Apéndice A

Varios aspectos matemáticos

A.1. Geometrización del formalismo de Schrödinger

En esta sección comentamos brevemente cómo se lleva a cabo la geometrización del formalismo de Schrödinger (que suponemos conocido) para sistemas de dimensión finita. Ver [4] para una introducción detallada y rigurosa.

Si $\{|b_i\rangle\}$ es una base del espacio de Hilbert del sistema cuántico (de dimensión finita n), entonces

$$|\psi_e\rangle = \sum c_j |b_j\rangle = \sum q_j + ip_j |b_j\rangle \equiv (q_1, p_1, \ldots, q_n, p_n) \qquad q_i, p_i \in \mathbb{R}$$

Ahora el conjunto de posibles estados parametrizados por $(q_1, p_1, \ldots, q_n, p_n)$ forma una variedad[1] que denotaremos como M_Q. Sobre el espacio tangente a dicha variedad definimos un campo vectorial que *groso modo* viene a ser otra copia del sistema. Si $|v\rangle = \sum v_j |b_j\rangle = \sum q_j + ip_j |b_j\rangle$, este campo toma la forma

$$X^v := \sum_i q_i \frac{\partial}{\partial q_i} + p_i \frac{\partial}{\partial p_i} \qquad q_i, p_i \in \mathbb{R} \qquad \frac{\partial}{\partial q_i}, \frac{\partial}{\partial p_i} \in T_v M$$

usando notación habitual de geometría diferencial. Lo necesitamos porque no podemos hacer operaciones como productos internos sobre puntos, sino sobre vectores. Sobre el espacio tangente se puede traducir el producto interno del espacio de Hilbert de dos estados

$$\langle v|w\rangle = \Re(\langle v|w\rangle) + i\,\Im(\langle v|w\rangle) := g(X^v, X^w) + i\omega(X^v, X^w)$$

denotando por X^v y X^w los campos vectoriales asociados a $|v\rangle$ y $|w\rangle$. El objeto g es simétrico y nos define una *métrica*. El objeto ω es antisimétrico y nos define una *forma simpléctica*.[2] Para acabar la traducción del producto interno, debemos definir un objeto J que sea equivalente a la unidad imaginaria i. Se define un campo tensorial (1,1) que cumple

$$J^2 = -\mathbb{1} \qquad g(JX^v, JX^w) = g(X^v, X^w) \qquad \omega(X^v, X^w) = g(JX^v, JX^w)$$

[1] Además es una variedad "plana", significando que el espacio tangente es isomorfo a la misma variedad. Esto es porque esencialmente estamos cogiendo las partes reales e imaginarias del espacio vectorial de Hilbert, y los espacios vectoriales son variedades planas o euclídeas.

[2] Para ganar intuición, cogiendo una base ortonormal del espacio tenemos: $\langle v|w\rangle = \sum(p_{vj} - iq_{vj})(p_{wj} + iq_{wj}) = \sum\left((q_{vj}, p_{vj}) \cdot (q_{wj}, p_{wj}) + i\begin{vmatrix} q_{vj} & q_{wj} \\ p_{vj} & p_{wj} \end{vmatrix}\right) := g(X, X') + i\omega(X, X')$ siendo X, X' los campos asociados a v y w. Tiene sentido usar la palabra "métrica" para g y forma para ω a luz de sus analogías con el producto escalar y el determinante.

Ahora, al conjunto (M_Q, g, ω, J) de la variedad y las estructuras definidas se le llama variedad de Kähler. Comentemos que lo novedoso que introduce la mecánica cuántica es la métrica g y el tensor J; en ausencia de estos objetos reproduciríamos la mecánica clásica habitual.

Llegados a este punto, tenemos descrito en términos geométricos el espacio de estados. En lo que concierne a los observables, lo que realmente es importante para las predicciones físicas es el valor esperado de un operador y sus autovalores y autoestados. Si definimos la función e_O asociada a un operador O como su valor medio

$$e_O(|\psi\rangle) = \frac{\langle\psi|O|\psi\rangle}{\langle\psi|\psi\rangle}$$

entonces ésta caracteriza completamente la física y podemos realizar las mismas predicciones experimentales. Nótese que si $|\psi\rangle$ es un autoestado de O, entonces $e_O(|\psi\rangle)$ es el valor del autovalor. Para hallar los autoestados, podemos imponer $de_O(v) = 0$, porque tenemos un mínimo, máximo o punto silla cuando estamos en un estado propio.[3]

Observemos que a todos los estados proporcionales entre sí les corresponde mismo valor esperado e_O. Para eliminar esta redundancia, es habitual trabajar con el *rayo* que denotamos $[|\psi\rangle]$ (el conjunto de rayos forma el espacio proyectivo asociado al espacio de Hilbert). Se puede geometrizar el espacio proyectivo, siendo una subvariedad de M_Q, $\mathcal{P} \subset M_Q$. También hay que definir nuevos objetos tensoriales para reducir sobre ella g y ω, los previamente introducidos sobre M_Q. Podemos llegar a definir un corchete de Poisson sobre el proyectivo, $\{\cdot,\cdot\}_{\mathcal{P}}$. Para más detalles matemáticos ver [13, Cap. 1], [18, 4].

Dado el corchete de Poisson $\{\cdot,\cdot\}_{\mathcal{P}}$, se puede definir el campo hamiltoniano asociado al valor esperado del hamiltoniano $e_{\mathcal{H}}$, $X_{e_{\mathcal{H}}}$. El flujo de este campo corresponde a la evolución temporal del sistema, y conlleva a la conservación del valor esperado de la energía.

A.2. Ecuación de Hamilton-Jacobi

Para esta sección se ha consultado principalmente [26]. Para conocer la evolución temporal de la posición $\mathbf{r}(t)$, podemos integrar el campo de velocidad:

$$\frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t} = \mathbf{v}(\mathbf{r}(t), t) \tag{A.1}$$

a su vez el campo de velocidades lo podemos conocer integrando su derivada

$$\frac{\mathrm{d}\mathbf{v}(\mathbf{r}(t),t)}{\mathrm{d}t} = \frac{\mathbf{F}(\mathbf{r}(t))}{m} = \frac{\partial\mathbf{v}}{\partial t}\frac{\mathrm{d}t}{\mathrm{d}t} + \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}\mathbf{r}}\cdot\frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t} = \frac{\partial\mathbf{v}}{\partial t} + (\mathbf{v}\cdot\nabla)\mathbf{v} \tag{A.2}$$

donde en la última parte se ha desarrollado la derivada total.

Ahora, se debe comentar que ya hemos hecho una simplificación al supone que la fuerza no depende de la velocidad ni del tiempo, sino de $\mathbf{r}$ únicamente. Además, se exige que la fuerza sea conservativa, es decir $\mathbf{F} = -\nabla U(\mathbf{r})$. Entonces, cogiendo coordenadas (para ver más fácilmente

[3] Para demostrar la propiedad, basta con coger una variación del autoestado, $|\psi\rangle + t\,|w\rangle$, coger la derivada con respecto a el parámetro t y ver que se anula $\frac{\mathrm{d}}{\mathrm{d}t}\frac{\langle\psi+tw|O|\psi+tw\rangle}{\langle\psi+tw|\psi+tw\rangle}\Big|_{t=0} = 0$

los desarrollos, tenemos), llamando $mv_j = p_j$ y despejando

$$\frac{\partial mv_i}{\partial t} + \sum_j \frac{1}{m}(mv_j \cdot \partial_j)mv_i + \partial_i U = 0 \tag{A.3}$$

Ahora, hemos reducido trabajar con magnitudes vectoriales con la fuerza introduciendo un potencial. Podemos intentar hacer lo mismo con el momento, $\mathbf{p} = \nabla S(\mathbf{r}, t)$. El hecho de ser el gradiente de un escalar implica que el rotacional de p es nulo, es decir

$$\partial_i p_j - \partial_j p_i = 0 \rightarrow \partial_i p_j = \partial_j p_i$$

Con todo esto en mente, la Eq. (A.3) queda

$$\frac{\partial p_i}{\partial t} + \sum_j \frac{1}{2m}\partial_i(p_j^2) + \partial_i U = 0 \rightarrow \partial_i \left[\frac{\partial S}{\partial t} + \sum_j \frac{\mathbf{p}^2}{2m} + U\right] = 0 \tag{A.4}$$

llegando a la ecuación de Hamilton-Jacobi[4]

$$\frac{\partial S}{\partial t} + \frac{\mathbf{p}^2}{2m} + U(\mathbf{r}) = \frac{\partial S}{\partial t} + H(\mathbf{r}, \mathbf{p}, t) = 0$$

La dependencia temporal de t en $H(\mathbf{r}, \mathbf{p}, t)$ no se ha considerado en este desarrollo, pero se puede ver mediante desarrollos matemáticos rigurosos [1, Cap. 9] que se puede mantener.

A.3. Formalismo de Heisenberg y geometrización

En un primer encuentro con la mecánica cuántica, uno se enfrenta a la ecuación de Schrödinger

$$i\frac{\mathrm{d}}{\mathrm{d}t}\psi_e = \mathcal{H}_e\psi_e$$

que describe la evolución del sistema cuántico. Un resultado básico de mecánica cuántica es la equivalencia de todos los estados que son proporcionales entre sí (todos estos estados $|\psi\rangle$ forman un "rayo", $[\psi]$). Normalmente uno se queda únicamente con estados de norma unidad como representantes de los rayos. Equivalentemente, podríamos hablar de *proyectores* (la relación entre un rayo y un proyector sobre el subespacio de dimensión 1 generado por él mismo es 1-1), definiendo la *aplicación momento* respecto a la acción natural del grupo unitario $U(n)$ sobre el espacio de Hilbert o la correspondiente acción sobre el proyectivo (ver [27]):

$$\mu(|[\psi]\rangle) = \frac{|\psi\rangle\langle\psi|}{\langle\psi|\psi\rangle} := \rho_\psi \tag{A.5}$$

que a cualquier estado le asigna el proyector correspondiente. Entonces un estado cuántico puro es equivalente a un proyector, que denotaremos como ρ_ψ. Su evolución temporal, usando la ecuación de Schrödinger, se puede ver que viene dada por

$$i\frac{\mathrm{d}\rho_\psi}{\mathrm{d}t}(t) = \dot{\rho}_\psi = -i[\mathcal{H}_e, \rho(t)_\psi]$$

[4] La derivada debe ser constante, y dicha constante se puede abosrber en el potencial U.

Llegados a este punto, podemos generalizar y definir los *estados mixtos* como sumas convexas de proyectores

$$\rho = \sum \lambda_i \rho_\psi$$

donde λ_i se interpreta como la probabilidad estar el sistema cuántico en el estado puro $|\phi_i\rangle$. La gran utilidad de los operadores de densidad está en poder trabajar con sistemas mixtos. Para hallar los valores medios, se usa $\langle \mathcal{H}_e \rangle_\rho \equiv \langle \mathcal{H}_e \rangle \equiv \operatorname{tr} \rho \mathcal{H}_e$, siendo $\operatorname{tr}(\cdot)$ la traza de un operador.

La traza denota la suma de los autovalores de un operador, y es independiente de la base. Es importante remarcar que la traza nos permite definir un producto interno para el espacio de operadores lineales,

$$\langle A, B \rangle = \operatorname{tr} AB \quad \forall A, B \in gl(n, \mathbb{C})$$

Por comodidad normalizaremos la traza sobre los operadores autoadjuntos, $\operatorname{tr} A \to \operatorname{tr} A / \operatorname{tr} \mathbb{1}$. En nuestro caso para un sistema de dos niveles, $1/\operatorname{tr} \mathbb{1} = 1/2$. Definiremos así

$$\langle A, B \rangle := \frac{1}{2} \operatorname{tr} AB,$$

para cualquier par de operadores autoadjuntos.

Sobre la geometrización del formalismo de Heisenberg, sobre el espacio de operadores autoadjuntos podemos considerar dos objetos tensoriales asociados a las estructuras de Lie y de Jordan, definidas por el conmutador y el anticonmutador de operadores (ver [4, 27]). En particular, asociado a la estructura de Lie aparece la estructura de Lie-Poisson, que dota al conjunto de operadores autoadjuntos (identificados a su espacio dual por el producto escalar antes mencionado) de una estructura de Poisson canónica. Respecto a esta estructura de Poisson, podemos considerar campos Hamiltonianos asociados a las funciones sobre el espacio de operadores.

Si consideramos la restricción de la estructura de Poisson a la subvariedad de estados puros, realizada como proyectores sobre subespacios de dimensión 1, es sencillo verificar que ésta corresponde a una hoja simpléctica de la foliación de la variedad de Poisson. Puede probarse también (ver [4]) que la estructura simpléctica correspondiente es la imagen por el *pullback* (de la inversa) de la aplicación momento antes mencionada (Ec. A.5) de la estructura simpléctica canónica del espacio proyectivo $\omega_{\mathcal{P}}$. Se puede demostrar también que la imagen del campo Hamiltoniano asociado a la función $e_{\mathcal{H}}$ en $\mathcal{P}$, por el *push-forward* de la aplicación momento, define entonces el campo Hamiltoniano asociado a la hoja simpléctica de la estructura de Lie-Poisson, respecto al operador $\mathcal{H}$.

A.4. Caracterización de la esfera de Bloch

En la Sección A.3 del apéndice hemos comentado la descripción equivalente de estados cuánticos mediante los operadores matriz de densidad. En esta sección mencionaremos brevemente varias de sus propiedades y su caracterización por la esfera de Bloch.

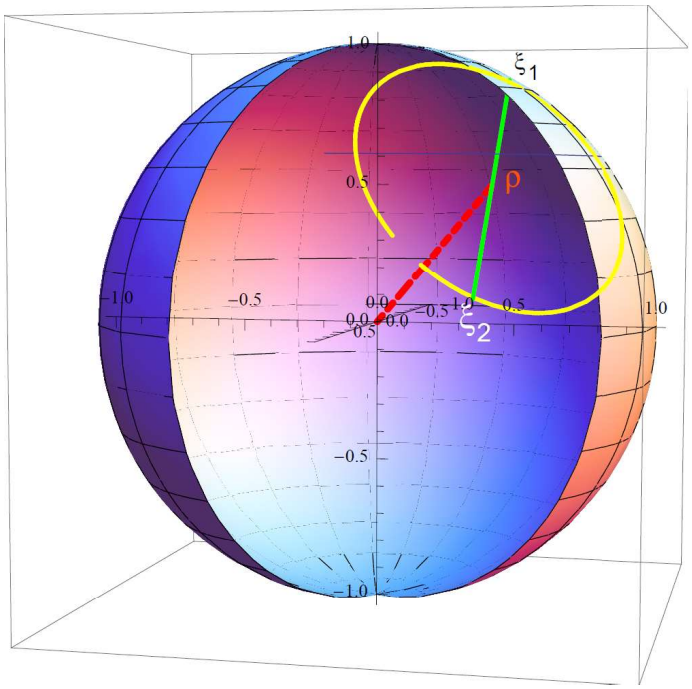

FIGURA A.1: En esta figura se puede observar una representación de la esfera de Bloch. Sobre ella están representados estados putos (círculo amarillo), teniendo a un operador matriz de densidad ρ genérico como combinación convexa de puntos de la superficie. Los puntos de la superficie denotan estados puros y los del interior, estados mixtos. Fuente de la imagen: [4].

Recordemos que

$$\rho = \sum_i \lambda_i \frac{|\psi_i\rangle\langle\psi_i|}{\langle\psi_i|\psi_i\rangle}$$

donde λ_i denota la probabilidad clásica de estar el sistema en el estado puro correspondiente a $|\psi_i\rangle$. La matriz densidad debe cumplir siempre $\operatorname{tr}\rho = 1$, porque la suma de todas las probabilidades debe de ser uno. Además $\rho \geq 0$, significando que su espectro es positivo (tiene sentido al interpretarse como probabilidades).

Considerando la base del espacio de operadores de dimensión dos, $\{\mathbb{1}, \sigma_x, \sigma_y, \sigma_z\}$, es fácil demostrar que los operadores matriz de densidad deben ser de la forma

$$\rho(t) = \frac{1}{2}(\mathbb{1} + \rho_x(t)\sigma_x + \rho_y(t)\sigma_y + \rho_z(t)\sigma_z) = \frac{1}{2}(\mathbb{1} + \boldsymbol{\rho}\cdot\boldsymbol{\sigma})$$

como consecuencia de la anterior propiedad ($\operatorname{tr}\rho = 1$), teniendo en cuenta que la traza es lineal, ($\operatorname{tr}(A+B) = \operatorname{tr}A + \operatorname{tr}B$) y que las matrices de Pauli tienen traza nula ($\operatorname{tr}\sigma_i = 0$). Por lo tanto, hemos reducido el número de coordenadas necesarias para describir un estado de cuatro a tres. Además, se puede ver que las matrices de densidad cumplen

$$0 < \operatorname{tr}\rho^2 = \rho_x^2 + \rho_y^2 + \rho_z^2 \leq 1$$

Esta propiedad tiene sentido: si $\sum\lambda_i = 1$, entonces obviamente $\sum\lambda_i^2 \leq 1$ porque $\lambda_i^2 < \lambda_i$ al

ser todos los λ_i positivos y menores o iguales que uno. Esta última propiedad nos reduce en el espacio tridimensional parametrizado por (ρ_x, ρ_y, ρ_z) a una esfera de radio 1. A esta esfera se le conoce como **esfera de Bloch**, y el "polo norte" corresponde a un estado cuántico y el "polo sur" al otro del sistema de dos niveles. Ver la Figura A.1 para una representación gráfica de la esfera.

El operador matriz de densidad ρ actúa sobre un observable, como por ejemplo el hamiltoniano electrónico del texto principal, $\mathcal{H}_e$ como sigue

$$\operatorname{tr} \rho \mathcal{H}_e = \langle \mathcal{H}_e \rangle$$

y su evolución temporal es

$$i\frac{\mathrm{d}\rho}{\mathrm{d}t}(t) = \dot{\rho} = -i[\mathcal{H}_e, \rho(t)]$$

devolviendo el valor esperado de dicho observable.

El valor medio de $\mathcal{H}_e$ en función de las coordenadas de ρ es $\operatorname{tr} \rho \mathcal{H}_e = \rho_i \mathcal{H}_i$ (para demostrarlo se debe tener en cuenta que $\sigma_a \sigma_b = \delta_{ab} \mathbb{1} + i\epsilon_{abc}\sigma_c$, donde ϵ_{abc} es el símbolo de Levi-Civita[5]). Para reescribir $-i[\mathcal{H}, \rho]$ se puede usar que $[\sigma_a, \sigma_b] = 2i\epsilon_{abc}\sigma_c$ y las definiciones, $\mathcal{H} = \mathcal{H}_i \sigma_i$ y $\rho = \frac{1}{2}(\mathbb{1} + \rho_j \sigma_j)$, dando lugar a $-i[\mathcal{H}, \rho] = \epsilon_{ijk}\mathcal{H}_i \rho_j \sigma_k$.

[5]Un objeto antisimétrico bajo permutaciones de los índices y no nulo salvo en el caso en el cual los índices *abc* son una permutación de 123, teniendo $\epsilon_{123} = 1$.

Apéndice B

Código Python para implementar el test de controlabilidad

En este código se calculan simbólicamente los corchetes de Lie de unos campos vectoriales y se comprueba que estos generan el espacio tangente. Ver el repositorio de GitHub del siguiente enlace para descargar cómodamente el código.

https://github.com/trahald/TFG

En el caso de no funcionar el enlace, contacte con el autor.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

### This is code which given some vector field X_1,X_2,..., generates
###  the corresponding space Lie(X_1,X_2,...) and checks whether it
### spans the tangent space or not.

from __future__ import division # so that 1/2 does not give the loathed 0
from IPython.display import display, Math, Latex # beautiful output

from sympy import *
import numpy as np
import cmath

init_printing()  # For pretty printing.

####### GLOBAL CONTROLS ######################################################

COUPLING_CONSTANT   = 1.0

MASS      = 1.0
K_SPRING = 1.0
GAMMA     = 1.0

####### Defining most of the variables #######################################

t = symbols('t', real=True);
q,p,rx,ry,rz= symbols('q p r_x r_y r_z');
```

```
epsilon = symbols('epsilon', real=True);

sig0   = Matrix([[1, 0],[0, 1]])#/sqrt(2)
sig1   = Matrix([[0, 1],[1, 0]])#/sqrt(2)
sig2   = Matrix([[0,-I],[I, 0]])#/sqrt(2)
sig3   = Matrix([[1, 0],[0,-1]])#/sqrt(2)
u1base = Matrix(4,1,([sig0,sig1,sig2,sig3]))

################################### New hamiltonian.

#######
t = symbols('t', real=True);

q,p,rx,ry,rz,Hx,Hy,Hz         = symbols('q p r_x r_y r_z H_x H_y H_z');
cq,cp,crx,cry,crz,cHx,cHy,cHz = symbols(
                    'c_q c_p c_rx c_ry c_rz c_Hx c_Hy c_Hz');

LeviCivita3 = np.array([[[int((i - j) * (j - k) * (k - i) / 2)
                    for k in range(3)] for j in range(3)]
                    for i in range(3)], dtype=np.float64)

x_tot     = Matrix([q,p,rx,ry,rz,Hx,Hy,Hz,cq,cp,crx,cry,crz,cHx,cHy,cHz])
size_xtot = int(len(x_tot))
x         = Matrix([q,p,rx,ry,rz,Hx,Hy,Hz])
cx        = Matrix([cq,cp,crx,cry,crz,cHx,cHy,cHz])
r         = Matrix([rx,ry,rz])
H         = Matrix([Hx,Hy,Hz])
cr        = Matrix([crx,cry,crz])
cH        = Matrix([cHx,cHy,cHz])

VARIABLES = Matrix([p,q,rx,ry,rz,Hx,Hy,Hz])

m, k = symbols('m k', real=True);

gamma = GAMMA

epsilon = symbols('epsilon', real=True);

####### THE FUNCTION f(q) which gives the coupling strength.

#f = 1/sqrt(1+q**2)
#f = 1 + COUPLING_CONSTANT*(q**2 + p**2)
#f=1 + COUPLING_CONSTANT*q*p
f=q

def sum1 ():
    sum1 = 0
    for i in range(0,3):
        for j in range(0,3):
            for k in range(0,3):
```

```
                sum1 = sum1 + -I*2*I* LeviCivita3[i,j,k] * r[j] * H[i] * cr[k]
    return sum1;


# This is Pontryagin's hamiltonian, see Pontryagin's theorem.
h = ( cq*p/m + cp*(-k*q -2* (rx*Hx+ry*Hy+rz*Hz)*(diff(f,q))) + sum1() +
     1/2*(cHx*cHx + cHy*cHy + cHz*cHz) )
h = simplify(h)

# The dynamics are given by δh/δc_x
xtotdot = []
for i in range(0,int(len(x_tot)/2)):
    xtotdot.append( diff(h,cx[i]))
#for i in range(0,int(len(x_tot)/2)):
#    xtotdot.append(-diff(h,x[i]))

# 'velocity' is the vector field which gives the dynamics, ẋ = X
velocity = Matrix(xtotdot)


# Now we decompose it in the form
X_0 = simplify(   velocity.subs(cHx,0).subs(cHy,0).subs(cHz,0))
X_1 = simplify(  (velocity-X_0).subs(cHy,0).subs(cHz,0)/cHx  )
X_2 = simplify(  (velocity-X_0).subs(cHx,0).subs(cHz,0)/cHy  )
X_3 = simplify(  (velocity-X_0).subs(cHx,0).subs(cHy,0)/cHz  )
X   = X_0 + cHx*X_1 + cHy*X_2 + cHz*X_3


################################################################################
####### COMMUTATORS ############################################################
################################################################################

# We assume next that x_a,x_b are components of vectors X_a, X_b in the basis
# v[0] = ∂_q, v[1] = ∂_p,
# v[2,3,4] = ∂_ρ,
# v[5,6,7] = ∂_H,
# Now, by analyzing [X_a,X_b] by hand where (implicit sum over repeated
# indices) we have that x_a = x_a^i ∂_i, x_b = x_b^j ∂_j,
# and from applying the operator to f, [x_a,x_b]f we get that
# [x_a,x_b]f = (x_a^i (∂_i x_b^j) − x_b^i ∂_i x_a^j) ∂_j f
# So now we just apply this formula.
def commutator_lie(x_a,x_b):
    #var     = [I_c,a_c,I_1,a_1,I_2,a_2]
    dim = VARIABLES.shape[0]
    new_res = []
    if x_a == x_b:
        for i in range(0,dim):
            new_res.append(0)
        return new_res

```

```
129     for j in range(0,dim):
130         summ = 0
131         for i in range(0,dim):
132             summ = summ + ( x_a[i]*diff(x_b[j],VARIABLES[i]) -
133                             x_b[i]*diff(x_a[j],VARIABLES[i])   )
134         new_res.append(summ)
135 
136     return simplify(new_res)
137 
138 def vector_in_span(family,candidate):
139     nr_fam = len(family)
140     dim    = len(family[0])
141     M      = Matrix( dim, 1, family[0])
142     for e in range(1,nr_fam):
143         M = M.col_insert(e,Matrix(family[e]))
144     rank_M    = len( (M.rref())[1] )
145     augM      = M.col_insert(nr_fam,Matrix(candidate))
146     rank_augM = len( augM.rref()[1] )
147     if rank_M == rank_augM:
148         return True
149     else:
150         return False
151 
152 # I am assuming that 'family' is linearly independent already.
153 def is_controllable(family):
154     dim    = len(family[0])
155     nr_fam = len(family)
156     if nr_fam == 1:
157         print('It cannot be controllable if it only has one dimension' +
158               'unless the space is one dimensional. We need at least two' +
159               'in order to take the lie brackets, on which this method is' +
160               'based.')
161         return False
162     M = Matrix( dim, 1, family[0])
163     for e in range(1,nr_fam):
164         M = M.col_insert(e,Matrix(family[e]))
165     rank_M = len( (M.rref())[1] )
166     if rank_M != nr_fam:
167         print('Warning: family not linearly independent.')
168 
169     fam = family  # We will use this auxiliary vector to modify it.
170     nr_fam = len(fam)
171     #print('nr_fam = ',nr_fam)
172     # We do this so we won't have repeated pairs like [1,2] and [2,1] but just
173     # [1,2]. This is because the commutator is symmetric [1,2] = -[2,1] so
174     # it's just redundant information.
175     index_pairs = []
176     i_list_aux = [i for i in range(0,nr_fam)]
177     j_list_aux = [j for j in range(0,nr_fam)]
178     while i_list_aux:  # while list not empty
```

```
179         i = i_list_aux.pop()
180         for j in j_list_aux:
181                 index_pairs.append([i,j])
182         j_list_aux.pop()
183     #print(index_pairs)
184 
185     new_len_fam = len(fam)
186     while ( new_len_fam<dim and index_pairs ):  # While index_pairs not empty.
187         [i, j] = index_pairs.pop()
188         candidate = simplify(commutator_lie(fam[i],fam[j]))
189         print(candidate)
190         if not vector_in_span(fam,candidate):
191             fam.append(candidate)
192             new_len_fam = len(fam)
193             for e in range(0,new_len_fam):
194                 index_pairs.append([new_len_fam-1,e])
195         new_len_fam = len(fam)
196     #display(fam)
197     if (len(fam)==dim):
198         M = Matrix( dim, 1, fam[0])
199         for e in range(1,new_len_fam):
200             M = M.col_insert(e,Matrix(fam[e]))
201         rank_M = len( (M.rref())[1] )
202         display(M.rref())
203         if (len(fam) == rank_M):
204             return True
205         else:
206             print('Warning: good length but rank unequal.')
207             return False
208     else:
209         return False
210 
211 print('Is this system controllable?', is_controllable([X_0,X_1,X_2]))
```

Apéndice C

Código Python para hallar el control óptimo

En este código usa un optimizador para hallar las condiciones iniciales óptimas para hallar una solución próxima a la conseguida mediante multiplicadores de Lagrange. Ver el repositorio de GitHub del siguiente enlace para descargar cómodamente el código.

https://github.com/trahald/TFG

En el caso de no funcionar el enlace, contacte con el autor.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

### Code to solve the differential equation in order to find the optimum
### trajectory minimizing a cost functional defines as
### J = ∫ u̇²dt + 1/γ² G(x(T))

from __future__ import division # so that 1/2 does not give the loathed 0
from IPython.display import display, Math, Latex # beautiful output

from scipy.integrate import odeint
import scipy.integrate as inte
from scipy.optimize import minimize

import matplotlib.pyplot as plt
import matplotlib
from matplotlib import cm, colors
from mpl_toolkits.mplot3d import Axes3D
import pylab
from sympy import *

import matplotlib as mpl

import numpy as np
import cmath

#init_printing()  # For pretty printing.

```

```
29 ####### Control constants ##########################################################
30 INTEGRATION_METHOD  = 'bdf'
31 NSTEPS              = 5000000
32 INTEGRATOR          = 'vode'
33 T0, T1              = 0.0, 1.0
34 DT                  = (T1-T0)/1000.0
35
36 NR_ITERATIONS       = 3
37 ITERATION_DELTA     = 1.0
38 GAMMA_QUANT         = (1e-6)**(0.5) ## NEVER 0, we divide by gamma
39 GAMMA_CLASS         = (1e-3)**(0.5)    ## NEVER 0, we divide by gamma
40 COUPLING_CONSTANT   = 1.0
41
42 MASS     = 1.0
43 K_SPRING = 0.001
44
45 GRAD_NEWTON_DELTA = 1e-6
46
47 # BFGS parameters
48 BFGS_ITERATION = 100
49 BFGS_ALPHA = 0.001
50 BFGS_TOLERANCE = 2
51
52 ## Gradient descent parameters.
53 GRAD_TOLERANCE = 1
54 GRAD_STEP      = 0.1
55 GRAD_ITER      = 1
56 RANDOM_Y0_CONTRIBUTION = 1
57
58 # Initial conditions for our variables..
59 def setup_coord (q0_fin, p0_fin, phi1, theta1, Hx0, Hy0, Hz0, cx):
60     phi1, theta1   = (phi1)*np.pi/180, (theta1)*np.pi/180
61     rx0_fin = 1 * np.cos(phi1) * np.sin(theta1)
62     ry0_fin = 1 * np.sin(phi1) * np.sin(theta1)
63     rz0_fin = 1 * np.cos(theta1)
64     y_blanco1    = np.array([q0_fin, p0_fin,
65                          rx0_fin, ry0_fin, rz0_fin,
66                          Hx0, Hy0, Hz0,
67                          cx[0], cx[1], cx[2],
68                          cx[3], cx[4], cx[5],
69                          cx[6], cx[7]], dtype=np.float64)
70     return y_blanco1
71
72 y0_ini = setup_coord(   1, 1.5,  90,  90, 0, 0, 0, [0,0,0,0,0,0,0,0])
73
74
75 NR_COORD = y0_ini.shape[0]
76
77 def first_half(v):
78     dim = v.shape[0]
```

```
    return np.array([v[i] for i in range(0, int(dim/2))], dtype=np.float64)
def second_half(v):
    dim = v.shape[0]
    return np.array([v[int(dim/2)+i] for i in range(0, int(dim/2))],
                                                        dtype=np.float64)

y0_ini = setup_coord(1.1, 0.2,  -21, 124, 0.1, 0.2, 0.3,
                                  [3.2,1,-2,-3.3,1.,6,.6,0.3])
y0_ini_x   = first_half(y0_ini)
y0_ini_cx  = second_half(y0_ini)
GLOB_y0_ini_x = y0_ini_x


#random_addition  = np.array([ random.random()
#                      for j in range(0,8) ],dtype=np.float64 )
#y0_ini_cx  = y0_ini_cx + RANDOM_Y0_CONTRIBUTION * random_addition
#y0_ini_cx = np.array([-0.00097155, -0.01765558,  0.02165133, -0.01016424,
#                       0.00900474,  0.01375255,  0.02176548,  0.00433647])


# These serve to impose the 'final' conditions on c_x. Note these aren't the
# the boundary conditions, just the objective states which we will need in
# order co compute c_x (T).
y_blanco = [first_half(setup_coord(1.5,  0.5,  36., 110., 0., 0., 0.,
                                                [0.,0.,0.,0.,0.,0.,0.,0.])),
            first_half(setup_coord(   2.,   2., -26.,  33., 0., 0., 0.,
                                                [0.,0.,0.,0.,0.,0.,0.,0.]))]
#            first_half(setup_coord(   3.,   -1., -65., -295., 0., 0., 0.,
#                                                  [0.,0.,0.,0.,0.,0.,0.,0.]))]
'''
y_blanco = [first_half(setup_coord(1.5, 0.5,  36, 110, 0, 0, 0,
                                                [0,0,0,0,0,0,0,0]))]
'''
NR_BLANCOS = len(y_blanco)


####### Defining most of the variables #########################################
t = symbols('t', real=True);

q,p,rx,ry,rz,Hx,Hy,Hz         = symbols('q p r_x r_y r_z H_x H_y H_z');
cq,cp,crx,cry,crz,cHx,cHy,cHz = symbols(
                    'c_q c_p c_rx c_ry c_rz c_Hx c_Hy c_Hz');

LeviCivita3 = np.array([[[int((i - j) * (j - k) * (k - i) / 2)
                    for k in range(3)] for j in range(3)]
                    for i in range(3)], dtype=np.float64)

x_tot     = Matrix([q,p,rx,ry,rz,Hx,Hy,Hz,cq,cp,crx,cry,crz,cHx,cHy,cHz])
size_xtot = int(len(x_tot))
x         = Matrix([q,p,rx,ry,rz,Hx,Hy,Hz])
```

```
cx        = Matrix([cq,cp,crx,cry,crz,cHx,cHy,cHz])
r         = Matrix([rx,ry,rz])
H         = Matrix([Hx,Hy,Hz])
cr        = Matrix([crx,cry,crz])
cH        = Matrix([cHx,cHy,cHz])

m, k = symbols('m k', real=True);

epsilon = symbols('epsilon', real=True);

####### THE FUNCTION f(q)

#f = 1/sqrt(1+q**2)
#f=1+COUPLING_CONSTANT*(q**2 + p**2)
#f=1+COUPLING_CONSTANT*q*p
f=q
#f=1
#######

def sum1():
    sum1 = 0
    for i in range(0,3):
        for j in range(0,3):
            for k in range(0,3):
                sum1 = sum1 + LeviCivita3[i,j,k] * f * H[i]  * r[j] * cr[k]
    return sum1


# Here we use Pontryagin's hamiltonian from Pontryagins theorem.
h = ( cq*p/m + cp*(-k*q -2*(rx*Hx+ry*Hy+rz*Hz)*(diff(f,q))) + sum1() +
     1/2*(cHx*cHx + cHy*cHy + cHz*cHz) )
h = simplify(h)

# These are then the differential equations:
xtotdot = []
for i in range(0,int(len(x_tot)/2)):
    xtotdot.append( diff(h,cx[i]))
for i in range(0,int(len(x_tot)/2)):
    xtotdot.append(-diff(h,x[i]))

velocity = Matrix(xtotdot)
longitud = velocity.shape[0]
velocity = velocity.subs(m,MASS).subs(k,K_SPRING)

## I will calculate jac. scipy ode for documentation.
def jacobian(X):
    var = x_tot
    long = int(len(x_tot))
    Dfun=zeros(long,long)
    for i in range(0,long):
```

```
179         for j in range(0,long):
180             Dfun[i,j]=diff(X[i],var[j])
181     return Dfun
182 
183 xtot_list = [q,p,rx,ry,rz,Hx,Hy,Hz,cq,cp,crx,cry,crz,cHx,cHy,cHz]
184 
185 Dfun_sym = jacobian(velocity)
186 
187 # Now, evaluating jacobian at a number is slow, thus we will
188 # lambdify the function to make it quicker.
189 def jac_lambdi(Dfun_sym):
190     Dfun_l = [None]*size_xtot
191     for index in range(0,size_xtot):
192         Dfun_l[index] = [None]*size_xtot
193     for i in range(0,size_xtot):
194         for j in range(0,size_xtot):
195             Dfun_l[i][j] =  lambdify( [xtot_list,t],
196                                       simplify(Dfun_sym[i,j]) )
197     return Dfun_l
198 
199 Dfun_l = jac_lambdi(Dfun_sym)
200 
201 def gradient(t,var):
202     res = [None]*size_xtot
203     for index in range(0,size_xtot):
204         res[index] = [None]*size_xtot
205     for i in range(0,size_xtot):
206         for j in range(0,size_xtot):
207             res[i][j] = (Dfun_l[i][j])(var,t)
208     return res;
209 
210 fx = []
211 for i in range(0,size_xtot):
212     fx.append( lambdify( [xtot_list,t], velocity[i]) )
213 def func(t,var):
214     res = [None]*size_xtot  # Initialized with 6 components
215     for i in range(0,size_xtot):
216         res[i] = fx[i](var,t)
217     return res;
218 
219 def f(t, y, arg1):
220     #return [1j*arg1*y[0] + y[1], -arg1*y[1]**2]
221     return func(t,y)
222 def jac(t, y, arg1):
223     #return [[1j*arg1, 1], [0, -arg1*2*y[1]]]
224     return gradient(t,y)
225 ode = inte.ode(f, jac).set_integrator(INTEGRATOR, nsteps=NSTEPS, #order=15,
226                                       method=INTEGRATION_METHOD)
227 
228 y0 = np.array(y0_ini, dtype=np.float64)
```

```
t0, dt, t1 = T0, DT, T1

### This is our time evolution core.
def L(cy0):
    y0 = np.append(GLOB_y0_ini_x, cy0)  # y0_ini_cx is a global variable
    ode = inte.ode(f, jac).set_integrator(INTEGRATOR, nsteps=NSTEPS, order=15,
                                  method=INTEGRATION_METHOD)
    ode.set_initial_value(y0,T0).set_f_params(1.0).set_jac_params(1.0)
    while ode.successful() and ode.t < T1:
        ode.integrate(ode.t + DT)
    return ode.y

# This calculates the gradient of G, somewhat self explanatory.
def dGdx (y, y_b):
    return -1.0* np.array( [ (y[0]-y_b[0])/(GAMMA_CLASS**2),
                             (y[1]-y_b[1])/(GAMMA_CLASS**2),
                             0,#(y[2]-y_b[2])/(GAMMA_QUANT**2),
                             0,#(y[3]-y_b[3])/(GAMMA_QUANT**2),
                             0,#(y[4]-y_b[4])/(GAMMA_QUANT**2),
                              0, 0, 0 ],  dtype=np.float64 )

# I use the following two functions later on to print debugging
# messages.
def coste_qp (y,y_b):
    return ( 0.5 * (y[0]-y_b[0])**2 /(GAMMA_CLASS**2) +
             0.5 * (y[1]-y_b[1])**2 /(GAMMA_CLASS**2) )

def coste_quant (y,y_b):
    return ( 0.5 * (y[2]-y_b[2])**2 /(GAMMA_QUANT**2) +
             0.5 * (y[3]-y_b[3])**2 /(GAMMA_QUANT**2) +
             0.5 * (y[4]-y_b[4])**2 /(GAMMA_QUANT**2))


def Newton_vect(cx_ini, y_b):
    y_iter = L(cx_ini)
    length = y_iter.shape[0]
    loop_max = int(length/2)

    cx_objective = dGdx(y_iter, y_b)

    cx_actual = []
    for i in range(0, loop_max):
        cx_actual.append(y_iter[loop_max+i])
    cx_actual = np.array(cx_actual, dtype=np.float64)

    return [cx_actual, cx_objective]

####################
# The following is the CORE function which measures how far we are from
# the optimal solution. We use a log to add numerical stability,
```

```
# it does not affect the position of extremum points.
# From here on we write some functions for the gradient of Newton_zero
# and other things. Not all functions will be used.
####################
def Newton_zero(y0_cx):
    var = Newton_vect(y0_cx, GLOB_y_blanco)
    return np.log((np.dot(var[0]-var[1],var[0]-var[1])))

cond_ini_cx = [1,2,3,4,5,6,7,5]
#basis:
identi = eye(len(cond_ini_cx))
basis_c = []
for e in range(0,len(cond_ini_cx)):
    basis_c.append(np.array(list(identi[e,:]), dtype=np.float64))


def partial_Newton(func, punto, basis_vector, epsi):
    return ( (func(punto + epsi*basis_vector) -
              func(punto - epsi*basis_vector) ) / (2*epsi) )


def grad_Newton(punto):
    #  We define basis vectores along which we will take the
    #  partial derivative.
    cond_ini_cx = [1,2,3,4,5,6,7,5]
    longi = len(cond_ini_cx)
    identi = eye(longi)
    basis_c = []
    for e in range(0,longi):
        basis_c.append(np.array(list(identi[e,:]), dtype=np.float64))

    #  Now we calculate all the different partial derivatives.
    punto = np.array(punto, dtype=np.float64)
    res = []  # We will store the result in 'res' and we need it to be a
                # an array type.
    for e in range(0,punto.shape[0]):
        res.append(partial_Newton(Newton_zero, punto, basis_c[e],
                                                        GRAD_NEWTON_DELTA))
        #res.append( (Newton_zero(punto + GRAD_NEWTON_DELTA*basis_c[e]) -
        #              Newton_zero(punto - GRAD_NEWTON_DELTA*basis_c[e]) ) /
        #                                  (2*GRAD_NEWTON_DELTA) )
    res = np.array(res, dtype=np.float64)
    return res

def hess_Newton(x,p):
    #  We define basis vectores along which we will take the
    #  partial derivative.
    cond_ini_cx = [1,2,3,4,5,6,7,5]
    longi = len(cond_ini_cx)
    identi = eye(longi)
```

```
329     basis_c = []
330     for e in range(0,longi):
331         basis_c.append(np.array(list(identi[e,:]), dtype=np.float64))
332 
333     #  Now we proceed to calculate the Hessian.
334     x = (np.matrix(x)).T
335     p = (np.matrix(p)).T
336     dim = x.shape[0]
337     B = np.matrix(np.eye(dim))
338     res = []
339     x = sanitize_x(x)
340     for i in range(0,dim):
341         aux =( ( grad_Newton(y0_ini_cx+GRAD_NEWTON_DELTA*basis_c[i])-
342                  grad_Newton(y0_ini_cx-GRAD_NEWTON_DELTA*basis_c[i])   )
343                   /2/GRAD_NEWTON_DELTA )
344         B[:,i] = np.matrix(aux).T
345     res = B*p
346     #return res ## This returns H*p, a vector.
347     return res
348 
349 ### NOTE: As is, it does not work very well.
350 def gradient_descent(y0_cx, tolerance, step_size, max_steps):
351     y0 = np.array(y0_cx, dtype=np.float64)
352     counter = 0
353     dist = 1e88
354     while dist>tolerance and counter<max_steps:
355         dist = Newton_zero(y0)
356         vect = grad_Newton(y0)
357         vect = vect/((np.inner(vect,vect))**0.5)
358         y0 = y0 - step_size*vect
359         counter = counter + 1
360         print('loop: ', counter, ' dist = ', dist)
361     return y0
362 
363 # Some functions are picky as to whether the input is an array or list.
364 def sanitize_x (x):
365     dim = x.shape[0]
366     res = [None]*dim
367     res = np.array(res, dtype=np.float64)
368     for i in range(0,dim):
369         res[i] = x[i,0]
370     return res
371 
372 # This is a manual implementation of the BFGS optimization algorithm,
373 # but SciPy's implementation is better, and even so it does not converge.
374 def BFGS(y0_cx):
375     # We are assuming y0_cx is an array.
376     dim = y0_cx.shape[0]
377     x = (np.matrix(y0_cx)).T  # now a vector
378     B = np.matrix(np.eye(dim))
```

```
379     inv_B = np.matrix(np.eye(dim))
380     alpha = BFGS_ALPHA
381     control_par = Newton_zero(sanitize_x(x))
382     for i in range(0, BFGS_ITERATION):
383         if (control_par > BFGS_TOLERANCE):
384             print('BFGS iter = ', i)
385             grad = np.matrix(grad_Newton(sanitize_x(x))).T
386             p = -inv_B*grad
387             s = alpha*p
388             x = x + s
389             grad_new = np.matrix(grad_Newton(sanitize_x(x))).T
390             y = grad_new - grad
391             yyT = y*y.T
392             yTs = (y.T*s)[0,0]
393             ssT = s*s.T
394             sTy = (s.T*y)[0,0]
395             syT = s*y.T
396             ysT = y*s.T
397             Bs = B*s
398             B = B + ( (yyT)/(yTs) - ( (Bs*(s.T*B)) /((s.T*(Bs))) ) )
399             inv_B = (inv_B + (ssT*(sTy + (y.T*(inv_B*y))[0,0]) / (sTy**2)) +
400                                     + ( inv_B*ysT + syT*inv_B)/sTy )
401             control_par = Newton_zero(sanitize_x(x))
402             print('|df/ds| = ', np.sqrt(np.linalg.norm(grad_new)))
403             print('dist = ',control_par)
404         else:
405             return sanitize_x(x)
406 
407     return sanitize_x(x)
408 
409 ### We will divide the [T0,T1] interval into various parts.
410 NR_POINTS, BIG_T0, BIG_T1 = len(y_blanco), 0.0, 1.0
411 #NR_POINTS = 1
412 T_INTERVAL_ITER = (BIG_T1-BIG_T0)/NR_POINTS
413 
414 ### We reasign values to the global variables just in case they changed
415 ### values along the way.
416 y0_ini_x   = first_half(y0_ini)
417 y0_ini_cx  = second_half(y0_ini)
418 GLOB_y0_ini_x  = y0_ini_x
419 GLOB_y0_ini_cx = y0_ini_cx
420 
421 ### Plot lists.
422 ts  = [[] for i in range(0,NR_POINTS)]  # Here we will store the time.
423 res = [[] for i in range(0,NR_POINTS)]  # Here we will store the variables.
424 print('GLOB_y0_ini_x= ', GLOB_y0_ini_x)
425 COST_QUANT = 0
426 COST_CLASS = 0
427 endpoints = []  # Here we will store the end points from each
428                 # interval [t_{j-1},t_j].
```

```
LIST_cx_ini_iterations = []
######################################################################
# THIS IS THE CORE LOOP WHERE WE OPTIMIZE EACH INTERVAL SEPARATELY
######################################################################
for iteration in range(0,NR_POINTS):
    print('Beginning loop')
    T0 = T_INTERVAL_ITER * iteration
    T1 = T_INTERVAL_ITER * (iteration + 1)
    DT = (T1-T0)/1000.0
    GLOB_y_blanco = y_blanco[iteration]
    #GLOB_y_blanco = y_blanco[1]
    print('GLOB_y_blanco = ', GLOB_y_blanco)
    print('Iteration nr. '+ str(iteration)+'. '
                  'Started optimization. Unknown wait time.')

    GLOB_y0_ini_cx = np.array([0,0,0,0,0,0,0,0], dtype=np.float64)
    OUTPUT = minimize(Newton_zero, GLOB_y0_ini_cx,
                   method='Nelder-Mead', # jac=grad_Newton, hess=hess_Newton,
                    options={'xtol': 1e-6, 'ftol':1e-6, #'adaptive': True,
                             'maxiter': 3000, 'disp': True})


    print('Final result. cx_0 = ', OUTPUT.x)
    y0_ini_cx_final = OUTPUT.x
    LIST_cx_ini_iterations.append(y0_ini_cx_final)  # All vectors saved in
                                                    # this list will be
                                                    # printed.

    print('Beginning to set initial values.')
    ode.set_initial_value(np.append(GLOB_y0_ini_x, y0_ini_cx_final),
                          T0).set_f_params(1.0).set_jac_params(1.0)
    print('Beginning to integrate.')
    while ode.successful() and ode.t < T1:
        ode.integrate(ode.t + DT)
        ts[iteration].append(ode.t)
        res[iteration].append(ode.y)
    print('Finished integration.')
    COST_CLASS = COST_CLASS + coste_qp(first_half(res[iteration][-1]),
                                       GLOB_y_blanco)
    COST_QUANT = COST_QUANT + coste_quant(first_half(res[iteration][-1]),
                                          GLOB_y_blanco)
    print(iteration, 'AUX cost class = ', COST_CLASS)
    print(iteration, 'AUX cost quant = ', COST_QUANT)
    GLOB_y0_ini_x = first_half(res[iteration][-1])
    endpoints.append(GLOB_y0_ini_x)

    print('GLOB_y0_ini_x= ', GLOB_y0_ini_x)
    print('Finished loop')


```

```
479 ##############################################
480 ### Here we write to file the calculates costates.
481
482 ####################
483 CONTROL_RUN = 20   # This is useful to label different system runs.
484 ####################
485
486 nombre_inicial = 'RUN='
487 file = open(nombre_inicial+str(CONTROL_RUN)+'cx_ini'+'.txt','w')
488 for el in range(0,len(LIST_cx_ini_iterations)):
489     string = ''
490     for e in range(0,len(LIST_cx_ini_iterations[el])):
491         string = string + str(LIST_cx_ini_iterations[el][e]) + ' '
492     file.write(string+'\n')
493 file.close()
494
495 nombre_inicial = 'RUN='
496 file = open(nombre_inicial+str(CONTROL_RUN)+'.yblanco'+'.txt','w')
497 for el in range(0,len(y_blanco)):
498     string = ''
499     for e in range(0,len(y_blanco[el])):
500         string = string + str(y_blanco[el][e]) + ' '
501     file.write(string+'\n')
502 file.close()
503
504 ########################################################################
505 ########################################################################
506 ### From here on, we are mainly concerned with plotting. The core
507 ### optimization and system evolution is done.
508 ########################################################################
509
510 aux_ts = []
511 for iteration in range(0,NR_POINTS):
512     for el in range(0,len(res[0])):
513         aux_ts.append(iteration+ts[iteration][el])
514 ts = aux_ts
515
516
517 plot_normarho = []
518 plot_usquared  = []
519 plotobject = [[] for e in range(0,16)]    # Beware! Change 16 to your
520                                           # number of variables.
521
522 ### Now we fuse all the differnet iterations so that we can plot all the
523 ### intervals together.
524 for nr_iter in range(0,NR_POINTS):
525     for el in range(0,len(res[0])):
526         for coord in range(0,16):
527             plotobject[coord].append(res[nr_iter][el][coord])
528         plot_normarho.append(  (res[nr_iter][el][2]**2 +
```

```
                                res[nr_iter][el][3]**2 +
                                res[nr_iter][el][4]**2)**(0.5) )
        plot_usquared.append( 0.5* (res[nr_iter][el][13]**2 +
                                    res[nr_iter][el][14]**2 +
                                    res[nr_iter][el][15]**2)   )

COSTE_ENERGETICO = 0
for i in range(0,len(ts)):
    COSTE_ENERGETICO = COSTE_ENERGETICO + plot_usquared[i]*DT
print('COSTE \int 0.5u^2 = ', COSTE_ENERGETICO)
print('COSTE QUANT = ', COST_QUANT)
print('COSTE CLASS = ', COST_CLASS)


#### PLOT

# Sources: http://www.randalolson.com/2014/06/28/how-to-make
#            -beautiful-data-visualizations-in-python-with-matplotlib/
# These are the "Tableau 20" colors as RGB.
tableau20=[(31, 119, 180),  (174, 199, 232), (255, 127, 14),  (255, 187, 120),
           (44, 160, 44),   (152, 223, 138), (214, 39, 40),   (255, 152, 150),
           (148, 103, 189), (197, 176, 213), (140, 86, 75),   (196, 156, 148),
           (227, 119, 194), (247, 182, 210), (127, 127, 127), (199, 199, 199),
           (188, 189, 34),  (219, 219, 141), (23, 190, 207),  (158, 218, 229)]
# Scale the RGB values to the [0, 1] range, which is the format matplotlib
# accepts.
for i in range(len(tableau20)):
    r, g, b = tableau20[i]
    tableau20[i] = (r / 255., g / 255., b / 255.)

def plot_qp ():
    plt.figure(figsize=(10, 8))

    plt.style.use('ggplot')
    plt.style.use('seaborn-talk')
    plt.xlabel(r'Posición ($q$)')
    plt.ylabel(r'Momento ($p$)')
    plt.plot(plotobject[0], plotobject[1],
             color=tableau20[0], label=r'$q$ y $p$')
    for i in range(0,len(y_blanco)):
        plt.plot(y_blanco[i][0],y_blanco[i][1],
             marker = 'D', label= ('Objetivo '+str(i)), color=tableau20[0])

    for i in range(0,len(endpoints)):
        plt.plot(endpoints[i][0],endpoints[i][1], marker= 'o',
                 color = tableau20[0])

    plt.plot(y0_ini[0],y0_ini[1], marker= 'o',
             label=  'Pto. de la trayectoria', color=tableau20[0])
    plt.title(r'Evolución de un sistema híbrido controlado')
```

```
579     plt.legend(loc='upper left')
580     plt.show()
581     #pylab.savefig('pq.png', bbox_inches='tight')
582 
583 def plot_purity():
584     plt.figure(figsize=(4, 4))
585     plt.style.use('ggplot')
586     plt.style.use('seaborn-talk')
587     plt.xlabel(r'Tiempo ($t$)')
588     plt.ylim(0,1.2)
589     plt.plot(ts, plot_normarho,  color=tableau20[2],
590             label = r'Pureza $\sim\ \sqrt{\operatorname{tr}\ \rho^2}$)')
591     plt.title(r'Evolución de la pureza')
592     plt.legend(loc='upper left')
593     plt.show()
594 
595 def plot_usquare():
596     plt.figure(figsize=(8, 8))
597     plt.style.use('ggplot')
598     plt.style.use('seaborn-talk')
599     plt.xlabel(r'Tiempo ($t$)')
600     #plt.ylim(0,1.2)
601     plt.plot(ts, plot_usquared,  color=tableau20[2],
602             label = r'$\frac{1}{2}u^2$')
603     plt.title(r'Coste energético')
604     plt.legend(loc='upper left')
605     plt.show()
606 
607 #### Hamiltonian plot
608 ##### Bloch sphere plot.
609 #Taken from: https://matplotlib.org/gallery/mplot3d/lines3d.html
610 def plot_Hamilt():
611     mpl.rcParams['legend.fontsize'] = 10
612 
613     fig = plt.figure(figsize=(9, 9))
614     plt.style.use('ggplot')
615     plt.style.use('seaborn-talk')
616 
617     #ax = fig.gca(projection='3d')
618     ax = fig.add_subplot(111, projection='3d')
619 
620     #x = np.array([ res[el][5] for el in range(0,len(ts))], dtype=np.float64)
621     #y = np.array([ res[el][6] for el in range(0,len(ts))], dtype=np.float64)
622     #z = np.array([ res[el][7] for el in range(0,len(ts))], dtype=np.float64)
623     x = np.array(plotobject[5])
624     y = np.array(plotobject[6])
625     z = np.array(plotobject[7])
626     # Make data
627     u = np.linspace(0, 2 * np.pi, 100)
628     v = np.linspace(0, np.pi, 100)
```

```
    x1 = 1 * np.outer(np.cos(u), np.sin(v))
    y1 = 1 * np.outer(np.sin(u), np.sin(v))
    z1 = 1 * np.outer(np.ones(np.size(u)), np.cos(v))

    ax.plot_wireframe(x1, y1, z1, color=tableau20[0], rstride=4, cstride=4,
                      alpha=0.25, label = 'Esfera de Bloch')

    ax.plot(x, y, z, color=tableau20[2],
            label = r'Hamiltoniano $\mathcal{H}$ en el tiempo')
    #ax.scatter(rx0,ry0,rz0, label = 'Puntos fijos')
    ax.scatter(y0_ini[5],y0_ini[6],y0_ini[7], marker = 'D',
                                label = 'Punto de inicio')

    ax.legend()

    plt.show()
    #pylab.savefig('bloch.png', bbox_inches='tight')

def plot_rho ():
    ##### Bloch sphere plot.
    #Taken from: https://matplotlib.org/gallery/mplot3d/lines3d.html

    mpl.rcParams['legend.fontsize'] = 10

    fig = plt.figure(figsize=(9, 9))
    plt.style.use('ggplot')
    plt.style.use('seaborn-talk')


    #ax = fig.gca(projection='3d')
    ax = fig.add_subplot(111, projection='3d')

    #2,3,4
    #x = np.array([ res[el][2] for el in range(0,len(ts))], dtype=np.float64)
    #y = np.array([ res[el][3] for el in range(0,len(ts))], dtype=np.float64)
    #z = np.array([ res[el][4] for el in range(0,len(ts))], dtype=np.float64)
    x = np.array(plotobject[2])
    y = np.array(plotobject[3])
    z = np.array(plotobject[4])
    # Make data
    u = np.linspace(0, 2 * np.pi, 100)
    v = np.linspace(0, np.pi, 100)
    x1 = 1 * np.outer(np.cos(u), np.sin(v))
    y1 = 1 * np.outer(np.sin(u), np.sin(v))
    z1 = 1 * np.outer(np.ones(np.size(u)), np.cos(v))

    ax.plot_wireframe(x1, y1, z1, color=tableau20[0], rstride=4, cstride=4,
                      alpha=0.25, label = 'Esfera de radio 1')

    ax.plot(x, y, z, color=tableau20[2], label = r'Trayectoria de $\rho$')
```

```
679     ax.set_xlabel('X')
680     ax.set_ylabel('Y')
681     ax.set_zlabel('Z')
682     ax.set_xticklabels([])
683     ax.set_yticklabels([])
684     ax.set_zticklabels([])
685 
686     ax.text(0, 0, +1.3, r'$\vert 1 \rangle$', size = 'xx-large')
687     ax.text(0, 0, -1.3, r'$\vert 0 \rangle$', size = 'xx-large')
688 
689     for i in range(0,len(y_blanco)):
690         ax.scatter(y_blanco[i][2], y_blanco[i][3], y_blanco[i][4],
691                    label = 'Objetivo '+str(i), marker = 'D')
692 
693     for i in range(0,len(endpoints)):
694         ax.scatter(endpoints[i][2], endpoints[i][3], endpoints[i][4],
695                    c=tableau20[2], marker = 'o')
696 
697     ax.scatter(y0_ini[2],y0_ini[3],y0_ini[4], marker = 'o', c=tableau20[2],
698                                label = 'Pto. trayectoria')
699 
700     ax.legend()
701     plt.show()
702     #pylab.savefig('bloch.png', bbox_inches='tight')
703 
704 def plot_costates():
705     labels = [r'$c_{q}$', r'$c_{p}$', r'$c_{\rho x}$', r'$c_{\rho y}$',
706               r'$c_{\rho z}$',  r'$c_{\mathcal{H} x}$',
707                r'$c_{\mathcal{H} y}$', r'$c_{\mathcal{H} z}$']
708     plt.figure(figsize=(8, 8))
709     plt.xlim(0,1.3)
710     plt.style.use('ggplot')
711     plt.style.use('seaborn-talk')
712     plt.xlabel(r'Tiempo ($t$)')
713     #plt.ylabel(r'Coestados')
714     #plt.ylim(0,1.2)
715     for el in range(0,8):
716         plt.plot(ts, plotobject[8+el],  color=tableau20[el],
717                  label = labels[el])
718     plt.title(r'Los coestados')
719     plt.legend(loc='center right')
720     plt.show()
721 
722 #plot_costates()
723 #plot_usquare()
724 plot_rho()
725 plot_qp()
726 #plot_Hamilt()
727 #plot_purity()
```