Control d'actitud de satèl·lits artificials mitjançant aprenentatge automàtic

UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques i Informàtica

Isaac de Palau

Facultat de Matemàtiques i Informàtica

Universitat de Barcelona

Treball de Final de Grau, Matemàtiques

2020 MSC: 68T07, 70E50, 93B52, 65D30

Primavera 2021

Agraïments

Als meus pares: per seu suport moral (i econòmic) constant durant tots aquests anys de carrera i, en especial, aquests últims mesos tant difícils.

A Mariana Ramírez: pels cafès, les *arepas* i les sessions maratonianes d'estudi.

A Hari Jeon: per fer més amens els laboratoris de Càlcul Diferencial i Sistemes Dinàmics.

A Helena Gabí: per les hores passades estudiant Mètodes Numèrics II i bevent té *matcha*.

Als gats de la universitat: per la companyia durant tots aquests anys.

I, de forma molt especial, al Dr. Gerard Gómez. En primer lloc, per haver accedit a tutelar-me el treball de final de grau durant aquests últims mesos de carrera. En segon lloc, pel seu suport i paciència constants al llarg de tot el grau i haver atès molt amablement totes les meves consultes.

Abstract

When a satellite moves arround any orbit, it will encounter constant enviromental perturbations that will produce small changes in the attitude (i.e. the orientation and angular velocity) that, in the long term, will cause unstable flying and rolling. Thus, the satellite will have to perform constant maneuvers in order to correct its attitude.

An attitude controller is an algorithm that derives the appropiate maneuvers needed to correct the attitude error (i.e. the difference between the current and desired attitude), usually by calculating how much torque must be applied by the satellite actuators.

The purpose of this project is the study of the attitude control procedures of artificial satellites (as rigid bodies) by fusing classical control algorithms with modern machine-learning techniques. This project consists of two clearly differentiated parts: the study of the attitude (Ch. 2-3) and the study of the attitude control (Ch. 4-5-6).

In Chapters 2 and 3 we will explore the mathematical tools that will allow us to express a rotation in three dimensions: Euler rotation theorem and the unit quaternions. We will also derive the differential equations that dictate the attitude changes of the satellite when subject to external torques.

In Chapter 4 we will study the different types of attitude errors and we will introduce some basic concepts and definitions about control algorithms.

In Chapter 5 we will discuss the implementation of a virtual simulation enviroment that will allow us to test the attitude changes of a virtual satellite when subject to various torques. We will also implement a control law that will allow us to stabilize the satellite.

In Chapter 6, we will explore the attitude control problem as a decision problem and how the Deep-Q Learning algorithm can be used to train a controller.

Índex

1 Introducció

Tot satèl·lit artificial que orbiti seguint una trajectòria haurà de realitzar constantment maniobres per corregir la seva actitud, i.e., la seva orientació i velocitat angular. Això pot ser degut a diversos motius que dependran tant de l'òrbita en la que es situï l'aparell com de la tasca que vulgui desenvolupar. Per exemple, alguns satèl·lits (com els de telecomunicacions o de cartografia) disposen d'antenes i altres sensors que han d'estar sempre enfocats cap a la Terra; altres, com el satèl·lit d'astrometria Gaia que giren al voltant d'un eix (en principi fix), reben constantment l'impacte de petits asteroides i partícules que produeixen un moviment de precessió que ha de ser contrarrestat.

El *control d'actitud* d'un satèl·lit artificial té com a objectiu trobar la seqüència de moviments que permetin orientar el satèl·lit cap a una direcció desitjada. Una *llei de control* (a vegades anomenada controlador, o simplement control) és un algorisme que permet assolir aquest objectiu, habitualment dictant quin és el *moment de força* (torque) que han de produir els actuadors del satèl·lit a cada moment en funció de la diferència entre l'actitud real i la desitjada. Tradicionalment, el problema del control d'actitud s'ha estudiat des del punt de vista de la teoria de sistemes dinàmics: l'equació diferencial que regeix l'error en l'actitud defineix un sistema dinàmic que, amb l'elecció d'una llei de control adequada, es pot transformar en un sistema que convergirà cap a l'error nul, independentment de les seves condicions inicials.

Els avenços en aprenentatge automàtic i, concretament, en l'aprenentatge per reforç permeten també estudiar el problema del control d'actitud com a un problema de decisió de Markov. Aquí el controlador serà un *agent intel·ligent* que haurà d'aprendre quin és el

torque òptim que s'ha d'aplicar per tal de poder maximitzar una recompensa inversament proporcional a l'error en l'actitud.

Sobre aquest treball

L'objectiu final d'aquest treball és l'estudi del control d'actitud de satèl·lits artificials com a sòlids rígids, mitjançant tècniques tradicionals de control amb algorismes d'aprenentatge automàtic. El treball té dues parts ben diferenciades: **l'estudi d'actitud** (Capítols 2 i 3) i **l'estudi del control d'actitud** (Capítols 4, 5 i 6).

En els Capítols 2 i 3 començarem explorant aquelles eines matemàtiques que ens permetran expressar l'orientació d'un satèl·lit artificial: el Teorema de rotació d'Euler i els quaternions unitaris. També deduïrem les equacions diferencials que ens permetran descriure els canvis d'actitud del satèl·lit a l'aplicar-li diferents torques.

En el Capítol 4 veurem les formes d'expressar l'error en l'actitud. També estudiarem els controladors, veurem quines propietats han de complir per poder orientar amb èxit un satèl·lit i acabarem veient-ne un exemple concret.

En el Capítol 5, prendrem un enfocament més pràctic i implementarem un entorn de simulació en llenguatge Python que ens permeti simular els canvis d'actitud d'un satèl·lit artificial quan és sotmès a l'acció d'un torque. Així mateix, implementarem el controlador que haurem vist al final del quart capítol, així com les funcions per simular pertorbacions aleatòries i finalment discutirem els resultats obtinguts en varis experiments.

El darrer capítol del treball consisteix en l'estudi del problema del control d'actitud com a problema de decisió i com es pot usar l'algorisme *DQN* (Deep Q Network) per resoldre'l. Finalment, s'ha realitzat la implementació completa d'un controlador d'actitud inspirat en [6] mitjançant els paquets d'aprenentatge automàtic Tensorflow i Keras per a Python. Cal remarcar que, si bé s'ha aconseguit realitzar la implementació de l'agent i l'algorisme d'aprenentatge per reforç, no ha estat possible entrenar-lo en un temps raonable amb els mitjans dels quals s'ha disposat per fer el treball i obtenir així un controlador eficaç. En lloc d'això, s'ha optat per entrenar un controlador per a una versió simplificada del problema que, si bé no té una utilitat pràctica, sí que demostra que l'algorisme d'aprenentatge per

reforç seria capaç de produïr un controlador útil, sempre i quan es disposés de suficient potència de càlcul i temps d'entrenament.

2
Rotacions en $\mathbb{R}^3$

Per tal de poder controlar l'actitud d'un vehicle espacial és necessari conèixer la seva orientació en l'espai Euclidià; a la vegada, això ens obliga a cercar la manera adequada de descriure tal orientació. Intuïtivament, és fàcil adonar-se que tots els canvis d'orientació d'un sòlid rígid es poden considerar rotacions al voltant del seu centre de masses (vegeu Capítol 3 per a la definició de sòlid rígid i centre de masses). Per tant, tot l'espai de configuracions que expressen l'orientació del satèl·lit es pot identificar amb les rotacions al voltant d'un cert punt fix.

En aquest capítol estudiarem primerament el grup ortogonal especial $SO(3)$. Utilitzant el *Teorema d'Euler* veurem que els elements de $SO(3)$ es poden identificar amb les rotacions de $\mathbb{R}^3$ d'angle α al voltant d'un cert eix $\mathbf{r}$. En segon lloc estudiarem l'anell $\mathbb{H}$ dels *quaternions*. En particular, veurem com el grup format pels quaternions unitaris pot identificar-se amb el grup unitari especial $SU(2)$ i, a la vegada, com $SO(3)$ pot identificar-se amb $SU(2)$. Això demostrarà que els quaternions, efectivament, poden ser utilitzats per expressar rotacions en l'espai $\mathbb{R}^3$.

El contingut d'aquest capítol seguirà, en la seva major part, l'estructura de l'**Apèndix B** de [1]. L'excepció serà l'apartat ***2.1.1***, on seguirem l'estructura de les seccions **XII.4** i **XII.6** de [2].

2.1 Els grups O(n, $\mathbb{K}$) i SO(n, $\mathbb{K}$)

Definició. (Grup ortogonal)

Sigui $\mathbb{K}$ un cos, i sigui $GL(n, \mathbb{K})$ el *grup lineal general* format per totes les matrius $M \in \mathbb{K}^{n\times n}$ invertibles. S'anomena *grup ortogonal de dimensió n sobre* $\mathbb{K}$, denotat per $O(n, \mathbb{K})$, al

conjunt:

$$O(n,\mathbb{K}) := \{A \in GL(n,\mathbb{K}) : AA^T = Id\} \ .$$

Fixem-nos que $O(n,\mathbb{K})$ serà, doncs, el conjunt de totes les matrius ortogonals de $GL(n,\mathbb{K})$. Ens centrarem exclusivament en el cas en que $\mathbb{K} = \mathbb{R}$, i escriurem $O(n)$ per denotar $O(n,\mathbb{R})$.

Definició. (Grup Ortogonal Especial)
S'anomena *Grup Ortogonal Especial* al subconjunt $SO(n) \subset O(n)$ definit per:

$$\begin{aligned} SO(n) :&= \{A \in O(n) : det(A) = +1\} \\ &= \{A \in \mathbb{R}^{n\times n} : AA^T = Id, det(A) = +1\} \ . \end{aligned}$$

Proposició. *SO(n) és subgrup de O(n).*

Demostració. En primer lloc és evident que $Id \in SO(n)$, per tant $SO(n) \neq \emptyset$.
Agafem ara dues matrius $A, B \in SO(n)$, i hem de veure que $AB^{-1} \in SO(n)$. Si $B \in SO(n) \Rightarrow B^{-1} = B^T$ i $det(B^{-1}) = \frac{1}{det(B)} = 1 \Rightarrow det(AB^{-1}) = det(A)det(B^{-1}) = +1 \Rightarrow AB^{-1} \in SO(n)$. □

2.1.1 El grup SO(2)

Considerem E un $\mathbb{R}$-espai vectorial Euclidià de dimensió 2, amb base ortonormal e_1, e_2. Tota matriu $A \in SO(2)$ de la forma

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix},$$

compleix $det(A) = +1$ i $A^TA = I \Rightarrow A^{-1} = A^T$, és a dir:

$$A^{-1} = \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} = A^T = \begin{pmatrix} a & c \\ b & d \end{pmatrix} \ .$$

Això força que $a = d$ i $c = -b$, amb $a^2 + b^2 = 1$.

Proposició 2.1.1. *$SO(2)$ és un grup commutatiu.*

Demostració. Agafant dos elements $f, g \in SO(2)$ i fent els productes $f \cdot g$ i $g \cdot f$, es veu immediatament que són iguals. □

Definició. (Angle)
Definim la següent relació d'equivalència sobre les parelles de vectors unitaris de E:

$$\begin{aligned}(u,u') \sim (v,v') &\Leftrightarrow \exists f \in SO(2) \text{ tal que } f(u)=v, f(u')=v' \\ &\Leftrightarrow \exists g \in SO(2) \text{ tal que } g(u)=u', g(v)=v'\end{aligned}$$

Aquestes dues condicions són equivalents ([2], pàg. 253, secció **XII.5**). Anomenarem *angle* a cada una de les classes d'equivalència per aquesta relació $\sim$. L'angle determinat per un parell de vectors (u,u') el denotarem per $\widehat{uu'}$. L'angle de dos vectors u, v no unitaris serà l'angle dels vectors $\frac{u}{||u||}$ i $\frac{v}{||v||}$.

Definim ara A com el conjunt de tots els angles. L'aplicació

$$\begin{array}{ccc} SO(2) & \longrightarrow & A \\ f & \longrightarrow & \widehat{vf(v)} \end{array}$$

(on v és qualsevol vector unitari) és bijectiva. Això ens permet, fixada una orientació en E, assignar a cada matriu $f \in SO(2)$ un angle $\alpha \in A$. Si escrivim ara f com:

$$\begin{pmatrix} a & -b \\ b & a \end{pmatrix}, \text{ amb } a^2+b^2=1\,,$$

es defineix el *cosinus* i el *sinus* de α com:

$$\cos(\alpha)=a\,, \qquad \sin(\alpha)=b\,.$$

Es pot comprovar que l'orientació de E únicament canvia el signe de $\sin(\alpha)$, però deixa intacte $\cos(\alpha)$. Es pot veure també ([2] pàg. 254) que el *sinus* i el *cosinus* definits d'aquesta manera compleixen les propietats trigonomètriques a les que estem habituats:

1. $\cos(\alpha)^2+\sin(\alpha)^2=1$
2. $\cos(0)=1$, $\sin(0)=0$ (és a dir, la matriu associada a la rotació d'angle nul és la identitat)
3. $\cos(-\alpha)=\cos(\alpha) \quad \sin(-\alpha)=-\sin(\alpha)$

Definició. (Rotació)
L'aplicació $f \in SO(2)$, corresponent a un angle α, es diu *rotació (vectorial) d'angle* α, i l'escriurem com:

$$f=\begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}\,.$$

Anomenarem $SO(2)$ el grup de les rotacions de E.

Observem que les rotacions vectorials són transformacions que deixen fix l'orígen (trivial: $f(\vec{0}) = \vec{0}$), conserven angles ([2], pàg. 258, Proposició 5.4) i preserven l'orientació (per definició, ja que el determinant de f és $+1$).

2.1.2 El grup SO(3)

Una rotació en $\mathbb{R}^3$ és una rotació d'angle α sobre un pla $\pi \subset \mathbb{R}^3$ perpendicular a un eix $\mathbf{r}$. Si considerem les rotacions com a transformacions tals que deixen fix l'orígen, preserven angles, distàncies i orientacions, és evident que la matriu associada a una rotació a $\mathbb{R}^3$ ha de ser ortogonal (per a poder preservar el producte escalar i, per tant, preservar angles i distàncies) i amb determinant $+1$ (per preservar l'orientació). Així doncs, totes les rotacions sobre $\mathbb{R}^3$ són elements de $SO(3)$.

Ens interessa, però, veure la implicació inversa: que *qualsevol* matriu de $SO(3)$ serveix per representar una rotació. Aquest resultat ens el dóna el Teorema d'Euler, el qual demostrarem en aquesta secció. Abans, però, és necessari introduir el següent lema:

Lema. *Totes les matrius de $SO(3)$ tenen un valor propi igual a +1.*

Demostració. Sabem que tots els valors propis d'una matriu $A \in SO(3)$ són les solucions del polinomi de tercer grau $det(A - \lambda Id) = 0$. Sabem també que almenys una d'aquestes arrels ha de ser real, ja que el polinomi és de grau senar.
Sigui λ un valor propi real i v un vector propi associat, és a dir: $Av = \lambda v$ i $v^T v = 1$.
En conseqüència,

$$\lambda^2 = (\lambda v)^T(\lambda v) = (Av)^T(Av) = v^T A^T A v = v^T v = 1\,.$$

Per tant, $\lambda = \pm 1$. Com que el determinant de A és el producte de tots els seus valors propis, i sabem que $det(A) = +1$, això ens deixa únicament dues possibilitats:

1. Si els tres valors propis són reals, només poden ser o bé $\{1, -1, -1\}$ o bé $\{1, 1, 1\}$.

2. Si hi ha dos valors complexos no-reals, han de ser $(1, \omega, \overline{\omega})$.

En tots els casos observem que hi ha un valor propi igual a $+1$. □

Teorema 2.1.1. ***(Teorema d'Euler)*** *Tot element $A \in SO(3)$ diferent de la identitat és una rotació d'angle α al voltant d'un cert eix $\mathbf{r}$.*

Demostració. Pel Lema anterior, cada matriu $A \in SO(3)$ té un vector propi de valor propi 1. Anomenem $\mathbf{r}$ a aquest vector propi; es compleix que $A\mathbf{r} = \mathbf{r}$, i la recta del vector director de $\mathbf{r}$ és invariant per a A.
Sigui ara π un pla perpendicular a $\mathbf{r}$:

$$\pi = \{y \in \mathbb{R}^3 : \ < \mathbf{r}, y >= 0\},$$

(on $<\cdot, \cdot>$ denota el producte escalar). Sigui $\{e_1, e_2\}$ una base ortogonal de π. En la base $\{\mathbf{r}, e_1, e_2\}$, la matriu de A es pot escriure com:

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & a & b \\ 0 & c & d \end{pmatrix}.$$

Fixem-nos amb la sub-matriu

$$B = \begin{pmatrix} a & b \\ c & d \end{pmatrix}.$$

Aquesta matriu és ortogonal:

$$AA^T = \begin{pmatrix} 1 & 0^T \\ 0 & B \end{pmatrix} \begin{pmatrix} 1 & 0^T \\ 0 & B^T \end{pmatrix} = \begin{pmatrix} 1 & 0^T \\ 0 & BB^T \end{pmatrix} = Id_3 \Rightarrow BB^T = Id_2\,.$$

A més, el seu determinant és el mateix que A (per l'expansió de Laplace). Per tant, aquesta sub-matriu pertany a $SO(2)$ i, en conseqüència, és una rotació d'un cert angle $\alpha \Rightarrow A$ és una rotació d'angle α sobre el pla π perpendicular a $\mathbf{r}$.

□

2.2 El grup SU(2)

Considerem ara el grup lineal sobre els nombres complexos $GL(n, \mathbb{C})$, és a dir:

$$GL(n, \mathbb{C}) := \{M \in \mathbb{C}^{n \times n} : M \text{ invertible}\}$$

Definim també el *grup unitari* $U(n)$ i el *grup unitari especial* $SU(n)$ com:

$$\begin{aligned} U(n) &:= \{A \in GL(n, \mathbb{C}) : < Ax, Ay >=< x, y >, \forall x, y \in \mathbb{C}\} \\ &= \{A \in GL(n, \mathbb{C}) : \ A^\dagger A = Id\}, \\ SU(n) &:= \{A \in U(n) : detA = +1\} \\ &= \{A \in GL(n, \mathbb{C}) : \ A^\dagger A = Id, \ det(A) = +1\}, \end{aligned}$$

on $A^\dagger = \overline{A}^T$, i $< \cdot, \cdot >$ indica el producte hermític de $\mathbb{C}^n$:

$$< x, y >= \sum_{i=1}^{n} x_i \overline{y_i}\,.$$

Recordem ara que, donat un nombre complex de la forma $\alpha = x + iy$ es compleix

$$|\alpha| =< \alpha, \alpha >= \overline{\alpha} \cdot \alpha = |x|^2 + |y|^2\,.$$

Així doncs, veiem que podem escriure el grup $SU(2)$ com:

$$SU(2) = \left\{ \begin{pmatrix} a & b \\ -\overline{b} & \overline{a} \end{pmatrix} \,:\, a, b \in \mathbb{C}, |a|^2 + |b|^2 = +1 \right\},$$

donat que

$$A^\dagger A = \overline{A}^T A = \begin{pmatrix} \overline{a} & -b \\ \overline{b} & a \end{pmatrix} \cdot \begin{pmatrix} a & -b \\ -\overline{b} & \overline{a} \end{pmatrix} = \begin{pmatrix} \overline{a}a + \overline{b}b & \overline{a}b - b\overline{a} \\ \overline{a}b - b\overline{a} & \overline{a}a + \overline{b}b \end{pmatrix} =$$
$$= (|a|^2 + |b|^2) \cdot \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = Id\,.$$

Proposició. *El grup $SU(2)$ és difeomorf a l'esfera 3-dimensional $S^3 = \{x \in \mathbb{R}^4 : ||x|| = 1\}$.*

Demostració. Si agafem $x := (a_0, a_1, a_2, a_3) \in \mathbb{R}^4$, podem definir el següent difeomorfisme:

$$\begin{array}{rcl} \phi : S^3 & \longrightarrow & SU(2) \\ \mathbf{x} & \longrightarrow & \begin{pmatrix} a_0 + i\,a_3 & a_2 + i\,a_1 \\ -a_2 + i\,a_1 & a_0 - i\,a_3 \end{pmatrix} \end{array} \tag{2.1}$$

Aquesta aplicació està ben definida. En primer lloc, veiem que el determinant és 1:

$$det(\phi(x)) = \begin{vmatrix} a_0 + i\,a_3 & a_2 + i\,a_1 \\ -a_2 + i\,a_1 & a_0 - i\,a_3 \end{vmatrix} = a_0^2 + a_1^2 + a_2^2 + a_3^2 = 1 \quad \text{(donat que } x \in S^3)\,.$$

A més, si posem $a := a_0 + ia_3,\ b := a_2 + ia_1$, és evident que la matriu $\phi(\mathbf{x})$ pertany a $SU(2)$:

$$\phi(\mathbf{x}) = \begin{pmatrix} a_0 + i\,a_3 & a_2 + i\,a_1 \\ -a_2 + i\,a_1 & a_0 - i\,a_3 \end{pmatrix} = \begin{pmatrix} a & b \\ -\overline{b} & a \end{pmatrix} \in SO(2)\,.$$

A més, aquesta aplicació té inversa:

$$\begin{array}{rcl} \phi^{-1} : SU(2) & \longrightarrow & S^3 \\ \begin{pmatrix} a & b \\ -\overline{b} & \overline{a} \end{pmatrix} & \longrightarrow & (Re(a), Im(b), Re(b), Im(a)) \end{array}$$

que està ben definida perque es compleix $Re(a)^2 + Im(b)^2 + Re(b)^2 + Im(a)^2 = |a|^2 + |b|^2 = 1$. A més, tant ϕ com ϕ^{-1} són diferenciables. Per tant, ϕ defineix un difeomorfisme entre S^3 i $SU(2)$. □

2.3 Quaternions

Els *quaternions* es poden considerar una generalització dels nombres complexos. Formalment definirem l'anell dels quaternions $\mathbb{H}$ com el generat sobre els reals per la unitat 1 i tres complexos imaginaris $\mathbf{i}$, $\mathbf{j}$, i $\mathbf{k}$, que compleixen les següents relacions:

$$\begin{aligned}
&1 \cdot \mathbf{i} = \mathbf{i}\,,\ 1 \cdot \mathbf{j} = \mathbf{j}\,,\ 1 \cdot \mathbf{k} = \mathbf{k},\\
&\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = -1,\\
&\mathbf{i} \cdot \mathbf{j} = -\mathbf{j} \cdot \mathbf{i} = \mathbf{k}\,,\ \mathbf{j} \cdot \mathbf{k} = -\mathbf{k} \cdot \mathbf{j} = \mathbf{i}\,,\ \mathbf{k} \cdot \mathbf{i} = -\mathbf{i} \cdot \mathbf{k} = \mathbf{j}
\end{aligned}$$

Els elements $\mathfrak{q} \in \mathbb{H}$ són vectors 4-dimensionals de la forma:

$$\mathfrak{q} = q_0 + q_1\,\mathbf{i} + q_2\,\mathbf{j} + q_3\,\mathbf{k}$$

on $q_0, q_1, q_2, q_3 \in \mathbb{R}$. Habitualment, però, els representarem com vectors de la forma:

$$\mathfrak{q} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} q_0 \\ \mathbf{q} \end{bmatrix}\ .$$

Donats dos quaternions $\mathfrak{a} = [a_0, a_1, a_2, a_3] = [a_0, \mathbf{a}]$, $\mathfrak{b} = [b_0, b_1, b_2, b_3] = [b_0, \mathbf{b}]$, la seva suma i producte es defineixen mitjançant:

$$\begin{aligned}
\mathfrak{a} + \mathfrak{b} &= (a_0 + b_0) + (a_1 + b_1)\mathbf{i} + (a_2 + b_2)\mathbf{j} + (a_3 + b_3)\mathbf{k}\,,\\
\mathfrak{a} \otimes \mathfrak{b} &= (a_0 b_0 - \mathbf{a} \cdot \mathbf{b}, a_0\mathbf{b} + \mathbf{a}b_0 + \mathbf{a} \times \mathbf{b})\,.
\end{aligned}$$

Denotarem per $\mathfrak{q}^{-1}$ el conjugat d'un quaternió, és a dir:

$$\mathfrak{q}^{-1} = \begin{bmatrix} q_0 \\ -\mathbf{q} \end{bmatrix} = \begin{bmatrix} q_0 \\ -q_1 \\ -q_2 \\ -q_3 \end{bmatrix}$$

Direm que un quaternió és *vectorial* si $q_0 = 0$. El mòdul d'un quaternió $\mathfrak{q}$, denotat per $|\mathfrak{q}|$, és la norma Euclídia $|\mathfrak{q}| = \sqrt{\mathfrak{q}^T\mathfrak{q}} = \sqrt{q_0^2 + \mathbf{q}^T\mathbf{q}}$. Els quaternions de mòdul unitari, $|\mathfrak{q}| = 1$, s'anomenen *quaternions unitaris*. És evident que, per definició, els quaternions unitaris formen l'esfera S^3 (com a sub-varietat de $\mathbb{R}^4$).

Proposició. *El conjunt dels quaternions unitaris* $\{\mathfrak{q} \in \mathbb{H} : |\mathfrak{q}| = 1\}$ *formen un grup isomorf amb* $SU(2)$.

Demostració. Tenint en compte que són iguals a l'esfera S^3, únicament cal aplicar el difeomorfisme ***2.1*** i la demostració és immediata. □

Volem ara cercar una forma d'expressar els quaternions unitaris com a matrius de rotació de $SO(3)$. Es pot veure que $SU(2)$, com a grup de Lie, forma un recobriment $2:1$ de $SO(3)$ (la demostració es pot trobar a [1] pàg. 100, Proposició **B.3.10**), i que la matriu associada $A \in SO(3)$ a un quaternió unitari $\mathbf{x} = (a_0, a_1, a_2, a_3)$ és:

$$A := \begin{pmatrix} 2a_0^2 + 2a_1^2 - 1 & 2(a_0a_3 + a_1a_2) & 2(-a_0a_2 + a_1a_3) \\ 2(-a_0a_3 + a_1a_2) & 2a_0^2 + 2a_2^2 - 1 & 2(a_0a_1 + a_2a_3) \\ 2(a_0a_2 + a_1a_3) & 2(-a_0a_1 + a_2a_3) & 2a_0^2 + 2a_3^2 - 1 \end{pmatrix}$$

(la deducció d'aquesta equivalència es pot trobar a les pàgines 101 i 102 de [1]).

El següent resultat ens permet identificar el producte de quaternions amb una rotació al voltant d'un eix arbitrari que passi per l'orígen:

Teorema 2.3.1. *Sigui $\boldsymbol{r} \in \mathbb{R}^3$ un eix de rotació, $\alpha \in \mathbb{R}$ un angle de gir, i $\mathbf{v} \in \mathbb{R}^3$ un vector arbitrari que volem fer girar un angle α al voltant de $\boldsymbol{r}$, i $\mathbf{v}_{rot}$ el vector transformat. Si definim $\widetilde{\mathbf{q}} = (\cos\frac{\alpha}{2}, \sin\frac{\alpha}{2}\,\mathbf{r})$, $\widetilde{\mathbf{v}} = (0, \mathbf{v})$ i $\widetilde{\mathbf{v}}_{rot} = (0, \mathbf{v}_{rot})$, es compleix:*

$$\widetilde{\mathbf{v}}_{rot} = \widetilde{\mathbf{q}}\widetilde{\mathbf{v}}\widetilde{\mathbf{q}}^{-1} \tag{2.2}$$

*(**Observació:** notem que un quaternió de la forma $(\cos\theta, \sin\theta\boldsymbol{u})$, amb $||\boldsymbol{u}|| = 1$, és un quaternió unitari)*

Demostració. Comencem per descomposar $\mathbf{v}$ en les components perpendicular ($\mathbf{v}_\perp$) i paral·lela ($\mathbf{v}_\parallel$) a $\mathbf{r}$:

$$\mathbf{v} = \mathbf{v}_\perp + \mathbf{v}_\parallel, \qquad \widetilde{\mathbf{v}} = \widetilde{\mathbf{v}}_\perp + \widetilde{\mathbf{v}}_\parallel = (0, \mathbf{v}_\perp) + (0, \mathbf{v}_\parallel)\,.$$

Definim també un tercer vector $\mathbf{v}_0$ perpendicular a $\mathbf{r}$, $\mathbf{v}$ i $\mathbf{v}_\perp$:

$$\mathbf{v}_0 := \mathbf{r} \times \mathbf{v} = \mathbf{r} \times \mathbf{v}_\perp\,.$$

Si $\mathbf{v}_{rot}$ és el vector transformat, el podem escriure com

$$\mathbf{v}_{rot} = \mathbf{v}_\parallel + \cos(\alpha)\mathbf{v}_\perp + \sin(\alpha)\mathbf{v}_0$$

Vegem ara que $\tilde{\mathbf{v}}_\parallel$ commuta amb $\tilde{\mathbf{q}}$. Aplicant la fòrmula del producte de dos quaternions:

$$\begin{aligned}
\tilde{\mathbf{q}}\tilde{\mathbf{v}}_\parallel &= \Big(\cos\frac{\alpha}{2}, \sin\frac{\alpha}{2}\mathbf{r}\Big)(0, \mathbf{v}_\parallel) \\
&= \Big(-\sin\frac{\alpha}{2}\mathbf{r}\cdot\mathbf{v}_\parallel, \cos\frac{\alpha}{2}\mathbf{v}_\parallel + \sin\frac{\alpha}{2}\mathbf{r}\times\mathbf{v}_\parallel\Big) \\
&= \Big(-\mathbf{v}_\parallel\cdot\sin\frac{\alpha}{2}\mathbf{r}, \cos\frac{\alpha}{2}\mathbf{v}_\parallel + \mathbf{v}_\parallel\times\sin\frac{\alpha}{2}\mathbf{r}\Big) \\
&= (0, \mathbf{v}_\parallel)\Big(\cos\frac{\alpha}{2}, \sin\frac{\alpha}{2}\mathbf{r}\Big) = \tilde{\mathbf{v}}_\parallel\tilde{\mathbf{q}}\ .
\end{aligned}$$

Finalment, veiem que $\tilde{\mathbf{q}}\tilde{\mathbf{v}}_\parallel\tilde{\mathbf{q}}^{-1}$ és el vector $\tilde{\mathbf{v}}_{rot}$:

$$\begin{aligned}
\tilde{\mathbf{q}}\tilde{\mathbf{v}}_\parallel\tilde{\mathbf{q}}^{-1} &= \tilde{\mathbf{q}}(\tilde{\mathbf{v}}_\parallel + \tilde{\mathbf{v}}_\perp)\tilde{\mathbf{q}}^{-1} \\
&= \tilde{\mathbf{q}}\tilde{\mathbf{v}}_\parallel\tilde{\mathbf{q}}^{-1} + \tilde{\mathbf{q}}\tilde{\mathbf{v}}_\perp\tilde{\mathbf{q}}^{-1} \\
&= \tilde{\mathbf{v}}_\parallel\tilde{\mathbf{q}}\tilde{\mathbf{q}}^{-1} + \tilde{\mathbf{q}}\tilde{\mathbf{v}}_\perp\tilde{\mathbf{q}}^{-1} \\
&= \tilde{\mathbf{v}}_\parallel + \tilde{\mathbf{q}}\tilde{\mathbf{v}}_\perp\tilde{\mathbf{q}}^{-1} \\
&= \tilde{\mathbf{v}}_\parallel + \Big(\cos\frac{\alpha}{2}, \sin\frac{\alpha}{2}\mathbf{r}\Big)(0, \mathbf{v}_\perp)\tilde{\mathbf{q}}^{-1} \\
&= \tilde{\mathbf{v}}_\parallel + \Big(-\sin\frac{\alpha}{2}(\mathbf{r}+\tilde{\mathbf{v}}_\perp), \cos\frac{\alpha}{2}\mathbf{v}_\perp + \sin\frac{\alpha}{2}(\mathbf{r}\times\mathbf{v}_\perp)\Big)\tilde{\mathbf{q}}^{-1} \\
&= \tilde{\mathbf{v}}_\parallel + \Big(0, \cos\frac{\alpha}{2}\mathbf{v}_\perp + \sin\frac{\alpha}{2}\mathbf{v}_0\Big)\Big(\cos\frac{\alpha}{2}, -\sin\frac{\alpha}{2}\mathbf{r}\Big) \\
&= \tilde{\mathbf{v}}_\parallel + \Big(\sin\frac{\alpha}{2}\cos\frac{\alpha}{2}(\mathbf{v}_\perp\cdot\mathbf{r} - \sin^2\frac{\alpha}{2}(\mathbf{v}_0\cdot\mathbf{r}), \\
&\qquad \cos^2\frac{\alpha}{2}\mathbf{v}_\perp + \sin\frac{\alpha}{2}\cos\frac{\alpha}{2}(\mathbf{v}_0 - (\mathbf{v}_\perp\times\mathbf{r})) - \sin^2\frac{\alpha}{2}(\mathbf{v}_0\times\mathbf{r})\Big) \\
&= \tilde{\mathbf{v}}_\parallel + \Big(0, \Big(\cos^2\frac{\alpha}{2} - \sin^2\frac{\alpha}{2}\Big)\mathbf{v}_\perp + 2\sin\frac{\alpha}{2}\cos\frac{\alpha}{2}\mathbf{v}_0\Big) \\
&= \tilde{\mathbf{v}}_\parallel + \big(0, \cos\alpha\mathbf{v}_\perp + \sin\alpha\mathbf{v}_0\big) \\
&= \tilde{\mathbf{v}}_\parallel + \cos\alpha\tilde{\mathbf{v}}_\perp + \sin\alpha\tilde{\mathbf{v}}_0 \\
&= \tilde{\mathbf{v}}_{rot} \qquad \textit{(per definició de } \tilde{\boldsymbol{v}}_{rot}\textit{)}\,.
\end{aligned}$$

□

Així doncs, veiem que és possible identificar un quaternió unitari $\tilde{\mathbf{q}} = (q_0, \mathbf{q}) = (\cos\frac{\alpha}{2}, \sin\frac{\alpha}{2}\mathbf{r})$ amb la rotació d'angle α i eix $\mathbf{r}$, i denotarem per $R(\tilde{\mathbf{q}})$ la matriu de $SO(3)$ associada a aquesta rotació.

Aquest teorema implica que donats dos sistemes de referència $\mathcal{A}$, $\mathcal{B}$ (vegeu la secció 3.1 del proper capítol per la definició de sistema de referència), i un vector $v \in \mathbb{R}^3$, existeix un quaternió $\mathfrak{q}$ tal que ${}^{\mathcal{A}}v = R(\mathfrak{q})\,{}^{\mathcal{B}}v$, és a dir, que permet transformar el vector v expressat en coordenades de $\mathcal{B}$ a coordenades de $\mathcal{A}$. En aquest cas, posarem $R_B^A := R(\mathfrak{q})$.

3

Dinàmica de sòlids rígids

Un sòlid rígid és un objecte indeformable, això és, que no pot canviar la seva forma per acció de forces externes. Si bé hom podria argumentar que a la pràctica no existeixen sòlids rígids (qualsevol cos és, en certa mesura, deformable) nosaltres considerarem que el nostre satèl·lit n'és un. Això ens permetrà estudiar l'actitud del satèl·lit des del punt de vista de la *dinàmica de sòlids rígids*, que és la branca de la mecànica clàssica que s'ocupa d'estudiar el moviment d'aquest tipus d'objectes a l'estar sotmesos a l'acció de forces externes.

Així doncs, l'objectiu d'aquest capítol serà el de derivar un model adequat que ens permeti descriure les variacions en l'actitud del satèl·lit quan està sotmès a moments de força externs, com poden ser els produïts pels actuadors del vehicle o per les pertorbacions ambientals.

La primera part d'aquest capítol, on s'introdueix la noció de sistema de referència i es defineixen formalment els sòlids rígids, està basada en les seccions **2.5.1** i **2.5.2** de [3]. La derivació de l'equació de l'actitud s'ha fet seguint els passos de les seccions **6.4.1** i **7.2.1** de [3].

3.1 Sistemes de referència

Considerem l'espai Euclidià E de dimensió 3, i denotem per $\vec{E}$ l'espai vectorial associat. Un *sistema de referència* $\mathcal{E} = \{O; \vec{e}_1, \vec{e}_2, \vec{e}_3\}$ és el conjunt format per tres vectors ortonormals $\vec{e}_1, \vec{e}_2, \vec{e}_3 \in \vec{E}$ que sorgeixen d'un punt orígen $O \in \mathbb{R}^3$.

Un vector $\vec{v} = \overrightarrow{OP}$ del punt O al punt $P \in \mathbb{R}^3$ està representat de forma única per les tres projeccions ortogonals $v_k = \vec{v} \cdot \vec{e}_k$, $k = 1, 2, 3$ sobre cada eix del sistema de referència. L'escalar v_k s'anomena la *k-èssima coordenada* del vector $\vec{v}$ en el sistema de referència $\mathcal{E}$. Habitualment quan parlem de vectors ens referim al vector columna format per les coordenades de $\vec{v}$: ${}^{\mathcal{E}}v = [v_1, v_2, v_3]^T$.

El *moviment* o trajectòria d'un punt P a l'espai 3D es defineix com el vector de coordenades $\vec{r}(t)$, expressat en algun sistema de referència, al llarg d'un interval de temps $t_0 \leq t \leq t_f$.

Per últim, direm que un sistema de referència $\mathcal{I} = \{O; \vec{i}_1, \vec{i}_2, \vec{i}_3\}$ és *inercial* si l'orígen O no està sotmès a cap tipus d'acceleració ($\ddot{\vec{r}}_O(t) = 0$) i els eixos no roten ($\dot{\vec{i}}_k = 0,\ k = 1, 2, 3$).

3.2 Sòlids rígids

Un sòlid rígid és una distribució contínua de partícules P de massa elemental dm situades a una posició $\vec{r}$ respecte d'un sistema de referència inercial, la posició relativa de les quals és invariant. És a dir, donades dues partícules P_a i P_b qualsevol del sòlid rígid, es compleix:

$$||\vec{r}_a - \vec{r}_b|| = \text{constant}$$

(per a tota norma vectorial $||\cdot||$, o mètrica). Per tal de poder estudiar el moviment i l'actitud del sòlid rígid ens caldrà definir dos sistemes de referència. El primer és el sistema de referència inercial, que ja hem introduït i que denotarem per $\mathcal{I} = \{O; \vec{i}_1, \vec{i}_2, \vec{i}_3\}$.

El segon serà un sistema de referència *solidari* amb el cos que denotarem per $\mathcal{B} = \{C, \vec{b}_1, \vec{b}_2, \vec{b}_3\}$. L'elecció de l'orígen C és arbitrària, malgrat que habitualment s'utilitza el centre de masses del sòlid rígid:

$$C = \vec{r}_{cdm} := \frac{\int \vec{r}\, dm}{\int \vec{r}}\ .$$

Per construir els tres vectors $\vec{b}_1, \vec{b}_2, \vec{b}_3$, suposarem que en el sòlid rígid hi ha tres partícules no alineades de coordenades $\vec{r}_1, \vec{r}_2, \vec{r}_3$. Si agafem els vectors no-colineals

$$\vec{u}_1 = \vec{r}_1 - \vec{r}_2, \quad \vec{u}_2 = \vec{r}_3 - \vec{r}_1$$

podem construir un sistema de referència ortonormal usant el mètode de Gram-Schmidt:

$$\vec{b}_1 = \frac{\vec{u}_1}{||\vec{u}_1||}, \quad \vec{b}_2 = \frac{\vec{u}_2 - < \vec{b}_1, \vec{u}_2 > \vec{b}_1}{||\vec{u}_2 - < \vec{b}_1, \vec{u}_2 > \vec{b}_1||}, \quad \vec{u}_3 = \vec{b}_1 \times \vec{b}_2$$

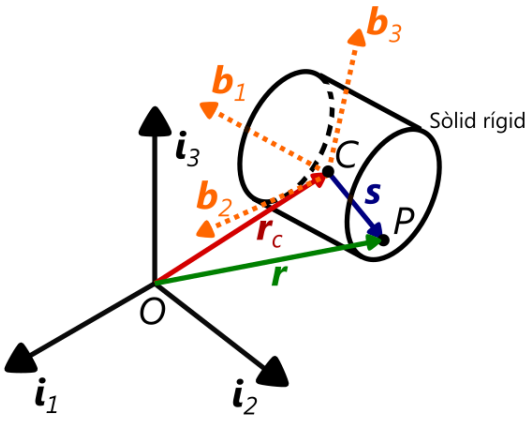
(**Nota:** Donat un vector $v \in \mathbb{R}^3$, utilitzarem la notació vectorial ($\vec{v}$) per expressar-lo en coordenades del sistema de referència $\mathcal{I}$. En canvi, usarem negreta ($\mathbf{v}$) quan l'expressem en el sistema de referència $\mathcal{B}$. És a dir, $\vec{v} = {}^{\mathcal{I}}v$, $\mathbf{v} = {}^{\mathcal{B}}v$.)

Siguin ara $\mathbf{r} = (x_i, y_i, z_i)$ les coordenades d'una massa puntual P_i en el sistema de referència $\mathcal{B}$. Aleshores es té

$$\vec{r}_i = \vec{r}_{cdm} + x_i \vec{b}_1 + y_i \vec{b}_2 + z_i \vec{b}_3 := \vec{r}_{cdm} + B \cdot \mathbf{r} \tag{3.1}$$

i, si definim $\vec{s}_i := \vec{r}_i - \vec{r}_{cdm}$, s'obté $\mathbf{r} = B^{-1}\vec{s}$.

La següent figura il·lustra el sistema en que es mou el sòlid rígid, tenint en compte totes les definicions que hem introduït fins ara:



Si derivem ara l'equació (3.1) respecte del temps, obtenim l'expressió de la velocitat del punt P_i:

$$\dot{\vec{r}}_i = \dot{\vec{r}}_{cdm} + \dot{B}\mathbf{r} + B\dot{\mathbf{r}}\,.$$

Les distàncies relatives entre dos punts d'un sòlid rígid són constants i en particular també ho és la distància entre C i P_i. Per tant les coordenades de $\mathbf{r}$ no canvien en funció del temps, i això farà que $\dot{\mathbf{r}} = 0$. En conseqüència l'equació de la velocitat es converteix en

$$\dot{\vec{r}}_i = \dot{\vec{r}}_{cdm} + \dot{B}\mathbf{r} = \dot{\vec{r}}_{cdm} + \dot{B}\dot{B}^{-1}\vec{s}\,.$$

Com que el que ens interessa és estudiar únicament les rotacions, ignorarem la translació de l'equació anterior i la reduïrem a

$$\dot{\vec{r}}_i = \dot{B}\mathbf{r} = \dot{B}\dot{B}^{-1}\vec{s}\,.$$

Definició. Sigui $\vec{q}$ el vector de coordenades d'un cert punt expressades en el sistema de referència $\mathcal{I}$, i $\mathbf{q}$ les coordenades del mateix punt expressades en $\mathcal{B}$. Es té

$$\dot{\vec{q}} = \dot{B}\mathbf{q} = \dot{B}B^{-1}\vec{q}\,.$$

Definim l'operador lineal $A := \dot{B}B^{-1} : \mathcal{I} \longrightarrow \mathcal{I}$.

Lema. *A és un operador antisimètric, és a dir, $A = -A^T$ (o, equivalentment, $A + A^T = 0$).*

Demostració. Si B és una rotació, llavors és una transformació ortogonal ($B^{-1} = B^T$) $\Rightarrow\ BB^T = BB^{-1} = Id$. Si derivem,

$$\dot{Id} = 0 = \dot{B}B^{-1} + B\dot{B}^{-1} = A + (B\dot{B}^T) = A + (\dot{B}B^T)^T = A + A^T$$

□

Lema. *Tot operador antisimètric A de l'espai euclideà orientat $\mathbb{R}^3$ es pot expressar com un producte vectorial per un vector fix $\vec{\omega} \in \mathbb{R}^3$:*

$$A\vec{q} = \vec{\omega} \times \vec{q}, \qquad \forall \vec{q} \in \mathbb{R}^3\ .$$

Demostració. Recordem que per tal que una matriu A compleixi $A = -A^T$, els seus elements han de complir $a_{ij} = -a_{ji}$. Per tant, si prenem coordenades cartesianes, la matriu de A tindrà la següent forma:

$$A = \begin{pmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{pmatrix}, \qquad \text{amb } \omega_1, \omega_2, \omega_3 \in \mathbb{R}\ .$$

Els operadors antisimètrics de $\mathbb{R}^3$ són, doncs, un espai vectorial de dimensió 3. Fixem ara un vector qualsevol $\vec{\omega}$ de $\mathbb{R}^3$, i definim l'operador $V_{\vec{\omega}}$ com:

$$\begin{aligned} V_{\vec{\omega}} : \mathbb{R}^3 &\longrightarrow \mathbb{R}^3 \\ \vec{q} &\longrightarrow \vec{\omega} \times \vec{q} \end{aligned}$$

aquest operador és lineal i antisimètric (ja que el producte vectorial de dos vectors també ho és). Per tant, la família de tots els operadors $V_{\vec{\omega}}$ amb $\vec{\omega} \in \mathbb{R}^3$ forma un subespai de l'espai vectorial dels operadors antisimètrics.

No obstant, aquest subespai té dimensió 3, aleshores podem identificar-lo amb l'espai dels operadors antisimètrics de $\mathbb{R}^3$ (que hem vist que també tenia dimensió 3). □

Com que $A = \dot{B}B^{-1}$, observem que

$$A\vec{q} = \vec{\omega} \times \vec{q} \ \Leftrightarrow\ \dot{B}\vec{q} = B(\vec{\omega} \times \vec{q}) \tag{3.2}$$

Una conseqüència immediata d'aquests dos lemes és que per a cada instant t, hi haurà un vector $\vec{\omega}_t \in \mathbb{R}^3$ tal que

$$\dot{\vec{q}}(t) = \vec{\omega}_t \times \vec{q}(t) \tag{3.3}$$

3.3 Equació de l'actitud

En aquesta secció deduirem l'equació que descriuen els canvis d'actitud que pateix el sòlid rígid.

3.3.1 Equació de rotació de Newton

El moment angular infinitesimal d'una partícula P de massa elemental dm es defineix com el producte $\vec{r} \times \dot{\vec{r}}\, dm$. Per tant, integrant sobre el volum del sòlid rígid B, s'obté el seu moment angular $\vec{H}$ respecte l'orígen O:

$$\vec{H} = \int \vec{r} \times \dot{\vec{r}}\, dm \ . \tag{3.4}$$

Derivant ara respecte al temps, i tenint en compte que $\dot{\vec{r}} \times \dot{\vec{r}} = 0$, s'obté

$$\dot{\vec{H}} = \int \vec{r} \times \ddot{\vec{r}}\, dm \ . \tag{3.5}$$

La segona llei de Newton, aplicada a una partícula de massa infinitesimal, es pot escriure com $\ddot{\vec{r}}\, dm = \vec{F}_{int} + \vec{F}$, on $\vec{F}$ denota la suma de les forces infinitesimals externes i $\vec{F}_{int}$ denota la suma de les internes. Fixem-nos ara amb el moment de força en relació a l'orígen O:

$$\vec{M} := \int_B \vec{r} \times (\vec{F}_{int} + \vec{F}) \ . \tag{3.6}$$

Com que estem treballant sobre un sòlid rígid, $\vec{F}_{int} = 0$, per tant:

$$\vec{M} = \int_B \vec{r} \times \vec{F} = \int_B \vec{r} \times \ddot{\vec{r}}\, dm \ . \tag{3.7}$$

De la igualtat entre les equacions (3.5) i (3.7) s'obté l'*equació de rotació de Newton*:

$$\dot{\vec{H}} = \int \vec{r} \times \ddot{\vec{r}}\, dm = \vec{M} \ . \tag{3.8}$$

3.3.2 Moment angular del centre de masses

En aquest apartat estudiarem l'equació del moment angular de P en relació al centre de masses C del sòlid rígid. Per definició, sabem que la posició i la velocitat del centre de masses C s'expressen com:

$$\vec{r}_c = \frac{1}{m} \int_B \vec{r}\, dm \qquad \dot{\vec{r}} = \frac{1}{m} \int_B \dot{\vec{r}}\, dm \ .$$

Sabem que el vector $\vec{r}$ es pot descomposar en $\vec{r} = \vec{r}_c + \vec{s}$. Aleshores, el moment angular i el moment de la força $\vec{F}$ sobre P (torque), respecte de C, es defineixen com

$$\vec{H}_c = \int_B \vec{s} \times \dot{\vec{s}}\, dm \qquad \vec{M}_c = \int_B \vec{s} \times \vec{F}\ .$$

El moment angular respecte O es podrà expressar llavors com:

$$\begin{aligned}
\vec{H} &= \int_B \vec{r} \times \dot{\vec{r}}\, dm \\
&= \int_B (\vec{r}_c + \vec{s}) \times (\dot{\vec{r}}_c + \dot{\vec{s}})\, dm \\
&= \vec{r}_c \times \dot{\vec{r}}_c \int_B dm + \vec{r}_c \times \int_B \dot{\vec{s}}\, dm + \int_B \vec{s}\, dm \times \vec{r}_c + \int_B \vec{s} \times \dot{\vec{s}}\, dm \\
&= m\vec{r}_c \times \dot{\vec{r}}_c + \int_B \vec{s} \times \dot{\vec{s}}\, dm \\
&= m\vec{r}_c \times \dot{\vec{r}}_c + \vec{H}_c\ .
\end{aligned}$$

De forma similar,

$$\begin{aligned}
\vec{M} = \int_B \vec{r} \times \vec{F} = \int_B (\vec{r}_c + \vec{s}) \times \vec{F} = \\
= \vec{r}_c \times \int_B \vec{F} + \int_B \vec{s} \times \vec{F} = \vec{r}_c \times \vec{F} + \vec{M}_c\ .
\end{aligned}$$

Derivem ara $\vec{H}$ respecte del temps:

$$\dot{\vec{H}} = \vec{r}_c \times m\ddot{\vec{r}}_c + \dot{\vec{H}}_c = \vec{r}_c \times \vec{F} + \dot{\vec{H}}_c\ . \tag{3.9}$$

Finalment, aplicant l'equació (3.8), es té

$$\dot{\vec{H}} = \vec{r}_c \times \vec{F} + \dot{\vec{H}}_c = \vec{M} = \vec{r}_c \times \vec{F} + \vec{M}_c \qquad \Rightarrow \quad \dot{\vec{H}}_c = \vec{M}_c\ . \tag{3.10}$$

3.3.3 Tensor d'inèrcia

Fins ara hem treballat únicament sobre el sistema de referència inercial $\mathcal{I}$. Ara ens interessarà estudiar com es pot expressar el moment angular en les coordenades de $\mathcal{B}$, $\mathbf{H}_c$. Recordem que:

$$\vec{H}_c = \int_B \vec{s} \times \dot{\vec{s}}\, dm\ .$$

Com que el cos és rígid, $\dot{\vec{s}} = \vec{\omega} \times \vec{s}$ per l'equació (3.3). És a dir, $\dot{\vec{s}}$ estarà produït per una rotació infinitesimal $|\vec{\omega}|\, dt$ al voltant d'un cert eix $\vec{e} = \frac{\vec{\omega}}{|\vec{\omega}|}$ que passa pel centre de masses. Per tant,

$$\begin{aligned}
\dot{\vec{s}} = \vec{\omega} \times \vec{s} \Leftrightarrow \dot{\mathbf{s}} = \boldsymbol{\omega} \times \mathbf{s} \qquad \Rightarrow\ \mathbf{H}_c &= \int_B \mathbf{s} \times (\boldsymbol{\omega} \times \mathbf{s}) \\
&= (\int_B ((\mathbf{s}^T\mathbf{s})\,\mathrm{Id}_3 - \mathbf{s}\mathbf{s}^T) dm)\boldsymbol{\omega} = J\,\boldsymbol{\omega}\,.
\end{aligned}$$

Anomenem a J el *tensor d'inèrcia* del sòlid rígid. Si posem $\mathbf{s} = (s_1, s_2, s_3)^T$, llavors

$$J = \begin{pmatrix} J_{11} & J_{12} & J_{13} \\ J_{21} & J_{22} & J_{23} \\ J_{31} & J_{32} & J_{33} \end{pmatrix} = \int_B \begin{pmatrix} s_2^2 + s_3^2 & -s_1 s_2 & -s_1 s_3 \\ -s_1 s_2 & s_1^2 + s_3^2 & -s_2 s_3 \\ -s_1 s_3 & -s_2 s_3 & s_1^2 + s_2^2 \end{pmatrix} dm \ . \tag{3.11}$$

3.3.4 Equació de rotació d'Euler

Havent trobat $\mathbf{H}_c$, ens interessarà poder expressar també l'equació de rotació de Newton (3.8) en les coordenades de $\mathcal{B}$. Definint $\mathbf{B} := (\vec{b}_1 | \vec{b}_2 | \vec{b}_3)$, podem escriure $\vec{H}_c = \mathbf{B}\,\mathbf{H}_c = \mathbf{B}\,J\,\boldsymbol{\omega}$. Aleshores, la derivada de $\vec{H}_c$ respecte al temps es pot escriure com

$$\dot{\vec{H}}_c = \mathbf{B}\dot{\mathbf{H}}_c + \dot{\mathbf{B}}\,\mathbf{H}_c = \mathbf{B}\,(\mathbf{H}_c + \boldsymbol{\omega} \times \mathbf{H}_c) = \mathbf{B}\,(J\,\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times J\,\boldsymbol{\omega}) \tag{3.12}$$

(observem que a la segona igualtat hem aplicat l'Eq. 3.2). Finalment, aplicant la igualtat $\dot{\vec{H}}_c = \vec{M}_c = \mathbf{B}\,\mathbf{M}_c$, obtenim l'*equació de rotació d'Euler clàssica*:

$$\dot{\boldsymbol{\omega}}(t) = -J^{-1}\,\boldsymbol{\omega} \times J\,\boldsymbol{\omega}(t) + J^{-1}\mathbf{M}_c \tag{3.13}$$

amb la condició inicial $\boldsymbol{\omega}(0) = \omega_0 \in \mathbb{R}^3$.

3.3.5 Equació d'espai d'estats

De moment hem trobat una expressió per als canvis de velocitat angular del sòlid rígid, però ens interessa conèixer també els canvis en la rotació (expressada en quaternions). En altres paraules, volem trobar una expressió per a $\dot{\mathfrak{q}}(t)$. Considerem un quaternió de la forma $\mathfrak{q}(t) = [q_0(t), \mathbf{q}(t)]$ en un instant de temps t, que es correspon a la rotació entre el sistema de referència inercial $\mathcal{I}$ i el sistema de referència del cos $\mathcal{B}$, i sigui Δt un interval de temps. Apliquem ara a $\mathfrak{q}(t)$ una petita rotació $(\alpha(t), \vec{e}(t))$, on $\vec{e}(t)$ és l'eix de rotació (expressat en les coordeandes de $\mathcal{B}$) i $\alpha(t)$ (l'angle) és la velocitat angular $\vec{\omega}(t)$ integrada al llarg de Δt:

$$\alpha(t) = \vec{\omega}(t)\Delta t\,.$$

Si denotem per $\mathbf{e}$ les coordenades de l'eix de rotació $\vec{e}$ en $\mathcal{B}$, veiem que el quaternió $\mathfrak{q}(t + \Delta t)$ és el resultat del següent producte de quaternions:

$$\begin{aligned} \mathfrak{q}(t + \Delta t) &= \mathfrak{q}(t) \otimes \begin{bmatrix} \cos(\vec{\omega}(t)\Delta t\,/\,2) \\ \sin(\vec{\omega}(t)\Delta t\,/\,2)\mathbf{e}(t) \end{bmatrix} \\ &= \cos(\vec{\omega}(t)\,/\,2)\mathfrak{q}(t) + \sin(\vec{\omega}\,/\,2) \begin{bmatrix} -\mathbf{q}^T\mathbf{e} \\ q_0\mathbf{e} + \mathbf{q} \times \mathbf{e} \end{bmatrix} \\ &= c(t)\mathfrak{q}(t) + \frac{1}{2}s(t)\mathfrak{q}(t) \otimes \mathfrak{w}(t)\Delta t \end{aligned}$$

on $c(t) = \cos(\vec{\omega}\Delta t/2)$, $s(t) = \frac{\sin(\vec{\omega}(t)\Delta t/2)}{\vec{\omega}(t)\Delta t/2}$, i $\mathfrak{w} = |\vec{\omega}|\mathbf{e} = \begin{bmatrix} 0 \\ \boldsymbol{\omega} \end{bmatrix}$ és el quaternió vectorial de la velocitat angular $\vec{\omega}$ en les coordenades del cos.
Construïm ara el quocient:

$$\frac{\mathfrak{q}(t+\Delta t) - \mathfrak{q}(t)}{\Delta t} = \frac{c-1}{\Delta t}\mathfrak{q}(t) + \frac{1}{2}s\,\mathfrak{q}(t) \otimes \mathfrak{w}(t)$$

que quan $\Delta t \to 0$ convergeix cap a l'*equació d'espai d'estats*

$$\dot{\mathfrak{q}}(t) = \frac{1}{2}\mathfrak{q}(t) \otimes \mathfrak{w}(t) \tag{3.14}$$

amb $\mathfrak{q}(0) = q_0 \in \mathbb{R}^4$, $|q_0| = 1$.

3.3.6 Equació diferencial per a l'actitud

Tenint en compte les Equacions (3.13) i (3.14), obtenim l'equació de l'actitud del sòlid rígid en el sistema de referència $\mathcal{B}$:

$$\begin{aligned} \dot{\mathfrak{q}}(t) &= \frac{1}{2}\mathfrak{q}(t) \otimes \mathfrak{w}(t) = \frac{1}{2}\mathfrak{q}(t) \otimes \begin{bmatrix} 0 \\ \boldsymbol{\omega}(t) \end{bmatrix}, \quad \mathfrak{q}(0) = q_0 \in \mathbb{R}^4,\ |q_0| = 1 \\ \dot{\boldsymbol{\omega}}(t) &= -J^{-1}\,\boldsymbol{\omega} \times J\,\boldsymbol{\omega}(t) + J^{-1}(\mathbf{M}_u + \mathbf{M}_d), \quad \boldsymbol{\omega}(0) = \omega_0 \in \mathbb{R}^3 \end{aligned} \tag{3.15}$$

on el torque $\mathbf{M}_c$ ha estat descomposat en la pertorbació $\mathbf{M}_d$ i la maniobra $\mathbf{M}_u$, que usarem per controlar l'actitud. Noti's que el torque corresponent a la pertorbació ha estat escrit com a funció no només del temps t, sinó també de $\mathfrak{q}$, ja que les pertorbacions produïdes per l'ambient en que es troba el sòlid rígid poden variar, en alguns contexts, segons l'orientació.

4

Estudi del control

Havent trobat la forma de descriure amb precisió l'actitud del nostre satèl·lit com a sòlid rígid, ens preguntem ara quines eines podem utilitzar per controlar la seva orientació. El control d'actitud d'un sòlid rígid té com a objectiu trobar la maniobra, o seqüència de maniobres, que permetin alinear el sistema de referència del cos $\mathcal{B} = \{C, \vec{b}_1, \vec{b}_2, \vec{b}_3\}$ amb un sistema de referència *destí* $\mathcal{R} = \{C, \vec{r}_1, \vec{r}_2, \vec{r}_3\}$. Un alineament perfecte és impossible a la pràctica per varies raons, entre elles el desconeixement precís dels dos sistemes de referència, imperfeccions en els actuadors, etc. Per aquesta raó a la pràctica l'objectiu del control d'actitud és la *reducció* de l'error, que informalment es defineix com el desalineament entre l'orientació desitjada i l'orientació mesurada pels sensors del sòlid rígid.

La Secció 4.1 d'aquest capítol, on introduïrem els diferents tipus d'error i deduïrem les equacions diferencials que els governen, es basa en els apartats **6.5.1**, **6.5.2** i **7.3.1** de [3]. Per la Secció 4.2 ens basarem en la teoria de la Secció 13.3.4 de [3] per les definicions de funció de Lyapunov i en el Capítol 3, Seccions **3.1**, **3.2** i **3.3** de [4] per la resta. L'exemple que apareix al final del capítol està extret de la pàgina 313 de [3], i és també el que implementarem al Capítol 4 (amb algunes lleugeres variacions) per tal de testejar l'entorn de simulació virtual.

4.1 Errors

L'*error vertader de seguiment d'actitud* es defineix com el quaternió $\mathfrak{e}_b^r$ (o, equivalentment, la matriu de rotació E_b^r) que ens permet alinear el sistema de referència final $\mathcal{R}$ amb el sistema de

referència del cos $\mathcal{B}$. Utilitzant un tercer sistema de referència $\mathcal{E} = \{C, \vec{e}_1, \vec{e}_2, \vec{e}_3\}$, i els quaternions $\mathfrak{q}_b^e$ i $\mathfrak{q}_e^r$ que transformen $\mathcal{B}$ en $\mathcal{E}$ i $\mathcal{E}$ en $\mathcal{R}$ respectivament, aquest error s'expressa com:

$$\mathfrak{e}_b^r = (\mathfrak{q}_r^e)^{-1} \otimes \mathfrak{q}_b^e = \mathfrak{q}_e^r \otimes \mathfrak{q}_b^e \ . \tag{4.1}$$

Nosaltres utilitzarem les següents notacions simplificades:

$$\widetilde{\mathfrak{q}}_r = \mathfrak{e}_b^r, \quad \mathfrak{q} = \mathfrak{q}_r^e \quad \mathfrak{q} = \mathfrak{q}_b^e \ .$$

Aleshores, podem reescriure l'equació (4.1) com:

$$\widetilde{\mathfrak{q}}_r = \mathfrak{q}_r^{-1} \otimes \mathfrak{q} \ . \tag{4.2}$$

Anomenem a $\mathfrak{q}_r$ el quaternió de *referència* i $\mathfrak{q}$ el quaternió real del sòlid rígid. Intuïtivament podem pensar aquesta última equació com la diferència entre l'orientació real del vehicle espacial i l'orientació desitjada.

Cal tenir en compte que, a la pràctica, no és possible conèixer $\mathfrak{q}$ amb total exactitud: els sensors del vehicle espacial que mesuren l'orientació sempre tenen imperfeccions, l'ordinador té precisió finita, etc. Per aquesta raó haurem d'introduir dos tipus d'error addicionals: $\breve{\mathfrak{q}}$ i $\widehat{q}$.

El primer, $\breve{\mathfrak{q}}$, s'anomena l'*orientació mesurada* i representa el valor de $\mathfrak{q}$ mesurat pels sensors del vehicle espacial en un cert instant t. Al ser una mesura puntual i sense refinar, el valor de $\breve{\mathfrak{q}}$ serà molt sensible al soroll i a les imperfeccions dels sensors. Per aquesta raó introduïrem també $\widehat{\mathfrak{q}}$, que serà la predicció de l'orientació del vehicle obtinguda pels seus sensors.
Definim ara l'error de predicció $\widetilde{\mathfrak{q}}$ com:

$$\widetilde{\mathfrak{q}} = \widehat{\mathfrak{q}}^{-1} \otimes \mathfrak{q} \ , \tag{4.3}$$

i l'error de model com:

$$\widetilde{\mathfrak{q}}_m = \mathfrak{q}^{-1} \otimes \breve{\mathfrak{q}} \ . \tag{4.4}$$

Tots els errors introduïts fins ara $(\widetilde{\mathfrak{q}}_r, \widetilde{\mathfrak{q}}, \widetilde{\mathfrak{q}}_m)$ s'anomenen *errors no-mesurables*, ja que depenen del quaternió real $\mathfrak{q}$, que mai podrem conèixer amb precisió (llevat d'entorns de simulació virtual). En contraposició a aquests errors, introduïm també els *errors mesurables* en els quals es substitueix el valor real $\mathfrak{q}$ per $\breve{\mathfrak{q}}$ o $\widehat{\mathfrak{q}}$. Els errors mesurables són:

- L'error de model mesurat: $\mathfrak{e}_m = \widehat{\mathfrak{q}}^{-1} \otimes \breve{\mathfrak{q}}$. Notem que és equivalent a $\widetilde{\mathfrak{q}}_m$, però usant la predicció $\widehat{\mathfrak{q}}$ en lloc de $\mathfrak{q}$.

- L'error de seguiment mesurat: $\mathfrak{e}_r = \mathfrak{q}_r^{-1} \otimes \widehat{\mathfrak{q}}$. Aquest error és equivalent a $\widetilde{\mathfrak{q}}_r$, però usant $\widehat{\mathfrak{q}}$ en lloc de $\mathfrak{q}$.

- L'error clàssic de control: $\mathfrak{e} = \mathfrak{q}_r^{-1} \otimes \breve{\mathfrak{q}}$.

Tota la teoria que apareix en aquest capítol la desenvoluparem en el cas ideal estudiant els errors no-mesurables. Malgrat això cal tenir en compte que, a la pràctica, quan es construeix un controlador s'hauran d'utilitzar els errors mesurables corresponents.

4.1.1 Equació diferencial de l'error

En el Capítol 2 hem deduït l'equació diferencial que regeix els canvis en l'actitud del vehicle espacial. En aquesta secció ens preguntem si és possible trobar una equació similar per a l'error $\widetilde{\mathfrak{q}}_r$, la qual ens permeti saber com varia en relació al temps (el procediment serà anàleg per als altres errors no-mesurables).

Hem vist que $\widetilde{\mathfrak{q}}_r$ és el quaternió que rota el sistema de referència destí $\mathcal{R}$ i l'alinea amb el sistema de referència del cos $\mathcal{B}$. Introduïm ara la velocitat angular de referència $\boldsymbol{\omega}_r$ (expressada en el sistema $\mathcal{R}$) i definim l'error en la velocitat angular $\widetilde{\boldsymbol{\omega}}$ com:

$$\widetilde{\boldsymbol{\omega}} := \boldsymbol{\omega} - R(\widetilde{\mathfrak{q}}_r)\boldsymbol{\omega}_r = \boldsymbol{\omega} - R_b^r\,\boldsymbol{\omega}_r := \boldsymbol{\omega} - \boldsymbol{\omega}_{rb}\ . \tag{4.5}$$

Derivant ara l'equació (4.2) respecte al temps, i tenint en compte l'equació (3.14), obtenim:

$$\begin{aligned}
\dot{\widetilde{\mathfrak{q}}}_r &= \dot{\mathfrak{q}}_r^{-1} \otimes \mathfrak{q} + \mathfrak{q}_r^{-1} \otimes \dot{\mathfrak{q}} \\
&= -\frac{1}{2}\mathfrak{w}_r \otimes \mathfrak{q}_r^{-1} \otimes \mathfrak{q} + \frac{1}{2}\mathfrak{q}_r^{-1} \otimes \mathfrak{q} \otimes \mathfrak{w} \\
&= -\frac{1}{2}\mathfrak{w}_r \otimes \widetilde{\mathfrak{q}}_r + \frac{1}{2}\widetilde{\mathfrak{q}}_r \otimes \mathfrak{w}\ ,
\end{aligned}$$

on $\mathfrak{w} = [0, \boldsymbol{\omega}]^T$ i $\mathfrak{w}_r = [0, \boldsymbol{\omega}_r]^T$. Desenvolupant ara el producte de l'última expressió, obtenim:

$$\begin{aligned}
\dot{\widetilde{\mathfrak{q}}}_r &= \frac{1}{2}\begin{bmatrix} -\widetilde{\mathbf{q}}_r\boldsymbol{\omega} \\ \widetilde{q}_{r0}\boldsymbol{\omega} + \widetilde{\mathbf{q}}_r \times \boldsymbol{\omega} \end{bmatrix} - \frac{1}{2}\begin{bmatrix} -\widetilde{\mathbf{q}}_r\boldsymbol{\omega}_{rb} \\ \boldsymbol{\omega}_{rb}\widetilde{q}_{r0} + \boldsymbol{\omega}_{rb} \times \widetilde{\mathbf{q}}_r \end{bmatrix} \\
&= \frac{1}{2}\begin{bmatrix} -\widetilde{\mathbf{q}}_r(\boldsymbol{\omega} - \boldsymbol{\omega}_{rb}) \\ \widetilde{q}_{r0}(\boldsymbol{\omega} - \boldsymbol{\omega}_{rb}) + \widetilde{\mathbf{q}}_r \times (\boldsymbol{\omega} - \boldsymbol{\omega}_{rb}) \end{bmatrix} = \frac{1}{2}\begin{pmatrix} -\widetilde{\mathbf{q}}_r\widetilde{\boldsymbol{\omega}}_r \\ \widetilde{q}_{r0}\widetilde{\boldsymbol{\omega}}_r + \widetilde{\mathbf{q}}_r\widetilde{\boldsymbol{\omega}}_r \end{pmatrix} \\
&= \frac{1}{2}\widetilde{\mathfrak{q}}_r \otimes \widetilde{\mathfrak{w}}_r\ ,
\end{aligned}$$

(on, evidentment, $\widetilde{\mathfrak{w}}_r = [0, \widetilde{\boldsymbol{\omega}}_r]$). Derivem ara $\widetilde{\boldsymbol{\omega}}_r$ respecte al temps:

$$\begin{aligned}\dot{\widetilde{\boldsymbol{\omega}}}_r &= \dot{\boldsymbol{\omega}} - \frac{d\, R_b^r\, \boldsymbol{\omega}_r}{dt} \\ &= -J^{-1}(\boldsymbol{\omega}_{rb} + \widetilde{\boldsymbol{\omega}}_r) \times J(\boldsymbol{\omega}_{rb} + \widetilde{\boldsymbol{\omega}}_r) + J^{-1}(\mathbf{M}_u + \mathbf{M}_d(\mathfrak{q}, t)) - \boldsymbol{\omega}_{rb} \times \widetilde{\boldsymbol{\omega}}_r - \dot{\boldsymbol{\omega}}_{rb}\ .\end{aligned}$$

Si definim $J_{123} := diag(J_{21} + J_{31} - J_{11}, J_{12} + J_{32} - J_{22}, J_{13} + J_{23} - J_{33})$, $A_r := -J_{123}\boldsymbol{\omega}_{rb}\times$ i $\mathbf{M}_r := J^{-1} \times \boldsymbol{\omega}_{rb} \times J\boldsymbol{\omega}_{rb} + \dot{\boldsymbol{\omega}}_{rb}$, podem escriure l'equació anterior com:

$$\dot{\widetilde{\boldsymbol{\omega}}}_r = -J^{-1}\widetilde{\boldsymbol{\omega}}_r \times J\widetilde{\boldsymbol{\omega}}_r + J^{-1}A_r\widetilde{\boldsymbol{\omega}}_r + J^{-1}\mathbf{M}_u + J^{-1}(\mathbf{M}_d(\mathfrak{q}, t) - \mathbf{M}_r)\ . \tag{4.6}$$

Així doncs, *l'equació diferencial de l'error* serà:

$$\dot{\widetilde{\mathfrak{q}}}_r = \frac{1}{2}\widetilde{\mathfrak{q}}_r \otimes \widetilde{\mathfrak{w}}_r \tag{4.7}$$

$$\dot{\widetilde{\boldsymbol{\omega}}}_r = -J^{-1}\widetilde{\boldsymbol{\omega}}_r \times J\widetilde{\boldsymbol{\omega}}_r + J^{-1}A_r\widetilde{\boldsymbol{\omega}}_r + J^{-1}\mathbf{M}_u + J^{-1}(\mathbf{M}_d(\mathfrak{q}, t) - \mathbf{M}_r)\ .$$

4.2 Controladors

Un controlador és un procediment que s'utilitza per conduir un sistema dinàmic a un punt d'equilibri desitjat (normalment, l'orígen). Suposem que tenim un sistema dinàmic de la forma

$$\dot{x}(t) = f(x(t), u(t)) \tag{4.8}$$

a on:

- Els vectors $x(t) \in \mathbb{R}^n$, $t \geq 0$ representen l'estat del sistema en l'instant t.
- Els vectors $u(t) \in U$ són els *valors d'entrada* o *controls*, la funció dels quals és conduir el sistema. U és un espai mètric amb distància d dins el qual les boles $\{v : d(\mu, v) \leq r\}$ són compactes $\forall v \in U$.
- $f : \Omega \subset \mathbb{R}^n \times U \longrightarrow \mathbb{R}^n$ és una funció (localment) Lipschitz en les variables (x, u).

Suposem també que existeix com a mínim un punt d'equilibri x_e tal que $f(x_e, 0) = 0$; l'objectiu del control és escollir una funció $u \in U$ tal que, si l'estat inicial és "suficientment pròxim" a aquesta posició d'equilibri x_e, la solució del sistema hi convergeixi quan $t \to \infty$. Aquesta propietat de "convergir a l'equilibri si hi comencem prou a prop" s'anomena estabilitat asimptòtica. La seva definició formal és la que segueix:

Definició. (Estabilitat de Lyapunov i asimptòtica)
Considerem una equació diferencial autònoma de la forma

$$\dot{x} = f(x), \quad x \in \Omega \subset \mathbb{R}^n, \tag{4.9}$$

on f és un camp vectorial n-diferenciable (amb $n \geq 3$) en un domini Ω. Suposem que existeix una posició d'equilibri $x_e \in \Omega$ tal que $f(x_e) = 0$.
Direm que x_e és una posició d'equilibri *Lyapunov-estable* si $\forall \epsilon > 0,\ \exists \delta_\epsilon > 0$, independent del temps t, tal que $\forall x_0 \in \Omega$ amb $||x_0 - x_e|| < \delta_\epsilon$, la solució de l'equació ϕ amb condició inicial $\phi(0) = x_0$ compleix que $\forall t > 0$ està definida i $||\phi(t) - x_e|| < \epsilon$.
Si, a més, $\lim_{t \to +\infty} \phi(t) = x_e$, direm que x_e és *assimptòticament estable*.

Així doncs, l'objectiu del control és trobar una funció $u(t)$ adequada que converteixi el sistema en asimptòticament estable al voltant d'un punt fix x_e al que ens interessi convergir.

Ens preguntem ara quines eines podem utilitzar que ens ajudin a demostrar l'estabilitat asimptòtica d'un punt d'equilibri. Per simplicitat, suposarem que aquest punt d'equilibri és l'orígen; no hi ha pèrdua de generalitat en fer tal suposició, donat que qualsevol punt d'equilibri pot ser convertit en l'orígen fent una translació pertinent. Una forma de demostrar l'estabilitat asimptòtica és mitjançant *funcions de Lyapunov*.

Definició. (Funció definida positiva)
Una funció $V : \mathbb{R}^n \to \mathbb{R}$ és *localment definida positiva* si, en un cert entorn $\mathcal{X}$ de l'orígen, es té

$$V(0) = 0, \quad V(\mathbf{x}) > 0 \quad \forall \mathbf{x} \in \mathcal{X},\ \mathbf{x} \neq 0. \tag{4.10}$$

Si $\mathcal{X} = \mathbb{R}^n$, llavors diem que V és *globalment definida positiva* (o simplement *definida positiva*). Si es té $V(\mathbf{x}) \geq 0 \quad \forall \mathbf{x} \in \mathcal{X},\ \mathbf{x} \neq 0$, direm que la funció és *semidefinida positiva*.

Definició. (Funció definida negativa)
Una funció $V : \mathbb{R}^n \to \mathbb{R}$ s'anomena *negativa definida* si $-V$ és positiva definida (resp. amb *negativa semidefinida*).

Definició. (Funció de Lyapunov)
Direm que una funció $V : \mathbb{R}^n \to \mathbb{R}$ és una *Funció de Lyapunov* per un sistema com el de l'equació (4.9) si en un entorn $\mathcal{X}$ de l'orígen es té:

- V és (localment) definida positiva i de classe $\mathcal{C}^1$.

- $\dot{V}$ és (localment) negativa semidefinida.

Teorema 4.2.1. ***(d'estabilitat local de Lyapunov)*** *Si el sistema de l'equació* (4.9) *admet una funció de Lyapunov V en algun entorn $\mathcal{X}$ de l'orígen i $\dot{V}$ és negativa definida, llavors $x_e = 0$ és asimptòticament estable.*

Teorema 4.2.2. ***(d'estabilitat asimptòtica global de Lyapunov)*** *Suposem que l'equació* (4.9) *admet una funció de Lyapunov V en tot $\mathbb{R}^n$, amb $\dot{V}$ negativa definida, i $V(x) \to \infty$ quan $|x| \to \infty$. Llavors, el punt d'equilibri $x_e = 0$ és asimptòticament estable globalment.*

Demostració. Vegeu [4] pàgina 47, Teorema 3.3 per a la demostració d'aquests dos teoremes. □

Els teoremes d'estabilitat local/global de Lyapunov estan limitats en el cas que $\dot{V}$ sigui una funció definida negativa, però no funciona en el cas més lax en el qual només sigui semidefinida negativa. En aquest cas, haurem d'utilitzar el *teorema d'invariància de LaSalle* (també anomenat *principi d'invariància de LaSalle*).

Teorema 4.2.3. ***(Principi d'invariància de LaSalle)*** *Considerem el sistema de l'equació* (4.9) *i suposem que existeix una funció de Lyapunov V definida sobre Ω. Sigui $\mathcal{V} = \{x \in \Omega | \dot{V}(x) = 0\}$, $\mathcal{S}$ el conjunt invariant més gran tal que $\mathcal{S} \subset \mathcal{V}$. Si $\mathcal{S}$ només conté els punts d'equilibri del sistema, llavors aquests són asimptòticament estables.*

Demostració. Vegeu [4] Teorema 3.5, pàgina 57. □

4.3 Aplicació al control d'actitud

En el context del control d'actitud de satèl·lits, el sistema dinàmic a estabilitzar ve donat per l'equació (4.7), que té dos punts d'equilibri en $\{\widetilde{\mathfrak{q}}_e = [\pm 1, \vec{0}], \widetilde{\boldsymbol{\omega}}_e = \vec{0}\}$ corresponents a l'error nul. Els actuadors de que està dotat el satèl·lit artificial permeten generar un cert moment; per tant, el controlador a buscar serà una expressió pel torque de control $\mathbf{M}_u$ que faci que els dos punts d'equilibri es converteixin en asimptòticament estables.

Cal notar que la manera concreta de produir aquest torque mitjançant actuadors (normalment motors d'impuls feble) queda fora de l'abast d'aquest treball, ja que depèn en gran mesura de la forma com es construeixi el satèl·lit i l'equipament de que estigui dotat.

Adoptem la següent llei de control per al moment $\mathbf{M}_u$:

$$\mathbf{M}_u = \mathbf{M}_r - k_q\, J^{-1} \overline{K}_q sgn(\widetilde{q}_{r0}\widetilde{\mathbf{q}}_r - J(K_\omega - J^{-1}A_r)\widetilde{\boldsymbol{\omega}}_r + (\widetilde{\boldsymbol{\omega}}_r \times J\widetilde{\boldsymbol{\omega}}_r - \mathbf{M}_d(\mathfrak{q}, t)), \quad (4.11)$$

on $k_q, > 0$ és un escalar positiu i $\overline{K}_q$, K_ω són matrius definides positives (a determinar). Si substituïm ara aquesta expressió a l'equació (4.7), obtenim la següent equació diferencial autònoma:

$$\begin{aligned}\dot{\tilde{\mathfrak{q}}}_r(t) &= \frac{1}{2}\tilde{\mathfrak{q}}_r \otimes \tilde{\mathfrak{w}}_r = f_1(\tilde{\mathfrak{q}}_r, \tilde{\boldsymbol{\omega}}_r),\\ \dot{\tilde{\boldsymbol{\omega}}}_r(t) &= -k_q\overline{K_q}sgn(\tilde{q}_{r0})\tilde{\mathbf{q}}_r - K_\omega\tilde{\boldsymbol{\omega}}_r = f_2(\mathfrak{q}_r, \tilde{\boldsymbol{\omega}}_r)\ .\end{aligned}$$

Fixem-nos, en primer lloc, que la funció $f = (f_1, f_2)$ és localment Lipschitz en $(\tilde{\mathfrak{q}}, \tilde{\boldsymbol{\omega}})$ (per ser de classe $\mathcal{C}^1$ sobre aquestes variables) i que continua tenint els dos zeros $\{\tilde{\mathfrak{q}}_e = [\pm 1, 0], \tilde{\boldsymbol{\omega}}_e = 0\}$ (corresponents a l'error nul). Per demostrar la seva estabilitat asimptòtica proposem la següent candidata a funció de Lyapunov:

$$V = 2k_q(1 - |\tilde{q}_0|) + \frac{1}{2}\tilde{\boldsymbol{\omega}}_r^T\overline{K}_q^{-1}\tilde{\boldsymbol{\omega}}_r\,. \tag{4.12}$$

Observem que V és positiva a tots els punts llevat els d'equilibri (on és zero). Cal demostrar ara que la seva derivada és semidefinida negativa:

$$\begin{aligned}\dot{V} &= -2k_q sgn(\tilde{q}_{r0})\dot{\tilde{q}}_{r0} + \tilde{\boldsymbol{\omega}}_r^T\overline{K}_q^{-1}\dot{\tilde{\boldsymbol{\omega}}}_r\\ &= k_q sgn(\tilde{q}_{r0})\tilde{\mathbf{q}}_r^T\tilde{\boldsymbol{\omega}}_r + \tilde{\boldsymbol{\omega}}_r^T\overline{K}_q^{-1}(-k_q\overline{K}_q sgn(\tilde{q}_{r0})\tilde{\mathbf{q}}_r - K_\omega\tilde{\boldsymbol{\omega}}_r)\\ &= -\tilde{\boldsymbol{\omega}}_r^T\overline{K}_q^{-1}K_\omega\tilde{\boldsymbol{\omega}}_r \le 0\ .\end{aligned} \tag{4.13}$$

Notem que $\dot{V} = 0$ si i només si $\tilde{\boldsymbol{\omega}}_r = 0$. Definim $\mathcal{V} := \{(\tilde{\mathfrak{q}}, \tilde{\boldsymbol{\omega}}) : \dot{V}(\tilde{\mathfrak{q}}, \tilde{\boldsymbol{\omega}}) = 0\} = \{(\tilde{\mathfrak{q}}, \tilde{\boldsymbol{\omega}}) : \tilde{\boldsymbol{\omega}}_r = 0\}$ i observem que els únics elements de $\mathcal{V}$ que compleixen la condició $\{\dot{\tilde{\mathfrak{q}}}_r = 0, \dot{\tilde{\boldsymbol{\omega}}}_r = 0\}$ són $([\pm 1, \vec{0}], \vec{0})$, els quals són també els dos *únics* punts d'equilibri que té el sistema. Per tant, són asimptòticament estables d'acord amb el principi d'invariància de LaSalle.

Això ens demostra que el control que hem elegit per $\mathbf{M}_u$ és adequat, ja que converteix el sistema (4.7) en un sistema asimptòticament estable al voltant dels punts d'equilibri corresponents a l'error nul.

5

Construcció del simulador virtual

Aquest capítol té dos objectius ben diferenciats. El primer és detallar com s'ha realitzat la construcció d'un entorn de simulació que permet estudiar els canvis d'actitud d'un satèl·lit virtual quan és sotmès a l'acció de diferents moments de força. En segon lloc, explicar la implementació del controlador vist al final del capítol anterior, que rebi l'actitud actual del satèl·lit com a entrada i calculi el torque que cal realitzar per estabilitzar-lo.

En aquest capítol es discuteix, de forma general, quins són els mètodes i algorismes que s'han utilitzat per programar l'entorn de simulació i el controlador. També s'analitzen els resultats obtinguts en alguns experiments realitzats per provar el bon funcionament dels programes. El codi amb la implementació exacta del simulador i totes les seves funcions es troba a l'Apèndix B. Les figures amb els resultats que es discuteixen al llarg del capítol es troben a l'Apèndix A.

La major part del contingut d'aquest capítol és d'elaboració pròpia, però s'han consultat algunes referències. Per la implementació del mètode de Runge-Kutta-Fehlberg 45 s'ha consultat el llibre [7], secció *9.5* . Per la implementació del generador de nombres aleatoris s'ha consultat el Capítol *7* de [8].

5.1 Esquema del programa de simulació

El programa que ens permet simular els canvis d'actitud del satèl·lit artificial consta de dos elements ben diferenciats. En primer lloc, tenim l'entorn de simulació que rebrà com a input un vector de $\mathbb{R}^3$ corresponent al torque del controlador amb les pertorbacions i integrarà numèricament l'equació (3.15) durant un interval de temps, donant com a resultat la nova

actitud del satèl·lit. A més, l'entorn de simulació serà l'encarregat de definir les condicions inicials del simulador: l'actitud inicial, l'actitud de referència i el tensor d'inèrcia.

En segon lloc, tenim el controlador en sí, que rebrà l'actitud actual del satèl·lit i calcularà el torque necessari per conduir-lo cap a l'actitud de referència. A més, a cada torque calculat pel controlador li sumarem un vector de pertorbacions que representen les distorsions ambientals i les imperfeccions dels actuadors.

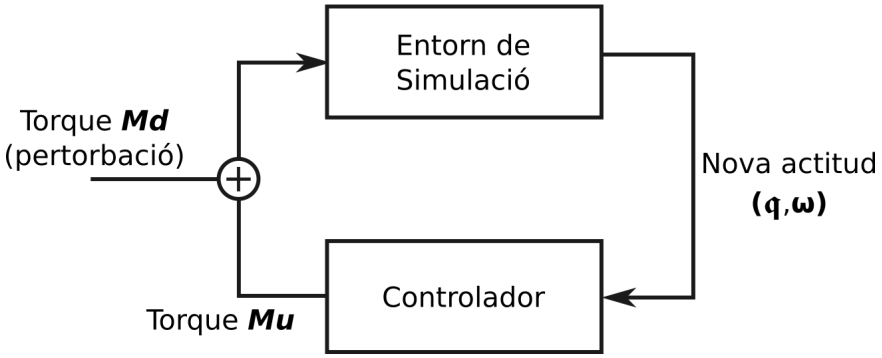


Figura 5.1: Esquema de funcionament del programa de simulació

Vegem ara el funcionament de cada una d'aquestes parts del programa.

5.2 Entorn de simulació

El primer pas per construir un entorn de simulació per al nostre satèl·lit consisteix en implementar un mètode per integrar numèricament l'equació (3.15) i així poder conèixer l'actitud del satèl·lit a cada instant de temps. Per a les primeres simulacions es va usar el mètode d'Euler per la seva senzillesa i rapidesa, però a causa de la seva poca precisió es va optar per implementar i usar el mètode *Runge-Kutta-Fehlberg 45* (sovint abreviat *RKF45*).

5.2.1 Integrador numèric (RKF-45)

El mètode de Runge-Kutta-Fehlberg 45 és un mètode per propagar numèricament condicions inicials associades a una equació diferencial i, a la vegada, controlar que a cada pas del mètode s'estigui usant un pas d'integració adequat que permeti controlar l'error. Per aconseguir-ho, el mètode calcula dues aproximacions de la solució a cada pas usant els mètodes de Runge-Kutta d'ordres 4 i 5. Si la diferència entre aquestes dues aproximacions està per sota una tolerància

predefinida, s'accepta l'aproximació de Runge-Kutta 5 com a vàlida i es procedeix a la següent iteració. En cas contrari, es redueix la distància de pas i es torna a calcular l'aproximació.

Suposem, doncs, que volem integrar un sistema de la forma $\dot{x} = f(t, x)$. Partint d'un x_0 inicial i, establint una distància de pas h, cada iteració del mètode RKF45 calcularà els valors k_i següents:

$$\begin{aligned}
k_1 &= h \cdot f(t_n, x_n), \\
k_2 &= h \cdot f\left(t_n + \frac{1}{4}h, x_n + \frac{1}{4}k_1\right), \\
k_3 &= h \cdot f\left(t_n + \frac{3}{8}h, x_n + \frac{3}{32}k_1 + \frac{9}{32}k_2\right), \\
k_4 &= h \cdot f\left(t_n + \frac{12}{13}h, x_n + \frac{1932}{2197}k_1 - \frac{7200}{2197}k_2 + \frac{7296}{2197}k_3\right), \\
k_5 &= h \cdot f\left(t_n + h, x_n + \frac{439}{216}k_1 - 8k_2 + \frac{3680}{513}k_3 - \frac{845}{4104}k_4\right), \\
k_6 &= h \cdot f\left(t_n + \frac{1}{2}h, x_n - \frac{8}{27}k_1 + 2k_2 - \frac{3544}{2565}k_3 + \frac{1859}{4104}k_4 - \frac{11}{40}k_5\right).
\end{aligned}$$

En el nostre cas particular, f és la funció que apareix a l'equació de l'actitud, que dependrà de t, $\mathfrak{q}$ i ω (el tensor d'inèrcia i el torque es tracten com si fóssin constants). Una vegada calculats aquests valors k_i, es calcula una aproximació de x_{n+1} usant el mètode de Runge Kutta d'ordre 4:

$$x_{n+1} = x_n + \frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4101}k_4 - \frac{1}{5}k_5\,.$$

També es calcula l'aproximació de x_{n+1} amb el mètode de Runge Kutta d'ordre 5. Per evitar confusió amb el punt calculat usant Runge-Kutta 4, li posem z_{n+1}:

$$z_{n+1} = x_n + \frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6\,.$$

Després, es comprova la distància entre x_{n+1} i z_{n+1} en alguna norma (en particular, en aquest treball s'ha usat la norma 2). Si aquesta diferència és superior a una tolerància, es rebutgen les aproximacions calculades, es divideix h entre 2 i es torna a calcular la iteració. En cas contrari, s'accepta z_{n+1} com a aproximació i s'avança a la següent iteració; en algunes ocasions, també es multiplica h entre 2. Existeixen fòrmules per calcular la distància de pas òptima en cada iteració, però queden fora de l'abast d'aquest treball i no s'han implementat.

Per assegurar el bon funcionament de l'integrador numèric, es va provar d'integrar l'equació del cercle en l'interval de temps $[0, 800]$ (unes 127 voltes aproximadament) amb una distància de pas inicial $h = 0.01$:

$$\dot{x} = y\,,$$
$$\dot{y} = -x\,,$$
$$(x_0, y_0) = (1, 0),$$

i es va comprovar que cada punt (x_n, y_n) calculat complís $x^2 + y^2 \approx 1$. Els resultats van ser satisfactoris i l'error, treballant amb precissió simple, es mantenia estable per sota de 10^{-6} sense espiralar en cap direcció.

5.2.2 Generació de nombres aleatoris

Per tal de poder simular pertorbacions en l'ambient i els actuadors, així com per poder generar infinitat de condicions inicials aleatòries que puguin ser subministrades com a input a la xarxa neuronal que s'exposa en el proper capítol, es necessari primerament tenir un algorisme per generar nombres aleatoris. Si bé és cert que el llenguatge Python, amb el que s'ha realitzat tota la part pràctica d'aquest treball, ja disposa de funcions molt robustes per generar nombres aleatoris, s'ha decidit implementar un *generador lineal de congruències* amb el qual poder generar nombres (pseudo)aleatoris.

Tal i com s'explica a la secció 5.3.2, el comportament de totes les pertorbacions s'ha modelitzat seguint una distribució normal. Per aquesta raó, es van provar de normalitzar els punts utilitzant la transformació de Box-Muller. Aquest mètode permet convertir un conjunt de punts aleatoris generats seguint una distribució uniforme en $(0, 1)$ en un altre conjunt de punts que segueix una distribució normal estàndar.

A la versió definitiva del programa del simulador (codi ***B.3.4***), els exemples es generen a partir de dues llavors aleatòries diferents: una per a les condicions inicials i l'altra per a les pertorbacions.

5.2.3 Condicions inicials

Les condicions inicials del simulador venen donades per tres elements: el tensor d'inèrcia, l'actitud inicial (el quaternió i la velocitat angular en l'instant $t = 0$) i l'actitud de referència (el quaternió i la velocitat angular desitjades).

Si bé s'hauria pogut generar un tensor d'inèrcia aleatori, totes les simulacions utilitzen el mateix tensor d'inèrcia i no hi ha variacions. Els primers prototips del programa es van realitzar amb un tensor d'inèrcia corresponent a una esfera de $83.6Kg$ i $29cm$ de radi (que són dimensions similars a les del satèl·lit *Sputnik 1*, llançat per la Unió Soviètica l'any 1958). El tensor d'inèrcia corresponent és:

$$J = \frac{2}{5} \cdot M \cdot R \cdot Id_{3\times 3}\,,$$

on $R = 0.29$ és el radi (en metres) i $M = 83.6$ és la massa (en quilos).
No obstant, el programa que s'ha inclòs junt amb la memòria utilitza un tensor d'inèrcia com el que es detalla a [6]:

$$J = \begin{pmatrix} 2.0257 & 0.6498 & 1.1226 \\ 0.6498 & 0.7998 & 0.1833 \\ 1.1226 & 0.1833 & 1.2753 \end{pmatrix}$$

Els resultats obtinguts utilitzant aquests dos tensors d'inèrcia no varien significativament (més enllà de la quantitat de torque que ha d'aplicar el controlador per estabilitzar l'actitud). Tots els experiments i resultats que es discuteixen a la Secció ***5.4*** utilitzen aquest segon tensor d'inèrcia.

Durant les primeres proves amb el programa, es va forçar manualment una actitud de referència i una actitud inicial per a cada experiment. Els resultats d'algunes d'aquestes primeres proves es poden veure a la Secció ***5.4***. Una vegada es va comprovar que el controlador que es detalla a la Secció ***5.3*** era capaç d'estabilitzar el satèl·lit davant de qualsevol condició inicial, es va modificar el codi per tal que generés condicions inicials semblants a les de l'article [6]. La idea és inicialitzar el satèl·lit amb un quaternió aleatori i una velocitat angular nul·la i després aplicar-li una pertorbació aleatòria durant un instant de temps. Això farà que la velocitat angular del satèl·lit augmenti considerablement, mentre que el quaternió es mantindrà pròxim a l'inicial. L'objectiu del controlador serà ara estabilitzar la velocitat angular del satèl·lit i reduir-la a zero.

Generar una orientació inicial a l'atzar requereix trobar una forma de generar quaternions aleatòriament. El mètode que s'ha escollit és el següent: es comença triant un valor de l'angle α entre $[-\pi, \pi]$ i es defineix $S := \sin(\alpha/2)$, $C := \cos(\alpha/2)$. Després es trien tres

nombres x, y, z a l'atzar per formar un vector (x, y, z), i es divideix pel mòdul per fer-lo unitari. Finalment, l'actitud de referència serà $q_r = [C, S \cdot x, S \cdot y, S \cdot z]$, $\omega_r = [0, 0, 0]$.

Una vegada generada l'actitud de referència, s'aplica una pertorbació (torque) aleatòria. Per generar el torque a l'atzar, es crea un vector de $\mathbb{R}^3$ on cada component és un nombre enter escollit uniformement entre -100 i 100. Després es multiplica cada una d'aquestes components per 0.01.

El quaternió i la velocitat angular resultant d'aplicar la pertorbació a l'actitud de referència durant un instant de temps serà l'actitud inicial del satèl·lit sobre la que el controlador començarà a treballar.

5.3 Controlador

5.3.1 Llei de control

La llei de control que hem implementat per al controlador és:

$$\mathbf{M}_u = \mathbf{M}_r - k_q \, J^{-1} \overline{K}_q sgn(\tilde{q}_{r0}\tilde{\mathbf{q}}_r - J(K_\omega - J^{-1}A_r)\tilde{\boldsymbol{\omega}}_r + \tilde{\boldsymbol{\omega}}_r \times J\tilde{\boldsymbol{\omega}}_r, \tag{5.1}$$

amb $k_q = 1$, i amb K_q i K_ω definides com :

$$K_q = K_\omega = \begin{pmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{pmatrix}.$$

(Es pot comprovar fàcilment que l'espectre d'aquesta matriu està format per nombres reals positius i, per tant, és una matriu definida positiva).

Notem que l'expressió de $\mathbf{M}_u$ és la mateixa que es menciona al final del Capítol 4, però sense tenir en compte el vector de pertorbacions $\mathbf{M}_d$. És a dir, el controlador només podrà actuar rebent l'error d'actitud del satèl·lit com a única informació.

5.3.2 Pertorbacions

Cada vegada que el controlador genera un torque, se li sumen dos vectors de pertorbació abans de passar-lo com a entrada a l'entorn de simulació. Un d'aquests vectors representa les imperfeccions en els actuadors, que a la pràctica no són mai capaços de produir *exactament* un torque de la magnitud desitjada. L'altre vector representa les pertorbacions ambientals, com les que es poden produïr pel fregament amb l'atmosfera (en el cas dels satèl·lits situats

en òrbites baixes), la interacció amb el camp magnètic de la Terra, l'impacte amb petits cossos, etc.

Tots dos vectors de pertorbacions es generen seguint exactament el mateix mètode: per cada component, es genera un nombre aleatori seguint una distribució $N(0,1)$, que produirà valors entre -1 i 1. Després es multiplica cada component per $1e-3$, ja que desconeixem els valors dels errors d'execució de maniobres i de les pertorbacions ambientals (que depenen de l'òrbita en la qual es mou el satèl·lit).

Al començar a escriure el codi d'aquest treball, es van intentar buscar les dades de les pertorbacions del telescopi Gaia (llançat per la ESA l'any 2013), amb la finalitat de comprovar si presentaven algun tipus de biaix que pogués ser emulat en el programa. Malauradament no va ser possible obtenir-les i és per aquesta raó que es va decidir modelitzar les pertorbacions mitjançant una distribució normal estàndar.

5.4 Discussió dels resultats obtinguts

Les primeres proves amb el simulador van consistir en comprovar que el controlador, efectivament, era capaç d'estabilitzar l'actitud davant de pràcticament qualsevol condició inicial. Primerament es va estudiar el comportament del controlador en absència de pertorbacions, introduint a mà l'actitud inicial, la de referència i fent actuar el controlador durant 30 segons. Les figures A.1 i A.2 mostren el comportament de les tres components del torque de control i l'error en l'actitud (en norma infinit) al llarg de 30 segons. En aquest cas particular, l'actitud inicial és

$$\mathfrak{q}_0 = [1,0,0,0], \quad \omega_0 = [0.1,0.1,0.1]\,,$$

i la actitud de referència és

$$\mathfrak{q}_r = [0.2346654, 0.4693307, 0.2346654, 0.8182866], \quad \omega_r = [0,0,0]\,.$$

El comportament que s'observa a la figura A.1 és l'habitual en molts controladors (vegeu [3] seccions **7.6** i **7.7.4**). Com es pot veure, el controlador tendeix a excedir-se quan comença a estabilitzar el satèl·lit, i després convergeix cap a zero. Aquest comportament d'*overshoot* inicial, present en d'altres procediments de control, resulta més evident si s'analitza un cas

on la diferència entre l'actitud inicial i l'actitud de referència (especialment en el vector de velocitat angular) sigui més extrema: a la figura A.3 podem observar el comportament del controlador amb les condicions inicials

$$\begin{aligned}
\mathfrak{q}_r &= [0.20277351, -0.57919723, -0.5306368, -0.58466919]\,, \\
\omega_r &= [0, 0, 0]\,, \\
\mathfrak{q}_0 &= [0.22905382, -0.5893222, -0.46071562, -0.62287625]\,, \\
\omega_0 &= [-1.95014915, 1.71295657, 2.15687746]\,.
\end{aligned}$$

Les següents proves que es discuteixen han estat realitzades amb la versió definitiva del controlador, generant condicions inicials a l'atzar i afegint pertorbacions al torque produït pel controlador. A més del vector d'actitud inicial i de referència de cada exemple també s'inclou la llavor que s'ha utilitzat per generar l'exemple i les pertorbacions.

Les figures A.4 i A.5 mostren el comportament del controlador i l'error de velocitat angular a l'exemple generat per les llavors 123 (per a les condicions inicials) i 1 (per a les pertorbacions). La llavor per a les condicions inicials es correspon als següents vectors d'actitud:

$$\begin{aligned}
\mathfrak{q}_r &= [0.67638755, -0.28871214, -0.39423871, -0.5511089]\,, \\
\omega_r &= [0, 0, 0]\,, \\
\mathfrak{q}_0 &= [0.67637345, -0.288869558, -0.39426377, -0.55111695]\,, \\
\omega_0 &= [0.00710342, -0.00442388, -0.01079256]\,.
\end{aligned}$$

Com es pot veure, el controlador és capaç d'estabilitzar l'actitud fins i tot davant la presència de pertorbacions. Noti's que a la figura A.5 no hem inclòs el l'error en el quaternió, ja que la diferència entre l'orientació inicial i la desitjada serà molt petita.

Una pregunta que podem fer-nos és si el controlador és capaç d'estabilitzar correctament la velocitat angular si limitem el torque que poden produir els actuadors del satèl·lit. Les figures A.6 i A.7 mostren el comportament del controlador i l'error davant el mateix exemple, però limitant el torque dels actuadors a un màxim de $\pm 0.0021\, N \cdot m$. Observem que el controlador té més dificultats per reduir l'error (de fet, creix considerablement al voltant dels 4-5 segons de simulació), però al final és possible acabar estabilitzant l'actitud.

En canvi, les figures A.8 i A.9 mostren el comportament del controlador i l'error quan el torque que poden produir els actuadors es limita a un màxim de $\pm 0.001\, N \cdot m$, el mateix valor màxim que poden prendre les pertorbacions. En aquest cas, veiem que el controlador és totalment incapaç d'estabilitzar la velocitat angular i tant l'error com el torque generat oscil·len de forma descontrolada.

6

Aprenentatge per reforç

L'aprenentatge per reforç és una àrea de l'aprenentatge automàtic que busca estratègies per aconseguir agents intel·ligents capaços de realitzar accions sobre un entorn. Aquestes accions han de ser escollides curosament per l'agent per tal de poder maximitzar una certa funció de recompensa que dependrà del problema en qüestió. En el nostre cas particular, l'agent serà el controlador del satèl·lit artificial, l'entorn serà el simulador virtual i les recompenses a maximitzar vindran donades de forma inversament proporcional a l'error en l'actitud.

L'objectiu original d'aquest capítol era explicar la implementació d'un algorisme de control semblant al que es detalla a [6] i discutir els resultats obtinguts al sotmetre el controlador a diferents proves. A causa de la gran dimensió de la xarxa neuronal a entrenar, de la relativa complexitat de l'entorn de simulació i de les característiques dels processadors dels quals s'ha disposat, no ha estat possible entrenar l'agent en un temps raonable i obtenir així un controlador que reprodueixi els resultats de [6]. En lloc d'això, s'ha optat per simplificar el problema (limitant les condicions inicials i utilitzant un mètode d'integració menys precís, però més ràpid) per tal d'obtenir un controlador que, si bé no pot tenir utilitat pràctica, pugui demostrar que l'algorisme implementat podria assolir el seu objectiu si es disposés de suficients recursos per a l'entrenament.

La major part del contingut d'aquest capítol està basat en [6]. Les nocions teòriques sobre processos de decisió de Markov, aprenentatge per reforç i Q-learning estan extretes de les seccions **17.1** i **21.3** de [5].

A causa del límit d'extensió d'aquest treball no s'explicarà com funcionen en general les neurones artificials, les xarxes neuronals multicapa ni els algorismes d'optimització. Una explicació complerta d'aquests conceptes es pot trobar a la secció **20.5** de [5].

6.1 Q-Learning

6.1.1 Agents intel·ligents i procesos de decisió de Markov

En l'àmbit de la intel·ligència artificial s'anomena *agent intel·ligent* a una entitat autònoma (un robot, un *softbot*, una persona, etc) capaç de percebre informació d'un entorn, processar-la i realitzar les accions adequades per assolir una certa *meta*. En l'aprenentatge automàtic i, més concretament, en l'aprenentatge per reforç, es tracta de dissenyar estratègies que permetin que l'agent sigui capaç d'aprendre i utilitzar el seu coneixement d'experiències passades per assolir aquesta meta.

Abans de parlar amb més detall sobre l'aprenentatge per reforç, cal introduir el concepte de *problema de decisió de Markov* (a vegades anomenat *procès* de decisió de Markov).

Definició. (Problema de decisió de Markov)
Un *Problema de decisió de Markov* ve donat per:

- Un conjunt numerable $\mathcal{S}$, que anomenarem conjunt d'estats. La situació de l'agent dins l'entorn a cada moment ve codificada per un dels estats de $\mathcal{S}$.
- Un conjunt numerable $\mathcal{A}$, que anomenarem conjunt d'accions. A cada estat $S_t \in \mathcal{S}$, l'agent pot elegir una acció $A_t \in \mathcal{A}$ per traslladar-se a un nou estat $S_{t+1} \in \mathcal{S}$.
- Una funció de transició $T(s, a, s') := \mathbb{P}(S_{t+1} = s' | S_t = s, A_t = a)$, que defineix la probabilitat de que, al realitzar l'acció a quan l'agent es troba a l'estat s, el condueixi a l'estat s'.
- Una funció de recompensa $R(s, a, s')$ que calcula la recompensa obtinguda per l'agent al transicionar de l'estat s a l'estat s' aplicant l'acció a.

La funció de transició ha de complir la *propietat de Markov*: l'estat actual només depèn de l'estat immediatament anterior i l'acció elegida. És a dir, $\mathbb{P}(s_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, ..., S_0 = s_0, A_0 = a_0) = \mathbb{P}(s_{t+1} = s' | S_t = s_t, A_t = a_t)$.

S'anomena *política* a una aplicació $\pi : \mathcal{S} \to \mathcal{A}$ que assigna una acció a cada estat. Aquesta política pot servir a l'agent per determinar quina acció ha de realitzar quan es troba sobre un estat determinat; l'objectiu del problema de decisió de Markov és definir una política òptima π^* tal que, si l'agent la segueix, es maximitzi el *valor esperat* de $\sum_t R(s_t, a_t, s_{t+1})$ $t = 0, 1, 2....$

6.1.2 Utilitat d'un estat (Q-valors)

L'aprenentatge per reforç és un cas particular de problema de decisió de Markov en el qual l'agent no coneix a priori ni la funció de transició ni la funció de recompensa. Per tal de trobar una política òptima l'agent haurà d'aprendre a estimar la *utilitat* de cada estat, és a dir, el valor esperat de la suma de recompenses a llarg plaç quan es realitza una acció sobre un cert estat. Una forma de representar la utilitat d'una parella estat-acció és mitjançant els Q-valors, que es defineixen com el valor esperat que obtindria l'agent si comença sobre l'estat s, realitza l'acció a, i després aplica una política òptima. És a dir:

$$Q(s,a) = \sum_{s'} T(s,a,s') \left(R(s,a,s') + \gamma \max_{a'} Q(s',a') \right) , \tag{6.1}$$

amb $\gamma \in (0,1]$, $s, s' \in \mathcal{S}$, $a, a' \in \mathcal{A}$. Els Q-valors normalment es representen en una taula, anomenada taula de Q-valors, que tindrà tants elements com parelles estat-acció hi hagi.

Donat que l'agent no té coneixement de T ni R, l'única manera de determinar els Q-valors serà a través de l'experiència (realitzant observacions de l'entorn) i actualitzant la taula de forma iterativa. Una forma d'aconseguir-ho és mitjançant l'*algorisme d'iteració de Q-valors*:

Algorisme 1: Iteració de Q-valors

```
Inicialitzar els elements de la taula Q(s,a) a 0 ;
Definir E màxim d'episodis ;
Definir M màxim de passos d'entrenament ;
Definir 0 < α < 1 ;
Definir 0 < ε < 1 ;
Definir 0 < γ < 1 ;
for i ∈ {0,1,2,...,E} do
    Inicialitzar estat inicial s ;
    for k ∈ {0,1,2,...,M} do
        Elegir una acció a aleatòria amb prob. ε, altrament elegir acció a amb Q(s,a)
         més alt ;
        Realitzar l'acció a, observar el nou estat s' i la recompensa r ;
        Q(s,a) ← (1 − α) · Q(s,a) + α(r + γ · max_a' Q(s',a')) ;
        s ← s';
        Si s és estat terminal, sortir del bucle ;
    end
end
```

6.2 Deep Q-Learning

L'algorisme Q-Learning és eficaç quan la quantitat d'accions/estats possibles és limitada. No obstant, es fàcil adonar-se que aquest algorisme presenta dos inconvenients importants quan s'enfronta a problemes de més complexitat:

1. La mida de la taula de Q-valors creix de forma proporcional a la quantitat d'estats i accions possibles. Això farà que per a problemes complexos (amb molts estats i/o accions) es requereixi gran quantitat de memòria per emmagatzemar tots els Q-valors.

2. Si la taula de Q-valors té molts elements, l'agent necessitarà més temps per explorar els estats de l'entorn i actualitzar la taula. Per entorns amb centenars o milers d'estats això pot ser impossible d'aconseguir en un temps raonable.

Una de les primeres solucions efectives a aquest problema fou plantejada per primer cop a l'article [9] i consisteix en utilitzar una xarxa neuronal per aproximar la taula dels Q-valors. És a dir, en lloc de calcular els Q-valors de cada parella estat-acció, la xarxa neuronal rebrà com a input un estat i donarà com a output el Q-valor aproximat de cada acció possible. L'agent pot després realitzar l'acció amb el valor màxim. La figura A.10 mostra de forma esquemàtica les diferències entre aquests dos algorismes.

6.3 DQN aplicat al control d'actitud

Aplicat en el context d'aquest treball, la xarxa neuronal és l'agent de control que calcularà quina quantitat de torque cal aplicar al satèl·lit segons l'actitud actual. El seu propòsit serà reduir la velocitat angular del satèl·lit a zero.

6.3.1 Representació dels estats i les accions

Els estats entre els que es podrà moure l'agent estaran representats per un vector de 7 elements, on els quatre primers representen el quaternió de l'orientació del satèl·lit i els tres últims representen la velocitat angular en els tres eixos principals del sistema de referència del cos del satèl·lit (en Newtons per metre).

$$[\ \overbrace{q_0, q_x, q_y, q_z}^{\text{orientació (quaternió)}}\ ,\ \underbrace{\omega_x, \omega_y, \omega_z}_{\text{velocitat angular}}\ ] \tag{6.2}$$

L'algorisme DQN és un algorisme de control discret: això vol dir que el conjunt d'accions possibles que pot realitzar l'agent ha de ser un conjunt discret. En conseqüència, cal discretitzar el torque de control i codificar-lo en un conjunt d'accions.

El vector del torque de control serà un vector $T = [T_x, T_y, T_z]$ les components del qual només podran prendre valors en el conjunt $\{-0.01, 0, +0.01\}$. Això ens permet definir 7 accions que podrà realitzar l'agent:

Components de T	Núm. Acció
$T = [0, 0, 0]$	Acció 1
$T = [-0.01, 0, 0]$	Acció 2
$T = [0.01, 0, 0]$	Acció 3
$T = [0, 0.01, 0]$	Acció 4
$T = [0, -0.01, 0]$	Acció 5
$T = [0, 0, -0.01]$	Acció 6
$T = [0, 0, 0.01$	Acció 7

6.3.2 Entorn de simulació

A l'igual que en el Capítol 5, l'entorn de simulació s'encarregarà de calcular com varia l'actitud a l'aplicar un torque (és a dir, calcularà les transicions entre estats). La diferència està en que, durant l'entrenament, ara també s'ocuparà d'informar a l'algorisme DQN quina és la recompensa de cada acció i també si un estat és terminal (vegeu figura A.11). Considerarem que un estat és terminal quan el temps de simulació ha arribat als 30 segons.

6.3.3 Funció de recompensa

L'objectiu del controlador és reduir la velocitat angular dels tres eixos a zero. Per contra, l'objectiu de l'algorisme DQN és el d'entrenar un agent que maximitzi la suma de les recompenses. És per aquesta raó que, a l'hora de dissenyar la funció de recompensa, cal que aquesta decreixi ràpidament de forma inversament proporcional a l'error en la velocitat angular. La funció que hem utilitzat és la funció Gaussiana, definida com:

$$g(\omega_i - \omega_0) = \frac{1}{\sqrt{2\pi}} e^{-1/2 \cdot (\omega_i - \omega_0)^2} \tag{6.3}$$

on ω_i és la velocitat angular actual del satèl·lit i ω_0 és la velocitat angular de referència (en el nostre cas particular, $\omega_0 = [0, 0, 0]$).

6.3.4 Arquitectura i hiperparàmetres de la xarxa

L'arquitectura de la xarxa serà la mateixa que es detalla a [6]: una capa d'entrada de 7 neurones, dues capes ocultes totalment conectades de 1024 i 2048 neurones respectivament i una capa de sortida de 7 neurones (una per a cada acció). La xarxa donarà com a sortida les utilitats aproximades que ha calculat per a cada una de les accions, i l'agent podrà seleccionar llavors l'acció amb major utilitat.

El búffer d'experiència tindrà una mida màxima de 500 exemples, el màxim d'episodis d'entrenament serà 3000.

Per tal d'elegir les accions a realitzar a cada iteració de l'algorisme utilitzarem una política ϵ-greedy amb factor de descompte 0.99. Això vol dir que cada vegada que l'agent hagi d'elegir una acció, n'escollirà una aleatòriament amb probabilitat ϵ o bé triarà l'acció de més utilitat amb probabilitat $(1 - \epsilon)$. Inicialment s'estableix $\epsilon = 1$, i a cada iteració es multiplica el valor de ϵ per 0.99, fins a un mínim de $\epsilon = 0.001$. D'aquesta forma, l'agent tendirà a explorar estratègies diferents al principi de l'entrenament i optarà per estratègies més conservadores cap al final.

6.3.5 Algorisme DQN en pseudocodi

Tot seguit s'especifica el funcionament en pseudocodi de l'algorisme d'entrenament del controlador. L'implementació en Python es pot trobar a la secció B.3.5 de l'Annex B. És important remarcar que aquest algorisme no serà el controlador pròpiament, sinó que s'encarregarà d'entrenar la xarxa neuronal que, després, podrà ser utilitzada com a controlador.

Algorisme 2: Entrenament del controlador amb l'algorisme DQN

Inicialitzar el búffer d'experiència M amb capacitat màxima M_{max} i mínima M_{min} ;
Inicialitzar $\epsilon = 1$;
Inicialitzar la xarxa neuronal Q amb pesos θ ;
Definir el màxim d'episodis d'entrenament E ;
for *episodi* $\in \{0, 1, 2, ..., E\}$ **do**
  Inicialitzar estat inicial s ;
  while *1* **do**
    Seleccionar una acció a aleatòria amb probabilitat ϵ, o elegir $\max Q(s; \theta)$;
    Realitzar a sobre l'estat s, i observar el nou estat s' i la recompensa r ;
    Guardar el vector (s, a, r, s') dins M ;
    $\epsilon \leftarrow \epsilon \cdot 0.99$;
    if $len(M) >= M_{min}$ **then**
      Elegir a l'atzar un subconjunt d'elements de M ;
      Per cada element (s_j, a_j, r_j, s'_j) de la xarxa, calcular la seva utilitat y_j:

$$y_j = \begin{cases} r_j & \text{si } s'_j \text{ és terminal} \\ r_j + \gamma \max Q(s'_j; \theta) & \text{si } s'_j \text{ no és terminal} \end{cases}$$

      ;
      Actualitzar els pesos θ de la xarxa utilitzant l'optimitzador SGD (descens per gradient estocàstic) per minimitzar $(y_j - Q(s_j; \theta))^2$;
    end
    $s \leftarrow s'$;
    Si el temps de la simulació és superior a 30 segons, sortir del bucle *while*.
  end
end

6.4 Implementació i discussió dels resultats obtinguts

6.4.1 Sobre la llibreria Keras

Per tal d'agilitzar la implementació de l'algorisme d'aprenentatge automàtic, es va optar per utilitzar la llibreria Keras per ajudar amb la construcció i l'entrenament de la xarxa neuronal. Keras és una llibreria de Python de codi obert i distribuïda sota llicència MIT que agilitza enormement el procés de crear i experimentar amb models de xarxes neuronals artificials. Actualment és una llibreria molt utilitzada en l'àmbit de l'aprenentatge automàtic i contè nombroses implementacions de funcions i mètodes empleats habitualment en la programació de xarxes neuronals (diferents tipus d'arquitectures i capes, funcions d'activació, optimitzadors, etc), a més de disposar d'alguns conjunts de dades estàndar per fer experiments

(com la base de dades de dígits MNIST).

Keras es caracteritza per la seva gran facilitat d'ús i una interfície de funcions molt amigable per als programadors.

6.4.2 Proves inicials

La primera versió de l'algorisme DQN per entrenar el controlador de satèl·lits és la que es pot veure al programa B.3.5 de l'Annex B. Abans d'aplicar aquesta implementació de l'algorisme DQN al problema del control d'actitud de satèl·lits artificials, es va decidir comprovar el seu bon funcionament aplicant-la a un altre problema més senzill i ben estudiat: l'anomenat *Cart Pole Balancing Problem* (vegeu [5], pàg. 886).

Aquest problema consisteix en entrenar un agent intel·ligent capaç de balancejar una barra de ferro col·locada verticalment sobre un carro mòbil que es mou a dreta o esquerra. Els estats venen codificats per l'angle d'inclinació i la velocitat de la barra, i la posició i velocitat del carro. La barra s'inicialitza amb un angle molt proper a $\pi/2$. L'agent pot moure el carro cap a la dreta o l'esquerra obtenint una recompensa de $+1$ per cada segon que la vara no caigui per sota un cert angle. Si la vara s'inclina massa o el carro s'allunya de la posició inicial, el problema finalitza.

Per tal de no haver de programar un nou entorn de simulació per aquest problema, es va optar per utilitzar la llibreria *OpenAI Gym* que proporciona varis entorns prefabricats per simular problemes d'aquest tipus. També van ser necessàries algunes modificacions puntuals al codi per adaptar aquest nou entorn de simulació al programa en el lloc de l'entorn de simulació de satèl·lits, i es va modificar el tamany de la xarxa neuronal: una capa d'entrada de 4 neurones, dues capes ocultes totalment conectades de 8 i 4 neurones i una capa de sortida de 2 neurones.

La xarxa neuronal es va entrenar durant 130 episodis. Una vegada entrenada, se li va fer resoldre el problema de nou durant 100 episodis de prova: en tots ells va obtenir una recompensa total de 200, que és el màxim permès per l'entorn de simulació utilitzat. També es va comparar el comportament amb el d'un agent aleatori (és a dir, un agent que sempre realitza accions a l'atzar) i es va observar una recompensa mitjana de 22.26, sense arribar a superar mai 50.

6.4.3 Simplificació del problema

Una vegada es va comprovar que l'algorisme DQN estava ben implementat, es va procedir a entrenar-lo amb l'entorn de simulació del satèl·lit i l'arquitectura mencionada a la Secció ***6.3.4***. Malauradament els únics processadors disponibles per fer l'entrenament (vegeu secció ***B.2***), en el millor dels casos, necessitaven un total de 18 hores per processar un únic episodi d'entrenament (d'un total de 3000).

Per tant, es va optar per simplificar el problema i intentar entrenar un controlador més simple, tal i com s'ha explicat al principi d'aquest capítol. Per simplificar el problema, es va començar per cambiar el mètode d'integració numèrica, per tal que les iteracions de cada episodi fossin més ràpides de calcular, a costa de perdre precisió. En lloc d'usar Runge-Kutta 45 es va optar per Runge-Kutta 5 sense control de pas. Després, es va modificar la forma de generar les condicions inicials del simulador: el satél·lit parteix sempre del quaternió $\mathfrak{q}_r = [1, 0, 0, 0]$ i se li aplica sempre el mateix torque de pertorbació: $T = [-2, -0.2, 0.2]$. En aquest cas, l'actitud després d'aplicar la pertorbació serà $\omega_0 = [0.02700035, -0.02000273, -0.01932673]$. Finalment, es va cambiar la mida de les capes ocultes de la xarxa neuronal per 128 i 256 neurones, i es va entrenar la xarxa durant tres dies fins a un total de 66 iteracions.

El resultat va ser un controlador que, si bé necessita més entrenament per resoldre aquest problema particular de forma òptima (com es pot observar a la figura A.12), és capaç d'obtenir una velocitat angular final amb un error (en norma infinit) per sota de $1e-3$. L'implementació dels programes simplificats es pot veure a les Seccions B.3.3 i B.3.6.

Bibliografia

[1] MARCOTE M. (2002). *Vuelo en formación de constelaciones de satélites* (Tesina). Universitat de Barcelona, Facultat de Matemàtiques, Barcelona.

[2] CASTELLET M., LLERENA I. CASACUBERTA C. (2000). *Àlgebra lineal i geometria*. Universitat Autònoma de Barcelona, Servei de Publicacions.

[3] CANUTO E., NOVARA C., MASSOTTI L., CARLUCCI D., MONTENEGRO C. (2018) *Spacecraft Dynamics and Control: The Embedded Model Control Approach*. Elsevier Aerospace Engineering Series

[4] HASSAN K. KHALIL (2015) *Nonlinear Control*. Pearson Education.

[5] RUSSELL S., NORVIG P. (2004) *Inteligencia artificial: un enfoque moderno*. Pearson Education.

[6] ZHONG MA et al. (2018) *Reinforcement Learning-Based Satellite Attitude Stabilization Method for Non-Cooperative Target Capturing*. Multidisciplinary Digital Publishing Institute.

[7] MATHEWS J., FINK K. (2004) *Numerical Methods Using Matlab, 4th Edition*. Prentice-Hall Press.

[8] HENRY W., ARNO S. (2007) *Numerical Recipes in C: The art of scientific computing*. Cambridge University Press.

[9] MIHN V., KAVUKCUOGLU K., et al. (2013) *Playing Atari with Deep Reinforcement Learning*. Deepmind Technologies.

Apèndixs

A
Figures

Aquesta secció conté les figures amb els resultats dels experiments que es discuteixen en diverses seccions del treball. Totes les figures han estat generades mitjançant Gnuplot.

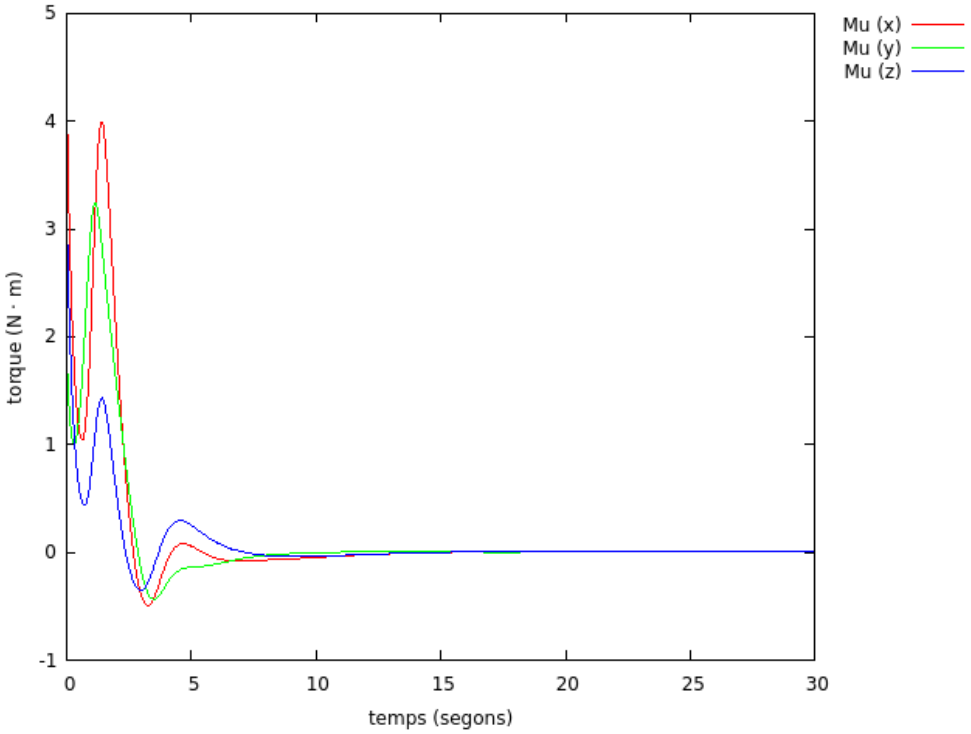


Figura A.1: Torque generat pel controlador al llarg de 30 segons.

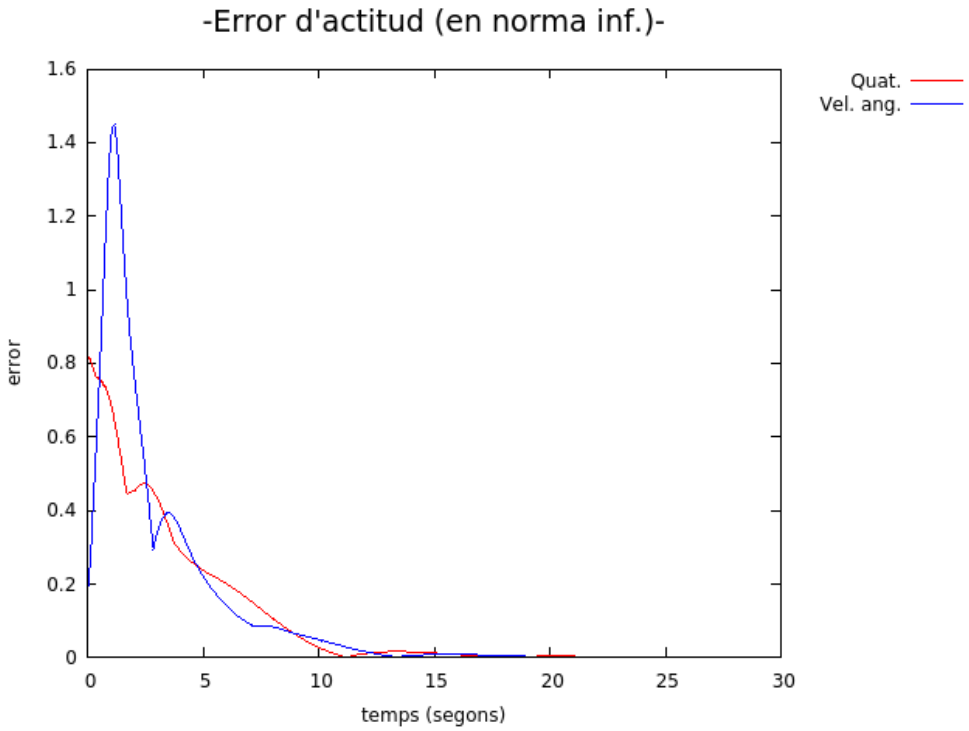


Figura A.2: Error d'actitud (en norma infinit).

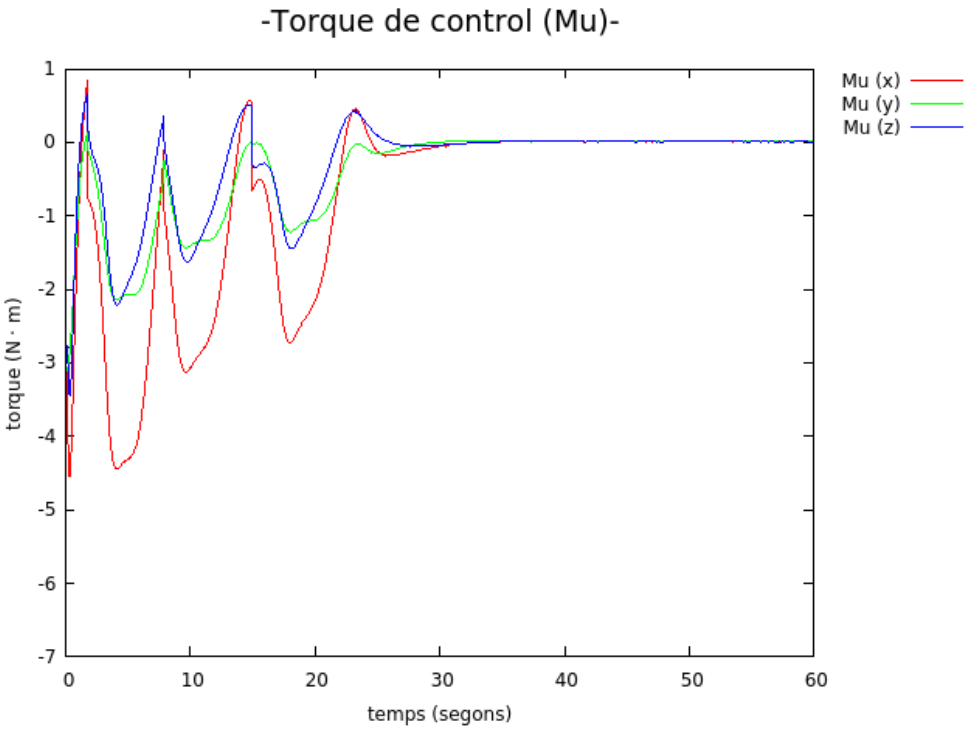


Figura A.3: En aquest cas, la velocitat angular inicial és elevada i el controlador reacciona bruscament per intentar equilibrar-la.

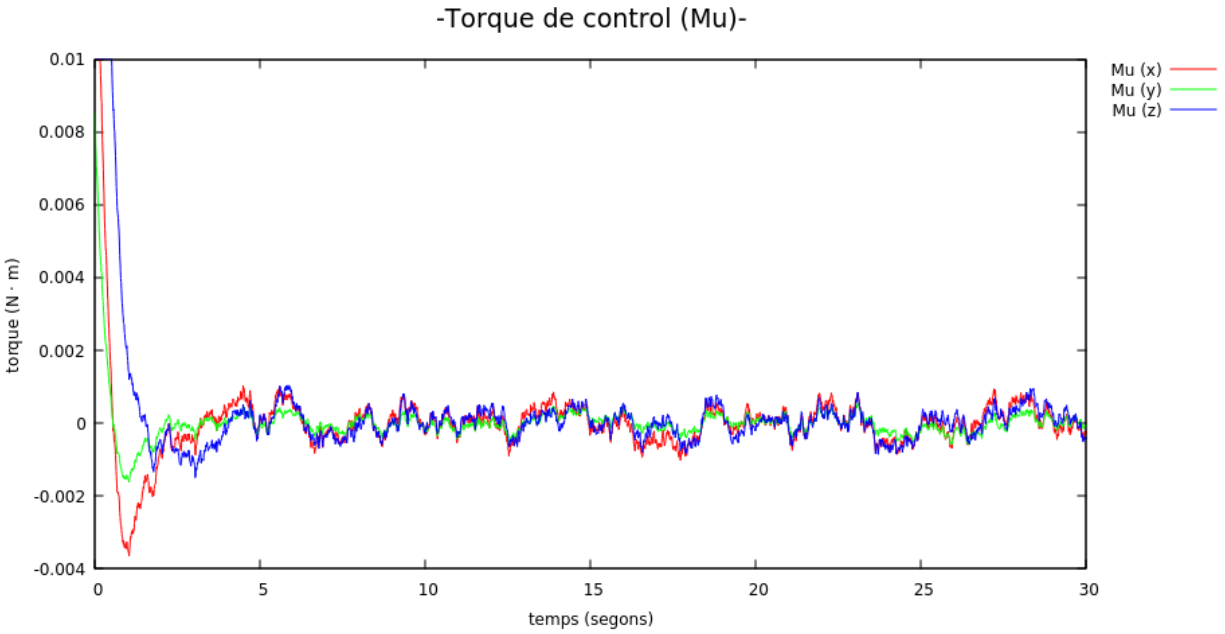


Figura A.4: Comportament del controlador a l'exemple generat per les llavors 123 (entorn) i 1 (pertorbació)

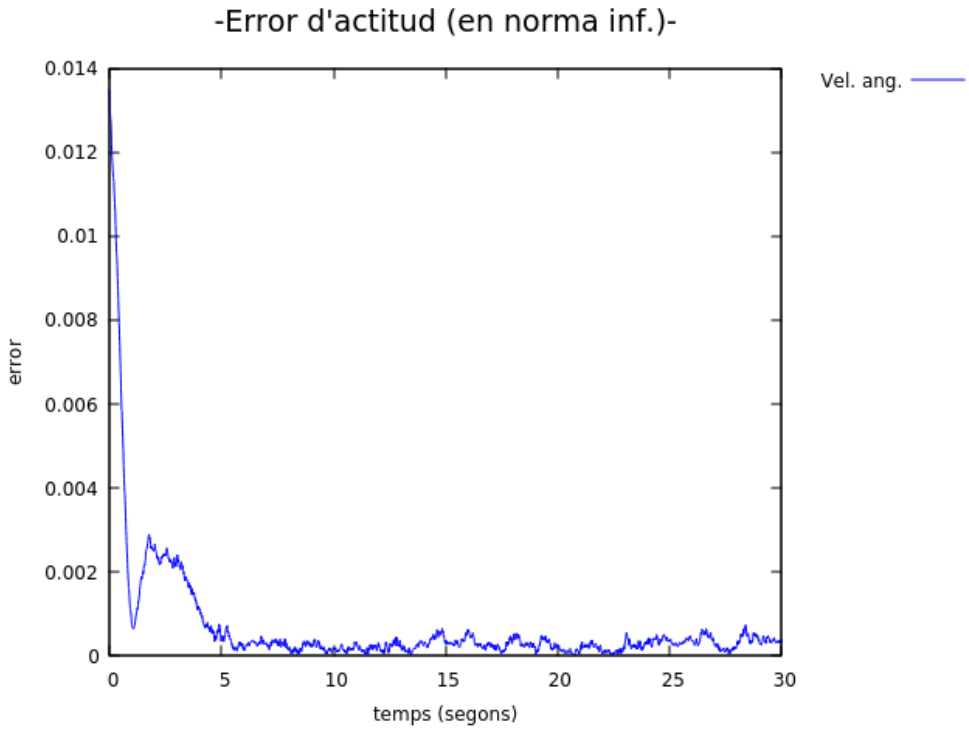


Figura A.5: Comportament de l'error a l'exemple generat per les llavors 123 (entorn) i 1 (pertorbació)

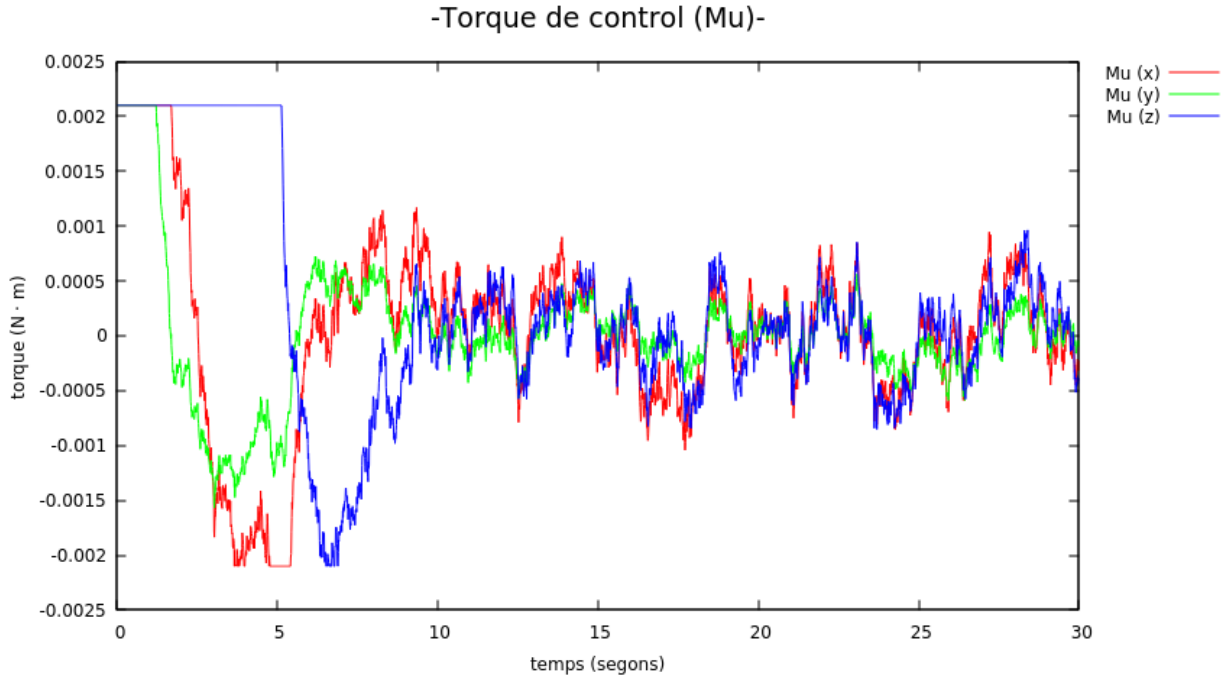


Figura A.6: Comportament del controlador a l'exemple generat per les llavors 123 (entorn) i 1 (pertorbació), limitant el torque que poden generar els actuadors a $\pm 0.0021\, N \cdot m$.

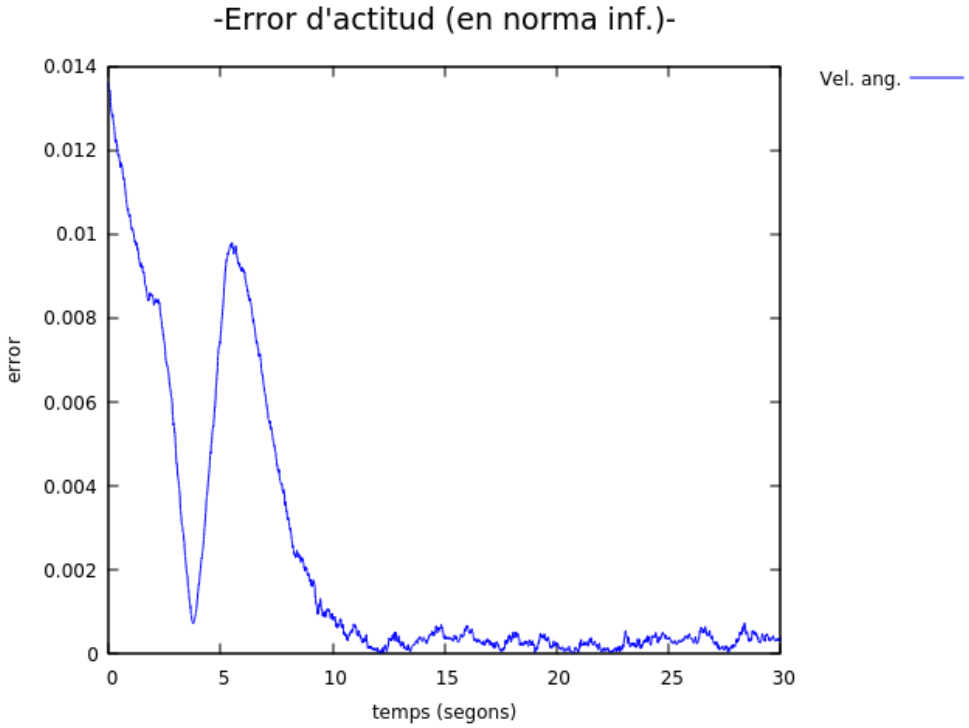


Figura A.7: Comportament de l'error a l'exemple generat per les llavors 123 (entorn) i 1 (pertorbació), limitant el torque que poden generar els actuadors a $\pm 0.0021\, N \cdot m$

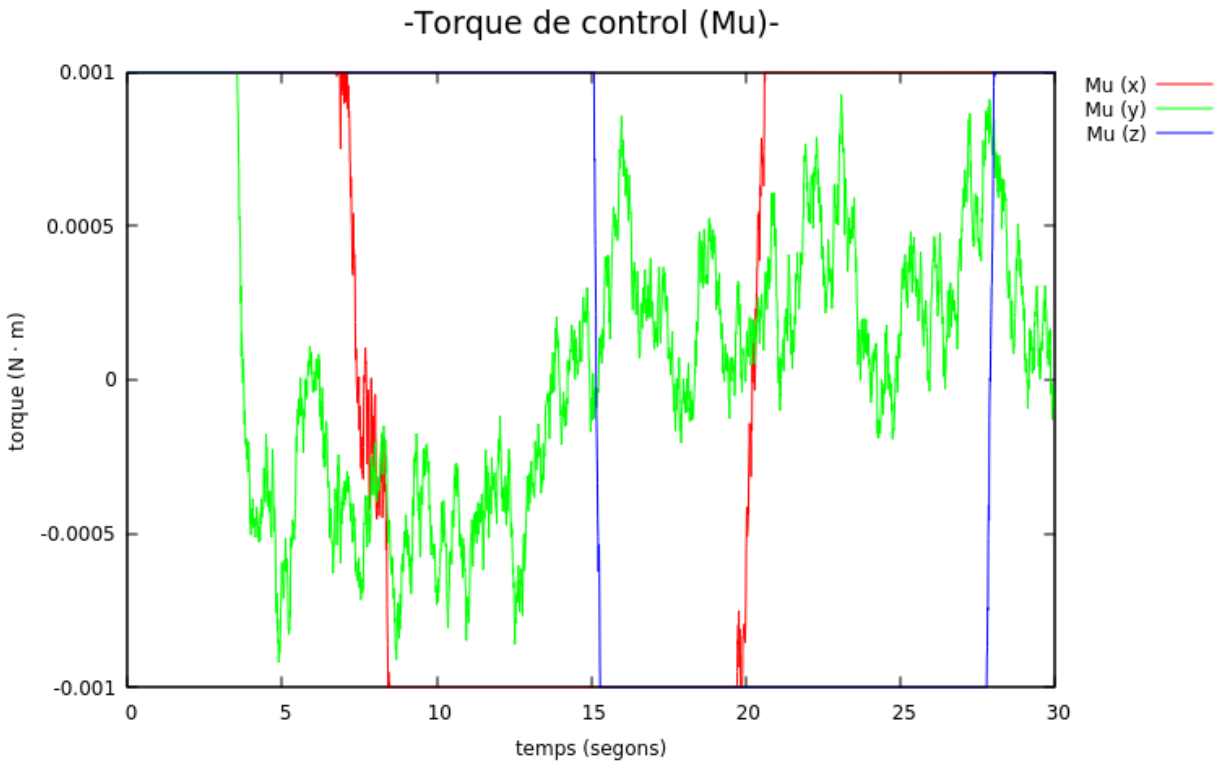


Figura A.8: Comportament del controlador a l'exemple generat per les llavors 123 (entorn) i 1 (pertorbació), limitant el torque que poden generar els actuadors a $\pm 0.001\, N \cdot m$.

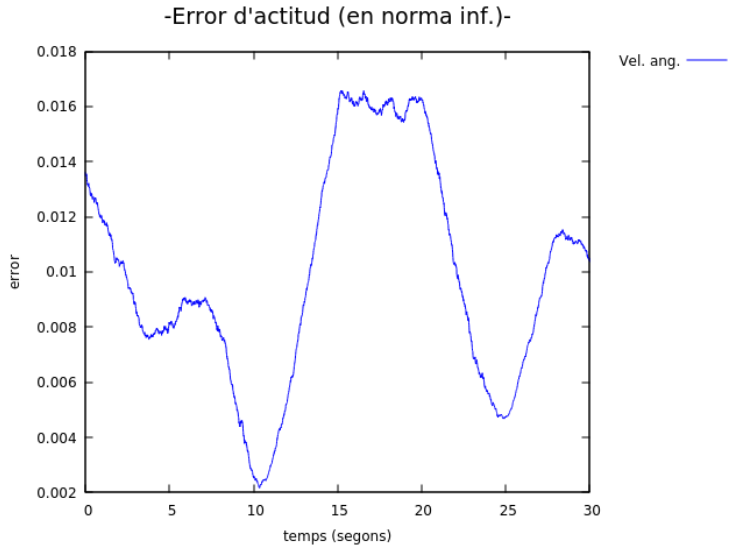


Figura A.9: Comportament de l'error a l'exemple generat per les llavors 123 (entorn) i 1 (pertorbació), limitant el torque que poden generar els actuadors a $\pm 0.001\, N \cdot m$

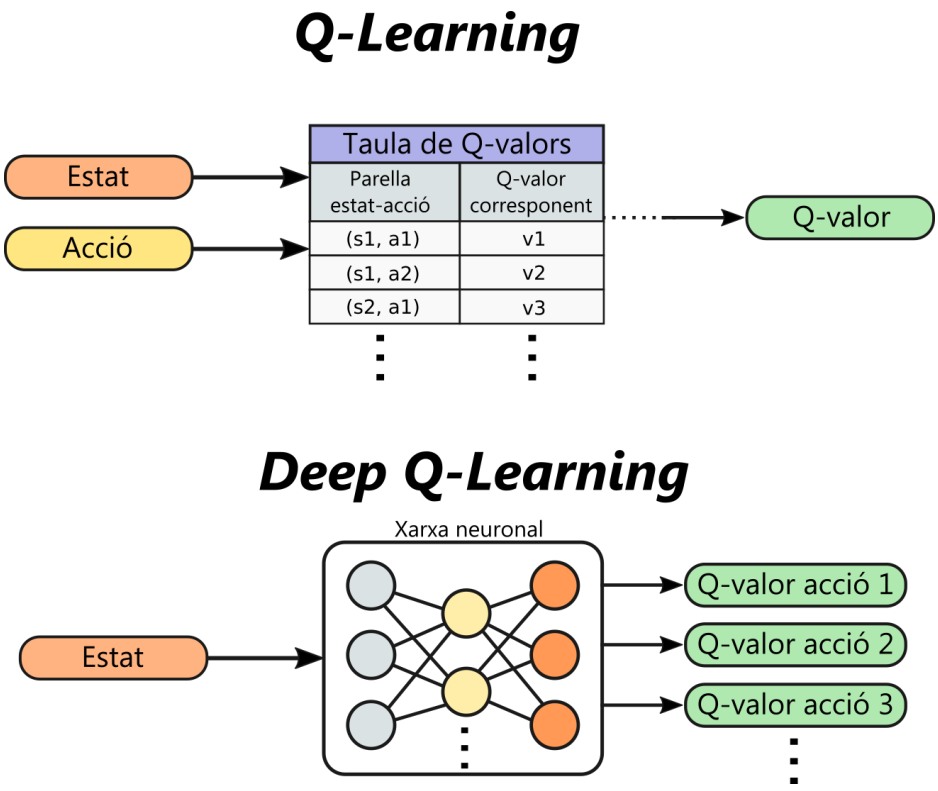


Figura A.10: Algorisme Q-learning vs. Deep Q-learning

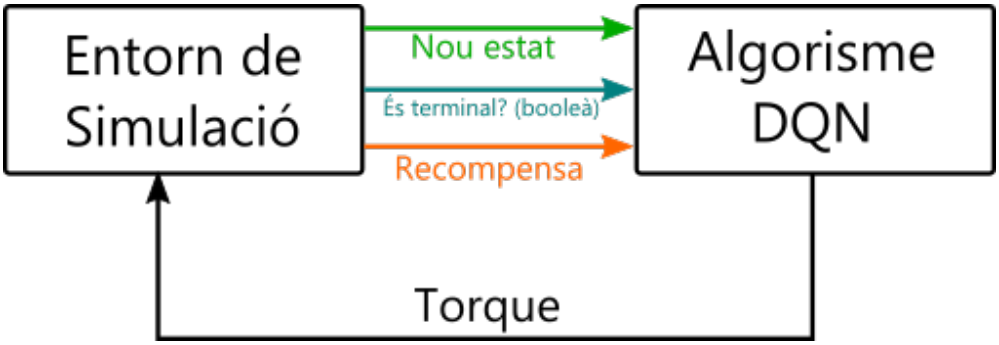


Figura A.11: Durant l'entrenament, l'entorn de simulació també informarà a l'algorisme DQN quines són les recompenses de cada estat i quins són terminals.

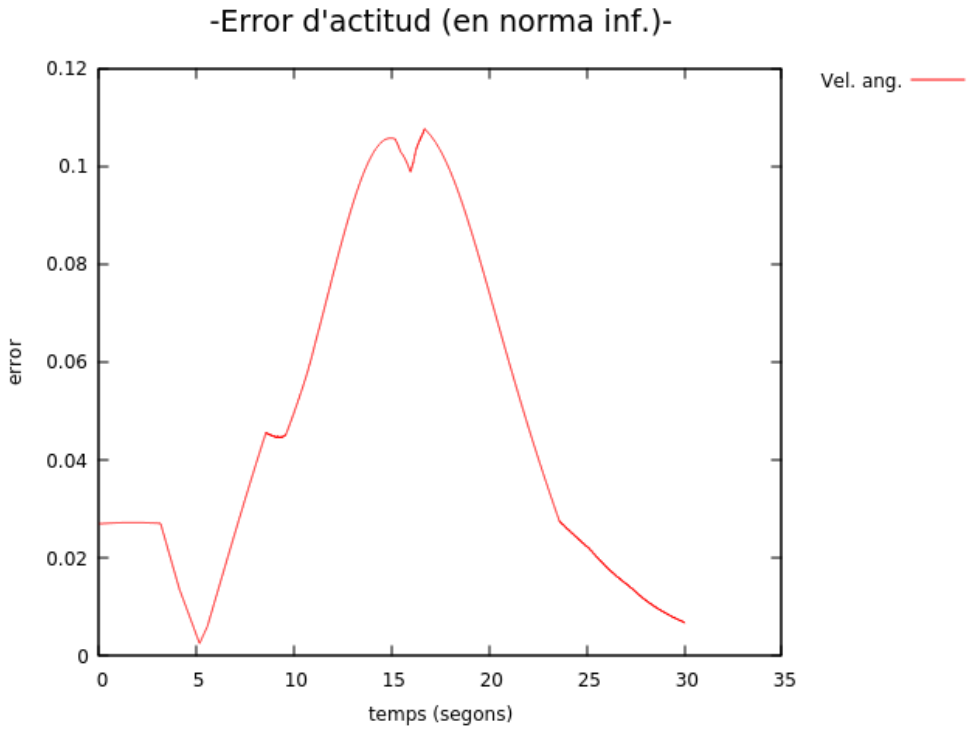


Figura A.12: Comportament de l'error en l'exemple d'aprenentatge per reforç. Veiem que, si bé el controlador és capaç d'estabilitzar l'actitud, necessitaria més entrenament per tenir un comportament òptim.

B
Programes

En aquest annex s'inclou el codi dels programes que es discuteixen al llarg de la memòria, juntament amb alguns comentaris sobre el funcionament i les llibreries utilitzades. També apareix una llista amb les característiques dels ordinadors utilitzats per aquest treball.

B.1 Llibreries i versions

Tots els programes han estat escrits en Python 3.8.5 i testejats amb Ubuntu 20.04 i Windows 10. Les següents llibreries i paquets han estat utilitzades en l'elaboració dels programes:

- **Numpy** (v. 1.17.4). Proporciona eines per operar i manipular matrius/vectors.
- **Collections**. Proporciona extensions a alguns objectes de python que permeten implementar fàcilment estructures habituals com cues circulars.
- **Tensorflow** (v. 2.3.1) amb **Keras** (v. 2.4.3). Proporcionen les eines per construir i entrenar fàcilment xarxes neuronals.
- **Random**. Utilitzat en els programes de Reinforcement Learning per seleccionar aleatòriament un subconjunt d'elements d'un array.

B.2 Especificacions dels ordinadors

Per fer aquest treball s'ha disposat dels següents ordinadors:

- Ordinador de torre amb AMD Phenom II X4, 4GB de RAM i Nvidia GT440.

- Ordinador de torre amb Ryzen 9, 16GB de memòria RAM i Nvidia RTX 3080. La GPU no ha pogut ser configurada correctament per funcionar amb Keras.
- Ordinador portàtil amb Intel i5, 4GB de memòria RAM i gràfica Intel HD Graphics 4000.

B.3 Programes complerts

B.3.1 Generador de nombres aleatoris

Codi de la classe LCG(), que implementa el generador lineal de congruències. Aquesta classe és carregada i utilitzada en altres programes del treball.

```
class LCG:
    # Custom random-number generator class.
    def __init__(self, seed):
        self.a = 1664525
        self.c = 1013904223
        self.m = 2**32

        self.seed = seed
    def randi(self, n_ini, n):
        # Returns a random integer between [n_ini, n]  (inclusive)
        self.seed = (self.a * self.seed + self.c) % self.m

        return n_ini + self.seed % (n+1)
    def randf(self):
        # Returns a random floating-point value between [0,1]
        self.seed = (self.a * self.seed + self.c) % self.m

        return self.seed / self.m
```

random_generator.py

B.3.2 Entorn de simulació

Aquest programa implementa l'entorn de simulació utilitzat en els exemples sense aprenentatge per reforç. També és l'entorn de simulació que s'utilitza juntament amb l'algorisme DQN.

```
import numpy as np
from random_generator import LCG

def qprod(a, b):
    # Returns the quaternion product a x b

    # If one of the parameters doesn't have dimension 4 x 1, we return
    error
    if a.shape != (4,) or b.shape != (4,):
        print("ERROR: Cannot multiply non-quaternion objects")
```

```
10          exit()
11
12      # We create a 4 x 1 vector to store the result:
13      ab = np.zeros(4)
14
15      # We calculate the product:
16      ab[0] = a[0]*b[0] - a[1]*b[1] - a[2]*b[2] - a[3]*b[3]
17      ab[1] = a[0]*b[1] + a[1]*b[0] + a[2]*b[3] - a[3]*b[2]
18      ab[2] = a[0]*b[2] - a[1]*b[3] + a[2]*b[0] + a[3]*b[1]
19      ab[3] = a[0]*b[3] + a[1]*b[2] - a[2]*b[1] + a[3]*b[0]
20
21      # We return the resulting quaternion:
22      return ab
23
24
25  class SatelliteEnviroment():
26
27      def __init__(self, seed):
28
29          # Main constructor for the satellite simulation enviroment.
30
31          self.s0 = np.zeros(7)
32
33          self.mass = 5.0
34          J = np.zeros((3,3))
35          J[0][0] = 2.0257
36          J[1][1] = 0.7998
37          J[2][2] = 1.2753
38          J[0][1] = 0.6498
39          J[0][2] = 1.1226
40          J[1][2] = 0.1833
41          J[1][0] = J[0][1]
42          J[2][0] = J[0][2]
43          J[2][1] = J[1][2]
44          self.Jinv = np.linalg.inv(J)
45          self.J = np.copy(J)
46
47          self.t = 0.0
48          self.t_max = 30.0
49
50          self.r = LCG(seed)
51
52          self.x = 0
53          self.y = 0
54          self.z = 0
55          self.angle = 0
56
57      def generate_random_torque(self):
58
59          TR = np.zeros(3)
60
61          TR[0] = self.r.randi(-100,100)
62          TR[1] = self.r.randi(-100,100)
63          TR[2] = self.r.randi(-100,100)
64
```

```
65          return 0.01 * TR
66
67      def reset(self):
68
69          angle = (self.r.randf() * 2.0 - 1.0) * np.pi
70          C = np.cos(angle / 2.0)
71          S = np.sin(angle / 2.0)
72          x = self.r.randf()
73          y = self.r.randf()
74          z = self.r.randf()
75          norm = (x*x+y*y+z*z)**0.5
76          q0 = np.array([C , S * (x/norm), S * (y/norm), S * (z/norm)])
77          w0 = np.zeros(3)
78          state = np.concatenate((q0, w0), axis=None)
79          self.s0 = state.copy()
80
81          T = self.generate_random_torque()
82
83          initial_state = self.s0.copy()
84          initial_state, _, _ = self.take_step(initial_state, T)
85
86          self.t = 0.0
87          self.t_max = 30.0
88
89          self.x = x
90          self.y = y
91          self.z = z
92          self.angle = angle
93
94          return initial_state
95
96      def reset_from_state_and_torque(self, state, T):
97
98          self.s0 = state
99
100         initial_state = self.s0.copy()
101         for i in range(10):
102             initial_state, _, _ = self.take_step(initial_state, T)
103
104         self.t = 0.0
105         self.t_max = 30.0
106
107         return initial_state
108
109     def reset_from_state(self, state):
110         self.s0 = state
111
112         T = self.generate_random_torque()
113
114         initial_state = self.s0.copy()
115         for i in range(10):
116             initial_state, _, _ = self.take_step(initial_state, T)
117
118         self.t = 0.0
119         self.t_max = 30.0
```

```
120
121         return initial_state
122
123
124     def get_inertia_tensor(self):
125
126         return self.J, self.Jinv
127
128     def get_current_time(self):
129
130         return self.t
131
132     def action_to_torque(self, index):
133
134         # Given an action index (integer between 0-6, inclusive)
135         # returns the corresponding torque vector.
136
137         T = np.zeros(3)
138
139         if index == 0:
140             pass
141         elif index == 1:
142             T[0] = -0.01
143         elif index == 2:
144             T[0] = 0.01
145         elif index == 3:
146             T[1] = -0.01
147         elif index == 4:
148             T[1] = 0.01
149         elif index == 5:
150             T[2] = -0.01
151         elif index == 6:
152             T[2] = 0.01
153
154         return T
155
156     def take_step_from_action(self, state, action):
157
158         T = self.action_to_torque(action)
159
160         new_state, reward, done = self.take_step(state, T)
161
162         return new_state, reward, done
163
164
165
166     def take_step(self, state, T):
167
168         h = 0.01
169         e_max = 0.0001
170
171         valido = False
172
173         q = state[0:4]
174         w = state[4:7]
```

```
175
176         qn = q.copy()
177         wn = w.copy()
178
179         #Integrate using RK-45:
180         while(not valido):
181             k1q = h * self.f1(qn, wn)
182             k1w = h * self.f2(qn, wn, T)
183
184             k2q = h * self.f1(qn + 0.25 * k1q, wn + 0.25 * k1w)
185             k2w = h * self.f2(qn + 0.25 * k1q, wn + 0.25 * k1w, T)
186
187             k3q = h * self.f1(qn + 3.0 / 32.0 * k1q + 9.0 / 32.0 * k2q, wn
     + 3.0 / 32.0 * k1w + 9.0 / 32.0 * k2w)
188             k3w = h * self.f2(qn + 3.0 / 32.0 * k1q + 9.0 / 32.0 * k2q, wn
     + 3.0 / 32.0 * k1w + 9.0 / 32.0 * k2w, T)
189
190             k4q = h * self.f1(qn + 1932.0 / 2197.0 * k1q - 7200.0 / 2197.0
     * k2q + 7296.0 / 2197.0 * k3q, wn + 1932.0 / 2197.0 * k1w - 7200.0 /
    2197.0 * k2w + 7296.0 / 2197.0 * k3w)
191             k4w = h * self.f2(qn + 1932.0 / 2197.0 * k1q - 7200.0 / 2197.0
     * k2q + 7296.0 / 2197.0 * k3q, wn + 1932.0 / 2197.0 * k1w - 7200.0 /
    2197.0 * k2w + 7296.0 / 2197.0 * k3w, T)
192
193             k5q = h * self.f1(qn + 439.0 / 216.0 * k1q - 8.0 * k2q +
    3680.0 / 513.0 * k3q - 845.0 / 4104.0 * k4q, wn + 439.0 / 216.0 * k1w
    - 8.0 * k2w + 3680.0 / 513.0 * k3w  - 845.0 / 4104.0 * k4w)
194             k5w = h * self.f2(qn + 439.0 / 216.0 * k1q - 8.0 * k2q +
    3680.0 / 513.0 * k3q - 845.0 / 4104.0 * k4q, wn + 439.0 / 216.0 * k1w
    - 8.0 * k2w + 3680.0 / 513.0 * k3w  - 845.0 / 4104.0 * k4w, T)
195
196             k6q = h * self.f1(qn - 8.0 / 27.0 * k1q + 2.0 * k2q - 3544.0 /
     2565.0 * k3q + 1859.0 / 4104.0 * k4q - 11.0 / 40.0 * k5q, wn - 8.0 /
    27.0 * k1w + 2.0 * k2w - 3544.0 / 2565.0 * k3w + 1859.0 / 4104.0 * k4w
     - 11.0 / 40.0 * k5w)
197             k6w = h * self.f2(qn - 8.0 / 27.0 * k1q + 2.0 * k2q - 3544.0 /
     2565.0 * k3q + 1859.0 / 4104.0 * k4q - 11.0 / 40.0 * k5q, wn - 8.0 /
    27.0 * k1w + 2.0 * k2w - 3544.0 / 2565.0 * k3w + 1859.0 / 4104.0 * k4w
     - 11.0 / 40.0 * k5w, T)
198
199
200             qn1 = qn + 25.0 / 216.0 * k1q + 1408.0 / 2565.0 * k3q + 2197.0
     / 4101.0 * k4q - 1.0 / 5.0 * k5q
201             wn1 = wn + 25.0 / 216.0 * k1w + 1408.0 / 2565.0 * k3w + 2197.0
     / 4101.0 * k4w - 1.0 / 5.0 * k5w
202
203             zq = qn + 16.0 / 135.0 * k1q + 6656.0 / 12825.0 * k3q +
    28561.0 / 56430.0 * k4q - 9.0 / 50.0 * k5q + 2.0 / 55.0 * k6q
204             zw = wn + 16.0 / 135.0 * k1w + 6656.0 / 12825.0 * k3w +
    28561.0 / 56430.0 * k4w - 9.0 / 50.0 * k5w + 2.0 / 55.0 * k6w
205
206             y = np.concatenate((qn1, wn1), axis = 0)
207             z = np.concatenate((zq, zw), axis = 0)
208
209             if(np.linalg.norm(z-y) > e_max):
```

```
210                 h = h / 2.0
211
212             else:
213                 qn = zq
214                 wn = zw
215                 valido = True
216
217         new_state = np.concatenate((qn, wn), axis = None)
218         reward = self.calculate_reward(z)
219
220         self.t += h
221
222
223
224         # Check if any of the termination conditions occurs:
225         done = False
226         if self.t >= self.t_max:
227             done = True
228
229         return new_state, reward, done
230
231
232     def calculate_reward(self, state):
233
234         wi = state[4:7]
235         w0 = self.s0[4:7]
236
237         norma = np.linalg.norm(wi-w0)
238
239         g = (1.0 / (2*np.pi)**(0.5) ) * np.e ** (-0.5 * norma * norma)
240
241         return g
242
243     def f1(self, q,w):
244        #f1 corresponds to the first part of equation 3.15  (\dot{q})
245        wq = np.array([0,w[0],w[1],w[2]])
246        q_result = 0.5 * qprod(q, wq)
247
248        return q_result
249
250     def f2(self, q,w, T):
251         #f2 corresponds to the second part of equation 3.15 (\dot{omega})
252
253         torque_Mu = T
254
255         w_result = - np.cross(self.Jinv @ w , self.J @ w) + self.Jinv @ (
    torque_Mu)
256
257         return w_result
```

satellite_enviroment.py

B.3.3 Entorn de simulació simplificat

Aquest és el codi de l'entorn de simulació simplificat que s'ha acabat utilitzant per entrenar el controlador. S'ha canviat el mètode d'integració per Runge-Kutta 5 i s'ha forçat que sempre es parteixi de les mateixes condicions inicials.

```
import numpy as np
from random_generator import LCG

def qprod(a, b):
    # Returns the quaternion product a x b

    # If one of the parameters doesn't have dimension 4 x 1, we return error
    if a.shape != (4,) or b.shape != (4,):
        print("ERROR: Cannot multiply non-quaternion objects")
        exit()

    # We create a 4 x 1 vector to store the result:
    ab = np.zeros(4)

    # We calculate the product:
    ab[0] = a[0]*b[0] - a[1]*b[1] - a[2]*b[2] - a[3]*b[3]
    ab[1] = a[0]*b[1] + a[1]*b[0] + a[2]*b[3] - a[3]*b[2]
    ab[2] = a[0]*b[2] - a[1]*b[3] + a[2]*b[0] + a[3]*b[1]
    ab[3] = a[0]*b[3] + a[1]*b[2] - a[2]*b[1] + a[3]*b[0]

    # We return the resulting quaternion:
    return ab


class SatelliteEnviroment():

    def __init__(self, seed):

        # Main constructor for the satellite simulation enviroment.

        self.s0 = np.zeros(7)

        self.mass = 5.0
        J = np.zeros((3,3))
        J[0][0] = 2.0257
        J[1][1] = 0.7998
        J[2][2] = 1.2753
        J[0][1] = 0.6498
        J[0][2] = 1.1226
        J[1][2] = 0.1833
        J[1][0] = J[0][1]
        J[2][0] = J[0][2]
        J[2][1] = J[1][2]
        self.Jinv = np.linalg.inv(J)
        self.J = np.copy(J)

        self.t = 0.0
```

```
48          self.t_max = 30.0
49
50          self.max_steps = 3000
51
52          self.r = LCG(seed)
53
54      def generate_random_torque(self):
55
56          return np.array([2, -0.2, 0.2])
57
58      def reset(self):
59
60          q0 = np.array([1,0,0,0])
61          w0 = np.zeros(3)
62          state = np.concatenate((q0, w0), axis=None)
63          self.s0 = state.copy()
64
65          T = self.generate_random_torque()
66
67          initial_state = self.s0.copy()
68          initial_state, _, _ = self.take_step(initial_state, T)
69
70          self.t = 0.0
71          self.t_max = 30.0
72
73          return initial_state
74
75      def reset_from_state_and_torque(self, state, T):
76
77          self.s0 = state
78
79          initial_state = self.s0.copy()
80          for i in range(10):
81              initial_state, _, _ = self.take_step(initial_state, T)
82
83          self.t = 0.0
84          self.t_max = 30.0
85
86          return initial_state
87
88      def reset_from_state(self, state):
89          self.s0 = state
90
91          T = self.generate_random_torque()
92
93          initial_state = self.s0.copy()
94          for i in range(10):
95              initial_state, _, _ = self.take_step(initial_state, T)
96
97          self.t = 0.0
98          self.t_max = 30.0
99
100         return initial_state
101
102
```

```
103     def get_inertia_tensor(self):
104 
105         return self.J, self.Jinv
106 
107     def get_current_time(self):
108 
109         return self.t
110 
111     def action_to_torque(self, index):
112 
113         # Given an action index (integer between 0-6, inclusive)
114         # returns the corresponding torque vector.
115 
116         T = np.zeros(3)
117 
118         if index == 0:
119             pass
120         elif index == 1:
121             T[0] = -0.01
122         elif index == 2:
123             T[0] = 0.01
124         elif index == 3:
125             T[1] = -0.01
126         elif index == 4:
127             T[1] = 0.01
128         elif index == 5:
129             T[2] = -0.01
130         elif index == 6:
131             T[2] = 0.01
132 
133         return T
134 
135     def take_step_from_action(self, state, action):
136 
137         T = self.action_to_torque(action)
138 
139         new_state, reward, done = self.take_step(state, T)
140 
141         return new_state, reward, done
142 
143 
144 
145     def take_step(self, state, T):
146 
147         h = 0.01
148 
149 
150         q = state[0:4]
151         w = state[4:7]
152 
153         qn = q.copy()
154         wn = w.copy()
155 
156 
157         k1q = h * self.f1(qn, wn)
```

```
158         k1w = h * self.f2(qn, wn, T)
159 
160         in_1 = qn + 0.25 * k1q
161         in_2 = wn + 0.25 * k1w
162         k2q = h * self.f1(in_1, in_2)
163         k2w = h * self.f2(in_1, in_2, T)
164 
165         in_1 = qn + 3.0 / 32.0 * k1q + 9.0 / 32.0 * k2q
166         in_2 = wn + 3.0 / 32.0 * k1w + 9.0 / 32.0 * k2w
167         k3q = h * self.f1(in_1, in_2)
168         k3w = h * self.f2(in_1, in_2, T)
169 
170         in_1 = qn + 1932.0 / 2197.0 * k1q - 7200.0 / 2197.0 * k2q + 7296.0
    / 2197.0 * k3q
171         in_2 = wn + 1932.0 / 2197.0 * k1w - 7200.0 / 2197.0 * k2w + 7296.0
    / 2197.0 * k3w
172         k4q = h * self.f1(in_1, in_2)
173         k4w = h * self.f2(in_1, in_2, T)
174 
175         in_1 = qn + 439.0 / 216.0 * k1q - 8.0 * k2q + 3680.0 / 513.0 * k3q
    - 845.0 / 4104.0 * k4q
176         in_2 = wn + 439.0 / 216.0 * k1w - 8.0 * k2w + 3680.0 / 513.0 * k3w
     - 845.0 / 4104.0 * k4w
177         k5q = h * self.f1(in_1, in_2)
178         k5w = h * self.f2(in_1, in_2, T)
179 
180         in_1 = qn - 8.0 / 27.0 * k1q + 2.0 * k2q - 3544.0 / 2565.0 * k3q +
    1859.0 / 4104.0 * k4q - 11.0 / 40.0 * k5q
181         in_2 = wn - 8.0 / 27.0 * k1w + 2.0 * k2w - 3544.0 / 2565.0 * k3w +
    1859.0 / 4104.0 * k4w - 11.0 / 40.0 * k5w
182         k6q = h * self.f1(in_1, in_2)
183         k6w = h * self.f2(in_1, in_2, T)
184 
185         zq = qn + 16.0 / 135.0 * k1q + 6656.0 / 12825.0 * k3q + 28561.0 /
   56430.0 * k4q - 9.0 / 50.0 * k5q + 2.0 / 55.0 * k6q
186         zw = wn + 16.0 / 135.0 * k1w + 6656.0 / 12825.0 * k3w + 28561.0 /
   56430.0 * k4w - 9.0 / 50.0 * k5w + 2.0 / 55.0 * k6w
187 
188         z = np.concatenate((zq, zw), axis = 0)
189 
190 
191 
192         new_state = z.copy()
193         reward = self.calculate_reward(z)
194 
195         self.t += h
196 
197 
198 
199         # Check if any of the termination conditions occurs:
200         done = False
201 
202         if self.t >= self.t_max:
203             done = True
204 
```

```
205         return new_state, reward, done
206
207
208     def calculate_reward(self, state):
209
210         wi = state[4:7]
211         w0 = self.s0[4:7]
212
213         norma = np.linalg.norm(wi-w0)
214
215         g = (1.0 / (2*np.pi)**(0.5) ) * np.e ** (-0.5 * norma * norma)
216
217         return g
218
219     def f1(self, q, w):
220        wq = np.array([0,w[0],w[1],w[2]])
221        q_result = 0.5 * qprod(q, wq)
222
223        return q_result
224
225     def f2(self, q,w, T):
226         torque_Mu = T
227
228         w_result = - np.cross(self.Jinv @ w , self.J @ w) + self.Jinv @ (
    torque_Mu)
229
230         return w_result
```

satellite_enviroment_simplified.py

B.3.4 Controlador sense aprenentatge per reforç

Aquest programa importa *random_generator.py* i *satellite_enviroment.py*, i permet testejar el controlador de la Secció ***5.3***. Les llavors es poden canviar modificant les línies 100 i 101. Per limitar el torque que poden aplicar els actuadors es pot descomentar la línia 127.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from satellite_enviroment import SatelliteEnviroment, qprod
4  from random_generator import LCG
5
6  def Md(generator):
7
8     # Md generates a random enviormental perturbation torque.
9
10    T = np.zeros(3)
11
12    num1 = generator.randf()
13    num2 = generator.randf()
14    T[0] = 1e-3 * (-2*np.log(num1))**(0.5) * np.cos(2*np.pi*num2)
15
16    num1 = generator.randf()
17    num2 = generator.randf()
```

```
18      T[1] = 1e-3 * (-2*np.log(num1))**(0.5) * np.cos(2*np.pi*num2)
19
20      num1 = generator.randf()
21      num2 = generator.randf()
22      T[2] = 1e-3 * (-2*np.log(num1))**(0.5) * np.cos(2*np.pi*num2)
23
24      return T
25
26  def Mu_actuator(generator):
27
28       #This function generates a random perturbation torque for the
        actuators
29
30      T = np.zeros(3)
31
32      num1 = generator.randf()
33      num2 = generator.randf()
34      T[0] = 1e-3 * (-2*np.log(num1))**(0.5) * np.cos(2*np.pi*num2)
35
36      num1 = generator.randf()
37      num2 = generator.randf()
38      T[1] = 1e-3 * (-2*np.log(num1))**(0.5) * np.cos(2*np.pi*num2)
39
40      num1 = generator.randf()
41      num2 = generator.randf()
42      T[2] = 1e-3 * (-2*np.log(num1))**(0.5) * np.cos(2*np.pi*num2)
43
44      return T
45
46
47  def Mu(current_state, desired_state, J, Jinv):
48
49       q = current_state[0:4]
50       w = current_state[4:7]
51       qr = desired_state[0:4]
52       wr = desired_state[4:7]
53
54       qr_inv = -1 * qr.copy()
55       qr_inv[0] = qr_inv[0] * -1
56
57       q_err = qprod(qr_inv, q)
58       w_err = w - wr
59
60       J123 = np.zeros((3,3))
61       J123[0][0] = J[0][1] + J[0][2] - J[0][0]
62       J123[1][1] = J[1][0] + J[1][2] - J[1][1]
63       J123[2][2] = J[2][0] + J[2][1] - J[2][2]
64
65       Mr = np.cross(Jinv @ wr, J @ wr)
66       kq = 1
67       Kq = np.array([[2,1,0],[1,2,1],[0,1,2]])
68       Kw = Kq.copy()
69
70       qvec = np.array([q_err[1], q_err[2], q_err[3]])
71
```

```
72      M_control = Jinv @ Mr - (kq * np.sign(q[0]) ) * Kq @ qvec - Kw @ w_err
        - Jinv @ (np.cross(J123 @ wr, w_err)) - Jinv @ (- np.cross(w_err, J @
        w_err))
73
74      result = J @ M_control
75
76      return (result)
77
78  def calculate_error_state(s1, s_desired):
79
80      q = s1[0:4]
81      w = s1[4:7]
82
83      q_desired = s_desired[0:4]
84      w_desired = s_desired[4:7]
85
86      q_err = 1 - abs(q_desired[0]*q[0] + q_desired[1]*q[1] + q_desired[2]*q
        [2] + q_desired[3]*q[3])
87      w_err = np.linalg.norm(w - w_desired)
88
89      return q_err, w_err
90
91  # Data files
92  f_Mu = open("control_data.dat", "w")
93  f_error = open("error_data.dat", "w")
94
95  # Time varibles
96  t = 0
97  t_max = 30
98
99  # Seeds for the random number generator(s):
100 seed_perturbation = 123
101 seed_enviroment = 1
102
103 # Random number generator, used to generate the perturbations:
104 generator = LCG(seed_perturbation)
105
106
107 # We initialize our satellite enviroment:
108 sat = SatelliteEnviroment(seed_enviroment)
109 state = sat.reset()
110
111 # We save our initial state (it is the attitude right after we apply the
        torque):
112 initial_noisy_state = state.copy()
113
114 # We store our desired (reference) state. (For debugging purposes)
115 desired_state = sat.s0
116
117 # The controller requires the inertia tensor to perform its calculations:
118 J, Jinv = sat.get_inertia_tensor()
119
120 # Main loop:
121 t = 0
122 while t <= 30:
```

```
123
124     # Generate control torque (Mu) from current state
125     T = Mu(state, desired_state, J, Jinv)
126     # We could limit the amount of torque that the actuators are able to
        produce:
127     #np.clip(T,-0.0021,0.0021,out=T)
128
129     # Write the current control torque values on a file:
130     f_Mu.write(str(t) + " " + str(T[0]) + " " + str(T[1]) + " " + str(T
        [2]) + "\n")
131
132
133     # We add some perturbation (due to the enviroment + actuator
        imperfections)
134     T = T + Md(generator) + Mu_actuator(generator)
135
136
137
138     # Calculate enviroment step from current state, applying Mu:
139     next_state, _, _ = sat.take_step(state, T)
140
141     # Calculate the current attitude error (inf. norm) and write it on a
        file:
142     q_norm, w_norm = calculate_error_state(next_state, desired_state)
143     f_error.write(str(t) + " " + str(q_norm) + " " + str(w_norm) + "\n")
144
145     # Get the current simulation time and print it (along with the aprox.
        attitude error)
146     t = sat.get_current_time()
147     #print("t = ", t, " | q error =", q_norm, " | w error = ", w_norm)
148
149     # Update current state
150     state = next_state.copy()
151
152 # Close files
153 f_Mu.close()
154 f_error.close()
155
156 # Show a brief summary to the user. Ideally, the final attitude and the
       desired attitude
157 # vectors shoud be very close...
158 print("==============================")
159 print("Attitude after " + str(t_max) + " sec. :")
160 print("q = ", state[0:4])
161 print("w = ", state[4:7])
162 print("\n Desired attitude:")
163 print("q = ", sat.s0[0:4])
164 print("w = ", sat.s0[4:7])
165 print("\n Initial (noisy) attitude:")
166 print("q = ", initial_noisy_state[0:4])
167 print("w = ", initial_noisy_state[4:7])
```

classic_controller.py

B.3.5 Controlador amb reinforcement-learning

El següent programa implementa l'algorisme DQN per entrenar una xarxa neuronal capaç de controlar un satèl·lit artificial, seguint la metodologia que es detalla a [6]. Com s'ha explicat en vàries ocasions al llarg del cos de la memòria, aquest programa no ha pogut ser ben testejat per falta de potència de càlcul per realitzar l'entrenament. Per aquesta raó és possible que, a la pràctica, s'hagi d'ajustar algun dels hiperparàmetres de l'algorisme per tal que l'entrenament sigui òptim.

```
import random
import numpy as np
from collections import deque

import keras
import tensorflow as tf
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.optimizers import SGD
from satellite_enviroment import SatelliteEnviroment


class DQNAgent:
    def __init__(self, state_size = 7, action_size = 7):
        self.state_size = state_size # Dimension of each state (7 equals 1 x7)
        self.action_size = action_size # Num. of possible actions

        self.memory = deque(maxlen=500) # Memory buffer, to store training ex.
        self.batch_size = 32
        self.train_start = 100 # Training starts when buffer has 100 examples

        self.gamma = 0.95 # Bellman equation update parameter

        self.epsilon = 1.0 # Exploration chance
        self.epsilon_min = 0.001 # Minimum posible exporation chance
        self.epsilon_decay = 0.99 # Exploration chance decay factor

        # Generate and compile our model:
        self.build_network()

    def build_network(self):
        X_input = Input((self.state_size,))

        # As stated in the research paper, the model consists of an input layer,
        # plus two dense hidden layers (1024 and 2048 neurons), plus linear
        # activation layer.
        X = Dense(1024, input_shape=(self.state_size,), activation="sigmoid")(X_input)
```

```
38          X = Dense(2048, activation="sigmoid")(X)
39          X = Dense(self.action_size, activation="linear")(X)
40
41          self.model = Model(inputs = X_input, outputs = X, name='
    Satellite_NN')
42          self.model.compile(loss="mse", optimizer=SGD(learning_rate=0.001),
     metrics=["accuracy"])
43
44
45      def remember(self, state, action, reward, next_state, done):
46          self.memory.append((state, action, reward, next_state, done))
47          if len(self.memory) > self.train_start and self.epsilon > self.
    epsilon_min:
48              self.epsilon = self.epsilon_decay * self.epsilon
49
50      def act(self, state):
51          if np.random.random() <= self.epsilon:
52              return np.random.randint(7)
53          else:
54              return np.argmax(self.model.predict(state))
55
56      def learn_from_experience(self):
57          if len(self.memory) < self.train_start:
58              return
59
60          # Randomly sample minibatch from the memory
61          minibatch = random.sample(self.memory, self.batch_size)
62
63          # Split the sample minibatch:
64          state = np.zeros((self.batch_size, self.state_size))
65          next_state = np.zeros((self.batch_size, self.state_size))
66          action, reward, done = [], [], []
67          for i in range(self.batch_size):
68              state[i] = minibatch[i][0]
69              action.append(minibatch[i][1])
70              reward.append(minibatch[i][2])
71              next_state[i] = minibatch[i][3]
72              done.append(minibatch[i][4])
73
74          # We predict the target values for the current and next state:
75          target = self.model.predict(state)
76          target_next = self.model.predict(next_state)
77
78          # We calculate the Q-values according to the Bellman Equation:
79          for i in range(self.batch_size):
80              if done[i]:
81                  target[i][action[i]] = reward[i]
82              else:
83                  target[i][action[i]] = reward[i] + self.gamma * (np.amax(
    target_next[i]))
84
85          # Perform SGD and update the model weights:
86          self.model.fit(state, target, batch_size=self.batch_size, verbose
    =0)
87
```

```
88 if __name__ == "__main__":
89
90     # Uncomment these next 3 lines if your computer has a GPU. (also:
       remember to adjust CPU cores!)
91     #config = tf.compat.v1.ConfigProto( device_count = {'GPU': 1 , 'CPU':
       68} )
92     #sess = tf.compat.v1.Session(config=config)
93     #keras.backend.set_session(sess)
94
95     # We create the RL agent:
96     agent = DQNAgent(7,7)
97
98     max_episodes = 500 # The max number of training episodes
99
100    # Create the enviroment:
101    env_seed = 123
102    env = SatelliteEnviroment(env_seed)
103
104    # Begin training...
105    for ep in range(max_episodes):
106        print("Episode {}/{} has begun.".format(ep, max_episodes))
107
108        # We reset the current example:
109        state = env.reset()
110        step = 0
111        total_reward = 0
112        done = False
113
114        # While t < 30 sec...
115        while not done:
116
117            # The agent chooses an action:
118            action = agent.act(np.reshape(state, [1, agent.state_size]))
119
120            # We perform this action in the enviroment:
121            next_state, reward, done = env.take_step_from_action(state,
       action)
122
123            # We add the reward (in order to calculate the mean)
124            total_reward += reward
125
126            # We add the current "step" to the replay buffer:
127            agent.remember(np.reshape(state, [1, agent.state_size]),
       action, reward, np.reshape(next_state, [1, agent.state_size]), done)
128            state = next_state
129
130            # We increase the step counter
131            step += 1
132
133            # If step == 1500 (t = 15 sec.) we are halfway through the
       simulation
134            if step == 1500:
135                print("Episode at 50%...")
136
```

```
137             # If we are done, we show a summary of the episode and save
    the trained model:
138             if done:
139                 print("Episode: {}/{}, Avg. Reward: {}, epsilon = {:.4} |
    Final rew. = {}, Inf. error = {}".format(ep, max_episodes,
    total_reward / step, agent.epsilon, reward,  max(abs(next_state[4:7]))
    ), " Final w = ", next_state[4:7])
140                 agent.model.save("episodes/satellite-{}.h5".format(ep))
141 
142             # The agent learns from the experience pool:
143             agent.learn_from_experience()
144 
145     # We save the trained model:
146     agent.model.save("satellite_finale.h5")
```

reinforcement_learning_FullNN.py

B.3.6 Controlador amb reinforcement-learning simplificat

Aquest és l'algorisme DQN simplificat que s'ha utilitzat a la pràctica per entrenar el controlador.

```
1 
2 import random
3 import numpy as np
4 from collections import deque
5 
6 import keras
7 import tensorflow as tf
8 from tensorflow.keras.models import Model, load_model
9 from tensorflow.keras.layers import Input, Dense
10 from tensorflow.keras.optimizers import SGD
11 from satellite_enviroment_simplified import SatelliteEnviroment
12 
13 
14 class DQNAgent:
15     def __init__(self, state_size = 7, action_size = 7):
16         self.state_size = state_size # Dimension of each state (7 equals 1
    x7)
17         self.action_size = action_size # Num. of possible actions
18 
19         self.memory = deque(maxlen=500) # Memory buffer, to store training
     ex.
20         self.batch_size = 32
21         self.train_start = 100 # Training starts when buffer has 100
    examples
22 
23         self.gamma = 0.95 # Bellman equation update parameter
24 
25         self.epsilon = 1.0 # Exploration chance
26         self.epsilon_min = 0.001 # Minimum posible exporation chance
27         self.epsilon_decay = 0.99 # Exploration chance decay factor
28 
```

```
29          # Generate and compile our model:
30          self.build_network()
31
32      def build_network(self):
33          X_input = Input((self.state_size,))
34
35          X = Dense(128, input_shape=(self.state_size,), activation="sigmoid
    ")(X_input)
36          X = Dense(256, activation="sigmoid")(X)
37          X = Dense(self.action_size, activation="linear")(X)
38
39          self.model = Model(inputs = X_input, outputs = X, name='
    Satellite_NN')
40          self.model.compile(loss="mse", optimizer=SGD(learning_rate=0.001),
     metrics=["accuracy"])
41
42
43      def remember(self, state, action, reward, next_state, done):
44          self.memory.append((state, action, reward, next_state, done))
45          if len(self.memory) > self.train_start and self.epsilon > self.
    epsilon_min:
46              self.epsilon = self.epsilon_decay * self.epsilon
47
48      def act(self, state):
49          if np.random.random() <= self.epsilon:
50              return np.random.randint(7)
51          else:
52              return np.argmax(self.model.predict(state))
53
54      def learn_from_experience(self):
55          if len(self.memory) < self.train_start:
56              return
57
58          # Randomly sample minibatch from the memory
59          minibatch = random.sample(self.memory, self.batch_size)
60
61          # Split the sample minibatch:
62          state = np.zeros((self.batch_size, self.state_size))
63          next_state = np.zeros((self.batch_size, self.state_size))
64          action, reward, done = [], [], []
65          for i in range(self.batch_size):
66              state[i] = minibatch[i][0]
67              action.append(minibatch[i][1])
68              reward.append(minibatch[i][2])
69              next_state[i] = minibatch[i][3]
70              done.append(minibatch[i][4])
71
72          # We predict the target values for the current and next state:
73          target = self.model.predict(state)
74          target_next = self.model.predict(next_state)
75
76          # We calculate the Q-values according to the Bellman Equation:
77          for i in range(self.batch_size):
78              if done[i]:
79                  target[i][action[i]] = reward[i]
```

```
80              else:
81                  target[i][action[i]] = reward[i] + self.gamma * (np.amax(
    target_next[i]))
82
83          # Perform SGD and update the model weights:
84          self.model.fit(state, target, batch_size=self.batch_size, verbose
    =0)
85
86 if __name__ == "__main__":
87
88      # Uncomment these next 3 lines if your computer has a GPU. (also:
    remember to adjust CPU cores!)
89      #config = tf.compat.v1.ConfigProto( device_count = {'GPU': 1 , 'CPU':
    68} )
90      #sess = tf.compat.v1.Session(config=config)
91      #keras.backend.set_session(sess)
92
93      # We create the RL agent:
94      agent = DQNAgent(7,7)
95
96      max_episodes = 500 # The max number of training episodes
97
98      # Create the enviroment:
99      env_seed = 123
100     env = SatelliteEnviroment(env_seed)
101
102     # Begin training...
103     for ep in range(max_episodes):
104         print("Episode {}/{} has begun.".format(ep, max_episodes))
105
106         # We reset the current example:
107         state = env.reset()
108         step = 0
109         total_reward = 0
110         done = False
111
112         # While t < 30 sec...
113         while not done:
114
115             # The agent chooses an action:
116             action = agent.act(np.reshape(state, [1, agent.state_size]))
117
118             # We perform this action in the enviroment:
119             next_state, reward, done = env.take_step_from_action(state,
    action)
120
121             # We add the reward (in order to calculate the mean)
122             total_reward += reward
123
124             # We add the current "step" to the replay buffer:
125             agent.remember(np.reshape(state, [1, agent.state_size]),
    action, reward, np.reshape(next_state, [1, agent.state_size]), done)
126             state = next_state
127
128             # We increase the step counter
```

```
129             step += 1
130 
131             # If step == 1500 (t = 15 sec.) we are halfway through the
    simulation
132             if step == 1500:
133                 print("Episode at 50%...")
134 
135             # If we are done, we show a summary of the episode and save
    the trained model:
136             if done:
137                 print("Episode: {}/{}, Avg. Reward: {}, epsilon = {:.4} |
    Final rew. = {}, Inf. error = {}".format(ep, max_episodes,
    total_reward / step, agent.epsilon, reward,  max(abs(next_state[4:7]))
    ), " Final w = ", next_state[4:7])
138                 agent.model.save("episodes/satellite-{}.h5".format(ep))
139 
140             # The agent learns from the experience pool:
141             agent.learn_from_experience()
142 
143     # We save the trained model:
144     agent.model.save("satellite_finale.h5")
```

reinforcement_learning_simplifiedNN.py