# Relatório do Exercício de Métodos Numéricos

Pedro Henrique Rodrigues Meira Bastos

01 de Maio de 2019

# Sumário

# 1 Resumo

Esse documento relata a resolução do exercício proposto no dia 25/04/2019 para o processo seletivo do Laboratório de Computação Ciêntifica e Visualização da Universidade Federal de Alagoas (LCCV-UFAL). O exercício pede que o candidato implemente uma série de métodos numéricos em Python, e compare-os com os implementações correspondentes fornecidas por dois módulos famosos da linguagem: o NumPy e o SciPy.

# 2 Objetivos

Implementar um número de métodos numéricos na linguagem Python, e comparar tais implementações com as suas correspondentes nos módulos NumPy e SciPy, no tocante de resultados obtidos e tempo de execução.

# 3 Introdução

É óbvio que em praticamente todos os ramos das ciências exatas e engenharias existe a necessidade de lidar com cálculos e contas, porém alguns ramos em particular vão lidar com mais cálculos que a maioria, seja por questão de quantidade ou complexidade de contas (ou até mesmo os dois), áreas que vão envolver desde soluções de imensos sistemas lineares até utilização intensa do método dos elementos finitos, e uma infinidade de possibilidades no meio do caminho.

Em outras palavras, vão existir situações onde a quantidade e/ou complexidade das contas será tão grande que a resolução humana fica muito dificultada, problemas esses que demorariam tempo demais para serem resolvidos "na mão", que também ficariam sujeitas ao erro humano.

É aí que entram o cálculo numérico e a computação. Cálculo numérico é o estudo dos métodos numéricos, que são, a rigor, algoritmos para a resolução de problemas geralmente mais complexos, através da busca de soluções aproximadas, o que, a primeiro momento pode gerar uma certa desconfiança, afinal uma solução aproximada sempre terá uma certa diferença comparada à exata.

Nesse momento que a computação suplanta: esses algoritmos podem ser facilmente reproduzidos em linguagem de programação, permitindo que os computadores realizem os cálculos necessários dentro de critérios bastante rigorosos de precisão, colocando qualquer diferença entre resultados aproximado e exato dentro de uma margem de erro segura para a dada aplicação.

Dito isso, é fácil conjecturar que esses dois recursos são amplamente utilizados na comunidade científica e no meio profissional, e realmente são, na verdade, são tão utilizados, que na linguagem Python (e certamente em outras linguagens), existem vários módulos (também chamados de bibliotecas) que oferecem a implementação de várias funções científicas e estatísticas, sendo que muitas dessas implementações utilizam métodos numéricos "por debaixo dos panos", e algumas dessa bibliotecas até oferecem a implementação direta de tais métodos.

No exercício proposto, serão analisadas as implementações dos populares módulos NumPy e SciPy para os seguintes situações:

- Resolução de sistemas de equações lineares
- Interpolação

- Ajuste
- Integração Numérica

Um fato interessante dessas duas bibliotecas é que seus rescpectivos *backends* foram desenolvidos majoritariamente nas linguagens C e FORTRAN, ambas de baixo nível, especialmente quando comparadas ao Python, isso significa que espera-se tempos de execução menores usando suas funções do que implementações diretas em Python "puro".

A análise será feita a partir da comparação entre as soluções que essas biblieotecas oferecem para os problemas acima e a implementação manual de métodos numéricos (que resolvem tais problemas também) em Python. O detalhamento da fundamentação teórica por trás de tais métodos está fora do escopo do exercício, ficando apenas umas resumidas explicações nesse relatório. Caso o leitor julgue necessário para o seu entedimento um aprofundamento maior, ele pode consultar a bibliografia mecionada no final do documento.

# 4 Metodologia

A implementação, execução e avaliação dos métodos foram todas feitas na *IDE Spyder* (versão 3.2.6) a versão do Python foi a 3.6.7, e os seguintes módulos foram usados:

- NumPy versão 1.16.2
- SciPy versão 1.2.1
- time (*built-in* no próprio Python)
- math (*built-in* no próprio Python)

Por mais que o Python seja uma linguagem que suporte orientação a objeto, decidiu-se implementar os métodos por programação estruturada, escrevendo-os na forma de "funções soltas", assim esses métodos poderão ser portados para dentro de um projeto mais complexo de forma simples e modular, caso seja necessário.

## 4.1 Implementação dos métodos

Os métodos pedidos pelo exercício foram:

- Gauss-Seidel para resolução de sistemas de equações lineares (e Newton-Raphson como extra)
- Lagrange para interplo]ação
- Método dos Mínimos Quadrados para ajuste (de polinômios até do quarto grau)
- Regra do Trapźeio para integração numérica (para polinômios de sexto grau)

### Os métodos de Gauss-Seidel e Newton-Raphson

O método de Gauss-Seidel é um método iterativo para a resolução de sistemas de equações lineares, isto é, seu algoritmo é rodado várias vezes, cada vez usando retornando uma resposta diferente, e cada iteração utilizando o resultado obtido na anterior. Isso signfica que, quanto maior o número de iterações, mais preciso será o resultado final.

Um sistema de equações lineares geralmente é mostrado da seguinte forma:

$$A * X = B \tag{1}$$

Onde A é a matriz dos coeficientes das equações que compõem o sistema, B é a matriz dos radicais livres, e X a matriz das raízes. Ou seja, montado o sistema, queremos achar o valor da matriz X.

O algoritmo recebe as matrizes $A$, $B$, e uma matriz $X0$, com uma estimativa inicial para os valores da matriz X (um "chute inicial"por assim dizer), e começa a rodar, parando de se repetir quando a norma da diferença entre a vetor resposta das iterações atual e anterior for menor que a tolerância estabelecida, ou então quando o número máximo de iterações for atingido. Quanto menor for a tolerância e maior for o limite de iterações, mais precisa será a resposta.

O algoritmo, em Python, ficou da forma abaixo:

```
def gauss_seidel (a, b, tol, nitemax, x0):
    nite = 0
    dist = 0
    size = len(a)
    x = x0
    s = [0] * 2
    while dist < tol and nite < nitemax:
        nite = nite+1
        for i in range (size):
            s[0] = b[i]
            s[1] = 0
            for j in range (i):
                s[0] = s[0] - a[i][j] * x[j]
            for j in range (i+1, size):
                s[1] = s[1] - a[i][j] * x0[j]
            x[i] = (s[0]+s[1])/a[i][i]
        aux = [x[i]-x0[i] for i in range(len(a))]
        dist = pow(sum(x**2 for x in aux),0.5)
        x0 = x
    return x0
```

Por outro lado, nem todas os sistemas lineares vão funcionar com método de Gauss-Seidel, ou pelo menos nem todas terão garantia que irão funcionar. Para saber se a solução aproximada pelo método vai convergir para a exata, é necessário que a matriz $A$ atenda pelo menos um de três critérios de convergência, isso será abordado um pouco mais na frente, no momento em que a avaliação do método for discutida.

Como questão extra, foi pedida a implementação do método de Newton-Raphson, um método numérico para resolver sistemas de equações não-lineares. Também é um método iterativo, que vai saber uma matriz $F$ que representa as funções não lineares (não apenas os coeficientes), uma matriz $X0$ que recebe o chamado "chute inicial", uma tolerância ($\epsilon$) e um número máximo de iterações.

O método roda enquanto a norma do sistema avaliado no resposta da iteração atual for menor que a tolerância ($\|F(X)\| < \epsilon$) ou então até o número máximo de iterações ser batido. O método em si é bem simples de aplicar, exceto pelo fato que, durante a sua aplicação, é calculada a jacobiana do conjunto das funções que formam o sistema, então essa funcionalidade teve que ser implementada primeiro, conforme o código abaixo:

```
def partialDifferenceQuotient (f, v, i, h):
    "retorna a derivada parcial no ponto"
    w = [vj + (h if j==i else 0) for j, vj in enumerate(v)]

    return (f(w) - f(v))/h

def estimateGradient(f, v, h = 1e-10):
    "usando a outra funcao, estima o gradiente"
    return [partialDifferenceQuotient(f,v,i,h) for i, _ in enumerate(v)]


def jacobian (functions, variables):
    "monta a jacobiana"
    j = list()
    size = len(functions)
    [j.append(estimateGradient(functions[x], variables)) for x in range (size)]
    return j
```

Além disso, a aplicação do método envolve a resolução de um sistema linear. O método de Gauss-Seidel até poderia ser usado, mas preferiu-se implementar o método da eliminação de Gauss (com pivoteamento) por ser um método direto, que dá a resposta exata, pois usar uma aproximação dentro do que já é uma aproximação pode acabar afastando a resposta aproximada do desejado. Além disso, o método de Gauss-Seidel só garante resultados convergentes se a matriz A do sistema atender um de seus três critérios, o que limita a sua usabilidade. O método de Gauss ficou assim:

```
def gauss(A, B):
    a = [list(x) for x in A]
    b = list(B)

    size = len (a)
    answer = [0] * size

    #primeira etapa do metodo, a eliminacao
    for k in range(size):
        if (abs(a[k][k]) < 1):
            auxmax = a[k][k]
            auxcount = k
            for m in range (k+1,size):
                if (abs(auxmax) < abs(a[m][k])):
                    auxcount = m
                    auxmax = a[m][k]

            auxrow = a[k]
            a[k] = a[auxcount]
            a[auxcount] = auxrow

            bauxrow = b[k]
            b[k] = b[auxcount]
            b[auxcount] = bauxrow

    for j in range (size):
        for i in range (j+1,size):
            aux = a[i][j]/a[j][j]
            for k in range (size):
                a[i][k] = a[i][k] - aux*a[j][k]
```

```
            b[i] = b[i] - aux*b[j]

    #segunda etapa do metodo, a resolucao do sistema triangular
    answer[size-1] = b[size-1]/a[size-1][size-1]
    for k in range (size-1, -1, -1):
        s = 0
        for j in range (k+1, size):
            s = s + a[k][j] * answer[j]
        answer[k] = (b[k] - s)/a[k][k]

    return answer
```

Por fim, foram implementadas algumas funções, que apesar de não serem essenciais para a solução do problema, ajudaram a deixar o código mais limpo e legível:

```
from math import sqrt

def evaluate(funcMatrix, values):
    size = len(funcMatrix)
    answer = [0] * size
    for x in range(size):
        answer[x]  = funcMatrix[x](values)
    return answer

def norm(x):
    return sqrt(sum(pow(i,2) for i in x))

def sumMatrices(a,b):
    size = len(a)
    result = [0]*size
    for i in range (size):
        result[i] = a[i] + b[i]
    return result
```

Finalmente, o método de Newton-Raphson ficou implementado da seguinte forma:

```
from gauss import gauss
from jacobiana import jacobian
from support import evaluate, norm, sumMatrices


def newtonRaphson(funcMatrix, x0, epsilon=0.0001, niteMax=10):
    """o metodo de newton raphson"""
    xk = [0] * (niteMax+1)
    xk[0] = x0

    for k in range(niteMax):
        if norm(evaluate(funcMatrix, xk[k])) < epsilon:
            break
        else:
            fxk = [i*-1 for  i in evaluate(funcMatrix,xk[k])]
            jxk = jacobian(funcMatrix, xk[k])
            dx = gauss(jxk, fxk)
            xk[k+1] = sumMatrices(xk[k], dx)

    return xk[k]
```

## O método de Lagrange

O método de Lagrange é um método para a interpolação de um conjunto de pontos discretos (pontos $x_i$ e seus correspondentes $f(x_i)$). Todo polinômio formado a partir da interpolação de Lagrange tem a seginte forma:

$$p(x) = f(x_1)L_1(x) + f(x_2)L_2(x) + ... + f(x_n)L_n(x) \tag{2}$$

Sendo p(x) o chamado *polinômio de Lagrange*, e $L_i(x)$ é definido como:

$$L_i = \prod_{k=1,k\neq i}^{n} \frac{x - x_k}{x_i - x_k} \tag{3}$$

Sua implementação é simples, ficando assim:

```
def lagrange (points, fpoints):
    """interpolacao pelo metodo de lagrange"""
    size = len(points)
    def p(x):
        p = 0
        for i in range (size):
            s = 1
            for j in range (size):
                if j != i:
                    s = (s * (x-points[j]))/(points[i]-points[j])
            p = p + fpoints[i]*s
        return p
    return p
```

No caso, a função está retornando o polinômio em forma de uma função (no Python, uma função pode retornar outra função), e não apenas o conjunto dos coeficientes, podendendo então ser usaa imediatamente.

### O Método dos Mínimos Quadrados

Dado um conjunto de pontos discretos (pontos $x_i$ e seus correspondentes $f(x_i)$), o método dos mínimos quadrados é um método iterativo que busca realizar o ajuste desses pontos. A equação do ajuste vai ter a forma:

$$q(x) = a_1 g_1(x) + a_{2g2}(x) + ... + a_n g_n(x) \tag{4}$$

Sendo que $q(x)$ será o mais próximo possível da função exata $f(x)$ O método faz isso escolhendo os coeficientes $a_i$ de forma que a soma dos quadrados dos desvios seja mínima:

$$S = \sum_{k=1}^{m} d_k^2 = \sum_{k=1}^{m} (f(x_k) - q(x_k))^2 \tag{5}$$

A execução do método é simples, porém, durante sua aplicação é necessária a resolução de um sistema linear de equações. Para isso, usou-se a mesma implementação do método de eliminação de Gauss utilizada na implementação do método de Newton-Raphson (e pelo mesmo motivo). A implementação do método dos mínimos quadrados, em Python, ficou da seguinte forma:

```
from gauss import gauss

def minimosQuadrados (points, fpoints, power = 4):

    n = int(power) + 1
    m = len(points)

    matrixA = [[0 for col in range(n)] for row in range(n)]
    vectorB = [0] * n

    xks = [0] * ((2*power)+1)
    xks[0] = m

    #montando os valores que vao entrar na matriz A
    for x in range (1, 2*power+1):
        xks[x] = sum([pow(i,x) for i in points])


    for x in range(n):
        #colocando os valores achados na matriz A
        for y in range(n):
            matrixA[x][y] = xks[x+y]

    #montando o vetor B
    for x in range (n):
        vectorB[x] = sum([pow(points[j],x)*fpoints[j] for j in range(m)])

    auxA = tuple([tuple(x) for x in matrixA])
    auxB = tuple(vectorB)

    coefs = gauss(auxA, auxB)

    def func(x):
        func = 0
        for count in range(power):
            func = func + coefs[count] * pow(x,count)
        return func

    return func, coefs
```

Aqui, o método irá retornar uma tupla contendo a função aproximada $q(x)$ e além disso, também retornará os coeficientes dessa função.

## A Regra do Trapézio

De todos os métodos, este foi de longe o mais simples de entender e implementar. A regra do trapézio é um método que, dado um intervalo de $n$ números, começando de $a$ e terminando em $b$, estimará o valor da integral de uma dada função nesse intervalo $[a, b]$:

$$\int_a^b f(x)dx = \frac{h}{2}[f(x_0) + 2\sum_{i=1}^{n-1} f(x_i) + f(x_n)] \tag{6}$$

Sendo $h$ a distância de um elemento $x_i$ dentro do intervalo para outro (é bom frisar que a distância de um ponto a outro é sempre a mesma). A implementação em Python fica da seguinte forma:

```
def trapezio (a, b, n, func):
    h = (b-a)/n
    Sum = 0
    for count in range (1,n):
        aux = a + h*(count-1)
```

```
        Sum = Sum + func(aux)
    return (h/2)*((2*Sum) + func(a) + func(b))
```

## 4.2 Avaliação dos métodos

Para cada método avaliado, foi elaborado uma função chamada *benchmark*, que importa o método implementado manualmente, seu correspondente no numpy e no scipy, e os roda dentro dos mesmos parametros, retornando então os resultados e o tempo de execução de cada um. Então, essa função é rodada três vezes, e dela são tiradas as médias dos tempos em cada execução. Obviamente, cada função benchmark vai ser diferente da outra, pois métodos diferentes exigem parâmetros diferentes para serem testados.

### Gauss-Seidel vs. NumPy

Aqui a função *benchmark* vai comparar o método de Gauss-Seidel feito à mão com o método *linalg.solve* do NumPy. Óbvio que, dada a natureza do exercício, que a comparação tem que ser o menos enviesada possível, para isso, as matrizes $A$ e $B$ são geradas aleatoriamente, ou quase que aleatoriamente.

A questão é que, como dito antes, nem todo sistema poderá ser resolvido pelo método de Gauss-Seidel, pelo menos não com respostas confiáveis. O único jeito de saber se um sistema será "compatível"com o método é conferindo se a matriz $A$ vai satisfazer um de três critérios: *o das linhas, das colonas ou o de Sassenfeld*. Como só é necessário que um dos critérios seja antendido, será abordado o critério das linhas, julgado o mais simples de entedimento e implementação.

O critério das linhas exige, basicamente, que cada elemento da diagonal principal da matriz $A$ seja maior que a soma dos outros elementos de sua respectiva linha. Dito isso, para que a análise seja justa, foi criado uma função, de caráter auxiliar, que gere uma matriz $A$ de números aleatórios que sempre vai atender esse critério.

A função basicamente gera uma matriz do tamanho desejado de valores aleatórios (usando a função *random.rand* do NumPy), e então perocorre a diagonal principal de tal matriz, substituindo os valores nela pela soma dos módulos de todos os elementos da respectiva linha (incluindo do próprio elemento a ser trocado), assim, os elementos da diagonal principal sempre

serão maiores que a soma do resto da linha. Em Python, a função ficou da seguinte forma:

```
def getMatrix (size = 20):
    """gera uma matriz que sempre vai convergir conforme o criterio das linhas"""
    matrix = np.random.rand(size,size)
    for x in range(size):
        matrix[x][x] = sum(map(abs,matrix[x]))
    return matrix
```

Como dito e visto, usou-se o NumPy para gerar os valores aleatórios, mas isso não enviesará a comparação pois o tempo de geração da matriz não será contado, apenas o tempo de execução dos métodos em si serão contabilizados, como mostra o código abaixo:

```
def benchmark(testeA, testeB):

    startNumPy = time()
    respostaNumPy = np.linalg.solve(testeA,testeB)
    endNumPy = time()

    execNumPy = endNumPy - startNumPy

    chute = np.zeros(len(testeB))
    startGauss = time()
    respostaGauss = gauss_seidel.gauss_seidel(testeA,testeB, 0.1, 10, chute)
    endGauss = time()

    execGauss = endGauss - startGauss

    return respostaNumPy, execNumPy, respostaGauss, execGauss
```

Como pode-se ver, as matrizes tem que ser geradas por fora, e então jogadas como argumento para a função de comparação. Para contar o tempo foi utilizada a função *time* do módulo *time* do próprio Python.

A matriz $B$ do sistema não precisa ser avaliada no tocante dos critérios citados anteriormente, então ela pode ser gerada aleatoriamente através da função *random.rand* do NumPy.

Como adendo, foram até feitas funções que, dada uma matriz, checam se ela vai convergir de acordo com os critérios do método, mas elas acabaram que sendo depreciadas, pois a solução da função *getMatrix* acima é muito mais prática e elegante do que ficar checando matriz aleatória por matriz aleatória até se chegar em uma matriz que vá convergir.

Por fim, pode ser visto que para o método de Gauss-Seidel, a tolerância adotada foi de 0.1, o limite de iterações foi 10 e o chute inicial foi uma matriz de zeros. Como especificado pelo exercício, foi usada um sistema de 20x20 para o teste.

### Lagrange vs. SciPy

A implementação do *benchmark* para o método de Lagrange foi mais díficil que a implementação do próprio método, pois o exercício pedia que, para essa

avaliação, sejam interpoladas as funções posições de uma matriz quadrilateral quadrática, que vão ter a seguinte formulação:

$$N_{ij}(x;y) = N_i(x) * N_j(y) \tag{7}$$

A implementação da formação da matriz ficou da seguinte forma:

```
def produtoDeFuncao (f,g):
    """pega uma funcao f(x) e g(y) e obtem fg(x,y) = f(x)*g(y)"""
    def fg(x,y):
        return f(x)*g(y)
    return fg

def quadrilateralQuadratico (funcList):
    """elabora a matriz quadrilateral quadratica"""
    size = len(funcList)
    functions = [[0 for col in range(size)] for row in range(size)]
    for x in range (size):
        for y in range(size):
            functions[x][y] = produtoDeFuncao(funcList[x],funcList[y])
    return functions
```

No caso, a funcão *quadrilateralQuadratico* vai retornar a propria matriz desejada, recebendo uma lista de funções que serão formadas a partir da interpolação por Lagrange, nas versões do SciPy (*interpolate.lagrange*) e feita à mão.

A função *benchmark* e suas auxiliares vão ficar da seguinte forma:

```
from lagrange import lagrange as pyLagrange
from quadrilateral_quadratico import quadrilateralQuadratico as quadLat
from scipy.interpolate import lagrange as spLagrange
from time import time
import numpy as np

def getValues(npoints, nfunctions):
    """cria os valores a serem usados para a interpola  o randomicamente"""
    values = list()
    for x in range (nfunctions):
        values.append([np.ravel(np.random.rand(1,npoints)), np.ravel(np.random.rand(1,npoints))])
    return values


def lagrangeList(values):
    """interpola os valores dados, usando o metodo de lagrange feito a mao, retornado uma lista de funcoes"""
    size = len(values)
    funcList = list()

    for x in range(size):
        funcList.append(pyLagrange(values[x][0],values[x][1]))

    return funcList

def SciPyList(values):
    """interpola os valores dados, usando o metodo de lagrange do SciPy, retornado uma lista de funcoes"""
    size = len(values)
    funcList = list()

    for x in range(size):
        funcList.append(spLagrange(values[x][0],values[x][1]))

    return funcList

def benchmark (valores):
    "teste dos dois metodos, passando como argumentos os valores de x e y e retornando as matrizes e o tempo"

    startSp = time()
    funcSp = SciPyList(valores)
    matrixSp = quadLat(funcSp)
```

```
    endSp = time()
    execSp = endSp - startSp

    startPy = time()
    funcPy = lagrangeList(valores)
    matrixPy = quadLat(funcPy)
    endPy = time()
    execPy = endPy - startPy

    return matrixPy, execPy, matrixSp, execSp
```

No caso, foram gerados três conjuntos de números através da *random.rand* do NumPy, que foram formatadas no através da função *ravel*, também do NumPy (sem essa formatação os métodos não conseguiam acessar os valores corretamente, gerando inúmeros problemas), e deles obtidas três funções, cujas combinações acabam formando uma matriz 3x3 de funções $N_{ij}$.

### Mínimos Quadrados vs. SciPy

O método dos mínimos quadrados será comparado com a função *optimize.curvefit* do SciPy, que recebe além dos $x_i$ e $f(x_i)$ uma função "genérica"à qual esses pontos serão ajustados. Tirando isso, a implementação do *benchmark* será bem direta, como mostra o código abaixo:

```
def func(x, a, b, c, d, e):
    """o scipy exige que seja dada uma funcao para a qual sera feito o ajuste"""
    return a + b*x + (c*pow(x,2)) + (d*pow(x,3)) + (e*pow(x,4))

def benchmark(xvalues, yvalues):

    startPy = time()
    funcPy = minQuad(xvalues, yvalues, 4)
    endPy = time()
    execPy = endPy - startPy


    startSciPy = time()
    funcScipy = curve_fit(func, xvalues,yvalues)
    endSciPy = time()
    execSciPy = endSciPy - startSciPy


    return funcPy, execPy, funcScipy, execSciPy
```

Como o método dos mínimos quadrados foi implementado de forma que só devolva polinômios, montou-se um polinômio genérico para ser usado como parâmetro da função do SciPy, de quarto grau porque foi o exigido pelo exercício.

### Regra do Trapézio vs. SciPy

A regra do trapézio foi o método mais simples de implementar, e também o mais simples para elaborar seu respectivo *benchmark* contra a função optimize.quad do SciPy, com apenas uma observação: a regra do trapézio, para

retornar um valor bem próximo do exato, precisa de um intervalo com muitas divisões, isto é, quanto maior for o $n$ estabelecido, maior será a precisão desejada.

Dito isso, para esse benchmark a função do método do trapézio feita à mão será rodada repetidas vezes, cada vez com um $n$ maior (partindo de $n = 10$) até que seja batida uma tolerância estipulada (ou o limite máximo de 1000 para $n$, para que não demore demais). O raciocínio por trás disso é que a ideia é contabilizar o tempo que as duas funções (do SciPy e feita à mão) vão retornar um valor próximo do exato, pois de pouco adianta rodar a regra do trapézio de forma que ela seja executada rapidamente, se o resultado obtido vai ter um desvio muito grande do exato.

Então, a função *benchmark* e suas auxiliares vão ficar da seguinte forma:

```
from trapezio import trapezio
from scipy.integrate import quad
from numpy.random import randint
from time import time

def teste (x):
    return pow(x,6)

def IntTest (a,b):
    return (pow(b,7)/7) - (pow(a,7)/7)

def trapezioConvergente (a, b, tol = 0.001, func = teste, intFunc = IntTest):
    count = 10
    while (intFunc(a,b) - trapezio(a,b, count, func)) > tol and count < 1000:
        count +=1
    return trapezio(a,b,count,func), count

def benchmark (a, b, tol = 0.001, func = teste, intFunc = IntTest):

    startPy = time()
    intPy = trapezioConvergente (a,b)
    endPy = time()
    execPy = endPy - startPy

    startSciPy = time()
    intSciPy = quad(func,a,b)
    endSciPy = time()
    execSciPy = endSciPy - startSciPy

    return intPy, execPy, intSciPy, execSciPy
```

Foi escolhido um simpĺes polinômio de grau seis, pois essa potência foi a exigida pelo exercício, e para gerar os números aleatórios do intervalo foi utilizada a função *random.randint* do NumPy, que gera valores inteiros aleatórios dentro de um intervalo (o limite inferior foi um número de 0 a 10, e o superior um de 10 a 100), pois se fosse utilizada uma função completamente randômica como o *random.rand*, poderiam ser utilizados intervalos problemáticos, com números muito pequenos que acabariam colocando a resposta fora do escopo do exercício (cálculo de área abaixo de uma curva), ou então de difícil intepretação.

# 5 Resultados e Discussões

## 5.1 Gauss-Seidel vs. NumPy

Abaixo estão as tabelas com os valores obtidos pelos dois métodos (em cada uma das três amostragens) e a média dos tempos de execução:

| NumPy | Gauss-Seidel | NumPy | Gauss-Seidel | NumPy | Gauss-Seidel |
|---|---|---|---|---|---|
| 0.034614 | 0.034614 | 0.034614 | 0.034614 | 0.034614 | 0.034614 |
| 0.000157638 | 0.000157586 | 0.000157638 | 0.000157586 | 0.000157638 | 0.000157586 |
| -0.0155427 | -0.0155427 | -0.0155427 | -0.0155427 | -0.0155427 | -0.0155427 |
| 0.0257738 | 0.0257738 | 0.0257738 | 0.0257738 | 0.0257738 | 0.0257738 |
| 0.0440241 | 0.044024 | 0.0440241 | 0.044024 | 0.0440241 | 0.044024 |
| 0.0667933 | 0.0667933 | 0.0667933 | 0.0667933 | 0.0667933 | 0.0667933 |
| -0.0157441 | -0.0157441 | -0.0157441 | -0.0157441 | -0.0157441 | -0.0157441 |
| 0.0366918 | 0.0366918 | 0.0366918 | 0.0366918 | 0.0366918 | 0.0366918 |
| 0.0360901 | 0.0360901 | 0.0360901 | 0.0360901 | 0.0360901 | 0.0360901 |
| 0.0472584 | 0.0472584 | 0.0472584 | 0.0472584 | 0.0472584 | 0.0472584 |
| 0.0249728 | 0.0249728 | 0.0249728 | 0.0249728 | 0.0249728 | 0.0249728 |
| -0.00379812 | -0.00379811 | -0.00379812 | -0.00379811 | -0.00379812 | -0.00379811 |
| 0.0189125 | 0.0189125 | 0.0189125 | 0.0189125 | 0.0189125 | 0.0189125 |
| 0.0768572 | 0.0768572 | 0.0768572 | 0.0768572 | 0.0768572 | 0.0768572 |
| 0.00134259 | 0.0013426 | 0.00134259 | 0.0013426 | 0.00134259 | 0.0013426 |
| 0.00454912 | 0.00454913 | 0.00454912 | 0.00454913 | 0.00454912 | 0.00454913 |
| 0.0276731 | 0.0276731 | 0.0276731 | 0.0276731 | 0.0276731 | 0.0276731 |
| 0.132068 | 0.132068 | 0.132068 | 0.132068 | 0.132068 | 0.132068 |
| 0.0189683 | 0.0189683 | 0.0189683 | 0.0189683 | 0.0189683 | 0.0189683 |
| -0.0168402 | -0.0168402 | -0.0168402 | -0.0168402 | -0.0168402 | -0.0168402 |

Tabela 1: Resultados obtidos por cada amostragem

| NumPy | Gauss-Seidel |
|---|---|
| 0.02053697903951009 | 0.026414155960083008 |

Tabela 2: Tempo de execução médio de cada método (em segundos)

É fácil ver que os resultados foram bastante parecidos, e todas as amostras foram idênticas para cada método. Além disso, o método do NumPy, em média, foi um pouco mais rápido que o de Gauss-Seidel, lembrando que se diminuirmos a tolerância e aumentarmos o limite de iterações do método de Gauss-Seidel, a diferença de resposta vai ser ainda menor, porém a disparidade de tempo aumentará. Considerando isso, e que o método do NumPy pode aceitar qualquer sistema de equações lineares, além de ser mais prático de usar, pois ele "já vem pronto"o torna a melhor opção.

## 5.2 Lagrange vs. SciPy

Esse é um método um pouquinho mais complicado de se avaliar, pois o resultado retorna, além dos tempos de execução, duas matrizes 3x3 (uma feita com o método feito à mão e a outra com o método do SciPy), com cada um dos seus elementos sendo uma função $N_{ij}(x; y)$.

O que se fez foi pegar uma dessas funções (em cada matriz), e comparar o seu valor com o produto $N_i * N_j$ num mesmo ponto $x$, obtendo-se os seguintes resultados:

| Método | Diferença do valor original |
|---|---|
| À mão | 0 (deu igual) |
| SciPy | 2.220446049250313e-16 |

Tabela 3: Tabela comparando uma das funções das matrizes obtidas pelos métodos com o respectivo valor original de $f(x_i)$

Curiosamente, pode-se ver que o método feito à mão foi mais preciso que o do SciPy, apesar de que o "erro"do SciPy tenha sido minúsculo. Repetindo o teste com outros valores aleatórios apresentou resultados similares. O que vai supreender são os tempos de execução:

| Método | Tempo médio (segundos) |
|---|---|
| À mão | 1.3589859008789062e-05 |
| SciPy | 0.0011642773946126301 |

Tabela 4: Tempo médio de execução dos métodos de interpolação

É supreendente o quão a implementação manual foi mais rápida que a do SciPy, além de ser aparentemente mais precisa. Considerando também que esse é um método de fácil implementação, fica díficil recomendar o SciPy para a maioria dos casos que necessitem do método de Lagrange.

## 5.3 Mínimos Quadrados vs. SciPy

Ao final dos testes, foi visto que, de forma similar ao que ocorreu com o teste do Gauss-Seidel, os valores obtidos nas três amostragens foram iguais dentro dos mesmos métodos. Percebe-se aqui que os coeficientes retornados

| Feito à mão | NumPy |
|---|---|
| -2.049578806620393 | -2.04958 |
| 48.14835949943654 | 48.1484 |
| -279.86012607065993 | -279.86 |
| 507.5468623477958 | 507.547 |
| -284.1348859341819 | -284.135 |

Tabela 5: Valores dos coeficientes obtidos pelas amostragens dos testes de ajuste

pelo SciPy estão arredondados. Tirando isso, não há diferenças de valor, dependendo a aplicação o método feito à possa ser mais interessante, já que ele devolve mais casas decimais, mas certamente o SciPy permite esse tipo de precisãoo também.

| Método | Tempo médio (segundos) |
|---|---|
| À mão | 0.00011340777079264323 |
| SciPy | 0.0010484059651692708 |

Tabela 6: Tempo médio de execução dos métodos de ajuste

Novamente temos uma surpresa: o método feito a mão provou-se mais rápido que o do SciPy (porém a diferença de tempo foi menor dessa vez), por outro lado, o SciPy tem uma vantagem bastante interessante que é a possibilidade de fazer ajustes à outros tipos de curva além da polinomial. Então, pode-se dizer que o SciPy, pela sua versatilidade, tem a maior aplicabilidade.

### 5.4 Regra do Trapézio vs. SciPy

Novamente, os resultados obtidos foram idênticos de um amostra pra outra, variando apenas o tempo de execução:

| Método | Resposta | Tempo de execução médio (segundos) |
|---|---|---|
| Feito à mão | 397449860893.05475 | 0.20513558387756348 |
| SciPy | 399908453779.5714 | 3.0835469563802086e-05 |

Tabela 7: Resultados numéricos e tempos de execução dos métodos de integração numérica

Essa talvez seja a avaliação mais subjetiva de todas, pois os resultados variavam muito a cada rodada de três amostragens, especialmente em relação ao método feito à mão, que ora apresentava resultado convergente, ora batia o limite estabelicdo para $n$. Além disso, o SciPy foi visivelmente mais rápido, e constantemente entregou resultados mais precisos. Sem sombra de dúvidas, a implementação do SciPy pode ser considerada a melhor aqui, por grandes margens.

## 6 Conclusão

Ao final do experimento, percebeu-se que o uso dessas bibliotecas, por mais que à rigor não seja obrigatório, é de grande ajuda para qualquer desenvolvedor, seja no meio ciêntifico ou no mercado, pois, acima de tudo, essa funções oferecem conveniência e uniformidade: numa situação realista, onde um projeto é desenvolvido por equipes de um variado número de pessoas, poder contar com métodos constantes que todos tenham fácil acesso e que apresentem resultados constantes é de fundamental ajuda.

Por outro lado, as bibliotecas acabam tendo um caráter "caixa preta": o usuário não se preocupa muito com o processo em si, ele entra com os valores, e recebe os resultados como retorno. Certamente vão existir aplicações onde esse aspecto das bibliotecas não será muito bem vindo, tornando a implementação manual dos métodos preferencial, mas para a maioria dos outros casos, as bibliotecas NumPy e SciPy oferecem praticidade, velocidade e uniformidade para os desenvolvedores.

# 7 Referências


- Documentação do SciPy e NumPy (link)
- RUGGIERO, Márcia, LOPES, Vera. *Cálculo Numérico – Aspectos Teóricos e Computacionais*. 2ª edição. São Paulo: Makron Books do Brasil. 1997
- BARROSO, Leônidas, BARROSSO, Magali, FILHO, Frederico, CARVALHO, Márcio, MAIA, Miriam. *Cálculo Numérico (com aplicações)*. 2ª edição. São Paulo: Harbra. 1987


# 8 Anexos

Repositório no Github com os códigos (link)