Universidade Federal do Rio de Janeiro

Departamento de Matemática Aplicada
Instituto de Matemática

Metódos Matemáticos em Mecânica Geométrica

Gil Sales Miranda Neto

Orientador Prof. Dr. Alejandro Cabrera
Instituto de Matemática
Departamento de Matemática Aplicada
Universidade Federal do Rio de Janeiro

31 de Julho de 2020

Contents

Introdução 1

O estudo de corpos deformáveis em rotação é de grande interesse para a Física, e talvez um bom exemplo seja o de um gato em queda.
A modelagem deste fenômeno pode trazer resultados interessantes para estudos como o de um atleta olímpico de saltos ornamentais.
Em busca de uma boa modelagem computacional para estes fenômenos se dá este trabalho, a partir do algoritmo implementado por um ex-aluno do projeto implementamos um novo algoritmo na forma de uma biblioteca Python.

1.1 Definições

Seja $Q = (\mathbb{R}^3)^N$, o espaço de configuração de todos os pontos que compõe o corpo deformável com origem no centro de massa. Seja $q = (r_1, \ldots, r_N) \in Q$ os elementos de Q.
Podemos definir o operador momento de inércia I como:

$$I(q)\omega = \sum_i^N m_i r_i \times (\omega \times r_i)$$

O operador momento angular L como:

$$L(q, v) = \sum_i^N m_i r_i \times v_i$$

1.2 Dados de Input

- $\vec{r}_{i0}(t) \in \mathbb{R}^3$, $i = 1, \ldots, N$, funções que representam as posições desde um sistema rotacional não-inercial
- $m_1, \ldots, m_N,$, massas do sistema
- $\vec{J}_0$, vetor do momento angular inicial

1.3 Dados de Output

$$\vec{r}_i(t) = R(t)\vec{r}_{i0}(t),\ i = 1, \ldots, N$$

Onde $\vec{r}_i(t)$ representa a posição a partir de um sistema inercial, com origem em $CM(t)$, centro de massa.
$R(t) \in SO(3)$ é a matriz de rotação que leva de um sistema não-inercial a um sistema inercial, é também a incógnita do nosso problema.

1.4 Sistema de Equações Diferenciais

Obtemos $R(t)$ a partir de um sistema de equações diferenciais que vem da conservação do momento angular, a qual equivale a uma equação de 2ª ordem para $R(t)$, quando visto de um referencial inercial. Introduzimos então uma nova variável $\vec{\pi}$ para termos um sistema de equações em primeira ordem

Temos as equações de movimentos para rotação de corpos deformáveis, onde podemos resolver a equação diferencial para uma configuração de um corpo, e recuperar as rotações do corpo em estudo.

Incógnitas do sistema de 1ª ordem induzido

- $R(t) \in SO(3)$, matriz de rotação
- $\vec{\pi}(t) \in \mathbb{R}^3$

O sistema

$$\begin{cases} R^{-1}(t)\dot{R}(t) & = \psi^{-1}(I_0^{-1}(t)(\vec{\pi} - \vec{L}_0(t)) \\ \dot{\vec{\pi}} & = \vec{\pi} \times (I_0^{-1}(t)(\vec{\pi} - \vec{L}_0(t)) \end{cases}$$

Condições iniciais da equação diferencial: $\begin{cases} R(t_0) = \mathbb{1} \\ \vec{\pi}(t_0) = \vec{J}_0 \end{cases}$

Onde ψ é um isomorfismo linear do conjunto de matrizes 3×3 antissimétricas no conjunto R^3 de vetores
$I_0(t)$ é a função tensor de inércia avaliada nas condições inicias $r_{i0}(t)$
$L_0(t)$ é a função momento angular avaliada nas condições inicias $r_{i0}(t)$

2 Quatérnios e Representações de Rotações

2.1 Os Quatérnios

Quatérnios são elementos do espaço vetorial H sobre o corpo dos reais, onde H é o espaço Euclidiano $\mathbb{R}^4$. Uma das importantes propriedades dos quatérnios é que a multiplicação neste espaço é associativa e distributiva sobre a adição, mas é não comutativa, então H é uma álgebra associativa não-comutativa sobre os reais.
Com quatérnios unitários podemos representar rotações de vetores do $\mathbb{R}^3$, que é a motivação por trás do uso dessa técnica, já que computacionalmente representar rotações por quatérnios não trazem um problema chamado "gimbal lock", que ocorre quando utilizamos matrizes de rotações e ângulos de Euler.

Denotamos por $\{\mathbf{1}, \mathbf{i}, \mathbf{j}, \mathbf{k}\}$ a base canônica de H: $\mathbf{1} = (1,0,0,0), \mathbf{i} = (0,1,0,0), \mathbf{j} = (0,0,1,0), \mathbf{k} = (0,0,0,1)$.
Um ponto $\mathbf{u} = (w,x,y,z) \in H$ pode ser escrito como $\mathbf{u} = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$.
A w damos o nome de parte real do quatérnio, e ao vetor $x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$ de parte vetorial de $\mathbf{u}$.
Se $v = (x,y,z) \in \mathbb{R}^3$, indicaremos por $\hat{\mathbf{v}}$ o quatérnio puro associado: $\hat{\mathbf{v}} = x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$
Chamamos de quatérnio puro um ponto $\mathbf{u} \in H$ da forma $\mathbf{u} = (0,x,y,z)$.
Com isto definido, podemos escrever qualquer quatérnio $\mathbf{u}$ na forma $\mathbf{u} = w + \hat{\mathbf{v}}$, $w \in \mathbb{R}$, $v \in \mathbb{R}^3$.
O conjugado de um quatérnio $\mathbf{u} = w + \hat{\mathbf{v}}$ é definido como $\mathbf{u}^* = w - \hat{\mathbf{v}}$

O produto dos elementos da base canônica é definido como na tabela

$\times$	**1**	**i**	**j**	**k**
1	**1**	**i**	**j**	**k**
i	**i**	**-1**	**k**	**-j**
j	**j**	**-k**	**-1**	**i**
k	**k**	**j**	**-i**	**-1**

Dado dois elementos do espaço H: $\mathbf{u}_1, \mathbf{u}_2 \in H$. Com $\mathbf{u}_1 = a_1 + \hat{\mathbf{v}}_1$ e $\mathbf{u}_2 = a_2 + \hat{\mathbf{v}}_2$, o produto entre dois elementos do espaço vetorial dos quatérnio é definido por:

$$\mathbf{u}_1\mathbf{u}_2 = a_1a_2 - \langle \hat{\mathbf{v}}_1, \hat{\mathbf{v}}_2 \rangle + a_1\hat{\mathbf{v}}_1 + a_2\hat{\mathbf{v}}_2 + \hat{\mathbf{v}}_1 \times \hat{\mathbf{v}}_2$$

Onde $\langle \cdot, \cdot \rangle$ é o produto escalar, e $\times$ é o produto vetorial
A norma de um quatérnio $\mathbf{u}$ é igual à norma de um vetor de $\mathbb{R}^4$, e é dada por:

$$|\mathbf{u}| = \sqrt{\langle \mathbf{u}, \mathbf{u} \rangle} = \sqrt{w^2 + x^2 + y^2 + z^2}$$

Se $|\mathbf{u}| = 1$, temos um quatérnio unitário. O produto de dois quatérnios unitários é um quatérnio unitário.
O inverso multiplicativo de um quatérnio $\mathbf{u}$ é $\mathbf{u}^{-1}$ tal que $\mathbf{pp}^{-1} = 1$ e é dado por:

$$\mathbf{u}^{-1} = \frac{\mathbf{u}^*}{|\mathbf{u}|^2}$$

Temos então que, se $|\mathbf{u}| = 1$, então $\mathbf{u}^{-1} = \mathbf{u}^*$

2.2 Rotação com Quatérnios

Dado um vetor $v = (x, y, z) \in \mathbb{R}^3$, podemos pensar v como um quatérnio puro, então $\mathbf{v} = (0, x, y, z)$, dado um quatérnio unitário $\mathbf{p}$, podemos representar a rotação de v em torno do eixo definido pela parte vetorial de $\mathbf{p}$ como:

$$v' = \mathbf{pvp}^{-1} = \mathbf{pvp}^*$$

Onde v' ainda é um elemento de $\mathbb{R}^3$, tomado como a parte vetorial de $\mathbf{pvp}^*$
Vejamos alguns quatérnios unitários para representar rotações em vetores do $\mathbb{R}^3$
Abaixo todas as rotações são feitas por um ângulo α, onde $\mathbf{p}_x$ é em torno do eixo x, assim como $\mathbf{p}_y, \mathbf{p}_z$ são em torno dos eixos y e z, respectivamente.

$$\mathbf{p}_x = \begin{bmatrix} \cos(\frac{\alpha}{2}) \\ 0 \\ \sin(\frac{\alpha}{2}) \\ 0 \end{bmatrix}, \mathbf{p}_y = \begin{bmatrix} \cos(\frac{\alpha}{2}) \\ \sin(\frac{\alpha}{2}) \\ 0 \\ 0 \end{bmatrix}, \mathbf{p}_z = \begin{bmatrix} \cos(\frac{\alpha}{2}) \\ 0 \\ 0 \\ \sin(\frac{\alpha}{2}) \end{bmatrix}$$

E para uma rotação em torno de um eixo descrito por $u = (u_x, u_y, u_z) \in \mathbb{R}^3$ por um ângulo α:

$$\mathbf{p} = \begin{bmatrix} \cos(\frac{\alpha}{2}) \\ \sin(\frac{\alpha}{2})u_x \\ \sin(\frac{\alpha}{2})u_y \\ \sin(\frac{\alpha}{2})u_z \end{bmatrix}$$

Todas as operações com Quatérnios são tratadas utilizando a biblioteca PyQuaternion, cuja documentação pode ser acessada em: https://kieranwynn.github.io/pyquaternion/

3 O algoritmo

3.1 Dependência de pacotes

O algoritmo é compatível com a linguagem Python na versão 3.x ou superior, e como dependência utiliza os seguintes pacotes:

- NumPy: `https://github.com/numpy/numpy` - para tratamentos de cálculos matemáticos e utilização de vetores e matrizes
- PyQuaternion: `https://github.com/KieranWynn/pyquaternion` - para lidar com operações de quaternions, a biblioteca é escrita em C++ e portada para Python para melhor velocidade
- SciPy: `https://github.com/scipy/scipy`
- MatPlotLib: `https://github.com/matplotlib/matplotlib` - para lidar com plotagem dos gráficos 2D/3D e animações

O repositório de códigos é: `https://github.com/mirandagil/geometric-mechanics`, para efetuar a instalação basta digitar no terminal

```
git clone https://github.com/mirandagil/geometric-mechanics
```

Ou acessar o repositório e baixar manualmente os arquivos. Para instalar todas as dependências listadas acima basta executar o comando

```
pip install -r requirements.txt
```

Na pasta onde baixou os códigos do projeto, utilizando pip ou o gerenciador de pacotes Python de sua preferência.

3.2 Estrutura do código

O código está estruturado em orientação a objetos, contendo três classes principais:

- deformableBody

- Physics
- Graphics

A classe `deformableBody` trata de instanciar os objetos de corpo rígido ou corpo deformável a serem usados nas modelagens.
A classe `Physics` trata de todas os cálculos referentes à física da modelagem. A classe `Graphics` trata de toda a parte gráfica, para plotagem de gráficos e criação de animações.

3.2.1 deformableBody

Para instanciar um objeto formado por n pontos de massa é necessário fornecer alguns parâmetros na sua chamada, sendo estes:

- `r_0`: Matriz $(nx3)$ de posições no espaço para cada ponto de massa
- `masses`: Vetor contendo a massa de cada ponto do corpo.
- `p_0`: Vetor contendo o momento de inércia do corpo
- `rot` é um vetor de tamanho n onde cada elemento deste vetor contém uma função do objeto `PyQuaternion` aplicada ao ponto de massa
- `num_times` é o tamanho da discretização do tempo
- `body_lines` é um vetor onde cada elemento é uma tupla, em cada tupla representa entre quais pontos de massa deve ser desenhado uma reta, este elemento é usado apenas com a finalidade de produzir gráficos e não interefere no algoritmo.

Exemplo de uso para instanciar um objeto

```
cube = deformableBody(r_0, masses, p_0, r,ot num_times, body_lines)
```

Listing 3.1: Instanciando o objeto cube

A classe `deformableBody` traz os seguintes metódos

- `translation_CM`: Ao ser chamada executa uma translação do centro de massa do objeto para o ponto $(0, 0, 0)$
- `set_positions`: Ao ser chamada salva as posições de toda a simulação do objeto em `self.positions`

3.2.2 Physics

Os metidos da classe `Physics` são do tipo estático, isto é, não dependem de um objeto instanciado para serem chamados, então não é necessário criar um objeto do tipo `Physics` ao longo da execução do código.
Esta classe traz os seguintes métodos

- `get_InertialTensor`: Calcula o Tensor de Inércia de um corpo, dados matriz de posições dos pontos de massa e um vetor contendo as massas de cada ponto
- `get_InternalAngularMomentum`: Calcula o momento angular interno de um corpo
- `CM` Calcula o centro de massa de um corpo
- `particles` Aplica uma rotação a cada ponto de massa
- `eqOfMotion` Equação Diferencial do Sistema
- `solve_eq` Chama a solução da Equação Diferencial durante o tempo determinado

As fórmulas utilizadas nos métodos da classe `Physics` estão explicitadas no apêndice.

3.2.3 Graphics

Os metódos da classe `Graphics` são do tipo estático, isto é, não dependem de um objeto instanciado para serem chamados, então não é necessário criar um objeto do tipo `Graphics` ao longo da execução do código.
Esta classe traz os seguintes metódos

- `update_plot`: Faz um novo plot das posições a cada frame da animação
- `create_plot`: Cria o plot do gráfico do Corpo e sua animação ao longo do tempo

3.3 Utilizando o algoritmo: Cubo com Hélice

Para um teste do algoritmo iremos olhar para a modelagem do Cubo com Hélice, definindo as posições iniciais e a massa de cada ponto.

```
masses = np.array([1,1,1,1,1,1,1,1,.5,.5])
n_mass = len(masses)
r_0 = [None]*n_mass

r_0[0] = [1, 1, 0]
r_0[1] = [1, -1, 0]
r_0[2] = [-1, 1, 0]
r_0[3] = [-1, -1, 0]
r_0[4] = [1, 1, -1]
r_0[5] = [1,-1,-1]
r_0[6] = [-1,1,-1]
r_0[7] = [-1, -1, -1]
r_0[8] = [0, -.5, .5]
r_0[9] = [0, .5, .5]
r_0 = np.array(r_0)
```

Listing 3.2: Posição inicial e massa do CubeCopter

Definimos também como se dá o desenho entre cada ponto de massa

```
body_lines = [
    ## top
    (0, 1),
    (0, 2),
    (1, 3),
    (3, 2),

    ## bottom
    (4, 5),
    (4, 6),
    (5, 7),
    (7, 6),

    ## vertical
    (0, 4),
    (1, 5),
    (3, 7),
    (2, 6),

    ## propeller
    (8, 9)
]
```

Listing 3.3: Posição inicial e massa do CubeCopter

Definindo o momento angular inicial, tempo de simulação e criando a função que aplica a rotação a cada ponto de massa do corpo

```
p_0 = [0,0,1]

tmax = 10
num_times = 400
time = np.linspace(0,tmax,num_times)
r = [0 for i in range(0, len(masses))]

for i in range(0,len(masses)):
    r[i] = lambda t, i=i: r_0[i, :]

tmax_r1 = tmax/20
ang_max = np.pi/2
freq = 2*np.pi/tmax
for i in range(len(masses)-2):
    def ri(t, j=i):
        q = pyQ.Quaternion(axis = [0,1,0], radians = freq*t/tmax_r1)
        return q.rotate(r_0[j])
    r[i] = ri

for i in (8,9):
    def ri(t, j=i):
        q = pyQ.Quaternion(axis = [0,1,0], radians = -freq*t/tmax_r1)
        return q.rotate(r_0[j])
    r[i] = ri
```

Listing 3.4: Posição inicial e massa do CubeCopter

Após importar o módulo `geometricmechanics.py`, para utilizar basta:

```
1 import geometricmechanics as gm
2
3 cube = gm.deformableBody(r_0, masses, p_0, r, num_times, body_lines)
4 cube.translation_CM()
5 q = gm.Physics.solve_eq(cube.p_0, time, r, masses)
6 cube.set_positions(q, cube.rot, tmax)
7 gm.Graphics.create_plot(cube.positions, body_lines, name = 'Cubo')
```

Listing 3.5: Instanciando o objeto cube

Na linha 1, importamos o módulo.
Na linha 3, criamos o objeto cubo a ser modelado.
Na linha 4, efetua-se a translação do Centro de Massa
Na linha 5, resolve-se a equação diferencial e salva na variável q
Na linha 6, guarda as posições do corpo
Na linha 7, cria-se a animação

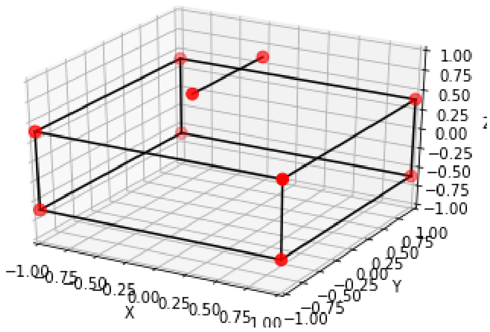

Fig. 3.1: CubeCopter

Apêndice

4

4.1 A função get_InertialTensor

`get_InertialTensor` é uma função estática da classe `Physics` Seja m_i a massa do ponto i e $r_i = (x_i, y_i, z_i)$ seu vetor posição, definindo:

$$I_{xx} = \sum_i m_i(y_i^2 + z_i^2)$$
$$I_{yy} = \sum_i m_i(z_i^2 + x_i^2)$$
$$I_{zz} = \sum_i m_i(x_i^2 + y_i^2)$$
$$I_{xy} = Iyx = -\sum_i m_i x_i y_i$$
$$I_{xz} = Izx = -\sum_i m_i x_i z_i$$
$$I_{yz} = Izy = -\sum_i m_i y_i z_i$$

Temos o Tensor Momento de Inércia **I**:

$$\mathbf{I} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix}$$

Para tirar vantagem da velocidade da biblioteca `Numpy`, fazemos as operações de forma matricial, sendo R a matriz com vetores linha posição r_i, temos que para o cálculo de I_{xx} podemos tomar uma submatriz $R_{2,3}$ que consta da matriz formada pelas colunas 2 e 3 da matriz R. Isto é, as coordenadas y_i, z_i em forma de vetores coluna. É efetuado a operação de elevar ao quadrado cada coordenada destes vetores coluna, e é efetuada a soma coordenada a coordenada entre os dois vetores, a este vetor daremos o nome de r', que contém em cada coordenada a soma $(y_i^2 + z_i^2)$. Então I_{xx} se resume a um produto escalar $\langle m, r' \rangle$. O código que efetua todas estas operações utiliza somente operações matriciais do `Numpy`, e portanto é mais veloz.

```
I[0][0] = np.dot(m,np.sum(r[:,1:3]**2, axis = 1))
```

O mesmo é feito para o cálculo de I_{yy} e I_{zz}, tomando as correspondentes colunas da matriz R.
Também para o cálculo de I_{xy}, I_{xz}, I_{yz} usamos a mesma idéia, mas agora multiplicando os vetores colunas coordenada a coordenada, como no código de exemplo:

```
I[0][1] = np.dot(m,-np.multiply(r[:,0],r[:,1]))
```

4.2 A função CM

`CM` calcula o centro de massa do corpo

$$CM = \frac{\sum_i m_i r_i}{M}$$

Onde $M = \sum_i m_i$
Para tirar vantagem da velocidade do `Numpy`, utilizamos novamente a fórmula de maneira matricial, seja R uma matriz onde cada linha é um vetor r_i posição do ponto i do corpo, e m o vetor de massas de cada ponto do corpo, podemos fazer:

$$CM = \frac{R^t m}{M}$$

E a função `translation_CM` translada o problema de maneira que o centro de massa esteja na origem

Bibliography

[1] Alejandro Cabrera *Fases Geométrica en Sistemas Mecánicos*. Tese de Doutorado em Universidad Nacional de La Plata, 2007.

[2] Luiz Velho e Jonas Gomes *Fundamentos da Computação Gráfica*. 1ª edição. Brasil. IMPA, 2015

[3] Iago Leal de Freitas *A geometria de um gato em queda e outros corpos deformáveis*.