

# Comentários acerca das questões de métodos numéricos

Bruno Felipe Firmino de Souza (brunofelip940@gmail.com)

Graduando em Engenharia Civil - UFAL

## Descrição e discussão dos códigos em python elaborados:

---

01) Programação em python do método de Gauss-Seidel para uma matriz de ordem n:

i) Procedimentos utilizados na programação do código:

Inicialmente foi necessário gerar matrizes de coeficientes do sistema linear dos mais variados tipos e de ordem  $n > 1$ , foi elaborado, então, a função **matconverg**(n,Linf,Lsup), em que  $n$  é a ordem da matriz (A) do sistema linear (matriz de coeficientes), onde um único vetor resposta  $x = [x_0, x_1, x_2, \dots, x_n]$  é obtido, além disso, é garantido que seja único, pelo critério de sassenfeld (análise de convergência). *Linf* e *Lsup* representam o intervalo de números inteiros em que os elementos de A fora da diagonal principal irão assumir aleatoriamente.

A ideia da montagem da matriz se baseia na seguinte condição: o critério de sassenfeld estabelece os  $\beta(i)$  ( $1 \leq i \leq n$ ), onde o valor de cada  $\beta$  deve ser inferior a 1, além disso  $\beta$  depende do elemento da diagonal principal de sua linha. Assim, podemos supor um problema reverso, dado um conjunto de  $\beta(i)$  ( $1 \leq i \leq n$ ) com valores aleatórios, sendo eles inferiores a 1 e os elementos da linha i da matriz (fora da diagonal principal), então podemos obter o valor do elemento da diagonal principal daquela linha em função desses elementos. Portanto, matematicamente, conforme descrito no livro *Cálculo Numérico - Aspectos Teóricos e Computacionais - 2ª Edição*, teremos:

$$\text{Equação 01: } \beta_1 = \frac{|a_{12}| + |a_{13}| + \dots + |a_{1n}|}{|a_{11}|} \Rightarrow |a_{11}| = \frac{|a_{12}| + |a_{13}| + \dots + |a_{1n}|}{\beta_1}$$

Para qualquer  $\beta(i)$  da linha da matriz A:

$$\text{Equação 02: } \beta_i = \frac{|a_{i1}|\beta_1 + |a_{i2}|\beta_2 + \dots + |a_{ii-1}|\beta_{i-1} + |a_{ii+1}| + \dots + |a_{in}|}{|a_{ii}|} \Rightarrow a_{ii} = \frac{|a_{i1}|\beta_1 + |a_{i2}|\beta_2 + \dots + |a_{ii-1}|\beta_{i-1} + |a_{ii+1}| + \dots + |a_{in}|}{\beta_i}$$

Portanto, primeiramente, importamos o random, definimos a matriz A por matQ, criamos uma lista de elementos nulos de ordem  $n \times n$ , criamos um vetor dos valores de  $\beta$  (denominado bet, de tamanho  $1 \times n$ ), além disso, a operação acima descrita para obtenção dos elementos da diagonal principal pode ser transformada em duas matrizes, uma matriz que apresenta os elementos da diagonal principal nulos (no entanto, esse fato já é levado em consideração na matriz beta, logo a diagonal principal, em tese, pode ser de qualquer valor inicial), pelo fato do zero se tratar de um elemento neutro da adição, e com os elementos fora da diagonal principal apresentando algum valor do conjunto dos inteiros e a matriz de beta (denominado beta, de ordem n). Por fim é criado o vetor soma (denominado somE, com tamanho  $1 \times n$ ), onde ele representa o numerador de cada equação de  $\beta_i$  ou da equação  $a_{ii}$ , onde ocorre a soma de cada elemento da matriz A multiplicado pelos seus respectivos betas. Para melhor visualização, tomemos uma matriz de coeficientes A com  $n=4$ , então para cada linha i, teremos a seguinte condição:

$$\text{Linha 1: } \text{somE}_0 = a_{00}.0 + a_{01}.1 + a_{02}.1 + a_{03}.1$$

$$\text{Linha 2: } \text{somE}_1 = a_{10}.\beta_0 + a_{11}.0 + a_{12}.1 + a_{13}.1$$

$$\text{Linha 3: } \text{somE}_2 = a_{20}.\beta_0 + a_{21}.\beta_1 + a_{22}.0 + a_{23}.1$$

Linha 4:  $somE_3 = a_{30}.\beta_0 + a_{31}.\beta_1 + a_{32}.\beta_2 + a_{33}.0$

Portanto, a matriz de coeficientes do numerador da equação de beta pode ser escrito como:

$$\begin{vmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{vmatrix}$$

A matriz beta pode ser escrita como:

$$\begin{vmatrix} 0 & 1 & 1 & 1 \\ \beta_0 & 0 & 1 & 1 \\ \beta_0 & \beta_1 & 0 & 1 \\ \beta_0 & \beta_1 & \beta_2 & 0 \end{vmatrix}$$

Agora fica fácil obter cada elemento do somE por um processo iterativo de repetição. Se tomarmos  $somE_i = 0$ , então basta tomar um índice i que fará a interação de transição da linha das matrizes de coeficientes e da matriz beta, além disso, devemos tomar o índice j, o que permitirá multiplicar cada elemento da linha da matriz de coeficientes com a linha da matriz beta, e a cada iteração de j, somE fica recebendo o seu próprio valor mais a próxima soma entre o produto dos elementos das matrizes já mencionadas.

Por fim, o vetor bet será útil, pois a partir dele, serão resgatados os betas de cada linha adotados pelo `random.uniform(a,b)`, com 'a' maior que 0 e 'b' menor que 1, e dividiremos cada elemento do  $somE_i$  pelo  $bet_i$ , obtendo-se, então o respectivo  $a_{ii}$ . Portanto, garantimos assim que a matriz satisfaz o critério de sassenfeld, é convergente, com determinante maior que zero e positiva, portanto, inversível. Uma observação a se fazer é que na função `random.uniform(a,b)`, foi definido a=0.1 e b=0.9, essa margem de segurança é levando-se em consideração o arredondamento realizado no número em que  $a_{ii}$  irá receber, além disso, se 'a' for muito pequeno, corre-se o risco de termos o valor de  $a_{ii}$  exageradamente grande. O livro, em que esse algoritmo foi tomado como base afirma que quanto menor for o valor de  $\beta$ , mais rápido será a convergência do método de Gauss-Seidel.

Mesmo com a obtenção da matriz convergente, a partir daqui é assumido que a matriz de coeficientes pode ou não satisfazer algum critério de convergência, com o objetivo de implementar todos os passos do método de Gauss-Seidel. Portanto, foi elaborado uma segunda função denominada **sassenfeld(matQ)**, onde será verificado o valor de beta correspondente em cada linha e irá retornar o vetor de todos os betas em uma lista, esse processo se utiliza basicamente da equação 02, onde para cada linha, teremos os coeficientes de linha da matriz i associadas aos pesos betas calculados em processos anteriores, obtendo-se o beta daquela linha. Na função foi definido inicialmente dois vetores bet e beta, com tamanhos 1:n, onde o bet começa inicialmente com os valores de seus elementos nulos (elemento neutro da adição) e os elementos de beta começam com valores unitários (elemento neutro da multiplicação), para cada iteração i, a variável div recebe o elemento da diagonal principal daquela linha, dentro da iteração de repetição i, definimos a iteração j, em que exceto na condição i ser igual a j, cada elemento de  $bet_i$  vai receber o seu próprio valor mais o módulo do elemento i,j da matriz dos coeficientes e, esse dividido por div, além disso, essa parcela está multiplicada por  $beta_j$ , após isso, é determinado o valor beta daquela linha, antes de acabar a iteração i, o vetor de elemento neutro da multiplicação (beta) recebe em i o valor atualizado daquela linha, portanto, na próxima iteração, o valor do elemento beta em  $j=i-1$  anterior ao i atual, associará o peso correspondente ao coeficiente da matriz em  $i-1,i$ .

Por fim é implementado a função **resov(matQ,vetB,lim,interac)**, onde essa função recebe a matriz de coeficientes, o vetor de termos constantes da equação matricial  $[A]X = [B]$ , a condição Gauss-Seidel implementada foi o algoritmo que estabelece a seguinte correspondência:  $(x)^{k+1} = [C](x)^k + [g]$ . Para

cada iteração k, é determinado o vetor x atualizado em função dos x anteriores calculados pela seguinte equação:

$$x_1^{(k+1)} = \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \dots - a_{1n}x_n^{(k)})$$

$$x_2^{(k+1)} = \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - \dots - a_{2n}x_n^{(k)})$$

$$x_3^{(k+1)} = \frac{1}{a_{33}}(b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k+1)} - a_{34}x_4^{(k)} - \dots - a_{3n}x_n^{(k)})$$

Por fim,

$$x_n^{(k+1)} = \frac{1}{a_{nn}}(b_n - a_{n1}x_1^{(k+1)} - a_{n2}x_2^{(k+1)} - a_{n3}x_3^{(k+1)} - \dots - a_{nn}x_{n-1}^{(k+1)})$$

Assim, o processo iterativo de Gauss-Seidel, no momento de se calcular  $x_j^{(k+1)}$ , são usados todos os valores  $x_1^{(k+1)}, \dots, x_{j-1}^{(k+1)}$  que já foram calculados e os valores  $x_{j+1}^{(k)}, \dots, x_n^{(k)}$  restantes.

No algoritmo foi definido os vetores x,r,s,t,X que, no final, terão de dimensões 1:n, onde todos os seus elementos serão neutros para a operação de adição. O critério de parada será pelo limite máximo de iterações ou caso o erro seja inferior a tolerância desejada. A partir daí será iniciado a iteração inicial, nesse primeiro momento cont=0, logo passa para a condição de estimativa do primeiro vetor resposta x atualizado, inicialmente é definido a iteração i, onde o elemento  $x_i$  do vetor x receberá o coeficiente constante do vetor B em i e será dividido pelo elemento da diagonal principal da linha da matriz dos coeficientes, após isso é definido a iteração j, onde o valor armazenado em  $x_i$  será subtraído de todos os elementos da linha i da matriz dos coeficientes (exceto a diagonal principal), sendo esses coeficientes divididos pelo elemento da diagonal principal da linha, além disso, essa razão está multiplicada pelo elemento do vetor X em j, notemos que em primeiro momento o vetor X representará o vetor atualizável em todo o momento, inicialmente começa com valores nulos, após a iteração j acabar, o vetor X em i receberá x em i (o que foi calculado atualmente na iteração i), e esse processo se repete até acabar toda iteração i. Quando cont>0 o vetor r receberá todos os vetores calculados, no final, teremos o vetor final X e o vetor r=X na iteração ( $i_{anterior} = i_{posterior} - 1$ ). Os vetores t e s receberão os módulos dos elementos de X e o módulo do incremento (X-r), respectivamente, assim, obteremos os máximos valores entre os elementos de t e s e o erro será a razão entre esses dois valores ( $\frac{\max(s)}{\max(t)}$ ), e isso deverá ser menor que a tolerância para que a função pare.

Por fim é implementado o módulo de solução baseado no algoritmo desenvolvido por mim e pelo módulo de solução numpy.linalg.solve(matQ,vetB). Neste momento agora será discutido a comparação entre as respostas e o tempo computacional.

Será indicado a ordem da matriz e a resposta fornecida para cada uma das soluções (para matrizes de ordem não muito grandes):

Digite a ordem da matriz n de coeficientes: 2

Matriz de Coeficientes Convergentes.

Número de tentativas: 9

duracao gauss-seidl: 0.000478

duracao np.linalg.solve: 0.000122

Digite a ordem da matriz n de coeficientes: 3

Matriz de Coeficientes Convergentes.

Número de tentativas: 25

duracao gauss-seidl: 0.000288

duracao np.linalg.solve: 0.000183

Digite a ordem da matriz n de coeficientes: 4  
Matriz de Coeficientes Convergentes.  
Número de tentativas: 10  
duracao gauss-seidl: 0.000185  
duracao np.linalg.solve: 0.000135

Digite a ordem da matriz n de coeficientes: 5  
Matriz de Coeficientes Convergentes.  
Número de tentativas: 10  
duracao gauss-seidl: 0.000276  
duracao np.linalg.solve: 0.000124

Digite a ordem da matriz n de coeficientes: 10  
Matriz de Coeficientes Convergentes.  
Número de tentativas: 10  
duracao gauss-seidl: 0.000620  
duracao np.linalg.solve: 0.001139

Digite a ordem da matriz n de coeficientes: 20  
Matriz de Coeficientes Convergentes.  
Número de tentativas: 8  
duracao gauss-seidl: 0.001668  
duracao np.linalg.solve: 0.000759

Digite a ordem da matriz n de coeficientes: 50  
Matriz de Coeficientes Convergentes.  
Número de tentativas: 7  
duracao gauss-seidl: 0.011031  
duracao np.linalg.solve: 0.001829

Digite a ordem da matriz n de coeficientes: 100  
Matriz de Coeficientes Convergentes.  
Número de tentativas: 7  
duracao gauss-seidl: 0.041553  
duracao np.linalg.solve: 0.003437

Digite a ordem da matriz n de coeficientes: 500  
Matriz de Coeficientes Convergentes.  
Número de tentativas: 6  
duracao gauss-seidl: 0.780878  
duracao np.linalg.solve: 0.047639

Digite a ordem da matriz n de coeficientes: 1000  
Matriz de Coeficientes Convergentes.  
Número de tentativas: 6  
duracao gauss-seidl: 3.238401  
duracao np.linalg.solve: 0.155515

Digite a ordem da matriz n de coeficientes: 5000  
Matriz de Coeficientes Convergentes.  
Número de tentativas: 5  
duracao gauss-seidl: 71.700235

duracao np.linalg.solve: 7.750064

Em relação a termos de tempo, quanto maior for a ordem da matriz de coeficientes  $n$ , maior também será o tempo de execução do código desenvolvido por mim. Porém o `numpy.linalg.solve()` se manteve bem eficiente para operações de grande ordem. É certo que a precisão entre o `numpy.linalg.solve()` e o meu algoritmo para sistemas de ordem inferiores a 1000 é garantido, no meu caso, o algoritmo desenvolvido por mim, até  $n=5000$ , ele se manteve bem preciso, poucas vezes, devido a singularidade, o solve do numpy (usando a função mais simples) não dava um resultado legal. Mas nessa execução de 5000, podemos comparar alguns valores que foram obtidos para  $x$ :


Meu algoritmo (os 20 primeiros elementos do vetor solução e os 10 finais):

```
0.0004868091887256847
-0.0006739322384696722
-0.00016154673686550255
0.0017770149114408058
-0.000976517721458
0.001045105868429568
0.0011071339063899095
1.1029598577188015e-05
-0.0019898490069666028
0.00027089630751834365
-0.0017829390158009871
0.0002882403908499187
0.00017775568536681442
7.828826965927868e-05
-0.0007271408172924619
-0.0009094733155610527
0.0023906254203058725
-0.0010131028145779842
0.0023161350642466765
-1.7746644525205226e-05
...
-0.0011316189051079186
-0.001972344013693668
-0.0037436555656576325
0.0019565569349933366
0.004028292831924728
-0.003185164938434359
0.00021771972593865953
0.00035737749250199186
0.00179346305905867
0.0010508797204541764
```

Solve (os 20 primeiros elementos do vetor solução e os 10 finais):

```
4.868091887251125714e-04
-6.739322384490694338e-04
-1.615467368991551549e-04
1.777014911460763789e-03
-9.765177214915433642e-04
```





1.045105868390279122e-03  
1.107133906379738353e-03  
1.102959856821258931e-05  
-1.989849006967898612e-03  
2.708963075271791400e-04  
-1.782939015780159175e-03  
2.882403908518910455e-04  
1.777556853511114555e-04  
7.828826967755237701e-05  
-7.271408173052454349e-04  
-9.094733155418549585e-04  
2.390625420288677513e-03  
-1.013102814631295029e-03  
2.316135064215087656e-03  
-1.774664453378577142e-05  
1.035929634695113193e-03  
...  
-1.131618905107844393e-03  
-1.972344013693610055e-03  
-3.743655565657499386e-03  
1.956556934993607631e-03  
4.028292831925356886e-03  
-3.185164938434735613e-03  
2.177197259386508060e-04  
3.573774925018420741e-04  
1.793463059058925417e-03  
1.050879720454527422e-03

Portanto, o algoritmo baseado no método de Gauss-Seidel fornece uma precisão aceitável para solução de sistemas de alta ordem.

## 01.2) Programação bônus Newton-Raphson :

Como esse algoritmo compõe em certa alguns dos elementos discutidos em '01)', então esse tópico será abordado de uma maneira resumida.

Como a matriz é gerada de modo que todos os elementos da matriz é constante e como o objetivo só era resolver uma matriz 20x20, foi definido as seguintes condições. Nesse algoritmo é considerado que o sistema resolvido é estritamente linear, ou seja, as funções são dadas por:

$$P_k(x_i) = a_{k0} + a_{k1}x_1^1 + a_{k2}x_2^1 + a_{k3}x_3^1 + a_{k4}x_4^1 + a_{k5}x_5^1 + a_{k6}x_6^1 + \dots + a_{kn}x_n^1 = 0, \text{ onde } P_k(x_i) = F_k(x_i) - b_k \text{ e } F_k(x_i) = b_k$$

Dessa forma conclui-se que  $a_0 = -b$  e tem-se  $1 \leq k \leq n$ . Além disso, a jacobiana de  $P_k(x_i)$  poderá ser dado por uma matriz de coeficientes constantes A de ordem n. Por exemplo, se a ordem da matriz for  $n=4$ , teremos a seguinte jacobiana:

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{vmatrix}$$

Tendo-se por base essa ideia, como a matriz de sassenfeld é convergente, com o determinante maior que 0 e inversível, então nesse algoritmo ela está definida como a jacobiana do sistema linear desejado (onde essa matriz é definida de modo aleatório, além disso, foi definida uma função denominada **pol**(jac,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16,x17,x18,x19,x20), em ela recebe a jacobiana e os valores  $x_i$  (soluções do sistema) calculados, dentro dessa função foi definido valores fixos para  $a_0 = -b$ , já que se esse coeficiente fosse 0, então a solução seria trivial, isto é, todos os valores de  $x_i$  seriam nulos. Assim, foi definido que para cada linha k da matriz de coeficientes, tem o seu próprio b (não aleatório, caso fosse, as condições dos sistemas seriam mudadas a cada interação da função).

Por fim o método de Newton-Raphson estabelece a seguinte condição: dado um vetor  $x_0$  inicial, o sistema é iterado pela condição  $x = x - \text{numpy.dot}(\text{np.linalg.inv}(\text{jac}), P(x))$ , para cada x obtido na interação do while loop, o novo x passa a ser o x inicial do sistema  $x_0 = x$ , e o processo se repete até que o sistema ultrapasse o número de interações estipulados ou quando tolerância é atingida. a tolerância é definida pela norma do vetor incremento, isto é,  $\text{np.linalg.norm}(\text{numpy.dot}(\text{np.linalg.inv}(\text{jac}), P(x)))$ .

Alguns dados calculados:

Primeira execução:

Tolerância: 0.1

Número de interações máxima: 1000

Os valores de x que satisfazem os coeficientes são:

```
0.46596126 -0.24915053 0.31987893 0.74437252
1.09099043 0.29514073 0.35265018 0.56028318
1.41661044 3.8762977 2.76465361 0.70053041
1.08397051 1.08153332 0.83189805 1.82542057
1.8134696 2.56606619 2.50366198 8.80807984
```

Os novos valores de P(x) em x solução são:

```
-0.0, 0.0, -0.0, 0.0, 0.0, 0.0, 0.0
-0.0, 0.0, 0.0, -0.0, 0.0, 0.0, 0.0
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
```

duracao algoritmo de Newton: 0.003238

Segunda Execução:

Tolerância: 0.000000000000001

Número de interações máxima: 1000

Os valores de x que satisfazem os coeficientes são:

0.79102269	-0.09549924	1.03666501	0.35211613
0.06661214	1.54003065	2.46993339	0.40374338
2.70293346	0.65661992	1.68798251	0.47468738
1.22378425	1.19876005	2.14122401	3.8471984
1.9311543	0.79717751	2.32589717	5.29207153

Os novos valores de  $P(x)$  em x solução são:

-0.0,	-0.0,	0.0,	0.0,	0.0,	-0.0,	0.0
0.0,	0.0,	0.0,	0.0,	0.0,	-0.0,	0.0
0.0,	0.0,	-0.0,	-0.0,	0.0,	-0.0	

duracao algoritmo de Newton: 0.003578



## 02) Programação de Lagrange e a montagem das funções de formas:

De acordo com a apostila do professor William Wagner Matos Lira, seja um polinômio  $P(x)$  com grau  $(n-1)$  que interpola  $f$  em  $x_1, x_2, x_3, \dots, x_n$ . Então  $P(x)$  pode ser representado na forma:

$$P(x) = f(x_1)L_1(x) + f(x_2)L_2(x) + \dots + f(x_n)L_n(x)$$

ou ainda

$$P(x) = \sum_{i=1}^n f(x_i)L_i(x)$$

Portanto, o polinômio de lagrange pode ser assim expresso:

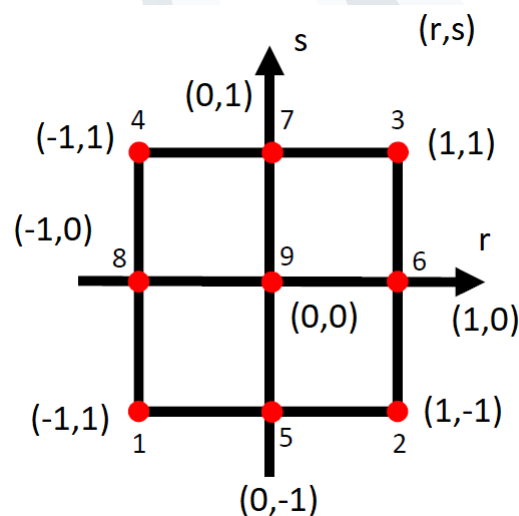
$$L_i(x) = \prod_{K=1, K \neq i}^n \frac{x - x_K}{x_i - x_K}$$

É importante mencionar que o algoritmo foi resolvido de modo simbólico com a biblioteca sympy, já que é uma forma mais útil de obter expressões que dê para comparar com as funções de formas fundamentadas nos livros de elementos finitos.

No python foi definido a função **lagr**(x,fx,type='x') em que x é um vetor de elementos que define as abscissas dos pontos a serem interpolados, como também fx representa o vetor das ordenadas correspondentes as abscissas, type permite definir as variáveis simbólicas a serem operadas.

No algoritmo é definido  $p=0$  (elemento neutro da adição) e a variável simbólica, é assumido que os vetores x e y devem apresentar as mesmas dimensões. Inicialmente iteramos i até o número de elementos representativo de x, após isso condicionamos o elemento neutro da multiplicação como s, dentro dessa iteração, devemos iterar também em j (até o número de elementos de x), restringimos a condição em que os elementos de x coincidam na mesma posição, pois nessa situação ocorre divisão por zero, então a cada iteração de j, s recebe a si mesmo multiplicado por  $\frac{var-x_j}{x_i-x_j}$ , a equação se tornará simbólica, porque var foi definido como simbólico em uma variável digitada. Por fim, a cada fim da iteração j rm cada i, p recebe uma parcela do polinômio interpolador(o que define o grau do polinômio).

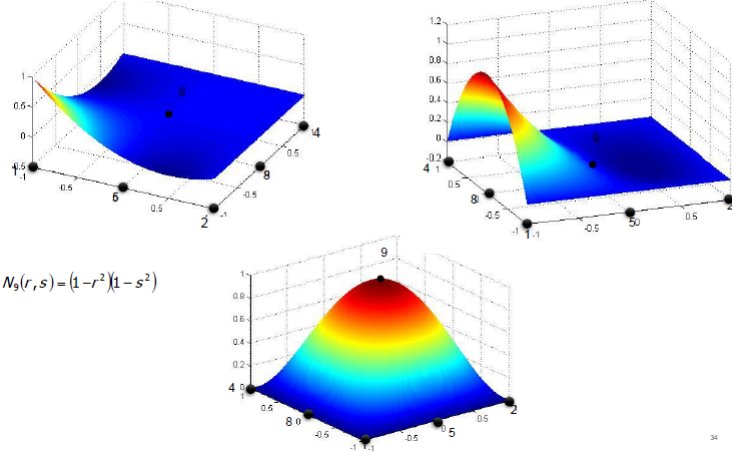
No problema que tange em obter as funções de forma de um elemento lagrangeano quadrilateral quadrático com 9 pontos indicados no problema, foram adotados como direções r (horizontal) e s (vertical), além das coordenadas sugeridas pelo professor Christiano.



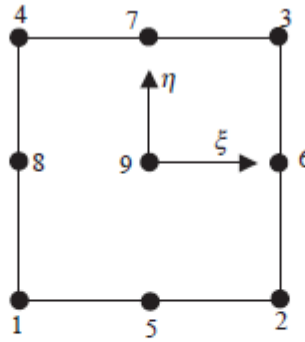
É interessante lucidar o que facilitou bastante em conseguir deduzir pelos 8 das 9 equações de formas foram as seguintes imagens:

9 nós

$$N_1(r,s) = \frac{1}{4}(1-r)(1-s) - \frac{1}{4}(1-r^2)(1-s) - \frac{1}{4}(1-r)(1-s^2) - \frac{1}{4}(1-r^2)(1-s^2) \quad N_8(r,s) = \frac{1}{2}(1-r)(1-s^2) - \frac{1}{2}(1-r^2)(1-s^2)$$



No livro The Finite Element Method: A Practical Course, G. R. LIU and S. S. QUEK, na página 157, os autores apresentam quais são as equações de forma para esse tipo de quadrícula:



**Figure 7.16.** Nine-node rectangular element.

Using Eqs. (7.100) and (4.82), the nine-node quadratic element shown in Figure 7.16 can be given by

$$\begin{aligned} N_1 &= N_1^{1D}(\xi)N_1^{1D}(\eta) = \frac{1}{4}\xi(1-\xi)\eta(1-\eta) \\ N_2 &= N_2^{1D}(\xi)N_1^{1D}(\eta) = -\frac{1}{4}\xi(1+\xi)\eta(1-\eta) \\ N_3 &= N_2^{1D}(\xi)N_2^{1D}(\eta) = \frac{1}{4}\xi(1+\xi)(1+\eta)\eta \\ N_4 &= N_1^{1D}(\xi)N_2^{1D}(\eta) = -\frac{1}{4}\xi(1-\xi)(1+\eta)\eta \\ N_5 &= N_3^{1D}(\xi)N_1^{1D}(\eta) = -\frac{1}{2}(1+\xi)(1-\xi)(1-\eta)\eta \\ N_6 &= N_2^{1D}(\xi)N_3^{1D}(\eta) = \frac{1}{2}\xi(1+\xi)(1+\eta)(1-\eta) \\ N_7 &= N_3^{1D}(\xi)N_2^{1D}(\eta) = \frac{1}{2}(1+\xi)(1-\xi)(1+\eta)\eta \\ N_8 &= N_1^{1D}(\xi)N_3^{1D}(\eta) = -\frac{1}{2}\xi(1-\xi)(1-\eta)\eta \\ N_9 &= N_3^{1D}(\xi)N_3^{1D}(\eta) = (1-\xi^2)(1-\eta^2) \end{aligned} \quad (7.101)$$

Após verificar os slides de determinadas universidades, esse livro de elementos finitos, o meu raciocínio se deu da seguinte forma: as funções de formas são geradas pelas equações de lagrange, em função da quantidade de pontos na quadrícula a serem interpolados, logo, como neste exemplo, a quantidade de pontos é 3, então a função de forma para a direção analisada é quadrática. A equação de forma de um ponto é definido como o produto das equações nas direções em que esse ponto é interceptado, assim, por exemplo, se é tomado o ponto 9 para a análise da função de forma nesse ponto, então, a condição de contorno é que nesse ponto, a altura seja unitária, para os demais pontos, nulos. Isso também serve para os outros pontos. Note que na imagem colocada, o ponto 9 fica definida na origem da quadrícula, como as equações das direções são de segundo grau, então, a "deformação" em que a quadrícula sofrerá será de tal forma, que em qualquer direção que passe por 9, também forme uma parábola, definindo então um cume. No ponto 1 por exemplo, imagino que uma parábola que atinge altura unitária nesse ponto, sendo os demais pontos presos, para as direções paralelas a r e s

Esse raciocínio valeu para todos os pontos, exceto o ponto 8, que tive que interpolar manualmente várias condições que me permitisse gerar duas equações quadráticas em função de r e s, que me permitisse expressá-la conforme a teoria. Portanto, em qualquer direção que eu tome a análise do ponto, os pontos de abscissas (r ou s) são constantes e iguais a [-1,0,1] para todos os pontos. Já as alturas (ordenadas) foram definidas da seguinte forma (ao longo dos pontos nas direções que o interceptam, no sentido de -1,0,1):

Ponto 1: ( $h_{N1r} = [1, 0, 0]$ ) e ( $h_{N1s} = [1, 0, 0]$ )  
 Ponto 2: ( $h_{N2r} = [0, 0, 1]$ ) e ( $h_{N2s} = [1, 0, 0]$ )  
 Ponto 3: ( $h_{N3r} = [0, 0, 1]$ ) e ( $h_{N3s} = [0, 0, 1]$ )  
 Ponto 4: ( $h_{N4r} = [1, 0, 0]$ ) e ( $h_{N4s} = [0, 0, 1]$ )  
 Ponto 5: ( $h_{N5r} = [0, 1, 0]$ ) e ( $h_{N5s} = [1, 0, 0]$ )  
 Ponto 6: ( $h_{N6r} = [0, 0, 1]$ ) e ( $h_{N6s} = [0, 1, 0]$ )  
 Ponto 7: ( $h_{N7r} = [0, 1, 0]$ ) e ( $h_{N7s} = [0, 0, 1]$ )  
 Ponto 8: ( $h_{N8r} = [-2, 0, 0]$ ) e ( $h_{N8s} = [1, 0, 0]$ )  
 Ponto 9: ( $h_{N1r} = [0, 1, 0]$ ) e ( $h_{N1s} = [0, 1, 0]$ )

Após essas construções, foram substituídos no lagrangeano do código que foi elaborado e também na função `scipy.interpolate` no pacote `lagrange`. Os parâmetros de entrada nas funções ficam `lagr([-1,0,1],hN1r)`, por exemplo. Para cada ponto foram obtidos as seguintes informações:

Ponto 1 (N1):

$$N1r = r \cdot (r - 1) / 2$$

$$N1s = s \cdot (s - 1) / 2$$

$$N1rs = r \cdot s \cdot (r - 1) \cdot (s - 1) / 4$$

$$\text{Expandido: } r^{**2} \cdot s^{**2} / 4 - r^{**2} \cdot s / 4 - r \cdot s^{**2} / 4 + r \cdot s / 4$$

A equação  $N1_{rs}$  pode ser escrita ainda como:

$$N1_{rs} = \frac{rs(-1-r)(-1-s)}{4} = \frac{r(1-r)s(1-s)}{4}$$

Ponto 2 (N2)

$$N2r = r \cdot (r + 1) / 2$$

$$N2s = s \cdot (s - 1) / 2$$

$$N2rs = r \cdot s \cdot (r + 1) \cdot (s - 1) / 4$$

$$\text{Expandido: } r^{**2} \cdot s^{**2} / 4 - r^{**2} \cdot s / 4 + r \cdot s^{**2} / 4 - r \cdot s / 4$$

A equação  $N2_{rs}$  pode ser escrita ainda como:

$$N2_{rs} = \frac{rs(r+1)(-1-s)}{4} = \frac{-r(r+1)s(1-s)}{4}$$

Ponto 3 N3

$$N3r = r \cdot (r + 1) / 2$$

$$N3s = s \cdot (s + 1) / 2$$

$$N3rs = r \cdot s \cdot (r + 1) \cdot (s + 1) / 4$$

$$\text{Expandido: } r^{**2} \cdot s^{**2} / 4 + r^{**2} \cdot s / 4 + r \cdot s^{**2} / 4 + r \cdot s / 4$$

A equação  $N3_{rs}$  pode ser escrita ainda como:

$$N3_{rs} = \frac{r(1+r)(1+s)s}{4}$$

Ponto 4 N4

$$N4r = r(r-1)/2$$

$$N4s = s(s+1)/2$$

$$N4rs = r*s*(r-1)*(s+1)/4$$

$$\text{Expandido: } r**2*s**2/4 + r**2*s/4 - r*s**2/4 - r*s/4$$

A equação  $N4_{rs}$  pode ser escrita ainda como:

$$N4_{rs} = \frac{rs(-(1-r))(1+s)}{4} = \frac{-r(r-1)(1+s)s}{4}$$

Ponto 5 N5

$$N5r = -(r-1)*(r+1)$$

$$N5s = s*(s-1)/2$$

$$N5rs = -s*(r-1)*(r+1)*(s-1)/2$$

$$\text{Expandido: } -r**2*s**2/2 + r**2*s/2 + s**2/2 - s/2$$

A equação  $N5_{rs}$  pode ser escrita ainda como:

$$N5_{rs} = \frac{-s(-(1-r))(1+r)(-(1-s))}{2} = \frac{-(1+r)(1-r)(1-s)s}{2}$$

Ponto 6 N6

$$N6r = r(r+1)/2$$

$$N6s = -(s-1)*(s+1)$$

$$N6rs = -r*(r+1)*(s-1)*(s+1)/2$$

$$\text{Expandido: } -r**2*s**2/2 + r**2/2 - r*s**2/2 + r/2$$

A equação  $N6_{rs}$  pode ser escrita ainda como:

$$N6_{rs} = \frac{-r(r+1)(-(1-s))(s+1)}{2} = \frac{r(r+1)(s+1)(1-s)}{2}$$

Ponto 7 N7

$$N7r = -(r-1)*(r+1)$$

$$N7s = s*(s+1)/2$$

$$N7rs = -s*(r-1)*(r+1)*(s+1)/2$$

$$\text{Expandido: } -r**2*s**2/2 - r**2*s/2 + s**2/2 + s/2$$

A equação  $N7_{rs}$  pode ser escrita ainda como:

$$N7_{rs} = \frac{-s(-(1-r))(1+r)(1+s)}{2} = \frac{(1+r)(1-r)(1+s)s}{2}$$

Ponto 8 N8

$$N8r = -r*(r-1)$$

$$N8s = s*(s-1)/2$$

$$N8rs = -r*s*(r-1)*(s-1)/2$$

$$\text{Expandido: } -r**2*s**2/2 + r**2*s/2 + r*s**2/2 - r*s/2$$

A equação  $N8_{rs}$  pode ser escrita ainda como:

$$N8_{rs} = \frac{-rs(-(1-r))(-(1-s))}{2} = \frac{-r(1-r)(1-s)s}{2}$$

Ponto 9 N9

$$N9r = -(r-1)*(r+1)$$

$$N9s = -(s-1)*(s+1)$$

$$N9rs = (r-1)*(r+1)*(s-1)*(s+1)$$

$$\text{Expandido: } r**2*s**2 - r**2 - s**2 + 1$$

A equação  $N9_{rs}$  pode ser escrita ainda como:

$$N9_{rs} = (r-1)*(r+1)*(s-1)*(s+1) = (r^2-1)(s^2-1) = (-(1-r^2))(-(1-s^2)) = (1-r^2)(1-s^2)$$

duracao algoritmo de lagrange programada: 0.050291

Com scipy e sympy

Ponto 1 N1

$$\text{Expandido: } 0.25*r**2*s**2 - 0.25*r**2*s - 0.25*r*s**2 + 0.25*r*s$$

Ponto 2 N2

$$\text{Expandido: } 0.25*r**2*s**2 - 0.25*r**2*s + 0.25*r*s**2 - 0.25*r*s$$

Ponto 3 N3

Expandido:  $0.25*r^{**2}*s^{**2} + 0.25*r^{**2}*s + 0.25*r*s^{**2} + 0.25*r*s$

Ponto 4 N4

Expandido:  $0.25*r^{**2}*s^{**2} + 0.25*r^{**2}*s - 0.25*r*s^{**2} - 0.25*r*s$

Ponto 5 N5

Expandido:  $-0.5*r^{**2}*s^{**2} + 0.5*r^{**2}*s + 0.5*s^{**2} - 0.5*s$

Ponto 6 N6

Expandido:  $-0.5*r^{**2}*s^{**2} + 0.5*r^{**2} - 0.5*r*s^{**2} + 0.5*r$

Ponto 7 N7

Expandido:  $-0.5*r^{**2}*s^{**2} - 0.5*r^{**2}*s + 0.5*s^{**2} + 0.5*s$

Ponto 8 N8

Expandido:  $-0.5*r^{**2}*s^{**2} + 0.5*r^{**2}*s + 0.5*r*s^{**2} - 0.5*r*s$

Ponto 9 N9

Expandido:  $1.0*r^{**2}*s^{**2} - 1.0*r^{**2} - 1.0*s^{**2} + 1.0$

duracao algoritmo scipy.interpolate: 0.034395

Por fim, se compararmos as expressões expandidas das duas situações (algoritmo desenvolvido e o scipy.interpolate), ambos geram as mesmas expressões, portanto, a precisão de ambas são excelentes, porém a segunda função foi mais eficiente do que a primeira em termos de velocidade de interpolação.



### 03) Método dos mínimos quadrados:

O algoritmo desenvolvido tem por base a discussão teórica da apostila do professor William. O ajuste tem por mérito tentar encontrar uma função  $p(x)$  que melhor satisfaça a melhor aproximação dessa função nos pontos de interesse, sendo assim, o ajuste não passa necessariamente por todos os pontos usados para sua determinação. A principal importância se dá pelo fato de podermos extrapolar ou prever regiões fora do intervalo considerado, porém deve-se ter em mente que o parâmetro da qualidade do ajuste se faz fundamental para que o erro de previsão seja o menor possível.

No ajuste polinomial devemos satisfazer a seguinte condição:

## MÉTODO DOS MÍNIMOS QUADRADOS

### Ajuste polinomial

#### • O sistema formado é do tipo:

$$\begin{bmatrix} m & \sum_{k=1}^m x_k & \sum_{k=1}^m x_k^2 & \dots & \sum_{k=1}^m x_k^{n-1} \\ \sum_{k=1}^m x_k & \sum_{k=1}^m x_k^2 & \sum_{k=1}^m x_k^3 & \dots & \sum_{k=1}^m x_k^n \\ \sum_{k=1}^m x_k^2 & \sum_{k=1}^m x_k^3 & \sum_{k=1}^m x_k^4 & \dots & \sum_{k=1}^m x_k^{n+1} \\ \dots & \dots & \dots & \ddots & \dots \\ \sum_{k=1}^m x_k^{n-1} & \sum_{k=1}^m x_k^n & \sum_{k=1}^m x_k^{n+1} & \dots & \sum_{k=1}^m x_k^{2(n-1)} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} \sum_{k=1}^m f(x_k) \\ \sum_{k=1}^m x_k f(x_k) \\ \sum_{k=1}^m x_k^2 f(x_k) \\ \vdots \\ \sum_{k=1}^m x_k^{n-1} f(x_k) \end{bmatrix}$$

No algoritmo, primeiramente, foi definida a função **pol**(n,x,y), onde n indica o número de coeficientes do polinômio de grau n-1 a ser ajustado, x representa o vetor das abscissas e y o vetor das ordenadas para o ajuste. A biblioteca numpy é importada já que a partir da função `numpy.poly1d()`, podemos transformar a lista ou array de coeficientes em uma função de uma variável. Tomamos uma matriz C de ordem n por n nula e um vetor b nulo, já que eles devem satisfazer a propriedade do elemento neutro da adição. Para a montagem da matriz C e b, começamos a iterar um loop em i e j (que irão repetir até n), onde os elementos da matriz C serão acessados, em seguida iteramos em k, onde um elemento da matriz nula receberá o somatório dos valores de x, acessados por k, onde a potência de elevação de cada elemento de x no respectivo  $C_{ij}$  será dado por  $(i+j)$ , pela propriedade de simetria dos elementos da matriz C, então a transposta de C ( $C_{ij}$ ) acaba recebendo o valor de  $C_{ij}$ . Na mesma iteração i, cada elemento de b é iterado em k, onde cada elemento  $b_i$  receberá o respectivo valor de y, na posição k, multiplicado pelo x na posição k, onde sua potência cresce em i. Uma vez obtidos os valores dos elementos da matriz C que corresponde na imagem a matriz nxn e b, o vetor de coeficientes constantes, aplicamos a inversa em C, resultando na matriz D, e realizamos o produto entre D e b, obtendo-se o vetor de coeficientes r, devemos lembrar que a função `poly1d()` do numpy ela toma o primeiro elemento do vetor de coeficientes como o grau mais alto do polinômio, a solução r está o inverso dessa condição, o primeiro elemento é o de menor grau, portanto, corrigimos as posições com o vetor auxiliar s, onde ele irá armazenar o último elemento de r como o seu primeiro, o penúltimo como o seu segundo, assim sucessivamente. Após esse tratamento, podemos aplicar s no `poly1d()`, onde será retornado um polinômio P(x). A qualidade de ajuste foi utilizado o  $r^2$ , conforme o Slide da professora Luciana C.L.M Viera, a sua equação é dada por:

$$\frac{\sum_{i=1}^n (y_i^{explicada} - y_{medio})}{\sum_{i=1}^n (y_i^{vetordoponto} - y_{medio})}$$

Dados Gerados:

Observação: o algoritmo foi realizado de tal forma que todas as entradas são aleatórias.

Primeira Execução:

Algoritmo elaborado:

243.8

$$r^2 = 3.361398873308735e - 26$$

$$-0.1202x + 274.1$$

$$r^2 = 0.013695013978668015$$

$$0.0006021x^2 - 0.4056x + 295.2$$

$$r^2 = 0.019833224126906038$$

$$-1.67e - 05x^3 + 0.01329x^2 - 2.953x + 386.5$$

$$r^2 = 0.08575633464642447$$

$$-4.82e - 08x^4 + 3.174e - 05x^3 - 0.002378x^2 - 1.203x^1 + 345.3$$

$$r^2 = 0.09690128331156522$$

duracão do algoritmo elaborado: 0.015122

---

243.8

$$-0.1202x + 274.1$$

$$0.0006021x^2 - 0.4056x + 295.2$$

$$-1.67e - 05x^3 + 0.01329x^2 - 2.953x + 386.5$$

$$-4.818e - 08x^4 + 3.174e - 05x^3 - 0.002378x^2 - 1.203x + 345.3$$

duracão da interpolação com numpy: 0.001918

---

243.8

$$-0.1202x + 274.1$$

$$0.0006021x^2 - 0.4056x + 295.2$$

$$-1.67e - 05x^3 + 0.01329x^2 - 2.953x + 386.5$$

$$8.492e - 09x^4 - 1.79e - 05x^3 + 0.01415x^2 - 3.119x + 391.2$$

duracão da interpolação com scipy: 0.005603

Na boa maioria dos casos, o algoritmo elaborado e a função `numpy.polyfit` são coincidentes, onde o pior desempenho de tempo é o do meu algoritmo e o melhor desempenho de tempo é o `numpy.polyfit`. Nota-se que também que há uma certa diferença nos coeficientes da função `scipy.optimize.curve_fit` para os coeficientes dos outros dois processos, quanto pior a qualidade do ajuste, maior é a divergência entre o `scipy` com relação as outras, certamente há funções que permitem melhorar mais o ajuste `scipy`.

#### 04) Integração Numérica (regra do trapézio repetida).

Foi implementado o algoritmo para a realização da regra do trapézio repetida. No algoritmo foi definido um vetor aleatório de 7 elementos, no qual representa os valores de coeficientes de um polinômio, quando aplicado no `numpy.poly1d(coef)`, obtendo-se  $P(x)$  de sexto grau, além disso foi utilizado a função `numpy.poly1d.deriv(P(x),m=2)`, obtendo-se a segunda derivada do polinômio  $P(x)$  para a estipulação do módulo do erro associado aos espaços que não foram considerados pelo método de integração numérica implementada. Com isso o usuário define o intervalo de integração e o espaçamento n correspondente, foi considerado a distância entre os pontos de intervalo  $h$  livre para efeitos de comparação, mas é plotado uma mensagem na tela para o usuário verificar se a integração está sendo realizada ou não com valores de intervalos maiores que 10.

A integral matemática:

$$\int_a^b f(x)dx$$

Pode ser representada segundo a metodologia desse método por:

$$I = \frac{h}{2} [f(x_0) + 2 \sum_{i=1}^{n-1} f(x_i) + f(x_n)]$$

O erro pode ser calculado por:

$$E(n) = -\frac{(b-a)^3}{12n^2} \max_{[a,b]}(f''(x))$$

Nesse algoritmo, após a inserção dos valores de  $x_i, x_f$  e  $n$ , o valor de  $h$  é calculado por  $h = \frac{b-a}{n}$ , quando  $n > 0$ , começa-se o cálculo do máximo valor da derivada no intervalo  $(a,b)$ , para isso, obtemos os valores da segunda derivada a cada incremento de 0.1 a partir de  $a$  até chegar em  $b$ , obtemos um vetor que contém os valores da segunda derivada de  $f(x)$ , após isso, obtemos o máximo valor da derivada, pelo máximo valor do vetor armazenado nele (de modo aproximado). A operação de cálculo da integração seguirá  $n$  passos, onde  $x_{pos} = x + i * h$ , onde  $i$  será o nosso processo iterativo, quando  $i=0$ ,  $x_{pos} = x_0$  e é calculado  $f$  em  $x_0$  e  $I$ , como elemento neutro da adição, recebe a si mais o valor calculado de  $f$ , quando  $0 < i < n-1$ ,  $I$  recebe a si mais o valor de  $2f$  em  $x_i$  e quando  $i=n$ ,  $I$  recebe a si mais o valor de  $f$  em  $x_f$ . Após isso basta multiplicar  $I$  por  $(h/2)$  e obter o valor da integração aproximada. É calculado também o erro  $Err$  e os limites de intervalo de resposta para a integração, conforme o algoritmo. Por último, foi usado o `scipy.integrate` para integrar o polinômio em questão e obter o seu valor de integração.

Primeira execução:

xi: 0

xf: 100

Espaçamento: 10

Valor de h: 10.0

Função:

$13x^6 - 9x^5 + 8x^4 - 11x^3 - 11x^2 + 10x^1 - 4$

A integral da função: 190671210014600.0.

O erro estimado: -18333.333333333336.

Intervalo de solução: 190671210032933.34 < 190671210014600.0 < 190671209996266.66

duracao algoritmo trapézio: 0.026732

Módulo Scipy Integração

(184230007097219.06, 2.0453639570619178)

duracao algoritmo scipy: 4.088576

Segunda Execução:

xi: 0

xf: 100

Espaçamento: 5

Valor de h: 20.0

Função:

$10x^6 + 11x^5 + 13x^4 - 11x^3 - 4x^2 + 15x$

A integral da função: 164634639115000.0.

O erro estimado: -26666.666666666668.

Intervalo de solução: 164634639141666.66<164634639115000.0<164634639088333.34

duracao algoritmo trapézio: 0.018652

Módulo Scipy Integração

(144716199932142.88, 1.6066725720092254)

duracao algoritmo scipy: 0.000410

Percebe-se com mais testes que quando maior o intervalo, mais impreciso se torna o algoritmo de integração do trapézio. Porém, dependendo da forma do polinômio de sexto grau, o scipy pode demorar um pouco, mas na maioria dos casos, ela é bem mais rápida do que o código elaborado e mais precisa.

Quando fazemos o espaçamento ser muito menor que 1, ocorre uma aproximação dos valores das duas ferramentas. Por exemplo:

Terceira Execução:

xi: 0

xf: 100

Espaçamento: 1000

Valor de h: 0.1

Função:

$-1x^6 - 15x^5 - 15x^4 - 14x^3 - 8x - 14$

A integral da função: -16816120627446.373.

O erro estimado: 0.0. Intervalo de solução: -16816120627446.373<-16816120627446.373<-16816120627446.373

duracao algoritmo trapézio: 0.033113

Módulo Scipy Integração

(-16816064327114.285, 0.1866958179954002)

duracao algoritmo scipy: 0.000552