Relatório Métodos Numéricos

Mateus Caracciolo Marques

21 de Outubro, 2018

Abstract

Projeto da disciplina de Métodos Numéricos(IF816) cujo propósito é apresentar diversos métodos para resolução numérica de equações diferenciais(EDO) de 1° ordem com valor inicial do tipo $y'(t) = f(t, y),\ y(t_0) = y_0$ escritos na linguagem Python, versão 3.7.0, utilizando a biblioteca Sympy [1].

[1] `https://www.sympy.org/en/index.html`

Contents

1 Introdução

Muitos problemas encontrados em Engenharia e em outros ciências podem ser formulados em termos de equações diferênciais ordinárias, tais como o crescimento populacional de uma certa espécie ou a evolução de componentes eletrônicos em um circuito elétrico. Dessa forma somos naturalmente levados à investigar soluções de tais equações, porém como grande parte das EDO's não possuem soluções elementares surge a necessidade do estudo de soluções numéricas q aproximam bem a solução em uma dada vizinhança. Iremos expor vários desses métodos, focados em EDO's de 1º ordem, visto que EDO's de ordens mais altas podem ser reduzidas a um sistema de 1º ordem mediante uma substituição adequada.

1.1 O objeto EDO

A implementação dos métodos numéricos serão todos implementados em Python no escopo da Classe EDO, que possui como parâmetros as condições iniciais da EDO, $t_0,\ y_0$ e uma string representando a $f(t, y(t))$:

Listing 1: Classe ODE

```
import sympy as sp

class ODE:

    adam = {}
    multon = {}
    diff = {}

    def __init__(self, t0, y0, str_expr):

        t, y = sp.symbols('t, y')
        expr = sp.sympify(str_expr)
        self.t0 = t0
        self.y0 = y0
        self.f = sp.lambdify((t, y), expr)
        return None
```

2 Métodos Numéricos

Boa parte dos métodos são basedos em técnicas de Integração Numérica derivação numérica e na expansão em série de Taylor. Nesta e na próxima seção daremos uma breve motivação de cada um dos métodos e suas respectivas implementações em Python.

2.1 Método de Euler

Considerando o problema de valor inicial $y'(t) = f(t, y(t)),\ y(t_0) = y_0$, temos que se f for suficientemente derivável em relação a t e y podemos expandir $y(t)$ em série de Taylor. Considerando a expansão até 1 ordem, temos:

$$y(t+h) = y(t) + y'(t)h + O(h^2) \tag{1}$$

Como $y'(t) = f(t, y(t))$, obtemos:

$$y(t+h) = y(t) + hf(t, y(t)) + O(h^2) \tag{2}$$

Ignorando os termos de ordem superior, obtemos o Método de Euler:

$$\begin{aligned} y(t_0) &= y_0 \\ t_{n+1} &= t_n + h \\ y_{n+1} &= y_n + hf(t_n, y_n) \end{aligned}$$

Onde deve-se ser especificado o tamanho de cada passo (h) e a quantidade de passo (nsteps). É uma consequência direta da série de Taylor que o erro de truncamento local é dado por $E = \frac{h^2}{2}y''(\xi)$. Segue implementação em Python:

Listing 2: Método de Euler

```
def euler(self, h, nsteps):
    res = []
    res.append([self.t0, self.y0])
    t, y = self.t0, self.y0
    for i in range(1, nsteps + 1):
        y = y + h*self.f(t, y)
        t = t + h
        res.append([t, y])
    return res
```

2.2 Método de Euler Inverso

Modificando (1) para partirmos de um passo na frente, temos:

$$y(t-h) = y(t) - y'(t)h + O(h^2)$$

$$y(t) = y(t-h) + f(t, y(t)) + O(h^2)$$

Dessa forma, se descartarmos os termos de ordem maior ou igual a 2 obtemos o Método de Euler Inverso:

$$y(t_0) = y_0$$
$$t_{n+1} = t_n + h$$

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}) \tag{3}$$

Segue o código em Python:

Listing 3: Método de Euler Inverso

```
def euler_inverso(self, h, nsteps):
    res = []
    res.append([self.t0, self.y0])
    t, y = self.t0, self.y0
    for i in range(1, nsteps + 1):
        k = y + h*self.f(t, y)
        y = y + h*self.f(t + h, k)
        t = t + h
        res.append([t, y]
    return res
```

2.3 Método de Euler aprimorado

O método de Euler e o de Euler Inverso, embora simples de aplicar, na prática não é tão útil do ponto de vista numérico, visto que é necessário ter um pequeno comprimento entre cada etapa e consequentemente uma quantidade maior de etapas. O método de Euler Aprimorado, ou método de Heunn, visa melhorar o método de Euler coletando termos de ordem maiores na série de Taylor e aproximando cada derivada pela sua respectiva diferença finita. Tecnicamente falando, considere a expansão de Taylor de ordem 2:

$$y(t + h) = y(t) + y'(t)h + \frac{y''(t)h^2}{2} + O(h^3) \tag{4}$$

Dado que $y'(t) = f(t, y(t))$, então temos que $y''(t) = \frac{df(t,y(t))}{dt}$. Aproximando essa derivada pela sua respectiva diferença finita, temos:

$$\frac{df(t, y(t))}{dt} \approx \frac{f(t + h, y(t + h)) - f(t, y(t))}{h} \tag{5}$$

Utilizando (4) em (3) e ignorando os termos de ordem superior obtemos o Método de Euler aprimorado:

$$y(t_0) = y_0$$
$$t_{n+1} = t_n + h$$

$$y(t_{n+1}) = y(t_n) + \frac{h}{2}(f(t_n, y(t_n)) + f(t_{n+1}, y(t_{n+1})))$$

Perceba que está equação também eh um equação implicita. Para conseguirmos utiliza-la novamente iremos utilizar o método de previsão com Euler, obtendo:

$$y_{n+1} = y_n + \frac{h}{2}(f(t_n, y_n) + f(t_{n+1}, y_n + hf(t_n, y_n)) \tag{6}$$

Listing 4: Método de Euler Aprimorado

```
def euler_aprimorado(self, h, nsteps):
    res = []
    res.append([self.t0, self.y0])
    t, y = self.t0, self.y0
    for i in range(1, nsteps + 1):
        k = y + h*self.f(t, y)
        y = y + 0.5*h*(self.f(t, y) + self.f(t + h, k))
        t = t + h
        res.append((t,y))
    return res
```

2.4 Runge-Kutta

Os métodos de Runge-Kutta é uma família importante de métodos que podem ser implícitos e explicitos que possuem alta precisão. Estes métodos são obtidos realizando uma integração numérica e depois fazendo com que essa aproximação satisfaça a série de Taylor, chegando em uma série de equações que possuem várias resposta, sendo escolhido uma delas para se obter um método conveniente. Formalmente, dado a EDO $y'(t) = f(t, y(t))$, podemos escreve-la de forma equivalente como:

$$y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} f(s, y(s))ds \tag{7}$$

O método de Runge-Kutta consiste em aproximar essa integral utilizando quadratura:

$$\int_{t_n}^{t_{n+1}} f(s, y(s))ds \approx \ h\sum_{k=1}^{n} w_i f(t_n + \beta_i h, y(t_n + \beta_i h)) \tag{8}$$

onde $w_i,\ \beta_i \in [0, 1]$. Essa quadratura pode se tornar bem difícil de calcular pois não conhecemos o termo $y(t_n + \beta_i h)$. Contornamos esse problema considerando $\beta_1 = 0$, tornando o primeiro termo da quadratura $K_1 = hf(t_n, y(t_n))$. Podemos utilizar essa informação para aproximar o segundo termo da quadratura $y(t_n + \beta_2 h)$, pois

$$y(t_n + \beta_2 h) = y(t_n) + \beta_2 hf(t_n, y_n) + O(h^2)$$

Assim, podemos considerar o segundo termo da quadratura como sendo $K_2 = hf(t_n + \alpha_2 h, y_n + \beta K_1)$. Continuamos dessa maneira de forma que o n-ésimo termo da quadratura toma a forma:

$$K_n = hf(t_n + \alpha_n h, y_n + \sum_{i=1}^{n} \beta_{ni} K_i)$$

Resta agora descobrir os coeficientes w_i, α_i, β_{ji}. Isto pode ser feito jogando a expressão da quadratura na série de Taylor e coletando potências de h. A escolha dos coeficientes é que vai dizer qual dos métodos de Runge- Kutta iremos utilizar.

$$y(t_{n+1}) = \sum_{k=0}^{n} \frac{y^{(k)} h^k}{k!} \tag{9}$$

O mais famoso dos métodos de Runge-Kutta é o caso $n = 4$, onde o método toma a forma:

$$\begin{aligned} k1 &= hf(t_n, y_n) \\ k2 &= hf(t_n + \frac{h}{2}, y_n + \frac{k1}{2}) \\ k3 &= hf(t_n + \frac{h}{2}, y_n + \frac{k2}{2}) \\ k4 &= hf(t_n + h, y_n + k3) \end{aligned}$$

$$y_{n+1} = y_n + \frac{h}{6}(k1 + 2k2 + 2k3 + k4)$$

Segue o código em Python:

Listing 5: Método de Runge-Kutta

```
def runge_kutta(self, h, nsteps):
    res = []
    res.append([self.t0, self.y0])
    t, y = self.t0, self.y0
    for i in range(1, nsteps + 1):
        k1 = h * self.f(t, y)
        k2 = h * self.f(t + 0.5*h, y + 0.5*k1)
        k3 = h * self.f(t + 0.5*h, y + 0.5*k2)
        k4 = h * self.f(t + h, y + k3)
        y = y + (1/6)*(k1 + 2*k2 + 2*k3 + k4)
        t = t + h
        res.append([t, y])
    return res
```

3 Fórmulas Multistep

Ao contrário dos métodos mostrados anteriormente, agora veremos métodos que utilizam informação de mais do que apenas 1 ponto anterior,sendo necessário a sua inicialização atráves de outros métodos que que não dependam de vários passos anteriores.

3.1 Método de Adams-Bashforth

Utilizando a representação equivalente da EDO:

$$y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} f(s, y(s))ds \tag{10}$$

Dessa forma iremos aproximar a integral através de um polinômio de grau s que interpole nos pontos $(t_{n-s}, y_{n-s}), ..., (t_n, y_n)$. Considerando $f(t_n, y_n) = f_n$, temos que esse polinômio é o polinômio interpolador de Lagrange e possui forma:

$$L_s(x) = \sum_{i=0}^{s-1} f_i \prod_{j \neq i} \frac{t - t_{n-j}}{t_{n-i} - t_{n-j}} \tag{11}$$

Substituindo (10) em (9) e fazendo a substituição $u = \frac{t-t_n}{h}$ na integral, chegamos no Método de Adams-Bashforth:

$$y(t_0) = y_0$$

$$t_{n+1} = t_n + h$$

$$y_{n+1} = y_n + h \sum_{i=1}^{s} \frac{(-1)^{s-i}}{(i-1)!(s-i)!} \int_0^1 \prod_{j=1, j \neq i}^{s} (t + s - j) \tag{12}$$

Dessa forma, para descrever o método de Adams-Bashforth, além de ser necessário o tamanho do passo (h) e a quantidade de passos (nsteps), também é necessário fornecer a ordem do método (order) e a maneira que vai ser obtido os 'order' primeiro valores. Segue o código em Python:

Listing 6: Método que armazena os coeficientes de Adams-Bashforth

```
def coef_adam(self, order):
    x = sp.symbols('x')
    if order not in self.adam:
        self.adam[order] = []
        for i in reversed(range(order)):
            str_expr = '1'
            for j in range(order):
                if j == order - i - 1:
                    continue
                str_expr += f' * (x + {j})'
            expr = sp.sympify(str_expr)
            expr = sp.lambdify((x), expr)
            value = sp.integrate(expr(x), (x, 0, 1))
            self.adam[order].append(((-1)**(order - i - 1) * value)
            / (sp.factorial(i) * sp.factorial((order - i - 1))))
        return None
```

Listing 7: Método que calcula próxima etapa de Adams-Bashforth

```
def new_y_adam(self, h, res, index, order):


    self.coef_adam(order)
    y = res[index - 1][1]
    for i in range(1, order + 1):
        y += (h * self.adam[order][i - 1]
              * self.f(res[index - i][0], res[index - i][1])
              )
    return y
```

Listing 8: Método de Adams-Bashforth

```
def adam_bashfort(self, h, nsteps, order, preview):


    metodo = {'euler': self.euler, 'euler_inverso': self.euler_inverso,
              'euler_aprimorado': self.euler_aprimorado,
              'runge_kutta': self.runge_kutta }
    res = metodo[preview](h, order - 1)
    t = self.t0 + (order - 1)*h
    for i in range(order, nsteps + 1):
        y = self.new_y_adam(h, res, i, order)
        t = t + h
        res.append([t, y])
    return res
```

Listing 9: Método de Adams-Bashforth com lista

```
def adam_bashfort_lista(self, h, nsteps, order, lista):
    res = lista
    t = self.t0 + (order - 1)*h
    for i in range(order, nsteps + 1):
        y = self.new_y_adam(h, res, i, order)
        t = t + h
        res.append([t, y])
    return res
```

3.2 Método de Adams-Moulton

O método de Adams-Multon é uma variação do método de Adams-Bashforth, porém se trata de um método implicito, visto que o polinômio dessa vez é escolhido para interpolar pelo ponto (t_{n+1}, y_{n+1}).

$$y(t_0) = y_0$$

$$t_{n+1} = t_n + h$$

$$y_{n+1} = y_n + h \sum_{i=1}^{s} \frac{(-1)^{s-i}}{(i-1)!(s-i)!} \int_0^1 \prod_{j=1, j\neq i}^{s} (t+s-j) \tag{13}$$

Segue o código de Python:

Listing 10: Método que calcula coeficientes de Adams-Multon

```
def coef_multon(self, order):
    x = sp.symbols('x')
    if order not in self.multon:
        self.multon[order] = []
        for i in reversed(range(order)):
            str_expr = '1'
            for j in range(order):
                if j == order - i - 1:
                    continue
                str_expr += f' * (x + {j} - 1)'
            expr = sp.sympify(str_expr)
            expr = sp.lambdify((x), expr)
            value = sp.integrate(expr(x), (x, 0, 1))
            self.multon[order].append(((-1)**(order - 1 - i) * value)
            / (sp.factorial(i) * sp.factorial((order - i - 1))))
```

Listing 11: Método que calcula próxima etápa de Adams-Multon

```
def new_y_multon(self, h, res, index, order):
    self.coef_multon(order)
    y = res[index - 1][1]
    y += (h * self.multon[order][0]
          * self.f(self.t0 + index*h,
          self.new_y_adam(h, res, index, order))
          )
    for i in range(2, order + 1):
        y += (h * self.multon[order][i - 1]
              * self.f(res[index - i + 1][0], res[index - i + 1][1])
              )
    return y
```

Listing 12: Método de Adams-Multon

```
def adam_multon(self, h, nsteps, order, preview):
    metodo = {'euler': self.euler, 'euler_inverso': self.euler_inverso,
              'euler_aprimorado': self.euler_aprimorado,
              'runge_kutta': self.runge_kutta }
    res = metodo[preview](h, order - 1)
    t = self.t0 + (order - 1)*h
    for i in range(order, nsteps + 1):
        y = self.new_y_multon(h, res, i, order)
        t = t + h
        res.append([t, y])
    return res
```

Listing 13: Método de Adams-Multon com lista

```
def adam_multon_lista(self, h, nsteps, order, lista):
    res = lista
    t = self.t0 + (order - 1)*h
    for i in range(order, nsteps + 1):
        y = self.new_y_multon(h, res, i, order)
        t = t + h
        res.append([t, y])
    return res
```

3.3 Fórmula da Diferenciação Inversa

O método da Diferenciação Inversa é obtido realizando a interpolação da função $y(t)$ nos pontos $(t_{n+1}, y_{n+1}), \dots, (t_{n-s+1}, y_{n-s+1})$ e em seguida calculando sua derivada em t_{n+1} e usando o fato que $y'(t) = f(t, y(t))$. Dessa forma, considerando a polinômio interpolador de Lagrange que interpola esses pontos, temos:

$$y(t) \approx \sum_{i=0}^{s} y_{n-i+1} L_i(t)$$

Derivando no ponto $t = t_{n+1}$, obtemos:

$$\frac{dy(t_{n+1})}{dt} \approx \sum_{i=0}^{s} \frac{dL_i(t_{n+1})}{dt} y_{n-s+1}$$

Tomando a mudança de variável $\tau = \frac{t_{n+1}-t}{h}$, a equação toma a forma:

$$\frac{dy(t_{n+1})}{dt} \approx \sum_{i=0}^{s} \frac{d\widehat{L}_i(0)}{d\tau} \frac{d\tau(t_{n+1})}{dt} y_{n-s+1}$$

Onde $\widehat{L}_i(t)$ é base de Lagrange nos pontos $t = 0, 1, \ldots, s$. Como $y'(t_{n+1}) = f(t_{n+1}, y_{n+1})$, obtemos:

$$f(t_{n+1}, y_{n+1}) = -\frac{1}{h} \sum_{i=0}^{s} \frac{d\widehat{L}_i(0)}{d\tau} y_{n-s+1}$$

Assim, obtemos que:

$$y_{n+1} = h\beta_0 f(t_{n+1}, y_{n+1}) + \sum_{i=0}^{s} \alpha_i y_{n-i+1}$$

Onde os coeficientes $\beta_0,\ \alpha_0,\ \ldots, \alpha_s$ satisfazem:

$$1 = -\beta_0 \frac{d\widehat{L}_0(0)}{d\tau}$$

$$\alpha_i = -\beta_0 \frac{d\widehat{L}_i(0)}{d\tau},\ i = 1, \ldots, s$$

Para mais referências, veja a bibliografia. Segue o código em Python:

Listing 14: Método que calcula coeficientes de Formula da Diferenciação Inversa

```
def coef_diff(self, order)
    x = sp.symbols('x')
    if order not in self.diff:
        self.diff[order] = []
        for i in range(0, order + 1):
            str_expr = '1'
            for j in range(order + 1):
                if j == i:
                    continue
                str_expr += f' * (x - {j})/({i} - {j})'
            expr = sp.sympify(str_expr)
            expr = sp.diff(expr, x)
            value = expr.evalf(subs={x: 0})
            if i == 0:
                self.diff[order].append(-1/value)
            else:
                self.diff[order].append(self.diff[order][0] * value)
```

Listing 15: Método que calcula próxima etapa de Formula da Diff. Inversa

```
def new_y_inversa(self, h, res, index, order):
    self.coef_diff(order)
    k = self.new_y_adam(h, res, index, order)
    y = (h * self.diff[order][0]
         * self.f(self.t0 + index*h , k)
         )
    for i in range(1, order + 1):
        y += self.diff[order][i] * res[index - i][1]
    return y
```

Listing 16: Método da Diferenciação Inversa

```
def formula_inversa(self, h, nsteps, order, preview):
    metodo = {'euler': self.euler, 'euler_inverso': self.euler_inverso,
              'euler_aprimorado': self.euler_aprimorado,
              'runge_kutta': self.runge_kutta }
    res = metodo[preview](h, order - 1)
    t = self.t0 + (order - 1)*h
    for i in range(order, nsteps + 1):
        y = self.new_y_inversa(h, res, i, order)
        t = t + h
    return res
```

Listing 17: Método da Diferenciação Inversa com lista

```
def formula_inversa_lista(self, h, nsteps, order, lista):
    res = lista
    t = self.t0 + (order - 1)*h
    for i in range(order, nsteps + 1):
        y = self.new_y_inversa(h, res, i, order)
        t = t + h
        res.append([t, y])
    return res
```

4 Conclusão

Foram discutidos vários métodos variando entre métodos implícitos e explicitos, métodos siglestep e multistep sendo destacado suas motivações. Como um apelo final para a precisão dos métodos e sua respectiva velocidade de convergência e implementação, será feito um comparativo entre os resultados da EDO $y'(t) = 1 - t + 4y,\ y(0) = 0,\ h = 0.1$ e $nsteps = 20$, onde os métodos são de uma certa ordem fixa.

nsteps	Euler	Euler Inverso	Euler Aprimorado	Runge-Kutta	Exata
1	0.1000000	0.1300000	0.1149999	0.1171999	x
2	0.2300000	0.3188000	0.2732000	0.2797378	x
3	0.4020000	0.5993280	0.4953360	0.5099075	x
4	0.6328000	1.0229516	0.8120972	0.8409660	x
5	0.9459200	1.6698046	1.6698046	1.2689039	x
10	5.486024	16.067189	9.5165615	10.293385	x
15	29.35651	148.05415	67.319100	75.760939	x
20	157.1904	1366.4463	477.00199	558.55790	x

Table 1: Métodos siglesteps

nsteps	Lista	Euler	Euler Inv.	Euler Apr.	Runge-Kutta	Exata
1	0.1000000	0.1000000	0.1300000	0.1149999	0.1171999	y
2	0.2300000	0.2300000	0.3188000	0.2732000	0.2797378	y
3	0.4020000	0.4020000	0.5993280	0.4953360	0.5099075	y
4	0.6328000	0.6328000	1.0229516	0.8120972	0.8409660	y
5	1.0450693	0.9459200	1.6698046	1.2689039	1.3225238	y
10	8.137554	7.4819863	12.882727	9.8866789	10.286722	y
15	59.74360	55.006224	94.759007	72.717284	75.661850	y
20	439.0230	404.81975	698.63138	535.69606	557.45934	y

Table 2: Método de Adams-Bashforth com lista e previsão

nsteps	Lista	Euler	Euler Inv.	Euler Apr.	Runge-Kutta	Exata
1	0.1000000	0.1000000	0.1300000	0.1149999	0.1171999	y
2	0.2300000	0.2300000	0.3188000	0.2732000	0.2797378	y
3	0.4020000	0.4020000	0.5993280	0.4953360	0.5099075	y
4	0.6328000	0.6328000	1.0229516	0.8120972	0.8409660	y
5	1.0450693	0.9459200	1.6698046	1.2689039	1.3225238	y
10	8.184077	7.4980668	12.880401	9.8972066	10.295895	y
15	60.18770	55.120684	94.900333	72.851521	75.798150	y
20	443.5840	406.19325	700.10897	537.19928	558.97073	y

Table 3: Método de Adams-Multon com lista e previsão

nsteps	Lista	Euler	Euler Inv.	Euler Apr.	Runge-Kutta	Exata
1	0.1000000	0.1000000	0.1300000	0.1149999	0.1171999	y
2	0.2300000	0.2300000	0.3188000	0.2732000	0.2797378	y
3	0.4020000	0.4020000	0.5993280	0.4953360	0.5099075	y
4	0.6328000	0.6328000	1.0229516	0.8120972	0.8409660	y
5	0.9913990	0.9459200	1.6698046	1.2689039	1.3225238	y
10	7.8660673	7.3316159	13.086945	9.8706402	10.518379	y
15	57.884391	53.928635	96.406430	72.670162	79.692108	y
20	426.79758	397.46606	711.30632	535.93831	604.89600	y

Table 4: Método da Diferenciação Inversa com lista e previsão

Com estes resultados podemos ver quais são os métodos mais rápidos e confirmar nossas suspeitas a respeito dos erros de trucamentos dos métodos que não demonstramos. Como um apelo gráfico, mostraremos um código que utilizará a biblioteca matplotlib[2] para fazer um comparativo de cada método em um gráfico:

[2] `https://matplotlib.org/`

Listing 18: Plotando o gráfico

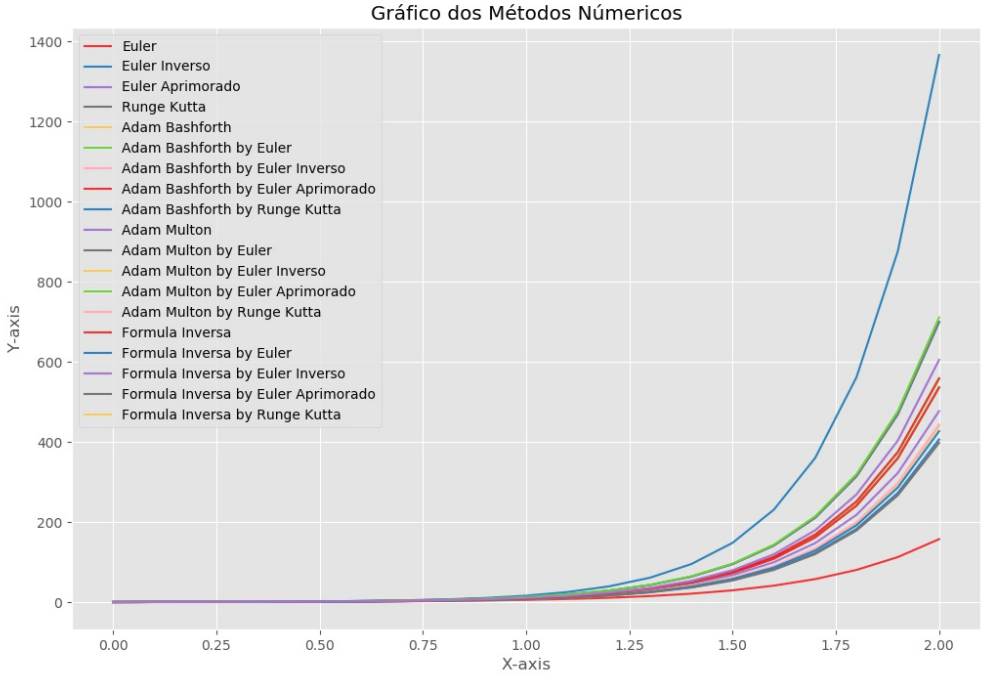
```
import matplotlib.pyplot as plt

plt.figure(figsize=(12,8))
plt.style.use('ggplot')

Ao final de cada metodo apresentado inclua:
    _t = [x[0] for x in res]
    _y = [x[1] for x in res]
    plt.plot(_t, _y, label='Nome_do_Metodo')

def _show():
        _str = 'Metodos.jpg'
    plt.title('Grafico dos Metodos Numericos')
    plt.xlabel('X-axis')
    plt.ylabel('Y-axis')
    plt.savefig(_str)
```

Esse código gerará a seguinte imagem:



References

[1] Boyce, Willian E. 10ª edição, Elementary Differential Equations and Boundary Value Problems

[2] Conte, S.D. 1ª edição, Elementos de Análise Numérica, Editora Globo: 1972

[3] https://ocw.mit.edu/courses/mechanical-engineering/2-086-numerical-computation-for-mechanical-engineers-fall-2012/readings/MIT2_086F12_notes_unit4.pdf

[4] https://www.ufrgs.br/reamat/CalculoNumerico/livro-oct/pdvi-metodo_de_adams-bashforth.html

[5] https://www.ufrgs.br/reamat/CalculoNumerico/livro-oct/pdvi-metodo_de_adams-moulton.html