Implementacija i provjera simulacijskog modela kvadrirotorske letjelice u stvarnom vremenu na Raspberry Pi-u

Jurinić, Dominik

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

DIPLOMSKI RAD

Dominik Jurinić

Zagreb, 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

IMPLEMENTACIJA I PROVJERA SIMULACIJSKOG MODELA KVADRIROTORSKE LETJELICE U STVARNOM VREMENU NA RASPBERRY PI - U

Mentor:
Doc. dr. sc. Dario Zlatar

Student:
Dominik Jurinić

Zagreb, 2020.

Izjavljujem da sam ovaj rad izradio samostalno koristeći stečena znanja na fakultetu i navedenu literaturu

Zahvaljujem se svom mentoru doc. dr. sc. Dariu Zlataru koji je u ovaj rad utkao mnogo pažnje i strpljenja. Zahvaljujem se također i Viktoru Pandži mag. ing. mech., na pomoći i savjetima s programiranjem. Na kraju, hvala obitelji i prijateljima koji su mi uvijek bili bezrezervna podrška.

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE
Središnje povjerenstvo za završne i diplomske ispite
Povjerenstvo za diplomske ispite studija strojarstva za smjerove:
procesno-energetski, konstrukcijski, brodostrojarski i inženjersko modeliranje i računalne simulacije

Sveučilište u Zagrebu Fakultet strojarstva i brodogradnje	
Datum:	Prilog:
Klasa: 602 - 04 / 20 - 6 / 3	
Ur. broj: 15 - 1703 - 20 -	

DIPLOMSKI ZADATAK

Student: **Dominik Jurinić** Mat. br.: 0035197320

Naslov rada na hrvatskom jeziku: **Implementacija i provjera simulacijskog modela kvadrirotorske letjelice u stvarnom vremenu na Raspberry Pi-u**

Naslov rada na engleskom jeziku: **Implementation and validation of a real-time simulation model of a quadrotor aerial vehicle on a Raspberry Pi**

Opis zadatka:

Između dostupnog hardvera za ugradbene sustave, odabrano je ARM ugradbeno računalo Raspberry Pi zbog svoje cijene, visoke učinkovitosti i predispozicije za izgradnju višerotorske letjelice. Za prijenos u najprikladniju programsku paradigmu (proceduralnu ili objektno orijentiranu) odabran je simulacijski model s novom numeričkom metodom vremenske integracije jediničnih kvaterniona, primijenjen na dinamičku analizu ugradbenog sustava s kvadrirotorskim propulzijskim pogonom. Nova metoda vremenske integracije jediničnih kvaterniona ne operira u vektorskom prostoru globalne parametarizacije rotacije letećeg objekta, već se inkrementalna integracija vrši na tangentnom prostoru rotacijske mnogostrukosti $SO(3)$, uz izravnu rekonstrukciju orijentacije na mnogostrukost jediničnih kvaterniona $SU(2)$. Razlog odabira spomenutog simulacijskog modela, temelji se u činjenici da nova metoda omogućava veću točnost numeričke integracije tijekom kinematičke rekonstrukcije orijentacije (stava) letećeg objekta, što je dobra naznaka za numeričku učinkovitost cjelokupne integracijske procedure, kao i mogućnost njene izvedbe u stvarnom vremenu.

S obzirom na gornje rečeno, u radu je potrebno:

1. Odabrati najprikladniju programsku paradigmu, tj. programski jezik i programsko okruženje koje podržava Raspberry Pi, za prijenos simulacijskog modela u stvarnom vremenu.
2. Simulacijski model se izvorno temelji na MATLAB kodu te ga kao takvog treba na optimalan način prebaciti u odabranu paradigmu.
3. Prije same implementacije postupka na Raspberry Pi, potrebno je provesti usporedne ('off-line') simulacije odabranog modela u odabranoj paradigmi na računalu s velikim računalnim resursima kroz testiranja, usporedbu s izvornim kodom i doradu do postizanja izvedbe u stvarnom vremenu.
4. Nakon implementacije, ocijeniti numeričku točnost i efikasnost algoritama integracije kvaterniona na Lievoj grupi u usporedbi sa standardnim modelima integracije s primjenom na rekonstrukciju rotacijskog gibanja kvadrirotorske letjelice kroz platformske ('on-line') simulacija na Raspberry Pi.

U radu je potrebno navesti korištenu literaturu i eventualno dobivenu pomoć.

Zadatak zadan:	Datum predaje rada:	Predviđeni datum obrane:
30. travnja 2020.	2. srpnja 2020.	6. – 10.7.2020.

Zadatak zadao: Predsjednica Povjerenstva:

Doc. dr. sc. Dario Zlatar Prof. dr. sc. Tanja Jurčević Lulić

Sadržaj

1 UVOD **1**
1.1 Rotacije kao linearne transformacije 1
1.2 Rotacije kao elementi grupe $SO(3)$ 2
1.3 Kvaternioni 4
1.4 Jednadžba kinematičke rekonstrukcije 19

2 GEOMETRIJSKA INTEGRACIJA **21**
2.1 Uvod u geometrijsku integraciju 21
2.2 Lieve grupa i Lieva algebra 21
2.3 Nova metoda integracije 24
2.4 Dinamički model kvadrirotorske letjelice 29
2.5 Implementacija nove numeričke metode vremenske integracije jediničnih kvaterniona 31

3 PROGRAMSKE PARADIGME U KONTEKSTU NUMERIČKIH SIMULACIJA **38**
3.1 Programske paradigme 38
3.2 Interpretirani jezici 40
3.3 Kompajlirani jezici 41
3.4 Simulacija na ugradbenim sustavima 42
3.5 Rasperry Pi 44
3.6 Dron 46

4 NUMERIČKI EKSPERIMENT **47**
4.1 Kvadrirotorska letjelica 47
4.2 Implementacija u Pythonu 49
4.3 Implementacija u C-u 55
4.4 Usporedba 'online' implementacija 58

5 ZAKLJUČAK **59**

LITERATURA **61**

PRILOG **63**

Popis slika

1 Pravilo za množenje kvaterniona 6

2 Shema integracijske procedure DAE sustava na Lievoj grupi [1] . . . 28
3 Kvadrirotorska letjelica u X konfiguraciji [2] 31

4 Raspberry Pi B 44
5 Navio modul 45
6 Naze32 kontrolor leta 46

7 Trajektorija letjelice 49
8 Usporedba elementa kvaterniona iz [2] *i Python implementacije* . . . 49
9 Usporedba elementa vektora položaja iz [2] *i Python implementacije* 50
10 Usporedba procesorskih vremena izvornog Matlab i novog Python koda za korak integracije $h = 1e-4$ 50
11 Usporedba procesorskih vremena različitih implementacija eksponencijalne mape za korak integracije $h = 1e-4$ 52
12 Usporedba procesorskih vremena različitih implementacija eksponencijalne mape za korak integracije $h = 1e-3$ 52
13 Usporedba procesorskih vremena izvornog Matlab i novog Python koda za korak integracije $h = 1e-2$ 53
14 Usporedba procesorskih vremena optimiziranog Python koda između usporedne ('off-line') i platformske ('on-line') simulacije za korak integracije $h = 1e-2$ 54
15 Usporedba procesorskih vremena izvornog Python i Cython koda za korak integracije $h = 1e-4$ 55
16 Usporedba procesorskih vremena izvornog Matlab i novog C koda za korak integracije $h = 1e-3$ 56
17 Usporedba procesorskih vremena izvornog Matlab i novog C koda za korak integracije $h = 1e-4$ 56
18 Usporedba procesorskih vremena optimiziranog C koda između usporedne ('off-line') i platformske ('on-line') simulacije za korak integracije $h = 1e-4$ 57
19 Usporedba procesorskih vremena C i Python koda za platformsku ('on-line') simulacije s korakom integracije $h = 1e-2$ 58

Popis tablica

1 Karakteristike kvadrirotorske letjelice [2] 47
2 Usporedba implementacija eksponencijalnih mapa 51

Popis oznaka

Oznaka	**Jedinica**	**Opis**
$\mathbf{i}, \mathbf{j}, \mathbf{k}$	-	Ortonormalni vektori
$\mathbf{R}$	-	Matrica rotacije
i, j, k	-	Imaginarne jedinice
u, z, w	-	Kompleksni brojevi
u^*, z^*, w^*	-	Kompleksno - konjugirani brojevi
$\mathbf{q}$	-	Matrica kompleksnog broja
$\mathbf{q}^*$	-	Kompleksno konjugirana matrica kompleksnog broja
$\mathbf{p}$	-	Imaginarni (čisti) kvaternion
p, t	-	Kvaternioni
$\mathbf{u}, \mathbf{v}$	-	Vektori
$\mathbf{I}^{n\times n}$	-	$n \times n$ jedinična matrica
e_0, e_1, e_2, e_3	-	Eulerovi parametri
ξ, η, ζ	-	Osi lokalnog koordinatnog sustava
$\mathbf{X}$	-	Generator rotacije
$\mathbf{L}$	-	Matrica transformacije kvaterniona
ω	rad/s	Kutna brzina
ω'	rad/s	Kutna brzina definirana u lokalnm koordinatnom sustavu
l	m	Duljina ruke
m	kg	Masa
F_Ti	N	Pogonska sila
T_Di	Nm	Moment parazitskog otpora
$\mathbf{J}$	kg m^2	Matrica inercije
J_{mzi}	kg m^2	Matrica inercije motor rotor

SAŽETAK

Tema ovog rada je implementacija integracijske procedure kinematičke rekonstrukcije orijentacije kvadrirotorske letjelice te provjera simulacijskog modela u stvarnom vremenu na računalu Raspberry Pi. U prvom poglavlju su između formalizama za opis rotacija odabrane i opisane matrice rotacije kao elementi $SO(3)$ grupe te jedinični kvaternioni iz $SU(2)$ grupe nakon čega su izvedene relacije temeljene na Eulerovim parametrima. Poglavlje završava izvodom jednadžbe kinematičke rekonstrukcije orijentacije. U drugom poglavlju dan je pregled nove metode vremenske integracije jediničnih kvaterniona koja inkrementalnu integraciju vrši na tangentnom prostoru rotacijske mnogostrukosti $SO(3)$ uz izravnu rekonstrukciju orijentacije na mnogostrukosti jediničnih kvaterniona $SU(2)$. Poglavlje završava opisom i komentarom detalja implementacije pojedinih relevantnih funkcija implementirane integracijske procedure. U trećem poglavlju opisane su i uspoređene programske paradigme u kontekstu numeričkih simulacija i znanstvenog računanja te je dan kratak opis hardverske infrastrukture na kojoj su simulacije realizirane. U zadnjem poglavlju, nakon validacije modela i rezultata, dan je niz usporedbi procesorskih vremena za 'off-line' i 'on-line' simulacije s različitim koracima integracije.

Ključne riječi: kvadrirotorska letjelica, numerička integracija, $SO(3)$ grupa, $SU(2)$ grupa, Raspberry Pi

SUMMARY

The topic of this paper is the implementation of the integration procedure for kinematic reconstruction of attitude for a quadrotor aerial vehicle and validation of a real-time simulation model on a Raspberry Pi computer. In the first chapter, between the formalisms for the description of rotations, the rotation matrices were selected and described as elements of the $SO(3)$ group and unit quaternions from the $SU(2)$ group, after which relations based on Euler parameters were derived. The chapter ends with a derivation of the equation of kinematic reconstruction of attitude. The second chapter provides an brief overview of a new integration method of unit quaternions that performs incremental integration on the tangential space of the rotational manifold $SO(3)$ with a direct reconstruction of the orientation on the manifold of unit quaternions $SU(2)$. The chapter ends with a description and commentary on the details of the implementation of certain relevant functions. The third chapter describes and compares programming paradigms in the context of numerical simulations and scientific computing, and gives a brief description of the hardware infrastructure on which the simulations were implemented. In the last chapter, after model and result validation, a series of comparisons of processor times for 'off-line' and 'on-line' simulations with different integration steps are given.

Keywords: quadrotor aerial vehicle, numerical integration, $SO(3)$ group, $SU(2)$ group, Raspberry Pi

1 UVOD

Značajan izazov pri projektiranju kontrolnih algoritama predstavlja inherentno visok stupanj nelinearnosti dinamike takvih letjelica. Stoga su algoritmi za simulaciju izravne dinamičke zadaće potrebni za pouzdanu procjenu položaja i orijentacije letjelice. U prvom ćemo poglavlju predstaviti ekvivalentne opise rotacija krutih tijela te pokazati njihovu vezu koju ćemo iskoristiti u drugom poglavlju. Poglavlje završava definiranjem dinamičkog modela kvadrirotorske letjelice te opisom konvencionalnog načina što će ujedno biti i uvod za drugo poglavlje.

1.1 Rotacije kao linearne transformacije

Poznato je da gibanje čestice u Euklidskom prostoru možemo smatrati opisanim ako znamo njezin položaj u svakom vremenskom trenutku u odnosu na inercijski Kartezijski koordinatni sustav. Za definiranje inercijskog koordinatnog sustava odabiremo tri ortonormalne osi pri čemu je položaj čestice opisan trojkom $(x, y, z) \in \mathbb{R}^3$. Svaka koordinata daje nam projekciju položaja čestice na odgovarajuću os. Putanja čestice opisana je parametriziranom krivuljom $p(t) = (x(t), y(t), z(t)) \in \mathbb{R}^3$. Međutim, nama su od interesa kruta tijela, a ne čestice, stoga je potrebno definirati kruto tijelo. Kruto tijelo je skup čestica takav da udaljenost između bilo koje dvije čestice ostaje stalna, neovisno o gibanju tijela ili bilo kakvim vanjskim silama koje djeluju na tijelo. Tu tvrdnju možemo formalizirati koristeći sljedeću relaciju:

$$\|p(t) - q(t)\| = \|p(0) - q(0)\| = konst. \tag{1}$$

Ako se objekt opiše kao podskup O od $\mathbb{R}^3$, gibanje krutog tijela opisano je kontinuiranim preslikavanjem $g(t) : O \to \mathbb{R}^3$. Takva transformacija sadrži opis preslikavanja svake točke relativno inercijskom koordinatnom sustavu. Uvjet da udaljenost između točaka krutog tijela ostaje sačuvana prilikom rotacije je nužan, ali ne i dovoljan, da bi preslikavanje bilo jednoznačno definirano. Primjerice, preslikavanje čuva udaljenosti, ali orijentacija tijela ne mora biti očuvana. Da bismo eliminirali tu mogućnost, zahtijevamo da je vektorski umnožak između vektora također sačuvan. Dakle, gibanje krutog tijela možemo definirati kao preslikavanje iz $\mathbb{R}^3$ u $\mathbb{R}^3$ koje mora zadovoljavati sljedeće uvjete:

1. *Očuvanje udaljenosti*: $\|g(p) - g(q)\| = \|p - q\|$ za sve točke p, $q \in \mathbb{R}^3$,
2. *Očuvanje vektorskog umnoška*: $g_*(v \times w) = g_*(v) \times g_*(w)$ za sve vektore $v, w \in \mathbb{R}^3$.

Iz gornje definicije, vidljivo je da je skalarni produkt invarijantan s obzirom na linearne transformacije, što znači da duljina vektora prije i nakon transformacije ostaje jednaka. Koristeći identitet polarizacije, prema [3]:

$$v_1^T v_2 = \frac{1}{4}(\|v_1 + v_2\|^2 - \|v_1 - v_2\|^2), \tag{2}$$

te relacijama

$$\|v_1 + v_2\| = \|g_*(v_1) + g_*(v_2)\|\,, \quad \|v_1 - v_2\| = \|g_*(v_1) - g_*(v_2)\|\,, \tag{3}$$

možemo zaključiti da za bilo koja dva vektora v_1 i v_2 vrijedi:

$$v_1^T v_2 = g_*(v_1)^T g_*(v_2), \tag{4}$$

čime smo pokazali očuvanje skalarnog umnoška što onda implicira očuvanje udaljenosti. Dakle duljina ortogonalnih vektora tijekom linearnih transformacija ostaje sačuvana.

1.2 Rotacije kao elementi grupe $SO(3)$

Sada ćemo definirati ortonormalne vektore $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z} \in \mathbb{R}^3$.Pri tome, ako vrijedi:

$$\mathbf{z} = \mathbf{x} \times \mathbf{y}, \tag{5}$$

onda oni definiraju desni koordinatni sustav. Linearne transformacije nakon uvođenja baze $\mathbb{R}^3$, mogu biti predstavljene matricama. Nakon definiranja ortonormalne baze $\mathbb{R}^3$, svaka rotacija je opisana matricom kojoj su $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3 \in \mathbb{R}^3$ stupci. Znamo da su stupci matrice $\mathbf{R}$ međusobno ortonormalni, iz čega slijedi da:

$$r_i^T r_j = \begin{cases} 0, & \text{ako } i \neq j \\ 1, & \text{ako } i = j. \end{cases} \tag{6}$$

Ove uvjete možemo preformulirati i oni mogu biti napisani kao:

$$\mathbf{R}\mathbf{R}^T = \mathbf{R}^T\mathbf{R} = \mathbf{I}. \tag{7}$$

Taj izraz onda implicira:

$$det\,\mathbf{R} = \pm 1. \tag{8}$$

Na temelju gore definiranih svojstava, možemo definirati prostor matrice rotacije kao:

$$SO(3) = \{\mathbf{R} \in \mathbb{R}^{3x3} \,|\, \mathbf{R}\mathbf{R}^T = I, det\,\mathbf{R} = +1\}. \tag{9}$$

Za određivanje ima li skup matrica strukturu grupe, potrebno je definirati aksiome grupe. Da bi skup svih matrica koje opisuju rotacije tijela bio grupa G, mora zadovoljiti binarnu operaciju $\circ$ definiranu na elementima skupa, te aksiome:

1. *Zatvorenost*: Ako su g_1 i $g_2 \in G$, onda vrijedi $g_1 \circ g_2 \in G$.

2. *Jedinični element*: Postoji jedinični element e tako da vrijedi $g \circ e = e \circ g = g$, za svaki $g \in G$.

3. *Inverzni element*: Za svaki $g \in G$ postoji jedinstveni inverzni element $g^{-1} \in G$, takav da vrijedi $g \circ g^{-1} = g^{-1} \circ g = e$.

4. *Asocijativnost*: Ako su $g_1, g_2, g_3 \in G$, onda vrijedi $(g_1 \circ g_2) \circ g_3 = g_1 \circ (g_2 \circ g_3)$.

Da bi pokazali da je $SO(3) \subset \mathbb{R}^{3x3}$ grupa zatvorena s obzirom na operaciju množenja, potrebno je pokazati da zadovoljava gore navedene aksiome:

1. ako su $\mathbf{R_1}, \mathbf{R_2} \in SO(3)$, tada $\mathbf{R_1}\mathbf{R_2} \in SO(3)$ jer vrijedi

$$\mathbf{R_1}\mathbf{R_2}(\mathbf{R_1}\mathbf{R_2})^T = \mathbf{R_1}\mathbf{R_2}\mathbf{R_2}^T\mathbf{R_1}^T = \mathbf{R_1}\mathbf{R_1}^T = \mathbf{I}\ ,$$
$$det(\mathbf{R_1}\mathbf{R_2}) = det(\mathbf{R_1})det(\mathbf{R_2}) = +1.$$

2. Jedinična matrica $\mathbf{I}$ jedinični je element.

3. Iz izraza (7) slijedi da je matrica $\mathbf{R}^T \in SO(3)$ inverz matrice $\mathbf{R} \in SO(3)$.

4. Asocijativnost $SO(3)$ grupe slijedi iz asocijativnosti umnoška matrica, dakle vrijedi $\mathbf{R_1}, \mathbf{R_2}, \mathbf{R_3} \in SO(3)$, iz čega slijedi $(\mathbf{R_1}\mathbf{R_2})\mathbf{R_3} = \mathbf{R_1}(\mathbf{R_2}\mathbf{R_3})$.

Vidimo da spomenuta svojstva matrica rotacije zadovoljavaju aksiome grupe na temelju čega možemo reći da skup svih ortogonalnih matrica čini grupu koju označavamo $SO(3)$. Grupu $SO(3)$ nazivamo specijalna ortonormalna grupa, a matrice koja ona sadrži nazivamo čistim rotacijama. Zanimljivo je napomenuti da podskup matrica koje imaju determinantu -1 ne zadovoljava aksiome grupe. Točnije prema (1.3) nije zadovoljen 1) aksiom grupe jer umnožak dvije matrice iz tog skupa daje matricu koja nije u tom skupu. Formalno govoreći, to znači da skup nije zatvoren pod operacijom množenja. Oba podskupa zajedno čine grupu $O(3)$. Bilo koja konfiguracija tijela koje može slobodno rotirati relativno inercijskom koordinatnom sustavu može biti identificirana s jedinstvenom matricom $\mathbf{R} \in SO(3)$. Sukladno tome, grupu $SO(3)$ možemo identificirati kao konfiguracijski prostor sustava pri čemu je putanja sustava krivulja $R(t) \in SO(3)$ za sve $t \in [0, T]$. Konfiguracijski prostor sustava je apstraktni prostor u kojem svaka točka tog prostora opisuje stanje sustava. Prateći krivulju $R(t)$, možemo pratiti evoluciju sustava u vremenu.

1.3 Kvaternioni

Kvaternioni su jedan od mogućih načina parametrizacije rotacije. Veći dio izlaganja u ovom poglavlju slijedi [4]. Za dvodimenzionalni slučaj rotacije u ravnini, oko ishodišta O za kut θ linearnu transformaciju možemo napisati kao:

$$\mathbf{R}_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}. \tag{10}$$

Ako točku (x, y) želimo rotirati za kut θ, pomnožimo ju matricom rotacije:

$$\mathbf{R}_\theta \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}. \tag{11}$$

Budući da množimo s lijeva, množeći prvo matricom $\mathbf{R}\phi$ onda $\mathbf{R}_\theta$ daje isti produkt kao da pomnožimo prvo matricom $\mathbf{R}_\theta$ s matricom $\mathbf{R}_\phi$. Redoslijed množenja nije bitan, što znači da rotacije u dvije dimenzije komutiraju, što neće biti slučaj za trodimenzionalne rotacije. Dakle, uzastopne rotacije možemo prikazati kao množenje matrica. Matrica $\mathbf{R}_\theta$ za sve kuteve θ formira grupu koju nazivamo specijalna ortogonalna grupa i označavamo je sa $SO(2)$. Svaku rotaciju u $\mathbf{R}_\theta$ dvodimenzionalnog prostora možemo predstaviti i kompleksnim brojem:

$$z_\theta = \cos\theta + i\sin\theta. \tag{12}$$

Jednostavno je pokazati da je množenje neke proizvoljne točke (x, y) kompleksnim brojem z_θ ekvivalentno množenju točke matricom rotacije:

$$\begin{aligned} z_\theta(x + iy) &= (\cos\theta + i\sin\theta)(x + iy) \\ &= x\cos\theta - y\sin\theta + i(x\sin\theta + y\cos\theta) \\ &= (x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta), \end{aligned} \tag{13}$$

iz čega je vidljivo da je rezultat isti kao u izrazu (11). Vidjeli smo da se dvodimenzionalne rotacije mogu opisati imaginarnim brojevima, dok ćemo za opis rotacija u trodimenzionalnom prostoru koristiti kvaternione. Kvaternionski sustav brojeva rezultat je trenutka inspiracije irskog fizičara Williama Rowana Hamiltona (1805 – 1865). Kao što su imaginarni brojevi dvodimenzionalna nadgradnja realnih brojeva, tako su kvaternioni četverodimenzionalna nadgradnja imaginarnih brojeva koji su se kasnije pokazali kao iznimno efikasan alat za opis rotacija. Operacije s kvaternionima imaju vrlo zanimljivu geometrijsku interpretaciju što se može vidjeti na [5]. Hamilton je dobar dio života proveo tražeći trodimenzionalni sustav brojeva analogan kompleksnim brojevima, međutim, problem je bio s množenjem trodimenzionalnih brojeva. To je bio problem koji je i priznao svome sinu u pismu, kojem je na pitanje zna li množiti trojke, odgovorio da ih zna samo zbrajati i oduzimati

[6]. Rješenje nije bilo dodati jednu dimenziju nego dvije dimenzije kompleksnim brojevima. Dakle, kvaternion se sastoji od jednog realnog i tri imaginarna dijela. Fundamentalna jednadžba kvaterniona je:

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1. \tag{14}$$

Skup kvaterniona pod operacijom zbrajanja i množenja zadovoljava sve aksiome polja, osim komutativnosti. Kvaternion također možemo smatrati četvorkom realnih brojeva, kao element $\mathbb{R}^4$ što možemo zapisati kao

$$q = (q_0, q_1, q_2, q_3), \tag{15}$$

gdje su q_0, q_1, q_2 i q_3 skalari koje nazivamo komponentama kvaterniona. Kvaternion možemo zapisati i matrično:

$$\mathbf{q} = \begin{bmatrix} a + id & -b - ic \\ b - ic & a - id \end{bmatrix}. \tag{16}$$

Skup matrica ovog oblika, koje imaju determinantu 1, formiraju specijalnu unitarnu grupu $SU(2)$. Prostor takve grupe možemo formalno definirati kao:

$$SU(2) = \left\{ \begin{bmatrix} z & -w^* \\ w & z^* \end{bmatrix} \quad | \quad z, w \in \mathbb{C}, |z|^2 + |w|^2 = 1 \right\}, \tag{17}$$

pri čemu su z^* i w^* kompleksno konjugirani imaginarni brojevi. Napomenimo još da je ova grupa izomorfna grupi jediničnih kvaterniona što je tvrdnja čije ćemo implikacije vidjeti kasnije u radu. Jasno da zbroj i umnožak dvije takve matrice opet daje matricu istog oblika. Kvadrat apsolutne vrijednosti $|q|^2$ kvaterniona q definiran je kao determinanta:

$$\det \mathbf{q} = det \begin{bmatrix} a + id & -b - ic \\ b - ic & a - id \end{bmatrix} = a^2 + b^2 + c^2 + d^2. \tag{18}$$

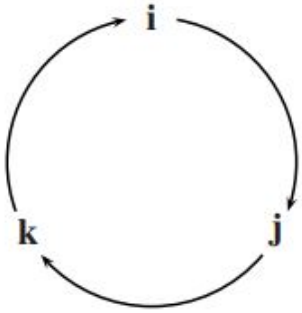
Dakle, $|q|^2$ kvadrat je udaljenosti točke (a, b, c, d) od ishodišta u $\mathbb{R}^4$. Kvaternioni ne zadovoljavaju zakon komutacije, što se može jednostavno pokazati. Prvo zapišemo:

$$\begin{bmatrix} a + id & -b - ic \\ b - ic & a - id \end{bmatrix} = a\mathbf{1} + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}, \tag{19}$$

pri čemu su

$$\mathbf{1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{i} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}, \quad \mathbf{j} = \begin{bmatrix} 0 & -i \\ -i & 0 \end{bmatrix}, \quad \mathbf{1} = \begin{bmatrix} i & 0 \\ 0 & -i \end{bmatrix}. \tag{20}$$

Iz gornjih izraza jasno je da vrijedi $\mathbf{i}^2 = -\mathbf{1}$ što je relacija poznata iz algebre imaginarnih brojeva. Međutim, sada definiramo i $\mathbf{j}^2 = \mathbf{k}^2 = -\mathbf{1}$. Nekomutacija kvaternionskog sustava sadržana je u umnoščima $\mathbf{i}$, $\mathbf{j}$, $\mathbf{k}$ koje shematski možemo prikazati



Slika 1: Pravilo za množenje kvaterniona

na slici 1. Primjerice $\mathbf{ij} = \mathbf{k}$ ili $\mathbf{ji} = -\mathbf{k}$ što dalje implicira $\mathbf{ij} \neq \mathbf{ji}$.
Popis svih relacija prikazan je ispod:

$$\begin{aligned} \mathbf{ij} = \mathbf{i} \times \mathbf{j} = \mathbf{k} = -\mathbf{j} \times \mathbf{i} = -\mathbf{ji} \\ \mathbf{jk} = \mathbf{j} \times \mathbf{k} = \mathbf{i} = -\mathbf{k} \times \mathbf{j} = -\mathbf{kj} \\ \mathbf{ki} = \mathbf{k} \times \mathbf{i} = \mathbf{j} = -\mathbf{i} \times \mathbf{k} = -\mathbf{ik}. \end{aligned} \tag{21}$$

Svojstvo kvaterniona da ne komutiraju zapravo je dobro, jer omogućuje da se kvaternionima opisuju transformacije koje ne komutiraju, što su, između ostalog, rotacije u tri ili četri dimenzije. Svojstvo kvaterniona da se može zapisati kao matrica možemo iskoristiti da izvedemo neke korisne relacije. Apsolutna vrijednost ima multiplikativno svojstvo $|\mathbf{q}_1\mathbf{q}_2| = |\mathbf{q}_1||\mathbf{q}_2|$ koje slijedi iz svojstva determinante matrice:

$$\det(\mathbf{q}_1\mathbf{q}_2) = \det(\mathbf{q}_1)\det(\mathbf{q}_2). \tag{22}$$

Svaki kvaternion $\mathbf{q}$ koji je različit od nule ima inverz $\mathbf{q}^{-1}$, što odgovara inverzu matrice. Inverz kvaternionu $\mathbf{q} = a\mathbf{1} + b\mathbf{i} + c\mathbf{j} + d\mathbf{k} \neq 0$ možemo računati prema relaciji:

$$\mathbf{q}^{-1} = \frac{1}{a^2 + b^2 + c^2 + d^2}(a\mathbf{1} + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}). \tag{23}$$

Kompleksno konjugirani par kvaterniona $\mathbf{q}$ ima oblik $\mathbf{q}^* = a\mathbf{1} - b\mathbf{i} - c\mathbf{j} - d\mathbf{k}$ pa vrijedi:

$$\mathbf{q}\mathbf{q}^* = a^2 + b^2 + c^2 + d^2 = \|q\|^2. \tag{24}$$

Kompleksno konjugirani kvaternion nije rezultat kompleksne konjugacije svakog člana u matrici $\mathbf{q}$. Kompleksno konjugirani kvaternion je rezultat kompleksne konjugacije svakog elementa transponirane matrice $\mathbf{q}^T$. Iz poznate relacije $(\mathbf{q}_1\mathbf{q}_2)^T = \mathbf{q}_2^T\mathbf{q}_1^T$ slijedi $(\mathbf{q}_1\mathbf{q}_2)^* = \mathbf{q}_2^*\mathbf{q}_1^*$. Kvaternion $\mathbf{q} = a\mathbf{1} + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$ kojemu je apsolutna vrijednost 1, ili jedinični kvaternion, zadovoljava uvjet:

$$a^2 + b^2 + c^2 + d^2 = 1. \tag{25}$$

Na temelju toga možemo zaključiti da takvi kvaternioni formiraju analog sfere, jediničnu sferu S^3 u prostoru $\mathbb{R}^4$. Iz svojstava multiplikacije slijedi da je umnožak jediničnih kvaterniona opet jedinični kvaternion iz čega onda dalje možemo zaključiti

da je S^3 grupa

$$S^3 = \left\{ \mathbf{q} \in \mathbb{R}^4 \quad | \quad |q| = 1 \right\}. \tag{26}$$

Da bismo dokazali da je skup S^3 grupa, potrebno je dokazati da skup zadovoljava aksiome:

1. *Zatvorenost*: Ako su g i $s \in S$, onda vrijedi $g \circ s \in S^3$.
2. *Jedinični element*: Jedinični element je kvaternion $q = (1, \mathbf{0})$.
3. *Inverzni element*: Inverzni element za svaki $q \in S^3$ je $q^* \in S^3$.
4. *Asocijativnost*: Ako su q, s, $p \in S^3$, tada vrijedi $(q \circ s) \circ p = q \circ (s \circ p)$.

Da bi se dokazalo svojstvo asocijativnosti, moramo pokazati množenje svih mogućih kombinacija baznih vektora. Iz baze $(1, i, j, k)$ odmah možemo ukloniti skalar 1 za kojeg znamo da vrijedi svojstvo asocijativnosti. Nakon toga nam ostaju tri člana, koji daju ukupno 27 mogućih kombinacije koje će biti pokazane u izrazu ispod.

$$
\begin{aligned}
(ii)i = -i = i(-1) = i(ii)\\
(ii)j = -j = ik = i(k) = i(ij)\\
(ii)k = -k = -ij = i(-j) = i(ik)\\
(ij)i = ki = j = -ik = i(-k) = i(ji)\\
(ij)j = kj = -i = i(-1) = i(jj)\\
(ij)k = kk = -1 = ii = i(i) = i(jk)\\
(ik)i = -ji = k = ij = i(j) = i(ki)\\
ik(j) = -jj = 1 = i(-i) = i(kj)\\
(ik)k = -jk = -i = i(-1) = i(kk)\\
(ji)i = -ki = -j = j(-1) = j(ii)\\
(ji)j = -kj = i = jk = j(k) = j(ij)\\
(ji)k = -kk = 1 = -jj = j(-j) = j(ik)\\
(jj)i = -i = -jk = j(-k) = j(ii)\\
jj(j) = -j = j(-1) = j(jj)\\
(jj)k = -k = ji = j(i) = j(jk)\\
(jk)i = ii = -1 = jj = j(j) = j(ki)\\
(jk)j = ij = k = -ji = j(-i) = j(kj)\\
(jk)k = ik = -j = j(-1) = j(kk)\\
(ki)i = ji = -k = k(-1) = k(ii)\\
(ki)j = jj = -1 = kk = k(k) = k(ij)\\
(ki)k = jk = i = -kj = k(-j) = k(ik)\\
(kj)i = -ii = 1 = -kk = k(-k) = k(ji)\\
(kj)j = -ij = -k = k(-1) = k(jj)\\
(kj)k = -ik = j = ki = k(i) = k(jk)\\
kk(i) = -i = kj = k(j) = k(ki)\\
(kk)j = -j = -ki = k(-i) = k(kj)\\
(kk)k = -k = k(-1) = k(kk).
\end{aligned}
\tag{27}
$$

U prethodnom smo izrazu dokazali da za sve moguće kombinacije množenja elemenata vrijedi svojstvo asocijativnosti. Dakle, kao i grupa matrica rotacija $SO(3)$, skup jediničnih kvaterniona zadovoljava svojstva grupe.

Vidjeli smo da množenje kompleksnih brojeva ima geometrijske implikacije. Pretpostavimo sada da je u kompleksni broj kome je apsolutna vrijednost 1. Trivijalno je pokazati da je množenjem brojem u rotacija. Uzmimo da su v i w kompleksni

brojevi koje pomnožimo brojem u. Udaljenost dva kompleksna broja sada možemo zapisati kao:

$$\|uv - uw\|. \tag{28}$$

Svojstvo distributivnosti nam dopušta da izlučimo u

$$\|u(v - w)\|, \tag{29}$$

iz čega dalje, na temelju multiplikativnog svojstva apsolutne vrijednosti, možemo napisati:

$$\|u\| \, \|v - w\|. \tag{30}$$

Znamo da smo krenuli s pretpostavkom da je $\|u\| = 1$ što znači da je udaljenost $\|v - w\|$ ostala sačuvana. Ovo je isti rezultat koji smo dokazali i na samom početku rada (4) gdje smo na temelju toga zaključili da je linearna transformacija koja čuva udaljenost između točaka zapravo rotacija. Istom linijom zaključivanja možemo poopćiti i množenje kvaterniona koje također čuva udaljenosti. Imaginarni dio kvaterniona (imaginarni kvaternion), dakle bez realnog dijela, možemo napisati kao:

$$\mathbf{p} = b\mathbf{i} + c\mathbf{j} + d\mathbf{k}. \tag{31}$$

Komponente $\mathbf{i}$, $\mathbf{j}$ i $\mathbf{k}$ kvaterniona 'razapinju' trodimenzionalni prostor koji možemo označiti $\mathbb{R}\mathbf{i} + \mathbb{R}\mathbf{j} + \mathbb{R}\mathbf{k}$ što je ekvivalentno prethodno definiranom $\mathbb{R}^3$. Prostor $\mathbb{R}\mathbf{i} + \mathbb{R}\mathbf{j} + \mathbb{R}\mathbf{k}$ je ortogonalni komplement liniji $\mathbb{R}\mathbf{1}$. Nju čine kvaternioni oblika $a\mathbf{1}$ koje ćemo jednostavno označavati s a i možemo ih intuitivno shvatiti kao liniju realnih brojeva. Zbroj bilo koja dva člana prostora $\mathbb{R}\mathbf{i} + \mathbb{R}\mathbf{j} + \mathbb{R}\mathbf{k}$ opet je član $\mathbb{R}\mathbf{i} + \mathbb{R}\mathbf{j} + \mathbb{R}\mathbf{k}$ što, međutim, ne vrijedi za umnoške. To se može lako pokazati ako uzmemo da je $\mathbf{u} = u_1\mathbf{i} + u_2\mathbf{j} + u_3\mathbf{k}$, a $\mathbf{v} = v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k}$. Tada je njihov umnožak:

$$\begin{aligned} \mathbf{uv} = &- (u_1v_1 + u_2v_2 + u_3v_3) \\ &+ (u_2v_3 - u_3v_2)\mathbf{i} - (u_1v_3 - u_3v_1)\mathbf{j} + (u_1v_2 - u_2v_1)\mathbf{k}. \end{aligned} \tag{32}$$

U gornjem izrazu lako je prepoznati da prvi dio desne strane predstavlja dobro poznati skalarni umnožak:

$$\mathbf{u} \cdot \mathbf{v} = u_1v_1 + u_2v_2 + u_3v_3, \tag{33}$$

dok drugi dio odgovara vektorskom umnošku

$$\mathbf{u} \times \mathbf{v} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \end{vmatrix} = (u_2v_3 - u_3v_2)\mathbf{i} - (u_1v_3 - u_3v_1)\mathbf{j} + (u_1v_2 - u_2v_1)\mathbf{k}. \tag{34}$$

Na temelju toga, možemo napisati

$$\mathbf{uv} = -\mathbf{u} \cdot \mathbf{v} + \mathbf{u} \times \mathbf{v}. \tag{35}$$

Znamo da je skalarni umnožak $\mathbf{u}\cdot\mathbf{v}$ realni broj, stoga je umnožak $\mathbf{uv}$ u $\mathbb{R}\mathbf{i}+\mathbb{R}\mathbf{j}+\mathbb{R}\mathbf{k}$ samo ako vrijedi $\mathbf{u}\cdot\mathbf{v}=0$, što je istinito samo ako je $\mathbf{u}$ okomit na $\mathbf{v}$. S druge strane, umnožak $\mathbf{uv}=-\mathbf{u}\cdot\mathbf{v}+\mathbf{u}\times\mathbf{v}$ realni je broj samo ako je $\mathbf{u}\times\mathbf{v}=0$ što je slučaj samo ako su $\mathbf{u}$ i $\mathbf{v}$ istog ili suprotnog smjera. Dakle ako imamo vektor $\mathbf{u}\in\mathbb{R}\mathbf{i}+\mathbb{R}\mathbf{j}+\mathbb{R}\mathbf{k}$ i vrijedi $\|\mathbf{u}\|=1$ znači da $\mathbf{u}$ jedinični vektor za kojeg vrijedi:

$$\mathbf{u}^2=-\mathbf{u}\cdot\mathbf{u}=-|\mathbf{u}|^2=-1. \tag{36}$$

Uzmimo sada da jedinični kvaternion t ima realni dio $\cos u$ te samo jednu imaginarnu komponentu $\mathbf{u}$ koju množimo s realnim brojem $\sin u$.

$$t=\cos u+\mathbf{u}\sin u. \tag{37}$$

Znamo da je $\mathbf{u}$ jedinični vektor u $\mathbb{R}\mathbf{i}+\mathbb{R}\mathbf{j}+\mathbb{R}\mathbf{k}$ za kojeg vrijedi $\mathbf{u}^2=-1$. Takav kvaternion inducira rotaciju $\mathbb{R}\mathbf{i}+\mathbb{R}\mathbf{j}+\mathbb{R}\mathbf{k}$, ali ne množenjem kao kod matrica, jer u općem slučaju, umnožak t i čistog kvaterniona $\mathbf{q}$ koji pripada prostoru $\mathbb{R}\mathbf{i}+\mathbb{R}\mathbf{j}+\mathbb{R}\mathbf{k}$ ne mora nužno pripadati prostoru $\mathbb{R}\mathbf{i}+\mathbb{R}\mathbf{j}+\mathbb{R}\mathbf{k}$. Zato definiramo $t^{-1}\mathbf{q}t$ čiji je rezultat element $\mathbb{R}\mathbf{i}+\mathbb{R}\mathbf{j}+\mathbb{R}\mathbf{k}$. Da bismo to dokazali, primijetimo da

$$t^{-1}=t^*/|t|^2=\cos u-\mathbf{u}\sin u. \tag{38}$$

Preslikavanje $q\to t^{-1}\mathbf{q}t$, koje nazivamo konjugacija, bijekcija je na $\mathbb{H}$, dakle preslikavanje $\mathbb{H}$ na samog sebe. Konjugacija s t također preslikava realnu liniju $\mathbb{R}$ na samu sebe, jer vrijedi $t^{-1}rt=r$. Dakle takva konjugacija neki realni broj r preslikava na samog sebe kao što preslikava i ortogonalni komplement $\mathbb{R}\mathbf{i}+\mathbb{R}\mathbf{j}+\mathbb{R}\mathbf{k}$ na samog sebe. Sada ćemo dokazati da konjugacija s t rotira $\mathbb{R}\mathbf{i}+\mathbb{R}\mathbf{j}+\mathbb{R}\mathbf{k}$ za kut $2u$. Uzmimo da je kvaternion $t=\cos u+\mathbf{u}\sin u$ pri čemu je $\mathbf{u}\in\mathbb{R}\mathbf{i}+\mathbb{R}\mathbf{j}+\mathbb{R}\mathbf{k}$. Prvo pokažimo da konjugacija ne utječe na liniju $\mathbb{R}\mathbf{u}$ realnih brojeva:

$$\begin{aligned}
t^{-1}\mathbf{u}t&=(\cos u-\mathbf{u}\sin u)\mathbf{u}(\cos u+\mathbf{u}\sin u)\\
&=(\mathbf{u}\cos u-\underbrace{\mathbf{u}^2}_{=-1}\sin u)(\cos u+\mathbf{u}\sin u)\\
&=(\mathbf{u}\cos u+\sin u)(\cos u+\mathbf{u}\sin u)\\
&=\mathbf{u}(\underbrace{\cos^2 u+\sin^2 u}_{=1})+\sin u\cos u+\underbrace{\mathbf{u}^2}_{=-1}\sin u\cos u\\
&=\mathbf{u}.
\end{aligned} \tag{39}$$

Konjugacija kvaternionom t je rotacija cijelog prostora $\mathbb{R}\mathbf{i}+\mathbb{R}\mathbf{j}+\mathbb{R}\mathbf{k}$. Da bi to vidjeli, uzmimo jedinični vektor $\mathbf{u}$ okomit na vektor $\mathbf{v}$ u $\mathbb{R}\mathbf{i}+\mathbb{R}\mathbf{j}+\mathbb{R}\mathbf{k}$ tako da vrijedi $\mathbf{u}\cdot\mathbf{v}=0$. Neka $\mathbf{w}=\mathbf{u}\times\mathbf{v}$ tako da $\mathbf{u},\mathbf{v},\mathbf{w}$ bude ortonormalna baza $\mathbb{R}\mathbf{i}+\mathbb{R}\mathbf{j}+\mathbb{R}\mathbf{k}$, isto kao u (5). Moramo pokazati da vrijedi:

$$t^{-1}\mathbf{v}t=\mathbf{v}\cos 2u-\mathbf{w}\sin 2u,\quad t^{-1}\mathbf{w}t=\mathbf{v}\sin 2u+\mathbf{w}\cos 2u, \tag{40}$$

jer to znači da konjugacija s t-om rotira vektore $\mathbf{v}$ i $\mathbf{w}$ i na taj način cijelu ravninu okomitu na liniju $\mathbb{R}\mathbf{u}$ za kut $2u$. To možemo pokazati sljedećim računom:

$$\begin{aligned} t^{-1}\mathbf{v}t &= (\cos u - \mathbf{u}\sin u)\mathbf{v}(cosu + \mathbf{u}\sin u) \\ &= (\mathbf{v}\cos u - \mathbf{uv}\sin u)(\cos u + \mathbf{u}\sin u) \\ &= \mathbf{v}^2\cos^2 u - \mathbf{uv}\sin u\cos u + \mathbf{vu}\sin u\cos u - \mathbf{uvu}\sin^2 u \\ &= \mathbf{v}\cos^2 u - 2\mathbf{uv}\sin u\cos u + \mathbf{u}^2\mathbf{v}\sin^2 u \\ &= \mathbf{v}(\cos^2 u - \sin^2 u) - 2\mathbf{w}\sin u\cos u \\ &= \mathbf{v}\cos 2u - \mathbf{w}\sin 2u. \end{aligned} \tag{41}$$

Na sličan se način može pokazati da vrijedi $t^{-1}\mathbf{w}t = \mathbf{v}\sin 2u + \mathbf{w}\cos 2u$. Ovime smo pokazali da je svaka rotacija u $\mathbb{R}^3$, u kojoj imamo os $\mathbf{u}$ i kut rotacije u, rezultat konjugacije jediničnim kvaternionom.

$$t = \cos\frac{u}{2} + \mathbf{u}\sin\frac{u}{2}. \tag{42}$$

Istu rotaciju inducira i $-t$, jer vrijedi

$$(-t)^{-1}\mathbf{s}(-t) = t^{-1}\mathbf{s}t. \tag{43}$$

Međutim, $\pm t$ jedini su kvaternioni koji induciraju ovu rotaciju. Znamo da se svaki jedinični kvaternion može jedinstveno izraziti u obliku $t = \cos\frac{u}{2} + \mathbf{u}\sin\frac{u}{2}$, a znamo također da je rotacija jedinstveno određena dvojkom $(\mathbf{u}, u)$ i $(-\mathbf{u}, -u)$. Za kvaternione t i $-t$ kažemo da su antipod jer na sferi jediničnih kvaterniona predstavljaju suprotne točke. Dakle, pokazali smo da rotacije u $\mathbb{R}^3$ odgovaraju antipodalnim parovima jediničnih kvaterniona. Iz te tvrdnje slijedi korolar da skup ovih rotacija formira grupu. Sada možemo razumjeti što znači kada se kaže da grupa jediničnih kvaterniona ima dvostruko pokrivanje grupe $SO(3)$. Dakle, dva elementa grupe jediničnih kvaterniona, odgovaraju jednoj matrici rotacije iz grupe $SO(3)$. Možemo još dokazati da je umnožak rotacija opet rotacija te da je inverz rotacije isto tako rotacija. Inverz rotacije oko osi $\mathbf{u}$ za kut u je rotacija oko osi $\mathbf{u}$ za kut $-u$. Prvo ćemo pokazati da umnožak dvije rotacije čini rotaciju. Uzmimo da r_1 rotacija oko osi $\mathbf{u}_1$ za kut u_1 a r_2 rotacija oko osi $\mathbf{u}_2$ za kut u_2. Tada rotaciju r_1 možemo napisati kao induciranu konjugacijom kvaternionom t_1:

$$t_1 = \cos\frac{u_1}{2} + \mathbf{u}_1\sin\frac{u_1}{2}, \tag{44}$$

dok, analogno, rotaciju r_2 možemo napisati

$$t_2 = \cos\frac{u_2}{2} + \mathbf{u}_2\sin\frac{u_2}{2}. \tag{45}$$

Rezultat prve rotacije r_1 pa zatim r_2

$$q \to t_2^{-1}(t_1^{-1}qt_1)t_2 = (t_1t_2)^{-1}q(t_1t_2). \tag{46}$$

Kako smo vidjeli, rotacija je jednoznačno određena ako znamo os rotacije i iznos kuta za koji se tijelo rotira. Sada nam se prirodno nameće Eulerov teorem. Prema njemu, svaka trodimenzionalna rotacija može biti prikazana kao rotacija oko fiksne osi za određeni kut. Koristeći Eulerove parametre, koji su se pokazali prikladnima za računalnu implementaciju, a nadovezujući se na prethodno razvijene spoznaje, razradit ćemo dalje Eulerove parametre. Uvodeći Eulerove parametre kao:

$$\begin{aligned} e_0 &= \cos\left(\left\|\frac{\mathbf{u}}{2}\right\|\right), \\ \mathbf{e} &= \frac{\mathbf{u}}{||u||}\sin\left(\left\|\frac{\mathbf{u}}{2}\right\|\right). \end{aligned} \tag{47}$$

Ako ih zapišemo u obliku kvaterniona:

$$p = (e_0, \mathbf{e}) = (\cos\left(\left\|\frac{u}{2}\right\|\right), \frac{\mathbf{u}}{||u||}\sin\left(\left\|\frac{\mathbf{u}}{2}\right\|\right). \tag{48}$$

Lako je pokazati da je norma gornjeg kvaterniona jednaka 1:

$$\begin{aligned} \|e\| = e_0^2 + \mathbf{e}^T\mathbf{e} &= \cos^2\left(\left\|\frac{u}{2}\right\|\right) + \frac{\sin^2\left(\left\|\frac{\mathbf{u}}{2}\right\|\right)}{\|\mathbf{u}\|^2}\mathbf{u}^T\mathbf{u} = \\ &= \cos^2\left(\left\|\frac{u}{2}\right\|\right) + \frac{\sin^2\left(\left\|\frac{\mathbf{u}}{2}\right\|\right)}{\|\mathbf{u}\|^2}\|\mathbf{u}\|^2 \\ &= \cos^2\left(\left\|\frac{u}{2}\right\|\right) + \sin^2\left(\left\|\frac{\mathbf{u}}{2}\right\|\right) = 1. \end{aligned} \tag{49}$$

Time smo dokazali da je kvaternion, koji sadrži Eulerove parametre, inherentno jedinični. Ako matricu rotacije $\mathbf{R}$ zapišemo preko Eulerovih parametara, koristeći jednadžbu:

$$\mathbf{R} = (2e_0^2 - 1)\mathbf{I} + 2e_0\tilde{\mathbf{e}} + 2\mathbf{e}\mathbf{e}^T, \tag{50}$$

iz čega dalje slijedi:

$$\begin{aligned} &\begin{bmatrix} 2e_0^2-1 & 0 & 0 \\ 0 & 2e_0^2-1 & 0 \\ 0 & 0 & 2e_0^2-1 \end{bmatrix} + 2\begin{bmatrix} 0 & -e_0e_3 & e_0e_2 \\ e_0e_3 & 0 & -e_0e_1 \\ -e_0e_2 & e_0e_1 & 0 \end{bmatrix} \\ &\qquad + 2\begin{bmatrix} e_1^2 & e_1e_2 & e_0e_3 \\ e_1e_2 & e_2^2 & e_2e_3 \\ e_1e_3 & e_2e_3 & e_3^2 \end{bmatrix}. \end{aligned} \tag{51}$$

Iz tog izraza, jednostavno slijedi konačan izraz matrice rotacije izražene preko Eulerovih parametara:

$$\mathbf{R} = 2\begin{bmatrix} e_0^2 + e_1^2 - \frac{1}{2} & e_1e_2 - e_0e_3 & e_1e_3 + e_0e_2 \\ e_1e_2 + e_0e_3 & e_0^2 + e_2^2 - \frac{1}{2} & e_2e_3 - e_0e_1 \\ e_1e_3 - e_0eq_2 & e_2e_3 + e_0e_1 & e_0^2 + e_3^2 - \frac{1}{2} \end{bmatrix}. \tag{52}$$

Sada možemo izvesti eksplicitne formule za Eulerove parametre s obzirom na elemente matrice transformacije, kao u [7]. Pretpostavimo da je devet kosinusa smjera dano transformacijskom matricom prema jednadžbi:

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{12} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}. \tag{53}$$

Trag matrice $\mathbf{R}$, što je zbroj elemenata dijagonale, definiramo:

$$tr\mathbf{R} = r_{11} + r_{22} + r_{33}. \tag{54}$$

Ako napišemo trag matrice (52), dobivamo:

$$tr\mathbf{R} = 2(3e_0 r + e_1^2 + e_2^2 + e_3^2) - 3 = 2(2e_0^2 + 1) - 3 = 4e_0^2 - 1. \tag{55}$$

Koristeći izraz

$$e_0^2 + \mathbf{e}^T\mathbf{e} = \mathbf{p}^T\mathbf{p} = 1, \tag{56}$$

jednadžba (55) prelazi u

$$e_0^2 = \frac{tr\mathbf{R} + 1}{4}. \tag{57}$$

Ako taj izraz uvrstimo u elemente na dijagonali rotacijske matrice (52), dobivamo:

$$r_{11} = 2(e_0^2 + e_1^2) - 1 = 2\left(\frac{tr\mathbf{R} + 1}{4} + e_1^2\right) - 1. \tag{58}$$

Na taj način možemo dobiti izraze za sva četiri Eulerova parametra:

$$\begin{aligned} e_0^2 &= \frac{tr\mathbf{R} + 1}{4} \\ e_1^2 &= \frac{1 + 2r_{11} - tr\mathbf{R}}{4} \\ e_2^2 &= \frac{1 + 2r_{22} - tr\mathbf{R}}{4} \\ e_3^2 &= \frac{1 + 2r_{33} - tr\mathbf{R}}{4}. \end{aligned} \tag{59}$$

Za ovakvu parametrizaciju, ne postoje kritični slučajevi u kojima su inverzne formule gornjih jednadžbi singularne. Možemo uvesti još neke relacije koje uključuju kososimetričnu matricu $\mathbf{e}$. Vrijede sljedeći izrazi:

$$\tilde{\mathbf{e}}\mathbf{e} = \mathbf{0}, \tag{60}$$

$$\tilde{\mathbf{e}}\tilde{\mathbf{e}} = \mathbf{e}\mathbf{e}^T - (\mathbf{e}^T\mathbf{e})\mathbf{I} = \mathbf{e}\mathbf{e}^T - (1 - e_0^2)\mathbf{I}. \tag{61}$$

Iz raspisa izraza (60) slijedi:

$$\begin{bmatrix} 0 & -e_3 & e_2 \\ e_3 & 0 & -e_1 \\ -e_2 & e_1 & 0 \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix} = \begin{bmatrix} -e_2e_3 + e_2e_3 \\ e_1e_3 - e_1e_3 \\ -e_1e_2 + e_1e_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}. \tag{62}$$

Da bi dokazali izraz (61) raspisat ćemo redom izraze $\tilde{\mathbf{e}}\tilde{\mathbf{e}}$, $\mathbf{e}\mathbf{e}^T - (\mathbf{e}^T\mathbf{e})\mathbf{I}$ te $\mathbf{e}\mathbf{e}^T - (1 - e_0^2)\mathbf{I}$.

$$\begin{aligned} \tilde{\mathbf{e}}\tilde{\mathbf{e}} &= \begin{bmatrix} 0 & -e_3 & e_2 \\ e_3 & 0 & -e_1 \\ -e_2 & e_1 & 0 \end{bmatrix} \begin{bmatrix} 0 & -e_3 & e_2 \\ e_3 & 0 & -e_1 \\ -e_2 & e_1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} -e_3^2 - e_2^2 & e_1e_2 & e_1e_3 \\ e_1e_2 & -e_3e^2 - e_1^2 & e_2e_3 \\ e_1e_3 & e_2e_3 & -e_2^2 - e_1^2 \end{bmatrix}. \end{aligned} \tag{63}$$

Iz raspisa $\mathbf{e}\mathbf{e}^T - (\mathbf{e}^T\mathbf{e})\mathbf{I}$ slijedi:

$$\begin{aligned} \mathbf{e}\mathbf{e}^T - (\mathbf{e}^T\mathbf{e})\mathbf{I} &= \left(\begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix} \begin{bmatrix} e_1 & e_2 & e_3 \end{bmatrix} - \begin{bmatrix} e_1 & e_2 & e_3 \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix} \right) \mathbf{I} = \\ &\begin{bmatrix} e_1^2 & e_1e_2 & e_1e_3 \\ e_1e_2 & e_2^2 & e_2e_3 \\ e_1e_3 & e_2e_3 & e_3^2 \end{bmatrix} - \begin{bmatrix} 1 - e_0^2 & 0 & 0 \\ 0 & 1 - e_0^2 & 0 \\ 0 & 0 & 1 - e_0^2 \end{bmatrix} = \\ &\begin{bmatrix} e_1^2 - 1 + e_0^2 & e_1e_2 & e_1e_3 \\ e_1e_2 & e_2^2 - 1 + e_0^2 & e_2e_3 \\ e_1e_3 & e_2e_3 & e_3^2 - 1 + e_0^2 \end{bmatrix}. \end{aligned} \tag{64}$$

Ako iskoristimo poznatu relaciju (56) i uvrstimo ju u izraz (64) on postaje:

$$\begin{bmatrix} -e_2^2 - e_3^2 & e_1e_2 & \\ e_1e_2 & -e_1^2 - e_3^2 & e_2e_3 \\ e_1e_3 & e_2e_3 & -e_1^2 - e_2^2 \end{bmatrix}. \tag{65}$$

čime smo dokazali izraz (61). U nastavku ćemo izvesti i dokazati još neke relacije koje povezuju vremenske derivacije i matrice rotacije. Na početku ćemo definirati dvije matrice $\mathbf{E}$ i $\mathbf{G}$ koje ćemo često koristiti:

$$\mathbf{E} = \begin{bmatrix} -e_1 & e_0 & -e_3 & e_2 \\ -e_2 & e_3 & e_0 & -e_1 \\ -e_3 & -e_2 & e_1 & e_0 \end{bmatrix} = \begin{bmatrix} -\mathbf{e} & \tilde{\mathbf{e}} + e_0\mathbf{I} \end{bmatrix}, \tag{66}$$

$$\mathbf{G} = \begin{bmatrix} -e_1 & e_0 & e_3 & -e_2 \\ -e_2 & e_3 & e_0 & e_1 \\ -e_3 & -e_2 & -e_1 & e_0 \end{bmatrix} = \begin{bmatrix} -\mathbf{e} & -\tilde{\mathbf{e}} + e_0\mathbf{I} \end{bmatrix}. \tag{67}$$

Lako je dokazati da je svaki red matrica $\mathbf{E}$ i $\mathbf{G}$ ortogonalan na vektor $\mathbf{p}$ koji čine Eulerovi parametri:

$$\mathbf{Ep} = \mathbf{Gp} = \mathbf{0}. \tag{68}$$

Množenjem $\mathbf{E}$ i $\mathbf{p}$ slijedi:

$$\mathbf{Ep} = \begin{bmatrix} -e_1 & e_0 & -e_3 & e_2 \\ -e_2 & e_3 & e_0 & -e_1 \\ -e_3 & -e_2 & e_1 & e_0 \end{bmatrix} \begin{bmatrix} e_0 \\ e_1 \\ e_2 \\ e_3 \end{bmatrix} = \begin{bmatrix} -e_0e_1 + e_0e_1 - e_2e_3 + e_2e_3 \\ -e_2e_0 + e_1e_3 + e_0e_2 - e_1e_3 \\ -e_0e_3 - e_2e_1 + e_1e_2 + e_0e_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{69}$$

Analogno vrijedi za $\mathbf{Gp}$:

$$\mathbf{Gp} = \begin{bmatrix} -e_1 & e_0 & e_3 & -e_2 \\ -e_2 & -e_3 & e_0 & e_1 \\ -e_3 & e_2 & -e_1 & e_0 \end{bmatrix} \begin{bmatrix} e_0 \\ e_1 \\ e_2 \\ e_3 \end{bmatrix} = \begin{bmatrix} -e_0e_1 + e_0e_1 + e_2e_3 - e_2e_3 \\ -e_2e_0 - e_1e_3 + e_0e_2 + e_1e_3 \\ -e_0e_3 + e_2e_1 - e_1e_2 + e_0e_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{70}$$

Iz svojstva ortogonalnosti redova matrica $\mathbf{E}$ i $\mathbf{G}$ slijedi da:

$$\mathbf{EE}^T = \mathbf{GG}^T = \mathbf{I}. \tag{71}$$

Prema izrazu $\mathbf{EE}^T$ vrijedi:

$$\mathbf{EE}^T = \begin{bmatrix} -e_1 & e_0 & e_3 & -e_2 \\ -e_2 & -e_3 & e_0 & e_1 \\ -e_3 & e_2 & -e_1 & e_0 \end{bmatrix} \begin{bmatrix} -e_1 & -e_2 & -e_3 \\ e_0 & e_3 & -e_2 \\ -e_3 & e_0 & e_1 \\ e_2 & -e_1 & e_0 \end{bmatrix} =$$

$$\begin{bmatrix} e_1^2 + e_0^2 + e_3^2 + e_2^2 & e_1e_2 - e_0e_3 + e_0e_3 - e_1e_2 & e_1e_3 + e_0e_2 - e_1e_3 - e_0e_2 \\ e_1e_2 - e_0e_3 + e_0e_3 - e_1e_2 & e_2^2 + e_3^2 + e_0^2 + e_1^2 & e_2e_3 - e_2e_3 - e_0e_1 + e_0e_1 \\ e_1e_3 + e_0e_2 - e_1e_3 - e_0e_2 & e_2e_3 - e_2e_3 - e_1e_0 + e_1e_0 & e_3^2 + e_2^2 + e_1^2 + e_0^2 \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{72}$$

Raspisivanjem izraza $\mathbf{G}\mathbf{G}^T$ dobijemo:

$$\mathbf{G}\mathbf{G}^T = \begin{bmatrix} -e_1 & e_0 & e_3 & -e_2 \\ -e_2 & -e_3 & e_0 & e_1 \\ -e_3 & e_2 & -e_1 & e_0 \end{bmatrix} \begin{bmatrix} -e_1 & -e_2 & -e_3 \\ e_0 & -e_3 & e_2 \\ e_3 & e_0 & -e_1 \\ -e_2 & e_1 & e_0 \end{bmatrix} =$$

$$\begin{bmatrix} e_1^2+e_0^2+e_3^2+e_2^2 & e_1e_2-e_0e_3+e_0e_3-e_1e_2 & e_1e_3+e_0e_2-e_1e_3-e_0e_2 \\ e_1e_2-e_0e_3+e_0e_3-e_1e_2 & e_2^2+e_3^2+e_0^2+e_1^2 & e_2e_3-e_2e_3-e_0e_1+e_0e_1 \\ e_1e_3+e_0e_2-e_1e_3-e_0e_2 & e_2e_3-e_2e_3-e_1e_0+e_1e_0 & e_3^2+e_2^2+e_1^2+e_0^2 \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{73}$$

Čime smo dokazali izraz (71). Za $\mathbf{E}^T\mathbf{E}$ i $\mathbf{G}^T\mathbf{G}$ vrijedi:

$$\mathbf{E}^T\mathbf{E} = \mathbf{G}^T\mathbf{G} = -\mathbf{p}\mathbf{p}^T + \mathbf{I}^{4\times4}, \tag{74}$$

pri čemu je $\mathbf{I}^{4\times4}$ jedinična matrica $4x4$. Iz množenja $\mathbf{E}^T\mathbf{E}$ slijedi:

$$\mathbf{E}^T\mathbf{E} = \begin{bmatrix} -e_1 & -e_2 & -e_3 \\ e_0 & e_3 & -e_2 \\ -e_3 & e_0 & e_1 \\ e_2 & -e_1 & e_0 \end{bmatrix} \begin{bmatrix} -e_1 & e_0 & e_3 & -e_2 \\ -e_2 & -e_3 & e_0 & e_1 \\ -e_3 & e_2 & -e_1 & e_0 \end{bmatrix} =$$

$$\begin{bmatrix} e_1^2+e_2^2+e_3^2 & -e_1e_0-e_2e_3+e_2e_3 & e_1e_3-e_0e_2-e_1e_3 & -e_1e_2+e_1e_2-e_3e_0 \\ -e_1e_0-e_2e_3+e_2e_3 & e_0^2+e_3^2+e_2^2 & -e_0e_3+e_0e_3-e_2e_1 & e_0e_2-e_1e_3-e_2e_0 \\ e_1e_3-e_0e_2-e_1e_3 & -e_0e_3+e_0e_3-e_1e_2 & e_3^2+e_0^2+e_1^2 & -e_2e_3-e_1e_0+e_1e_0 \\ -e_1e_2+e_1e_2-e_0e_3 & e_0e_2-e_1e_3-e_0e_2 & -e_2e_3-e_0e_1+e_0e_1 & e_2^2+e_1^2+e_0^2 \end{bmatrix} =$$

$$\begin{bmatrix} e_1^2+e_2^2+e_3^2 & -e_0e_1 & -e_0e_2 & -e_0e_3 \\ -e_1e_0 & e_0^2+e_3^2+e_2^2 & -e_1e_2 & -e_1e_3 \\ -e_0e_2 & -e_1e_2 & e_3^2+e_0^2+e_1^2 & -e_2e_3 \\ -e_0e_3 & -e_1e_3 & -e_3e_3 & e_2^2+e_1^2+e_0^2 \end{bmatrix}. \tag{75}$$

koji, nakon što uvrstimo (56), prelazi u:

$$\mathbf{E}^T\mathbf{E} = \begin{bmatrix} 1-e_0^2 & -e_0e_1 & -e_0e_2 & -e_0e_3 \\ -e_1e_0 & 1-e_1^2 & -e_1e_2 & -e_1e_3 \\ -e_0e_2 & -e_1e_2 & 1-e_2^2 & -e_2e_3 \\ -e_0e_3 & -e_1e_3 & -e_3e_3 & 1-e_4^2 \end{bmatrix} \tag{76}$$

Analogno, ako izraz $\mathbf{G}^T\mathbf{G}$ raspišemo:

$$\mathbf{G}^T\mathbf{G} = \begin{bmatrix} -e_1 & -e_2 & -e_3 \\ e_0 & -e_3 & e_2 \\ e_3 & e_0 & -e_1 \\ -e_2 & e_1 & e_0 \end{bmatrix} \begin{bmatrix} -e_1 & e_0 & e_3 & -e_2 \\ -e_2 & -e_3 & e_0 & e_1 \\ -e_3 & e_2 & -e_1 & e_0 \end{bmatrix} =$$

$$\begin{bmatrix} e_1^2+e_2^2+e_3^2 & -e_0e_1+e_2e_3-e_2e_3 & -e_1e_3-e_0e_2+e_1e_3 & e_1e_2-e_1e_2-e_0e_3 \\ -e_0e_1+e_2e_3-e_2e_3 & e_0^2+e_3^2+e_2^2 & e_0e_3-e_0e_3-e_1e_2 & -e_0e_2-e_1e_3+e_0e_2 \\ -e_1e_3-e_0e_2+e_1e_3 & e_0e_3-e_0e_3-e_1e_2 & e_3^2+e_0^2+e_1^2 & -e_2e_3+e_0e_1+e_0e_1 \\ e_1e_2-e_1e_2-e_0e_3 & -e_0e_2-e_1e_3+e_0e_2 & -e_2e_3+e_0e_1-e_0e_1 & e_2^2+e_1^2+e_0^2 \end{bmatrix} =$$

$$\begin{bmatrix} 1-e_0^2 & -e_0e_1 & -e_0e_2 & -e_0e_3 \\ -e_0e_1 & 1-e_1^2 & -e_1e_2 & -e_1e_3 \\ -e_0e_2 & -e_1e_2 & 1-e_2^2 & -e_2e_3 \\ -e_0e_3 & -e_1e_3 & -e_3e_3 & 1-e_4^2 \end{bmatrix}. \tag{77}$$

Opet, uvrštavanjem relacije (56), dobivamo:

$$\mathbf{G}^T\mathbf{G} = \begin{bmatrix} 1-e_0^2 & -e_0e_1 & -e_0e_2 & -e_0e_3 \\ -e_1e_0 & 1-e_1^2 & -e_1e_2 & -e_1e_3 \\ -e_0e_2 & -e_1e_2 & 1-e_2^2 & -e_2e_3 \\ -e_0e_3 & -e_1e_3 & -e_3e_3 & 1-e_4^2 \end{bmatrix}. \tag{78}$$

Ako raspišemo izraz $-\mathbf{p}\mathbf{p}^T + \mathbf{I}^{4\times4}$:

$$-\mathbf{p}\mathbf{p}^T + \mathbf{I}^{4\times4} = -\begin{bmatrix} e_0 \\ e_1 \\ e_2 \\ e_3 \end{bmatrix} \begin{bmatrix} e_0 & e_1 & e_2 & e_3 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} =$$

$$-\begin{bmatrix} 1-e_0^2 & -e_0e_1 & -e_0e_2 & -e_0e_3 \\ -e_0e_1 & 1-e_1^2 & -e_1e_2 & -e_1e_3 \\ -e_0e_2 & -e_1e_2 & 1-e_2^2 & -e_2e_3 \\ -e_0e_3 & -e_1e_3 & -e_2e_3 & 1-e_e^2 \end{bmatrix}. \tag{79}$$

Uspoređujući jednadžbe (76), (78) i (79) dokazali smo relaciju (74).
Sada ćemo dokazati:

$$\mathbf{E}\mathbf{G}^T = (2e_0^2 - 1)\mathbf{I} + 2(\mathbf{e}\mathbf{e}^T + e_0\tilde{\mathbf{e}}). \tag{80}$$

Množenjem $\mathbf{E}\mathbf{G}^T$ dobijemo:

$$\mathbf{E}\mathbf{G}^T = \begin{bmatrix} -e_1 & e_0 & e_3 & -e_2 \\ -e_2 & -e_3 & e_0 & e_1 \\ -e_3 & e_2 & -e_1 & e_0 \end{bmatrix} \begin{bmatrix} -e_1 & -e_2 & -e_3 \\ e_0 & -e_3 & e_2 \\ e_3 & e_0 & -e_1 \\ -e_2 & e_1 & e_0 \end{bmatrix} =$$

$$\begin{bmatrix} e_1^2 + e_0^2 - e_3^2 - e_2^2 & e_1e_2 - e_0e_3 - e_0e_3 + e_1e_2 & e_1e_3 + e_0e_2 + e_1e_3 \\ e_1e_2 + e_0e_3 + e_0e_3 + e_1e_2 & e_2^2 - e_3^2 + e_0^2 - e_1^2 & e_2e_3 + e_2e_3 - e_0e_1e_0e_1 \\ e_1e_3 + e_1e_3 - e_0e_2 & e_2e_3 + e_2e_3 + e_1e_0 + e_1e_0 & e_3^2 - e_2^2 - e_1^2 + e_0^2 \end{bmatrix}. \tag{81}$$

Izlučivanjem broja 2 ispred zagrade te primjenjujući dobro poznatu relaciju (56) dobije se:

$$\mathbf{E}\mathbf{G}^T = 2 \begin{bmatrix} e_0^2 + e_1^2 - \frac{1}{2} & e_1e_2 - e_0e_3 & e_1e_3 + e_0e_2 \\ e_1e_2 + e_0e_3 & e_0^2 + e_2^2 - \frac{1}{2} & e_2e_3 - e_0e_1 \\ e_1e_3 - e_0e_2 & e_2e_3 + e_0e_1 & e_0^2 + e_3^2 - \frac{1}{2} \end{bmatrix}, \tag{82}$$

što je izraz ekvivalentan (52):

$$\mathbf{R} = \mathbf{E}\mathbf{G}^T. \tag{83}$$

Gornju relaciju možemo shvatiti kao rezultat dvije uzastopne rotacije. Ako izraz (56) deriviramo, dobijemo:

$$\mathbf{p}^T\dot{\mathbf{p}} = \dot{\mathbf{p}}^T\mathbf{p} = 0. \tag{84}$$

Deriviranje izraza (68) rezultira:

$$\mathbf{E}\dot{\mathbf{p}} = -\dot{\mathbf{E}}\mathbf{p}, \tag{85}$$

$$\mathbf{G}\dot{\mathbf{p}} = -\dot{\mathbf{G}}\mathbf{p}. \tag{86}$$

Vrijedi također:

$$\dot{\mathbf{E}}\dot{\mathbf{p}} = \dot{\mathbf{G}}\dot{\mathbf{p}} = \mathbf{0}. \tag{87}$$

Sada ćemo pokazati da vrijedi:

$$\mathbf{E}\dot{\mathbf{G}}^T = \dot{\mathbf{E}}\mathbf{G}^T \tag{88}$$

. Umnožak $\mathbf{E}\dot{\mathbf{G}}^T$ daje:

$$\mathbf{E}\dot{\mathbf{G}}^T = \begin{bmatrix} -e_1 & e_0 & -e_3 & e_2 \\ -e_2 & e_3 & e_0 & -e_1 \\ -e_3 & -e_2 & e_1 & e_0 \end{bmatrix} \begin{bmatrix} -\dot{e_1} & -\dot{e_2} & -\dot{e_3} \\ \dot{e_0} & -\dot{e_3} & \dot{e_2} \\ \dot{e_3} & \dot{e_0} & -\dot{e_1} \\ -\dot{e_2} & \dot{e_1} & \dot{e_0} \end{bmatrix} =$$

$$= \begin{bmatrix} e_1\dot{e_1} + e_0\dot{e_0} - e_3\dot{e_3} - e_2\dot{e_2} & e_1\dot{e_2} - e_0\dot{e_3} - e_3\dot{e_0} + e_2\dot{e_1} & e_1\dot{e_3} + e_0\dot{e_2} + e_3\dot{e_1} + e_2\dot{e_0} \\ e_2\dot{e_1} + e_3\dot{e_0} + e_0\dot{e_3} + e_1\dot{e_2} & e_2\dot{e_2} - e_3\dot{e_3} + e_0\dot{e_0} - e_1\dot{e_1} & e_2\dot{e_3} + e_3\dot{e_2} + e_0\dot{e_1} - e_1\dot{e_0} \\ e_3\dot{e_1} - e_2\dot{e_0} + e_1\dot{e_3} - e_0\dot{e_2} & e_3\dot{e_2} + e_2\dot{e_3} + e_1\dot{e_0} + e_0\dot{e_1} & e_3\dot{e_3} - e_2\dot{e_2} - e_1\dot{e_1} + e_0\dot{e_0} \end{bmatrix}, \tag{89}$$

a umnožak $\dot{\mathbf{E}}\mathbf{G}^T$ daje:

$$\dot{\mathbf{E}}\mathbf{G}^T = \begin{bmatrix} -\dot{e}_1 & \dot{e}_0 & -\dot{e}_3 & \dot{e}_2 \\ \dot{e}_2 & \dot{e}_3 & \dot{e}_0 & -\dot{e}_1 \\ -\dot{e}_3 & -\dot{e}_2 & \dot{e}_1 & \dot{e}_0 \end{bmatrix} \begin{bmatrix} -e_1 & -e_2 & -e_3 \\ e_0 & -e_3 & e_2 \\ e_3 & e_0 & -e_1 \\ -e_2 & e_1 & e_0 \end{bmatrix} =$$

$$= \begin{bmatrix} e_1\dot{e}_1 + e_0\dot{e}_0 - e_3\dot{e}_3 - e_2\dot{e}_2 & \dot{e}_1e_2 - \dot{e}_0e_3 - e_0\dot{e}_3 + e_1\dot{e}_2 & \dot{e}_1e_3 + \dot{e}_0e_2 + \dot{e}_3e_1 + \dot{e}_2e_0 \\ e_1\dot{e}_2 + e_0\dot{e}_3 + e_3\dot{e}_0 + e_2\dot{e}_1 & e_2\dot{e}_2 - e_3\dot{e}_3 + e_0\dot{e}_0 - e_1\dot{e}_1 & \dot{e}_2e_3 + \dot{e}_3e_2 + e_1\dot{e}_0 - e_0\dot{e}_1 \\ e_1\dot{e}_3 - e_0\dot{e}_2 + e_1\dot{e}_3 - e_2\dot{e}_0 & e_2\dot{e}_3 + \dot{e}_2e_3 + e_0\dot{e}_1 + \dot{e}_0e_1 & e_3\dot{e}_3 - e_2\dot{e}_2 - e_1\dot{e}_1 + e_0\dot{e}_0 \end{bmatrix}. \tag{90}$$

Usporedbom jednadžbi (89) te (90), dokazali smo relaciju (88).

1.4 Jednadžba kinematičke rekonstrukcije

Matricu transformacije iz lokalnog u globalni koordinatni sustav označimo $\mathbf{R}$, tada vrijedi:

$$\mathbf{v} = \mathbf{R}\mathbf{v}'. \tag{91}$$

U gornjoj jednadžbi, $\mathbf{v}$ i $\mathbf{v}'$ predstavljaju globalne i lokalne komponente vektora $\vec{v}$ fiksiran na tijelo, u lokalnom koordinatnom sustavu $\xi - \eta - \zeta$.Vektorski umnožak vektora $\vec{v}$ s proizvoljnim vektorom $\vec{a}$ koji rezultira vektorom $\vec{b}$ onda to možemo napisati

$$\mathbf{b} = \tilde{\mathbf{v}}\mathbf{a}, \tag{92}$$

te

$$\mathbf{b}' = \tilde{\mathbf{v}}'\mathbf{a}'. \tag{93}$$

Kako vrijedi da je $\mathbf{b} = \mathbf{R}\mathbf{b}'$ i $\mathbf{a} = \mathbf{R}\mathbf{a}'$,jednadžba (92) postaje:

$$\mathbf{R}\mathbf{b}' = \tilde{\mathbf{v}}\mathbf{R}\mathbf{a}'. \tag{94}$$

Kada jednadžbu (93) uvrstimo u jednadžbu (94) slijedi:

$$\mathbf{R}\tilde{\mathbf{v}}'\mathbf{a}' = \tilde{\mathbf{v}}\mathbf{R}\mathbf{a}', \tag{95}$$

koji nakon množenja s lijeva $\mathbf{R}^T$ prelazi u:

$$\tilde{\mathbf{v}}' = \mathbf{R}^T\tilde{\mathbf{v}}\mathbf{R}, \tag{96}$$

koji dalje možemo preurediti u:

$$\tilde{\mathbf{v}} = \mathbf{R}\tilde{\mathbf{v}}'\mathbf{R}^T. \tag{97}$$

Pretpostavljeno je da koordinatni sustav $\xi - \eta - \zeta$ fiksiran na tijelo rotira brzinom $\vec{\omega}(t)$ rotira s obzirom na globalni koordinatni sustav $x - y - z$. Komponente kutne brzine $\vec{\omega}$ u globalnom koordinatnom sustavu $x - y - z$ i lokalnom koordinatnom sustavu $\xi - \eta - \zeta$ su $\omega = \begin{bmatrix} \omega_x & \omega_y & \omega_z \end{bmatrix}^T$ i $\begin{bmatrix} \omega' = \begin{bmatrix} \omega_\xi & \omega_\eta & \omega_{zeta} \end{bmatrix} \end{bmatrix}$ Derivacija po vremenu iznosi

$$\dot{\mathbf{v}} = \dot{\mathbf{R}}\mathbf{v}' + \mathbf{R}\dot{\mathbf{v}}'. \tag{98}$$

Znamo da je vektor $\vec{v}$ fikisran za tijelo, dakle u lokalnom koordinatnom sustavu $\xi - \eta - \zeta$ slijedi da je $\dot{\mathbf{v}} = \mathbf{0}$ iz čega slijedi:

$$\dot{\mathbf{v}} = \dot{\mathbf{R}}\mathbf{v}'. \tag{99}$$

Vektor $\dot{\vec{v}}$ možemo prikazati preko kutne brzine $\vec{\omega}$ kao $\dot{\vec{v}} = \vec{\omega} \times \vec{v}$, tj:

$$\dot{\mathbf{v}} = \tilde{\omega}\mathbf{v}. \tag{100}$$

Koristeći (91) gornja jednadžba postaje:

$$\dot{\mathbf{v}} = \tilde{\omega}\mathbf{R}\mathbf{v}'. \tag{101}$$

Iz jednadžbi (99) i (101) slijedi relacija:

$$\dot{\mathbf{R}} = \tilde{\omega}\mathbf{R}. \tag{102}$$

Ako uvrstimo izraz (96) u gornju jednadžbu dobijemo:

$$\dot{\mathbf{R}} = \mathbf{R}\tilde{\omega}'. \tag{103}$$

Množeći gornji izraz s lijeva matricom $\mathbf{R}^T$ dobijemo konačan izraz:

$$\tilde{\omega}' = \mathbf{R}^T\dot{\mathbf{R}}. \tag{104}$$

2 GEOMETRIJSKA INTEGRACIJA

Integracijske metode koje 'razumiju' fizikalnu pozadinu, odnosno algoritamski su prilagođene dinamičko-geometrijskoj strukturi dinamičkog sustava [1], predstavljale su određenu promjenu paradigme u području numeričkih metoda. Stoga ćemo u ovom poglavlju predstaviti osnovne pojmove vezane za Lieve algebre i Lieve grupe i dati obrise novo predložene metode integracije. Naglasak poglavlja je na detaljima implementacije metode u kodu.

2.1 Uvod u geometrijsku integraciju

Matematičko modeliranje i numeričke simulacije velikih trodimenzionalnih gibanja te računalni tretman kinematičkih ograničenja središnji su problemi računalne dinamike više tijela. Za razliku od problema koji pripadaju klasičnoj dinamici u kojoj se radi malih pomaka, problemi svode na linearne, modeli dinamike više tijela moraju se nositi s kinematskim ograničenjima, velikim rotacijama te geometrijskim nelinearnostima. Znamo da velike trodimenzionalne rotacije nisu vektorske veličine što znači da njihov prikaz u računskim procedurama nije trivijalan. Zapravo je matematička reprezentacija 3D rotacija te modeliranje njezine dinamike vezanu za određenu reprezentaciju točka u kojima se jedna računalna procedura i integracijska shema razlikuje od druge. U prvom smo poglavlju vidjeli da mogućnost parametrizacija trodimenzionalne rotacije dolazi iz same matematičke strukture rotacije, a to je da se ono može opisati matricom rotacije $\mathbf{R}$ koja pripada grupi $SO(3)$. Specijalna ortogonalna grupa $SO(3)$, koja je ujedno i mnogostrukost, što je zapravo nelinearni prostor koji se lokalno može parametrizirati koordinatama koje tvore kartu. Na temelju toga, Eulerovi kutevi, jedna od mogućih parametrizacija, formiraju kartu na $SO(3)$ mnogostrukosti, pružajući na taj način mogućnost prebacivanja izračuna s originalne nelinearne rotacijske mnogostrukosti u linearni parametarski prostor u kojem se mogu koristiti standardne metode koje operiraju na linearnim vektorskim prostorima [8].

2.2 Lieve grupa i Lieva algebra

Grupe matrica rotacija i kvaterniona kojima smo opisivali transformacije tijela kontinuirane su grupe. Kontinuirane grupe su one grupe u kojima postoji beskonačno mnogo elemenata koji transformiraju tijelo, pri čemu spomenuta transforma-

cija može biti parametrizirana parametrom koji može zauzeti bilo koju vrijednost. Primjerice, ako za matricu rotacije kao parametar uzmemo kut rotacije, znamo da on može uzeti bilo koju vrijednost. Za razliku od toga, za zrcaljenje, što je diskretna transformacija, ne postoji kontinuirana parametrizacija. Bitno je naglasiti da za kontinuirane simetrije, postoje elementi koji su proizvoljno blizu identitetu transformacije, što se ne može reći za diskontinuirane transformacije. Matematički, ako s $\mathbf{I}$ označimo identitet, element g grupe koji je blizu identiteta možemo zapisati kao:

$$\mathbf{g}(\epsilon) = \mathbf{I} + \epsilon\mathbf{X} \tag{105}$$

pri čemu je ϵ proizvoljno mali broj a $\mathbf{X}$ nazivamo generator rotacije. Možemo reći da su takve transformacije infinitezimalno male, dakle kao da se objekt uopće ne transformira. Međutim, ponavljajući takve infinitezimalne transformacije dovoljan broj puta , možemo ostvariti konačnu rotaciju. Gornju tvrdnju možemo matematički zapisati na sljedeći način:

$$\mathbf{h}(\theta) = (\mathbf{I} + \epsilon\mathbf{X})(\mathbf{I} + \epsilon\mathbf{X})(\mathbf{I} + \epsilon\mathbf{X})... = (\mathbf{I} + \epsilon\mathbf{X})^k, \tag{106}$$

gdje k označava broj ponavljanja transformacije. Uzimajući da nam θ predstavlja neku konačnu transformaciju, a N je veliki broj koji predstavlja broj infinitezimalnih transformacija, onda element grupe blizu identiteta možemo napisati kao:

$$\mathbf{g}(\theta) = \mathbf{I} + \frac{\theta}{N}\mathbf{X}. \tag{107}$$

Da bi transformacije koje predstavljamo, bile što manje moguće, zahtijevamo da N bude što veći broj, što možemo zapisati kao $N \to \infty$. Da bi dobili konačnu transformaciju, infinitezimalnu transformaciju je potrebno ponoviti beskonačno mnogo puta što se može zapisati:

$$\mathbf{h}(\theta) = \lim_{N\to\infty} \left(\mathbf{I} + \frac{\theta}{N}\mathbf{X}\right)^N, \tag{108}$$

a što je zapravo poznati izraz napisan kao limes, koji se može izraziti u obliku eksponencijalne funkcije:

$$\mathbf{h}(\theta) = \lim_{N\to\infty} \left(\mathbf{I} + \frac{\theta}{N}\mathbf{X}\right)^N = e^{\theta\mathbf{X}}. \tag{109}$$

Za matrične Lieve grupe definiramo odgovarajuću Lievu algebru kao skup objekata koji daju element grupe kada ih kombiniramo u argumentu eksponencijalne funkcije. Za Lievu grupu G, Lievu algebra $\mathfrak{g}$ čini skup matrica $\mathbf{X}$ takav da vrijedi $e^{t\mathbf{X}} \in G$ za $t \in R$ zajedno s operacijom koju nazivamo Lieva zagrada koja definira pravilo kombiniranje spomenutih matrica. Znamo da grupa nije samo skup transformacija nego uključuje i binarnu operaciju '$\circ$ koja sadrži uputu kako kombinirati dva elementa grupe. Za Lieve grupe kao što je npr $SO(3)$ ta je operacija najobičnije matrično

množenje. To, međutim, ne vrijedi za Lievu algebru. Elementi Lieve grupe su matrice, međutim, množenje dvije matrice ne mora nužno dati element te iste Lieve algebre. Eksplicitno, Lieva algebra nije skup zatvoren pod matričnim množenjem. Postoji, međutim, drugo pravilo za kombiniranje elemenata Lieve algebre. Veza između pravila za kombiniranje elemenata Lieve grupe i Lieve algebre povezana je sljedećom relacijom:

$$g \circ h = e^{\mathbf{X}} \circ e^{\mathbf{Y}} = e^{\mathbf{X}+\mathbf{Y}+\frac{1}{2}[\mathbf{X},\mathbf{Y}]+\frac{1}{12}[\mathbf{X},[\mathbf{X},\mathbf{Y}]]-\frac{1}{12}[\mathbf{Y},[\mathbf{X},\mathbf{Y}]..} \tag{110}$$

pri čemu su $\mathbf{X}$, $\mathbf{Y} \in \mathfrak{g}$. S desne strane gornje relacije imamo jedan element grupe i umnožak elementa grupe je preveden u zbroj elemenata Lieve algebre. Novi simbol $[\quad,\quad]$ nazivamo Lieva zagrada i za matrične grupe je definirana kao:

$$[\mathbf{X}, \mathbf{Y}] = \mathbf{XY} - \mathbf{YX}, \tag{111}$$

i naziva se komutator. $\mathbf{XY}$ i $\mathbf{YX}$ nisu nužno elementi Lieve algebre, međutim, njihova razlika uvijek je element Lieve grupe. Dakle, Lieva algebra je zatvorena pod Lievom zagradom, kao što je i grupa transformacija zatvorena pod matričnim množenjem. Formalno, Lievu algebru možemo definirati kao:

$$so(3) = \{\mathbf{S} \in \mathbb{R}^{3\times 3} \quad | \quad \mathbf{S}^T = -\mathbf{S}\}. \tag{112}$$

Lieva algebra je vektorski prostor $\mathfrak{g}$ opremljen binarnom operacijom $[,] : \mathfrak{g} \times \mathfrak{g} \to \mathfrak{g}$. Binarna operacija zadovoljava sljedeće aksiome:

1. *Bilinearnost*: $[aX+bY, Z] = a[X, Z]+b[Y, Z]$ i $[Z, aX+bY] = a[Z, X]+b[Z, Y]$ za bilo koje brojeve a, b $\forall X, Y, Z \in \mathfrak{g}$,
2. *Antikomutativnost*: $[X, Y] = -[Y, X] \quad \forall X, Y \in \mathfrak{g}$,
3. *Jakobijev identitet*: $[X, [Y, Z]] + [Z, [X, Y]] + [Y, [Z, X]] = 0 \quad \forall X, Y \in \mathfrak{g}$.

U nastavku ćemo vidjeti da grupe $SU(2)$ i $SO(3)$ imaju istu Lievu algebru. Prethodno smo već pokazali jedan-na-jedan mapiranje između $SU(2)$ i jediničnih kvaterniona. Ako kvaternion definiramo u obliku $q = a\mathbf{1} + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$, tada ga možemo smatrati jediničnim ako vrijedi:

$$a^2 + b^2 + c^2 + d^2 = 1, \tag{113}$$

što je zapravo isto kao i uvjet kojim smo definirali sferu jediničnih kvaterniona. Na temelju toga, možemo zaključiti da nam kvaternioni služe kao preslikavanje između $SU(2)$ i grupe jediničnih kvaterniona $Sp(1)$. Ovo preslikavanje je izomorfno (jedan-na-jedan) što zapravo znači da su $SU(2)$ i grupa jediničnih kvaterniona $Sp(1)$ zapravo iste strukture. Važno je naglasiti da postoji izomorfizam između Lieve algebre grupe $SU(2)$ te Lieve algebre grupe $SO(3)$.

2.3 Nova metoda integracije

Znamo da kvaternioni, rotaciju, koja je po naravi trodimenzionalni fenomen, opisuju s četiri parametra, što daje dodatnu jednadžbu. To implicira da kvaternion mora imati jediničnu normu što dodaje još jednu jednadžbu. Postojanje tog algebarskog uvjeta zahtjeva u numeričkoj integraciji uporabu stabilizacijskih metoda. Novopredložena metoda rješava taj problem tako da, iako radi izravno s četiri komponente kvaterniona, bazira se na numeričkom rješavanju tri obične diferencijalne jednadžbe. Posljedično, nema potrebe za dodatnom stabilizacijom jer integracijska procedura operira u trodimenzionalnom vektorskom prostoru lokalnih rotacijskih parametara.

Jedan redundantni parametar, problem pretvara u zahtjevnu zadaću rješavanja sustava 4 diferencijalno algebarske jednadžbe. Problem možemo izbjeći ako proces ažuriranja integracije prebacimo s parametrizacijske mnogostrukosti, dakle kvaterniona, na lokalnu tangencijalnu ravninu inkrementalnih rotacija. To je dozvoljeno radi već spomenutnog izomorfizma Lievih algebri grupe matrica rotacije $SO(3)$ te grupe jediničnih kvaterniona S^3. Radi toga, integracija kinematičkih diferencijalnih jednadžbi, događa se na Lievoj algebri $so(3)$ grupe rotacija $SO(3)$. Budući da se obične diferencijalne jednadžbe na $so(3)$ mogu riješiti bilo kojom od standardnih metoda integracije, koristit će se Runge-Kutta metoda drugog ili četvrtog reda točnosti. Počnimo s već spomenutom jednadžbom kinematičke rekonstrukcije, (102), izvedenom u prošlom poglavlju.

$$\dot{\mathbf{R}} = \mathbf{R}(t)\tilde{\omega}(t). \tag{114}$$

Tom je relacijom povezana kutna brzina $\omega(t) \in \mathbb{R}^3$ izražena u lokalnom koordinatnom sustavu i derivacija po vremenu matrice rotacije $\mathbf{R}(t)$. Dakle, ovim je izrazom opisano 'ponašanje' s obzirom na globalni inercijski koordinatni sustav. Radi izomorfizma $so(3)$ te $\mathbb{R}^3$, možemo napisati vezu između kososimetrične matrice $\tilde{\omega} \in so(3)$ te vektora $\omega \in \mathbb{R}^3$. Numeričkim rješavanjem jednadžbe (114), što je zapravo obične diferencijalne jednadžbe na Lievoj grupi $SO(3)$, možemo rekonstruirati matricu rotacije $\mathbf{R}(t)$ od lijevo invarijantnog polja $\tilde{\omega}(t) \in so(3)$, gdje je $so(3)$ već spomenuta Lieva algebra grupe $SO(3)$. Za rješavanje običnih diferencijalnih jednadžbi na Lievim grupama, zahtijevamo rješenje u obliku:

$$\mathbf{R}(t) = \mathbf{R_0}\exp(\tilde{\mathbf{u}}(t)), \tag{115}$$

pri čemu je zatvorena forma eksponencijalne mape na $SO(3)$ dana Rodriguesovom formulom:

$$\exp_{SO(3)}(\tilde{\mathbf{u}}) = \mathbf{I}_3 + \frac{\sin(\|\mathbf{u}\|)}{\|\mathbf{u}\|}\tilde{\mathbf{u}} + \frac{1-\cos(\|\mathbf{u}\|)}{\|\mathbf{u}\|^2}\tilde{\mathbf{u}}, \tag{116}$$

a $\mathbf{u}(t) \in \mathbb{R}^3$ je trenutni lokalni rotacijski vektor, dok je $\mathbf{R}_0$ matrica početne rotacije.

Tada, $\tilde{\mathbf{u}} \in so(3)$ predstavlja sustav običnih diferencijalnih jednadžbi na Lievoj algebri:

$$\dot{\tilde{\mathbf{u}}} = \mathrm{dexp}^{-1}_{-\tilde{\mathbf{u}}}(\tilde{\omega}(\mathbf{R}(\mathbf{t}))), \tag{117}$$

pri čemu se operator $\mathrm{dexp}^{-1}_{-\tilde{\mathbf{u}}}$ može razviti u red:

$$\mathrm{dexp}^{-1}_{-\tilde{\mathbf{u}}}(\tilde{\boldsymbol{\omega}}) = \tilde{\boldsymbol{\omega}} + \frac{1}{2}[\tilde{\mathbf{u}}, \tilde{\boldsymbol{\omega}}] + \frac{1}{12}[\tilde{\mathbf{u}}, [\tilde{\mathbf{u}}, \tilde{\boldsymbol{\omega}}]] + ... = \sum_{j=0}^{\infty} \frac{B_j}{j!}(-ad^j_{\tilde{\mathbf{u}}}(\tilde{\boldsymbol{\omega}})), \tag{118}$$

pri čemu su B_j Bernoullievi brojevi. Adjungirani operater $\mathrm{ad}_{\tilde{\omega}}$ definiran je Lievom zagradom:

$$\mathrm{ad}_{\tilde{\omega}} = \tilde{\mathbf{u}}\tilde{\boldsymbol{\omega}} - \tilde{\boldsymbol{\omega}}\tilde{\mathbf{u}} = [\tilde{\mathbf{u}}, \tilde{\boldsymbol{\omega}}], \quad \text{za sve} \quad \tilde{\boldsymbol{\omega}}(t), \tilde{\mathbf{u}} \in so(3). \tag{119}$$

Izraz (119) možemo napisati u zatvorenoj formi:

$$\mathrm{dexp}^{-1}_{-\tilde{\mathbf{u}}}(\tilde{\boldsymbol{\omega}}) = \boldsymbol{\omega} + \frac{1}{2}\tilde{\mathbf{u}}\boldsymbol{\omega} - \frac{\|\mathbf{u}\| \cot(\frac{\|\mathbf{u}\|}{2}) - 2}{2\|\mathbf{u}\|^2}\tilde{\mathbf{u}}^2\boldsymbol{\omega}. \tag{120}$$

Podsjetimo se da smo kvaternion $q \in \mathbb{H} = \mathbb{R}^4$ definirali kao 4 vektor:

$$q = (q_0, \mathbf{q}). \tag{121}$$

Vektorski dio možemo napisati kao $\mathbf{q} = q_i\mathbf{e}_i$, pri čemu je $\mathbf{e}_i = 1, 2, 3$ standardna ortonormalna baza u $\mathbb{R}^3$. Radi preglednosti, ponovo ćemo navesti produkt dva kvaterniona:

$$q \circ p = (q_0p_0 - \mathbf{q} \cdot \mathbf{p}, q_0\mathbf{p} + p_0\mathbf{q} + \mathbf{q} \times \mathbf{p}), \tag{122}$$

pri čemu vrijedi $q, p \in \mathbb{H}$, a standardni skalarni umnožak definiramo kao $\mathbf{q} \cdot \mathbf{p} = q_ip_i$. Jedinični kvaternioni čine grupu koja je izomorfna simplektičkoj grupi $Sp(1)$ kao i specijalnoj unitarnoj grupi $SU(2)$. Grupa jediničnih kvaterniona također je izomorfna jediničnoj sferi u $\mathbb{R}^4$, čiju ćemo definiciju ponoviti:

$$\mathcal{S}^3 = \left\{ q \in \mathbb{R}^4 \quad | \quad \|q\| = 1 \right\}, \tag{123}$$

Rotacijsko je gibanje opisano kao krivulja $q(t) \in \mathcal{S}^3$ u vremenu. Brzine, $\dot{q} \in T_q\mathcal{S}^3$ možemo zapisati uvodeći prvo prostor koso simplektičkih kvaterniona:

$$sp(1) = \left\{ \mathbf{w} \in \mathbb{R}^4 \quad | \quad \mathbf{w} + \tilde{\mathbf{w}} = (0, \mathbf{0}) \right\}, \tag{124}$$

pri čemu je $\tilde{\mathbf{w}}$ konjugacija čistog kvaterniona $\mathbf{w}$. O $sp(1)$ možemo razmišljati kao o tangentnom prostoru $Sp(1) \cong \mathbb{S}^3$ blizu identiteta grupe. Također, $sp(1)$ je ujedno i Lieva algebra grupe $Sp(1)$, izomorfna $so(3)$ i $\mathbb{R}^3$. Lieva algebra $sp(1)$ je skup čistih kvaterniona izomorfnih $\mathbb{R}^3$, tako da element $\mathbf{w} \in sp(1)$ možemo pridružiti vektoru $\mathbf{u} \in \mathbb{R}^3$. Sjetimo se da smo za $so(3)$ pridružili $\tilde{\boldsymbol{u}} \in so(3)$. Da bismo osigurali da su $sp(1)$, $so(3)$ i $\mathbb{R}^3$ izomorfne Lieve algebre, element $\mathbf{w} = (0, \frac{1}{2}\mathbf{u}) \in sp(1)$ mora

biti vezan s $\mathbf{u} \in \mathbb{R}^3$. Slično kao i za $\exp_{SO(3)}$, postoji zapis eksponencijalne mape $\exp_{S^3} : sp(1) \cong \mathbb{R}^3 \to Sp(1) \cong S^3$ razvijene u red

$$\exp_{S^3}(\mathbf{w}) = \sum_{k=0}^{\infty} \frac{\mathbf{w}^k}{k!} = \sum_{k=0}^{\infty} \left\{ \frac{(0, \frac{1}{2}\mathbf{u})^{2k}}{(2k)!} + \frac{(0, \frac{1}{2}\mathbf{u})^{2k+1}}{(2k+1)!} \right\}, \tag{125}$$

koja ako uzmemo $\mathbf{w}^2 = -\left\|\frac{1}{2}\mathbf{u}\right\|^2 (1, \mathbf{0})$, prelazi u:

$$\exp_{\mathcal{S}^3}(\mathbf{w}) = \sum_{k=0}^{\infty} \frac{(-1)^k \left\|\frac{1}{2}\mathbf{u}\right\|^{2k}}{(2k)!}(1, \mathbf{0}) + \frac{1}{\|\mathbf{u}\|} \sum_{k=0}^{\infty} \frac{(-1)^k \left\|\frac{1}{2}\mathbf{u}\right\|^{2k+1}}{(2k+1)!}(1, \mathbf{u}), \tag{126}$$

a u zatvorenoj formi ima oblik:

$$\exp_{\mathcal{S}^3}(\mathbf{w}) = \cos\left(\frac{1}{2}\|\mathbf{u}\|\right)(1, \mathbf{0}) + \frac{\sin(\frac{1}{2}\|\mathbf{u}\|)}{\|\mathbf{u}\|}(0, \mathbf{u}). \tag{127}$$

Jednadžba (127) daje nam jedinični kvaternion (četiri Eulerova parametra) koja opisuju rotaciju oko osi određene jediničnim vektorom paralelnim s $\mathbf{u} \in \mathbb{R}^3$ i kutem određenim s $\|\mathbf{u}\|$. Grupa kvaterniona, dakle grupa $SU(2)$, ima dvostruko prekrivanje grupe $SO(3)$. To je geometrijski razlog što možemo postići parametrizaciju slobodnu od singularnosti, na račun četiri redundantna parametra. Ovo također implicira da dva kvaterniona $q = (q_0, \mathbf{q})$ i $-q = (-q_0, -\mathbf{q})$ opisuju istu rotaciju [9]. Sada vidimo pozadinu tog rezultata koji smo opisali u prošlom poglavlju.

U standardnoj diferencijalno algebarskoj (DAE) formi integracije kvaterniona, jednadžba ograničenja mora eksplicitno biti uključena u matematički model i njezino zadovoljenje mora biti numerički forsirano tijekom integracije. Ova se redundantnost može izbjeći novopredloženom metodom integracije koja inherentno poštuje uvjet jediničnih kvaterniona bez eksplicitnog zahtjeva. Osim toga, možda je još važnije da novu metodu karakterizira formulacija kinematičkih diferencijalnih jednadžbi u prikladnijem obliku koji omogućuju veće korake integracije. Preciznije, standardni algoritam implicira numeričko rješenje globalnih linearnih kinematičkih diferencijalnih jednadžbi, iako je pozadinski proces trodimenzionalne rotacije esencijalno nelinearan proces baziran na $SO(3)$ mnogostrukosti. Inherentna linearizacija može utjecati na sveukupnu preciznost i efikasnost prevenirajući korištenje dužih koraka integracije koji bi eventualno mogli biti korišteni da smo primijenili lokalni nelinearni model običnih diferencijalnih jednadžbi [10].

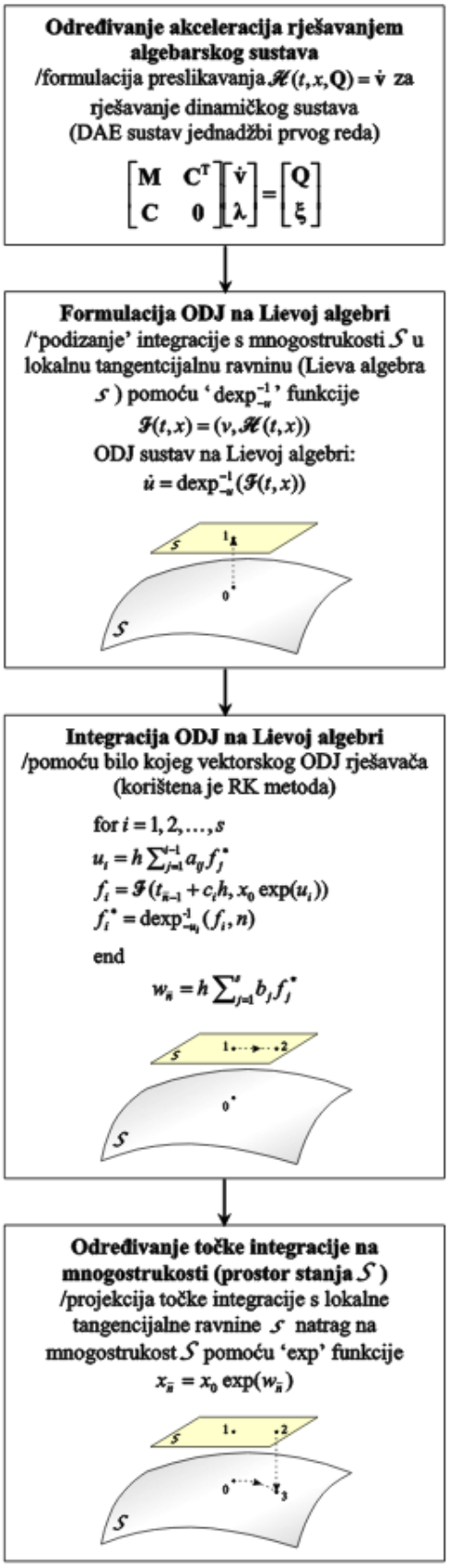
Prema predloženom algoritmu, ažuriranje rotacije u $n$tom koraku integracije, bazira se na eksponencijalnoj mapi (116) pri čemu će $\mathbf{u} \in \mathbb{R}^3$ predstavljati inkrementalnu rotaciju vektora koji ažurira rotaciju od q_n do q_{n+1}. Dakle, prema jednadžbi (115), uzorak za napredovanje koraka integracije oblika

$$q_{n+1} = q_n \circ \exp_{\mathcal{S}^3}(\mathbf{w}_n) = q_n \circ \exp_{S^3}\left(\left(0, \frac{1}{2}\mathbf{u}_n\right)\right), \tag{128}$$

pri čemu je zatvorena forma eksponencijalnog mapiranja na $\mathcal{S}^3$ dana jednadžbom (127), $\mathbf{w}_n$ je element Lieve algebre $sp(1)$ povezane s vektorom inkrementalne rotacije, a $\mathbf{u}_n \in \mathbb{R}^3$ je $n$ti inkrementalni vektor rotacije. Da bismo odredili vektor rotacije $\mathbf{u}_n$, pišemo jednadžbu kao

$$\dot{\tilde{\mathbf{u}}}_n = \mathrm{dexp}^{-1}_{-\tilde{\mathbf{u}}_n}\left(\tilde{\boldsymbol{\omega}}\left(q(t)\right)\right), \quad \tilde{\mathbf{u}}_{n_0} = \mathbf{0}, \tag{129}$$

gdje je operator dexp^{-1}_{-u} predstavljen u jednadžbi (119), a $\tilde{\mathbf{u}}_{n_0} = \mathbf{0}$ početni uvjet. Ove dvije jednadžbe bi trebale biti integrirane u svakom koraku integracije zajedno s jednadžbama dinamike kojima je opisano gibanje, a koje određuju polje brzina $\boldsymbol{\omega}$ iz polja akceleracija $\dot{\boldsymbol{\omega}}$. S obzirom da je gornja jednadžba obična diferencijalna jednadžba definirana na Lievoj algebri, dakle u linearnom vektorskom prostoru, za njeno rješavanje može se koristiti bilo koji standardni postupak za rješavanje običnih diferencijalnih jednadžbi [9]. Shematski prikaz opisane integracijske procedure prikazan je na slici 2. Da rezimiramo, procedura kreće od određivanja akceleracija za trenutni korak integracije, rješavanjem linearnog algebarskog sustava formuliranog kao DAE sustav jednadžbi prvog reda koji predstavlja dinamički model u matričnoj notaciji. Točka integracije se nakon toga 'podiže' iz nelinearnog prostora stanja mnogostrukosti, tj Lieve grupe u njezin lokalni (vektorski) tangentni prostor, dakle Lievu algebru gdje se provodi integracija klasičnih ODJ na linearnom vektorskom prostoru izraženih pomoću novouvedenih lokalnih koordinata. Po završetku integracije ODJ, točka integracije se projicira natrag na prostor stanja mnogostrukosti pomoću 'exp' funkcije, čime je korak integracije završen. Dakle, princip rješavanja se temelji na podizanju točke integracije s mnogostrukosti iz prostora stanja u lokalnu tangencijalnu ravninu iz koje se nakon integracijskog koraka vrši projekcija natrag na Lievu grupu prostora stanja [1].



Slika 2: Shema integracijske procedure DAE sustava na Lievoj grupi [1]

Red točnosti ukupnog algoritma ovisi samo o točnosti ODJ integratora korištenog za rješavanje gornje jednadžbe te indeksu skraćenja. U ovom radu, korištena je Runge Kutta metoda četvrtog reda točnosti, prikazana ispod

$$\begin{aligned}
&\boldsymbol{\omega}_0, q_0, \Delta t,\\
K_1 &= \Delta t \cdot \mathrm{dexp}(\boldsymbol{\omega_0}, \mathbf{0}),\\
k_1 &= \Delta t \cdot f(\boldsymbol{\omega_0}, q_0),\\
K_2 &= \Delta t \cdot \mathrm{dexp}\left(\boldsymbol{\omega}_0 + \frac{1}{2}k_1, \frac{1}{2}K_1\right)\\
k_2 &= \Delta t \cdot f\left(\boldsymbol{\omega}_0 + \frac{1}{2}k_1, q_0 \circ \exp_{\mathcal{S}^3}\left(\frac{1}{2}K_1\right)\right)\\
&\vdots\\
K_4 &= \Delta t \cdot \mathrm{dexp}(\boldsymbol{\omega}_0 + k_3, K_3)\\
k_4 &= \Delta t \cdot f\left(\boldsymbol{\omega}_0 + k_3, q_0 \circ \exp_{\mathcal{S}^3}(K_3)\right)\\
q_{out} &= q_0 \circ \exp_{\mathcal{S}^3}\left(\frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4)\right)\\
\boldsymbol{\omega}_{out} &= \boldsymbol{\omega_0} + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)
\end{aligned} \tag{130}$$

Jednadžba (127) može biti iskorištena kako bismo konstruirali algoritam za kinematičku rekonstrukciju rotacijskog kvaterniona $\mathbf{Q}$ iz polja kutnih brzina $\boldsymbol{\omega}$. Prvo određujemo vektor inkrementalne rotacije prema jednadžbi

$$\dot{\mathbf{u}} = \mathrm{dexp}^{-1}_{-\tilde{\mathbf{u}}}(\boldsymbol{\omega}) = \boldsymbol{\omega} + \frac{1}{2}\tilde{\mathbf{u}}\boldsymbol{\omega} - \frac{\|\mathbf{u}\|\cot(\frac{\|\mathbf{u}\|}{2}) - 2}{2\|\mathbf{u}\|^2}\tilde{\mathbf{u}}^2\boldsymbol{\omega}, \tag{131}$$

rekonstruirajući zatim orijentaciju letjelice koristeći jednadžbu (127). Drugi način za izbjegavanje problema koji se javljaju prilikom korištenja konvencionalnih metoda, korištenje je slične teorije, međutim, ovaj puta se translacijski i rotacijski dio ne rješavaju odvojeno nego zajedno i cjelokupno je gibanje opisano dualnim kvaternionima [2]. Kako ovaj pristup nije implementiran, nećemo ga pobliže opisivati.

2.4 Dinamički model kvadrirotorske letjelice

Za razvoj računalnog modela bit će korištena kvadrirotorska letjelica u X konfiguraciji kao u [2]. U svrhu pojednostavljenja modela, zanemarena je aerodinamička fenomenologija kako bi fokus bio na direktnoj dinamici. Pretpostavljamo da je pogonska sila u smjeru osi motora F_{Ti} a moment parazitskog otpora $\boldsymbol{T}_{Di}$ te se obje veličine mogu napisati u funkcijskoj vezi s kutnom brzinom rotora:

$$F_{T^i} = f(\omega_i), \tag{132}$$

$$\mathbf{T}_{D^i} = f(\omega_i), \tag{133}$$

pri čemu ω_i predstavlja kutnu brzinu i-tog rotora. Dio dinamičkog modela koji se odnosi na translaciju može biti zapisan kao:

$$m\dot{\boldsymbol{v}} = \boldsymbol{R}\begin{bmatrix} 0 \\ 0 \\ \sum_{i=1}^{4} F_{Ti} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix}, \tag{134}$$

pri čemu m predstavlja masu letjelice, $\boldsymbol{v}$ je translacijska brzina letjelice izražena u globalnom koordinatnom sustavu, $\boldsymbol{R}$ je matrica rotacije koju možemo konstruirati preko poznatog kvaterniona:

$$\boldsymbol{R} = 2\begin{bmatrix} q_0^2 + q_1^2 - \frac{1}{2} & q_1q_2 - q_0q_3 & q_1q_3 + q_0q_2 \\ q_1q_2 + q_0q_3 & q_0^2 + q_2^2 - \frac{1}{2} & q_2q_3 - q_0q_1 \\ q_1q_3 - q_0q_2 & q_2q_3 + q_0q_1 & q_0^2 + q_3^2 - \frac{1}{2} \end{bmatrix}. \tag{135}$$

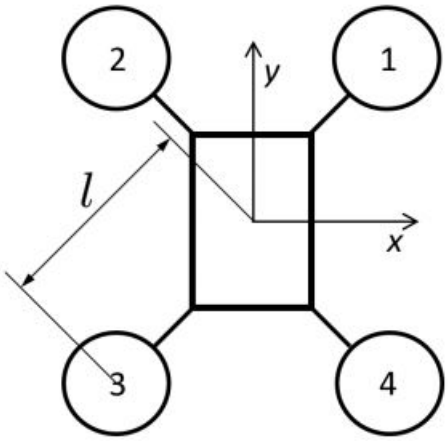
Jednadžbu koja opisuje translaciju možemo napisati i kao vezanu za pomični koordinatni sustav, vezana za letjelicu:

$$m\dot{\boldsymbol{v}}' = \begin{bmatrix} 0 \\ 0 \\ \sum_{i=1}^{4} F_{Ti} \end{bmatrix} + \boldsymbol{R}^T\begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix} - m\tilde{\boldsymbol{\omega}}\boldsymbol{v}', \tag{136}$$

pri čemu $\boldsymbol{v}'$ i $\boldsymbol{\omega}$ predstavljaju translacijsku i kutnu brzinu izraženu u koordinatnom sustavu letjelice. Iz toga slijedi da ukupnu rotacijsku dinamiku kvadrotorske letjelice možemo opisati jednadžbom:

$$\boldsymbol{J}\dot{\boldsymbol{\omega}} = \sum_{i=1}^{4}\boldsymbol{T}_{Ti} + \sum_{i=1}^{4}\boldsymbol{T}_{Di} - \tilde{\boldsymbol{\omega}}\sum_{i=1}^{4}\boldsymbol{J}_{mi}\boldsymbol{\omega}_{mi} - \sum_{i=1}^{4}\boldsymbol{J}_{mi}\dot{\boldsymbol{\omega}}_{mi}, \tag{137}$$

pri čemu je $\boldsymbol{J}$ matrica inercije letjelice a $\boldsymbol{J}_{mi}$ i $\boldsymbol{\omega}_{mi}$ predstavljaju matricu inercije i relativnu kutnu brzinu i-tog rotora s obzirom na letjelicu. Pogonski moment $\boldsymbol{T}_{Ti}$ računamo kao $\boldsymbol{T}_{Ti} = \sum_{i=1}^{4}\tilde{r}_i F_{Ti}$ gdje r_i predstavlja ishodište koordinatnog sustava vezanog za i-ti rotor. Svi izrazi izraženi su s obzirom na koordinatni sustav vezan za tijelo kvadririotorske letjelice.



Slika 3: Kvadrirotorska letjelica u X konfiguraciji [2]

2.5 Implementacija nove numeričke metode vremenske integracije jediničnih kvaterniona

Sada ćemo navesti detalje implementacije za neke dijelove algoritma. Radi preglednosti, prikazani će biti samo isječci Python koda, dok će C implementacija istih procedura biti dana u prilogu. Redoslijed izlaganja pratit će sliku 2.
Deklaracija varijabli zajedno s početnim vrijednostima prikazana je ispod:

```
#initial rotational quaternion
q = np.zeros((4,N))
q[:,[0]] = np.array([[1],[0],[0],[0]])

#initial angular speed
omegaVector = np.zeros((3,N))
omegaVector[:,[0]] = np.array([[0],[0],[0]])

#initial position
p = np.zeros((3, N))
p[:,[0]] = np.array([[0],[0],[-20]])

#initial velocity
v = np.zeros((3, N))
v[:,[0]] = np.array([[0],[0],[0]])
```

Potrebno je također definirati i parametre manevra, za naš slučaj to je valjanje (engl. *roll*). Izlaz iz te funkcije služi kao ulazni parametar za funkcije koja na temelju tih rezultata računaju odgovarajuće sile i momente koji rotori moraju postići da bi se ostvario željeni manevar. Napominjemo da postoji i mogućnost simuliranja manevra skretanja (engl.*yaw*) te penjanja (engl. *yaw climb*).

```
def omegamfun(t, type):
    if type == 'roll':
```

```
        omegam_out = (2*np.pi/60)*((0*np.sqrt(1/(6.11e-8)))*np.
                                                array([[1],[1],[0],[0]])
                                                 np.sqrt(1.4516625/(6.11e
                                                -8))*np.array([[0],[0],[1
                                                ],[1]]))
    return omegam_out
```

Prvo je potrebno odrediti akceleracije rješavanjem algebarskog sustava. Stoga prvo konstruiramo matricu rotacije iz ulaznog kvaterniona, prema relaciji (135). Varijabla 'dv' predstavlja ubrzanje, koje računamo iz translacijskog dijela dinamike letjelice prema (134) dok se u varijablu 'rotational' sprema rotacijski dio dinamike letjelice prema izrazu (137). Sada su postignuti uvjeti da bi se pristupilo rješavanju algebarskog sustava što je i učinjeno u zadnjem redu.

```
def new_eul(aircraft,motor,F_t,T_t,T_d,omega,omegam,q):
    R = np.zeros((3,3,))
    R[0,0] = 2*(q[0]**2+q[1]**2-0.5)
    R[0,1] = 2*(q[1]*q[2]-q[0]*q[3])
    R[0,2] = 2*(q[1]*q[3]+q[0]*q[2])
    R[1,0] = 2*(q[1]*q[2]+q[0]*q[3])
    R[1,1] = 2*(q[0]**2+q[2]**2-0.5)
    R[1,2] = 2*(q[2]*q[3]-q[0]*q[1])
    R[2,0] = 2*(q[1]*q[3]-q[0]*q[2])
    R[2,1] = 2*(q[2]*q[3]+q[0]*q[1])
    R[2,2] = 2*(q[0]**2+q[3]**2-0.5)
    vector = np.zeros((3,1))
    vector[2,0] = sum(F_t)
    dv = (np.add(np.matmul(R, vector), np.array([[0],[0],[aircraft.
                                        mass*9.80665]])))/aircraft.
                                        mass
    rotational = T_t + T_d - np.matmul(np.matmul(skew(omega),
                                        aircraft.J),omega) - np.
                                        matmul(skew(omega),np.array([
                                        [0],[0],[motor.inertia * np.
                                        sum(np.multiply(omegam,motor.
                                        rot_dir))]]))
    domega = np.linalg.solve(aircraft.J,rotational)
    return dv,domega
```

Pomoću diferencijalnog eksponencijalnog operatera, problem integracije prebacuje se na lokalnu tangencijalnu ravninu ili Lievu algebru.

```
def dexp(omega,u):
    norm_u = np.linalg.norm(u)
    skew_u = skew(u)
    du = SO3_to_R3(skew(omega))
    return du
```

Eksponencijalna mapa služi za projekciju točke integracije s lokalne tangencijalne razine (Lieve algebre) natrag na mnogostrukost. Implementacija slijedi iz izraza (127).

```
def exp_S3(u):
    norm_u = np.linalg.norm(u)
    if norm_u < 10**(-12):
        u_S3 = np.array([[1],[0],[0],[0]])
    else:
        u_S3 = np.cos((1/2) * norm_u) * np.array([[1],[0],[0],[0]])
                                          + np.sin((1/2) * norm_u)
                                         /norm_u * np.concatenate
                                         ((np.array([[0]]),u))
    return u_S3
```

Postoji međutim i alternativan zapis eksponencijalne mape, a onda i implementacija, prema formuli (146). Iz priloženog isječka koda vidljivo je da smo koristili 5 članova reda.

```
def exp_S3(u):
    norm_u = np.linalg.norm(u)
    firstTerm = 0
    secondTerm = 0
    if norm_u < 10**(-12):
       u_S3 = np.array([[1],[0],[0],[0]])
    elif norm_u > 10**(-12):
       for k in range(5): #number of terms in expansion
        firstTerm = firstTerm + ((-1)**(k)*(np.linalg.norm(0.5*u))
                                          **(2.*k))/(np.math.
                                          factorial(2*k))
        secondTerm = secondTerm + 1./np.linalg.norm(u)*(((-1)**(k)*
                                           np.linalg.norm(0.5*u)**(
                                          2*k+1))/(np.math.
                                          factorial(2*k+1)))
    u_S3 = np.array([[firstTerm],[secondTerm*u[0]],[secondTerm*u[1]
                                    ],[secondTerm*u[2]]])
    return u_S3
```

Cjelokupna integracijska procedura prikazana je na isječku koda ispod:

```
def rk4geom(t, h, aircraft, motor, omega, v, q, p, type):
    L = np.zeros((3,4))
    L[0,0] = -q[1]
    L[0,1] =  q[0]
    L[0,2] =  q[3]
    L[0,3] = -q[2]
    L[1,0] = -q[2]
    L[1,1] = -q[3]
```

```
    L[1,2] =  q[0]
    L[1,3] =  q[1]
    L[2,0] = -q[3]
    L[2,1] =  q[2]
    L[2,2] = -q[1]
    L[2,3] =  q[0]
    appendedMatrix = np.append(q,np.transpose(L), axis = 1)      #
                                        this was added later
    omegam = omegamfun(t,type)
    (vk1, k1) = new_eul(aircraft, motor, thrust_forces(motor,
                                        om2rpm(omegam)),
                                        thrust_torque(motor, aircraft
                                        , om2rpm(omegam)),
                                        drag_torque(motor, om2rpm(
                                        omegam)), omega, omegam, q)
    uk1 =  dexp(omega, np.zeros((3,1)))
    pk1 = v

    omegam = omegamfun(t+1/2*h,type)
    (vk2, k2) = new_eul(aircraft, motor, thrust_forces(motor,
                                        om2rpm(omegam)),thrust_torque
                                        (motor,aircraft,om2rpm(omegam
                                        )), drag_torque(motor,om2rpm(
                                        omegam)), omega + 1/2*h*k1,
                                        omegam, np.matmul(np.
                                        concatenate((q,np.transpose(L
                                        )),axis = 1),exp_S3(1/2*h *
                                        uk1)))
    uk2 = dexp(omega + 1/2 * h * k1, 1/2 * h* uk1)
    pk2 = v + 1/2 * h * vk1

    omegam = omegamfun(t+1/2*h, type)
    (vk3,k3) = new_eul(aircraft, motor, thrust_forces(motor, om2rpm
                                        (omegam)),thrust_torque(motor
                                        ,aircraft,om2rpm(omegam)),
                                        drag_torque(motor,om2rpm(
                                        omegam)), omega + 1/2*h*k2,
                                        omegam, np.matmul(np.
                                        concatenate((q,np.transpose(L
                                        )),axis = 1),exp_S3(1/2*h *
                                        uk2)))
    uk3 = dexp(omega + 1/2 * h * k2, 1/2 * h * uk2)
    pk3 = v + 1/2 * h * vk2

    omegam = omegamfun(t + h, type)
    (vk4, k4) = new_eul(aircraft, motor, thrust_forces(motor,
                                        om2rpm(omegam)),thrust_torque
```

```
                                    (motor,aircraft,om2rpm(omegam
                                    )), drag_torque(motor,om2rpm(
                                    omegam)), omega + h*k3,
                                    omegam, np.matmul(np.
                                    concatenate((q,np.transpose(L
                                    )),axis = 1),exp_S3(h * uk3))
                                    )
    uk4 = dexp(omega + h*k3, h * uk3)
    pk4 = v + h *vk3

    omega_out = omega + 1/6 * h * (k1 + 2*k2 + 2*k3 + k4)
    #q_out = np.matmul(np.concatenate((q,np.transpose(L)),axis = 1)
                                    ,exp_S3(1/6 *h * (uk1+ 2*uk2
                                    + 2 * uk3+ uk4)))
    q_out = np.matmul(appendedMatrix, exp_S3(1/6 *h * (uk1+ 2*uk2 +
                                     2 * uk3+ uk4)))     #this
                                    line was added later

    v_out = v + 1/6 * h * (vk1 + 2*vk2 + 2*vk3 + vk4)
    p_out = p + 1/6 * h * (pk1 + 2*pk2 + 2*pk3 + pk4)

    return omega_out, v_out, q_out, p_out
```

Sada možemo usporediti implementaciju opisane metodu s klasičnom metodom. U njoj je prvo eksplicitno prisutan uvjet jedinične duljine kvaterniona, a onda i potreba za stabilizacijom nakon svakog koraka integracije. Neće se ulaziti u cjelokupan izvod, nego se čitatelja upućuje na [10]. Formulacija kompletnog algoritma slijedi ispod.

$$
\begin{aligned}
\boldsymbol{\omega}_0, q_0, \Delta t \\
K_1 = \Delta t \cdot \frac{1}{2} \cdot \mathbf{L}^T(q_0) \cdot \boldsymbol{\omega}_0 \\
k_1 = \Delta t \cdot f(\boldsymbol{\omega_0}, q_0), \\
K_2 = \Delta t \cdot \frac{1}{2} \cdot \mathbf{L}^T(q_0) \cdot \left(q_0 + \frac{1}{2}K_1\right) \cdot \left(\boldsymbol{\omega}_0 + \frac{1}{2}k_1\right) \\
k_2 = \Delta t \cdot f\left(\boldsymbol{\omega}_0 + \frac{1}{2}k_1, q_0 + \frac{1}{2}K_1\right) \\
\vdots \\
K_4 = \Delta t \cdot \frac{1}{2} \cdot \mathbf{L}^T(q_0 + K_3) \cdot (\boldsymbol{\omega}_0 + k_3) \\
k_4 = \Delta t \cdot f\left(\boldsymbol{\omega}_0 + k_3, q_0 + K_3\right) \\
q_{out} = q_0 + \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4) \\
\omega_{out} = \omega_0 + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)
\end{aligned}
\tag{138}
$$

Određivanje derivacije kutne brzine $\boldsymbol{\omega}$ određuje se iz funkcije $f(\boldsymbol{\omega}, q)$ dok $\mathbf{L}$ označava

matricu transformacija kvaterniona koja je definirana kao:

$$\mathbf{L} = \begin{bmatrix} -q_1 & q_0 & q_3 & -q_2 \\ -q_2 & -q_3 & q_0 & q_1 \\ -q_3 & q_2 & -q_1 & q_0 \end{bmatrix}. \tag{139}$$

Kinematske diferencijalne jednadžbe koje povezuju kutnu brzinu su prikazane ispod. Jednadžba (140) povezuje derivacije kvaterniona i kutne brzine dok je jednadžba (141) dodatni uvjet na jediničnost kvaterniona i on je kao takav algebarska jednadžba.

$$\dot{q} = \frac{1}{2}\mathbf{L}^T\boldsymbol{\omega}, \tag{140}$$

$$q^T q - 1 = 0. \tag{141}$$

Treba naglasiti da se numerički izračunati kvaternion q_{out} mora stabilizirati nakon svakog koraka integracije prema jednadžbi:

$$q = \frac{1}{\sqrt{q+\delta}} q_{out}, \tag{142}$$

gdje je $\delta = q_{out}^T q_{out} - 1$. Implementacija u Pythonu dana je ispod.

```
def rk4cl(t, h, aircraft, motor, omega, v, q, p, type):
    omegam = omegamfun(t, type)
    (vk1,k1) = new_eul(aircraft, motor, thrust_forces(motor, om2rpm
                                        (omegam)), thrust_torque(
                                        motor, aircraft, om2rpm(
                                        omegam)), drag_torque(motor,
                                        om2rpm(omegam)), omega,
                                        omegam, q)
    qk1 = 1/2 * quat_vect(q,omega)
    pk1 = v

    omegam = omegamfun((t+1)/(2*h),type)
    (vk2, k2) = new_eul(aircraft, motor, thrust_forces(motor,
                                        om2rpm(omegam)),
                                        thrust_torque(motor, aircraft
                                        , om2rpm(omegam)),
                                        drag_torque(motor, om2rpm(
                                        omegam)), omega + 1/2*h*k1,
                                        omegam, q+1/2*h*qk1)
    qk2 = 1/2 * quat_vect(q+1/2*h*qk1,omega + 1/2*h*k1)
    pk2 = v + 1/2*h*vk1

    omegam = omegamfun(t+1/2*h, type)
    (vk3,k3) = new_eul(aircraft, motor, thrust_forces(motor, om2rpm
                                        (omegam)), thrust_torque(
```

```
                                        motor, aircraft, om2rpm(
                                        omegam)), drag_torque(motor,
                                        om2rpm(omegam)), omega + 1/2*
                                        h*k2, omegam, q+1/2*h*qk2)
    qk3 = 1/2 * quat_vect(q+1/2*h*qk2, omega + 1/2*h * k2)
    pk3 = v + 1/2*h*vk2

    omegam = omegamfun(t+h, type)
    (vk4, k4) = new_eul(aircraft, motor, thrust_forces(motor,
                                        om2rpm(omegam)),
                                        thrust_torque(motor, aircraft
                                        , om2rpm(omegam)),
                                        drag_torque(motor, om2rpm(
                                        omegam)), omega + h*k3,
                                        omegam, q+h*qk3)
    qk4 = 1/2 * quat_vect(q+h*qk3, omega + h * k3)
    pk4 = v + h*vk3

    omega_out = omega + 1/6*h * (k1 + 2*k2 + 2*k3 + k4)
    q_out = q + 1/6*h * (qk1 + 2*qk2 + 2*qk3 + qk4)
    #stabilization
    q_out = q_out / np.sqrt(np.matmul(np.transpose(q_out),q_out))

    v_out = v + 1/6*h * (vk1 + 2*vk2 + 2*vk3 + vk4)
    p_out = p + 1/6*h * (pk1 + 2*pk2 + 2*pk3 + pk4)

    return omega_out, v_out, q_out, p_out
```

Uspoređujući dvije metode, vidimo da su glavne razlike u kinematičkim jednadžbama te u ažuriranju rotacijskog kvaterniona od q_n do q_{n+1}. U novopredloženoj metodi, ažuriranje rotacije temeljeno je na eksponencijalnoj mapi, dok se u klasičnoj proceduri ažuriranje odvija na standardni način množenjem derivacija kvaterniona duljinom koraka integracije. Bitno je napomenuti da oba algoritma imaju jednako ažuriranje kutne brzine.

$$\dot{q} = \frac{1}{2}\mathbf{L}^T\boldsymbol{\omega}, \tag{143}$$

$$\dot{\tilde{\mathbf{u}}}_n = \mathrm{dexp}^{-1}_{-\tilde{\mathbf{u}}_n}\left(\tilde{\boldsymbol{\omega}}\left(q(t)\right)\right). \tag{144}$$

Izravno uspoređujući metode, jednadžba (143) čini sustav od 4 linearne jednadžbe dok jednadžba (144) čini sustav 3 nelinearne jednadžbe koje je potrebno riješiti.

3 PROGRAMSKE PARADIGME U KONTEKSTU NUMERIČKIH SIMULACIJA

Nakon podjele programskih jezike na danas dominantne programske paradigme, za svaku pojedinačno ćemo, uz kratki opis, ukratko predstaviti jedan karakterističan jezik. Zatim ćemo postaviti predstavljene paradigme u kontekst simulacije mehaničkih sustava te relevantnost svake od njih za provedbu simulacije. Poglavlje ćemo završiti opisom hardverske infrastrukture korištene za realizaciju simulacije.

3.1 Programske paradigme

Programske paradigme su način klasifikacije programskih jezika prema njihovim značajkama. Može se reći da je glavna podjela na imperativnu i deklarativnu programsku paradigmu. Ugrubo, u imperativnoj paradigmi programer instruira računalo da mijenja stanje dok u deklarativnoj paradigmi programer definira svojstva očekivanih rezultata, ali ne i način kako doći do njih. Imperativnu paradigmu dalje možemo podijeliti na proceduralnu i objektno orijentiranu. Kao glavnu razliku između dvije spomenute paradigme, možemo istaknuti da se u proceduralnom programiranju specificiraju koraci koje program mora obaviti da bi postiglo željeno stanje za razliku od objektno orijentiranog programiranja, gdje je kod organiziran na način da su u objektima grupirani podaci i metode te opis njihove međusobne interakcije. Dakle, proceduralno programiranje koristi procedure da bi operiralo na strukturama podataka dok se kod objektno orijentiranog programiranja metode i strukture podataka grupiraju u već spomenute objekte. Druga podjela programskih jezika koju moramo spomenuti je na one koji su interpretirani i one koji su kompajlirani. Kompajlirani jezik je onaj u kojem je program, nakon kompajliranja, skup instrukcija u strojnom kodu namijenjen za izvršavanje na procesoru. Kod interpretiranog jezika, instrukcije nisu izvršene odmah od strane procesora, nego ih izvršava neki drugi, posrednički program. U modernim programskim jezicima, za krajnjeg korisnika, razlika između paradigmi postaju sve manje vidljive. Međutim, potrebno je istaknuti neke razlike koje dolaze do izražaja prilikom implementacije. Generalno, može se reći da se programi pisani u kompajliranim jezicima izvode brže. Za razliku od toga, interpretirani jezici su fleksibilniji, nema potrebe za kompilacijom i može se reći da je lakše raditi s njima. Međutim, cijena spomenute fleksibilnosti najjasnije dolazi do izražaja u vremenu izvođenja programa koje može biti i nezanemarivo duže, pogotovo u kontekstu simulacija s ograničenim računalnim

resursima. Kao najklasičniji primjer, možemo navesti inicijalizaciju varijabli. Kompajlirani jezici, zahtijevaju da se definira tip svake varijable prije kompilacije, dok interpretirani jezici dopuštaju inicijalizaciju varijable, bez deklaracije, koju obavlja interpreter. U domeni znanstvenog računanja (engl. *scientific computing*), dominantna paradigma je imperativna, a kao relevantne jezike, uzet ćemo C i Fortran77. Kada govorimo o numeričkim zadaćama, spomenuti jezici imaju nekoliko značajnih prednosti nad drugim jezicima. Ugrubo, može se reći da oba jezika sliče strojnom kodu u koji su kompajlirani. Još slikovitije, može se reći da je kompajleru jasno kako će određenu rutinu prevesti u strojni kod. Nedostatak kompleksnijih struktura u jeziku, kao što su predlošci (engl. *templates*) i hijerarhije nasljeđivanja, kod čine preglednijim i shvatljivijim. Jednostavnije strukture podataka u pravilu znače i jednostavnije korištenje alata za otklanjanje grešaka u kodu (engl. *debugging*). Najjasnije se razlika s obzirom na interpretirani jezik ogleda u opsegu koda. Za istu namjenu, izvorni kod za neku aplikaciju napisanu u C-u može biti i red veličine veći nego kod nekih interpretiranih jezika. Također, mnoge mehanizme koji su inherentno ugrađeni u interpretirane jezike, korisnik mora samostalno napisati. Jedan od najznačajnijih nedostataka C-a i FORTRANA u kontekstu znanstvenog računanja, nedostatak je prostora imena (engl. *namespace*). Nedostatak hijerarhijske strukture varijabli i funkcija rezultira nespretno dugim imenima, sve kako bi se izbjegao problem kolizije u nomenklaturi varijabli, što pogotovo može doći do izražaja prilikom istovremenog korištenja više vanjskih knjižnica. Drugi problem je ručno upravljanje memorijom, što najviše do izražaja dolazi u C-u. Za razliku od C-a, u C++-u se naime, objekti automatski stvaraju ili brišu u prostoru funkcije ili programa (engl. *scope*) i programer ne mora eksplicitno brinuti o tome. Objekti su strukture koje sadrže varijable i funkcije ili metode koje operiraju na varijablama. C kao takav ne poznaje objekte kao strukture podataka, nego se ista stvar postiže strukturama i pokazivačima na funkcije, što je nesumnjivo skok u kompleksnosti. Objektno orijentirana paradigma ima izvrsne mehanizme organizacije u prostoru imena (engl. *namespace*). Međutim, upravo to može biti nedostatak ako se stvore prekompleksne hijerarhije, što je problem nekih modernih knjižnica za znanstveno računanje. Najpopularnija pristup generiranju koda u području znanstvenog računanja su C++ predlošci. Pristup omogućuje pisanje koda u kojem su postavljena rezervirana mjesta (engl. *placeholder*) na čije mjesto dolazi konkretna varijabla prilikom kompilacije. Višejezični kod (engl. *multilanguage code*) dozvoljava programeru da kombinira dobre strane različitih programskih paradigmi. Popularna kombinacija u znanstvenom računanju je brzina C-a ili Fortrana i fleksibilnost interpretiranog jezika kakav je Python. Specijalizirani Python alati, poput Cytona ili SWIG-a, razvijeni su za ovijanje (engl. *wrapping*) C knjižnica. Cython potpuno automatizira manipulaciju strukturama podataka, pozivima Pythonovih rutina te

pretvorbom varijabli. Objektna struktura C++-a može biti u potpunosti preslikana na objektnu strukturu Pythona. Također, Cython se brine i o rukovanju (engl. *handling*) pogrešaka i iznimaka. Lakoća procesiranja niza (engl. *string*) u Pythonu te manipulacija strukturama podataka, korisne su za upravljanje ulazom i izlazom podataka. Mnoge knjižnice koriste Python kao krovni jezik za kontrolu, dok se za 'niske' numeričke procedure okreću C-u ili Fortranu. Koristeći spomenute alate (Cython, SWIG), proces je moguće gotovo u potpunosti automatizirati. Pythonova fleksibilnost prilikom pristupa sučelju za programiranje aplikacija (engl. *application programming interface* - API) pojedine knjižnice, omogućuje kombiniranje više knjižnica. Kombinacija knjižnica namijenjenih za rješavanje različitih fizikalnih problema može se iskoristiti prilikom 'napada' na kompleksniju problematiku, kao što su npr. višerazinski ili multifizikalni problemi. Treba napomenuti da ovijanje omogućuje korištenje CUDA ili OpenGL knjižnica koje su predviđene za paralelno računanje na grafičkim karticama. Glavni nedostatak višejezičnih kodova dolazi do izražaja prilikom traženja grešaka u kodu (engl. *debugging*). Do dana današnjeg ne postoje razvijeni alati, prvenstveno se pod time misli na alate za traženje pogreške u kodu (engl. *debugger*) koji bi prilikom traženja grešaka u kodu, jednostavno prelazili iz jedne paradigme u drugu.

3.2 Interpretirani jezici

Interpretirani jezici kao što je Python spregnut s Numpy knjižnicom za numeričke simulacije, prikladni su za brzu prototipizaciju, razvoj algoritma ili analizu podataka u znanstvenom računanju. Međutim, radi relativno niske numeričke efikasnosti, to ih najčešće ne čini dobrim izborom za simulacije u realnom vremenu. Umjesto toga, kompajlirani jezici kao što su C, C++ ili Fortran, koristeći drugačije numeričke knjižnice za matrični račun će biti češće korišteni za simulacije. Prema drugoj podjeli, Python spada u objektno orijentiranu paradigmu što je dobra programska paradigma za izgradnju modularnog, skalabilnog i ponovo upotrebljivog softvera.
Python je jezik visoke razne, interpretiran i opće namjene te je trenutno jedan od najkorištenijih jezika u mnogim područjima, između ostalog i u polju znanstvenog računanja. Iako prvotna namjena jezika nije bila znanstveno računanje, njegove inherentne karakteristike čine ga dobro opremljenim za tu svrhu. Prije svega, Python je poznat radi posjedovanja čiste i intuitivne sintakse. Takva sintaksa čini ga lakšim za održavanje, što generalno rezultira manjim brojem grešaka ali i olakšava razvoj koda. Znamo da Python, kao interpretirani jezik, u većini slučajeva ne može konkurirati kompajliranim jezicima što se tiče brzine izvođenja, međutim u slučaju implementacije novog rješenja, radi svoje fleksibilnosti Python je bolji izbor. S jedne strane performanse koda, a s druge vrijeme razvoja koda, predstavljaju dva kon-

tradiktorna uvjeta između kojih programer mora pronaći sredinu. Da bi se doskočilo tome, uz Python kao temelj, kroz godine je izrastao cijeli ekosustav alata i knjižnica dizajniranih specifično s namjerom da budu korištene u svrhe znanstvenog računanja. Ekosustav uključuje sve od integriranih razvojnih okolina kao što je npr. *Spyder* pa do knjižnica od kojih su najkorištenije *NumPy* i *SciPy*. U njih su implementirane knjižnice optimizirane za operacije s vektorima i matricama što je zapravo jezgra koja omogućuje uporabu Pythona za znanstveno računanje. Ne smijemo naravno zaboraviti da su spomenute knjižnice otvorenog koda što znači da su svakome dostupne za besplatno korištenje [11].

3.3 Kompajlirani jezici

Za razliku od interpretiranih jezika, kod kojih se prevođenje događa paralelno za vrijeme izvođenja, kompajler kompajlirani jezik prevodi u strojni jezik prije izvođenja. Dakle, kompajler simboličke naredbe (tekst) koje programer napiše prevodi u strojni jezik koji je niz binarnih znakova predviđen za obradu na procesoru. Radi toga, generalno se može reći da je izvođenje kompajliranih jezika brže nego interpretiranih. Niži programski jezici su u pravilu kompajlirani. Međutim, kompajler kod prevodi u strojni jezik sintaksa koja je određena prema arhitekturi procesora koji će taj kod izvršavati. Dakle, kod kompajliran na jednoj platformi neće uvijek biti moguće pokrenuti na drugom računalu.

C je proceduralni jezik opće namjene. Razvijan je zajedno s Unix operativnim sustavom sedamdesetih godina prošlog stoljeća u Bell laboratorijima pri čemu su najznačajniju ulogu odigrali Dennis Ritchie i Ken Thomphson. Iako je vrlo prikladan za pisanje kompajlera i operativnih sustava za što je originalno zamišljen, C se pokazao kao vrlo dobar programski jezik za pisanje programa u mnogo različitih domena. C je relativno nizak programski jezik. U prijevodu, to znači da C inherentno ne pruža nikakve načine za baratanje kompozitnim objektima kao što su skupovi ili liste. Ne postoje operacije kojima se manipulira čitavim nizom iako se strukture mogu kopirati kao cjelina. Jezik ne definira nikakav mehanizam za alokaciju memorije osim statičkih definicija kao što također nema ni mehanizam za automatskim upravljanje memorijom sakupljanje smeća (engl. *garbage collection*) [12].

Cyton je istodobno i programski jezik koji spaja Python s nekim od statičkih jezika kao što su C/C++ ali je također i kompajler koji prevodi izvorni kod (engl. *source code*) napisan u Pythonu u efikasan C/C++ kod. Vidjeli smo već da su Python i C vrlo različiti jezici, međutim upravo ih te suprotnosti čine komplementarnima. Uspoređujući vremena izvođenja, Python može biti i nekoliko redova veličine sporiji nego C. Zanimljivo je da je Python kod odmah i valjani Cython kod i može se ga se kao takvog odmah kompajlirati. Međutim, dodajući statičke deklaracije u Python

kod, na neki način olakšavamo kompajleru prevođenje u C kod i radi toga onda možemo očekivati značajnija ubrzanja. Cython je, međutim, moguće koristiti i u drugom smjeru (od C-a prema Pythonu) ali u tu problematiku nećemo previše ulaziti [13].
Matlab je jezik visoke razine namijenjen za znanstveno računanje i vizualizaciju podataka, izgrađen oko interaktivnog okoliša za programiranje. Polako postaje 'mainstream' platforma za znanstveno računanje, kako u industriji, tako i na brojnim institutima i sveučilištima. Ono što ga čini iznimno popularnim je lakoća pisanja i testiranja koda, dozvoljavajući pritom programeru da se više koncentrira na rješenje problema, a manje na detalje implementacije. S obzirom na to da ne postoji potreba za kompajliranjem te ponovnim izvršavanjem nakon svake nove kompilacije, vrijeme razvoja programa u Matlabu mnogo je kraće nego za ekvivalentne programe pisane u npr C-u ili Fortranu. Spomenimo međutim i negativnu stranu, a to je da Matlab ne može generirati samostojeće (engl. *stand alone*) aplikacije što znači da se program može pokrenuti samo na računalu na kojem je već instaliran Matlab. Neke od prednosti koje Matlab posjeduje nad drugim općeprihvaćenim jezicima su to što sadržava velik broj funkcija kojima su u pozadini numeričke knjižnice kao npr. LINPACK ili EISPACK, što znači da mnogi uobičajeni zadaci (npr. rješavanje sustava jednadžbi) mogu biti riješeni pozivom jedne funkcije.

3.4 Simulacija na ugradbenim sustavima

Simulacije dinamike sustava na ugradbenim računalima od posebnog su interesa prilikom projektiranja algoritama za kontrolu koji se oslanjaju na dinamički model sustava [14]. Ispravan odabir odgovarajuće programske paradigme može značajno skratiti vrijeme razvoja algoritma. Međutim, kao što će i kasnije biti vidljivo, u različitim stadijima razvoja softvera, različiti jezici su prikladni. Prema [14], razvoj softvera za upravljanje možemo ugrubo podijeliti u tri stadija:

1. Razvoj upravljačkog softvera u okruženju koje ne zahtijeva simulaciju procesa u realnom vremenu,

2. HIL (engl. *hardware-in-loop*) tehnika u kojem računalo simulira proces dok se upravljanje vrši na stvarnom kontroleru,

3. Testiranje i validacija upravljačkog algoritma na ugradbenom sustavu u stvarnom sustavu.

Primjenjujući gornje točke na naš slučaj, u kojem bi bilo potrebno projektirati upravljački algoritam za kvadrirotorsku letjelicu, možemo reći da bi se prva točka odnosila na razvoj upravljačkog algoritma na računalu koje bi ujedno i simuliralo

dinamiku letjelice. U drugoj točki dinamika procesa simuliralo bi računalo, dok bi upravljanje radio stvarni kontroler, dakle ugradbeno računalo. Na koncu, validacija kontrolnog algoritma odvijala bi se na realnoj kvadrirotorskoj letjelica kojom bi upravljalo ugradbeno računalo. Dakle, kao što je u uvodu rada navedeno, cilj ovog rada bio je razviti implementacija algoritma za rekonstrukciju orijentacije (stava) kvadrirotrske letjelice. Međutim, potrebno je ostvariti takvu implementaciju da bi se ona mogla simulirati u realnom vremenu na odabranom ugradbenom sustavu što bi onda služilo kao temelj algoritmu za upravljanje. Pitanje koje se nameće samo od sebe je zašto se dinamika sustava mora simulirati na ugradbenom računalu koje nedvojbeno ima puno manje računalnih resursi na raspolaganju. Ne ulazeći previše u problematiku teorije upravljanja, odgovor je da sama anatomija nekih upravljačkih algoritama zahtjeva simulaciju i procesa i algoritma upravljanja zajedno, dakle, nemoguće je odvojiti te dvije domene [14]. Radi već spomenutih ograničenih resursi na ugradbenim računalima koje je potrebno efikasno raspodijeliti, odabir odgovarajuće programske paradigme iznimno je bitan. Treba naglasiti da zahvaljujući kontinuiranom napretku procesora ugradbeni sustavi pronalaze široku primjenu, pa i onim područjima koja su do nedavno bila rezervirana samo za računala. Radi brojnih faktora (npr. cijena, praktičnost) ARM arhitektura postala je vodeća arhitektura za ugradbene sustave. ARM arhitektura omogućuju korištenje modernih operativnih sustava kao što je npr. *Linux*. ARM floating point arhitektura pruža hardversku podršku za računske operacije u dvostrukoj preciznosti. Najviše zbog te karakteristike, ugradbeni sustavi temeljeni na ARM arhitekturi pokazali su se kao idealan odabir za provođenje numerički intenzivnih simulacija. U našem slučaju, imamo diskretnu simulacija u kojoj vrijeme napreduje u jednakim vremenskim koracima. Da bi riješili matematičku funkciju i jednadžbu u određenom vremenskom koraku, svaka varijabla je riješena u funkciji prethodnog vremenskog koraka. Za vrijeme diskretne simulacije, količina realnog vremena potrebnog da se izračunaju sve jednadžbe i funkcije može biti duže od samog trajanja simulacije. Kada simulaciju provodimo 'off-line', vrijeme izvođenja najčešće nije od primarnog interesa. Kao dvije varijable koje najviše utječu na brzinu izvođenja simulacije možemo navesti računalne resurse na kojima se provodi simulacija te kompleksnost matematičkog modela. Za razliku od toga, simulacija u realnom vremenu mora izračunati i 'isporučiti' sve varijable u vremenu kraćem nego što bi proces koji se simulira trajao. U slučaju koji se obrađuje u ovom radu, važno je postići rezultat simulacije u jedinici vremena kraćoj od one koju se simulira kako bi se rezultati simulacije, npr. položaj, mogli poslati na ulazne/izlazne portove ugradbenog sustava, u našem slučaju Raspberry Pi. Kao zanimljivost također možemo napomenuti da je korištenje diskretnih rješavača inherentni zahtjev u nekim sustavima, kako bi se između diskretnih koraka omogućila izmjena podataka s vanjskim uređajem, npr I/O kanali, što se može

pokazati kao problematičan zahtjev ako je dinamika sustava koji se simulira visoko nelinearna [15].

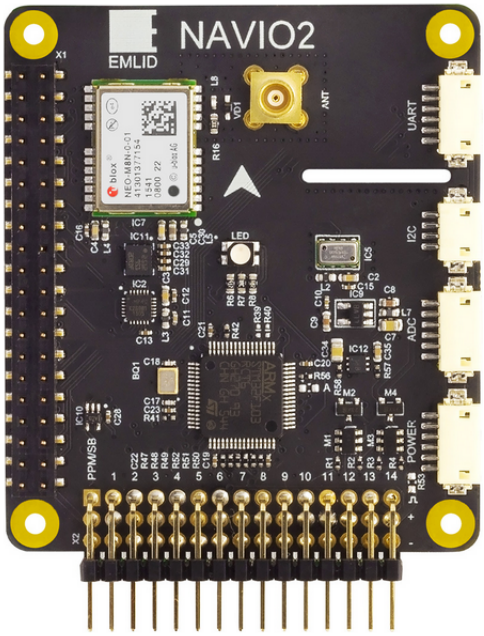
3.5 Rasperry Pi

Zahvaljujući kontinuiranoj inovaciji u području procesorske tehnologije, mogućnosti procesora povećavaju se prema eksponencijalnom zakonu. Takva evolucija dovela je do više kompleksnih jezgara koje pružaju bolje performanse na račun povećanja broja računalnih elemenata po mikroprocesoru. U ugradbenim sustavima, potrošnja energije je kritičan uvjet koji ograničava performanse procesora [16]. Da bi se zadovoljila rastuća potreba za performansama, uz minimalnu potrošnju energije, napredna višejezgrena arhitektura je razvijena.

Slika 4: Raspberry Pi B

ARM, poznati i kao Advanced RISC Machine, familija su arhitektura za kompjutorske procesore podešene za različita okruženja. Procesori s RISC arhitekturom tipično zahtijevaju manje tranzistora nego oni s CISC arhitekturom, kao što su to x86 procesori koje nalazimo u većini osobnih računala, što u konačnici smanjuje potrošnju energije. Radi toga, pogodni su za lagane, prijenosne i baterijom napajane uređaje poput pametnih telefona, osobnih računala te ugradbenih sustava. Ideja iza razvoja Raspberry Pi platforme, bila je razvoj male i cjenovno dostupne platforme za računanje koja bi bila služila poticanju interesa djece u području informacijsko komunikacijskih tehnologija (engl. *Information and Communications Technologies*

- *ICT*). Razvojem tzv. sustava-na-čipu (engl. *System on Chip - SoC*) koji kombiniraju različite elektroničke komponente na jedan integrirani čip, a koji su bili prvotno namijenjeni za mobilne uređaje, Raspberry Pi je postao široko dostupan. Novost je naime bila da se za nisku cijenu moglo dobiti kompletno računalo kojemu se hardver i softver mogao slobodno modificirati, dakle bio je otvorenog koda (engl. *open source*). Rezultati prodaje su bili izvan svih očekivanja. Naime, u svega 3 godine prodano je više od 5 milijuna Raspberry Pi uređaja. Raspberry Pi uređaji su sami po sebi previše kompleksni za opće pučanstvo, međutim, mogućnost da se na pločici 'vrti' Linux operativni sustav, čini platformu dostupnom i prilagodljivom. Takva platforma pokazala je da može odgovoriti na potrebe velikog broja područja, od pametnih građevina i interneta stvari (engl. *Internet of Things - IOT*) pa sve do robotike i automatskog upravljanja. Što se tiče hardvera, Raspberry Pi korišten za simulaciju raspolaže sljedećim komponentama: procesor Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz s 4 GB RAM memorije, Bluetooth modulima te 40 GPIO (engl. *General Purpose Input Output*) iglice za povezivanje s perifernim komponentama. Znamo da je namjera koristiti Raspberry Pi kao ugradbeno računalo u bespilotnoj letjelici, stoga ga gornje karakteristike čine idealnim za tu namjenu. Raspberry Pi, međutim, ne može služiti kao sustav za kontrolu leta (engl. *flight controller*). Modul NAVIO, kompatibilan s, Raspberry



Slika 5: Navio modul

Pi-em može služiti za tu namjenu. Druga je mogućnost korištenje Rasberry Pi-a s nekim kontrolerom, npr Pixhawk ili STM32F405 ARM Cortex-M4 kontrolerom.

3.6 Dron

Kao logičan nastavak ovog rada, nakon uspješne implementacije numeričkog algoritma na ugradbenom računalu, otvara se mogućnost izgradnje letjelice koja bi služila kao platforma za različite eksperimente. U ovom odlomku ćemo probati dati neke smjernice za buduću gradnju drona. Ako dron podijelimo na podsustave, možemo reći da se sastoji od okvira, motora, propelera, sustava za kontrolu leta (engl. *flight controller*). Ugradbeni sustav korišten za simulaciju, Raspberry Pi, ne može samostalno funkcionirati kao sustav za kontrolu leta. Preporuka je da se Raspberry Pi koristi kao pratitelj (engl. *onboard companion*) koji onda komunicira sa sustavom za kontrolu leta. Postoji upravo za tu namjenu razvijen modul (engl. *shield*) za Raspberry Pi naziva Navio2. Navio2 bi onda služio kao sustav za kontrolu leta, dok bi uspostava komunikacije između Raspberry Pi-a i Navio2 modula bila bi unaprijed definirana. U slučaju da kao sustav za kontrolu leta odabire neka druga nadogradnja, izbor tada nije jednoznačan. Preporučuje se odabir one pločice koja relativno jednostavno može uspostaviti komunikaciju s Raspberry Pi kontrolerom a to je opet uvjetovano da se na kontrolor leta može instalirati (engl. *flash*) s ArduPilot softver koji je otvorenog koda (engl. *open source*). Spomenuti kontroler putem MAVLink protokola komunicira s Raspberry Pi ugradbenim sustavom.

Slika 6: Naze32 kontrolor leta

Sada se možemo dotaknuti problema koji se potencijalno može dogoditi u letu drona. Naime, radi različitih razloga, Raspberry Pi se tijekom leta može ugasiti što za posljedicu može imati rušenje drona. Kako bi se premostio taj problem, u ArduPilot softveru su implementirane procedure predviđene za takve scenarije. U vremenu koje je potrebno da se Raspberry Pi ponovo pokrene, upravljanje drona u potpunosti preuzima kontroler leta, npr. Naze32. U tom slučaju dron najčešće prekida manevar ili neku drugu aktivnost kojom je rukovodio Raspberry Pi, te ga sustav za kontrolu leta drži u stanju mirovanja sve dok se ponovo ne uspostavi veza s Raspberry Pi-em što je znak da je on ponovo operativan te može nastaviti s prekinutom djelatnošću.

4 NUMERIČKI EKSPERIMENT

Nakon što smo pobliže opisali problematiku opisa rotacije te novopredloženu metodu integraciju, stavili smo to u kontekst simulacija u realnom vremenu na računalu s ograničenim računalnim resursima. U zadnjem poglavlju, nakon što damo karakteristike drona, usporedit ćemo rezultate dobivene u odabranim programskim paradigmama s relevantnim rezultatima [2] kako bismo validirali simulaciju. Nakon toga, dat ćemo usporedbu vremena izvedbe u različitim programskim paradigmama na različitim platformama.

4.1 Kvadrirotorska letjelica

Karakteristike letjelice, prema [2], prikazane su u tablici ispod:

Opis	Oznaka	Vrijednost
masa	m	0.5
matrica inercije	$\mathbf{J}$	$\begin{bmatrix} 0.00365 & 0 & 0 \\ 0 & 0.00368 & 0 \\ 0 & 0 & 0.00703 \end{bmatrix}$
duljina ruke	l	0.17
matrica inercije motor-rotor	J_{mzi}	0.0002271
pogonska sila	$\lvert F_{Ti}(\boldsymbol{\omega}_{mi})\rvert$	$5.57 \cdot 10^{-6} \cdot \boldsymbol{\omega}_{mi}^2$
moment parazitskog otpora	$\lvert T_{Di}(\boldsymbol{\omega}_{mi})\rvert$	$\begin{bmatrix} 0 & 0 & -1.37 \cdot 10^{-7} \cdot \lvert\omega_{mi}\rvert \cdot \omega_{mi} \end{bmatrix}^T$

Tablica 1: Karakteristike kvadrirotorske letjelice [2]

Za prijenos simulacijskih modela u stvarno vrijeme, za svaku od spomenutih programskih paradigmi, odabran je karakterističan jezik. Tako je Python odabran kao interpretirani jezik, dok je C primjer tipičnog kompajliranog jezika. Treba napomenuti da oba jezika podržava ARM arhitektura, koja čini bazu Raspberry Pi platforme, što je u konačnici i cilj. U obje paradigme je implementiran postupak integracije dinamičkog modela ugradbenog sustava s kvadrirotorskim propulzijskim pogonom. Novopredložena metoda, u kojoj se matematički model u trenutnoj točki integracije reducira u minimalnu dimenzijsku formu tangentnog vektorskog prostora, izomorfnog tangentnom vektorskom prostoru ambijentalne mnogostrukosti, trebala bi imati konkretne rezultate u vidu točnosti, neovisno o implementacijskoj paradigmi. Veća točnost numeričke integracije tijekom kinematičke rekonstrukcije

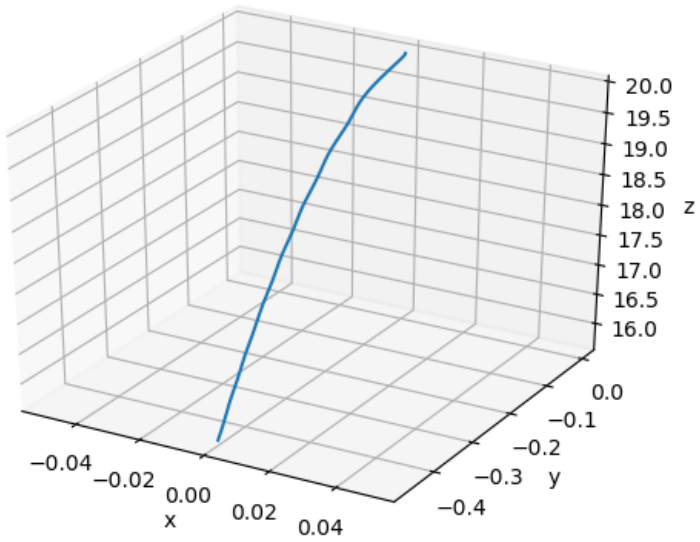
orijentacije (stava) objekta, dobra je naznaka kako za numeričku učinkovitost tako i za izvedbu u realnom vremenu. Prilikom implementacije, kao referentni kod uzimao se Matlab kod, na način da su implementacije u odabranim paradigmama postupno dorađivane i simulirane ('off-line') na računalu s većim računalnim resursima, sve dok ne bi bila postignuta zadovoljavajuća brzina izvedbe, nakon čega bi se pristupilo simulaciji na Raspberry Pi uređaju. Računalo s većim računalnim resursima je osobno računalo s Ubuntu distribucijom Linux operativnog sustava sa sljedećim specifikacijama:

- *Grafička kartica*: NVIDIA GeForce 840M
- *Procesor*: Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz
- *RAM*: 8GB,

dok ugradbeni sustav Raspberry Pi raspolaže s:

- *Procesor*: Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
- *RAM*: 8GB.

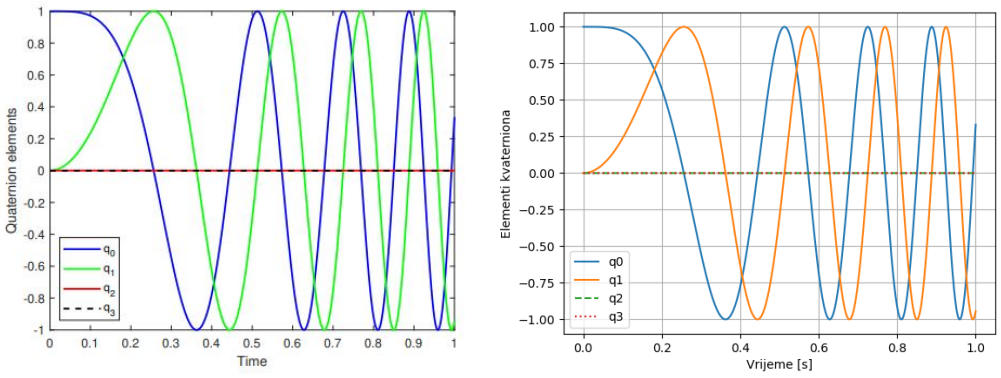
Da bi evaluacija i komparacija rezultata bila relevantna, potrebno je odrediti metriku kojom ćemo mjeriti pogreške u položaju i orijentaciji letjelice. Pogrešku u položaju računat će se kao Euklidska norma vektora razlike između točnog i postignutog položaja. Euklidska norma također je odabrana za evaluaciju orijentacije. Treba reći da postoje i druge metrike, međutim, odabir metrike invarijantan je na zaključke koji će biti izneseni. Ponovit ćemo da je kvadrirotorska letjelica u X konfiguraciji, s karakteristikama kao u tablici iznad, pri čemu duljina ruke odgovara horizontalnoj udaljenosti središta mase letjelice do osi motora dok matrica motor-rotor predstavlja inerciju cijelog sklopa koji rotira oko rotirajuće osi. Prvo ćemo prikazati validaciju implementacije u Pythonu te usporedbu s Matlab kodom. Isto ćemo ponoviti i za implementaciju u C-u, dok će na kraju biti dana međusobna komparacija rezultata. Za generiranje svih grafika, korištena je kombinacija knjižnica otvorenog koda, *Matplotlib* te *Pandas*, koji su često korišteni alati u domenama znanstvenog računanja ili podatkovne znanosti.
Simulirat će se manevar valjanja (engl. *roll maneuver*). U početku, letjelica se nalazi na visini od 20 metara nakon čega se rotori 1 i 2 ubrzavaju dok se 3 i 4 usporavaju, što odgovara manevru valjanja - rotaciji letjelice oko x osi. Trajektorija centra mase letjelice, prikazana je na slici ispod.



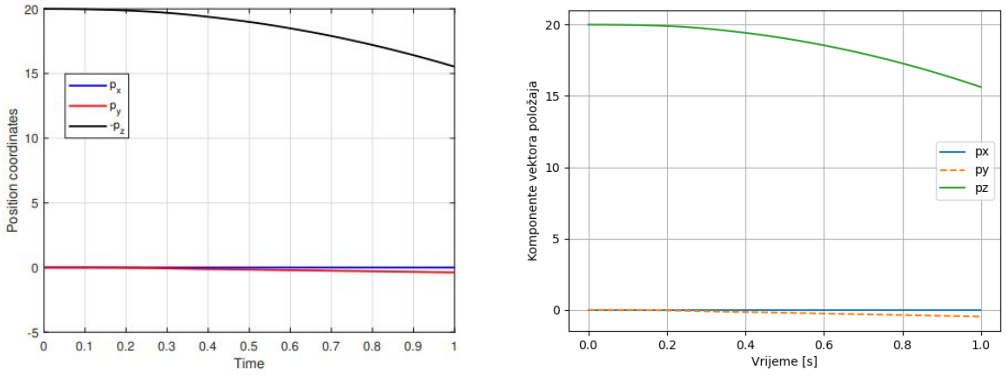
Slika 7: Trajektorija letjelice

4.2 Implementacija u Pythonu

Za validaciju rezultata, prvo ćemo prikazati ovisnost elemenata kvaterniona o vremenu, što je vidljivo na slici
Evolucija elementa kvaterniona u vremenu odgovara relevantnom rezultatu, prema [2], na temelju čega možemo zaključiti da je implementacija valjana. Možemo također pokazati i ovisnost komponenta vektora položaja o vremenu, što također mora odgovarati [2].

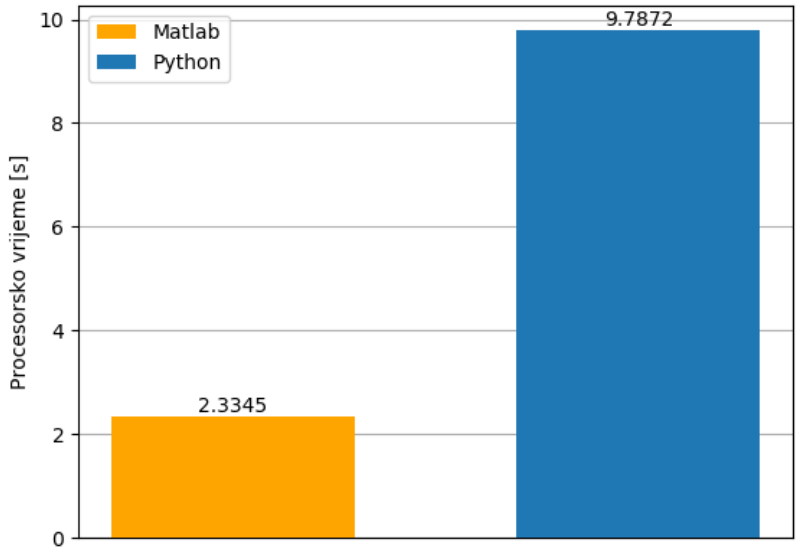


Slika 8: Usporedba elementa kvaterniona iz [2] i Python implementacije



Slika 9: Usporedba elementa vektora položaja iz [2] *i Python implementacije*

Novost metode, trebala bi se ogledati da, neovisno o implementacijskoj paradigmi te izboru koraka integracije, iznos Euklidske norme (koju koristimo kao metriku) bude konstantan. Usporedbom rezultata elemenata vektora položaja te elemenata kvaterniona, utvrđena je podudarnost rezultata do razine strojne točnosti, što možemo uzeti kao indikator valjanosti implementacije simulacije. Dijagrami usporedbe procesorskih vremena Matlab koda i novog Python koda odabranog simulacijskog modela koji se temelje na integracijskoj proceduri četvrtog reda točnosti za korak integracije $h = 1e - 4$ prikazan je na slici ispod. Simulacije su provedene 'off-line', dakle na računalu s većim računalnim resursima nego onima kojima raspolaže Raspberry Pi.



Slika 10: Usporedba procesorskih vremena izvornog Matlab i novog Python koda za korak integracije $h = 1e - 4$

Kao potencijalno rješenje za ubrzanje vremena izvođenja, predloženo je prethodno opisano prebacivanje eksponencijalne mape u oblik koji ne uključuje trigonometrijske funkcije, kako bi se izbjegli pozivi na vanjske knjižnice koje računaju te funkcije. U nastavku slijedi komparacija vremena izvođenja za implementaciju reda s pet članova te implementacije s trigonometrijskim funkcijama, pri čemu korak ostaje $h = 1e-4$. Radi preglednosti ćemo ponoviti izraze prema kojima smo implementirali eksponencijalnu mapu. Zatvorena forma eksponencijalne mape sadrži trigonometrijske funkcije:

$$\exp_{S^3}(\mathbf{w}) = \cos\left(\frac{1}{2}\left\|\mathbf{u}\right\|\right)(1,\mathbf{0}) + \frac{\sin(\frac{1}{2}\left\|\mathbf{u}\right\|)}{\left\|\mathbf{u}\right\|}(0,\mathbf{u}), \tag{145}$$

dok razvoj u red ne:

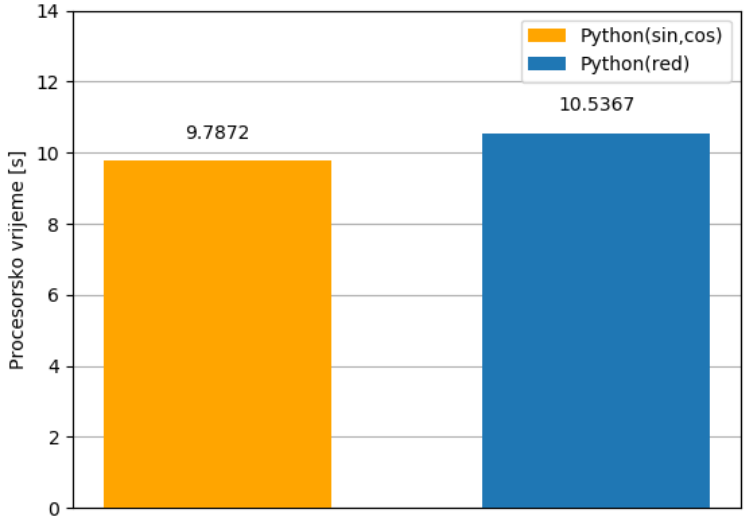
$$\exp_{S^3}(\mathbf{w}) = \sum_{k=0}^{\infty}\frac{\mathbf{w}^k}{k!} = \sum_{k=0}^{\infty}\left\{\frac{(0,\frac{1}{2}\mathbf{u})^{2k}}{(2k)!} + \frac{(0,\frac{1}{2}\mathbf{u})^{2k+1}}{(2k+1)!}\right\}. \tag{146}$$

Profiliranjem koda, dobivamo sljedeće podatke.

Implementacija	Broj poziva	Ukupno vrijeme	Kumulativno vrijeme
$expS3(sin, cos)$	39996	0.641s	1.333s
$expS3(red, 5clanova)$	39996	0.577s	1.103s

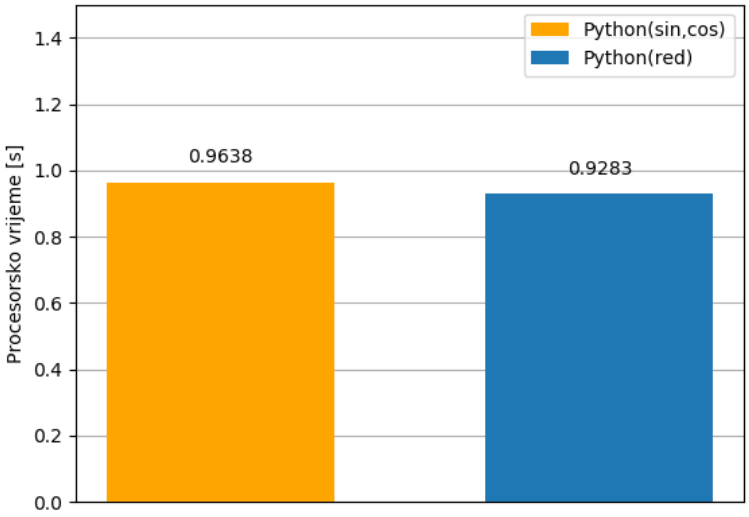
Tablica 2: Usporedba implementacija eksponencijalnih mapa

Drugi stupac prikazuje ukupan broj poziva funkcije, treći prikazuje udio vremena provedenog u funkciji, dok zadnji stupac prikazuje iznos kumulativnog vremena provedenog u toj te drugim podfunkcijama koje se pozivaju iz te funkcije.



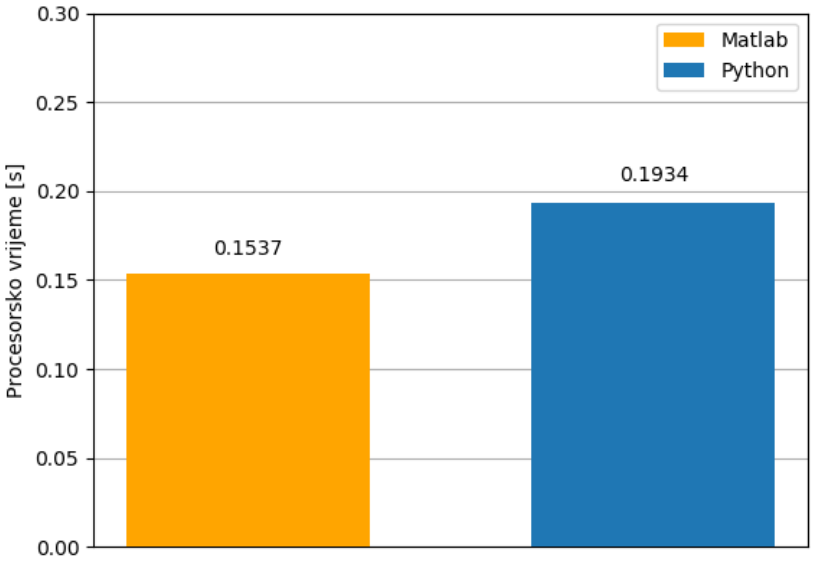
Slika 11: Usporedba procesorskih vremena različitih implementacija eksponencijalne mape za korak integracije $h = 1e - 4$

Razumljivo je da je iznos kumulativnog vremena za implementaciju sa sinusima i kosinusima veći, jer zahtjeva poziv na vanjsku biblioteku, međutim, vidimo da je razlika nije drastična, stoga bi se za buduće optimizacije, 'uska grla' u kodu trebala tražiti na drugim mjestima. Sada ćemo također usporediti implementacije, međutim, ovaj puta za grublji korak, dakle $h = 1e - 3$.



Slika 12: Usporedba procesorskih vremena različitih implementacija eksponencijalne mape za korak integracije $h = 1e - 3$

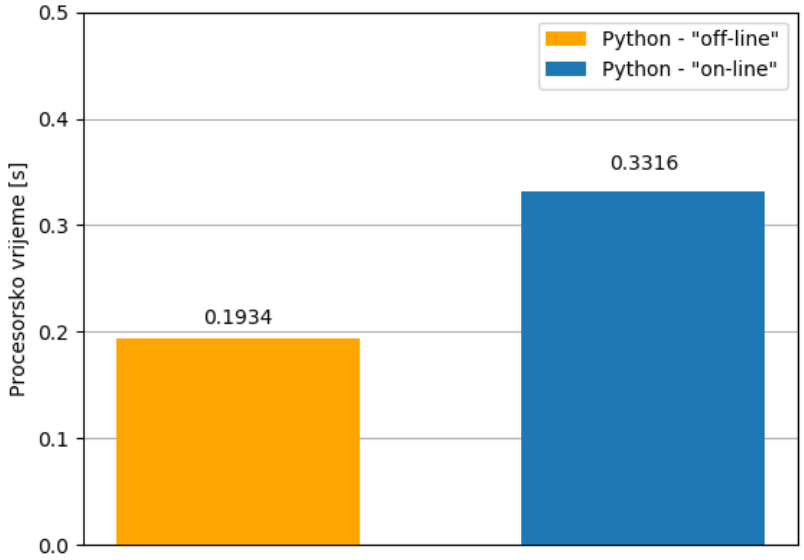
Vidljivo je također da je razlika između pojedinih implementacija eksponencijalne mape u razini greške mjerenja. Za razliku od prethodnog slučaja, za novi Python kod, ovdje postižemo izvedbu u stvarnom vremenu, međutim, nedovoljno da bi se postigao potreban faktor sigurnosti za osiguravanje izvedbe platformskih ('on-line') simulacija. Uzimajući grublji korak integracije, dobivamo sljedeće rezultate:



Slika 13: Usporedba procesorskih vremena izvornog Matlab i novog Python koda za korak integracije $h = 1e-2$

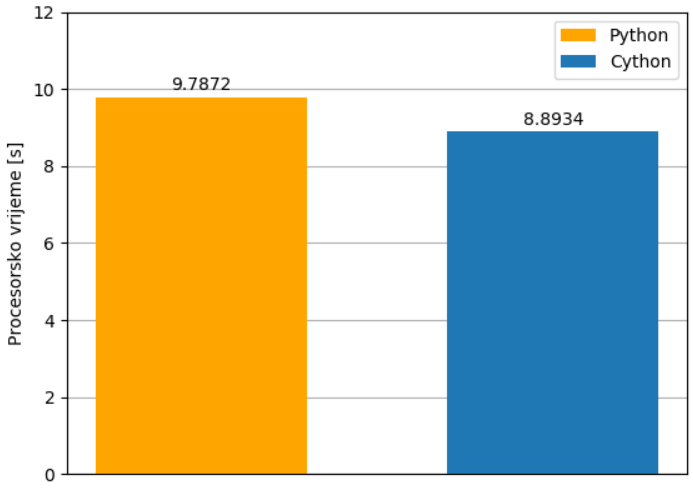
Iz gore priložene grafike, kao kandidat za platformsku simulaciju u stvarnom vremenu ističe se Python kod simulacijskog modela temeljenog na integracijskoj proceduri četvrtog reda točnosti i koraku integracije $h = 1e-2$. Novopredložena metoda omogućuje uzimanje u obzir većeg koraka integracije, bez narušavanja točnosti numeričke integracije tijekom kinematičke rekonstrukcije orijentacije (stava) letećeg objekta. Kako je pokazano, nova metoda rekonstruira stav letjelice iz izračunate funkcije kutne brzine točno za sve testirane vremenske korake, dok standardna metoda pokazuje uobičajenu konvergenciju prema točnom rješenju smanjivanjem vremenskog koraka integracije [2]. Pogreška koja nastaje kod klasične metode povećavanjem vremenskog koraka, nastaje zbog krive kinematičke rekonstrukcije stava letjelice iz polja vektorskih brzina - što je linearizacija inherentno nelinearnog problema kakav je trodimenzionalna rotacija.

Na temelju numeričkih pokusa, provedenih 'on-line' i 'off-line', može se reći da sve verzije odabranog simulacijskog modela, imaju tri do četri puta duže vrijeme izvođenja na odabranom ARM ugradbenom računalu Raspberry Pi Model B. Na temelju tih rezultata, Python kod je idealan za rapidnu prototipizaciju koda, među-



Slika 14: Usporedba procesorskih vremena optimiziranog Python koda između usporedne ('off-line') i platformske ('on-line') simulacije za korak integracije $h = 1e - 2$

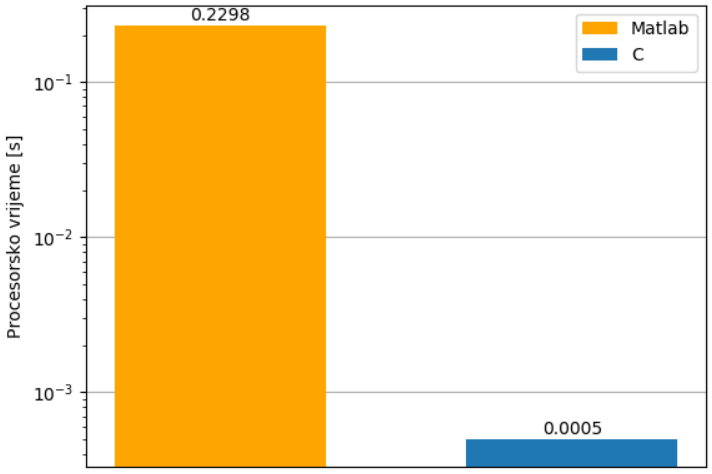
tim, zbog inherentnih karakteristika interpretiranih jezika, postoje ograničenja u vidu njegove upotrebljivosti u simulacijama u realnom vremenu, što posebno vrijedi za njihovu izvedbu na ugradbenim računalima s ograničenim računalnim resursima. Možemo također usporediti vrijeme izvođenja Python koda i Cythona koji je bio spomenut u radu. Sa slike 15 vidimo da poboljšanje postoji, međutim, nije značajno. To je i očekivani rezultat, jer se od programera očekuje da ručno poboljša Python kod namjenjen za prevođenje uz pomoć Cythona. Dakle, iako Cython nudi stanovit potencijal za ubrzanje vremena izvođenja Python koda, potreban je dodatni angažman programera kako bi iskoristio poluge koje mu Cython pruža.



Slika 15: Usporedba procesorskih vremena izvornog Python i Cython koda za korak integracije $h = 1e - 4$

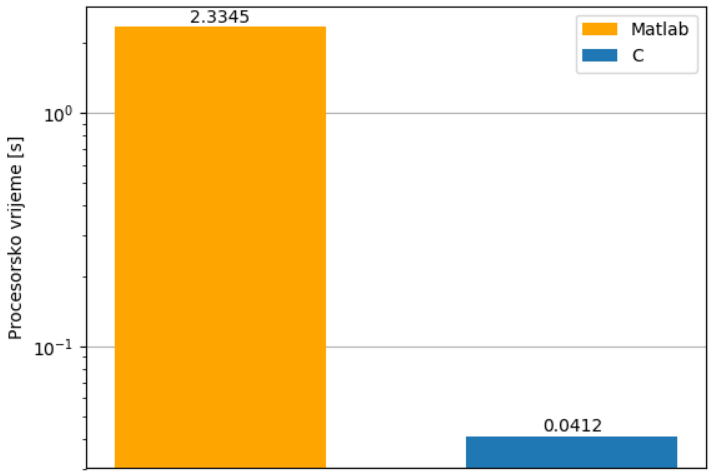
4.3 Implementacija u C-u

Za programiranje u C-u, na Linux distribuciji korištena je integrirana razvojna okolina (engl. *Integrated Development Environment - IDE*) naziva Visual Studio Code. Kao kompajler, korišten je GNU Compiler Collection (GNU), koji dolazi kao sastavni dio svakog Linux operativnog sustava, čime su riješeni potencijalni problemi prilikom prebacivanja koda s jedna platfome na drugu. Naime, i na Raspberry Pi-u je također instaliran Linux operativni sustav sa Raspbian distribucijom, koji dolazi s GCC kompajlerom.



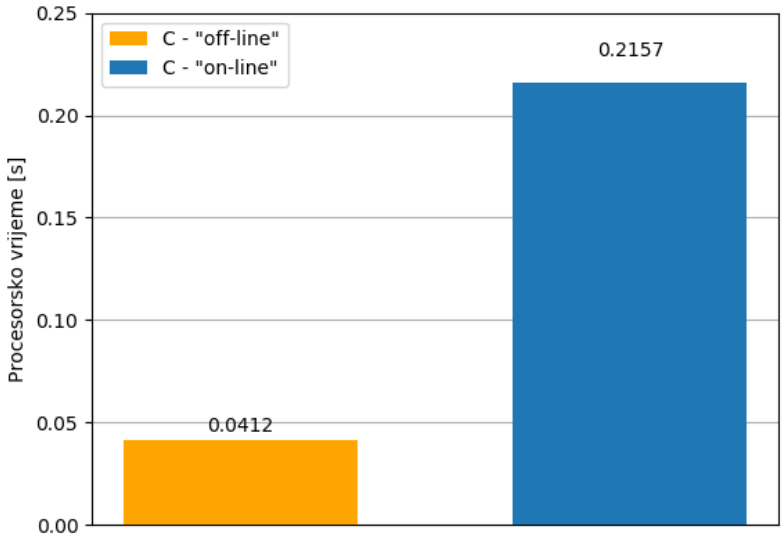
Slika 16: Usporedba procesorskih vremena izvornog Matlab i novog C koda za korak integracije $h = 1e - 3$

Kako nam je Matlab kod relevantan, prvo dajemo usporedbu njega s implementacijom u C-u za korak integracije $h = 1e-3$, pri čemu je simulacija provedena 'off-line'. Radi preglednosti, nećemo ponavljati dijagrame validacije modela za implementaciju u C-u . Odmah do izražaja dolazi iznimna efikasnost kompajliranih jezika. Već za korak integracije $h = 1e - 3$ vidimo da je zadovoljen faktor sigurnosti platformskih simulacija. Napominjemo da radi iznimne brzine, implementacija eksponencijalne mape kao reda, za razliku od slučaja s Pythonom, nije bila potrebna.



Slika 17: Usporedba procesorskih vremena izvornog Matlab i novog C koda za korak integracije $h = 1e - 4$

Povećanje vremena izvođenja je primjetno, međutim, i dalje je faktor sigurnosti za osiguravanje platformskih izvedbi zadovoljavajuć. Radi takvih rezultata, kao kandidata za platformsku simulacija u stvarnom vremenu uzimamo C kod simulacijskog modela temeljenog na integracijskoj proceduri četvrtog reda točnosti s korakom integracije $h = 1e - 4$. Sada ćemo dati usporedbu 'on-line' i 'off-line' vremena za $1e - 4$ korak integracije.

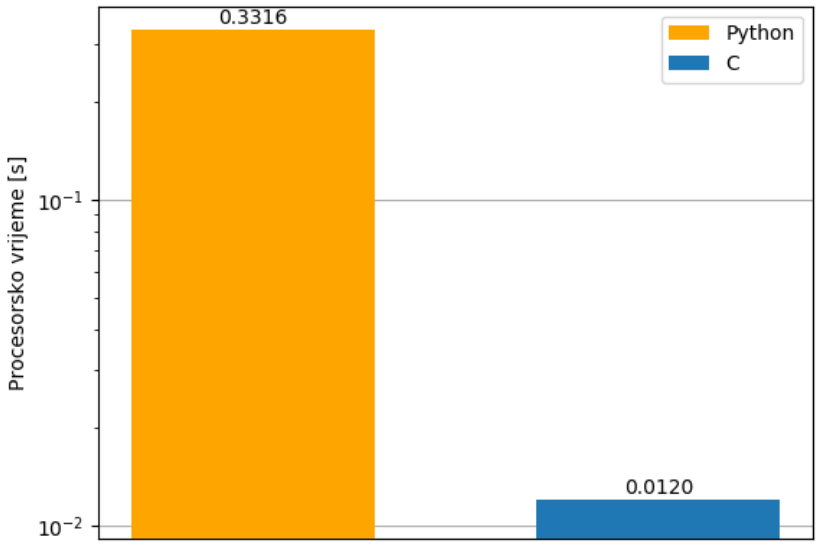


Slika 18: Usporedba procesorskih vremena optimiziranog C koda između usporedne ('off-line') i platformske ('on-line') simulacije za korak integracije $h = 1e - 4$

S obzirom na dostupne računalne resurse, za očekivati je bilo da će za *'on-line'* platformsku simulaciju vrijeme izvođenja biti duže. Vidljivo je međutim da unatoč vrlo malom koraku integracije, faktor sigurnosti i dalje ostaje zadovoljavajući. Na temelju tih rezultata, možemo zaključiti da su spomenute prednosti kompajliranih jezika u vidu brzine izvođenja doista i došle do izražaja, čak i u slučaju simulacije na računalu sa smanjenim resursima kakvo je Raspberry Pi.

4.4 Usporedba 'online' implementacija

Prikazat ćemo usporedbu platformskih ('on-line') simulacija za interpretiranu i kompajlirnu paradigmu, i to za korak integracije $h = 1e - 2$.



Slika 19: Usporedba procesorskih vremena C i Python koda za platformsku ('on-line') simulacije s korakom integracije $h = 1e - 2$

Iako obje paradigme zadovoljavaju faktor sigurnosti, opet do izražaja dolazi brzina kompajlirane paradigme. Iz priloženih rezultata, potvrđujemo pretpostavke do kojih smo došli u prethodnom poglavlju. Iz priloženih se rezultata jasno može vidjeti brzina i efikasnost kompajliranog jezika C, ali usporedbom priloženih isječaka koda, jasno je da intuitivnija i 'prirodnija' sintaksa Pythona, može biti prikladna za druge namjene. U kontekstu simulacija u realnom vremenu na računalima s ograničenim resursima, jasno je da su u prednosti kompajlirani jezici. Simulacijski model s novom numeričkom metodom vremenske integracije jediničnih kvaterniona, primijenjen na dinamičku analizu ugradbenog sustava s kvadri- rotorskim propulzijskim pogonom, prenesen u Python odabran kao interpretirani jezik i C kao primjer tipičnog kompajliranog jezika, prije same izgradnje višerotorske letjelice opremljene ARM ugradbenim računalom Raspberry Pi (što nije bila zadaća ovog rada), omogućit će provedbu nekoliko platformskih ('on-line') simulacija na Raspberry Pi-u, na način da se isprogramiraju lažni ulazni i izlani ('input/output') signali koji će predstavljati signale od senzorike i motora. Na taj će se način dobiti informacija kako najbolje konstruirati, a zatim i izgraditi višerotorsku letjelicu na bazi Raspberry Pi-a, kao i koje složene manevre koristiti u fazi testiranja izgrađene letjelice.

5 ZAKLJUČAK

U prvom su poglavlju opisane linearne transformacije te svojstva takvih transformacija. Nakon toga je opisan skup matrica rotacija $SO(3)$ za koji je pokazano da zadovoljava aksiome grupe posjedujući time matematičku strukturu grupe. Slijedi opis kvaternionskog sustava brojeva čija inherentna svojstva odgovaraju svojstvima rotacije čineći ga time prikladnim za opis trodimenzionalnih rotacija. Nakon definicije jediničnih vektora kao elemenata grupe S^3 pokazana je veza između rotacije na jediničnoj sferi te pravih rotacija u trodimenzionalnom prostoru što se može sažeti na način da grupa jediničnih kvaterniona ima dvostruko pokrivanje grupe $SO(3)$. Na tim temeljima uvedeni su i razrađeni Eulerovi parametri koji inherentno poštujući jediničnu normu kvaterniona, zaobilaze poznati problem singularnosti u različitim parametrizacijama rotacije, a i pokazali su se prikladnima za računalnu implementaciju. Poglavlje završava izvodom jednadžbe kinematičke rekonstrukcije. Drugo poglavlje počinje uvođenjem pojma Lieve grupe i Lieve algebre karakterističnih za kontinuirane grupe. Nakon toga su ukratko dani obrisi novo predložene metode integracije gdje se problem integracije podiže na tangentni prostor rotacijske mnogostrukosti dok se rekonstrukcija orijentacije vrši izravno na mnogostrukosti jediničnih kvaterniona. Zatim je dan dinamički model drona koji uključuje jednadžbe kojima je opisano gibanje drona, a koje je potrebno riješiti kako bi iz polja akceleracija odredilo polje brzina. Na kraju drugog poglavlja dani su detalji za implementaciju nekih karakterističnih funkcija nove metode poput eksponencijalne mape ili Runge Kutta metode integracije.

Treće poglavlje sadrži sažet opis i komparaciju različitih programskih paradigmi u kontekstu znanstvenog računanja i numeričkih simulacija. Ukratko su opisani karakteristični predstavnici relevantnih paradigmi za koje je i ostvarena implementacija. Na kraju je dan kratak opis Raspberry Pi ugradbenog računala na kojem su provedene 'on-line' simulacije te perspektiva izgradnje drona koji bi služio kao platforma za empirijsku provjeru rezultata.

Zadnje poglavlje sadrži fizikalne karakteristike drona te verifikaciju dobivenih rezultata u usporedbi s relevantnim rezultatima. Nakon toga je dani čitav niz komparacija 'on-line' i 'off-line' simulacija s različitim koracima i u različitim programskim paradigmama. Potvrdili smo pretpostavku da se interpretirani jezici su se radi svoje fleksibilnosti pokazali prihvatljivijima za početak implementacije. Međutim, očekivana razlika u vremenu izvođenju koda između kompajliranog i interpretiranog jezika došla je do izražaja te se za interpretirani jezik kao jedina moguća simulacija u realnom vremenu na ugradbenom računalu pokazala ona s vremenskim korakom

$h = 1e - 2$. Nasuprot tome, kompajlirani jezik C, pokazao se kao izniman u vremenu izvođenja, neovisno o vremenskom koraku. Tako je potreban faktor sigurnosti zadovoljen čak i za najfiniji vremenski korak na računalu sa smanjenim računalnim resursima. Pritom, međutim, treba imati na umu da je programiranje u C-u zahtjevnije te traži dublje poznavanje programskih metodologija i načina pisanja koda. Također smo pokazali da točnost novopredloženog algoritma ne ovisi o vremenskom koraku, što znači da bi implementacija s najgrubljim korakom u interpretiranom jeziku mogla dobro funkcionirati.

LITERATURA

[1] D. Zlatar, *Geometrijski integratori za numeričku simulaciju dinamike letjelica*. PhD thesis, Sveučilište u Zagrebu, Fakultet strojarstva i brodogradnje, 2015.

[2] Z. Terze, D. Zlatar, V. Pandža, and A. Müller, "Quaternion attitude reconstruction of a quadrotor aerial vehicle," 2014.

[3] R. M. Murray, Z. Li, S. S. Sastry, and S. S. Sastry, *A mathematical introduction to robotic manipulation*. CRC press, 1994.

[4] J. Stillwell, *Naive lie theory*. Springer Science & Business Media, 2008.

[5] *Visualizing quaternions (4d numbers) with stereographic projection*. https://www.youtube.com/watch?v=d4EgbgTm0Bg&t=19s.

[6] *Letters describing the Discovery of Quaternions*. https://www.maths.tcd.ie/pub/HistMath/People/Hamilton/Letters/BroomeBridge.html.

[7] M. Ban, "Kinematička rekonstrukcija rotacije satelita," 2018.

[8] Z. Terze, A. Müller, and D. Zlatar, "Lie-group integration method for constrained multibody systems in state space," *Multibody System Dynamics*, vol. 34, no. 3, pp. 275–305, 2015.

[9] Z. Terze, A. Müller, and D. Zlatar, "Singularity-free time integration of rotational quaternions using non-redundant ordinary differential equations," *Multibody system dynamics*, vol. 38, no. 3, pp. 201–225, 2016.

[10] Z. Terze, D. Zlatar, and V. Pandža, "Aircraft attitude reconstruction via novel quaternion-integration procedure," *Aerospace Science and Technology*, vol. 97, p. 105617, 2020.

[11] R. Johansson, *Numerical Python: Scientific Computing and Data Science Applications with Numpy, SciPy and Matplotlib*. USA: Apress, 2nd ed., 2018.

[12] B. W. Kernighan and D. M. Ritchie, *The C programming language*. 2006.

[13] K. W. Smith, *Cython: A Guide for Python Programmers*. 2015.

[14] R. Pastorino, F. Cosco, F. Naets, W. Desmet, and J. Cuadrado, "Hard real-time multibody simulations using arm-based embedded systems," *Multibody System Dynamics*, vol. 37, no. 1, pp. 127–143, 2016.

[15] J. Bélanger, P. Venne, and J.-N. Paquin, "The what, where, and why of real-time simulation," *Planet RT*, pp. 37–49, 01 2010.

[16] A. J. Rodríguez, R. Pastorino, Á. Carro-Lagoa, K. Janssens, and M. Á. Naya, "Hardware acceleration of multibody simulations for real-time embedded applications," *Multibody System Dynamics*, pp. 1–19, 2020.

PRILOG

prototypes.h

```
struct Motor
{

  double k_d;
  double k_t;
  double rot_dir[4][1];
  double inertia;
};


struct Aircraft
{

  double mass;
  double J[3][3];
  double lev_arm;
  double matrix_FT[3][4];
};



//dexp
void dexp (double omega[3][1], double u[3][1], double du[3][1]);

void drag_torque (struct Motor SpecificMotor, double rpm[4][1],
                  double T_d[3][1]);

void exp_S3 (double u[3][1], double u_S3[4][1]);

void lie_bracket (double skew_x[][3], double skew_y[][3],
                  double skew_out[][3]);

void new_eul (struct Motor SpecificMotor, struct Aircraft SpecificAircraft,
```

```
            double F_t[4][1], double T_t[3][1], double T_d[3][1],
            double omega[3][1], double omegam[4][1], double q[4][1],
            double dv[3][1], double domega[3][1]);

void om2rpm (double omega[4][1], double rpm_out[4][1]);

void omegamfun (char type[], double omegam_out[4][1]);

void quat_rot (double q[4][1], double v[3][1], double v_out[3][1]);

void quat_vect (double q[4][1], double v[3][1], double q_out[4][1]);

void skew (double x[3][1], double x_skew[3][3]);

void SO3_to_R3 (double x_skew[3][3], double x[3][1]);

void thrust_forces (struct Motor SpecificMotor, double rpm[4][1],
                    double F_t[4][1]);

void thrust_torque (struct Motor SpecificMotor,
                    struct Aircraft SpecificAircraft, double rpm[4][1],
                    double T_t[4][1]);


void rk2cl (double t, double h, struct Motor SpecificMotor,
            struct Aircraft SpecificAircraft, double omega[3][1],
            double v[3][1], double q[4][1], double p[3][1], char type[],
            double omega_out[3][1], double v_out[3][1], double q_out[4][1],
            double p_out[3][1]);


void rk2geom (double t, double h, struct Motor SpecificMotor,
              struct Aircraft SpecificAircraft, double omega[3][1],
              double v[3][1], double q[4][1], double p[3][1], char type[],
              double omega_out[3][1], double v_out[3][1], double q_out[4][1],
              double p_out[3][1]);

void rk4cl (double t, double h, struct Motor SpecificMotor,
            struct Aircraft SpecificAircraft, double omega[3][1],
```

```
            double v[3][1], double q[4][1], double p[3][1], char type[],
            double omega_out[3][1], double v_out[3][1], double q_out[4][1],
            double p_out[3][1]);


void rk4geom (double t, double h, struct Motor SpecificMotor,
              struct Aircraft SpecificAircraft, double omega[3][1],
              double v[3][1], double q[4][1], double p[3][1], char type[],
              double omega_out[3][1], double v_out[3][1], double q_out[4][1],
              double p_out[3][1]);
```

new eul.c

```
#include <stdio.h>
#include <math.h>


void
new_eul (struct Motor SpecificMotor, struct Aircraft SpecificAircraft,
         double F_t[4][1], double T_t[3][1], double T_d[3][1],
         double omega[3][1], double omegam[4][1], double q[4][1],
         double dv[3][1], double domega[3][1])
{


//initialize function local omega
  double localOmega[3][1] = { 0 };
  localOmega[0][0] = omega[0][0];
  localOmega[1][0] = omega[1][0];
  localOmega[2][0] = omega[2][0];


  double R[3][3] = { 0 };
  R[0][0] = 2 * (pow (q[0][0], 2) + pow (q[1][0], 2) - 0.5);
  R[0][1] = 2 * (q[1][0] * q[2][0] - q[0][0] * q[3][0]);
  R[0][2] = 2 * (q[1][0] * q[3][0] + q[0][0] * q[2][0]);
  R[1][0] = 2 * (q[1][0] * q[2][0] + q[0][0] * q[3][0]);
  R[1][1] = 2 * (pow (q[0][0], 2) + pow (q[2][0], 2) - 0.5);
  R[1][2] = 2 * (q[2][0] * q[3][0] - q[0][0] * q[1][0]);
  R[2][0] = 2 * (q[1][0] * q[3][0] - q[0][0] * q[2][0]);
  R[2][1] = 2 * (q[2][0] * q[3][0] + q[0][0] * q[1][0]);
  R[2][2] = 2 * (pow (q[0][0], 2) + pow (q[3][0], 2) - 0.5);

  double sumOfFt = 0;
  for (int i = 0; i < 4; i++)
    {
      for (int j = 0; j < 1; j++)
        {
          sumOfFt = sumOfFt + F_t[i][j];
        }
    }
```

```
  double vector[3][1] = { 0 };
  vector[2][0] = sumOfFt;
//now mulitply R[3x3] and vector[3x1]
  double sum = 0;
  double RvVector[3][1] = { 0 };
  for (int i = 0; i < 3; i++)
    {                                   //first matrix first dim
      for (int j = 0; j < 1; j++)
        {                           //second matrix second dim
          for (int k = 0; k < 3; k++)
            {                           //first matrix second dim
              sum = sum + R[i][k] * vector[k][j];
            }
          RvVector[i][j] = sum;
          sum = 0;
        }
    }

//initalize new vector to be added to RvVector
  double toAddVector[3][1] = { 0 };
  toAddVector[2][0] = SpecificAircraft.mass * 9.80665;
//initialize dv and add RvVector and toAddVector
//double dv[3][1] = {0};
  for (int i = 0; i < 3; i++)
    {
      for (int j = 0; j < 1; j++)
        {
          dv[i][j] =
            (1. / SpecificAircraft.mass) * (RvVector[i][j] +
                                            toAddVector[i][j]);
        }
    }


//first calc dVskew(omega)
  double skew_out[3][3] = { 0 };
  skew (localOmega, skew_out);
//now multiply skew_out[3][3] with aircraft.J [3x3]
```

```
  double firstResultMatrix[3][3] = { 0 };        //intermediate vector

  double sum2 = 0;
  for (int i = 0; i < 3; i++)
    {                                  //first matrix first dim
      for (int j = 0; j < 3; j++)
        {                           //second matrix second dim
          for (int k = 0; k < 3; k++)
            {                           //first matrix second dim
              sum2 = sum2 + skew_out[i][k] * SpecificAircraft.J[k][j];
            }
          firstResultMatrix[i][j] = sum2;
          sum2 = 0;
        }
    }

//now multiply firstResultMatrix[3x3] with omega[3x1]

  double sum3 = 0;
  double secondTermVector[3][1] = { 0 };
  for (int i = 0; i < 3; i++)
    {                                  //first matrix first dim
      for (int j = 0; j < 1; j++)
        {                           //second matrix second dim
          for (int k = 0; k < 3; k++)
            {                           //first matrix second dim
              sum3 = sum3 + firstResultMatrix[i][k] * localOmega[k][j];
            }
          secondTermVector[i][j] = sum3;
          sum3 = 0;
        }
    }


///////third term in expression for domega
//this is piecewise multip
  double vectorPieceWise[4][1] = { 0 };
```

```
  for (int i = 0; i < 4; i++)
    {
      for (int j = 0; j < 1; j++)
        {
          vectorPieceWise[i][j] = omegam[i][j] * SpecificMotor.rot_dir[i][j];
        }
    }

//now sum all elements of matrixPieceWise
  double sumVectorPieceWise = 0;
  for (int i = 0; i < 4; i++)
    {
      for (int j = 0; j < 1; j++)
        {
          sumVectorPieceWise = sumVectorPieceWise + vectorPieceWise[i][j];
        }
    }
//put motorInertia * matrixPieceWise to last place in column
  double lastVector[3][1] = { 0 };
  lastVector[2][0] = SpecificMotor.inertia * sumVectorPieceWise;

//multiply skew(omega) * lastVectr
//skew out [3x3]
  double thirdTermVector[3][1] = { 0 };
  double sum4 = 0;
  for (int i = 0; i < 3; i++)
    {                                   //first matrix first dim
      for (int j = 0; j < 1; j++)
        {                           //second matrix second dim
          for (int k = 0; k < 3; k++)
            {                           //first matrix second dim
              sum4 = sum4 + skew_out[i][k] * lastVector[k][j];
            }
          thirdTermVector[i][j] = sum4;
          sum4 = 0;
        }
    }
//now calculate variable for linear system dimension is [3x1]
```

```
  double variable[3][1] = { 0 };
  for (int i = 0; i < 3; i++)
    {
      for (int j = 0; j < 1; j++)
        {
          variable[i][j] =
            T_t[i][j] + T_d[i][j] - secondTermVector[i][j] -
            thirdTermVector[i][j];
        }
    }


////now solve system Ax = b - LU Decomposition
//A = AircraftJ - diag mattrix
//b = variable -sum of upper



  double L[3][3] = { 0 };
  double U[3][3] = { 0 };
  double B[3] = { variable[0][0], variable[1][0], variable[2][0] };
  double X[3] = { 0 };
  double Y[3] = { 0 };
  int i, j, k;
  int n = 3;

  for (j = 0; j < n; j++)
    {
      for (i = 0; i < n; i++)
        {
          if (i <= j)
            {
              U[i][j] = SpecificAircraft.J[i][j];
              for (k = 0; k < i - 1; k++)
                U[i][j] -= L[i][k] * U[k][j];
              if (i == j)
                L[i][j] = 1;
              else
                L[i][j] = 0;
```

```
            }
          else
            {
              L[i][j] = SpecificAircraft.J[i][j];
              for (k = 0; k <= j - 1; k++)
                L[i][j] -= L[i][k] * U[k][j];
              L[i][j] /= U[j][j];
              U[i][j] = 0;
            }
        }
    }

  for (i = 0; i < n; i++)
    {
      Y[i] = B[i];
      for (j = 0; j < i; j++)
        {
          Y[i] -= L[i][j] * Y[j];
        }
    }

  for (i = n - 1; i >= 0; i--)
    {
      X[i] = Y[i];
      for (j = i + 1; j < n; j++)
        {
          X[i] -= U[i][j] * X[j];
        }
      X[i] /= U[i][i];
    }

  domega[0][0] = X[0];
  domega[1][0] = X[1];
  domega[2][0] = X[2];



}
```

dexp.c

```
void
dexp (double omega[3][1], double u[3][1], double du[3][1])
{

  double norm_u = 0;
  double skew_u[3][3] = { 0 };
  double SO3_matrix[3][3] = { 0 };
  norm_u = sqrt (pow (u[0][0], 2) + pow (u[1][0], 2) + pow (u[2][0], 2));


  skew (u, skew_u);
  skew (omega, SO3_matrix);
  SO3_to_R3 (SO3_matrix, du);

}
```

exp-S3sin/cos.c

```
void exp_S3(double u[3][1], double u_S3[4][1]){

double norm_u = 0;
double unitQuaternion [4][1] = {{1},{0},{0},{0}};


//norm of vector u
norm_u = sqrt(pow(u[0][0],2)+pow(u[1][0],2)+pow(u[2][0],2));
double secondTerm = 1./norm_u * sinf(0.5*norm_u);

//sin-cos implementation
u_S3[0][0] = cosf(0.5*norm_u);
u_S3[1][0] = secondTerm * u[0][0];
u_S3[2][0] = secondTerm * u[1][0];
u_S3[3][0] = secondTerm * u[2][0];


}
```

rk4geom.c

```
void
rk4geom (double t, double h, struct Motor SpecificMotor,
         struct Aircraft SpecificAircraft, double omega[3][1], double v[3][1],
         double q[4][1], double p[3][1], char type[], double omega_out[3][1],
         double v_out[3][1], double q_out[4][1], double p_out[3][1])
{

  double L[3][4] = { 0 };
  L[0][0] = -q[1][0];
  L[0][1] = q[0][0];
  L[0][2] = q[3][0];
  L[0][3] = -q[2][0];
  L[1][0] = -q[2][0];
  L[1][1] = -q[3][0];
  L[1][2] = q[0][0];
  L[1][3] = q[1][0];
  L[2][0] = -q[3][0];
  L[2][1] = q[2][0];
  L[2][2] = -q[1][0];
  L[2][3] = q[0][0];

  double appendedMatrix[4][4] = { 0 };
  appendedMatrix[0][0] = q[0][0];
  appendedMatrix[1][0] = q[1][0];
  appendedMatrix[2][0] = q[2][0];
  appendedMatrix[3][0] = q[3][0];

  appendedMatrix[0][1] = L[0][0];
  appendedMatrix[1][1] = L[0][1];
  appendedMatrix[2][1] = L[0][2];
  appendedMatrix[3][1] = L[0][3];

  appendedMatrix[0][2] = L[1][0];
  appendedMatrix[1][2] = L[1][1];
  appendedMatrix[2][2] = L[1][2];
  appendedMatrix[3][2] = L[1][3];

  appendedMatrix[0][3] = L[2][0];
```

```
  appendedMatrix[1][3] = L[2][1];
  appendedMatrix[2][3] = L[2][2];
  appendedMatrix[3][3] = L[2][3];

  double omegam[4][1] = { 0 };
  double rpm_out[4][1] = { 0 };
  omegamfun (type, omegam);         //changes omegam
  om2rpm (omegam, rpm_out);         //rpm out is changed

  double F_t[4][1] = { 0 };
  double T_t[3][1] = { 0 };
  double T_d[3][1] = { 0 };
  thrust_forces (SpecificMotor, rpm_out, F_t);
  thrust_torque (SpecificMotor, SpecificAircraft, rpm_out, T_t);
  drag_torque (SpecificMotor, rpm_out, T_d);

  double vk1[3][1] = { 0 };
  double k1[3][1] = { 0 };

  new_eul (SpecificMotor, SpecificAircraft, F_t, T_t, T_d, omega, omegam, q,
           vk1, k1);


  double u[3][1] = { 0 };
  double uk1[3][1] = { 0 };
  dexp (omega, u, uk1);

  double pk1[3][1] = { {v[0][0]}, {v[1][0]}, {v[2][0]} };

////////////////////////////////////////////////////////////


  omegamfun (type, omegam);         //t argument should be t + 0.5*h
  double vk2[3][1] = { 0 };
  double k2[3][1] = { 0 };
//now new eul - omega argument is omega + h * k1
  double newOmega[3][1] = { 0 };
  newOmega[0][0] = omega[0][0] + 0.5 * h * k1[0][0];
  newOmega[1][0] = omega[1][0] + 0.5 * h * k1[1][0];
```

```
  newOmega[2][0] = omega[2][0] + 0.5 * h * k1[2][0];

//new quaternion argument for omega is appendmatrix[4x4] * exp_S3(0.5 h*uk1)

//define input[3x1] and output[4x1] variable for exp_S3

  double inputExp[3][1] =
    { {0.5 * h * uk1[0][0]}, {0.5 * h * uk1[1][0]}, {0.5 * h * uk1[2][0]} };
  double outputExp[4][1] = { 0 };
  exp_S3 (inputExp, outputExp);
//now multiply appendmatrix[4x4] * outputExp[4][1] - out is newq[4x1]

  double newq[4][1] = { 0 };        //input from multip below
//now multipy
  for (int i = 0; i < 4; i++)
    {                                //first matrix first dim
      for (int j = 0; j < 1; j++)
        {                          //second matrix second dim
          for (int k = 0; k < 4; k++)
            {                          //first matrix second dim
              newq[i][j] =
                newq[i][j] + appendedMatrix[i][k] * outputExp[k][j];
            }
        }
    }

//now we have everything for new_eul
  new_eul (SpecificMotor, SpecificAircraft, F_t, T_t, T_d, newOmega, omegam,
           newq, vk2, k2);

//dexp(omegaForDexp[3x1],uForDexp[3x1],du[3,1]) example
  double omegaForDexp[3][1] = { 0 };
  omegaForDexp[0][0] = omega[0][0] + 0.5 * h * k1[0][0];
  omegaForDexp[1][0] = omega[1][0] + 0.5 * h * k1[1][0];
  omegaForDexp[2][0] = omega[2][0] + 0.5 * h * k1[2][0];

  double uForDexp[3][1] = { 0 };
  uForDexp[0][0] = 0.5 * h * uk1[0][0];
  uForDexp[1][0] = 0.5 * h * uk1[1][0];
```

```
  uForDexp[2][0] = 0.5 * h * uk1[2][0];

  double uk2[3][1] = { 0 };
  dexp (omegaForDexp, uForDexp, uk2);

  double pk2[3][1] = { 0 };
  pk2[0][0] = v[0][0] + 0.5 * h * vk1[0][0];
  pk2[1][0] = v[1][0] + 0.5 * h * vk1[1][0];
  pk2[2][0] = v[2][0] + 0.5 * h * vk1[2][0];

////////////////////////////////third//////////////////////////7

  omegamfun (type, omegam);         //t argument should be t + 0.5*h
  double vk3[3][1] = { 0 };
  double k3[3][1] = { 0 };
//now new eul - omega argument is omega + h * k2
  double newOmegaThird[3][1] = { 0 };
  newOmegaThird[0][0] = omega[0][0] + 0.5 * h * k2[0][0];
  newOmegaThird[1][0] = omega[1][0] + 0.5 * h * k2[1][0];
  newOmegaThird[2][0] = omega[2][0] + 0.5 * h * k2[2][0];

//new quaternion argument for omega is appendmatrix[4x4] * exp_S3(0.5 h*uk2)

//define input[3x1] and output[4x1] variable for exp_S3

  double inputExpThird[3][1] =
    { {0.5 * h * uk2[0][0]}, {0.5 * h * uk2[1][0]}, {0.5 * h * uk2[2][0]} };
  double outputExpThird[4][1] = { 0 };
  exp_S3 (inputExpThird, outputExpThird);
//now multiply appendmatrix[4x4] * outputExp[4][1] - out is newq[4x1]

  double newqThird[4][1] = { 0 };
//now multipy
  for (int i = 0; i < 4; i++)
    {
      for (int j = 0; j < 1; j++)
        {
          for (int k = 0; k < 4; k++)
            {                          /
```

```
            newqThird[i][j] =
              newqThird[i][j] + appendedMatrix[i][k] * outputExpThird[k][j];
          }
        }
    }


  new_eul (SpecificMotor, SpecificAircraft, F_t, T_t, T_d, newOmegaThird,
           omegam, newqThird, vk3, k3);


//dexp(omegaForDexp[3x1],uForDexp[3x1],du[3,1]) example
  double omegaForDexpThird[3][1] = { 0 };
  omegaForDexpThird[0][0] = omega[0][0] + 0.5 * h * k2[0][0];
  omegaForDexpThird[1][0] = omega[1][0] + 0.5 * h * k2[1][0];
  omegaForDexpThird[2][0] = omega[2][0] + 0.5 * h * k2[2][0];

  double uForDexpThird[3][1] = { 0 };
  uForDexpThird[0][0] = 0.5 * h * uk2[0][0];
  uForDexpThird[1][0] = 0.5 * h * uk2[1][0];
  uForDexpThird[2][0] = 0.5 * h * uk2[2][0];

  double uk3[3][1] = { 0 };
  dexp (omegaForDexpThird, uForDexpThird, uk3);

  double pk3[3][1] = { 0 };
  pk3[0][0] = v[0][0] + 0.5 * h * vk2[0][0];
  pk3[1][0] = v[1][0] + 0.5 * h * vk2[1][0];
  pk3[2][0] = v[2][0] + 0.5 * h * vk2[2][0];


////////////////////////////////////////////////////////

  omegamfun (type, omegam);
  double vk4[3][1] = { 0 };
  double k4[3][1] = { 0 };
//now new eul - omega argument is omega + h * k2
  double newOmegaFourth[3][1] = { 0 };
  newOmegaFourth[0][0] = omega[0][0] + h * k3[0][0];
```

```
  newOmegaFourth[1][0] = omega[1][0] + h * k3[1][0];
  newOmegaFourth[2][0] = omega[2][0] + h * k3[2][0];


//new quaternion argument for omega is appendmatrix[4x4] * exp_S3(h*uk3)

//define input[3x1] and output[4x1] variable for exp_S3

  double inputExpFourth[3][1] =
    { {h * uk3[0][0]}, {h * uk3[1][0]}, {h * uk3[2][0]} };
  double outputExpFourth[4][1] = { 0 };
  exp_S3 (inputExpFourth, outputExpFourth);
//now multiply appendmatrix[4x4] * outputExp[4][1] - out is newq[4x1]

  double newqFourth[4][1] = { 0 };

  for (int i = 0; i < 4; i++)
    {
      for (int j = 0; j < 1; j++)
        {
          for (int k = 0; k < 4; k++)
            {
              newqFourth[i][j] =
                newqFourth[i][j] +
                appendedMatrix[i][k] * outputExpFourth[k][j];
            }
        }
    }


  new_eul (SpecificMotor, SpecificAircraft, F_t, T_t, T_d, newOmegaFourth,
           omegam, newqFourth, vk4, k4);

//dexp(omegaForDexp[3x1],uForDexp[3x1],du[3,1]) example
  double omegaForDexpFourth[3][1] = { 0 };
  omegaForDexpFourth[0][0] = omega[0][0] + h * k3[0][0];
  omegaForDexpFourth[1][0] = omega[1][0] + h * k3[1][0];
  omegaForDexpFourth[2][0] = omega[2][0] + h * k3[2][0];
```

```
  double uForDexpFourth[3][1] = { 0 };
  uForDexpFourth[0][0] = h * uk3[0][0];
  uForDexpFourth[1][0] = h * uk3[1][0];
  uForDexpFourth[2][0] = h * uk3[2][0];

  double uk4[3][1] = { 0 };
  dexp (omegaForDexpFourth, uForDexpFourth, uk4);

  double pk4[3][1] = { 0 };
  pk4[0][0] = v[0][0] + h * vk3[0][0];
  pk4[1][0] = v[1][0] + h * vk3[1][0];
  pk4[2][0] = v[2][0] + h * vk3[2][0];

////////////////////////////////////

  omega_out[0][0] =
    omega[0][0] + 1. / 6. * h * (k1[0][0] + 2 * k2[0][0] + 2 * k3[0][0] +
                                 k4[0][0]);
  omega_out[1][0] =
    omega[1][0] + 1. / 6. * h * (k1[1][0] + 2 * k2[1][0] + 2 * k3[1][0] +
                                 k4[1][0]);
  omega_out[2][0] =
    omega[2][0] + 1. / 6. * h * (k1[2][0] + 2 * k2[2][0] + 2 * k3[2][0] +
                                 k4[2][0]);

//declare varibles for exp_s3(input(3x1), out(4x1))
  double inputExpEnd[3][1] = { 0 };
  double outExpEnd[4][1] = { 0 };

  inputExpEnd[0][0] =
    1. / 6. * h * (uk1[0][0] + 2 * uk2[0][0] + 2 * uk3[0][0] + uk4[0][0]);
  inputExpEnd[1][0] =
    1. / 6. * h * (uk1[1][0] + 2 * uk2[1][0] + 2 * uk3[1][0] + uk4[1][0]);
  inputExpEnd[2][0] =
    1. / 6. * h * (uk1[2][0] + 2 * uk2[2][0] + 2 * uk3[2][0] + uk4[2][0]);

  exp_S3 (inputExpEnd, outExpEnd);
```

```
//now multip appended matrix (4x4) times outExpEnd)
  for (int i = 0; i < 4; i++)
    {
      for (int j = 0; j < 1; j++)
        {
          for (int k = 0; k < 4; k++)
            {
              q_out[i][j] =
                q_out[i][j] + appendedMatrix[i][k] * outExpEnd[k][j];
            }
        }
    }


  v_out[0][0] =
    v[0][0] + 1. / 6. * h * (vk1[0][0] + 2 * vk2[0][0] + 2 * vk3[0][0] +
                             vk4[0][0]);
  v_out[1][0] =
    v[1][0] + 1. / 6. * h * (vk1[1][0] + 2 * vk2[1][0] + 2 * vk3[1][0] +
                             vk4[1][0]);
  v_out[2][0] =
    v[2][0] + 1. / 6. * h * (vk1[2][0] + 2 * vk2[2][0] + 2 * vk3[2][0] +
                             vk4[2][0]);


  p_out[0][0] =
    p[0][0] + 1. / 6. * h * (pk1[0][0] + 2 * pk2[0][0] + 2 * pk3[0][0] +
                             pk4[0][0]);
  p_out[1][0] =
    p[1][0] + 1. / 6. * h * (pk1[1][0] + 2 * pk2[1][0] + 2 * pk3[1][0] +
                             pk4[1][0]);
  p_out[2][0] =
    p[2][0] + 1. / 6. * h * (pk1[2][0] + 2 * pk2[2][0] + 2 * pk3[2][0] +
                             pk4[2][0]);



}
```

rk4cl.c

```
void
rk4cl (double t, double h, struct Motor SpecificMotor,
       struct Aircraft SpecificAircraft, double omega[3][1], double v[3][1],
       double q[4][1], double p[3][1], char type[], double omega_out[3][1],
       double v_out[3][1], double q_out[4][1], double p_out[3][1])
{

  double omegam[4][1] = { 0 };
  double rpm_out[4][1] = { 0 };
  omegamfun (type, omegam);
  om2rpm (omegam, rpm_out);
  double F_t[4][1] = { 0 };
  double T_t[3][1] = { 0 };
  double T_d[3][1] = { 0 };
  thrust_forces (SpecificMotor, rpm_out, F_t);
  thrust_torque (SpecificMotor, SpecificAircraft, rpm_out, T_t);
  drag_torque (SpecificMotor, rpm_out, T_d);


  double vk1[3][1] = { 0 };
  double k1[3][1] = { 0 };


  new_eul (SpecificMotor, SpecificAircraft, F_t, T_t, T_d, omega, omegam, q,
           vk1, k1);


  double qk1[4][1] = { 0 };
  quat_vect (q, omega, qk1);

  for (int i = 0; i < 4; i++)
    {
      for (int j = 0; j < 1; j++)
        {
          qk1[i][j] = 0.5 * qk1[i][j];
        }
    }
```

```
double pk1[3][1] = { {v[0][0]}, {v[1][0]}, {v[2][0]} };


omegamfun (type, omegam);
double vk2[3][1] = { 0 };
double k2[3][1] = { 0 };

double newOmega[3][1] =
  { {omega[0][0] + 0.5 * h * k1[0][0]}, {omega[1][0] + 0.5 * h * k1[1][0]},
  {omega[2][0] + 0.5 * h * k1[2][0]} };


double newq[4][1] =
  { {q[0][0] + 0.5 * h * qk1[0][0]}, {q[1][0] + 0.5 * h * qk1[1][0]},
  {q[2][0] + 0.5 * h * qk1[2][0]}, {q[3][0] + 0.5 * h * qk1[3][0]} };


new_eul (SpecificMotor, SpecificAircraft, F_t, T_t, T_d, newOmega, omegam,
         newq, vk2, k2);

double qk2[4][1] = { 0 };
quat_vect (newq, newOmega, qk2);

for (int i = 0; i < 4; i++)
  {
    for (int j = 0; j < 1; j++)
      {
        qk2[i][j] = 0.5 * qk2[i][j];
      }
  }
double pk2[3][1] =
  { {v[0][0] + 0.5 * h * vk1[0][0]}, {v[1][0] + 0.5 * h * vk1[1][0]},
  {v[2][0] + 0.5 * h * vk1[2][0]} };


omegamfun (type, omegam);
double vk3[3][1] = { 0 };
double k3[3][1] = { 0 };
```

```
double newOmegaThree[3][1] =
  { {omega[0][0] + 0.5 * h * k2[0][0]}, {omega[1][0] + 0.5 * h * k2[1][0]},
  {omega[2][0] + 0.5 * h * k2[2][0]} };

double newqThree[4][1] =
  { {q[0][0] + 0.5 * h * qk2[0][0]}, {q[1][0] + 0.5 * h * qk2[1][0]},
  {q[2][0] + 0.5 * h * qk2[2][0]}, {q[3][0] + 0.5 * h * qk2[3][0]} };

new_eul (SpecificMotor, SpecificAircraft, F_t, T_t, T_d, newOmegaThree,
         omegam, newqThree, vk3, k3);

double qk3[4][1] = { 0 };
quat_vect (newqThree, newOmegaThree, qk3);

for (int i = 0; i < 4; i++)
  {
    for (int j = 0; j < 1; j++)
      {
        qk3[i][j] = 0.5 * qk3[i][j];
      }
  }

double pk3[3][1] =
  { {v[0][0] + 0.5 * h * vk2[0][0]}, {v[1][0] + 0.5 * h * vk2[1][0]},
  {v[2][0] + 0.5 * h * vk2[2][0]} };

double vk4[3][1] = { 0 };
double k4[3][1] = { 0 };

double newOmegaFour[3][1] =
  { {omega[0][0] + h * k3[0][0]}, {omega[1][0] + h * k3[1][0]},
  {omega[2][0] + h * k3[2][0]} };

double newqFour[4][1] =
  { {q[0][0] + h * qk3[0][0]}, {q[1][0] + h * qk3[1][0]},
  {q[2][0] + h * qk3[2][0]}, {q[3][0] + h * qk3[3][0]} };

new_eul (SpecificMotor, SpecificAircraft, F_t, T_t, T_d, newOmegaFour,
         omegam, newqFour, vk4, k4);
```

```
double qk4[4][1] = { 0 };
quat_vect (newqFour, newOmegaFour, qk4);

for (int i = 0; i < 4; i++)
  {
    for (int j = 0; j < 1; j++)
      {
        qk4[i][j] = 0.5 * qk4[i][j];
      }
  }

double pk4[3][1] =
  { {v[0][0] + h * vk3[0][0]}, {v[1][0] + h * vk3[1][0]},
  {v[2][0] + h * vk3[2][0]} };



omega_out[0][0] =
  omega[0][0] + 1. / 6. * h * (k1[0][0] + 2 * k2[0][0] + 2 * k3[0][0] +
                               k4[0][0]);
omega_out[1][0] =
  omega[1][0] + 1. / 6. * h * (k1[1][0] + 2 * k2[1][0] + 2 * k3[1][0] +
                               k4[1][0]);
omega_out[2][0] =
  omega[2][0] + 1. / 6. * h * (k1[2][0] + 2 * k2[2][0] + 2 * k3[2][0] +
                               k4[2][0]);



q_out[0][0] =
  q[0][0] + 1. / 6. * h * (qk1[0][0] + 2 * qk2[0][0] + 2 * qk3[0][0] +
                           qk4[0][0]);
q_out[1][0] =
  q[1][0] + 1. / 6. * h * (qk1[1][0] + 2 * qk2[1][0] + 2 * qk3[1][0] +
                           qk4[1][0]);
q_out[2][0] =
  q[2][0] + 1. / 6. * h * (qk1[2][0] + 2 * qk2[2][0] + 2 * qk3[2][0] +
                           qk4[2][0]);
q_out[3][0] =
```

```
  q[3][0] + 1. / 6. * h * (qk1[3][0] + 2 * qk2[3][0] + 2 * qk3[3][0] +
                           qk4[3][0]);


double q_outTransposed[1][4] =
  { q_out[0][0], q_out[1][0], q_out[2][0], q_out[3][0] };
double sumToBeSquared = 0;

for (int i = 0; i < 1; i++)
  {
    for (int j = 0; j < 1; j++)
      {
        for (int k = 0; k < 4; k++)
          {
            sumToBeSquared =
              sumToBeSquared + q_outTransposed[i][k] * q_out[k][j];
          }
      }
  }

q_out[0][0] = q_out[0][0] / sqrt (sumToBeSquared);
q_out[1][0] = q_out[1][0] / sqrt (sumToBeSquared);
q_out[2][0] = q_out[2][0] / sqrt (sumToBeSquared);
q_out[3][0] = q_out[3][0] / sqrt (sumToBeSquared);



v_out[0][0] =
  v[0][0] + 1. / 6. * h * (vk1[0][0] + 2 * vk2[0][0] + 2 * vk3[0][0] +
                           vk4[0][0]);
v_out[1][0] =
  v[1][0] + 1. / 6. * h * (vk1[1][0] + 2 * vk2[1][0] + 2 * vk3[1][0] +
                           vk4[1][0]);
v_out[2][0] =
  v[2][0] + 1. / 6. * h * (vk1[2][0] + 2 * vk2[2][0] + 2 * vk3[2][0] +
                           vk4[2][0]);


p_out[0][0] =
```

```
    p[0][0] + 1. / 6. * h * (pk1[0][0] + 2 * pk2[0][0] + 2 * pk3[0][0] +
                             pk4[0][0]);
  p_out[1][0] =
    p[1][0] + 1. / 6. * h * (pk1[1][0] + 2 * pk2[1][0] + 2 * pk3[1][0] +
                             pk4[1][0]);
  p_out[2][0] =
    p[2][0] + 1. / 6. * h * (pk1[2][0] + 2 * pk2[2][0] + 2 * pk3[2][0] +
                             pk4[2][0]);


}
```