Relatório Técnico

MAP3122 – Cursos Cooperativos
02.04.2024

Caio Escórcio Lima Dourado
NUSP: 13680313

Victor Pedreira dos Santos Pepe
NUSP: 13679565

Conteúdo

1 Introdução

Johannes Kepler, físico famoso pela elaboração das Leis de Kepler da Astronomia, como bem se sabe, dedicou a maior parte da sua vida ao estudo dos corpos celestes e de suas propriedades. Contudo, seja pela tecnologia da época ou até mesmo pela complexidade matemática que envolve o estudo dos astros, muitos de seus problemas ficaram em aberto para as próximas gerações.

Um grande exemplo de problema não solucionado por Kepler é o Problema dos Dois Corpos, que envolvia a análise do movimento de planetas sujeitos aos campos gravitacionais um do outro. Tal problema, apesar de ter sido proposto e estudado pelo astrônomo em 1609, foi solucionado somente cerca de 78 anos depois, em 1687, graças outro tão famoso físico, Isaac Newton.

Assim, tomando como inspiração esse problema quase secular e a relação de movimento Terra-Sol-Lua, surgiu, por volta dos anos 1700, postulado pelo próprio Newton, uma nuance do que seria a ser Problema dos Três Corpos. À época, os objetivos de Newton foram apenas de tentar achar uma estabilidade de movimento entre os corpos do nosso sistema solar, o que não gerou muitas conclusões inovadoras ao considerarmos as suas outras contribuições para a matemática de sua época.

Contudo, com as novas leis físicas impostas por Newton, diversos cientistas do mundo inteiro, agora armados como o Cálculo e com as Leis de Newton, estavam dispostos a explorar as lacunas deixadas pelo físico, entre elas, o Problema dos Três Corpos. Assim, com o passar dos anos, tal desafio tornou-se mais e mais famoso no meios científicos, passando pelas mãos de matemáticos como D'Alembert e Poincaré, até chegar à proposição que temos hoje.

Fato é, esse problema se perpetuou desde o início da teoria gravitacional até os dias de hoje e, afirmativamente, é objeto de estudo de diversas áreas matemáticas, como busca-se mostrar neste documento.

2 Modelagem Matemática

O Problema dos Três Corpos (*Three-Body Problem*, TBP) atualmente envolve diversas áreas matemática, como Equações Diferenciais, Geometria Euclidiana e até mesmo Caos. Sua proposição mais famosa consiste em um sistema de 3 corpos de mesma massa, esféricos e pontuais, localizados nos vértices de um triângulo pitagórico com lados de 3, de 4 e de 5 unidades arbitrariamente grandes quando comparadas ao raio dos corpos – *condições iniciais* do sistema – que são atraídos gravitacionalmente uns pelos outros seguindo a Lei de Newton para Gravitação:

$$\overrightarrow{F} = -\frac{G \cdot m_1 \cdot m_2}{r^2} \cdot \hat{r}$$

onde:

- G é uma constante;
- r é a distância entre os planetas;
- m1 e m2 são as massas dos planetas 1 e 2, respectivamente;

Assim, uma vez que:

$$\overrightarrow{F_x} = m_x \ddot{r_x} \cdot \hat{r_x}$$

E, levando em consideração o plano vetorial bidimensional em que os corpos se encontram, bem como a interação gravitacional par a par temos os seguinte sistema diferencial:

$$\begin{cases} \ddot{\overrightarrow{r_1}} = -Gm_2\frac{\overrightarrow{r_1}-\overrightarrow{r_2}}{|\overrightarrow{r_1}-\overrightarrow{r_2}|^3} - Gm_3\frac{\overrightarrow{r_1}-\overrightarrow{r_3}}{\left|\overrightarrow{r_1}-\overrightarrow{r_3}\right|^3} \\ \ddot{\overrightarrow{r_2}} = -Gm_1\frac{\overrightarrow{r_2}-\overrightarrow{r_1}}{|\overrightarrow{r_2}-\overrightarrow{r_1}|^3} - Gm_3\frac{\overrightarrow{r_2}-\overrightarrow{r_3}}{\left|\overrightarrow{r_2}-\overrightarrow{r_3}\right|^3} \\ \ddot{\overrightarrow{r_3}} = -Gm_1\frac{\overrightarrow{r_3}-\overrightarrow{r_1}}{|\overrightarrow{r_3}-\overrightarrow{r_1}|^3} - Gm_2\frac{\overrightarrow{r_3}-\overrightarrow{r_2}}{\left|\overrightarrow{r_3}-\overrightarrow{r_2}\right|^3} \end{cases}$$

Assim, a partir da análise de diversos trabalhos de pesquisa que envolvem o TBP, percebeu-se que, dependendo das condições iniciais escolhidas, a abordagem do problema tomaria configurações caóticas, fato que será descorrido em tópicos posteriores. Portanto, uma vez que o propósito deste relatório é essencialmente a análise de equações diferenciais a partir de métodos numéricos para compará-las com resultados previstos, optou-se por escolher uma configuração que obedecesse tal critério de previsibilidade. Então, definiu-se como condições iniciais uma configuração que, baseada em diversos estudos, tanto resultasse em um sistema harmônico (com orbitas controladas) quanto em uma figura conhecida e fácil de ser analisada: a configuração do infinito.

$$\begin{cases} \overrightarrow{r_1}_{(0)} = (-0.97000436, 0.24308753, 0); \\ \overrightarrow{r_2}_{(0)} = (0, 0, 0); \\ \overrightarrow{r_3}_{(0)} = (0.97000436, -0.24308753, 0); \end{cases}$$

E:

$$m_1 = m_2 = m_3 = m = 1;$$

Para $\overrightarrow{r_1}$, $\overrightarrow{r_2}$ e $\overrightarrow{r_3}$ vetores do plano $\mathbb{R}^3$ com unidade de medida de distância arbitrária muito maiores que o raio esférico dos corpos – a fim de garantir a sua característica pontual. Por sua vez, as massas m_1, m_2 e m_3 possuem todas massas escalares reais m, também em unidades arbitrárias, para garantir a proporcionalidade das forças e assim atender os critérios para evitar o caos.

Adicionalmente, tem-se que também atribuir condições iniciais nulas de velocidade ($\dot{\vec{r}}$) a fim de se ter puramente o movimento das forças gravitacionais, então:

$$\begin{cases} \dot{\overrightarrow{r_1}}_{(0)} = (0.4662036850, 0.4323657300, 0) \\ \dot{\overrightarrow{r_2}}_{(0)} = (-0.93240737, -0.86473146, 0) \\ \dot{\overrightarrow{r_3}}_{(0)} = (0.4662036850, 0.4323657300, 0) \end{cases}$$

Em unidades arbitrárias de velocidade.

Finalmente, vale destacar que, apesar das massas serem todas iguais, a razão m/G, com G a constante gravitacional, pode alterar a trajetoria dos planetas, uma vez que é o fato multiplicativo da aceleração dos corpos. Portanto, como última condição para o problema, definiu-se $G = 1$, na mesma ordem de grandeza das massas.

Para adaptar o TBP para uma forma legível para discretização, são necessárias algumas tranformações de variáveis já que, essencialmente, esse problema contém configurações de EDO's de segunda ordem (acelerações). Transformações essas que, por comodidade, envolvem o seguinte modelo:

$$\ddot{y} = \frac{d\dot{y}}{dt} = \frac{d(f(y,t))}{dt} = F(y,t)$$

$$\rightarrow \begin{cases} \dot{y} = h \\ \dot{h} = F(y,t) \end{cases}$$

Que aumenta o número de variáveis do sistema. Portanto, ao aplicar essa tranformação em cada um dos eixos x, y e z do $\mathbb{R}^3$ nos vetores que descrevem os movimentos dos corpos, temos:

$$\begin{cases} \dot{\overrightarrow{r_1}} = \overrightarrow{v_1} \\ \dot{\overrightarrow{r_2}} = \overrightarrow{v_2} \\ \dot{\overrightarrow{r_3}} = \overrightarrow{v_3} \\ \\ \dot{\overrightarrow{v_1}} = -Gm_2 \frac{\overrightarrow{r_1}-\overrightarrow{r_2}}{|\overrightarrow{r_1}-\overrightarrow{r_2}|^3} - Gm_3 \frac{\overrightarrow{r_1}-\overrightarrow{r_3}}{\left|\overrightarrow{r_1}-\overrightarrow{r_3}\right|^3} \\ \\ \dot{\overrightarrow{v_2}} = -Gm_1 \frac{\overrightarrow{r_2}-\overrightarrow{r_1}}{|\overrightarrow{r_2}-\overrightarrow{r_1}|^3} - Gm_3 \frac{\overrightarrow{r_2}-\overrightarrow{r_3}}{\left|\overrightarrow{r_2}-\overrightarrow{r_3}\right|^3} \\ \\ \dot{\overrightarrow{v_3}} = -Gm_1 \frac{\overrightarrow{r_3}-\overrightarrow{r_1}}{|\overrightarrow{r_3}-\overrightarrow{r_1}|^3} - Gm_2 \frac{\overrightarrow{r_3}-\overrightarrow{r_2}}{\left|\overrightarrow{r_3}-\overrightarrow{r_2}\right|^3} \end{cases}$$

Vale ressaltar que, apesar de todos os vetores possuírem dimensionalidade 3, o fato deles pertencerem inicalmente ao mesmo triângulo no plano $z = 0$ e inexistirem forças externas atuando no eixo z do sistema – vide 1ª lei de Newton – pode-se considerar o sistema como bidimensional.

3 Metodologia Numérica

3.1 Método de discretização

3.1.1 Método de Runge-Kutta Clássico

Devido à natureza caótica do Problema dos Três Corpos, algumas observações devem ser feitas para que se possa calcular com melhor eficiência o desdobrar as EDO's – vide o apêndice sobre Caos.

Portanto, devido à quantidade de dados a serem processados durante o cálculo das posições dos corpos e a necessidade de precisão nessas medidas, optou-se por selecionar um método de discretização que ao mesmo tempo fosse simples, como também possuísse uma ordem de erro muito pequena: o Método de Runge-Kutta Clássico.

Então, seguindo o estudo do método, dado um Problema de Cauchy, vetorialmente:

$$\begin{cases} \dot{\overrightarrow{y}} = f(t, \overrightarrow{y}) \\ \overrightarrow{y}_{(t_0)} = \overrightarrow{y_0} \end{cases}$$

Com $t \in I = [a, b]$. A discretização pelo Método de Runge-Kutta Clássico é dada por:

$$\overrightarrow{y}_{k+1} = \overrightarrow{y_k} + \overrightarrow{\Phi}(t_k, \Delta t, \overrightarrow{y_k})$$

para:

$$\begin{cases} t_k = t_0 + k\Delta t \quad com \;\; k \in \mathbb{N} : 0, ..., n \\ \Delta t = \frac{b-a}{n} \end{cases}$$

e, dados $\overrightarrow{K_1}$, $\overrightarrow{K_2}$, $\overrightarrow{K_3}$ e $\overrightarrow{K_4}$:

$$\begin{cases} \overrightarrow{K_1} = \Delta t f(t_k, \overrightarrow{y_k}) \\ \overrightarrow{K_2} = \Delta t f(t_k + \frac{1}{2}\Delta t, \overrightarrow{y_k} + \frac{1}{2}\overrightarrow{K_1}) \\ \overrightarrow{K_3} = \Delta t f(t_k + \frac{1}{2}\Delta t, \overrightarrow{y_k} + \frac{1}{2}\overrightarrow{K_2}) \\ \overrightarrow{K_4} = \Delta t f(t_k + \Delta t, \overrightarrow{y_k} + \overrightarrow{K_3}) \end{cases}$$

então:

$$\begin{cases} \overrightarrow{y}_{k+1} = \overrightarrow{y_k} + \overrightarrow{\Phi}(t_k, \Delta t, \overrightarrow{y_k}) \\ \overrightarrow{\Phi}(t_k, \Delta t, \overrightarrow{y_k}) = \frac{1}{6}(\overrightarrow{K_1} + 2\overrightarrow{K_2} + 2\overrightarrow{K_3} + \overrightarrow{K_4}) \end{cases}$$

Dessa forma, com esse método implícito, para utilizando as fórmulas para o cálculo do erro de discretização local, obtém-se que a ordem do erro é $O(\Delta t^4)$, que será mostrada nas próximas seções, na Análise de Resultados.

3.1.2 Implementção em código

Para implementar o algoritmo previamente descrito, optou-se por implementar uma abordagem orientada a objetos em Python(*Object-Oriented Programming*), tomando como base do código a classe *Planeta.py*:

```
import numpy as np

#Definicao de variaveis globais
G = 1 #fonte: https://pt.wikipedia.org/wiki/Constante_gravitacional_universal 6,6743xE-11

class Planeta:

    #Construtor do planeta
    #Cada planeta possui posicao, massa e velocidade como caracteristicas instrinsecas
    def __init__ (self, posicao:np.array, massa:int, velocidade:np.array):
        self.posicao = posicao
        self.massa = massa
        self.velocidade = velocidade

    #Mede a distancia entre dois planetas (funcao estatica)
    def distancia (planeta_1, planeta_2):
        delta = planeta_1.posicao - planeta_2.posicao
        d = np.sqrt(delta.dot(delta)) #raiz do produto escalar entre duas distancias vetoriais

        return d #retorna a distancia escalar entre dois vetores

    def ac_rel (self, planeta_outro):
        ac = np.array([0, 0])
```

```
24          if(np.any(self.posicao != planeta_outro.posicao)):
25              ac = ((-1)*G*(planeta_outro.massa)*np.subtract(
    self.posicao, planeta_outro.posicao))/(Planeta.distancia(
    self, planeta_outro)**3) #ac_rel 1 para 2 = -G*(R_1 - R_2)
    /d^3
26          return ac
27
```

Nessa classe, definie-se um Planeta (corpo), com uma posição, uma massa e uma velocidade. Também implementou-se funções de cálculo de distâncias, que calcula a distância entre dois planetas usando os seus vetores de posição e uma função para calcular a aceleração relativa entre o planeta declarado e um outro planeta, utilizando a equação do $\vec{v}$ da seção 2.

Para o método de discretização, implementou-se uma classe *Discretizacao.py*, para receber as condições iniciais dos planetas e retornar os pontos de suas trajetórias:

```
1 import numpy as np
2 from Classes.Planeta import Planeta
3
4 massa = 1
5 G = 1
6
7 class Discretizacao:
8
9     def __init__(self):
10          pass
11
12      #f(t, y) para o caso dos planetas
13      def f(self, t, y): #y_k[i]
14          #funcao para calcular a aceleracao
15          a1 = lambda c1, c2, c3: c1.ac_rel(c2) + c1.ac_rel(c3)
16          a2 = lambda c1, c2, c3: c2.ac_rel(c1) + c2.ac_rel(c3)
17          a3 = lambda c1, c2, c3: c3.ac_rel(c2) + c3.ac_rel(c1)
18
19          #Criando os planetas
20          planeta_1 = Planeta(np.array([y[0], y[1]]), massa, np
    .array([y[6], y[7]]))
21          planeta_2 = Planeta(np.array([y[2], y[3]]), massa, np
    .array([y[8], y[9]]))
22          planeta_3 = Planeta(np.array([y[4], y[5]]), massa, np
    .array([y[10], y[11]]))
23
```

```
24          #declara o vetor de retorno
25          k = np.array([0.0, 0.0,
26                        0.0, 0.0,
27                        0.0, 0.0,
28                        0.0, 0.0,
29                        0.0, 0.0,
30                        0.0, 0.0])
31
32          #derivada da posicao = velocidade
33          k[0], k[1] = planeta_1.velocidade[0], planeta_1.
    velocidade[1]
34          k[2], k[3] = planeta_2.velocidade[0], planeta_2.
    velocidade[1]
35          k[4], k[5] = planeta_3.velocidade[0], planeta_3.
    velocidade[1]
36
37          #derivada da velocidade = aceleracao
38          k[6], k[7] = a1(planeta_1, planeta_2, planeta_3)
39          k[8], k[9] = a2(planeta_1, planeta_2, planeta_3)
40          k[10], k[11] = a3(planeta_1, planeta_2, planeta_3)
41
42          return k
43
44      #phi generico
45      def phi(self, t, y, h):
46          #declarando phi do Runge-Kutta
47          #PLANETA
48          # k_1 = h*self.f(t, y)
49          # k_2 = h*self.f(t + h/2, y + k_1/2)
50          # k_3 = h*self.f(t + h/2, y + k_2/2)
51          # k_4 = h*self.f(t + h, y + k_3)
52
53          #MANUFATURADA
54          k_1 = h*self.f_manufaturada(t, y)
55          k_2 = h*self.f_manufaturada(t + h/2, y + k_1/2)
56          k_3 = h*self.f_manufaturada(t + h/2, y + k_2/2)
57          k_4 = h*self.f_manufaturada(t + h, y + k_3)
58
59          return (k_1 + 2*k_2 + 2*k_3 + k_4)/6
60
61      def f_manufaturada(self, t, y): #f = y' ; y = e^(3t)*sin
    (5t) - 5; y(0) = -5
62          return np.exp(3*t)*3*np.sin(5*t) + 5*np.exp(3*t)*np.
    cos(5*t)
63
```

```
64      def exata_manufaturada(self, t):
65          """y = e^(3t)*sin(5t) - 5"""
66          return np.exp(3*t)*np.sin(5*t) - 5
67
68      #metodo de discretizacao
69      def calcula(self, n, intervalo_t, dim):
70          #discretizacao do intervalo
71          h = (intervalo_t[1] - intervalo_t[0])/n
72
73          #inicializacao das variaveis
74          i = 0
75          #PLANETAS
76          # t_k = np.zeros((30*n+1, 1))
77          # y_k = np.zeros((30*n+1, dim))
78
79          #MANUFATURA
80          t_k = np.zeros((n+1, 1))
81          y_k = np.zeros((n+1, dim))
82
83          #inicializacao das condicoes iniciais
84          t_k[0] = intervalo_t[0]
85          #PLANETAS
86          # y_k[0] = np.array([
87          #     -0.97000436, 0.24308753,        #R_1: 0, 1
88          #     0.97000436, -0.24308753,        #R_2: 2, 3
89          #     0, 0,                           #R_3: 4, 5
90
91          #     0.4662036850, 0.4323657300,     #V_1: 6, 7
92          #     0.4662036850, 0.4323657300,     #V_2: 8, 9
93          #     -0.93240737, -0.86473146        #V_3: 10, 11
94          # ])
95
96          #MANUFATURA
97          y_k[0] = np.array([-5])
98
99
100         #loop para calcular os valores de y
101
102         #PLANETA
103         #for i in range(30*n):
104
105         #Calculo da discretizacao
106         for i in range(n):
107             y_k[i+1] = y_k[i] + self.phi(t_k[i], y_k[i], h)
108             t_k[i+1] = t_k[i] + h
```

```
109
110         return y_k, t_k
111
112     def converte(self, n):
113         #y_k, t_k = self.calcula(100, [0, 1], 12)
114         y_k, t_k = self.calcula(n, [0, 2], 1)
115         i = 1
116         #PLANETAS
117         # pos1 = np.array([y_k[0][0], y_k[0][1]])
118         # pos2 = np.array([y_k[0][2], y_k[0][3]])
119         # pos3 = np.array([y_k[0][4], y_k[0][5]])
120         # for i in range(30*n):
121         #     pos1 = np.vstack([pos1, np.array([y_k[i][0],
    y_k[i][1]])])
122         #     pos2 = np.vstack([pos2, np.array([y_k[i][2],
    y_k[i][3]])])
123         #     pos3 = np.vstack([pos3, np.array([y_k[i][4],
    y_k[i][5]])])
124         #return pos1, pos2, pos3
125
126         #MANUFATURA
127         return y_k, t_k
128
```

Em uma breve leitura do código, podemos destacar 3 funções principais: *phi()*, *f()* ou $f_{manufaturada}()$ e *calcula()*. Genericamente, cada função representa os passos do método de Runge-Kutta, obedecendo necessariamente a ordem de operação descrito na seção de Descrição do Método. Vale ressaltar que o sistema foi montado para que, modificando adequadamente a função *f()* e o vetor das condições iniciais $y_k[0]$, é possível discretizar quaisquer problemas dados.

Para esclarecer o funcionamento das funções, podemos resumi-las da seguinte maneira:

- phi(t, y, h): Recebe o vetor $y = y_k$, o instante $t = t_k$ e o incremento $h = \Delta t$ e calcular os κ_1, κ_2, κ_3 e κ_4 utilizando a função $f()$ ou $f_{manufaturada}$ e os argumentos da função. Ela retorna o incremento vetorial de y_k para o cálculo de y_{k+1}

- f(t, y): Recebe o vetor $y = y_k$, o instante $t = t_k$ e, dependendo do problema proposto, pode ser modificada livremente para a discretização

do problema. Ela retorna o que corresponderia a $\overrightarrow{\dot{y_k}}$ e é utilizada para o cáculos dos κ's.

- calcula(n, $intervalo_t$, dim): recebe o número de passos n, o intervalo de discretização $intervalo_t$ e o inteiro dim que indica a dimensão dos vetores de y_k do problema. Vale ressaltar que o seu papel é simplesmente discretizar o intervalo de tempo, iniciar o vetor dos y's e realizar o *loop* para o cálculo dos demais pontos da função utilizando phi(t, y, h) e f(t, y). Ela retorna os vetores finais de todos os y_k e os t_K do intervalo.

Finalmente, vale ressaltar o papel da função *converte()*, que simplesmente gera os vetores de menor dimensão que serão plotados/utilizados para a verificação do método. Uma vez que o objetivo da disciplina não é necessariamente ensinar lógica de programação, o funcionamento a fundo do código não será explicado. Também recomenda-se uma breve leitura sobre o funcionamento da linguagem Python e da sua biblioteca NumPy para o entendimento completo do programa.

3.2 Modelagem polinomial dos resultados obtidos

A partir dos pontos gerados pelo algoritmo de Runge-Kutta multidimensional para o problema, utilizou-se os pontos obtidos para a modelagem da trajetória dos corpos via Spline Vinculado.

3.2.1 Algoritmo do Spline Vinculado e resolução de sistemas lineares

Após a discretização do Problema dos 3 Corpos com o uso do Runge-Kutta Clássico, foram obtidos - para cada corpo - múltiplos pares coordenados (x, y), os quais descrevem a trajetória percorrida por eles. Com todos esses valores em mãos, é possível realizar uma modelagem polinomial por meio dos splines cúbicos.

Esse método numérico é muito utilizado quando se deseja aproximar (ou mesmo se obter) uma função desconhecida a partir de pontos conhecidos. A ideia básica dele é aproximar a função desconhecida utilizando múltiplos polinômios de terceiro grau, cada um deles abrangendo um sub-intervalo (todos de mesmo tamanho) da função. Quanto aos coeficientes dos polinômios, esses serão obtidos a partir da resolução de sistemas lineares, como veremos mais adiante.

Uma grande vantagem do uso dos splines cúbicos sobre outros métodos, como a interpolaçao de Lagrange, é que este último - principalmente quando se trata de problemas complexos e caóticos - gera um polinômio de grau elevado, o qual pode flutuar de forma drástica, de modo que uma pequena flutuação localizada pode induzir flutuações grandes no intervalo inteiro. Já o método dos splines cúbicos, formado por múltiplos polinômios sucessivos que apresentam continuidade primeira e na segunda derivada, resulta numa forma mais suave. Um exemplo gráfico da aproximação por splines cúbicos pode ser visualizado na figura 1:

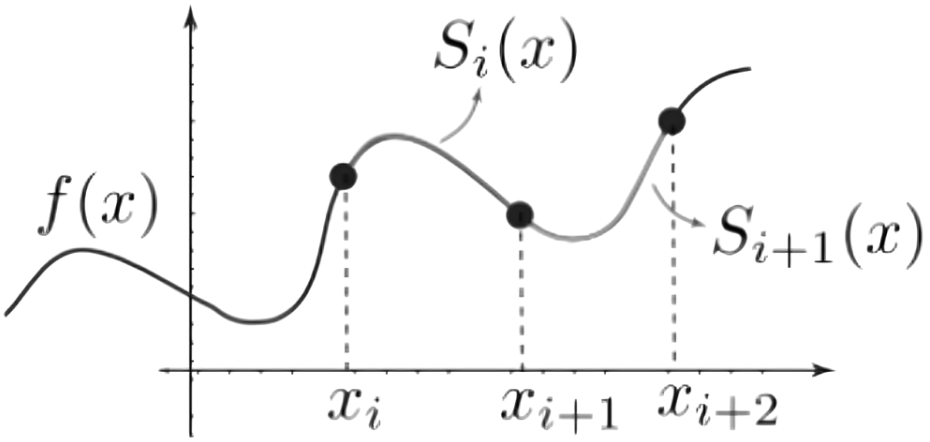

Figura 1: Interpolação por splines cúbicos

No problema abordado nesse relatório, foi possível obter uma série de pontos aproximados a partir da discretização de EDOs, mas não as funções que deram origem a elas, o que torna o uso dos splines extremamente conveniente na modelagem do problema dos 3 corpos.

Vale ressaltar que, como o problema é caótico, os polinômios obtidos a partir da resolução dos sistemas lineares de cada sistema será muito diferente dos demais para cada condição inicial selecionada.

Quanto ao método dos splines cúbicos, vamos supor que desejamos aproximar uma função $f(x)$, da qual utilizaremos n pontos igualmente espaçados no eixo x. Seja $x = x_0, x_1, x_2, ..., x_n$ e $f(x) = f(x_0), f(x_1), f(x_2), ..., f(x_n)$ o conjunto de pontos coletados, temos que $h_0, h_1, h_2, ..., h_{n-1}$ são os intervalos entre duas abscissas consecutivas (ou seja, $h_k = x_{k+1} - x_k$). Além disso, denominamos $S(x)$ o polinômio completo para todo o intervalo $I = [a, b]$ no qual os x_n pontos estão contidos.

Importante ressaltar também que iremos trabalhar com splines *vinculados*, ou seja, a primeira derivada de $S(x)$ coincide com a segunda derivada da f tanto no início quanto no final de I. Assim, temos:

$$\begin{cases} S(x) = S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3, \quad x \in [x_j, x_{j+1}] \\ S(x_j) = f(x_j), \quad j = 0, 1, 2, ..., n \\ S_{j+1}(x_{j+1}) = S_j(x_{j+1}), \quad j = 0, 1, 2, ..., n-2 \\ S'_{j+1}(x_{j+1}) = S'_j(x_{j+1}), \quad j = 0, 1, 2, ..., n-2 \\ S''_{j+1}(x_{j+1}) = S''_j(x_{j+1}), \quad j = 0, 1, 2, ..., n-2 \\ S''(x_0) = f'(x_0) \\ S''(x_n) = f'(x_n) \end{cases}$$

Com todas as propriedades do spline vinculado descritas nas equações acima e realizando algumas substituições, será possivel determinar os coeficientes dos polinômios a partir de um sistema linear. Inicialmente, tomemos $x = x_{j+1}$:

$$\begin{cases} S_{j+1}(x_{j+1}) = S_j(x_{j+1}) = a_{j+1} = a_j + b_j h_j + c_j h_j^2 + d_j h_j^3, \quad j = 0, 1, 2, ..., n-1 & (1) \\ S'_{j+1}(x_{j+1}) = S'_j(x_{j+1}) = b_{j+1} = b_j + c_j h_j + 3 d_j h_j^2, \quad j = 0, 1, 2, ..., n-1 & (2) \\ S''_{j+1}(x_{j+1}) = S''_j(x_{j+1}) = 2c_{j+1} = 2c_j + 6 d_j h_j, \quad j = 0, 1, 2, ..., n-1 & (3) \end{cases}$$

Assim, definimos $S'(x_n) = b_n$, $\frac{S''(x_n)}{2} = c_n$ e $f(x_n) = S(x_n) = a_n$. Isolando d_j em (3) e substiuindo em (1) e (2), chegamos que:

$$\begin{cases} a_{j+1} = a_j + b_j h_J + \dfrac{h_j^2}{3}(2c_j + c_{j+1}) & (4) \\ b_{j+1} = b_j + h_j(c_j + c_{j+1}) & (5) \end{cases}$$

Isolando b_j em (4) e substituindo j por $j - 1$ chegamos que:

$$b_{j-1} = \frac{1}{h_{j-1}}(a_j - a_{j-1}) - \frac{h_{j-1}}{3}(2c_{j-1} + c_j) \qquad (6)$$

Substituindo j por $j - 1$ em (5) e realizando algumas substituições com (6), temos que:

$$h_{j-1} + 2(h_{j-1} + h_j)c_j + h_j c_{j+1} = \frac{3}{h_j}(a_{j+1} - a_j) - \frac{3}{h_{j-1}}(a_j - a_{j-1}) \qquad (7)$$

Com todas as equações numeradas, conseguimos formar um sistema possível e determinado cujas incógnitas são os coeficientes dos polinômios interpoladores. Desse modo, teremos as seguintes matrizes:

$$A = \begin{bmatrix} 2h_0 & h_0 & 0 & \dots & \dots & 0 \\ h_0 & 2(h_0+h_1) & h_1 & \ddots & \ddots & \vdots \\ 0 & h_1 & 2(h_1+h_2) & h_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \\ 0 & \dots & \dots & 0 & h_{n-1} & 2h_{n-1} \end{bmatrix},$$

$$X = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix},$$

e

$$b = \begin{bmatrix} \frac{3}{h_0}(a_1 - a_0) - 3f'(a) \\ \frac{3}{h_1}(a_2 - a_1) - \frac{3}{h_0}(a_1 - a_0) \\ \vdots \\ \frac{3}{h_{n-1}}(a_n - a_{n-1}) - \frac{3}{h_{n-2}}(a_{n-1} - a_{n-2}) \\ 3f'(b) - \frac{3}{h_{n-1}}(a_n - a_{n-1}) \end{bmatrix}$$

Por fim, teremos:

$$A \cdot X = b$$

Após realizado todo esse processo, teremos interpolado os pontos obtidos pela discretização dos planetas através do Runge-Kutta Clássico com o uso de splines cúbicos. Computacionalmente, será possível resolver o sistema linear descrito e obter todas os $c_j, j \in [0, n]$. Para obter os coeficientes d_j, basta substituir c_j em (3). Já para os coeficientes b_j, basta substituir c_j e d_j e a_j em (1). Note que não precisamos calcular a_j, pois definimos $a_n = f(x_n)$. Importante ressaltar também que, pelo teorema da existência e unicidade do spline vinculado, temos que o conjunto de pontos $(x_k, f(x_k)), k \in 0, 1, 2, ..., n$ admite um único spline vinculado $S(x)$.

3.2.2 Implementção em código

Para implementar o spline em Python, utilizou-se como base o algoritmo em pseudocódigo da página 149-150 do livro Análise Numérica, BURDEN et al:

"To construct the cubic spline interpolant S for the function f , defined at the numbers $x_0 < x_1 < ... < x_n$, satisfying $S(x_0) = S(x_n) = 0$:

INPUT n; x_0 , x_1 , ... , x_n ; $a_0 = f(x_0)$, $a_1 = f(x_1)$, ..., $a_n = f(x_n)$.

OUTPUT a_j , b_j , c_j , d_j for j = 0, 1, ..., n - 1.

(Note: $S(x) = S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3$ for $x_j \leq x \leq x_{j+1}$.)

Step 1: For i = 0, 1, ..., n - 1 set $h_i = x_{i+1} - x_i$.
Step 2: For i = 1, 2, ..., n - 1 set

$$\alpha_i = \frac{3}{h_1}(a_{i+1} - a_i) - \frac{3}{h_{i-1}}(a_i - a_{i-1})$$

Step 3: Set:

$$l_0 = 1;$$

$$\mu_0 = 0;$$

$$z_0 = 0;$$

(Steps 3, 4, 5, and part of Step 6 solve a tridiagonal linear system using a method described in Algorithm 6.7.)

Step 4: For i = 1, 2, ..., n - 1
set $l_i = 2(x_{i+1} - x_{i-1}) - h_{i-1}\mu_{i-1}$;
$\mu_i = h_1/l_1$;
$z_i = (\alpha_i - h_{i-1}z_{i-1})/l_i$.

Step 5: Set:

$$l_n = 1;$$

$$\mu_n = 0;$$

$$z_n = 0;$$

Step 6: For j = n - 1, n - 2, ..., 0

set $c_j = z_j - \mu_j c_{j+1}$;
$b_j = (a_{j+1} - a_j)/h_j - h_j(c_{j+1} + 2c_j)/3$;
$d_j = (c_{j+1} - c_j)/(3h_j)$.

Step 7: OUTPUT a_j, b_j, c_j, d_j for j = 0, 1, ..., n - 1;

STOP."

Vale ressaltar que a implementação desse algoritmo não só gera os pontos do Spline como soluciona o sistema linear tridiagonal que aparece para solucionar as equações do método.

Traduzindo o método para uma linguagem de programação, Python, elaborou-se a seguinte classe *Spline.py*:

```
1
2
3 import numpy as np
4
5 class CubicSpline():
6     def __init__(self, x, y):
7         assert len(x) == len(y), "x e y devem ter o mesmo
    tamanho!"
8         self.x = x
9         self.y = y
10        self.n = len(x) #tamanho do array (OBS: self.n = 'n'
    + 1 do livro!)
11
12        #ALGORITMO DO SPLINE VINCULADO: LIVRO DO BURDEN PAG
    149
13        #PASSO 1
14        self.h = np.diff(x) #deltas x (h[0] = x[1] - x[0])
15                            #h vai de 0 ate n-1
16
17        #PASSO 2
18        self.alpha = np.zeros(self.n)
19        self.alpha[1:-1] = (3/self.h[1:])*(self.y[2:] - self.
    y[1:-1]) - (3/self.h[:-1])*(self.y[1:-1] - self.y[:-2])
20
21        #PASSO 3
22        self.l = np.zeros(self.n)
23        self.mu = np.zeros(self.n)
24        self.z = np.zeros(self.n)
25        self.l[0] = 1
26        self.mu[0] = 0
```

```
27          self.z[0] = 0
28
29          # PASSO 4
30          for i in range(1, self.n-1):
31              self.l[i] = 2*(self.x[i+1] - self.x[i-1]) - self.
    h[i-1]*self.mu[i-1]
32              self.mu[i] = self.h[i]/self.l[i]
33              self.z[i] = (self.alpha[i] - self.h[i-1]*self.z[i
    -1])/self.l[i]
34
35          #PASSO 5
36          self.l[-1] = 1
37          self.z[-1] = 0
38          self.c = np.zeros(self.n)
39
40          #PASSO 6
41          self.b = np.zeros(self.n)
42          self.d = np.zeros(self.n)
43
44          print(len(self.c), len(self.z), len(self.mu), len(
    self.y), len(self.h), len(self.b))
45          print(len(self.c[-1:1:-1]), len( 2*self.c[-2:0:-1]))
46          for i in range(self.n-2, 0, -1):
47              self.c[i] = self.z[i] - self.mu[i]*self.c[i+1]
48
49              self.b[i] = (self.y[i+1] - self.y[i])/self.h[i] -
     self.h[i]*(self.c[i+1] + 2*self.c[i])/3
50
51              self.d[i] = (self.c[i+1] - self.c[i])/(3*self.h[i
    ])
52
53          #PASSO 7
54          self.splines = np.vstack((self.y[0:-1], self.b[0:-1],
     self.c[0:-1], self.d[0:-1])).T
55
56      def __call__(self, x): #recebe um vetor X para devolver
    um vetor Y interpolado pelo spline cubico
57          y = np.vectorize(self.Substitui)(x)
58          return y
59
60      def Substitui(self, x):
61          idx = np.searchsorted(self.x, x, side="right") - 1 #
    encontra o indice do intervalo no qual x esta contido
62          return self.splines[idx][0] + self.splines[idx][1]*(x
    -self.x[idx]) + self.splines[idx][2]*(x-self.x[idx])**2 +
```

```
self.splines[idx][3]*(x-self.x[idx])**3
```

Em resumo, a classe recebe um vetor 'x' e um vetor 'y', aplica vetorialmente o algoritmo em pseudocódigo descrito anteriormente, e retorna os pontos 'y' encontrados via Spline a partir dos pontos 'x' das trajetórias dos corpos.

Enfim, com todos os códigos em mãos, configurou-se o seguinte código para a função *main.py*:

```
import numpy as np
from Classes.Discretizacao import Discretizacao as disc
from Classes.Planeta import Planeta
from Classes.Splines import CubicSpline
#from scipy.interpolate import CubicSpline
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import plotly.graph_objects as go


#massa padrao de todos os planetas
massa = 1
v_0 = np.array([0.4662036850, 0.4323657300])
a, b = 3, 4

corpo_1 = Planeta(np.array([-0.97000436, 0.24308753]), massa,
    v_0)
corpo_2 = Planeta(np.array([0.97000436, -0.24308753]), massa,
    v_0)
corpo_3 = Planeta(np.array([0, 0]), massa, np.array
   ([-0.93240737, -0.86473146]))
labels = ['Corpo 1', 'Corpo 2', 'Corpo 3']
n = 100
T = 1
discretizacao = disc()
np.set_printoptions(threshold=np.inf)
pos1, pos2, pos3 = discretizacao.rungekutta_planetas(corpo_1,
    corpo_2, corpo_3, n, T)

fig, ax = plt.subplots()

ax.set_xlim(-1.5, 1.5)
ax.set_ylim(-1.5, 1.5)

# Create empty lines for the plot
line1, = ax.plot([], [], 'r-', lw=2)
```

```
line2, = ax.plot([], [], 'g-', lw=2)
line3, = ax.plot([], [], 'b-', lw=2)

# Create empty scatter plots for the balls
ball1, = ax.plot([], [], 'ro', markersize=5)
ball2, = ax.plot([], [], 'go', markersize=5)
ball3, = ax.plot([], [], 'bo', markersize=5)

# Animation function
def update(frame):
    line1.set_data(pos1[:frame, 0], pos1[:frame, 1])
    line2.set_data(pos2[:frame, 0], pos2[:frame, 1])
    line3.set_data(pos3[:frame, 0], pos3[:frame, 1])

# Create empty scatter plots for the balls
ball1, = ax.plot([], [], 'ro', markersize=5)
ball2, = ax.plot([], [], 'go', markersize=5)
ball3, = ax.plot([], [], 'bo', markersize=5)

# Animation function
def update(frame):
    line1.set_data(pos1[:frame, 0], pos1[:frame, 1])
    line2.set_data(pos2[:frame, 0], pos2[:frame, 1])
    line3.set_data(pos3[:frame, 0], pos3[:frame, 1])
    ball1.set_data([pos1[frame, 0]], [pos1[frame, 1]])
    ball2.set_data([pos2[frame, 0]], [pos2[frame, 1]])
    ball3.set_data([pos3[frame, 0]], [pos3[frame, 1]])

    return line1, line2, line3, ball1, ball2, ball3

# Create the animation
ani = FuncAnimation(fig, update, frames=len(pos1), interval
    =1, blit=True)
plt.show()

#SPLINE CUBICO
pos_stack = np.array( [pos1, pos2, pos3] )
t = np.linspace(0, 30*T, len(pos1[:,0]))
t_plot = np.arange(0, 30*T,  T/(20*n))
csx = np.array(list(map( lambda pos: CubicSpline(t, pos[:,
    0]), pos_stack)))
csy = np.array(list(map( lambda pos: CubicSpline(t, pos[:,
    1]), pos_stack)))

# Create figure
```

```
fig = go.Figure()

# Add real values
fig.add_trace(go.Scatter(x=pos1[:, 0], y=pos1[:, 1], mode='
    lines', name='Real Pos1'))
fig.add_trace(go.Scatter(x=csx[0](t_plot)[0:], y=csy[0](
    t_plot)[0:], mode='lines', name=f'Spline Pos 1'))

# Set layout
fig.update_layout(
    title='Movimento de tres bolas',
    xaxis_title='X (m)',
    yaxis_title='Y (m)',
    legend_title='Legend',
    showlegend=True
)

fig.show()
```

Em resumo, nesse código são inseridas as declarações dos 3 planetas, com suas respectivas condições iniciais como variáveis para que seja feita a discretização. Após a criação dos corpos, declara-se a classe de discretização *disc()* para que sejam retornados 3 vetores de posição, um para cada planeta. Após isso, têm-se uma função de plot para a vizualização da trajetória. Finalmente, são declarados obejtos *CubicSpline()* para que sejam gerados os pontos do Spline. Também foi implementada uma função de plot para a sua vizualização.

Após a execução do código (*"python3 -m Main.main"*, no terminal aberto na pasta do projeto), é possível vizualizar as seguintes figuras:

Respectivamente, a trajetória dos 3 planetas formando o símbolo do infinito e uma vista em escala mínima do gráfico de spline gerado por uma das trajetórias. Detalhe para a segunda imagem, onde as linhas em azul são os pontos gerados pelo Runge-Kutta e onde a linha vermelha é o spline que passa por todas os pontos. Os resultados gráficos obtidos foram extremamente satisfatórios.

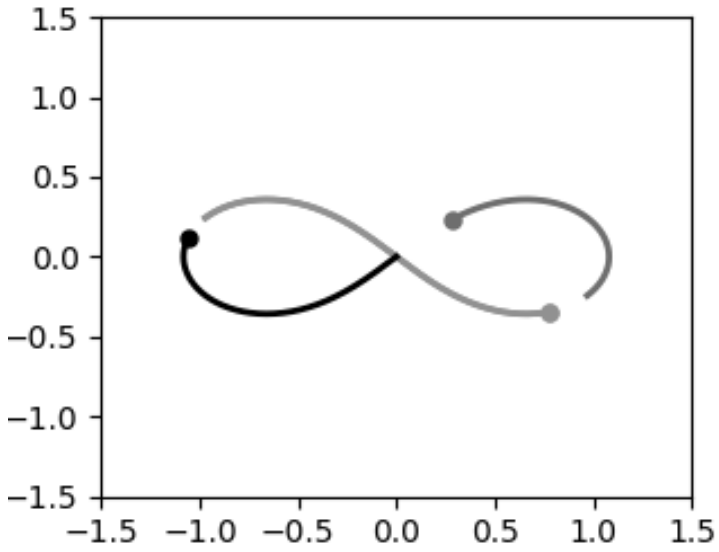

Figura 2: Discretização do problema dos 3 corpos com Runge Kutta

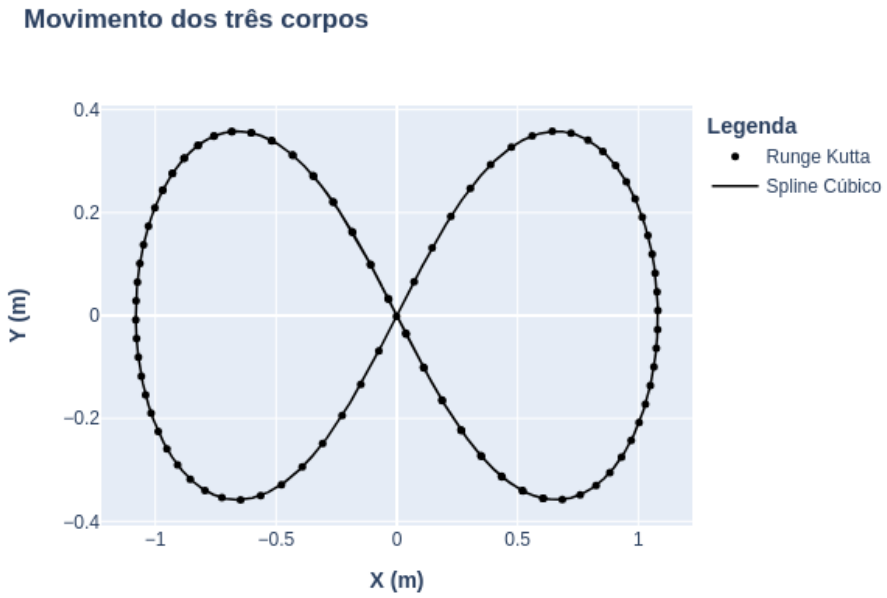

Figura 3: Zoom na modelagem polinomial com splines cúbicos

4 Resultados

4.1 Verificação via solução manufaturada

Após implementados os algoritmos em Python, como descrito na seção anterior, foi possível aproximar numericamente a solução do TBP. Para isso, problema foi discretizado com o método Runge Kutta e os pontos obtidos foram modelados com splines cúbicos, sendo que esse último método demandou técnicas de resolução de sistemas lineares para obter os coeficientes dos polinômios interpoladores.

Antes de apresentar os resultados e as tabelas de convergência para o TBP, é preciso verificar se os algoritmos implementados funcionam de forma ideal. Isso é necessário pois o problema do TBP é caótico e, portanto, não possui solução exata conhecida. Portanto, basta tomar um problema com solução exata conhecida, discretizá-lo com o Runge Kutta e analisar sua ordem de convergência. Tomemos, então, a seguinte igualdade:

$$y = e^{3t} \cdot sen(5t) - 5 \tag{8}$$

Assim, podemos discretizá-lo para manufaturar nosso problema:

$$y' = 3e^{3t} \cdot sen(5t) + 5e^{3t} \cdot cos(5t) = f(t) \tag{9}$$

Definida o problema manufaturado, será preciso modificar a função de discretização utilizada no código (sem, entretanto, modificar nenhum outro trecho do algoritmo) a fim de garantir que o método utilizado converge com a ordem esperada (no caso do Runge Kutta, ordem 4).

Assim, dentro da classe *Discretizacao*, implementamos dois novos métodos. Um deles é a função de discretização *f_manufaturada*, que recebe a variável independente t como parâmetro e devolve o resultado descrito em (9). O outro é a solução exata *exata_manufaturada*, que possui a mesma entrada do método anterior e devolve a solução exata (8). Observe a baixo a codificação de cada um desses métodos:

```
1     def f_manufaturada(self, t):
2         """f = y' ; y = e^(3t)*sin(5t) - 5; y(0) = -5"""
3         return np.exp(3*t)*3*np.sin(5*t) + 5*np.exp(3*t)*np.
    cos(5*t)
4
5     def exata_manufaturada(self, t):
6         """y = e^(3t)*sin(5t) - 5"""
7         return np.exp(3*t)*np.sin(5*t) - 5
```

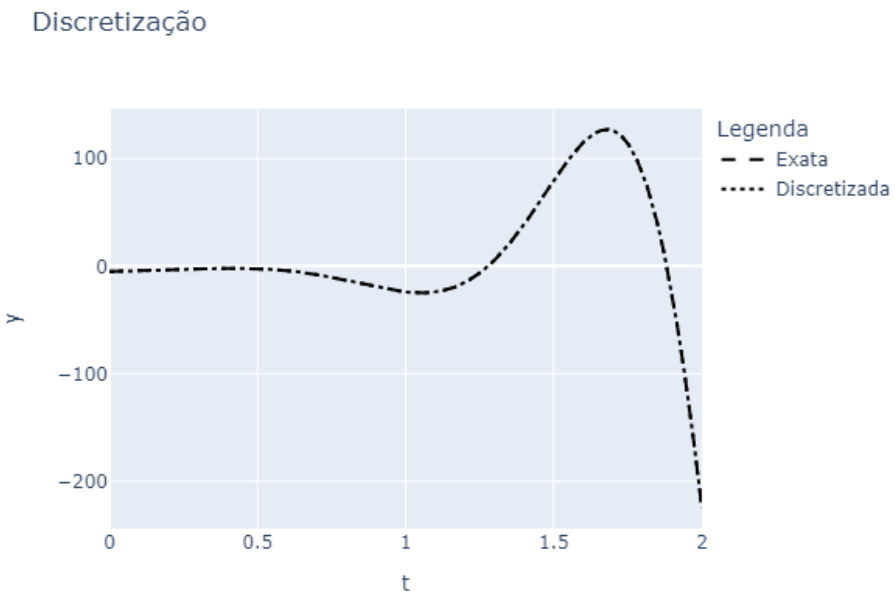

Figura 4: Discretização do problema manufaturado com Runge Kutta

Após executar o código *main_ manufaturada.py* com o intervalo $I = [0, 2]$ com 4096 passos, obtivemos o gráfico contido na figura 4.

Como visto na figura, a aproximação é tão precisa que ocorre sobreposição de pixels na imagem. Somente após incrementar múltiplas vezes a escala da imagem (dentro do próprio Python com o pacote do *Plotly* utilizado) é que torna-se possível observar uma mínima diferença nos pixels, o que torna o resultado obtido - sob uma primeira impressão - bastante satisfatório.

Além da inspeção visual do resultado, foram geradas também duas tabelas de convergência para o problema manufaturado.

A primeira tabela foi contruída com base na solução exata. Inicialmente, foi preciso calcular o erro e das discretizações para cada número de passos $n \in [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]$. Em seguida, foi calculada a *ordem* $\overline{p}$ do método. Para isso, utilizamos as seguintes expressões:

$$e(T, h_n) = |y(T) - \eta(T, h)|$$

$$\log_r \frac{e(T, h_n)}{e(T, h_{n+1)}}$$

Onde r é a razão entre os intervalos de tempo h para cada n (no caso dos valores escolhidos, $r = 2$, T é o instante final de tempo - ou seja, o extremo direito de I) e $\eta(T, h)$ é a aproximação numérica no instante $t = T$ com passo de tamanho h. Observe abaixo a tabela de convergência para o problema manufaturado:

n	h	$erro$	$ordem\ \overline{p}$
2.0000e+00	1.0000e+00	8.4996e+01	0.0000e+00
4.0000e+00	5.0000e-01	1.0854e+01	2.9692e+00
8.0000e+00	2.5000e-01	6.5063e-01	4.0602e+00
1.6000e+01	1.2500e-01	3.9891e-02	4.0277e+00
3.2000e+01	6.2500e-02	2.4801e-03	4.0076e+00
6.4000e+01	3.1250e-02	1.5480e-04	4.0019e+00
1.2800e+02	1.5625e-02	9.6717e-06	4.0005e+00
2.5600e+02	7.8125e-03	6.0443e-07	4.0001e+00
5.1200e+02	3.9062e-03	3.7776e-08	4.0000e+00
1.0240e+03	1.9531e-03	2.3610e-09	4.0000e+00
2.0480e+03	9.7656e-04	1.4737e-10	4.0019e+00
4.0960e+03	4.8828e-04	9.2086e-12	4.0003e+00

Tabela 1: Convergência do problema manufaturado via solução exata

Como esperado, temos que para números de passos progressivamente maiores, o erro converge a 0 com ordem 4.

4.2 Aplicação

Tendo em vista que a tabela anterior trata-se de um problema manufaturado, com solução exata e conhecida, sua análise singular é insuficiente para contemplar a veracidade do método para o problema dos 3 corpos. Para múltiplos outros problemas, cuja solução não é conhecida (como sistemas caóticos, similares ao TBP), é preciso utilizar outra abordagem para calcular a ordem de convergência. Uma maneira de obter essa ordem é através das diferenças relativas entre aproximações sucessivamente mais refinadas. Para isso, utilizamos a seguinte expressão:

$$\overline{p} \approx \log_r \frac{e(T, h_n)}{e(T, h_{n+1})} = \log_r \left(\left| \frac{\eta(t, rh) - \eta(t, h)}{\eta(t, h) - \eta(t, h/r)} \right| \right)$$

Aplicando esse método nos pontos obtidos com a solução manufaturada, obtém-se a seguinte tabela:

$\eta(t, 2h)$	$\eta(t, h)$	$\eta(t, h/2)$	$ordem\ \overline{p}$
-1.3948e+02	-2.1362e+02	-2.2382e+02	2.8613e+00
-2.1362e+02	-2.2382e+02	-2.2443e+02	4.0623e+00
-2.2382e+02	-2.2443e+02	-2.2447e+02	4.0290e+00
-2.2443e+02	-2.2447e+02	-2.2447e+02	4.0080e+00
-2.2447e+02	-2.2447e+02	-2.2447e+02	4.0020e+00
-2.2447e+02	-2.2447e+02	-2.2447e+02	4.0005e+00
-2.2447e+02	-2.2447e+02	-2.2447e+02	4.0001e+00
-2.2447e+02	-2.2447e+02	-2.2447e+02	4.0000e+00
-2.2447e+02	-2.2447e+02	-2.2447e+02	3.9999e+00
-2.2447e+02	-2.2447e+02	-2.2447e+02	4.0020e+00

Tabela 2: Convergência do problema manufaturado via diferenças relativas

Novamente, ainda que sem o uso da solução exata, verifica-se que o método converge a 0 com ordem 4. Importante ressaltar que enquanto o segundo método indica a proximidade entre as próprias discretizações, o primeiro revela a proximidade entre as discretizações com o resultado exato (para h progressivamente refinado).

Aplicando, agora, os algoritmos implementados no problema dos 3 corpos, no intervalo de tempo $I = [0, 8]$ com $n \in [4, 8, 16, 32, 64, 128]$ passos, utilizando as seguintes condições iniciais:

Corpo	$x(0)$ [m]	$y(0)$ [m]	$v_x(0)$ [m/s]	$v_y(0)$ [m/s]
1	-9.7000436e-01	2.4308753e-01	4.66203685e-01	4.32365730e-01
2	9.7000436e-01	-2.4308753e-01	4.66203685e-01	4.66203685e-01
3	0	0	-9.3240737e-01	-8.6473146e-01

Tabela 3: Condições inicias do TBP

Para os cálculos foi utilizada a mesma massa de 1 Kg para cada corpo e a constante gravitacional G = 1. Após gerar os pontos com as condições iniciais descritas, é possível obter uma interessante trajetória na qual os corpos movimentam-se formando o símbolo do infinito, como mostrado na figura 2.

Como há um grande número de variáveis no problema (6 ao todo, sendo elas a posição vertical e horizontal de cada um dos 3 corpos), optou-se por

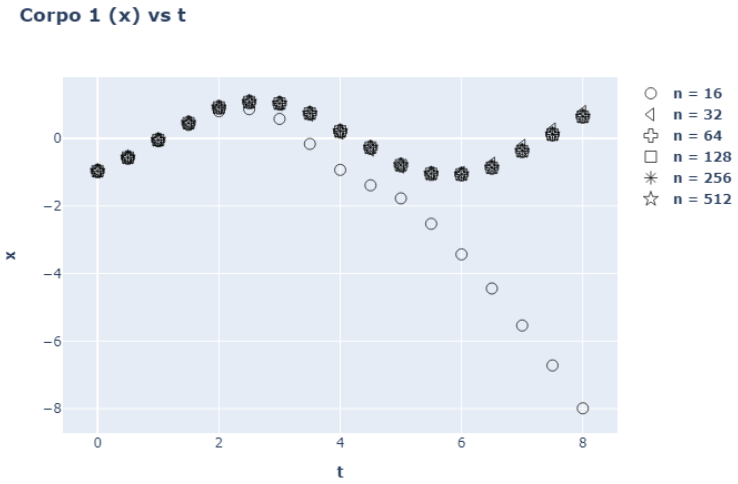

Figura 5: Discretização do TBP com Runge Kutta - Corpo 1 no eixo x

gerar um gráfico para cada uma delas a fim de facilitar a visualização, visto que a sobreposição de todos os elementos impossibilita a distinção das trajetórias para n diferentes. As posições x e y de cada corpo, para múltiplos valores de n se encontram nas figura 5 a 10:

Pelos gráficos, fica nítido como as trajetórias convergem para n progressivamente maiores na razão 2, tendo em vista que as linhas de movimento de cada corpo começam a ficar indistinguíveis para $n > 512$. Observe, agora, as tablas de convergência via diferenças relativas para as posições x e y obtidas em cada corpo com o uso do Runge Kutta:

h	$x_1(2h)$	$x_1(h)$	$x_1(h/2)$	ordem $\overline{p}$
3.1250e-02	-7.9874e+00	7.8274e-01	6.4587e-01	6.0017e+00
1.5630e-02	7.8274e-01	6.4587e-01	6.4002e-01	4.5492e+00
7.8100e-03	6.4587e-01	6.4002e-01	6.3979e-01	4.6863e+00
3.9100e-03	6.4002e-01	6.3979e-01	6.3978e-01	4.5725e+00

Tabela 4: Convergência com Runge Kutta: Corpo 1 (x)

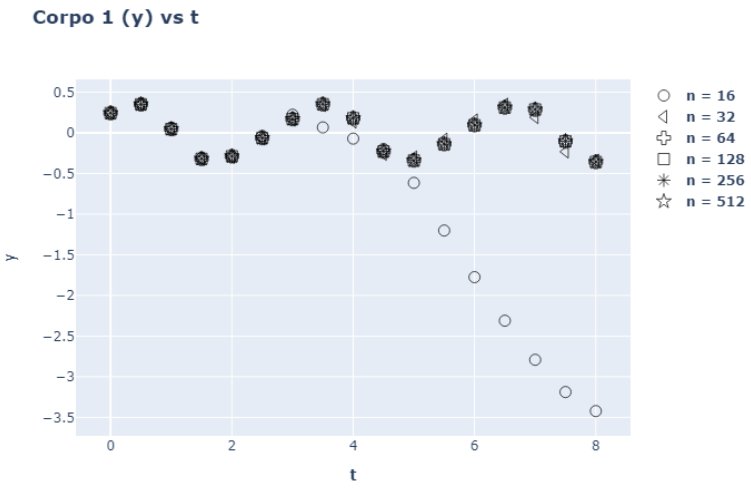

Figura 6: Discretização do TBP com Runge Kutta - Corpo 1 no eixo y

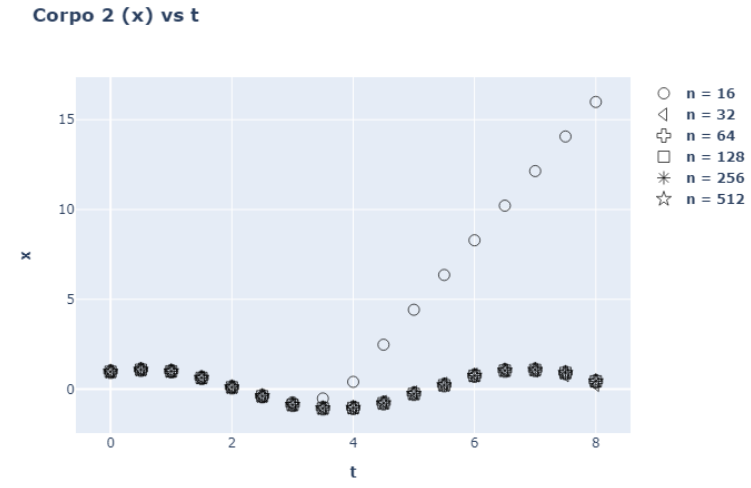

Figura 7: Discretização do TBP com Runge Kutta - Corpo 2 no eixo x

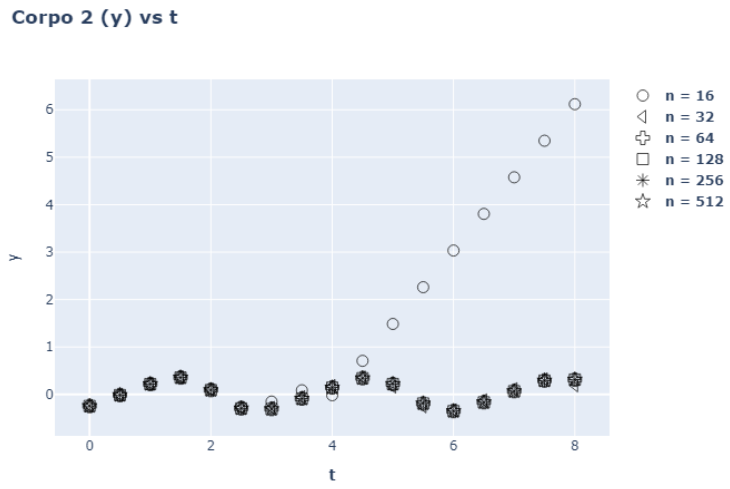

Figura 8: Discretização do TBP com Runge Kutta - Corpo 2 no eixo y

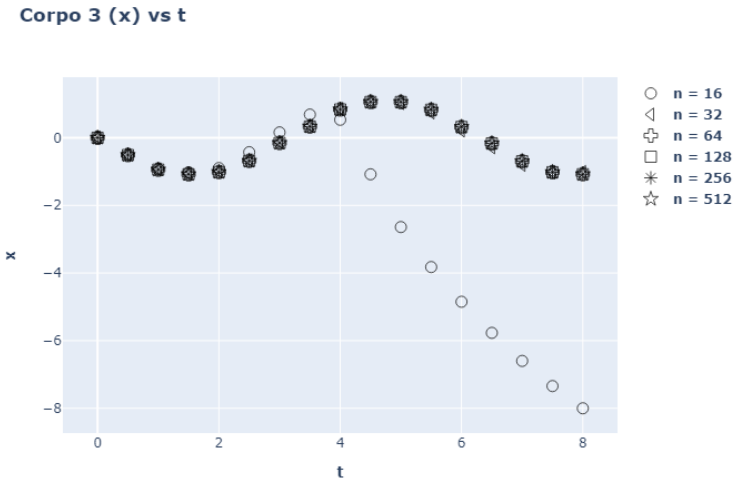

Figura 9: Discretização do TBP com Runge Kutta - Corpo 3 no eixo x

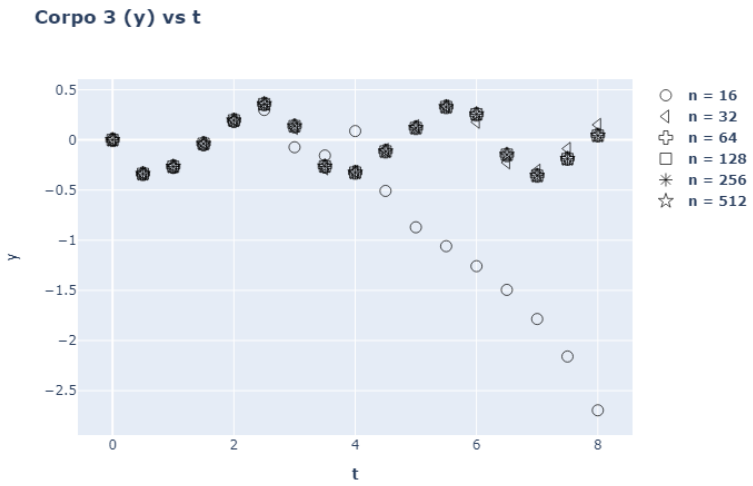

Figura 10: Discretização do TBP com Runge Kutta - Corpo 3 no eixo y

h	$y_1(2h)$	$y_1(h)$	$y_1(h/2)$	ordem $\overline{p}$
3.1250e-02	6.1181e+00	1.8986e-01	3.1083e-01	5.6149e+00
1.5630e-02	1.8986e-01	3.1083e-01	3.1429e-01	5.1272e+00
7.8100e-03	3.1083e-01	3.1429e-01	3.1442e-01	4.7231e+00
3.9100e-03	3.1429e-01	3.1442e-01	3.1442e-01	4.5909e+00

Tabela 5: Convergência com Runge Kutta: Corpo 1 (y)

h	$x_2(2h)$	$x_2(h)$	$x_2(h/2)$	ordem $\overline{p}$
3.1250e-02	-3.4226e+00	-3.4049e-01	-3.5816e-01	7.4468e+00
1.5630e-02	-3.4049e-01	-3.5816e-01	-3.5772e-01	5.3572e+00
7.8100e-03	-3.5816e-01	-3.5772e-01	-3.5770e-01	3.9144e+00
3.9100e-03	-3.5772e-01	-3.5770e-01	-3.5769e-01	4.0507e+00

Tabela 6: Convergência com Runge Kutta: Corpo 2 (x)

Pelas tabelas, tal como esperado, observamos que a ordem de convergência tende a 4 (tal como no problema manufaturado). Entretanto, vale observar que, em algumas tabelas, esse valor até mesmo extrapola a ordem 4, como na tabela 7. Esse resultado pode ter ocorrido devido a natureza caótica e/ou às condições iniciais do problema. Entretanto, dado que o algoritmo

h	$y_2(2h)$	$y_2(h)$	$y_2(h/2)$	ordem $\overline{p}$
3.1250e-02	-8.0006e+00	-1.0098e+00	-1.0761e+00	6.7216e+00
1.5630e-02	-1.0098e+00	-1.0761e+00	-1.0779e+00	5.1922e+00
7.8100e-03	-1.0761e+00	-1.0779e+00	-1.0779e+00	4.7313e+00
3.9100e-03	-1.0779e+00	-1.0779e+00	-1.0779e+00	4.6017e+00

Tabela 7: Convergência com Runge Kutta: Corpo 2 (y)

h	$x_3(2h)$	$x_3(h)$	$x_3(h/2)$	ordem $\overline{p}$
3.1250e-02	1.5988e+01	2.2709e-01	4.3020e-01	6.2779e+00
1.5630e-02	2.2709e-01	4.3020e-01	4.3786e-01	4.7292e+00
7.8100e-03	4.3020e-01	4.3786e-01	4.3816e-01	4.6968e+00
3.9100e-03	4.3786e-01	4.3816e-01	4.3817e-01	4.5792e+00

Tabela 8: Convergência com Runge Kutta: Corpo 3 (x)

h	$y_3(2h)$	$y_3(h)$	$y_3(h/2)$	ordem $\overline{p}$
3.1250e-02	-2.6955e+00	1.5063e-01	4.7330e-02	4.7841e+00
1.5630e-02	1.5063e-01	4.7330e-02	4.3440e-02	4.7301e+00
7.8100e-03	4.7330e-02	4.3440e-02	4.3280e-02	4.6078e+00
3.9100e-03	4.3440e-02	4.3280e-02	4.3270e-02	4.4781e+00

Tabela 9: Convergência com Runge Kutta: Corpo 3 (y)

do Runge Kutta utilizado foi o mesmo tanto para o problema manufaturado quanto para o TBP (alterando-se somente as condições iniciais e a função de discretização), sabemos que esse ruído no valor da ordem $\overline{p}$ não é derivado do método usado. Como forma de reiterar a validade dos programas, manufaturamos outro problema, mas agora de segunda ordem, a fim de garantir que o algoritmo funciona em problemas multidimensionais. Usamos o seguinte problema de Cauchy:

$$\begin{cases} y''(t) = y(t) \\ y(0) = 2 \\ y'(0) = 3 \end{cases}$$

Cuja solução exata é dada por:

$$y(t) = 2,5 \cdot e^{t} - 0,5 \cdot e^{-t}$$

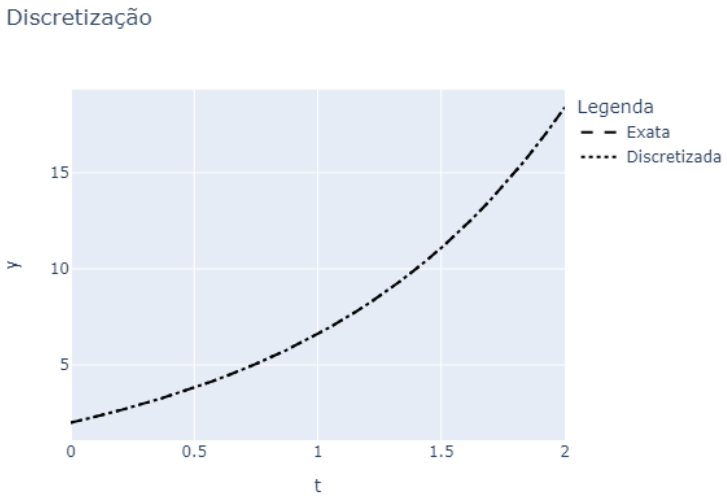

Figura 11: Discretização do problema manufaturado de segunda ordem com Runge Kutta

Assim, podemos discretizá-lo da seguinte forma:

$$\begin{cases} y_1(t) = y(t) \\ y_2(t) = y_1'(t) \\ y_2'(t) = y_1(t) \end{cases}$$

Com isso, obtemos os seguintes resultados:

Pelo gráfico e pela tabela de convergência expostos, é possível garantir, mais uma vez, que o algoritmo do Runge Kutta implementado funciona de forma correta, visto que tanto para problemas unidimensionais quanto multidimensionais - com solução exata e conhecida - o erro decai a 0 com ordem 4 para intervalos de tempo h progressivamente refinados.

Além do método de discretização, é preciso validar tambéma implementação do spline cúbico vinculado. Para isso, utilizaremos duas abordagens: a primeira será comparar os resultados obtidos pelo spline com um *ground true*, ou seja, uma referência científica de implementação correta (a biblioteca SciPy). Já a segunda será verificar se o spline cúbico vinculado, quando aplicado num polinômio cúbico, resulta num erro nulo. Essa segunda verificação

n	h	erro	ordem $\overline{p}$
2.0000e+00	1.0000e+00	-1.3761e-01	0.0000e+00
4.0000e+00	5.0000e-01	-1.2822e-02	3.4239e+00
8.0000e+00	2.5000e-01	-9.8249e-04	3.7060e+00
1.6000e+01	1.2500e-01	-6.8046e-05	3.8519e+00
3.2000e+01	6.2500e-02	-4.4777e-06	3.9257e+00
6.4000e+01	3.1250e-02	-2.8717e-07	3.9628e+00
1.2800e+02	1.5625e-02	-1.8182e-08	3.9814e+00
2.5600e+02	7.8125e-03	-1.1437e-09	3.9907e+00
5.1200e+02	3.9062e-03	-7.1669e-11	3.9963e+00
1.0240e+03	1.9531e-03	-4.4693e-12	4.0032e+00
2.0480e+03	9.7656e-04	-3.0909e-13	3.8540e+00

Tabela 10: Convergência do problema manufaturado de segunda ordem via solução exata

se baseia no fato de que o spline vinculado de terceiro grau se ajusar perfeitamente a quaisquer polinômios com grau menor ou igual a 3. Tomemos, então, o seguinte polinômio:

$$\begin{cases} f(t) = 4t^3 + 2t^2 + 5t + 7 \\ f'(t) = 12t^2 + 4t + 5 \end{cases}$$

Iremos interpolar o polinômio no intervalo $I = [0; 5]$ com $n = 11$ pontos e passo de tamanho $h = \frac{(5-0)}{11-1} = 0,5$ (perceba que, se utilizamos 11 pontos, temos 10 intervalos intermediários). Assim, passamos como parâmetros para a construção dos splines cúbicos o intervalo de tempo I, discretizado com passos de tamanho 0,5, os valores de $f(t)$ calculados no intervalo I e as primeiras derivadas calculadas nos extremos do intervalo (ou seja, $f'(0)$ e $f'(5)$).

A fim de verificar os resultados da construção de spline, basta testá-lo em pontos que não pertencem a I. Para isso, utilizaremos o intervalo $I' = [0; 3]$, com passos de tamanho $h' = 0.33$, garantindo que nenhum ponto testado pertence a I. Após executados o algoritmo e os testes, é possível obter a tabela 11.

Nessa tabela, temos que $f(t)$ é a função do polinômio exato, $S(t)$ é o polinômio interpolado com splines cúbicos, cuja implementação é autoral, e $H(t)$ é o polinômio interpolado com o pacote *SciPy* - referência científica no

que tange a cálculos numéricos e estatísticos na linguagem Python.

Através dos resultados, é possível validar que a implementação dos splines cúbicos foi efetuada de forma correta, tendo em vista que não só a diferença entre $S(t)$ e $H(t)$ é nula num polinômio de grau 3, mas também que $H(t)$ e $S(t)$ coincidem em todos os pontos testados. Dessa forma, temos que os coeficientes do polinômio interpolador autoral e os do pacote *SciPy* são iguais em todos os intervalos testados. Vale ressaltar que o fato de alguns pontos apresentarem erro diferente de 0, mas com ordem de grandeza significativamente pequena (ordem de 10^{-15}), ocorre devido a erros de aproximação, já que a aritmética de ponto flutuante possui precisão finita.

t	$f(t)$	$H(t)$	$S(t)$	$\lvert H(t) - f(t)\rvert$	$\lvert S(t) - f(t)\rvert$
3.300e-01	9.012e+00	9.012e+00	9.012e+00	0.000e+00	0.000e+0
6.600e-01	1.232e+01	1.232e+01	1.232e+01	0.000e+00	0.000e+0
9.900e-01	1.779e+01	1.779e+01	1.779e+01	0.000e+00	0.000e+0
1.320e+00	2.628e+01	2.628e+01	2.628e+01	3.553e-15	3.553e-15
1.650e+00	3.866e+01	3.866e+01	3.866e+01	0.000e+00	0.000e+0
1.980e+00	5.579e+01	5.579e+01	5.579e+01	7.105e-15	7.105e-15
2.310e+00	7.853e+01	7.853e+01	7.853e+01	1.421e-14	1.421e-14
2.640e+00	1.077e+02	1.077e+02	1.077e+02	0.000e+00	0.000e+0
2.970e+00	1.443e+02	1.443e+02	1.443e+02	0.000e+00	0.000e+0

Tabela 11: Erros do polinômio interpolado em $f(t)$ com splines cúbicos no intervalo I'

Validada a implementação dos splines cúbicos, foi possível interpolar a rota percorrida pelo corpo 1 e gerar gráficos em pontos que não foram utilizados no processo de contrução do spline. Observe as trajetórias em função do tempo nas figuras 12 e 13.

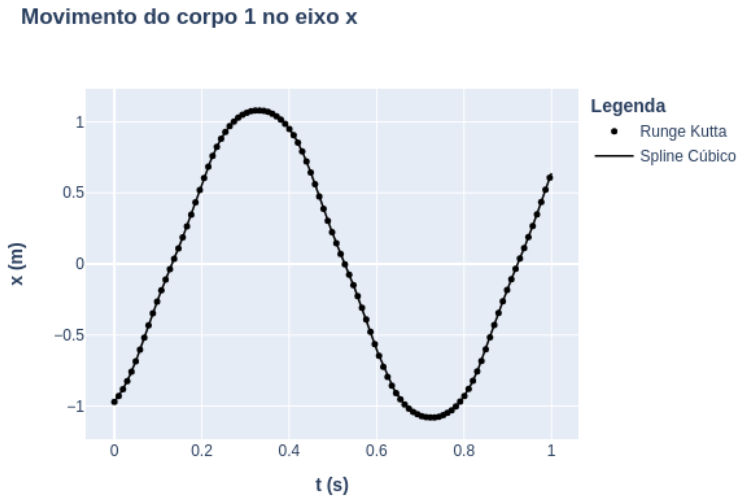

Figura 12: Aproximação da trajetória do corpo 1 no eixo x com splines cúbicos vinculados

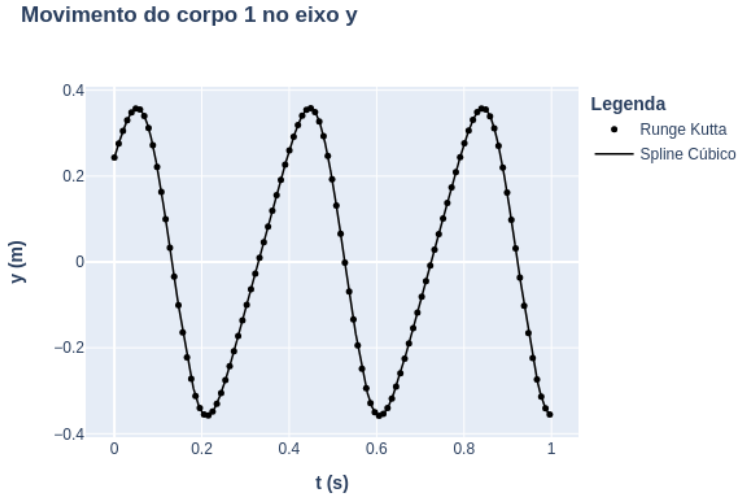

Figura 13: Aproximação da trajetória do corpo 1 no eixo y com splines cúbicos vinculados

5 Conclusão

Em conclusão, o trabalho realizado para a análise do Problema dos Três Corpos, com métodos numéricos, mostrou-se proveitoso tanto para o desenvolvimento das habilidades computacionais dos membros da equipe, quanto para o aprendizado na disciplina no geral.

Com esse trabalho, elaborou-se um código em Python que aproxima, com base nas leis de Newton, a trajetória de 3 corpos pontuais de massas, posições e velocidades iniciais arbitrárias. Esse problema foi modelado com equações diferenciais de segunda ordem de forma multidimensional (2 eixos de movimento e 3 corpos distintos). Para discretizar o problema, foi utilizado o método de Runge-Kutta clássico – com ordem de convergência 4 – e os pontos obtidos na discretização foram modelados com polinômios de grau 3 via Splines Cúbicos Vinculados. Com tal implementação, foi possível gerar um acervo de códigos para simulação de diversas trajetórias de quaisquer três corpos em um espaço tridimensional ou até mesmo analisar outros problemas que utilizam EDO's pelo método de Runge-Kutta.

Em relação aos resultados obtidos, julgou-se como completa a análise e validação de todos os métodos aplicados, desde a classe *Planeta.py*, até a verificação dos resultados dos *Splines*. Também cabe ressaltar que foi satisfatória a escolha das condições iniciais para a observação de uma trajetória cíclica e, com licensa poética, bela.

Finalmente, o grupo agradece a toda a equipe de MAP3122 pelo desenvolvimento e ensino sobre métodos numéricos, especialmente ao Prof. Alexandre Roma, que se mostrou empático e compreensivo para com os alunos. O grupo também agradece à equipe de monitores pelo trabalho feito e espera que possa haver outras oportunidades de realizar trabalhos como este que, apesar de difícil, trouxe grandes aprendizados.

---

6 Apêndices e Bibliografia

6.1 Apêndice: Teoria do Caos e o TBP

A Teoria do Caos e o Problema dos Três Corpos (TBP) são conceitos fascinantes que se situam na interseção da física, da matemática e da filosofia, oferecendo perspectivas únicas sobre a natureza imprevisível do universo. Enquanto a Teoria do Caos investiga como pequenas variações nas condições iniciais de um sistema dinâmico podem levar a diferenças enormemente divergentes em seu comportamento futuro, o Problema dos Três Corpos lida com a complexidade de prever o movimento de três objetos celestes sob influência gravitacional mútua, sem soluções exatas para a maioria dos casos.

Este último é um exemplo clássico de um sistema caótico. Originalmente formulado por Isaac Newton no século XVII, o problema desafiou matemáticos e físicos por séculos. A dificuldade reside na interação não-linear entre os corpos, que pode resultar em trajetórias extremamente sensíveis às condições iniciais. Mesmo a menor das mudanças pode levar a resultados drasticamente diferentes, um princípio central na Teoria do Caos conhecido como o efeito borboleta.

Por outro lado, as equações da entropia se conectam profundamente com estes conceitos, fornecendo uma medida da desordem ou da incerteza dentro de um sistema. A entropia é um pilar da segunda lei da termodinâmica, que afirma que, em um sistema isolado, a desordem tende a aumentar com o tempo. Isso não apenas ressoa com a natureza imprevisível dos sistemas caóticos, mas também destaca um fluxo unidirecional do tempo, do ordenado para o desordenado.

A interação entre essas áreas — caos, movimentos celestes, e entropia — ilustra a complexidade inerente ao universo. Ela nos lembra que, apesar dos avanços na ciência e na matemática, ainda existem limites para o nosso entendimento e previsão dos fenômenos naturais. No entanto, é precisamente essa complexidade que incita a curiosidade e impulsiona a busca contínua por conhecimento, desdobrando-se em uma jornada interminável de descoberta.

Por fim, tendo em vista que o objetivo desse projeto de Métodos Numéricos não busca atingir conhecimentos profundos sobre entropia e de caos, conclui-se que apesar de fascinante, um estudo aprofundado sobre o que é caos e os seus desdobramentos no *TBP* são motivações de outras disciplinas.

6.2 Bibliografia

A bibliografia utilizada foi:

- UNESP: História de Newton
- UNESP: História de Kepler
- Wikipedia: Three-Body Problem
- Wikipedia: Poincaré and the Three-Body Problem
- Material da disciplina: Three-Body Problem: A Precise Simulation
- Stephen Wolfram, A New Kind of Science – Notes for Chapter 7: Mechanisms in Programs and Nature; Section: Chaos Theory and Randomness from Initial Conditions
- Artigos Harvard: The Complete Solution of a General Problem of Three Bodies; Victor Szebehely and C. Frederick Peters
- Stanford Encyclopedia of Philosophy – Chaos, 2008
- Oliveira, V., Cruz, I. – O Problema dos Três Corpos
- Método de Runge-Kutta de Discretização
- Interpolação polinomial: Polinômio de Lagrange
- Interpolação Polinomial: Splines Cúbicos Análise Numérica, BURDEN et al.