$W_{i+1} = W_i\exp(h(b_{11}\Sigma_1 + b_{12}\Sigma_2 + b_{13}\Sigma_3 + b_{14}\Sigma_4 + b_{15}\Sigma_5 + b_{16}\Sigma_6))$$
$$Z_{i+1} = W_i\exp(h(b_{21}\Sigma_1 + b_{22}\Sigma_2 + b_{23}\Sigma_3 + b_{24}\Sigma_4 + b_{25}\Sigma_5 + b_{26}\Sigma_6))$$

der for eksempel a_{21} er elementet på rad 2 og kolonne 1 i A (skjema) og b kan bli funnet på samme måte. Σ_i regnes ut for hvert ledd fra $\vec{\sigma_i}$ på samme måte som i RK4. Dette gir Σ_i og $\vec{\sigma_i}$ på formen

$$\vec{\sigma} = [\sigma_x \quad \sigma_y \quad \sigma_z]^T$$

$$\Sigma = \begin{bmatrix} 0 & -\sigma_z & \sigma_y \\ \sigma_z & 0 & -\sigma_x \\ -\sigma_y & \sigma_x & 0 \end{bmatrix}$$

Videre må vi regne ut $\Delta W = W_{i+1} - Z_{i+1}$ og $E = \sqrt{\text{trace}(\Delta W^T \Delta W)}$ der trace er summen av diagonalelementene. E er da et tilnærmet mål på feilen i utregningen av W_{i+1} og er denne vi erstatter e_i med i (6.64) fra Sauer [1].

Likningen for den relative-feil-testen blir på matriseform gitt ved

$$\frac{E_i}{|W_i|} < T \tag{1}$$

der E er feilen, T er feiltoleransen (tol) og $|W|$ er normen av W.

For den variable steglengden brukte vi fremgangsmåten gitt for RKF45 i kapittel 6.5 i Sauer [1]. En må definere en relativ feiltoleranse T og en initiell steglengde h. Etter å ha fulført alle stegene over, utfører man relativ-feil-testen (1) og bestemmer hva man gjør basert på utfallet av denne. Dersom den er suksessfull, er svaret for dette steget Z_{i+1} og man går videre til neste steg. Hvis testen feiler, prøver man igjen med en ny h kalkulert utifra formelen

$$h = 0.8(\frac{T|W_i|}{E_i})^{\frac{1}{p+1}}h_i \tag{2}$$

med $p = 4$ som er orden av metoden som produserer W_i.

Med den nye h gjør man så alle stegene på nytt før man kjører testen på ny. Dersom den skulle feile igjen, noe som sjeldent skjer, halverer man steglengden og prøver på nytt frem til testen bestås.
Ved neste steg tar man igjen utgangspunkt i h fra forrige steg, og tar igjen i bruk (2) ved feilet test.

3.6.4 Feil i høyere ordens metoder

For metoder av orden p vil den lokale trunkeringsfeilen være gitt ved $O(h^{p+1})$. Det vil si at for RK4 vil den lokale trunkeringsfeilen være gitt ved $O(h^5)$ [8].
Det viser seg at metoder som løser ordinære differensiallikninger av høyere orden er foretrukket sammenlignet med metoder av lavere orden. Dette kommer som et resultat av at konvergensen, det vil si hvor fort den globale feilen til tilnærmingen går mot null, er langt bedre for høyere ordens metoder enn lavere. For eksempel for RK4, som er av fjerde orden, vil feilen, for hver halvering av steglengden h, minke med en faktor på $2^4 = 16$ [1](s.316)

3.7 Rotasjonsmekanikk

3.7.1 Energien til et roterende legeme

Dersom vi har fått oppgitt et legeme som roterer vil den kinetiske rotasjonsenergien være gitt ved

$$K = \frac{1}{2}\vec{L} \cdot \vec{\omega}$$

der $\vec{L}$ er dreiemomentet og $\vec{\omega}$ er rotasjonsvektoren

3.7.2 Treghetsmoment

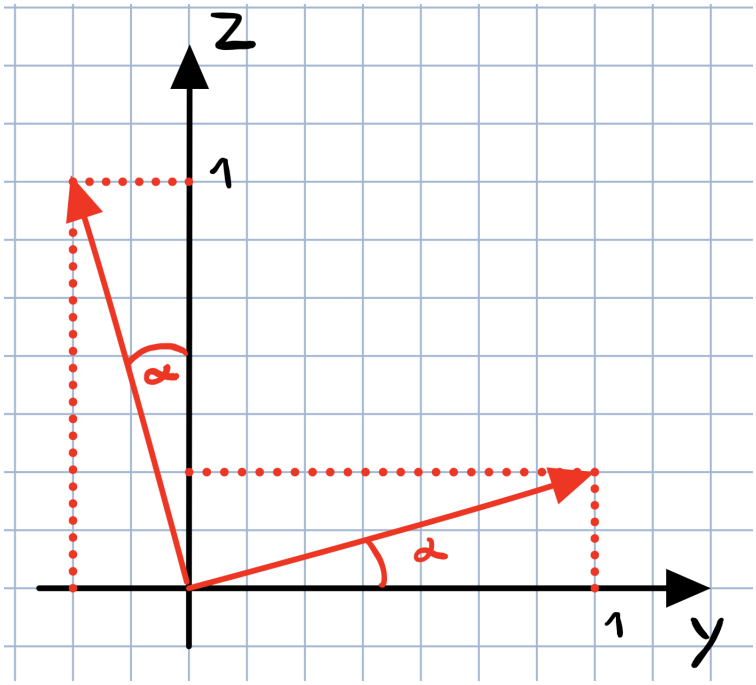
Når man har et legeme som roterer vil man kunne måle rotasjonstregheten. Denne rotasjonstregheten defineres i fysikken som treghetsmomentet I, med SI-enheten $kg \cdot m^2$. Rent praktisk vil treghetsmomentet si noe om hvor vanskelig det vil være å få et legeme som står i ro til å begynne å rotere om aksen, altså hvor "tregt" det er. Dersom man hadde hatt et større svinghjul med stort treghetsmoment ville dette implisert at en hadde trengt en større kraft for å få hjulet til å rotere, enn om dette hjulet var mindre/lettere og med et mindre treghetsmoment.

3.7.3 Rotasjonsmatrise

En rotasjonsmatrise er en matrise som definerer en rotasjon om et punkt, gjerne origo. I tre dimensjoner kan rotasjonen beskrives som en 3x3 matrise. En generell rotasjon kan uttrykkes som en matrise;

$$R = \begin{bmatrix} R_{x1} & R_{y1} & R_{z1} \\ R_{x2} & R_{y2} & R_{z2} \\ R_{x3} & R_{y3} & R_{z3} \end{bmatrix}$$

Denne er satt sammen av de tre rotasjonsmatrisene for hver dimensjon x, y, z. Ved å se på hver parkombinasjon av akser i koordinatsystemet som hvert sitt todimensjonale koordinatsystem kan vi utlede en rotasjonsmatrise om hver av aksene i tre dimensjoner. Vi kan bruke en rotasjon om x-aksen som eksempel.



Figur 2: Rotasjon om x-aksen. X-aksen peker ut av skjermen og positiv rotasjonsretning er definert som mot klokka.

Her vil alle x-verdier forbli de samme, mens y- og z-verdier endres. Samtidig vil x- y- og z-aksene fortsatt stå vinkelrett på hverandre. Dette fører til at R_{x1} må være lik 1 , slik at en vektor vi multipliserer med matrisen ikke skal få endret x-verdiene sine. Videre må R_{x2} og R_{x3} være 0 ettersom vektorproduktet av

x- y- og z-aksene skal være lik 0. Dette impliserer at de er innbyrdes ortogonale. Deretter må vi sette $R_{y1} = R_{z1} = 0$ slik at x-verdien til resultantvektoren fra matrisemultiplikasjonen ikke endres. Til slutt regner vi ut de nye y- og z-verdiene ved hjelp av illustrasjonen over. Vi ser at om vi roterer koordinatsystemet med en vinkel α kan vi uttrykke denne forflyttelsen som en rettvinklet trekant hvor høyden og grunnlinjens lengde er de nye y- og z-verdiene. Vi kan starte med y-aksens nye koordinater etter rotasjonen.
Av figuren ser vi at de nye verdiene vil være gitt ved $y = cos(\alpha), z = sin(\alpha)$. På samme måte regner vi ut for z-aksen; $y = -sin(\alpha), z = cos(\alpha)$. Setter vi alt dette inn i matrisa får vi rotasjonsmatrisa

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

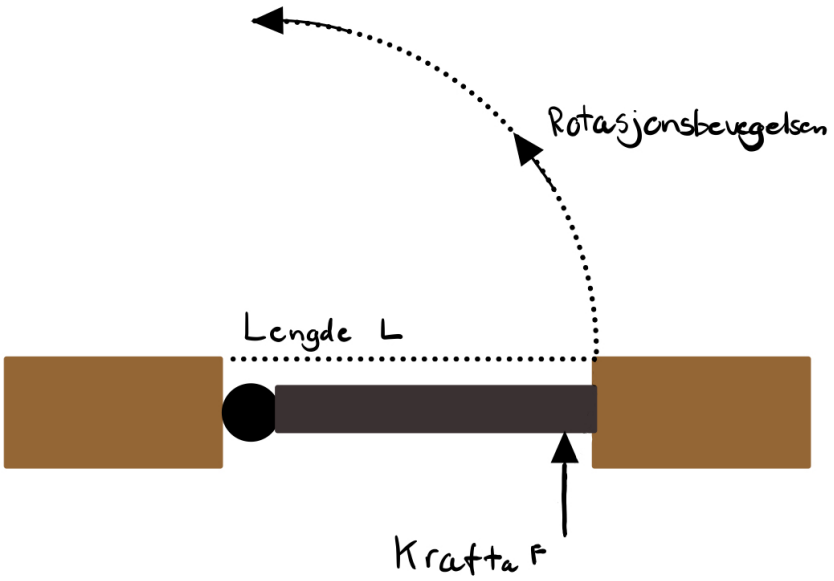
På samme måte kan man regne ut rotasjonsmatrisa for de resterende to aksene, og bruke matrisemultiplikasjon for å danne ulike generelle rotasjonsmatriser. Disse blir ulike siden matrisemultiplikasjon ikke er kommutativt.

3.7.4 Dreiemoment

Dreiemomentet har symbolet τ, og er relatert til kraften som foråsaker rotasjonsbevegelsen. Helt konkret vil det beskrive en krafts evne til å forårsake eller endre rotasjonsbevegelsen til et legeme om sin egen akse [2](s. 303). Videre er dreiemomentet τ gitt ved

$$\tau = L \times F$$

der L er vektoren fra rotasjonsenteret til der hvor kraften virker. F er kraften som virker på legemet.



Figur 3: Illustrasjon av dreiemoment

3.8 Euler-vinkler i en 3x3 rotasjonsmatrise

En rotasjonsmatrise med form 3x3 kan beskrives som en sum av 3 grunnleggende rotasjonsmatriser, når R er definert som:

$$R = \begin{bmatrix} R11 & R12 & R13 \\ R21 & R22 & R23 \\ R31 & R32 & R33 \end{bmatrix}$$

En rotasjon Rx på ψ radianer rundt x-aksen, beskrevet som:

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

En rotasjon Ry på θ radianer rundt y-aksen, beskrevet som:

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

En rotasjon Rz på ϕ radianer rundt z-aksen, skrevet som:

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

Her blir da rotasjonsmatrisen R = Rz(ϕ)Ry(θ)Rx(ψ).
Man kaller ψ, θ, ϕ for Euler-vinklene til rotasjonsmatrisen R. Ut i fra disse grunnleggende matrisene, kan man regne ut at Euler-vinklene blir: [3]:

$\theta = -\arcsin(R31)$

$\psi = atan2(\frac{R32}{cos(\theta)}, \frac{R33}{cos(\theta)})$

$\phi = atan2(\frac{R21}{cos(\theta)}, \frac{R11}{cos(\theta)})$

4 Resultater

4.1 Oppgave 1

4.1.1 Oppgave a

I denne oppgaven skulle vi definere eksponentfunksjonen

$$\exp(h\Omega) = \mathrm{I} + (1 - \cos\omega h)\frac{\Omega^2}{\omega^2} + (\sin\omega h)\frac{\Omega}{\omega}$$

Input til funksjonen vil være steglengden h og matrisen Ω gitt ved

$$\Omega = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}$$

Ut fra matrisen Ω kan vi hente ut verdiene ω_x, ω_y, ω_z slik at vi kan finne

$$\omega = |\vec{\omega}| = \sqrt{\omega_x^2 + \omega_y^2 + \omega_z^2}$$

Deretter definerer vi identitetsmatrisen I

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

Da har man funnet alle ukjente i ligningen, og setter inn verdiene.
I oppg1.py kjører vi 10 000 iterasjoner hvor vi tester for ulike verdier for Ω:

```
h = 0.001
    for i in range(10000):
        random_rotation_matrix = np.random.randint(1000, size=(1, 3))
        if not test_exponent_function(random_rotation_matrix[0], h):
            print("Test did not succeed")
            break
    print("Test did succeed")
```

Selve testen er programmert slik at den tar høyde for at elementene ikke er nøyaktig like, men har en relativ toleranse på 1e-05 og en absolutt toleranse på 1e-08:

```
identity_matrix = np.identity(3)

    ω_x = ω[0]
    ω_y = ω[1]
    ω_z = ω[2]

    Ω = np.array(
        [[0, -ω_z, ω_y],
         [ω_z, 0, -ω_x],
         [-ω_y, ω_x, 0]]
    )
    X_approx = exponent_function(h, Ω)
    should_be_identity_matrix = np.dot(np.transpose(X_approx), X_approx)
    return np.allclose(should_be_identity_matrix, identity_matrix)
```

For å illustrere hvordan vi tester setter vi steglengden $h = 0.001$, $\omega_x = 24$, $\omega_y = 85$, $\omega_z = 29$ slik at vi får at

$$\Omega = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} = \begin{bmatrix} 0 & -29 & 85 \\ 29 & 0 & -24 \\ -85 & 24 & 0 \end{bmatrix}$$

$\exp(\Omega h)$ gir oss matrisen

$$\begin{bmatrix} 0.94290715 & -0.0567886 & 0.32817855 \\ 0.05758372 & 0.99831397 & 0.0073032 \\ -0.32803997 & 0.0120115 & 0.94458748 \end{bmatrix}$$

Deretter skal vi vise at formelen $\exp(h\Omega)$ tilfredstiller likningen $\exp(h\Omega)\exp(h\Omega)^T = X \cdot X^t = I_{d3x3}$. Dette er da et element i den spesielle ortogonale gruppen (SO3).

$$\exp(h\Omega)\exp(h\Omega)^T$$

$$= \begin{bmatrix} 0.94290715 & -0.0567886 & 0.32817855 \\ 0.05758372 & 0.99831397 & 0.0073032 \\ -0.32803997 & 0.0120115 & 0.94458748 \end{bmatrix} \begin{bmatrix} 0.94290715 & 0.05758372 & -0.32803997 \\ -0.0567886 & 0.99831397 & 0.0120115 \\ 0.32817855 & 0.0073032 & 0.94458748 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & -6.580e-18 & 3.270e-17 \\ -6.580e-18 & 1 & 4.677e-17 \\ 3.270e-17 & 4.677e-17 & 1 \end{bmatrix}$$

Vi ser at vi får en matrise med veldig små tall som vi kan tilnærme lik identitets-matrisen. Til sammenligning er maskin-epsilon (double precision, binary64) $\approx$

2.22e-16 [5].

$$\approx \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

4.1.2 Oppgave b

Viser til oppg1.py for implementasjonen av funksjonen.

4.1.3 Oppgave c

I oppgaven skulle vi regne ut treghetsmomentet til T-nøkkelen gitt ved:

$$I_{xx} = \frac{M_1 R_1^2}{4} + \frac{M_1 L_1^2}{12} + \frac{M_2 R_2^2}{2}$$

$$I_{yy} = M_1 R_1^2 + \frac{M_2 L_2^2}{4} + \frac{M_1 R_1^2}{2} + \frac{M_2 R_2^2}{4} + \frac{M_2 L_2^2}{12}$$

$$I_{zz} = M_1 R_1^2 + \frac{M_2 L_2^2}{4} + \frac{M_1 R_1^2}{4} + \frac{M_1 L_1^2}{12} + \frac{M_2 R_2^2}{4} + \frac{M_2 L_2^2}{12}$$

$$\text{og } I_{xy} = I_{xz} = I_{yz} = 0$$

For T-nøkkelen fikk vi oppgitt følgende verdier:

- $L_1 = 8.0\text{cm} = 0.08\text{m}$
- $R_1 = 1.0\text{cm} = 0.01\text{m}$
- $L_2 = 4.0\text{cm} = 0.04\text{m}$
- $R_2 = 1.0\text{cm} = 0.01\text{m}$
- $\rho = 6.7 \frac{\text{g}}{\text{cm}^3}$

Massesenteret til håndtaket ligger i punktet $x = -1.0cm$ på x-aksen, mens massesenteret til sylinderet ligger i punktet $x = 2.0cm$ på x-aksen. I tillegg ligger massesenteret til hele T-nøkkelen i origo.

Volumet til et sylinder er gitt ved $V = \pi \cdot r^2 \cdot h$
Bruker denne formelen til å finne volumet for de to sylindrene:

$V_{\text{håndtak}} = \pi \cdot R_1^2 \cdot L_1 = \pi \cdot (1.0\text{cm})^2 \cdot 8.0\text{cm} = 8\pi\text{cm}^3$

$V_{\text{sylinder}} = \pi \cdot R_2^2 \cdot L_2 = \pi \cdot (1.0\text{cm})^2 \cdot 4.0\text{cm} = 4\pi\text{cm}^3$

Deretter kan vi finne ut massen til de to:

$$m_1 = m_{\text{håndtak}} = V_{\text{håndtak}} \cdot \rho = 8\pi\text{cm}^3 \cdot 6.7\tfrac{\text{g}}{\text{cm}^3} = \tfrac{268}{5}\pi \text{ g} = 0.1683893662\text{kg}$$

$$m_2 = m_{\text{sylinder}} = V_{\text{sylinder}} \cdot \rho = 4\pi\text{cm}^3 \cdot 6.7\tfrac{\text{g}}{\text{cm}^3} = \tfrac{134}{5}\pi \text{ g} = 0.08419468312\text{kg}$$

Når vi har funnet ut massen kan vi bruke likningene for treghetsmomentet til nøkkelen direkte:

$$\begin{aligned} I_{xx} &= \frac{0.1683893662\text{kg} \cdot (0.01\text{m})^2}{4} + \frac{0.1683893662\text{kg} \cdot (0.08\text{m})^2}{12} \\ &\quad + \frac{0.08419468312\text{kg} \cdot (0.01\text{m})^2}{2} \\ &= 9.82271303e-05 \text{ kgm}^2 \end{aligned}$$

$$\begin{aligned} I_{yy} &= 0.1683893662\text{kg} \cdot (0.01\text{m})^2 + \frac{0.08419468312\text{kg} \cdot (0.04\text{m})^2}{4} \\ &\quad + \frac{0.1683893662\text{kg} \cdot (0.01\text{m})^2}{2} + \frac{0.08419468312\text{kg} \cdot (0.01\text{m})^2}{4} \\ &\quad + \frac{0.08419468312\text{kg} \cdot (0.04\text{m})^2}{12} \\ &= 7.22671030e-05 \text{ kgm}^2 \end{aligned}$$

$$\begin{aligned} I_{zz} &= 0.1683893662\text{kg} \cdot (0.01\text{m})^2 + \frac{0.08419468312\text{kg} \cdot (0.04\text{m})^2}{4} \\ &\quad + \frac{0.1683893662\text{kg} \cdot (0.01\text{m})^2}{4} + \frac{0.1683893662\text{kg} \cdot (0.08\text{m})^2}{12} \\ &\quad + \frac{0.08419468312\text{kg} \cdots (0.01\text{m})^2}{4} + \frac{0.08419468312\text{kg} \cdot (0.04\text{m})^2}{12} \\ &= 1.57865031e-04 \text{ kgm}^2 \end{aligned}$$

Dermed kan vi konkludere med at treghetsmomentet til T-nøkkelen er gitt ved:

$$\begin{bmatrix} 9.82271303e-05 & 0 & 0 \\ 0 & 7.22671030e-05 & 0 \\ 0 & 0 & 1.57865031e-04 \end{bmatrix}$$

hvor enheten for hvert element i matrisen er kgm^2

4.2 Oppgave 2

Vi fikk oppgitt spesialtilellet hvor legemet vi ser på er en kule og treghetsmomentet I er lik identitetsmatrisen

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

Startbetingelsen til X(T) er også lik identitetsmatrisen

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

I tillegg til at dreiemomentet $\vec{L}$ er

$$(1, 0, 0)$$

Vi skal så løse likningen (19) og (20) eksakt.
For likning (19) kan vi sette opp verdiene direkte inn i formelen

$$\vec{\omega_b} = (XI)^{-1}\vec{L}$$
$$= \left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \right)^{-1} \cdot \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$
$$= \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$

Likning (20) sier at

$$\frac{dX}{dt} = F(X) = X\Omega$$

Fra likning (19) har vi at $\vec{\omega_b} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$, det vil si at $\omega_x = 1$, og $\omega_y = \omega_z = 0$.
Vi setter inn disse i matrisen Ω:

$$\Omega = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$$

X er definert som matrisen som inneholder posisjonen til aksene av koordinatsystemet som roterer med legemet. Denne kan skrives slik:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}$$

ved å bruke likning (20) og at $\vec{\omega_b} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$, kan vi dermed skrive X' slik:

$$X' = X\Omega = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & x_{13} & -x_{12} \\ 0 & x_{23} & -x_{22} \\ 0 & x_{33} & -x_{32} \end{bmatrix}$$

Dette gir oss videre:

$$\begin{bmatrix} x'_{11} & x'_{12} & x'_{13} \\ x'_{21} & x'_{22} & x'_{23} \\ x'_{31} & x'_{32} & x'_{33} \end{bmatrix} = \begin{bmatrix} 0 & x_{13} & -x_{12} \\ 0 & x_{23} & -x_{22} \\ 0 & x_{33} & -x_{32} \end{bmatrix}$$

Fra definisjonen av X, vet vi at denne gir oss posisjonen til aksene til koordinatsystemet. Den er nemlig satt sammen slik:

$$X = [\hat{i} \;\; \hat{j} \;\; \hat{k}],$$

hvor $\hat{i}$, $\hat{j}$ *og* $\hat{k}$ er enhetsvektorene som har samme retning som aksene i koordinatsystemet. Disse må dermed være ortogonale. Det vil si at prikkproduktet dem imellom må være 0. Dette gir oss at X kan skrives annerledes og som en konsekvens kan også X' skrives om slik:

$$X = \begin{bmatrix} 1 & 0 & 0 \\ 0 & x_{22} & x_{23} \\ 0 & x_{32} & x_{33} \end{bmatrix} \implies X' = \begin{bmatrix} 0 & 0 & 0 \\ 0 & x_{23} & -x_{22} \\ 0 & x_{33} & -x_{32} \end{bmatrix},$$

der x_{11} må være 1 ettersom vi roterer om x-aksen og vi derfor må bevare alle koordinater som befinner seg på denne aksen.

Vi står igjen med 4 differensiallikninger:

1. $x'_{22} = x_{23}$
2. $x'_{23} = -x_{22}$
3. $x'_{32} = x_{33}$
4. $x'_{33} = -x_{32}$

Av disse likningene ser vi at x_{22} henger sammen med x_{23} på samme måte som x_{32} henger sammen med x_{33}. Trenger altså bare løse for et av parene, med en liten forskjell som vi kommer tilbake til. Løser vi differensiallikningene for x_{22} og x_{23} kommer vi frem til to uttrykk.

$$x'_{22} = x_{23},\ x'_{23} = -x_{22} \implies x''_{22} = -x_{22},\ x''_{23} = -x_{23}$$
$$x''_{22} + x_{22} = 0 \implies x_{22}(t) = Asin(t) + Bcos(t)$$

Benytter så initialverdien til X:

$$X(0) = I_{d3x3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

setter inn for x_{22} og x_{23} og løser for A og B:

$$x_{22}(0) = 1 = Asin(0) + Bcos(0) = Bcos(0) \implies B = 1$$
$$x'_{22}(0) = 0 = x_{23}(0) = Acos(0) - Bsin(0) \implies A = 0$$

Setter deretter inn verdiene for A og B:

$$x_{22} = 0 * sin(t) + 1 * cos(t) = cos(t)$$
$$x_{23} = 0 * cos(t) - 1 * sin(t) = -sin(t)$$

På samme måte som over kan vi løse for x_{32} og x_{33}. Forskjellen vi nevnte tidligere er den at i dette tilfellet vil det være den deriverte som er lik 1 i steden for 0:

$$x_{32} = 0 = Asin(0) + Bcos(0) = Bcos(0) \implies B = 0$$
$$x'_{32} = Acos(0) - Bsin(0) = 1 \implies A = 1$$

Setter inn verdiene for A og B:

$$x_{32} = 1 * sin(t) + 0 * cos(t) = sin(t)$$
$$x_{33} = 1 * cos(t) - 0 * sin(t) = cos(t)$$

Setter vi alt sammen, får vi vår endelige eksakte løsning for X(t):

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

4.3 Oppgave 3

I denne oppgaven skulle vi implementere varianten av Eulers metode gitt i likning (25) i avsnitt 4.1.

Vi startet med å definere en metode som tar inn steglengde h, et intervall, X_0, treghetsmomentet I og dreiemomentet L. Denne metoden skal returnere en liste med de tilnærmede posisjonene til aksene for en gitt tid.

Videre må vi danne Ω_i for hvert ledd. Vi har at komponentene i Ω_i er å finne i

$$\vec{\omega_b} = [\omega_x \quad \omega_y \quad \omega_z]^T$$

Og vi kan regne ut $\vec{\omega_i}$ for hvert ledd, gitt ved likningen

$$\vec{\omega_i} = I^{-1} W_i^T \vec{L}$$

Vi har Ω_i på formen

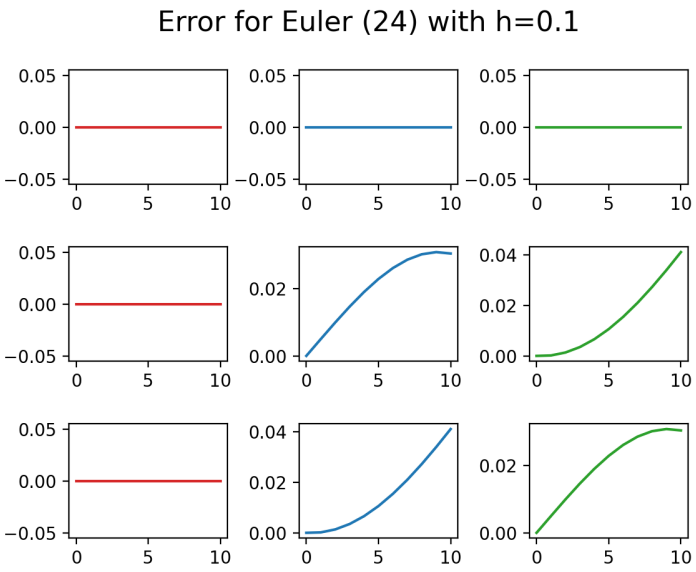
$$\Omega = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}$$

og vi kan starte med Eulers metode. Eulers metode er definert i likning (25) som

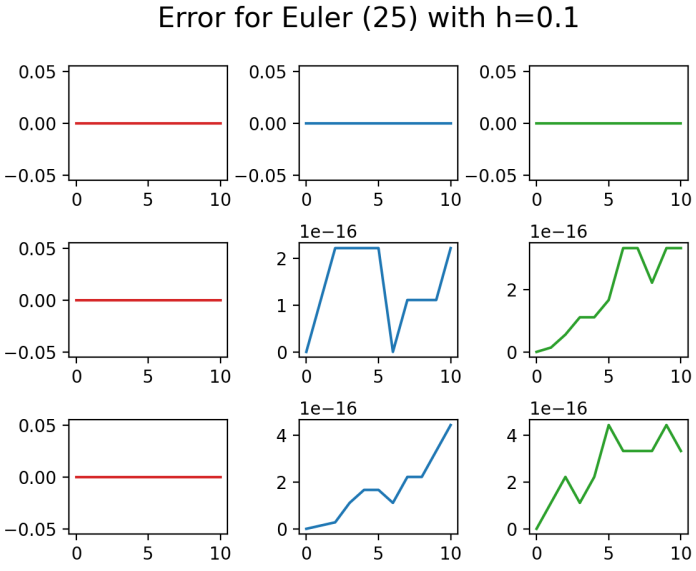
$$W_0 = X_0$$
$$W_{i+1} = W_i \exp(h\Omega_i)$$

der vi bruker eksponentfunksjonen vi lagde i oppgave 1.

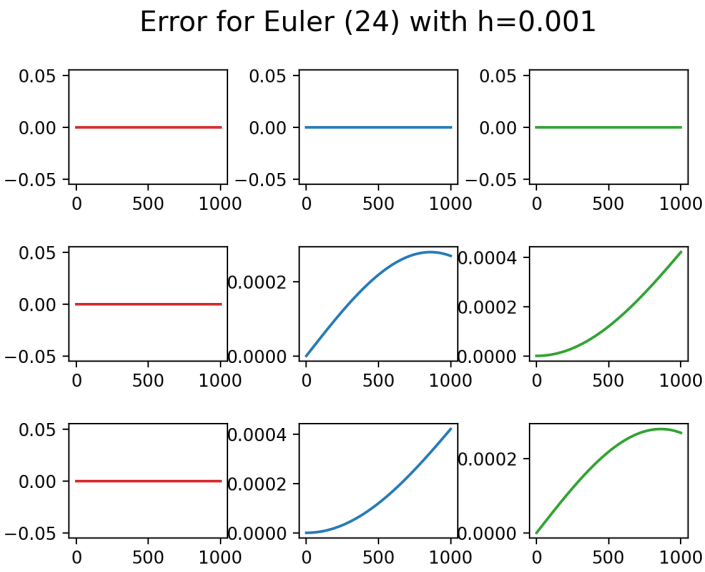
For å teste metoden brukte vi spesialtilfellet hvor legemet vi ser på er en kule, gitt i oppgave 2. Da har vi at I og X(T) er lik identitetsmatrisen, og at L er lik vektoren (1, 0, 0). Vi tester Eulermetoden med steglengde $h = 0.1$ på intervallet [0, 1], og observerer en relativt konstant endring av rotasjon. Vi bestemte oss så for å implementere Eulermetoden i likning (24) for å sammenlikne disse to. Her ser vi at denne blir mindre nøyaktig, og vi bestemte oss for å undersøke forskjellen litt nøyere. De neste sidene viser plottinger av feilen med ulik steglengde h.



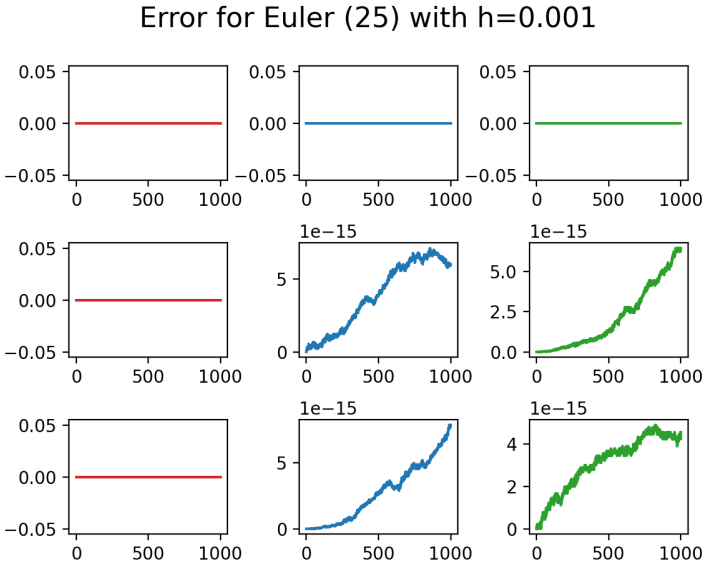
Figur 4: Elementvis feil i Euler (24) med h=0.1



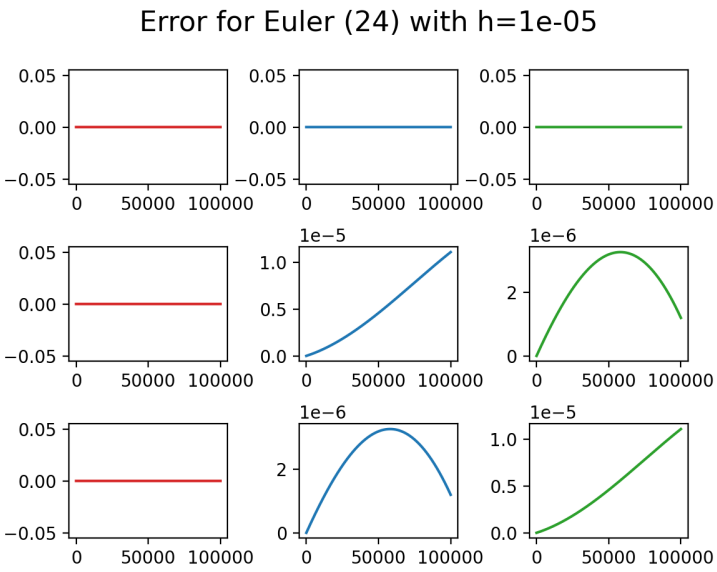
Figur 5: Elementvis feil i Euler (25) med h=0.1



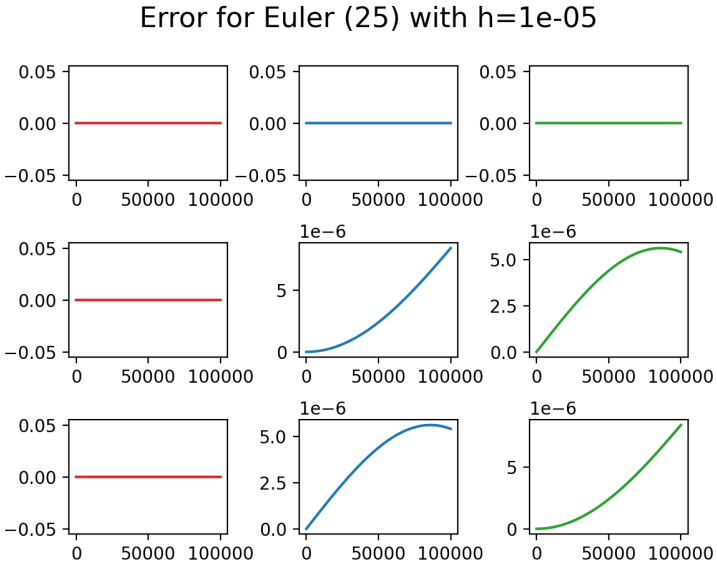
Figur 6: Elementvis feil i Euler (24) med h=0.001



Figur 7: Elementvis feil i Euler (25) med h=0.001

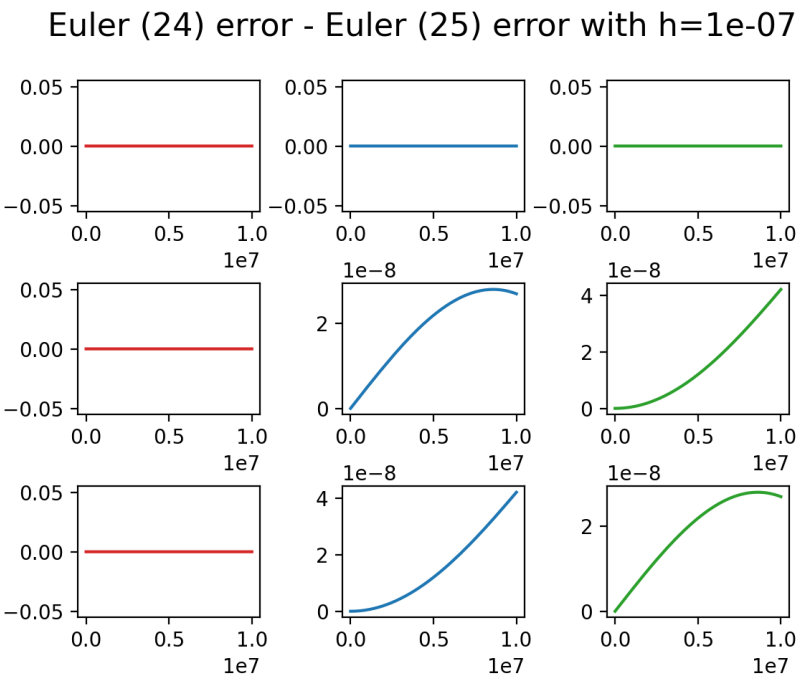


Figur 8: Elementvis feil i Euler (25) med h=0.00001



Figur 9: Elementvis feil i Euler (25) med h=0.00001

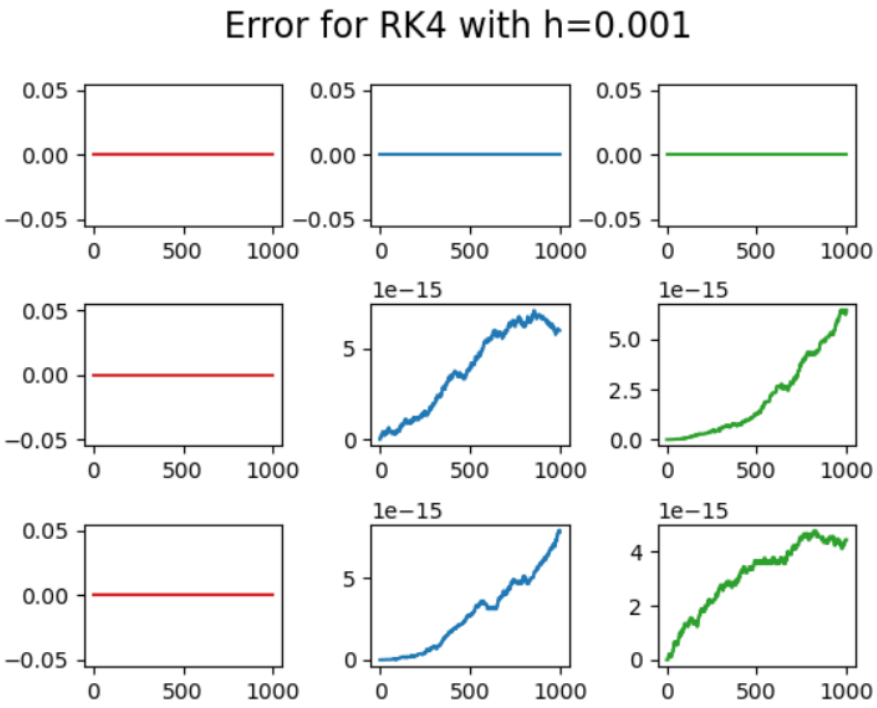
Av de to variantene vi implementerte kan vi gjøre to interessante observasjoner. For det første ser vi at feilen er desidert minst på likning (25) kontra likning (24) for steglengde $h = 0.1$ og $h = 0.001$. Men her kommer observasjon to inn i bildet. Dersom vi minsker steglengden h enda mer ser vi et mønster; jo mindre h blir, jo mindre blir forskjellen i nøyaktighet mellom (24) og (25)! Dette kan vi illustrere ved å plotte forskjellen mellom de to metodene ved en lav steglengde, som for eksempel ved $h = 0.0000001$:



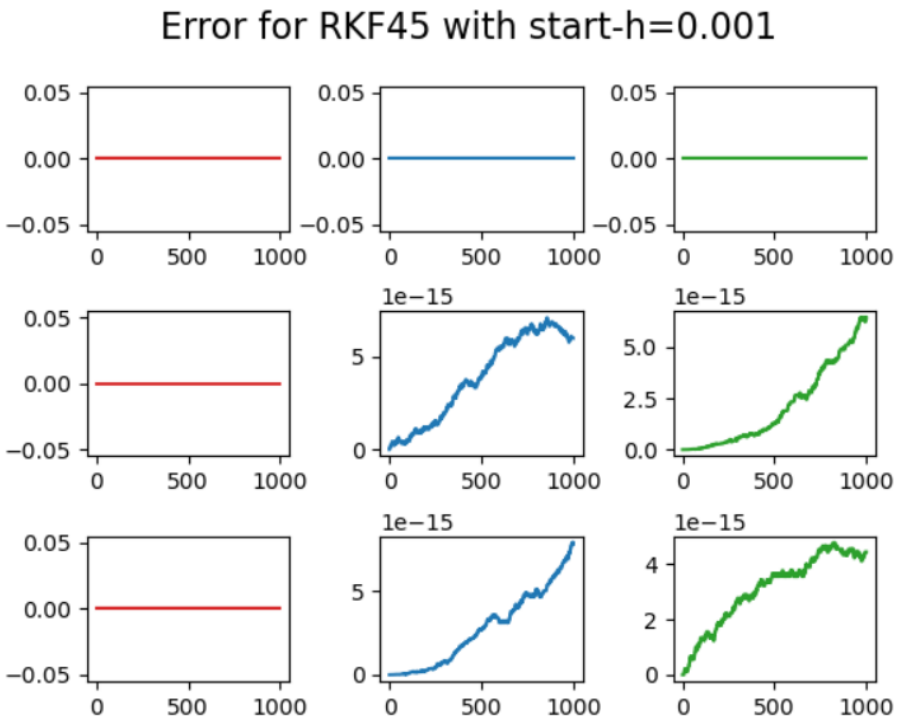
Figur 10: Elementvis forskjell på Euler (24) og Euler (25) med h=0.0000001

4.4 Oppgave 4

I oppgave 4 valgte vi å implementere både Runge-kutta 4 (RK4) og Runge-Kutta-Fehlberg (RKF45). Implementasjonen baserte seg på metodene gitt i prosjektoppgaven, som beskrevet i teorikapittelet. Det eneste unntaket er utregningen av ny steglengde h i RKF45 som er hentet fra Sauer [1] kapittel 6.5.1 likning 6.57 og kapittel 6.5.2. For å illustrere nøyaktigheten av de to, sammenlikner vi dem mot den eksakte løsningen med samme startverdier som i oppgave 3. Dette gir oss følgende grafer over feilen:

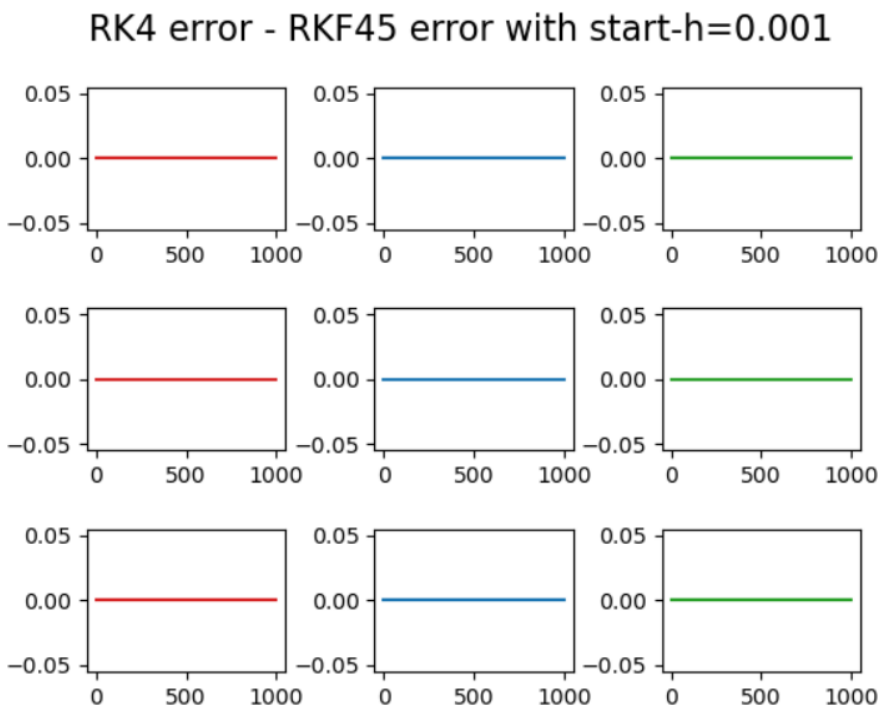


Figur 11: Elementvis feil i RK4 med h=0.001



Figur 12: Elementvis feil i RKF45 med h=0.001

Som vi ser er det vanskelig å skille mellom de to ved å bare se på disse grafene. I dette tilfellet skyldes det at de faktisk er identiske. Under har vi sammenlignet feilen for de to mot hverandre og vi ser at det stemmer:



Figur 13: Elementvis feil i RK4 vs. RKF45 med h=0.001

4.5 Oppgave 5

Fra oppgave 4 implementerte vi RK4 og RKF45. Dette dannet utgangspunktet for det som skulle gjøres i denne oppgaven. I oppgaveteksten blir det gitt at vi skulle bruke treghetsmomentet til en T-nøkkel som vi regnet ut i oppgave 1c:

$$\begin{bmatrix} 9.82271303e-05 & 0 & 0 \\ 0 & 7.22671030e-05 & 0 \\ 0 & 0 & 1.57865031e-04 \end{bmatrix}$$

Vi får oppgitt at X(0) er gitt ved Id_{3x3}:

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

Vi skal løse systemet ved å teste for tre ulike verdier for vinkelfarten $\vec{\omega}$. I første omgang tester vi med den implementerte RK4-metoden. Vi setter $h = 0.01$ på intervallet [0,1], og tester for

$$\vec{\omega}(0) = \begin{bmatrix} 1 & 0.05 & 0 \end{bmatrix}^T,$$
$$\vec{\omega}(0) = \begin{bmatrix} 0 & 1 & 0.05 \end{bmatrix}^T \text{ og}$$
$$\vec{\omega}(0) = \begin{bmatrix} 0.05 & 0 & 1 \end{bmatrix}^T$$

Vi kommer nærmere inn på forklaringen av resultatene i oppgave 6.

4.6 Oppgave 6

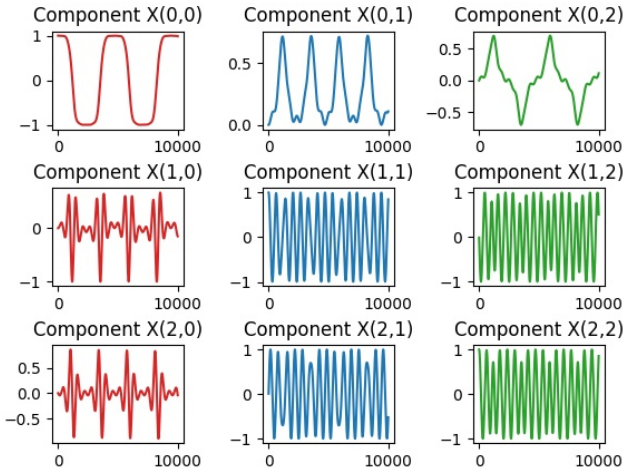
Vi skal se på komponentene til løsning X som en funksjon av tiden t. Dette gjør vi ved å finne resultatene fra RK4-metoden som ble funnet i oppgave 5, og deretter visualisere de grafisk. Her velger vi å se på komponentene tilknyttet eulervinklene[3] for x-, y-, og z-aksen, for å drøfte hvordan bevegelsen til t-nøkkelen vil se ut i følge disse grafene.

$$\theta = -\arcsin(R31)$$
$$\psi = atan2(\frac{R32}{cos(\theta)}, \frac{R33}{cos(\theta)})$$
$$\phi = atan2(\frac{R21}{cos(\theta)}, \frac{R11}{cos(\theta)})$$

Vi visualiserer endringen for løsning X ved å ha en graf per komponent i løsningsmatrisen. Her er X formatert slik:

$$\begin{bmatrix} (0,0) & (0,1) & (0,2) \\ (1,0) & (1,1) & (1,2) \\ (2,0) & (2,1) & (2,2) \end{bmatrix}$$

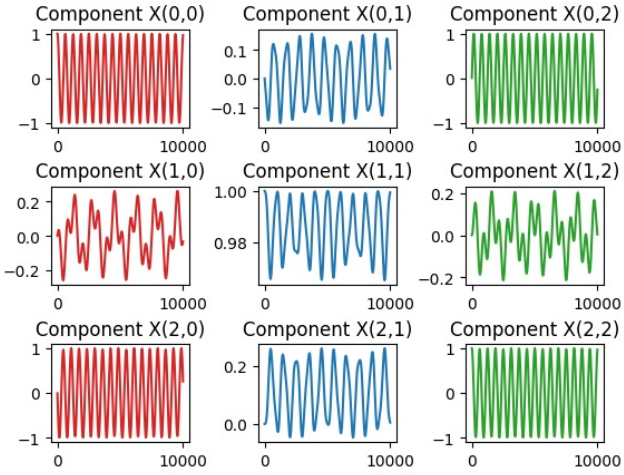
Vi ser først på løsningen for oppgave 5a)
Vi setter h = 0.01 på intervallet [0,100], og $\vec{\omega}(0) = \begin{bmatrix} 1 & 0.05 & 0 \end{bmatrix}^T$, og får:



Figur 14: Fremstilling av X(a)

Vi oppdager at alle 9 komponentene i tilnærming X fra RK4-metoden har en konstant periodisk endringsrate. Vi kan se på eulervinklene til rotasjonsmatrisen X, for å se på hvordan denne vil beskrive bevegelsen til t-nøkkelen. Vi vet at eulervinkelen for x-aksen er avhengig av komponentene X(1,1), X(1,2), X(2,1) og X(2,2), som vi ser har både veldig kort periode og høy varians, som vi kan tolke som en konstant, rask, fullstendig rotasjon rundt x-aksen. Eulervinkelen for y-aksen er avhengig av komponent X(2,0), som har en relativ stor periode, og vil føre til jevnlige rotasjoner rundt y-aksen. Den siste eulervinkelen for z-aksen er avhengig av komponentene X(1,0) og X(0,0). Disse komponentene har en veldig lik periode, som fører til at vi kan tenke oss at rotasjonen rundt y- og z-aksen vil skje samtidig. Det vil si at vi kan konkludere med at ut i fra denne grafen, kan vi tolke bevegelsen til t-nøkkel som en konstant, fullstendig rotasjon rundt x-aksen, med jevnlige rotasjoner rundt y- og z-aksen som vil mest sannsynlig skje samtidig.

Vi ser nå på løsningen for oppgave 5b)
Her setter vi h = 0.01 på intervallet [0,100], og $\vec{\omega}(0) = \begin{bmatrix} 0 & 1 & 0.05 \end{bmatrix}^T$, og får:

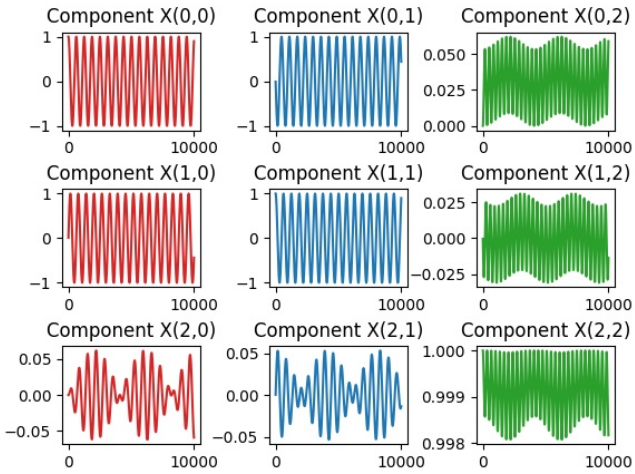


Figur 15: Fremstilling av X(b)

Det første man observerer er at her er det også en konstant periodisk endringsrate for alle komponenter, som fører til at t-nøkkelen også har en konstant bevegelse i dette tilfellet. Vi ser først på komponentene som eulervinkelen til x-aksen er avhengige av, altså X(1,1), X(1,2), X(2,1) og X(2,2). Her observerer

vi at 3 av 4 komponenter varierer i liten grad i forhold til andre komponentene, som vil føre til at rotasjonen rundt x-aksen er veldig liten. Eulervinkelen til y-aksen er avhengig av komponent X(2,0), som i dette tilfellet har variasjon fra [-1,1] og kort periode. Dette kan tolkes som at rotasjonen rundt y-aksen er rask og at t-nøkkelen vil dreie helt rundt y-aksen. Eulervinkelen til Z-aksen er avhengige av komponentene X(0,0) og X(1,0), som i dette tilfellet fører til at rotasjonsbevegelsen rundt z-aksen er ikke stor. Vi konkluderer her med at t-nøkkelen vil bevege seg rundt y-aksen med små jevnlige rotasjonsbevegelser rundt x- og z-aksen.

Vi ser nå på løsningen for oppgave 5c)
Her setter vi h = 0.01 på intervallet [0,100], og $\vec{\omega}(0) = \begin{bmatrix} 0.05 & 0 & 1 \end{bmatrix}^T$, og får:



Figur 16: Fremstilling av X(c)

For siste løsningen fra oppgave 5, observerer vi at variansen til komponentene X(1,1), X(1,2), X(2,1) og X(2,2) som eulervinkelen til x-aksen er avhengig av er veldig liten. Her vil rotasjonsbevegelsen rundt x-aksen være periodisk, men være veldig liten. Eulervinkelen til y-aksen er avhengig av komponent X(2,0), som vil også føre til en periodisk rotasjonsbevegelse, men denne er også veldig liten fordi komponenten har liten varians. Vi ser til slutt at komponentene som eulervinkelen til Z-aksen er avhengige av X(0,0) og X(1,0) har veldig høy varians og liten periode. Vi konkluderer dermed med at bevegelsen til t-nøkkelen i dette tilfelle vil være en konstant rask fullstendig rotasjon rundt z-aksen, med

små periodiske rotasjonsbevegelser i mot x- og y-aksene.

5 Diskusjon

I løpet av dette prosjektet har vi fått jobbet med flere ulike problemstillinger som omhandler rotasjon av stive legemer. Vi har brukt ulike numeriske metoder for å tilnærme løsninger til differnsiallikninger, deriblant Euler, Runge Kutta 4 og Runge Kutta-Fehlberg.

De tre deloppgavene i oppgave 1 la grunnlaget for arbeidet videre. Her definerte vi funksjoner som vi senere fikk bruk for når vi skulle tilnærme løsninger til differensiallikningene.

Oppgave 2 viste seg å være mer utfordrende enn først antatt, men vi klarte omsider å finne den eksakte løsningen ved spesialtilfellet der legemet er en kule. Denne oppgaven var viktig å få til ettersom vi i senere oppgaver brukte den eksakte løsningen som referansepunkt for å sammenligne de numeriske løsningene vi implementerte. Slik kunne vi enkelt finne ut nøyaktigheten til tilnærmningene.

I oppgave 3 skulle vi begynne med å numerisk approksimere verdier ved bruk av Eulers metode. Vi startet her med å implementere likning (25) fra prosjektheftet [4], slik oppgaven tilsa, og tolket så resultatene vi fikk ut. Ved bare å se på de tilnærmede verdiene (vedlegg 8.0.2, figur 16) så vi at dette så lovende ut, da de foreslo at figuren roterte relativt konstant, noe vi observerte kun ved å se på ulike verdier for X og t. Videre testet vi den tilnærmede løsning opp mot den eksakte løsningen fra oppgave 2, og ser at for rad 1 og kolonne 1 fikk vi nøyaktig det samme som i den eksakte løsningen. Dette var å forvente, men det er på de resterende elementene vi kunne se en differanse. På f.eks det midtre elementet W_{22} kunne vi se at feilen gikk fra 0 til å bli omtrent $2.2 * 10^{-16}$ ved intervallslutt (se figur 3, side 21).

For å ha noe å sammenlikne med valgte vi implementere likning (24) også, vel vitende om at denne hadde en rekke svakheter. En av disse svakhetene var for eksempel at W_i ikke var ortogonal. Når vi plottet denne for å se feilen opp i mot den eksakte løsningen fikk vi til sammenlikning 0.03 på det midtre elementet ved intervallslutt (se figur 2, side 21). Dette viser en mye mindre nøyaktig tilnærming. Vi testet også for steglengde $h = 0.001$ for å se om denne nøyaktigheten var lik ved lavere steglengde. Her fikk vi at (24) ble noe mer nøyaktig med 0.0002 på samme midtre element i forhold til 0.03, men den er

enda langt unna (25) som fikk $5.99 * 10^{-15}$. Basert på disse observasjonene kan vi konkludere med at den forbedrede versjonen der W_i er ortogonal, gitt i likning (25), har en mye bedre nøyaktighet enn varianten gitt i likning (24). Dette var dog ikke et faktum for veldig små steglengder h, slik vi observerte og beskrev i resultatkapittelet. Da ble forskjellen mellom de to variantene minimal.

På samme måte sjekket vi resultatet for Runge-Kutta metodene RK4 og RKF45. Sammenlignet med Euler-metodene, bød disse på en god del mindre feil som følge av at disse er av fjerde eller høyere orden, men da vi satte de to Runge-Kutta metodene opp mot hverandre så vi ingen forskjell på de to. Her kan vi spekulere i hva som fører til dette, men vår mistanke er at dette skyldes at RK4 allerede konvergerer fort mot en nøyaktig løsning. På den annen side, kan det hende at verdiene er sammenlignet feil. RKF45 er en variabel steglengde metode og dermed vil man ikke få eksakt samme tidsverdier som for RK4 om RKF45 møter på et sted hvor den estimerte feilen går over den tillatte toleransen. Om dette er tilfellet, kan det som sagt være at sammenligningen mellom de to blir unøyaktig. Samtidig forsøkte vi å sjekke feilen ved de spesifikke tidsenhetene som RKF45 benytter seg av og deretter sammenligne mot RK4. Dette ga samme resultat - vi fikk ingen forskjell.

Til slutt i oppgave 6 fikk vi plotte løsningene som metoden RK4 returnerte som en funksjon av tiden, for å visualisere bevegelsen av t-nøkkelen. Her valgte vi å se på eulervinklene til løsningsmatrisene for å tolke bevegelsen. Ved å bruke eulervinklene definert av komponentene i matrisene, fikk vi en veldig illustrerende beskrivelse av hvordan T-nøkkelen vil bevege seg. Dersom T-nøkkelen får en dominant rotasjonshastighet over x-aksen som i oppgave 5a, vil nøkkelen konstant rotere rundt x-aksen og periodisk rotere rundt y- og z-aksen. Vi kan tenke oss at bevegelsen rundt y- og z-aksen vil se ut som at sylinderen til t-nøkkelen periodisk bytter side av y-aksen. Dersom t-nøkkelen får en rotasjonshastighet hovedsaklig rundt y-, eller z-aksen som i oppgave 5b og c derimot, vil rotasjonsbevegelsen hovedsaklig bare skje over én akse.

Dersom vi hadde hatt mer tid på prosjektet er det flere ting vi gjerne skulle ha undersøkt litt nærmere. Blant annet hadde det vært interesant å se hvordan T-nøkkelen hadde forandret seg over tid ved å animere T-nøkkelen i et X, Y, Z koordinatsystem. I tillegg hadde det vært interessant å undersøke andre numeriske metoder for å se hvor store forskjeller, eventuelt likheter, det var mellom metodene.

6 Konklusjon

I forkant av prosjektet hadde vi hatt forelesninger om flere ulike metoder for å tilnærme løsninger på differensiallikninger. Dette var helt nødvendig for at vi skulle kunne klare å løse oppgavene i prosjektet, men det viste seg at vi hadde behov for å lese oss enda mer opp. Gruppemedlemmene var flinke på å gjøre research underveis, og på å formidle den kunnskapen videre til de resterende medlemmene. Dette førte med seg en god arbeidsflyt innad i teamet.

I tillegg til det matematiske aspektet var det også nødvendig å lese oss opp på fysikken for å få et fullstendig bilde av hva som måtte gjøres.

Vi vil understreke at vi er tilfredstilt med endt resultat, og kan konkludere med at vi har tilegnet oss nyttig kunnskap innenfor feltet. I tillegg har vi fått god erfaring med bruk av programmeringsspråket Python. Til sammen ble dette brukt til å analysere den spinnende t-nøkkelen. Utifra resultatene våre kan vi konkludere med at den slettes ikke roterer som en skulle tro til å begynne med!

7 Referanseliste

[1] Sauer, T. (2012). "Numerical analysis, second edition". Pearson Education, Inc. George Mason University

[2] Young, H. D. & Freedman, R. A. (2015). "University Physics with Modern Physics, 14th edition". Pearson Education, Inc. University of California, Santa Barbara

[3] Slabaugh, G. G. (2014). "Computing Euler angles from a rotation matrix". https://www.gregslabaugh.net/publications/euler.pdf

[4] Rivertz, H. J. (2020). "Rotasjon av Stive Legemer. Prosjektoppgave i TDAT3024"

[5] Maskin epsilon, Wikipedia. Hentet ut den 28.10.20 fra https://en.wikipedia.org/wiki/Machine_epsilon

[6] Euler, Store norske leksikon. Hentet ut den 28.10.20 fra https://snl.no/Eulers_metode

[7] Lie Group, Wikipedia. Hentet ut den 28.10.20 fra https://no.wikipedia.org/wiki/Lie-gruppe

[8] Runge-Kutta, Wikipedia. Hentet ut den 30.10.20 fra https://en.wikipedia.org/wiki/Runge\OT1\textendashKutta_methods

8 Vedlegg - utskrift fra programmer

8.1 oppg1.py

```
The calculated moment of intertia of the T-handle is:
[[9.82271303e-05 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 7.22671030e-05 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 1.57865031e-04]]

Testing the exponential function from task b....
Test did succeed
```

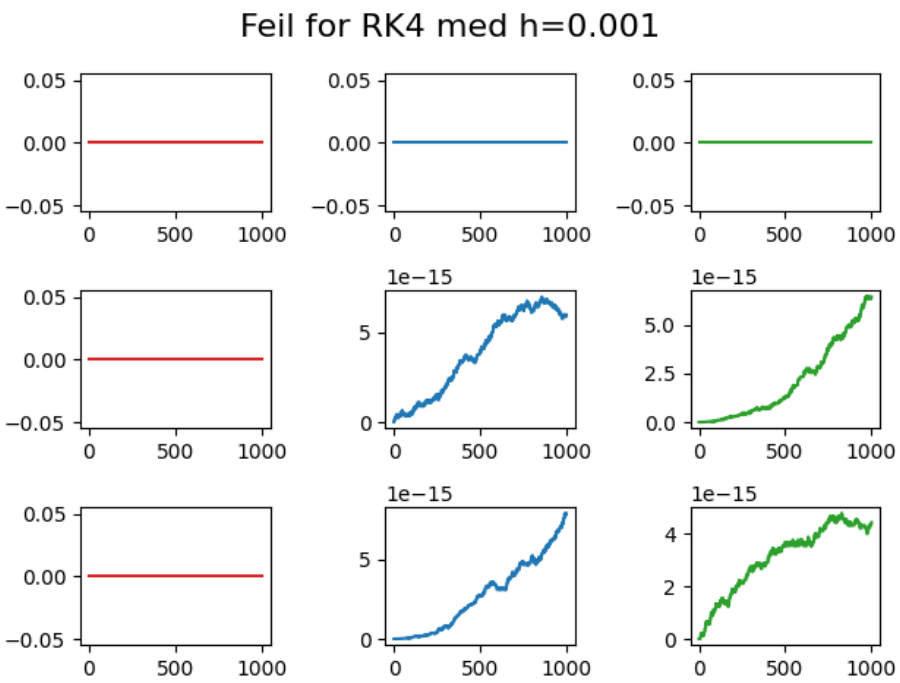
Figur 17: Utskrift oppgave 1

8.2 oppg3.py

```
  i    t_i  Euler (24)                                Euler (25)
---  -----  ------------------------------------------  ------------------------------------------
  0    0    [[1. 0. 0.]                                 [[1. 0. 0.]
             [0. 1. 0.]                                  [0. 1. 0.]
             [0. 0. 1.]]                                 [0. 0. 1.]]
  1    0.1  [[ 1.   0.   0. ]                           [[ 1.          0.          0.        ]
             [ 0.   1.  -0.1]                            [ 0.          0.99500417 -0.09983342]
             [ 0.   0.1  1. ]]                           [ 0.          0.09983342  0.99500417]]
  2    0.2  [[ 1.    0.    0.  ]                        [[ 1.          0.          0.        ]
             [ 0.    0.99 -0.2 ]                         [ 0.          0.98006658 -0.19866933]
             [ 0.    0.2   0.99]]                        [ 0.          0.19866933  0.98006658]]
  3    0.3  [[ 1.     0.     0.   ]                     [[ 1.          0.          0.        ]
             [ 0.     0.97  -0.299]                      [ 0.          0.95533649 -0.29552021]
             [ 0.     0.299  0.97 ]]                     [ 0.          0.29552021  0.95533649]]
  4    0.4  [[ 1.      0.      0.    ]                  [[ 1.          0.          0.        ]
             [ 0.      0.9401 -0.396 ]                   [ 0.          0.92106099 -0.38941834]
             [ 0.      0.396   0.9401]]                  [ 0.          0.38941834  0.92106099]]
  5    0.5  [[ 1.       0.       0.     ]                [[ 1.          0.          0.        ]
             [ 0.       0.9005  -0.49001]                 [ 0.          0.87758256 -0.47942554]
             [ 0.       0.49001  0.9005 ]]                [ 0.          0.47942554  0.87758256]]
  6    0.6  [[ 1.        0.        0.      ]              [[ 1.          0.          0.        ]
             [ 0.        0.851499 -0.58006 ]               [ 0.          0.82533561 -0.56464247]
             [ 0.        0.58006   0.851499]]              [ 0.          0.56464247  0.82533561]]
  7    0.7  [[ 1.         0.         0.       ]           [[ 1.          0.          0.        ]
             [ 0.         0.793493  -0.6652099]            [ 0.          0.76484219 -0.64421769]
             [ 0.         0.6652099  0.793493 ]]           [ 0.          0.64421769  0.76484219]]
  8    0.8  [[ 1.          0.          0.        ]        [[ 1.          0.          0.        ]
             [ 0.          0.72697201 -0.7445592 ]        [ 0.          0.69670671 -0.71735609]
             [ 0.          0.7445592   0.72697201]]       [ 0.          0.71735609  0.69670671]]
  9    0.9  [[ 1.          0.          0.        ]        [[ 1.          0.          0.        ]
             [ 0.          0.65251609 -0.8172564 ]        [ 0.          0.62160997 -0.78332691]
             [ 0.          0.8172564   0.65251609]]       [ 0.          0.78332691  0.62160997]]
 10    1    [[ 1.          0.          0.        ]        [[ 1.          0.          0.        ]
             [ 0.          0.57079045 -0.88250801]        [ 0.          0.54030231 -0.84147098]
             [ 0.          0.88250801  0.57079045]]       [ 0.          0.84147098  0.54030231]]
```

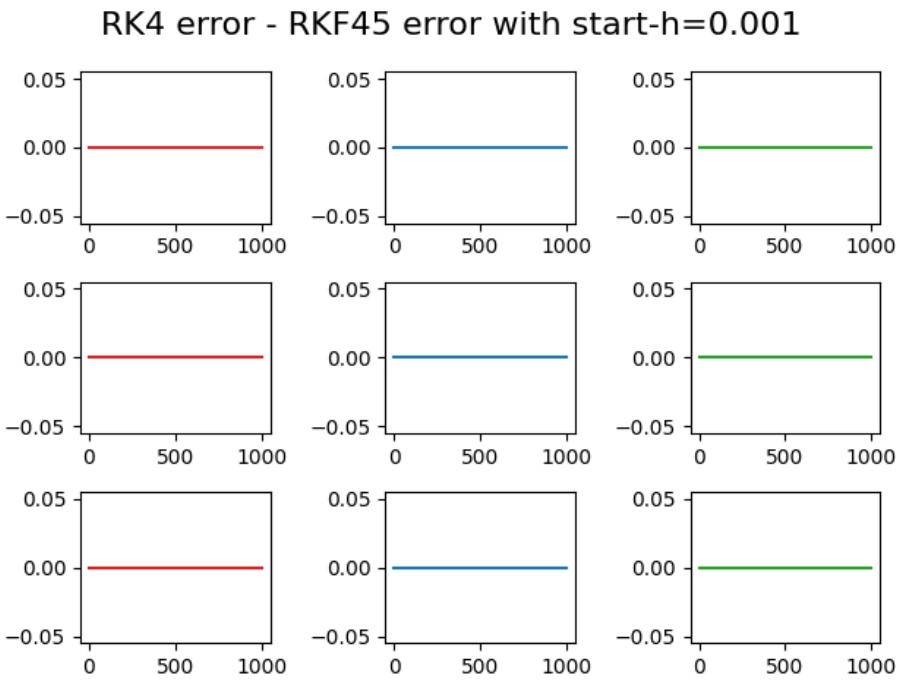
Figur 18: Utskrift av de tilnærmede verdiene for rotasjon ved steglengde $h = 0.1$, for begge implementerte varianter av Euler

8.3 oppg4.py



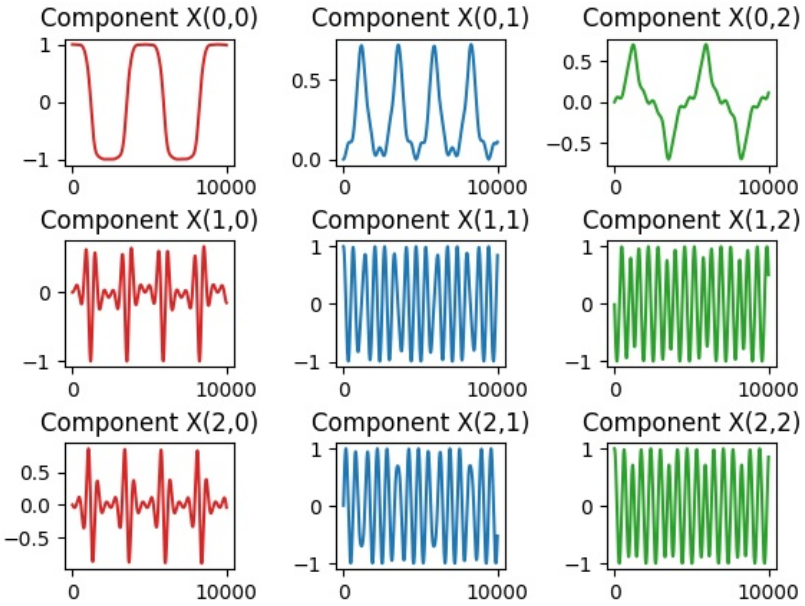
Figur 19: Utskrift oppgave 4

8.4 oppg4_fehlberg.py

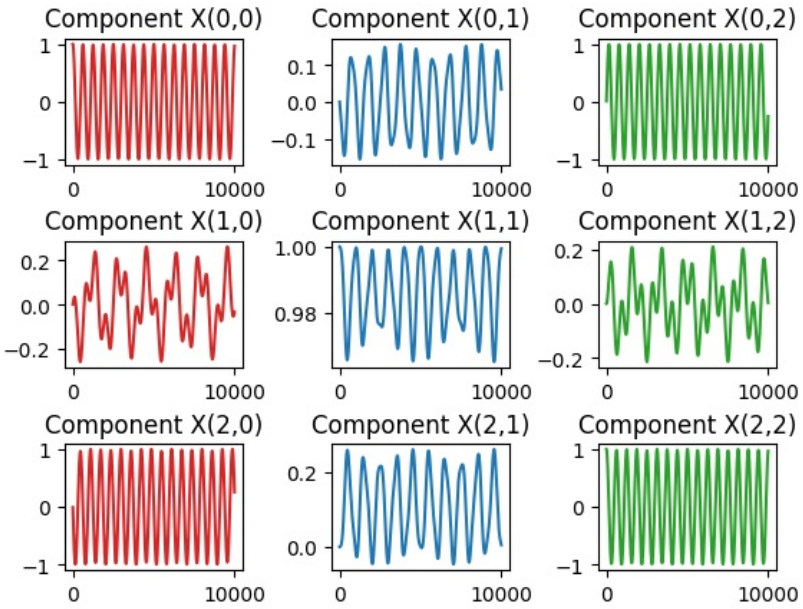


Figur 20: Utskrift oppgave 4

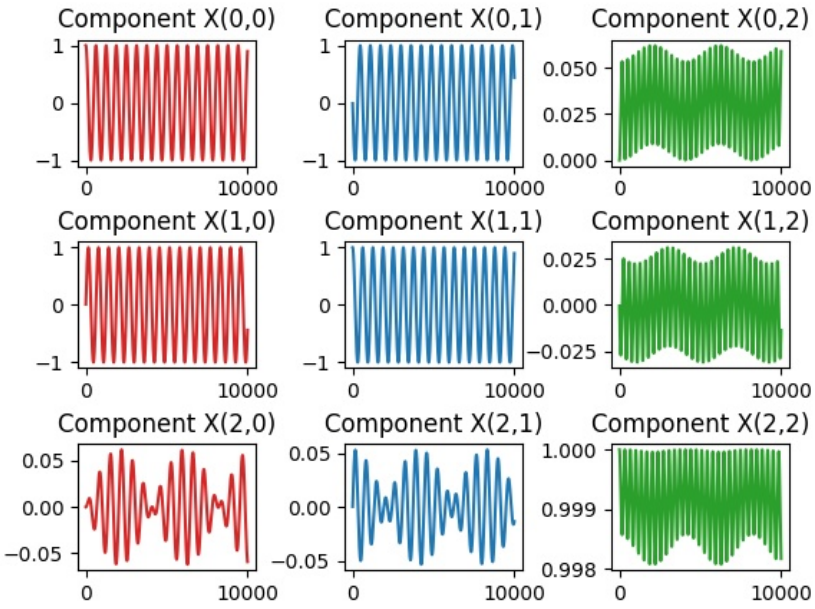
8.5 oppg6.py



Figur 21: Utskrift oppgave 6



Figur 22: Utskrift oppgave 6



Figur 23: Utskrift oppgave 6

9 Vedlegg - Skrevet kode

9.1 oppg1.py

```
import numpy as np
import math


# Oppgave a)
def exponent_function(h, Ω):
    """
    Approximates the position of the axis
    @param h: Step size
    @param Ω: Rotation-matrix
    @return: Matrix X
    """
    ω_x, ω_y, ω_z = Ω[2, 1], Ω[0, 2], Ω[1, 0]

    ω = np.sqrt(ω_x ** 2 + ω_y ** 2 + ω_z ** 2)

    identity_matrix = np.identity(3)
    return identity_matrix + ((1 - np.cos(ω * h)) * ((np.dot(Ω,
    ↪  Ω)) / ω ** 2)) + (np.sin(ω * h) * (np.divide(Ω, ω)))


def test_exponent_function(ω, h):
    """
    Test that the exponential function satisfies the requirements
↪   of X^T * X = Id_(3x)
    @param ω:
    @param h: step-size
    @return: true if the requirement is satisfied
    """
    identity_matrix = np.identity(3)

    ω_x = ω[0]
    ω_y = ω[1]
    ω_z = ω[2]
```

```
    Ω = np.array(
        [[0, -ω_z, ω_y],
         [ω_z, 0, -ω_x],
         [-ω_y, ω_x, 0]]
    )
    X_approx = exponent_function(h, Ω)
    should_be_identity_matrix = np.dot(np.transpose(X_approx),
    ↪  X_approx)
    return np.allclose(should_be_identity_matrix,
    ↪  identity_matrix)


# Oppgave b)
def torque_L(I, ω):
    """
    Calculates the torque of a rigid body
    @param I: Moment of inertia of the rigid body
    @param ω: Rotation vector (1 x 3)
    @return: Torque
    """
    # Calculates the torque from the components of I
    I_xx = I[0, 0]
    I_xy = I[0, 1]
    I_xz = I[0, 2]
    I_yx = I[1, 0]
    I_yy = I[1, 1]
    I_yz = I[1, 2]
    I_zx = I[2, 0]
    I_zy = I[2, 1]
    I_zz = I[2, 2]

    ω_x, ω_y, ω_z = ω[0], ω[1], ω[2]

    # Finds the components of torque L
    L_x = (I_xx * ω_x) - (I_xy * ω_y) - (I_xz * ω_z)
    L_y = (-I_yx * ω_x) + (I_yy * ω_y) - (I_yz * ω_z)
    L_z = (-I_zx * ω_x) - (I_zy * ω_y) + (I_zz * ω_z)
```

```
    return np.array([L_x, L_y, L_z])


def energy_function(I, ω):
    """
    Calculates the energy of a rotating rigid body
    @param I: Moment of inertia of the rigid body
    @param ω: Rotation-vector
    @return: The calculated kinetic energy
    """
    L = torque_L(I, ω)
    K = (1 / 2) * (np.dot(L, ω))
    return abs(K)


# Oppgave c
def calculate_moment_of_inertia_of_T():
    """
    Calculates moment of inertia of T-handle
    @return: The moment of inertia (3 x 3) - matrix
    """
    length_handle = 0.08  # 8 cm
    radius_handle = 0.01  # 1 cm

    length_cylinder = 0.04  # 4 cm
    radius_cylinder = 0.01  # 1 cm

    density = 6700  # 6.7 gram per cubic centimeters = 6700 kg/m

    volume_handle = math.pi * math.pow(radius_handle, 2) *
    ↪  length_handle
    mass_handle = volume_handle * density

    volume_cylinder = math.pi * math.pow(radius_cylinder, 2) *
    ↪  length_cylinder
    mass_cylinder = volume_cylinder * density
```

```
    I_xx = ((mass_handle * math.pow(radius_handle, 2)) / 4) +
    ↪  ((mass_handle * math.pow(length_handle, 2)) / 12) + (
            (mass_cylinder * math.pow(radius_cylinder, 2)) / 2)

    I_yy = (mass_handle * math.pow(radius_handle, 2)) +
    ↪  ((mass_cylinder * math.pow(length_cylinder, 2)) / 4) + (
            (mass_handle * math.pow(radius_handle, 2)) / 2) + (
                   (mass_cylinder * math.pow(radius_cylinder, 2))
                   ↪  / 4) + (
                   (mass_cylinder * math.pow(length_cylinder, 2))
                   ↪  / 12)

    I_zz = (mass_handle * math.pow(radius_handle, 2)) +
    ↪  ((mass_cylinder * math.pow(length_cylinder, 2)) / 4) + (
            (mass_handle * math.pow(radius_handle, 2)) / 4) +
            ↪  ((mass_handle * math.pow(length_handle, 2)) / 12)
            ↪  + (
                   (mass_cylinder * math.pow(radius_cylinder, 2))
                   ↪  / 4) + (
                   (mass_cylinder * math.pow(length_cylinder, 2))
                   ↪  / 12)

    return np.array([[I_xx, 0, 0],
                     [0, I_yy, 0],
                     [0, 0, I_zz]])


if __name__ == "__main__":
    print("The calculated moment of inertia of the T-handle is:")
    print(calculate_moment_of_inertia_of_T())

    print("\nTesting the exponential function from task b....")
    h = 0.001
    for i in range(10000):
        random_rotation_matrix = np.random.randint(1000, size=(1,
        ↪  3))
        if not test_exponent_function(random_rotation_matrix[0],
        ↪  h):
```

```
            print("Test did not succeed")
            break
    print("Test did succeed")
```

9.2 oppg2.py

```
import numpy as np
import math


def exact_solution(t):
    """
    Calculates the position of the components at given time t
    @param t: time
    @return: The exact solution X given the time t
    """
    X = np.array([
        [1, 0, 0],
        [0, math.cos(t), -(math.sin(t))],
        [0, math.sin(t), math.cos(t)]
    ])
    return X
```

9.3 oppg3.py

```
import numpy as np
import test_utils
from oppg1 import exponent_function


def run_steps(h, X_0, I, L, interval, method_for_ω_i,
↪  method_for_W_i):
    """
    This method run the steps for Euler with a given method for
↪  calculating ω_i and W_i, as this is what differentiates
↪  method (24) and (25)
    @param h: Step-size
    @param X_0: Initial condition
    @param I: Moment of inertia
    @param L: Torque
    @param interval:
    @param method_for_ω_i: Calculates the ω
    @param method_for_W_i: Calculates the W
    @return: An approximation of the positions for any given time
↪  in the interval
    """
    W_i = []
    t_i = []
    n_steps = int(interval[1] / h) + 1
    for i in range(interval[0], n_steps):
        t_i.append(i * h)

        if i == 0:
            W_i.append(X_0)
        else:
            ω_i = method_for_ω_i(I, W_i[i - 1], L)
            Ω = np.array([[0, -ω_i[2], ω_i[1]],
                          [ω_i[2], 0, -ω_i[0]],
                          [-ω_i[1], ω_i[0], 0]])
            W_i.append(method_for_W_i(W_i, i, h, Ω))
    return W_i, t_i
```

```
def Euler(h, X_0, I, L, interval=[0, 1]):
    """
    Calculates the given differential equation on matrix-form
↪   using the Euler method

    @param h: step-size
    @param X_0: Initial condition
    @param I: Moment of inertia
    @param L: Torque
    @param interval: Interval
    @return: An approximation of the positions for any given time
↪   in the interval and the steps
    """

    def calculate_ω_i(I, W_i, L):
        "Calculates ω_i as this gives us the components of Ω,
        ↪  based on the fact that ω_i = [ω_x ω_y ω_z]^T"
        return np.dot(np.linalg.inv(np.dot(W_i, I)), L)

    def calculate_W_i(W_i, i, h, Ω):
        return W_i[i - 1] + np.dot(h, W_i[i - 1] @ Ω)

    W_i, t_i = run_steps(h, X_0, I, L, interval, calculate_ω_i,
    ↪  calculate_W_i)

    return W_i, t_i


def Euler_improved(h, X_0, I, L, interval=[0, 1]):
    """
    Calculates the given differential equation on matrix-form
↪   using the Euler method, and fixing the weakness of
↪   Euler_formula_24 that W is not ortogonal"

    @param h: step-size
    @param X_0: Initial condition
    @param I: Moment of inertia
```

```
    @param L: Torque
    @param interval: Interval
    @return: An approximation of the positions for any given time
↪   in the interval and the steps
    """

    def calculate_ω_i(I, W_i, L):
        "Calculates ω_i as this gives us the components of Ω,
        ↪  based on the fact that ω_i = [ω_x ω_y ω_z]^T"
        I_inv = np.linalg.inv(I)
        W_trans = np.transpose(W_i)
        return np.dot(np.dot(I_inv, W_trans), L)

    def calculate_W_i(W_i, i, h, Ω):
        return W_i[i - 1] @ exponent_function(h=h, Ω=Ω)

    W_i, t_i = run_steps(h, X_0, I, L, interval, calculate_ω_i,
    ↪  calculate_W_i)

    return W_i, t_i


if __name__ == '__main__':
    h = 0.1
    interval = [0, 1]
    L = np.array([1, 0, 0])
    X_0 = I = np.eye(3)

    res0, t_i0 = Euler(X_0=X_0, I=I, L=L, h=h, interval=interval)
    res1, t_i1 = Euler_improved(X_0=X_0, I=I, L=L, h=h,
    ↪  interval=interval)

    test_utils.print_table(t_i1, [res0, res1], ['Euler (24)',
    ↪  'Euler (25)'])

    # Plot the change of global truncation error in (24) and (25)
    # err0 = test_utils.test_vals_on_sphere(res0, t_i0, h,
    ↪  interval, "Euler (24)")
```

```
# err1 = test_utils.test_vals_on_sphere(res1, t_i1, h,
↪  interval, "Euler (25)")

# test_utils.plot_error_graph(interval, h, err0, f"Error for
↪  Euler (24) with h={h}")
# test_utils.plot_error_graph(interval, h, err1, f"Error for
↪  Euler (25) with h={h}")

# Plot the difference between the two implementations
# test_utils.plot_error_graph(interval, h,
↪  abs(np.subtract(err0, err1)), f"Euler (24) error - Euler
↪  (25) error with h={h}")
```

9.4 oppg4.py

```
import numpy as np
from numpy import linalg as lin
from oppg1 import torque_L, exponent_function
import test_utils


def find_Σ(σ):
    # Separating components
    x = σ[0]
    y = σ[1]
    z = σ[2]

    Σ = np.array(
        [[0, -z, y],
         [z, 0, -x],
         [-y, x, 0]]
    )
    return Σ


# RK4
def step(I_inv, W, L, h):  # Method calculating one step of RK4
    """
    Calculates one step of Runge-Kutta 4
    @param I_inv: The inverse of moment of inertia
    @param W: Approximation of that step
    @param L: Torque
    @param h: step size
    @return: The next approximation
    """
    W_trans = np.transpose(W)

    σ_1 = np.dot(np.dot(I_inv, W_trans), L)

    Σ_1 = find_Σ(σ_1)
    σ_2 = np.dot(
```

```
        np.dot(np.dot(I_inv, exponent_function((-h / 2), Σ_1)),
        ↪  W_trans), L)

    Σ_2 = find_Σ(σ_2)
    σ_3 = np.dot(
        np.dot(np.dot(I_inv, exponent_function((-h / 2), Σ_2)),
        ↪  W_trans), L)

    Σ_3 = find_Σ(σ_3)
    σ_4 = np.dot(
        np.dot(np.dot(I_inv, exponent_function(-h, Σ_3)),
        ↪  W_trans), L)

    Σ_4 = find_Σ(σ_4)
    W_next = np.dot(W, exponent_function(
        h / 6, (Σ_1 + (2 * Σ_2) + (2 * Σ_3) + Σ_4)))

    return W_next


def RK4(W_init, I, h, interval, ω):
    """
    Implementation of RK4
    @param W_init:
    @param I: moment of inertia matrix
    @param h: step size
    @param interval: 1x2 matrix containing start- and endpoint of
↪   sampling
    @param ω: matrix containing the angular velocity for each
↪   axis x,y,z
    @return: An approximation of the positions for any given time
↪   in the interval and the steps
    """

    I_inverse = lin.inv(I)
    n_steps = (interval[1] - interval[0]) / h
    L = torque_L(I, ω)
    W_values = [W_init]
```

```
    t_values = [interval[0]]

    for i in range(int(n_steps)):
        W_values.append(step(I_inverse, W_values[-1], L, h))
        t_values.append(t_values[-1] + h)

    return W_values, t_values


if __name__ == "__main__":
    X_0 = I = np.eye(3)
    h = 0.001
    interval = [0, 1]
    ω = [1, 0, 0]
    approximations_RK4, t_i = RK4(X_0, I, h, interval, ω)
    err = test_utils.test_vals_on_sphere(approximations_RK4, t_i,
    ↪  h, interval, "RK4")
    test_utils.plot_error_graph(interval, h, err, "Feil for RK4
    ↪  med h=0.001")
```

9.5 oppg4_fehlberg.py

```
import numpy as np
from numpy import linalg as lin
from oppg1 import torque_L, exponent_function
from oppg4 import find_Σ, RK4
import test_utils


# RKF45
class scheme():
    "The butcher table for RKF45"

    def __init__(self):
        self.c = np.array([[0],
                           [1 / 4],
                           [3 / 8],
                           [12 / 13],
                           [1],
                           [1 / 2],
                           [0],
                           [0]])
        self.A = np.array([[0, 0, 0, 0, 0],
                           [1 / 4, 0, 0, 0, 0],
                           [3 / 32, 9 / 32, 0, 0, 0],
                           [1932 / 2197, -7200 / 2197, 7296 /
                           ↪  2197, 0, 0],
                           [439 / 216, -8, 3680 / 513, -845 /
                           ↪  4104, 0],
                           [-8 / 27, 2, -3544 / 2565, 1859 /
                           ↪  4104, -11 / 40]])
        self.B = np.array([[25 / 216, 0, 1408 / 2565, 2197 /
        ↪  4104, -1 / 5, 0],
                           [16 / 135, 0, 6656 / 12825, 28561 /
                           ↪  56430, -9 / 50, 2 / 55]])

    def print_scheme(self):
        print(np.array([[self.c, self.A], ["", self.B]]))
```

```
    def a(self, row, column):
        return self.A[row - 1][column - 1]

    def b(self, row, column):
        return self.B[row - 1][column - 1]

    def c(self, row, column):
        return self.c[row - 1][column - 1]


def step(I, W_i, L, h):
    """
    Method calculating one step of RKF45
    @param I: Moment of inertia of rigid body
    @param W_i:
    @param L: Torque
    @param h: Step size
    @return: The next approximation
    """
    tab = scheme()
    I_inv = lin.inv(I)
    W_trans = np.transpose(W_i)

    def calculate_σ_i(I_inv, W_trans, L):
        return np.dot(np.dot(I_inv, W_trans), L)

    σ_1 = calculate_σ_i(I_inv, W_trans, L)

    Σ_1 = find_Σ(σ_1)
    σ_2 = I_inv @ exponent_function(-h, (tab.a(2, 1) * Σ_1)) @
    ↪  W_trans @ L

    Σ_2 = find_Σ(σ_2)
    σ_3 = I_inv @ exponent_function(-h, ((tab.a(3, 1) * Σ_1) +
    ↪  (tab.a(3, 2) * Σ_2))) @ W_trans @ L

    Σ_3 = find_Σ(σ_3)
```

```
σ_4 = I_inv @ exponent_function(-h, ((tab.a(4, 1) * Σ_1) +
↪  (tab.a(4, 2) * Σ_2) + (tab.a(4, 3) * Σ_3))) @ W_trans @ L

Σ_4 = find_Σ(σ_4)
σ_5 = I_inv @ exponent_function(-h, ((tab.a(5, 1) * Σ_1) +
↪  (tab.a(5, 2) * Σ_2) +
                                    (tab.a(5, 3) * Σ_3) +
                                    ↪  (tab.a(5, 4) *
                                    ↪  Σ_4))) @ W_trans @ L

Σ_5 = find_Σ(σ_5)
σ_6 = I_inv @ exponent_function(-h, ((tab.a(6, 1) * Σ_1) +
↪  (tab.a(6, 2) * Σ_2) + (tab.a(6, 3) * Σ_3) +
                                    (tab.a(6, 4) * Σ_4) +
                                    ↪  (tab.a(6, 5) *
                                    ↪  Σ_5))) @ W_trans @ L

Σ_6 = find_Σ(σ_6)

W_next = W_i @ exponent_function(h, ((tab.b(1, 1) * Σ_1) +
↪  (tab.b(1, 2) * Σ_2) + (tab.b(1, 3) * Σ_3) +
                                    (tab.b(1, 4) * Σ_4) +
                                    ↪  (tab.b(1, 5) * Σ_5)
                                    ↪  + (tab.b(1, 6) *
                                    ↪  Σ_6)))

Z_next = W_i @ exponent_function(h, ((tab.b(2, 1) * Σ_1) +
↪  (tab.b(2, 2) * Σ_2) + (tab.b(2, 3) * Σ_3) +
                                    (tab.b(2, 4) * Σ_4) +
                                    ↪  (tab.b(2, 5) * Σ_5)
                                    ↪  + (tab.b(2, 6) *
                                    ↪  Σ_6)))

delta_W = W_next - Z_next

trace_product = np.transpose(delta_W) @ delta_W
E = np.sqrt(trace_product[0][0] +
            trace_product[1][1] + trace_product[2][2])
```

```
    return W_next, Z_next, E


def in_tolerance(err_tolerance, W_next, E):
    "The relative error test for RKF45"
    return (E / lin.norm(W_next)) < err_tolerance


def run_steps(err_tolerance, I, W_i, L, h, interval):
    """
    Method calculating all the steps of RKF45 with the relative
↪   error test for each step
    @param err_tolerance:
    @param I: Moment of inertia
    @param W_i: Approximation
    @param L: Torque
    @param h: Step-size
    @param interval: 1x2 matrix containing start- and endpoint of
↪   sampling
    @return: An approximation of the positions for any given time
↪   in the interval and the steps
    """
    list_of_W = [W_i]
    list_of_t = [interval[0]]
    finished = False
    safety_factor = 0.8  # Safety factor to be conservative

    while not finished:
        W_next, Z_next, E = step(I, list_of_W[-1], L, h)
        half = False

        while not in_tolerance(err_tolerance, W_next, E):
            if not half:
                h = safety_factor * (
                        ((err_tolerance * lin.norm(W_next)) / E)
                        ↪  ** (1 / 5)) * h  # (6.57) from Sauer,
                        ↪  page 326
```

```
                half = True
            else:
                h = h / 2
            W_next, Z_next, E = step(I, W_next, L, h)

        list_of_W.append(Z_next)
        list_of_t.append(list_of_t[-1] + h)
        if list_of_t[-1] >= interval[1]:
            finished = True

    return list_of_W, list_of_t


def RKF45(W_init, I, h, interval, ω, err_tolerance=10 ** (-6)):
    """
    Implementation of Runge Kutta Fehlberg
    @param W_init:
    @param I: Moment of inertia
    @param h: step size
    @param interval: 1x2 matrix containing start- and endpoint of
↪   sampling
    @param ω: Matrix containing the rotation in each dimension
↪   x,y,z
    @param err_tolerance: tolerance, default 10^-6 from Sauer
    @return: An approximation of the positions for any given time
↪   in the interval and the steps
    """
    L = torque_L(I, ω)
    tol = err_tolerance

    return run_steps(tol, I, W_init, L, h, interval)


if __name__ == "__main__":
    h = 0.001
    interval = [0, 1]
    X_0 = I = np.eye(3)
    ω = [1, 0, 0]
```

```
res0, t_i0 = run_steps(10 ** (-6), I, X_0, ω, h, interval)
res1, t_i1 = RK4(X_0, I, h, interval, ω)

err0 = test_utils.test_vals_on_sphere(res0, t_i0, h,
↪  interval, "RKF45")
err1 = test_utils.test_vals_on_sphere(res1, t_i1, h,
↪  interval, "RK4")

test_utils.plot_error_graph(interval, h,
↪  abs(np.subtract(err0, err1)), "RK4 error - RKF45 error
↪  with start-h=0.001")
```

9.6 oppg5.py

```python
import numpy as np
from oppg4 import RK4
from oppg4_fehlberg import RKF45
from oppg1 import calculate_moment_of_inertia_of_T


def solve_with_RK4(ω, h, interval):
    X_0 = np.identity(3, dtype=float)
    moment_of_inertia = calculate_moment_of_inertia_of_T()
    W_values, t_values = RK4(X_0, moment_of_inertia, h, interval,
    ↪  ω)
    return W_values, t_values


def solve_with_RK45(ω, h, interval):
    X_0 = np.identity(3, dtype=float)
    moment_of_inertia = calculate_moment_of_inertia_of_T()
    tol = 10 ** (-6)
    a_W, _ = RKF45(X_0, moment_of_inertia, h, interval, ω, tol)
```

9.7 oppg6.py

```
import matplotlib.pyplot as plt
import numpy as np
from oppg5 import solve_with_RK4


def plot_components_with_RK4(ω):
    h = 0.005
    interval = [0, 50]
    result_RK4, t_values_RK4 = solve_with_RK4(ω, h, interval)

    x00 = []
    x01 = []
    x02 = []
    x10 = []
    x11 = []
    x12 = []
    x20 = []
    x21 = []
    x22 = []

    # We split up our result matrix into 9 sub-matrices in order
    ↪  to plot every component:
    for i in range(int((interval[1]-interval[0]) / h)):
        x00.append(result_RK4[i][0][0])
        x01.append(result_RK4[i][0][1])
        x02.append(result_RK4[i][0][2])
        x10.append(result_RK4[i][1][0])
        x11.append(result_RK4[i][1][1])
        x12.append(result_RK4[i][1][2])
        x20.append(result_RK4[i][2][0])
        x21.append(result_RK4[i][2][1])
        x22.append(result_RK4[i][2][2])

    # We plot our sub-matrices colour-coded in reference to
    ↪  vectors i,j,k in matrix X
    fig, axs = plt.subplots(3, 3)
    axs[0, 0].plot(x00, 'tab:red')
```

```
    axs[0, 0].set_title('Component X(0,0)')
    axs[0, 1].plot(x01, 'tab:blue')
    axs[0, 1].set_title('Component X(0,1)')
    axs[0, 2].plot(x02, 'tab:green')
    axs[0, 2].set_title('Component X(0,2)')
    axs[1, 0].plot(x10, 'tab:red')
    axs[1, 0].set_title('Component X(1,0)')
    axs[1, 1].plot(x11, 'tab:blue')
    axs[1, 1].set_title('Component X(1,1)')
    axs[1, 2].plot(x12, 'tab:green')
    axs[1, 2].set_title('Component X(1,2)')
    axs[2, 0].plot(x20, 'tab:red')
    axs[2, 0].set_title('Component X(2,0)')
    axs[2, 1].plot(x21, 'tab:blue')
    axs[2, 1].set_title('Component X(2,1)')
    axs[2, 2].plot(x22, 'tab:green')
    axs[2, 2].set_title('Component X(2,2)')

    plt.show()


if __name__ == "__main__":
    a = np.array([1, 0.05, 0])
    b = np.array([0, 1, 0.05])
    c = np.array([0.05, 0, 1])

    plot_components_with_RK4(a)
    plot_components_with_RK4(b)
    plot_components_with_RK4(c)
```

9.8 test_utils.py

```
from operator import index
import numpy as np
import matplotlib.pyplot as plt
from tabulate import tabulate
from oppg2 import exact_solution


def exact(interval, h):
    """
    Calculates the exact solutions
    @param interval:
    @param h: step size
    @return: a list of the exact solutions
    """
    arr = np.linspace(interval[0], interval[1], int(
        (interval[1] - interval[0]) / h + 1))
    result = []
    for a in arr:
        result.append(exact_solution(a))

    return result


def test_vals_on_sphere(method_result, t_i, h, interval,
↪   method_name):
    """
    Testing numeric method on special case of object being a
↪   sphere.
    @param method_result: Matrix containing the resulting values
↪   from a given numeric method
    @param t_i: t for given step
    @param h: Step size
    @param interval: 2x1 array containing the start and end for
↪   the desired interval
    @param method_name: String containing name of the method
↪   used. Just a QOL variable to help setting the title of the
↪   graphs easily.
```

```
    @return: a list of the errors
    """

    error = []
    exact_list = exact(interval, h)
    num_of_steps = np.linspace(0, 1, int((interval[1] -
    ↪  interval[0]) / h)).size + 1
    for i in range(num_of_steps):
        error.append(abs(method_result[i] - exact_list[i]))

    # print_table(t_i, lists_of_W=[error],
    ↪  headers_for_W=[method_name])

    return error


def plot_error_graph(interval, h, error, title):
    x_11 = []
    x_12 = []
    x_13 = []
    y_21 = []
    y_22 = []
    y_23 = []
    z_31 = []
    z_32 = []
    z_33 = []
    for i in range(int(((interval[1] - interval[0]) / h) + 1)):
        x_11.append(error[i][0][0])
        x_12.append(error[i][0][1])
        x_13.append(error[i][0][2])
        y_21.append(error[i][1][0])
        y_22.append(error[i][1][1])
        y_23.append(error[i][1][2])
        z_31.append(error[i][2][0])
        z_32.append(error[i][2][1])
        z_33.append(error[i][2][2])

    fig, axs = plt.subplots(3, 3)
```

```
    plt.subplots_adjust(top=0.88, bottom=0.1, hspace=0.55,
    ↪  wspace=0.45)
    fig.suptitle(f"{title}", fontsize=16)
    axs[0, 0].plot(x_11, 'tab:red')
    axs[0, 1].plot(x_12, 'tab:blue')
    axs[0, 2].plot(x_13, 'tab:green')
    axs[1, 0].plot(y_21, 'tab:red')
    axs[1, 1].plot(y_22, 'tab:blue')
    axs[1, 2].plot(y_23, 'tab:green')
    axs[2, 0].plot(z_31, 'tab:red')
    axs[2, 1].plot(z_32, 'tab:blue')
    axs[2, 2].plot(z_33, 'tab:green')

    plt.show()


def print_table(t_values, lists_of_W, headers_for_W):
    assert len({len(i) for i in lists_of_W}
               ) == 1, "The lists are not the same length, use
               ↪  the same h and interval!"
    tableArray = []
    for i, W_i in enumerate(zip(*lists_of_W)):
        row = []
        row.append(i)
        row.append(t_values[i])
        for x in W_i:
            row.append(x)
        tableArray.append(row)
    headers = ['i', 't_i']
    for x in headers_for_W:
        headers.append(x)
    print(tabulate(tableArray, headers=headers))

    """
    with open('./print.txt', 'w+') as f:
        f.write(tabulate(tableArray, headers=headers))
    """
```

10 Erklæring

"Vi er ved felles enighet om at samtlige av gruppens medlemmer har bidratt til dette gruppearbeidet. Vi har møtt opp på alle møter og gjort ting underveis som et team. Den innleverte besvarelsen er et produkt vi alle har vært med å produsere."

- Kristian Macdonald William Gulaker, Max Torre Schau, Martin Johannes Nilsen, Ole Jonas Liahagen