Implementação de método computacional para resolução de problemas físicos

Luís Sousa
Faculdade de Ciências Exatas e da Engenharia, Universidade da Madeira

I. Introdução

A capacidade de resolver problemas físicos de forma computacional têm muitos benefícios, pois podendo prever o comportamento de problemas físicos permitira a realização de menos experiências que podem ter custos elevados e tempo de preparação muito elevados, podendo estimar qual serão os parâmetros que irão levar aos resultados ótimos podemos assim realizar apenas uma experiência para comprovar se os resultados são os esperados. Muitos dos problemas físicos podem ser representados por *ODEs* pois representam sistemas que relacionam vários pontos adjacentes. Estes problemas podem vir sobre duas formas BVPs (problemas com condições de fronteira) e IVPs (problemas com condições iniciais). Por estes tipos de problemas serem tão úteis foi pedido pela capacidade de resolver *BVPs* de segundo grau e resolver *BVPs* de sistemas de *ODEs* de primeiro grau.

A. Descrição do problema

BVPs (*Boundary Value Problem*) são problemas de equações diferenciais onde a equação diferencial é restringida por condições de fronteira mas nunca tendo toda a informação nos pontos iniciais, daí diferenciando-se de um problema de IVP (*Initial Value Problem*) pois num problema de IVP o problema é definida no ponto inicial permitindo resolve-la com uma evolução para o futuro da equação através da utilização de qualquer método que utiliza a equação diferencial e os pontos anteriores para obter uma novo ponto repetindo assim este passo até chegar ao fim do problema. Para BVPs não temos informação suficiente para iterar para um novo ponto somente com a informação anterior por isso temos que utilizar qualquer método que com base nas condições de fronteira consiga melhorar o resultado ate obter uma solução perto o suficiente da real.

Este tipo de problema é muito comum em problemas físicos ou problemas baseados na realidade (como por exemplo a *Heat Equation* que é um problema de uma vara onde sabemos a temperatura nas duas extremidades e sabemos a propriedade do material assim podendo calcular a temperatura no meio vara) pois muitas vezes não conseguimos reduzir todos os problemas BVPs em IVPs porque não conseguimos obter todos os parâmetros iniciais de um sistema mas com informação inicial e final do sistema podemos após resolver a BVP saber qual eram todos os parâmetros iniciais.

B. Acknowledgement

Este trabalho foi desenvolvido no âmbito do projeto de investigação "StickeR: Técnica inovadora de reforço estrutural baseada em laminados CFRP de requisitos multifuncionais e aplicados com avançado adesivo de matriz cimentícia", com a referencia POCI-01-0247-FEDER-039755, apoiado pelo Fundo Europeu de Desenvolvimento Regional (FEDER) através do Programa Operacional Fatores de Competitividade e internacionalização (POCI) e por fundos nacionais através da FCT – Fundação Portuguesa de Ciência e Tecnologia.

II. Solução

Para resolver problemas de fronteira BVPs existem vários algoritmos que podem ser utilizados, entre os que foram pedidos para serem utilizados foi o *Shooting method*[1] e o *Finite Difference Method*[2].

O *Shooting method* consiste em repetidamente resolver problemas de IVP em que o valor inicial é uma estimativa e de acordo com os resultados nos valores de fronteira ajustar e resolver um novo IVP com uma nova estimativa inicial, até conseguir cumprir as regras de fronteira do BVP.

O *Finite difference method* consiste em dividir o problema em vários pontos e fazer uma aproximação para cada ponto tipicamente existem três tipos de aproximação *forward difference*, que permite obter o valor posterior a partir do anterior $(x_0, y_0) \rightarrow (x_1, y_1)$, o *backward difference*, que permite obter o valor anterior a partir do posterior $(x_0, y_0) \rightarrow (x_{-1}, y_{-1})$, o terceiro tipo de aproximação é *central difference* que consiste em obter um valor a partir do seu posterior e anterior $((x_{-1}, y_{-1}), (x_1, y_1)) \rightarrow (x_0, y_0)$, com estas aproximações é possível obter aproximações para todos os pontos necessários e obter um resultado.

III. Revisão de Literatura

A pesquisa foi realizada pela principalmente pela *internet* onde foram efetuadas pesquisas sobre os tópicos e áreas por volta deste tópico, uma das ferramentas mais úteis foi a *Wikipedia* por ser um aglomerado de informação e conter as referencias para artigos e livros onde o conteúdo pode ser mais aprofundado. Outra ferramenta muito utilizada foram notas de disciplinas como a de *Lamar University* sobre os problemas de condições de fronteira[3].

A. ODEs

Equações Ordinárias Diferenciais (*Ordinary Differential Equations*) são funções diferenciais[4] (funções que relacionam uma função as suas derivadas) que contém apenas uma variável independente.[5] Estas equações podem ser representadas de forma explicita $F(x, y, y', ..., y^{n-1}) = y^n$ ou forma implícita $F(x, y, y', y'', ..., y^n) = 0$ estas equações também podem vir em sistemas de equações ordinárias diferenciais da forma explicita (1) ou da forma implícita (2)

$$\begin{pmatrix} y_1^{(n)} \\ y_2^{(n)} \\ \vdots \\ y_m^{(n)} \end{pmatrix} = \begin{pmatrix} f_1(x, y, y', y'', ..., y^{(n-1)}) \\ f_2(x, y, y', y'', ..., y^{(n-1)}) \\ \vdots \\ f_m(x, y, y', y'', ..., y^{(n-1)}) \end{pmatrix} \tag{1}$$

$$\begin{pmatrix} f_1(x, y, y', y'', ..., y^{(n)}) \\ f_2(x, y, y', y'', ..., y^{(n)}) \\ \vdots \\ f_m(x, y, y', y'', ..., y^{(n)}) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{2}$$

Este tipo de equações são muito úteis para modelar problemas reais desde propagação de doenças [6], trajetória balística [7], até estimar a população de predadores e de presas[8].

Muitas vezes estes problemas não podem ser resolvidos analiticamente ou a sua resolução é demasiado complexa, para poder resolver este tipo de problema podemos tomar uma solução numérica para o problema. Quando resolvemos ODEs de forma numérica podemos dividir em dois tipos, os problemas de valor inicial (*Initial Value Problem*) ou problemas de valor de fronteira (*Boundary Value Problem*).

B. IVP

Initial Value Problems consistem em problemas cujo os valores estão apenas definidos no inicio do problema, o valor inicial pode ser definido como $y(x_0) = \alpha$. Para resolver problemas IVP podemos repetidamente resolver a seguinte equação $y_{n+1} = y_n + g(x, y', ..., y^n)$ para cada ponto após o valor inicial até o ponto que decidimos ser o final. O método mais simples é o método de euler que consiste em utilizar apenas uma derivada para estimar o próximo ponto da equação com um *step* h:

$$x_0 = \alpha, y_0 = \beta$$

$$(x_n, y_n) \to (x_{n+1}, y_{n+1})$$

$$y_{n+1} = y_n + hy'(x_n)$$

$$x_{n+1} = x_n + h$$

Os algoritmos mais complexos para resolver IVPs consistem em fazer várias estimativas das derivadas a pontos intermédios entre o ponto atual $y(x_n)$ e o ponto seguinte $y(x_{n+1})$ estes métodos são tipicamente chamados métodos de *Runge-Kutta* estes métodos seguem o formato geral de[9]:

$$y_{n+1} = y_n + h\sum_{i=1}^{s} b_i k_i$$

$$k_i = f(t_n + c_i h, y_n + h\sigma_{j=1}^{s} a_{ij} k_j), i = 1, ..., s.$$

os valores de c_i e a_{ij} seguem o formato de uma *Butcher tableu* do seguinte formato:

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & ... & a_{1s} \\ c_2 & a_{21} & a_{22} & ... & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & ... & a_{ss} \\ \hline & b_1 & b_2 & ... & b_2 \end{array}$$

O método mais comum de *Runge-Kutta* é o RK4 ou *classic Runge-Kutta* cuja a tabela de *Butcher* é a seguinte[10]:

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

a função RK4 fica como:

$$k_1 = f(x_n, y_n)$$
$$k_2 = f(x_n + \frac{h}{2}, y_n + h\frac{k_1}{2})$$
$$k_3 = f(x_n + \frac{h}{2}, y_n + h\frac{k_2}{2})$$
$$k_4 = f(x_n + h, y_n + hk_3)$$
$$y_{n+1} = y_n + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4)$$
$$x_{n+1} = x_n + h$$

estes métodos conseguem resolver problemas de primeira ordem, para conseguir resolver problemas de ordem mais alta podemos utilizar métodos especificamente feitos para a ordem necessária ou transformar os problemas de ordens superiores em sistema de primeira ordem e resolver repetidamente.

Um exemplo de um método para resolver ordens superiores é o *Runge-Kutta-Nyström* que funciona na seguinte forma[11]:

$$k_1 = \frac{1}{2}hf(x_{n-1}, y_{n-1}, y'_{n-1})$$
$$K = \frac{1}{2}h(y'_{n-1} + \frac{1}{2}k_1)$$
$$k_2 = \frac{1}{2}hf(x_{n-1} + \frac{1}{2}h, y_{n-1} + K, y'_{n-1} + k_1)$$
$$k_3 = \frac{1}{2}hf(x_{n-1} + \frac{1}{2}h, y_{n-1} + K, y'_{n-1} + k_2)$$
$$L = h(y'_{n-1} + \frac{1}{2}k_3)$$
$$k_4 = \frac{1}{2}hf(x_{n-1} + h, y_{n-1} + L, y'_{n-1} + 2k_3)$$
$$y_n = y_{n-1} + h(y'_{n-1} + \frac{1}{3}(k_1 + k_2 + k_3))$$
$$y'_n = y'_{n-1} + \frac{1}{3}(k_1 + 2k_2 + 2k_3 + k_4)$$

Para resolver outras ordens ou criar um método que consiga resolver um problema com ordem arbitrária temos a opção de utilizar um dos métodos de primeira ordem e transformá-lo de modo a resolver as ordens repetidamente intercaladamente[12]. Para resolver problemas de ordem superior pode-se transformar o RK4 para a seguinte forma:

$$k_{1t} = f_n(x_{n-1}, y^{(1)}_{n-1}, y^{(2)}_{n-1}, ..., y^{(t)}_{n-1}) \text{ em } t = 1, 2, 3, ..., order$$
$$k_{2t} = f_n(x_{n-1} + \frac{h}{2}, y^{(1)}_{n-1} + h\frac{k_{11}}{2}, y^{(2)}_{n-1} + h\frac{k_{12}}{2}, ..., y^{(t)}_{n-1} + h\frac{k_{1t}}{2}) \text{ em } t = 1, 2, 3, ..., order$$
$$k_{3t} = f_n(x_{n-1} + \frac{h}{2}, y^{(1)}_{n-1} + h\frac{k_{21}}{2}, y^{(2)}_{n-1} + h\frac{k_{22}}{2}, ..., y^{(t)}_{n-1} + h\frac{k_{2t}}{2}) \text{ em } t = 1, 2, 3, ..., order$$
$$k_{4t} = f_n(x_{n-1} + h, y^{(1)}_{n-1} + hk_{31}, y^{(2)}_{n-1} + hk_{32}, ..., y^{(t)}_{n-1} + hk_{3t}) \text{ em } t = 1, 2, 3, ..., order$$
$$y^{(t)}_{n+1} = y^{(t)}_n + \frac{1}{6}h(k_{1t} + 2k_{2t} + 2k_{3t} + k_{4t}) \text{ em } t = 1, 2, 3, ..., order$$
$$x_{n+1} = x_n + h$$

C. BVP

Boundary Value Problems são problemas que são apenas definidas por condições de fronteira onde nem todos os valores iniciais estão definidos, as condições de fronteira para por exemplo um problema de segundo grau podem ser descritas da seguinte forma[3]:

$$y(x_0) = y_0,\ y(x_1) = y_1$$
$$y'(x_0) = y_0,\ y'(x_1) = y_1$$
$$y'(x_0) = y_0,\ y(x_1) = y_1$$
$$y(x_0) = y_0,\ y'(x_1) = y_1$$

BVPs são tipicamente resolvidos por dois métodos, *Shooting Method* ou *Finite Difference Method*

1) Shooting Method:

O *Shooting Method* consiste em obter uma estimativa para os valores iniciais que não sabemos. Deste modo podemos resolver o problema como um *Initial Value Problem*, comparar o resultado com as condições de fronteira e arranjar uma estimativa melhor para os valores que não sabemos. Se tivermos um bom método para obter a nova estimativa, podemos continuamente aproximar as estimativas até as condições de fronteira sejam cumpridas[1].

Para aproximarmos em cada iteração para um melhor valor podemos mudar as condições de fronteira para uma forma explicita em que temos por exemplo $y(x_0) - y_{0guess} = 0$ se resolvermos esta equação teremos obtido o valor que cumpre aquela condição de fronteira.

Um dos métodos que pode ser utilizado para resolver a equação $y(x_0) - y_{0guess} = 0$ é o *fixed point iteration method* que consiste em uma função que podemos iterar para obter uma sequência de valores que devem convergir no ponto entendido[13].

$$x_{n+1} = f(x_n),\ n = 0, 1, 2, ...$$
$$|x_n - c| > |x_{n+1} - c|$$

A maneira mais fácil de implementar este método para este problema é reescrevendo a condição de fronteira para $y(x_0) - y_{0guess} + t_n = t_{n+1}$ deste modo à medida que t_n aproxima-se do valor real $y(x_0) - y_{0guess}$ vai aproximando-se de 0, assim podemos em cada iteração aproximar o valor real de y_{0guess}. Apesar deste método funcionar em muitos casos, é preciso uma estimativa inicial perto do valor real e em certas situações em que $y(x)$ varia demasiado o problema pode não convergir.

2) Finite Difference Method:

O *Finite Difference Method* consiste em subdividir o problema em vários pontos e usar uma aproximação da derivada para cada ponto utilizando diferenças finitas que relacionam-se com os pontos adjacentes. A forma mais comum de utilizar este método é pela substituição das derivadas por uma aproximação criada a partir da série de *Taylor*, deste modo tipicamente referenciamos três possíveis aproximações para o primeiro grau[14]:

$$\frac{\partial F}{\partial x} = \frac{F(x + \Delta x) - F(x)}{\Delta x},\ \textit{Forward Difference}$$
$$\frac{\partial F}{\partial x} = \frac{F(x) - F(x - \Delta x)}{\Delta x},\ \textit{Backawards Difference}$$
$$\frac{\partial F}{\partial x} = \frac{F(x + \Delta x) - F(x - \Delta x)}{2\Delta x},\ \textit{Centered Difference}$$

para resolvermos por exemplo o seguinte problema $\frac{d^2F}{dx^2} + h'(F_a - T) = 0,\ F(0) = F_0\ T(L) = F_L$ (sendo isto um *Heat Equation problem*[15]), para este problema necessitamos de uma aproximação de segundo grau que utilizaremos a seguinte:

$$\frac{\partial^2 F}{\partial x^2} = \frac{F(x + \Delta x) - 2F(x) + F(x - \Delta x)}{\Delta x^2} = \frac{F_{i+1} - 2F_i + F_{i-1}}{\Delta x^2}$$

utilizando esta aproximação podemos transformar a equação em:

$$\frac{F_{i+1} - 2T_i - T_{i-1}}{\Delta x^2} + h'(F_a - F_i) = 0 \Leftrightarrow -F_{i-1} + (2 + h'\Delta x^2)F_i - F_{i+1} = h'\Delta x^2 F_a$$

sendo $h'\Delta x^2 F_a$ constante podemos transformar este problema num sistema de equações lineares:

$$\begin{cases} -F_0 + (2 + h'\Delta x^2)F_1 - F_2 = h'\Delta x^2 F_a \\ -F_1 + (2 + h'\Delta x^2)F_2 - F_3 = h'\Delta x^2 F_a \\ \qquad\qquad ... \\ -F_{n-2} + (2 + h'\Delta x^2)F_{n-1} - F_n = h'\Delta x^2 F_a \end{cases}$$

este sistema de equações pode ser resolvido com qualquer método para resolver sistemas de equações lineares.

Outro método para usar o *Finite Difference Method* é utilizando o *Simpson Method* que consiste num *Collocation Method* que é equivalente a *3-stage Lobatto IIIa implicit Runge-Kutta*[16] que pode ser representado da seguinte forma[17]:

$$y_{i+1} = y_i + \frac{h_i}{6}(f(x_i, y_i) + f(x_{i+1}, y_{i+1})) + \frac{3h_i}{3} f(x_i + \frac{h_i}{2}, \frac{y_i + y_{i+1}}{2} - \frac{h_i}{8}(f(x_{i+1}, y_{i+1}) - f(x_i, y_i)))$$

após utilizar esta fórmula para cada ponto do sistema podemos avaliar os erros residuais para saber se o número de pontos utilizados são suficiente para formar uma boa estimativa, de acordo com essa avaliação podemos dinamicamente acrescentar mais pontos nos intervalos de pontos já existentes, de modo a melhorar o erro nesse intervalo.

D. Tecnologias Utilizadas

Para a realização deste projeto foi utilizado o C como linguagem de programação, todas as bibliotecas utilizadas fazem parte da biblioteca padrão do C. Para uma mais fácil intercompatibilidade entre sistemas operativos foi utilizado *CMAKE* que possibilita a simplificação de mudar o IDE utilizado para o projeto.

Para a compilação foi utilizado o *GCC*[18] em *WSL2*[19] que permite executar código para *Linux* em *Windows*.

Para a testagem foi utilizada a biblioteca *SciPy*[20] para comparar os resultados obtidos no código criado com uma implementação já reconhecida.

E. Revisão

Percebendo a importância das *ODEs* para modelarem problemas reais conseguimos encontrar vários métodos para as resolver. Se tivermos perante um problema onde sabemos as condições iniciais podemos utilizar métodos de resolução de *IVPs* como o método de *Euler* ou o método de *Runge-Kütta* que apesar de serem métodos de primeiro grau podem ser transformados para serem utilizados em graus superiores. Quando estamos perante um problema com condições de fronteira podemos utilizar métodos de resolução de *BVPs*, existem dois métodos predominantes, O *Shooting Method* que funciona por utilizar os métodos para resolver *IVPs* utilizando uma estimativa para o valor inicial e melhorando essa estimativa, o outro método para resolver *BVPs* é o *Finite Difference Method* que consiste em dividir o problema em vários pontos e resolver o sistema de equações desses pontos para obter o valor para esses pontos.

IV. Implementação

A. Shooting Method

No inicio da implementação para a melhor compreensão do problema comecei por implementar um método de resolução de IVP, para isso implementei três métodos de resolução de IVPs, começando pelo método mais simples de resolver IVPs que é o método de euler.

O método de euler consiste em incrementar o valor atual pela sua derivada nesse ponto, utilizámos um passo constante como o intervalo entre dois pontos.

$$y_0 = \alpha$$

$$y_{n+1} = y_n + hy'_n$$

sendo que a implementação deste algoritmo em código parecera-se um pouco com este pseudocódigo:

Algorithm 1: Euler method

$x = x_0$;
$y = y_0$;
while $x < x_n$ **do**
    $y = y + hy'(x, y)$;
    $x = x + h$;
end

após a implementação deste algoritmo já se consegue resolver alguns problemas IVPs, com os resultados desses testes conseguimos ver que este método é muito simples e que por causa disso chegamos a uma aproximação da função real um pouco afastada do resultado previsto.

Para obter melhores resultados podemos utilizar algum algoritmo mais complexo como é o exemplo do método Runge-Kutta que funciona utilizando mais aproximações do próximo ponto utilizando mais declives. Isto pode ser denotado da seguinte forma:

$$y_{n+1} = y_n + h\sum_{i=1}^{s}(b_i k_i)$$

em que b_i são os pesos de cada novo declive k_i. O primeiro método *Runge-Kütta* implementado foi o RK4 que consiste do seguinte código:

Algorithm 2: classic Runge-Kutta method (RK4)

$x = x_0$;
$y = y_0$;
while $x < x_n$ **do**
  $k_1 = f(x, y)$;
  $k_2 = f(x + \frac{h}{2}, y_n + h\frac{k_1}{2})$;
  $k_3 = f(x + \frac{h}{2}, y_n + h\frac{k_2}{2})$;
  $k_4 = f(x + h, y + hk_3)$;
  $y_{n+1} = y_n + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4)$;
  $x = x + h$;
end

Estes dois métodos têm a limitação de só resolverem IVPs de primeira ordem, para estender para ordens superiores temos duas maneiras, ou transformar os problemas em problemas de primeira ordem ou implementar algoritmos e métodos que consigam resolver em ordens superiores. Para uma melhor compreensão dos métodos foi implementada um algoritmo que resolve IVPs de segunda ordem *Runge-Kutta-Nyström* que é baseado no *Runge-Kutta* normal otimizado para problemas de segunda ordem, o algoritmo é o seguinte:

Algorithm 3: Runge-Kutta-Nyström (RKN)

$x = x_0$;
$y = y_0$;
$y' = y'_0$;
while $x < x_n$ **do**
  $k_1 = \frac{1}{2}hf(x, y, y')$;
  $K = \frac{1}{2}h(y' + \frac{1}{2}k_1)$;
  $k_2 = \frac{1}{2}hf(x + \frac{1}{2}h, y + K, y' + k_1)$;
  $k_3 = \frac{1}{2}hf(x + \frac{1}{2}h, y + K, y' + k_2)$;
  $L = h(y' + \frac{1}{2}k_3)$;
  $k_4 = \frac{1}{2}hf(x + h, y + L, y' + 2k_3)$;
  $y = y + h(y' + \frac{1}{3}(k_1 + k_2 + k_3))$;
  $y' = y' + \frac{1}{3}(k_1 + 2k_2 + 2k_3 + k_4)$;
  $x = x + h$;
end

Após a implementação destes dois algoritmos já temos uma boa compreensão de como resolvemos problemas de IVPs, já podemos começar a resolver BVPs utilizando o *Shooting method* que consiste em utilizar uma estimativa inicial, para resolver o problema como um IVP e de acordo com o resultado obtido melhorar a estimativa inicial, para isso podemos utilizar interpolação linear:

$$y_{newguess} = y_{min} + \frac{y'_{max} - y'_{min}}{(r_{max} - r_{min})(y_n - r_{min})}$$

Nesta altura foram implementados dois algoritmos, um algoritmo que apenas faz uma iteração e outro que tenta repetidamente chegar mais perto do resultado correto.

Para ambos os algoritmos é necessário um intervalo y'_{min}, y'_{max} para criar uma nova estimativa dentro desse intervalo, no algoritmo que repetidamente tenta aproximar do resultado correto é necessário uma condição de iterações máxima no caso do resultado não estiver a aproximar do real.

Algorithm 4: Shooting Method (Runge-Kutta-Nyström)

$x = x_0$;
$y_0 = y_0$;
$y_n = y_n$;
$y'_{min} = y'_{0min}$;
$y'_{max} = y'_{0max}$;
$r_{min} = rkn(x, y, y_{min})$;
$r_{max} = rkn(x, y, y_{max})$;
$y'_{guess} = y_{min} + (y'_{max} - y'_{min})/((r_{max} - r_{min}) * (y_n - r_{min}))$;
$y_{finalguess} = rkn(x, y, y_{guess})$;

algoritmo que aproxima-se ao resultado real:

Algorithm 5: Shooting Method (Runge-Kutta-Nyström) repetitivo

$x = x_0$;
$y_0 = y_0$;
$y_n = y_n$;
$i = 0$;
$y'_{min} = y'_{0min}$;
$y'_{max} = y'_{0max}$;
$r_{min} = rkn(x, y, y_{min})$;
$r_{max} = rkn(x, y, y_{max})$;
do
  $y'_{guess} = y_{min} + (y'_{max} - y'_{min})/(r_{max} - r_{min} * (y_n - r_{min}))$;
  $y_{newguess} = rkn(x, y, y_{guess})$;
  if $y_{newguess} < y_{end}$ **then**
    $y_{min} = y'_{guess}$;
    $r_{min} = y'_{newguess}$;
  else
    $y_{max} = y'_{guess}$;
    $r_{max} = y'_{newguess}$;
  $i = i + 1$;
while $i < limit \wedge |y_{newguess} - y_{end}| >= accuracy$;

Este algoritmo consegue resolver BVPs de segundo grau mas apenas para problemas $y(x_0) = y_0\ y(x_n) = y_n$ para resolver outros problemas de segundo grau teríamos que diretamente alterar o código. Só resolver problemas de segundo grau também é uma grande limitação, pois muitas vezes temos que resolver problemas de grau superior e a capacidade de resolver problemas de n grau seria útil.

Para resolver problemas de n grau com *Shooting Method* teríamos que implementar um algoritmo que resolvesse IVP para qualquer grau, para isso podemos fazer alterações na implementação do RK4 de modo a resolver repetidamente o método para o grau superior.

Algorithm 6: Runge-Kutta (RK4) n^{th} order

$x = x_0$;
$y = y_0$;
while $x < x_n$ **do**
  for $i \leftarrow 0$ **to** $order$ **by** 1 **do**
    $k_1[i] = f[i](x, y_n[i])h$;
  end
  for $i \leftarrow 0$ **to** $order$ **by** 1 **do**
    $k_2[i] = f[i](x + \frac{h}{2}, y_n[i] + h\frac{k_1[i]}{2})$
  end
  for $i \leftarrow 0$ **to** $order$ **by** 1 **do**
    $k_3[i] = f[i](x + \frac{h}{2}, y_n[i] + h\frac{k_2[i]}{2})$
  end
  for $i \leftarrow 0$ **to** $order$ **by** 1 **do**
    $k_4[i] = f[i](x + h, y_n[i] + hk_3[i])$
  end
  for $i \leftarrow 0$ **to** $order$ **by** 1 **do**
    $y_{n+1}[i] = y_n[i] + \frac{1}{6}h(k_1[i] + 2k_2[i] + 2k_3[i] + k_4[i])$;
  end
  $x = x + h$;
end

Após termos implementado um novo algoritmo para resolver IVPs será necessário investigar em como o vamos utilizar para resolver BVPs. Quando estávamos a resolver problemas de primeiro ou segundo grau conseguíamos utilizar métodos de interpolação linear facilmente pois tínhamos apenas uma incógnita, se quisermos que o nosso novo método seja capaz de resolver problemas com múltiplas incógnitas teremos que utilizar um outro método, neste caso foi decidido utilizar o método de *fixed-point iteration* $y_n = f(y_{n-1})$ que é modificado para $y_{newGuess} = y_{oldGuess} + r_{erro} * a_{attenuation}$ isto permite aproximar a estimativa ao valor real em cada iteração. Este método é muito limitado pois a estimativa tem que ser perto do valor real.

Algorithm 7: Shooting Method with RK4 n^{th} order

$y_{guesses} = guesses$;
$conditionsMet = True$;
$iter = 0$;
do
  $vals = y_{guesses} \wedge y_{vals}$;
  $result = RK4(vals)$;
  $used = False$; **for** $i \leftarrow 0$ **to** $order$ **by** 1 **do**
    if $\exists condition$ **then**
      for $t \leftarrow 0$ **to** $order$ **by** 1 **do**
        if $\exists guess \wedge used[t]$ **then**
          $y_{guesses}[t] = condition[i](result) * attenuation + y_{guesses}[t]$;
          $used[t] = True$;
      end
      if $|condition[i](result)| > tolerance$ **then**
        $conditionsMet = False$;
  end
while $!conditionsMet \wedge iter < MaxIterations$;

Este algoritmo consegue resolver BVPs de n ordem, apesar disso este algoritmo é muito limitado por vários fatores, o facto da estimativa ter que ser muito perto do valor real, se o intervalo não for pequeno o suficiente para funções que variam com uma frequência alta o problema pode não ser possível resolver e usando um intervalo muito pequeno aumentara o tempo para resolver o problema.

B. Finite Difference Method

Na implementação do algoritmo para resolver o método de diferenças finitas a maioria da literatura refere métodos de substituição, este tipo não poderia ser utilizado para resolver o problema pedido pela necessidade de ser capaz de resolver sistemas de ODEs de primeiro grau. Pelas restrições de tempo a necessidade de encontrar uma boa solução era imperativa, para isso foi seguida a implementação do *solve_bvp* da biblioteca *SciPy* do python[21]. Antes de podermos começar a implementação do método próprio necessitávamos de implementar várias funções para operações em matrizes, que algumas tiveram que ser implementadas para as duas estruturas de matrizes, uma para matrizes de duas dimensões e outra para três dimensões, também foi criadas estruturas para simplificar a utilização de *arrays* em C. A maioria destas funções existe na biblioteca de python *NumPy* que foi utilizada para testar e verificar o resultado das funções criadas:

- Repeat
- Unravel
- FindMax
- Reshape
- Linspace
- Diff
- Tile
- Arange
- Mul
- Add
- Sub
- Mul (Escalar)
- Dot
- Abs
- Sort
- StackedMul
- Stack
- Identity
- Transpose

Para além da implementação destas operações sobre matrizes foi necessário implementar um método para resolver sistemas de equações lineares, para isso foi implementado decomposição de LU (*Lower Upper*) que consta em decompor a matriz do sistema de equações no seguinte formato:

$$A = LU$$

sendo um exemplo desta decomposição a seguinte:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \tag{3}$$

a decomposição pode ser feita utilizando pivotamento do seguinte formato $PA = LU$[22] que simplifica a computação, se quisermos resolver uma equação linear no formato $Ax = b$ observamos que é equivalente a $PA = LU \Leftrightarrow LUx = Pb$ e como

as matrizes L e U são triangulares podem ser resolvidas por substituição. Após várias tentativas de implementar decomposição com pivotamento completo, foi implementado o seguinte algoritmo[23]:

Algorithm 8: LUP Decomposition

```
Input: A = matriz, N = Tamanho da matriz, Tol = Tolerância
Output: P = Matriz de permutação
for i ← 0 to N by 1 do
    P[i] = i;
end
for i ← 0 to N by 1 do
    maxA = 0;
    imax = i;
    for k ← i to N by 1 do
        if |A[k][i]| > maxA then
            maxA = |A[k][i]|;
            imax = k;
    end
    if imax ≠ i then
        swap(P[i], P[imax]);
        swap(A[i], A[imax]);
        P[N] = P[N] + 1;
    for j = i + 1 to N by 1 do
        A[j][i] = A[j][i]/A[i][i];
        for k = i + 1 to N by 1 do
            A[j][k] = A[j][k] − A[j][i] ∗ A[i][k];
        end
    end
end
```

com a decomposição podemos utilizar o seguinte algoritmo para obter um resultado:

Algorithm 9: LUP Solve

```
Input: A = matriz, N = Tamanho da matriz, P = Matriz de permutação, b = vetor
Output: x = solução
for i ← 0 to N by 1 do
    x[i] = b[P[i]];
    for k ← 0 to N by 1 do
        x[i] = x[i] − A[i][k] ∗ x[k];
    end
end
for i ← N − 1 to 0 by −1 do
    for k ← i + 1 to N by 1 do
        x[i] = x[i] − A[i][k] ∗ x[k];
    end
    x[i] = x[i]/A[i][i];
end
```

Também necessitaremos do método de newton para resolver sistemas de equações lineares que estamos a resolver no problema, para utilizar o método de newton é preciso usar uma derivada de ordem superior das equações lineares, como não temos acesso a essa derivada uma aproximação é criada a partir de *forward difference*[14], para aproximar os resultados ao valor real também temos de criar a derivada das condições de fronteira utilizando o mesmo método.

Após calcularmos as aproximações de derivadas uma matriz Jacobiana é construída que segue o seguinte formato (exemplo

de segunda ordem e quatro pontos)[24]:

$$\begin{bmatrix} 1 & 1 & 2 & 2 & 0 & 0 & 0 & 0 \\ 1 & 1 & 2 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 \\ 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 2 & 2 \\ 0 & 0 & 0 & 0 & 1 & 1 & 2 & 2 \\ \\ 3 & 3 & 0 & 0 & 0 & 0 & 4 & 4 \\ 3 & 3 & 0 & 0 & 0 & 0 & 4 & 4 \end{bmatrix} \tag{4}$$

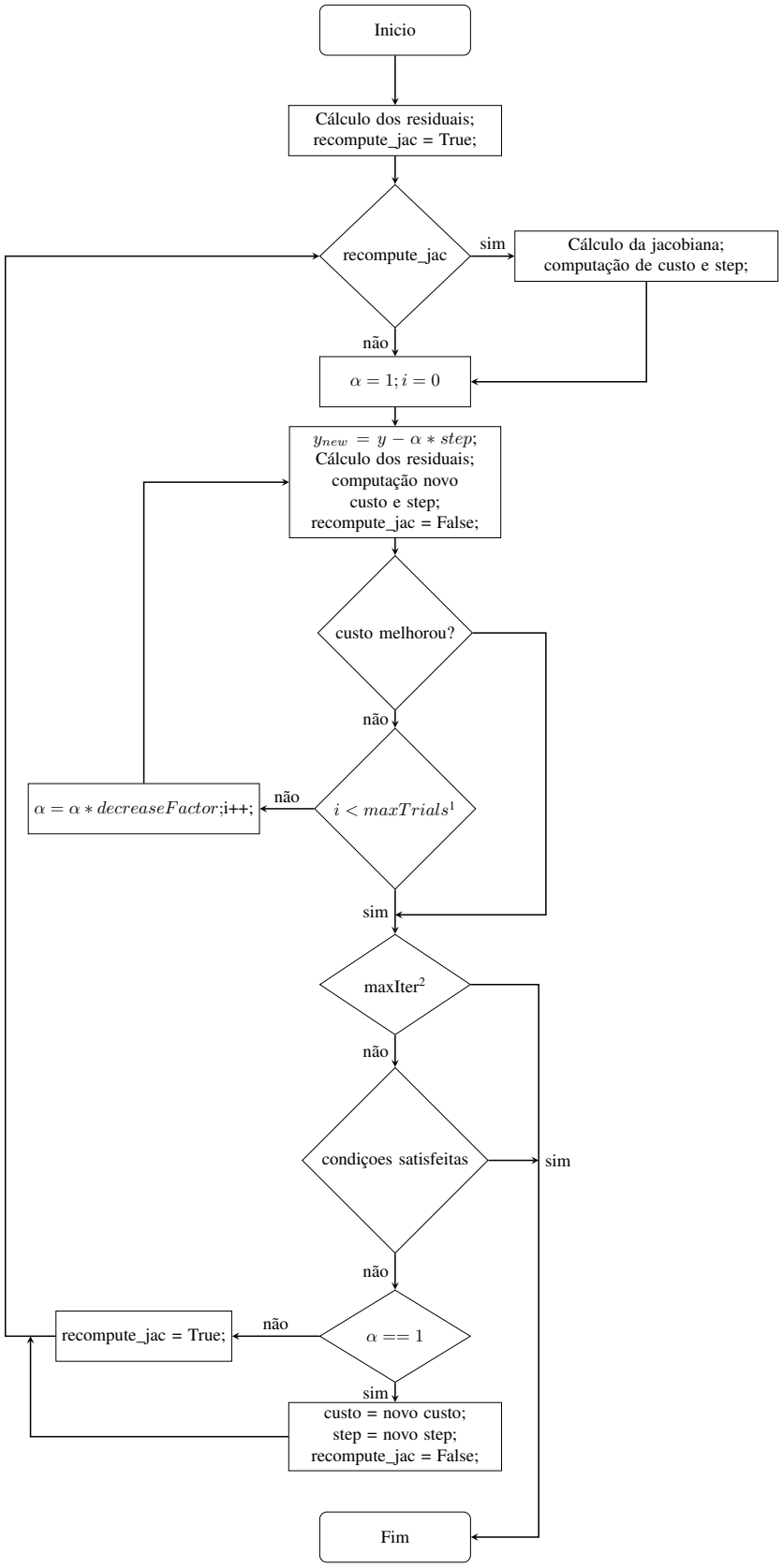
Os zeros correspondem a zeros na matriz jacobiana, os uns correspondem à diagonal do número de pontos - 1 e são utilizados para os residuais de colocação, os dois correspondem a diagonal secundaria do número de pontos - 1 também são utilizados para os residuais de colocação, os três são utilizados para as restrições iniciais das condições de fronteira e os quatro são utilizados para as restrições finais das condições de fronteira.

Com esta matriz podemos aplicar a decomposição de LU e resolver com os residuais de colocação para calcular por quanto é que se deve alterar a atual estimativa, também utilizámos um escalar α para controlar se estamos realmente a aproximar do valor real.

$$y_{step} = solveLU(J, residuais), \text{ transformar o } y_{step} \text{ numa matriz como } y$$

$$y = y_{old} - \alpha * y_{step}$$

com isto temos uma nova aproximação do valor real se compararmos as condições de fronteira proximidade do valor anterior com as da nova estimativa sabemos se estamos a aproximar do valor real. Se não estivermos a aproximar temos que sair do método de newton porque provavelmente o número de pontos que estamos a utilizar não é suficiente. Este processo de aproximar-se ao valor real repete-se até chegarmos a uma aproximação das condições de fronteira dentro das tolerâncias necessárias, se ainda não estamos dentro das tolerâncias necessárias decidimos se é necessário recalcular a Jacobiana (dependendo se o α teve que ser reduzido) e se nunca chegarmos a cumprir as tolerâncias necessárias saímos da rotina para adicionar mais pontos no problema.



[1]o numero máximo de tentativas foi realizado?
[2]chegou a iteração máxima?

Algorithm 10: Newton Method (FDM)[25]

Input: y = matriz, N = Tamanho da matriz, b = vetor
Output: x = solução
$residuals =$ colFun(y);
$recompute\_jac = True$;
$njev = 0$;
for $i \leftarrow 0$ **to** *MaxIteration + 1* **by** 1 **do**
    if $recompute\_jac$ **then**
        $njev = njev + 1$;
        $J =$ createJacobian();
        $P =$ LUPDecomposition$(A = J,\ N = y.size,\ T = Tolerance)$;
        $step =$ LUPSolve$(residuals)$;
        $cost =$ dot$(step, step)$;
    $\alpha = 1$;
    for $t \leftarrow 0$ **to** $maxTrials$ **by** 1 **do**
        $y_{new} = y - \alpha * step$;
        $residuals =$ colFun(y_{new});
        $newStep =$ LUPSolve$(residuals)$;
        $newCost =$ dot$(newStep, newStep)$;
        if $newCost < (1 - 2 * \alpha *$ *MinImprovement*$) * cost$ **then**
            break
        if $t < maxTrials$ **then**
            $\alpha = \alpha *$ decreaseFactor;
    end
    if $njev == maxFullIterations$ **then**
        break
    if *boundary conditions inside tolerances* **then**
        break
    if $\alpha == 1$ **then**
        $step = newStep$;
        $cost = newCost$;
        $recompute\_jac = False$;
    else
        $recompute\_jac = True$;
end

Após resolvermos o método de newton podemos criar uma spline cúbica que servirá para fazer interpolação de valores para além dos pontos que foram utilizados[26].

Com esta spline podemos calcular o *Root Mean Square* dos residuais entre os pontos existentes que nos permite estimar a proximidade da nossa estimativa com a real[27]. Para esta implementação é utilizado três novos pontos em cada intervalo que depois é reduzido para um único no intervalo.

Algorithm 11: Estimate RMS Residuals [28]

$s = 1/2 * h * \sqrt{3/7}$;
for $i \leftarrow 0$ **to** *número de pontos - 1* **by** 1 **do**
  $x_{middle}[i] = x[i] + 0.5 * h$;
  $x_1[i] = x_{middle}[i] + s$;
  $x_2[i] = x_{middle}[i] + s$;
end
$y_1 = \text{spline}(x_1)$;
$y_2 = \text{spline}(x_2)$;
$y_{middle} = \text{spline}(x_{middle})$;
$y'_1 = \text{spline}(x_1, 1)$; *//second order*
$y'_2 = \text{spline}(x_2, 1)$;
$y'_{middle} = \text{spline}(x_{middle}, 1)$;
$f_1 = fun(x_1, y_1)$; *//real value*
$f_2 = fun(x_2, y_2)$;
$f_{middle} = fun(x_{middle}, y_{middle})$;
$r_1 = (y'_1 - f_1)/(1 + |f_1|)$;
$r_2 = (y'_2 - f_2)/(1 + |f_2|)$;
$r_{middle} = (y'_{middle} - f_{middle})/(1 + |f_{middle}|)$;
$rT_1 = 0$;
$rT_2 = 0$;
$rT_{middle} = 0$;
for $i \leftarrow 0$ **to** *número de pontos - 1* **by** 1 **do**
  $rT_1 = rT_1 + r_1^2$;
  $rT_2 = rT_1 + r_2^2$;
  $rT_{middle} = rT_1 + r_{middle}^2$;
end
$rmsResult = \sqrt{1/2 * (32/45 * r_{middle} + 49/90 * (r_1 + r_2))}$;

utilizando este resultado podemos saber o erro relativo em cada intervalo e se o erro for superior à tolerância permitida podemos acrescentar novos pontos nesses intervalos, para acelerar o processo ainda mais podemos acrescentar dois novos pontos quando o erro relativo é muito superior à tolerância permitida.

Algorithm 12: add necessary points

$insert1 = []$;
$insert2 = []$;
for i **to** *numero de pontos - 1* **by** 1 **do**
  if $rmsResult[i] > tolerance$ **then**
    if $rmsResult[i] > tolerance * 100$ **then**
      $insert2$.push(i);
    else
      $insert1$.push(i);
end
for i **to** $insert1.length - 1$ **by** 1 **do**
  x.push$(1/2 * (x[insert1[i]] + x[insert1[i] + 1]))$;
end
for i **to** $insert2.length - 1$ **by** 1 **do**
  x.push$((2 * x[insert2[i]] + x[insert2[i] + 1])/3)$;
end
for i **to** $insert2.length - 1$ **by** 1 **do**
  x.push$((x[insert2[i]] + 2 * x[insert2[i] + 1])/3)$;
end
$x = \text{sort}(x)$;

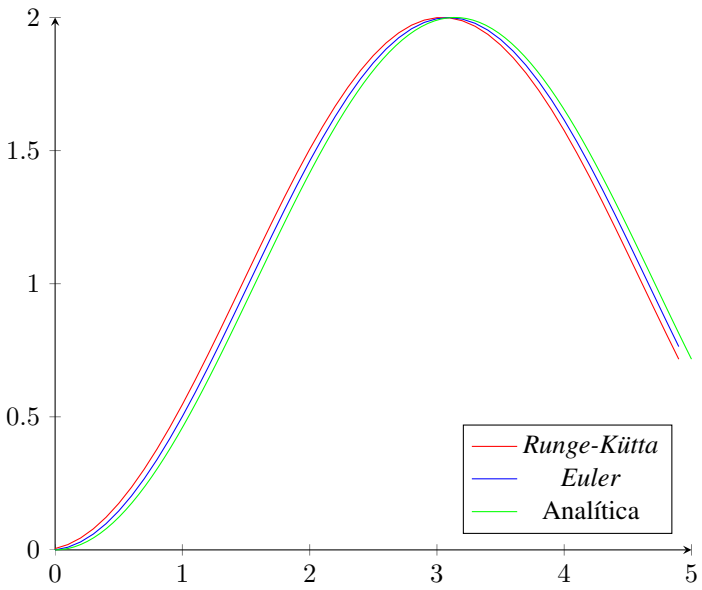
Repetidamente resolvendo este processo até chegarmos a tolerância necessária permite-nos resolver BVPs com uma elevada complexidade sendo que o algoritmo tem muitos benefícios por automaticamente recalcular os pontos do problema e por calcular a Jacobiana por *forward difference* não é necessário a introdução de uma função jacobiana para o nosso problema simplificando a sua utilização.

V. Avaliação

Com o objetivo de compreender se os algoritmos implementados foram realizados testes para testar a capacidade dos algoritmos em obter resultados fidedignos e no caso do *Finite Difference Method* foi testado o seu desempenho por ser um método pouco implementado em linguagens de baixo nível, por isso era interessante comparar com o desempenho de uma outra implementação. Para a testagem dos algoritmos estes foram comparados com algoritmos implementados em python ou comparados com soluções analíticas.

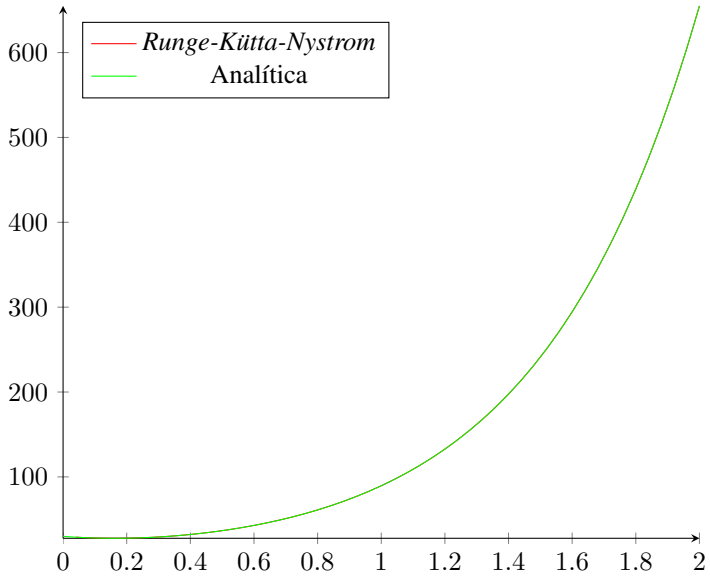
A. IVP

Os algoritmos de *Euler* e de *Runge-Kütta* foram utilizadas equações simples, neste caso foi utilizado $y' = sin(x)$ cuja integral é $y = -cos(x)$ utilizando um intervalo de $\Delta x = 0.1$:



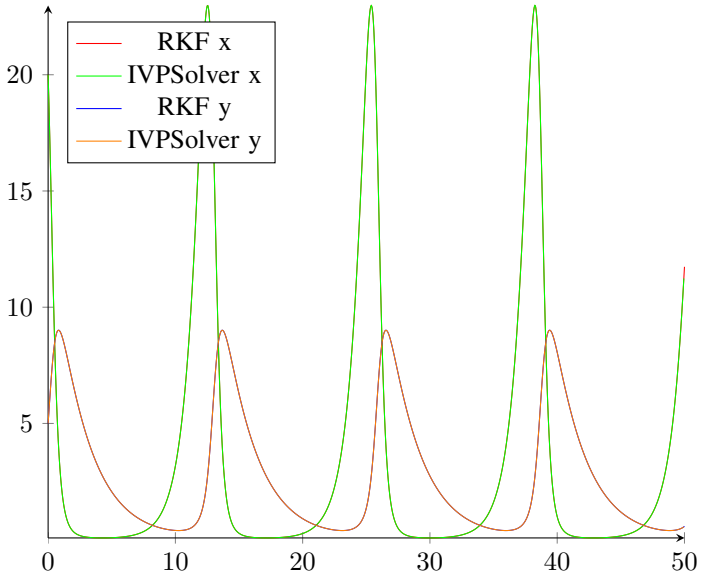
como podemos ver esta implementação funciona bem para problemas simples apesar de ser limitado a problemas de primeira ordem.

Para problemas de ordem mais elevada podemos utilizar o método de *Runge-Kütta-Nystrom* podemos utilizar o seguinte caso $y'' = 6y - y'$ que analiticamente é $y = 3e^{2x} + 2e^{-3x}$ utilizando um intervalo de $\Delta x = 0.04$



Como podemos ver quando utilizámos um Δx baixo obtemos bons resultados ao custo de performance pois estamos a computar mais pontos.

Também foi implementado *Runge-Kutta (RK4) nth order* (chamado internamente como *ivpSolver*) que nos permite resolver IVPs de ordem arbitrária, para esta teste utilizámos *Lotka–Volterra equations*[8] que são as seguintes equações $\frac{dx}{dt} = \alpha x - \beta xy$, $\frac{dy}{dt} = \delta xy - \gamma y$ que usamos os seguintes parâmetros $\alpha = 1.1,\ \beta = 0.4,\ \delta = 0.1,\ \gamma = 0.4,\ x = 20,\ y = 5,\ t = [0, 50]$ utilizando um intervalo de $\Delta x = 0.05$, uma solução exata para este problema só é conhecida quando $\alpha = \gamma$[29] por isso teremos que utilizar uma aproximação, para isso foi utilizado uma implementação de *Runge-Kutta-Fehlberg*[30] utilizada para resolver as equações de *Lotka-Volterra*.

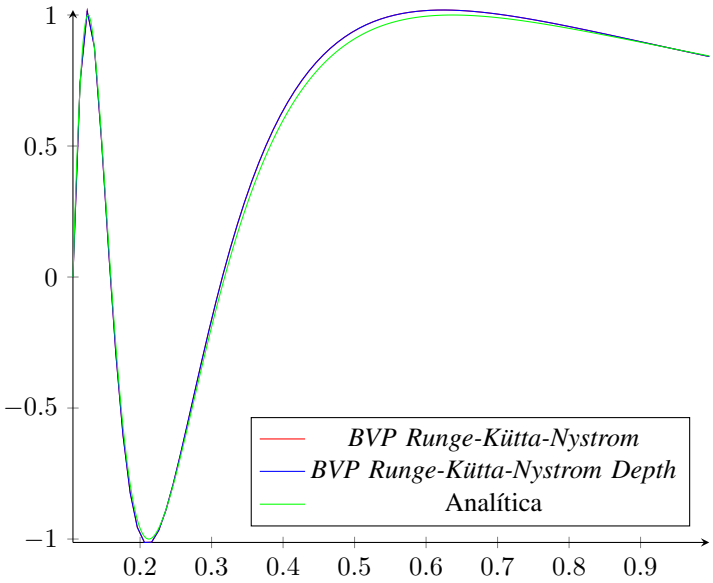


Nesta implementação também temos bons resultados pois ambos os métodos são semelhantes apesar disto, serve como confirmação que este algoritmo foi bem implementado.

B. BVP

1) Shooting Method:

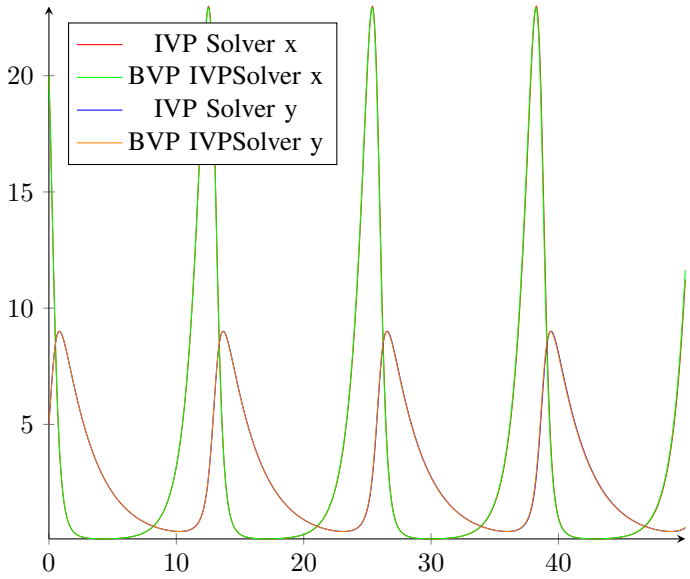
Para a implementação de *Runge-Kütta-Nystrom Shooting Method* utilizamos $y'' = -\frac{2y'}{x} - \frac{y}{x^4}$ com as condições $x = \left[\frac{1}{3\pi}, 1\right]$ $y(\frac{1}{3\pi}) = 0$, $y(1) = sin(1)$ com um intervalo de $\Delta x = 0.01$, sendo a equação analítica $y(x) = sin(\frac{1}{x})$



Podemos ver que apesar de ambos estarem muito próximos do resultado real, a implementação com aproximação repetitiva não foi muito melhor que a implementação com apenas uma iteração, isto deve-se em apenas uma iteração a aproximação já é muito precisa sendo $y(1) = sin(1) \approx 0.84147098480789651$ o valor obtido na implementação de apenas uma iteração foi $y(1) \approx 0.8414709848078962$ e na iteração repetida chegamos a $y(1) \approx 0.8414709848078964$ uma diferença de $\Delta = 2 * 10^{-16}$,

com problemas mais complexos a implementação repetitiva poderá ser mais útil, mas neste caso ambas implementações são competentes.

Para testar-mos a implementação do *Shooting Method RK4* n^{th} *order* foi utilizado as equações de *Lotka-Volterra* utilizando os mesmo parâmetros que foram utilizado para resolver o IVP, simplesmente transformando num problema BVP do tipo $x(0) = \alpha$ $x(50) = \beta$ removendo assim a condição inicial do $y(0) = \gamma$.



Este teste principalmente serve para perceber a eficiência do *fixed-point iteration* neste algoritmo pois como é a repetida utilização do *IVPSolver* já sabemos que se introduzirmos as condições inicias corretas o método consegue resolver, só é preciso testar o algoritmo que chega as condições inicias corretas.

2) Finite Difference Method:

A implementação de *Finite Difference Method* foi testada com um problema físico de um *pull-out* do deslizamento de um laminado de *FRP* e um bloco de betão. Este problema pode ser representado pela seguinte ODE:

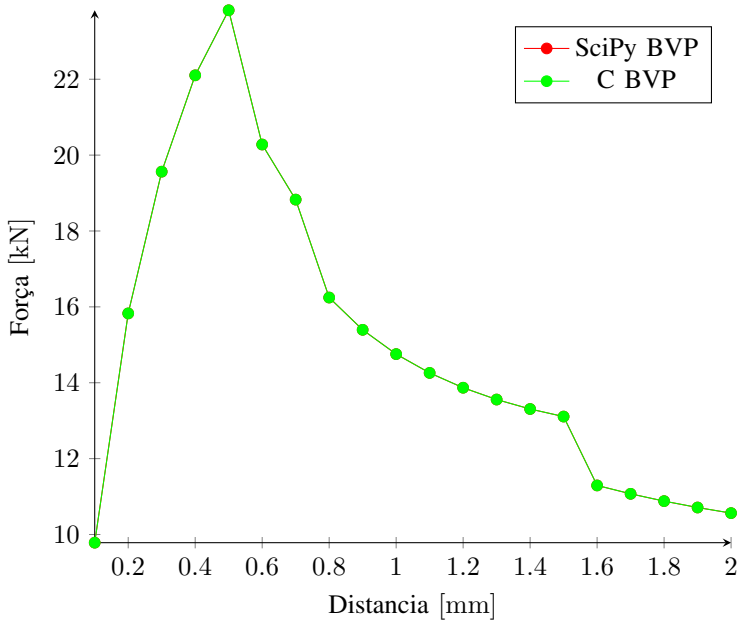
$$\frac{d^2s}{dx^2} = \frac{P_f}{A * E_f} * \tau$$

$$\tau = \begin{cases} \tau_m * \left(\frac{s}{s_m}\right)^{\alpha} & \text{se } s \leq s_m \\ \tau_m * \left(\frac{s}{s_m}\right)^{-\alpha'} & \text{se } s > s_m \end{cases}$$

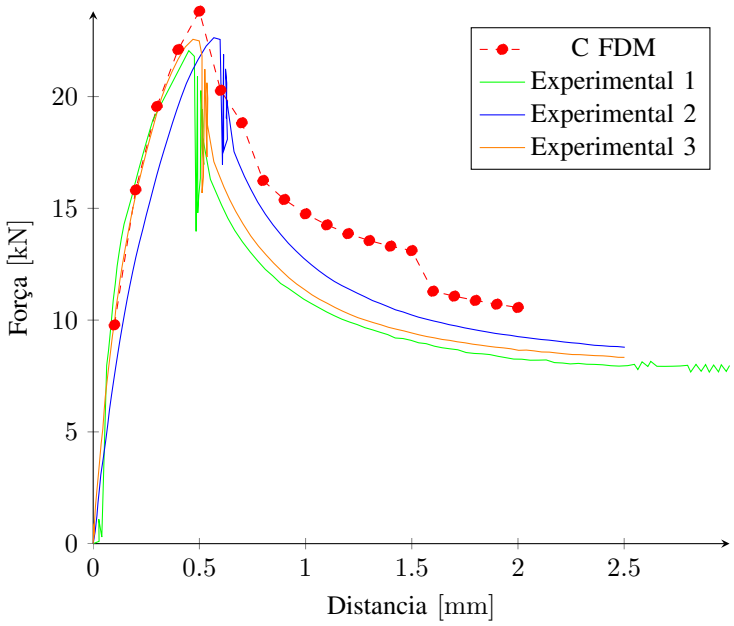
onde P_f é o perímetro do laminado em [mm], A é a área do laminado [mm^2], E_f é o Modulo da elasticidade do laminado [MPa], τ_m é a tensão máxima de corte [MPa], s_m é o deslizamento correspondente à tensão máxima de corte [mm], α é o parâmetro que define a curva, parte ascendente, α' é o parâmetro que define a curva, parte descendente. os valores são os seguintes:

$$P_f = 22.9 \text{ mm},\ A = 14 \text{ mm}^2,\ E_f = 169500 \text{ MPa},\ \tau_m = 18.35 \text{ MPa},\ s_m = 0.21 \text{ mm},\ \alpha = 0.5,\ \alpha' = 0.48$$

O problema consiste em resolver este BVP repetidamente e obter o τ no ultimo ponto para diferentes condições de fronteira, a $s(60) = [0.1, 2.0]$, $\Delta = 0.1$ é utilizado para transform num problema finito, a condição inicial é sempre $s(0) = 0$. Utilizando este método podemos obter o valor de $\frac{ds}{dx}$ no ultimo ponto e utiliza-lo para calcular a força a ser exercida, que têm a seguinte formula $N = E_f * A * \frac{ds}{dx}$, com esses pontos podemos coloca-los num gráfico e compara-los com a implementação de *SciPy*.

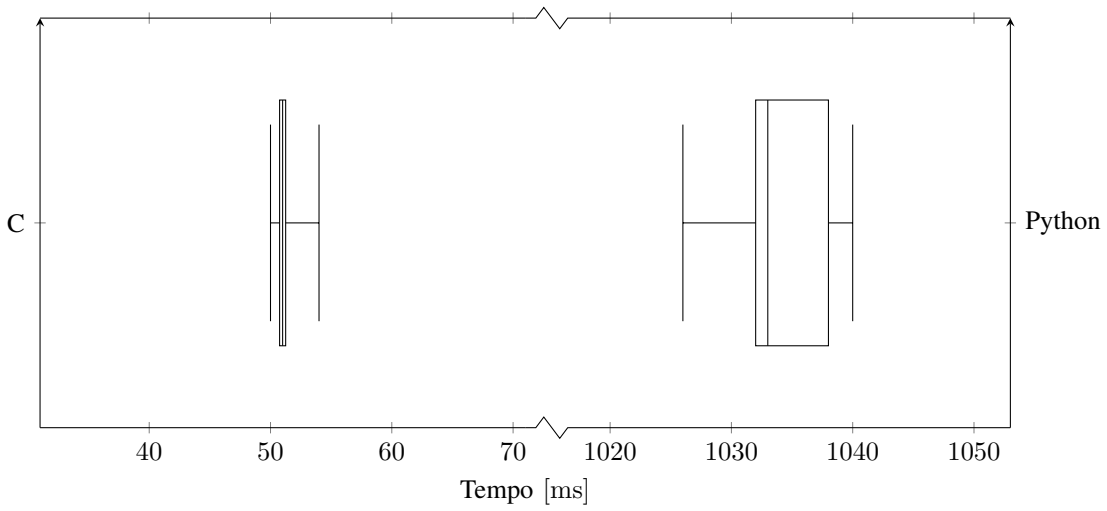


Conseguimos ver que por as duas implementações serem iguais, apenas mudando a linguagem de programação utilizada, os resultados são exatamente os mesmos. Podemos agora comparar com os valores obtidos experimentalmente:



Como podemos ver os valores não divergem muito dos valores obtidos experimentalmente, apesar de haver algumas discrepâncias especialmente por volta dos $1.5mm$ mas estes erros podem ser atribuídos a simplicidade da aproximação e as aproximações realizadas para conseguirmos modelar este problema, assim demonstrando a capacidade e o beneficio de utilizar métodos como estes para prever resultados experimentais, reduzindo no numero de experiências realizadas e conseguindo acelerar todo o processo de investigação.

Para alem das comparações em resultados podemos comparar a velocidade de execução do problema entre a implementação de *Python* e a implementação em *C*. Para obter resultados consistentes foram realizados 10 testes onde o tempo de execução não foi tido em conta seguido de 10 testes onde guardamos o tempo de execução para comparar entre os dois, este procedimento remove a irregularidade inicial pois as primeira execuções sofrem de não estarem *cached* nem outras otimizações utilizadas pelo sistema operativo e mesmo o próprio *CPU* fazendo-las mais lentas do que as execuções seguintes por isso é que realizamos o teste 10 vezes sem guardar os tempos de execução. Para medir o tempo de execução do *python* foi utilizado o comando do *Powershell* **Measure-Command** que permite medir o tempo de execução de qualquer comando executado à partir do *Powershell*, a implementação em *C* foi executada em *WSL2* por isso foi utilizada o comando **time** do *Unix*, o facto da implementação estar a correr em *WSL2* não deve afetar muito o desempenho pois deve continuar a executar sem interpretação.



Como podemos ver a implementação em *C* é muito mais rápida que a implementação em *python* e também é muito mais regular no seu tempo de execução. Isto deve-se a muitos fatores possivelmente o mais importante sendo o facto da implementação em *C* estar a correr em código de maquina e a implementação em *Python* estar a ser interpretada, apesar disto uma das outras possíveis razões pela qual o *Python* têm um *performance* tão mau como este pode ser a própria inicialização do interpretador de *Python* o que significa que se o problema fosse mais complexo e tivesse que correr mais código em *Python* o tempo de execução poderia não aumentar de uma forma linear em relação a estes testes, esse *overhead* não é apercetível na implementação em *C* por correr diretamente em código maquina e não necessitar de uma aplicação adicional para interpretar o código.

VI. Discussão

A. Sucesso

Com tudo o que foi implementado conseguimos resolver *BVPs* e *IVPs* que foi o que foi inicialmente pedido, tendo resultados comparativos com implementações já disponíveis publicamente, com um desempenho superior as implementações testas. Sendo o desempenho uma das principais razões pela qual este projeto foi direcionado pode-se dizer que o projeto cumpriu os objetivos com sucesso.

B. Proximos Passos

Apesar das implementações funcionarem nos problemas testados, isso não significa que não existem melhorias que podem ser feitas. Para a implementação *Shooting Method with RK4* n^{th} *order* podemos alterar o *Fixed Point Iteration Method* por um método como o método de *Newton* utilizado no *Finite Difference Method Solver* que melhoraria a eficácia do algoritmo de se aproximar do valor real e de não necessitar de uma estimativa inicial muito boa. Para o algoritmo que utiliza *Finite Difference Method* podemos implementar matrizes esparsas[31] que por apenas armazenarem os valores não zeros da matriz reduzem a memoria utilizada e permitem que quando operações são utilizadas sobre a matriz não necessite de calcular para valores não importantes, implementando matrizes esparsas pelo menos para as operações com a matriz jacobiana melhoraria desempenho pois a matriz jacobiana vai sempre conter muitos zeros sendo quase uma matriz diagonal. Em conjunto com as matrizes esparsas podemos usar o beneficio da *GPU* com ferramentas como *CUDA*[32] que tem uma biblioteca para operações sobre matrizes esparsas chamada *cuSPARSE*[33], como esta especializa-se em operações sobre matrizes esparsas com a grande capacidade de realizar operações em paralelo teria um grande aumento em desempenho especialmente em problemas de ordem elevada. Outro dos melhoramentos era alterar o algoritmo utilizado para resolver *LU Decomposition* que se resolvermos utilizar a biblioteca *cuSPARSE*, esta já contem funções para calcular a decomposição de LU, a necessidade de implementar uma melhor solução para a decomposição de LU vêm do facto da implementação atual não conseguir identificar se o problema é singular ou haver possibilidade de resolver sistemas de equações lineares mais complexas.

C. Conclusão

Neste projeto foi pedido pela implementação de diferentes métodos de resolver *ODEs* em problemas de fronteira e problemas iniciais, as implementações utilizadas para resolver *IVPs* foram o método de *Euler* e métodos baseados a volta de *Runge-Kütta*, após a implementação de simples algoritmos para resolver *IVPs* foram utilizados para resolver *BVPs* utilizando o *Shooting Method* que funciona resolvendo *IVPs* repetidamente aproximando a estimativa dos valores iniciais que não sabemos e melhorando de acordo com os valores de fronteira. Foi também *Finite Difference Method* que serve para resolver *BVPs* e que pode ser mais eficiente do que *Shooting Methods*. Após a implementação de todos os métodos foram realizados testes para comprovar a capacidade de resolver problemas e o seu desempenho com isto conseguimos estabelecer que todos os métodos foram implementados corretamente.

REFERENCES

[1] M. Osborne, "On shooting methods for boundary value problems," *Journal of Mathematical Analysis and Applications*, vol. 27, no. 2, pp. 417–433, 1969. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0022247X69900596

[2] C. Grossmann, H. Roos, and M. Stynes, *Numerical Treatment of Partial Differential Equations*, ser. Universitext. Springer Berlin Heidelberg, 2007. [Online]. Available: https://books.google.pt/books?id=Nu7-ZZAyPZYC

[3] P. Dawkins, "Section 8-1: Boundary value problems," accessed: 2021-06-20. [Online]. Available: https://tutorial.math.lamar.edu/classes/de/BoundaryValueProblem.aspx

[4] D. Zill, *A First Course in Differential Equations*, ser. The Prindle, Weber, and Schmidt Series in Mathematics. PWS-Kent Publishing Company, 1993. [Online]. Available: https://books.google.pt/books?id=HscNVZBjKykC

[5] G. Hemming, *An Elementary Treatise on the Differential and Integral Calculus ...* Macmillan & Company, 1852. [Online]. Available: https://books.google.co.ao/books?id=IqkKAAAAYAAJ

[6] F. Dorner and R. Mosleh, "Analysis of ode models for malaria propagation," *Acta Marisiensis. Seria Technologica*, vol. 17, no. 1, pp. 31–39, 2020.

[7] Y. Kashiwagi, "Prediction of ballistic missile trajectories," STANFORD RESEARCH INST MENLO PARK CA, Tech. Rep., 1968.

[8] A. J. Lotka, "Analytical note on certain rhythmic relations in organic systems," *Proceedings of the National Academy of Sciences*, vol. 6, no. 7, pp. 410–415, 1920.

[9] A. Iserles, *A First Course in the Numerical Analysis of Differential Equations*, ser. A First Course in the Numerical Analysis of Differential Equations. Cambridge University Press, 2009. [Online]. Available: https://books.google.pt/books?id=M0tkw4oUucoC

[10] E. Süli and D. F. Mayers, *An introduction to numerical analysis*. Cambridge university press, 2003.

[11] V. M. Cunha, J. A. Barros, and J. Sena-Cruz, "Pullout behaviour of hooked-end steel fibres in self-compacting concrete," Universidade do Minho. Departamento de Engenharia Civil (DEC), Tech. Rep., 2007.

[12] "Runge kutta 4 method to solve second order ode," accessed: 2021-06-20. [Online]. Available: https://www.mathworks.com/matlabcentral/answers/648103-runge-kutta-4-method-to-solve-second-order-ode

[13] "Numerical methods: Fixed point iteration," accessed: 2021-07-7. [Online]. Available: https://metric.ma.ic.ac.uk/metric_public/numerical_methods/iteration/fixed_point_iteration.html

[14] A. Yew, "Numerical differentiation: finite differences," accessed: 2021-07-8. [Online]. Available: https://www.dam.brown.edu/people/alcyew/handouts/numdiff.pdf

[15] G. Nagy, "Solving the heat equation," accessed: 2021-07-8. [Online]. Available: https://users.math.msu.edu/users/gnagy/teaching/12-spring/mth235/L38-235.pdf

[16] L. O. Jay, "Lobatto methods," in *Encyclopedia of Applied and Computational Mathematics*. Springer Berlin Heidelberg, 2015, pp. 817–826. [Online]. Available: https://doi.org/10.1007%2F978-3-540-70529-1_123

[17] J. Kierzenka and L. F. Shampine, "A bvp solver based on residual control and the maltab pse," *ACM Trans. Math. Softw.*, vol. 27, no. 3, p. 299–316, Sep. 2001. [Online]. Available: https://doi.org/10.1145/502800.502801

[18] "Gcc, gnu compiler," accessed: 2021-07-22. [Online]. Available: https://gcc.gnu.org/

[19] "Wsl, windows subsystem for linux," accessed: 2021-07-22. [Online]. Available: https://docs.microsoft.com/en-us/windows/wsl/install-win10

[20] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[21] "scipy.integrate.solve_bvp," accessed: 2021-07-9. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_bvp.html

[22] P. Okunev and C. R. Johnson, "Necessary and sufficient conditions for existence of the lu factorization of an arbitrary matrix," *arXiv preprint math/0506382*, 2005.

[23] "Lu decomposition," accessed: 2021-07-10. [Online]. Available: https://en.wikipedia.org/wiki/LU_decomposition#C_code_example

[24] "Construct global jacobian," accessed: 2021-07-13. [Online]. Available: https://github.com/scipy/scipy/blob/44e31b1eb202ef91caad059952ba268d2610a60b/scipy/integrate/_bvp.py#L158

[25] "Solve newton," accessed: 2021-07-15. [Online]. Available: https://github.com/scipy/scipy/blob/v1.7.0/scipy/integrate/_bvp.py#L347-L499

[26] "Cubic spline," accessed: 2021-07-15. [Online]. Available: https://mathworld.wolfram.com/CubicSpline.html

[27] "Root mean square," accessed: 2021-07-13. [Online]. Available: https://en.wikipedia.org/wiki/Root-mean-square_deviation

[28] "Estimate rms residuals," accessed: 2021-07-15. [Online]. Available: https://github.com/scipy/scipy/blob/v1.7.0/scipy/

integrate/_bvp.py#L526-L574
[29] K. Murty and D. Rao, "Approximate analytical solutions of general lotka-volterra equations," *Journal of Mathematical Analysis and Applications*, vol. 122, no. 2, pp. 582–588, 1987. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0022247X8790285X
[30] "Lotka-volterra equation solver," accessed: 2021-07-16. [Online]. Available: https://fusion809.github.io/LotkaVolterra/
[31] R. Yuster and U. Zwick, "Fast sparse matrix multiplication," *ACM Transactions On Algorithms (TALG)*, vol. 1, no. 1, pp. 2–13, 2005.
[32] "Cuda," accessed: 2021-07-22. [Online]. Available: https://developer.nvidia.com/cuda-zone
[33] "cusparse - cuda," accessed: 2021-07-20. [Online]. Available: https://docs.nvidia.com/cuda/cusparse/index.html