Matrizes Esparsas

June 2, 2019

Este relatório tem por objetivo registrar experiências com Sistemas Lineares com Matrizes Esparsas. As experiências abordam 2 tipos de algoritmos de solução de Sistemas, a saber, SOR e Método do Gradiente Conjugado. Os algoritmos serão aqui implementados, e serão executados para 4 matrizes de características diferentes, incluindo uma matriz densa. Serão coletados os dados: norma do erro, quantidade de iterações para convergência, e tempos de execução(sabemos qual é a operação mais cara: o produto matriz vetor). Ao final, haverá uma experiência apenas para o Método do Gradiente Conjugado, e que abordará um par de matrizes de mesma configuração, sendo que uma delas contém denormais. Sobre os códigos, será abordado apenas o que for relevante, sendo a documentação feita por comentários no corpo do script.

Antes de iniciar com o código, devemos comentar a escolha da estrutura de dados escolhida para processar as matrizes esparsas, o CSR(Compressed Sparse Row). Esta decisão se deve ao fato da operação mais cara utilizar muito 'row slicing', e o profiler da nossa IDE identificar este como o trecho do código mais caro. Sendo assim, nossa pesquisa sobre bibliotecas encontrou esta como a melhor solução, além de um truque adicional no método SOR que será descrito junto do código.

Carregadas as bibliotecas, partiremos para os algoritmos.

[220]:
```
#from tqdm import tqdm_notebook # barra de progresso
import datetime as dt # para contar tempo de processamento
from statistics import mean # para calcular tempos médios de processamento
import numpy as np # biblioteca para matrizes
import pandas as pd # biblioteca para dataframes
from scipy.io import mmread # biblioteca para ler o arquivo .mtx
```

A função abaixo implementa o algoritmo SOR e coleta dados de tempo de execução, sendo 3 dados de tempo calculados:

1 - média do tempo por iteração

2 - média do tempo do produto linha da matriz X vetor

3 - razão do tempo total do produto matriz X vetor pelo tempo total de execução

Os dados retornados se referem a: erro estimado, iterações para convergência, e 1, 2 e 3 acima

Sobre a implementação, gostaríamos de descrever o procedimento feito para melhorar o tempo do 'row slicing'. Basicamente, a idéia foi evitá-lo. Obtivemos os valores da diagonal principal, criamos nova matriz e zeramos a diagonal da nova matriz. Assim, quando fazemos o produto interno da linha pelo vetor, o elemento referente à diagonal não é somado. Além disso, usamos apenas um vetor para o produto interno, sendo que as posições certas já contém os valores a serem calculados(é desnecessário criar um vetor para o x 'antigo' e outro para o x 'novo'). Com esses ajustes, evitamos a segmentação tanto das linhas da matriz, quanto dos vetores, evitando o 'row slicing'.

[221]:
```python
def SOR(A, # matriz da transformação linear
        b, # vetor resultante da transformação linear
        w, # ômega
        N, # número máximo de iterações
        TOL = 10**(-5) # precisão desejada
       ):

    # obtendo o número de equações
    # inicializando o vetor iteração anterior
    # inicialmente, x = x0, e para evitar slices
    # inicializando o vetor de tempo de execução por linha
    # inicializando o vetor de médias de tempo de execução
    # alterando a diagonal da matriz
    # inicializando a soma dos tempos de execução da operação mais cara
    # iniciando a contagem do tempo total de execução
    n = A.get_shape()[0]
    x0 = np.zeros(n)
    x = np.zeros(n)
    time_dot = np.empty(n)
    mean_dot = np.empty(N)
    D = A.diagonal(0); newA = A.copy(); newA.setdiag(0); newA.eliminate_zeros()
    sum_dot = 0
    start_it = dt.datetime.today().timestamp()

    # substituir para rodar com barra de progresso
    # for k in tqdm_notebook(range(N)):
    for k in range(N):

        for i in range(n):

            start_dot = dt.datetime.today().timestamp()
            summ = newA[i,].dot(x) # a operação mais cara, evitando slices
            time_dot[i] = dt.datetime.today().timestamp() - start_dot
            x[i] = (1-w)*x0[i] + (w*(- summ + b[i]))/D[i] #calculando x[i]

        sum_dot += sum(time_dot); mean_dot[k] = mean(time_dot)
        diff = np.linalg.norm(x - x0)
        if diff < TOL: break
        k += 1; x0[:] = x

    time_it = dt.datetime.today().timestamp() - start_it
    per_it = time_it/k; per_dot = mean(mean_dot[:k+1])

    return diff, k, per_it, per_dot, sum_dot/time_it
```

Aqui implementamos o algoritmo do Gradiente Conjugado e coletamos dados de tempo de execução, sendo 3 os dados de tempo calculados:

1 - média do tempo do produto matriz X vetor

2 - média do tempo por iteração
3 - razão do tempo total do produto matriz X vetor pelo tempo total de execução
Os dados retornados se referem a: erro estimado, iterações para convergência, e 1, 2 e 3 acima.
O algoritmo implementado é idêntico ao do livro.

[222]:
```
def ConjGrad(A, # matriz da transformação linear
             b, # vetor resultante da transformação linear
             N, # número máximo de iterações
             TOL = 10**(-5) # precisão desejada
             ):

    # obtendo o número de equações
    # inicializando o vetor iteração anterior
    # inicializando o vetor de contagem de tempo
    n = A.get_shape()[0]
    x = np.zeros(n)
    time_dot = np.empty(N)

    # inicializando r, v, alfa
    r = b - A.dot(x); v = r; alfa = r.dot(r)

    start_it = dt.datetime.today().timestamp()

    for k in range(N):

        if np.linalg.norm(v) < TOL: break # primeiro critério de parada

        start_dot = dt.datetime.today().timestamp()
        u = A.dot(v) # a operação mais cara
        time_dot[k] = dt.datetime.today().timestamp() - start_dot
        # calculando t, x, r e beta
        t = alfa/v.dot(u); x = x + t*v; r = r - t*u; beta = r.dot(r)

        # segundo critério de parada
        if beta < TOL: if np.linalg.norm(r) < TOL: break

        # calculando s, v, e atualizando alfa
        s = beta/alfa; v = r + s*v; alfa = beta

    time_it = dt.datetime.today().timestamp() - start_it
    per_it = time_it/k; per_dot = mean(time_dot[:k+1])

    return np.linalg.norm(r), k, per_it, per_dot, sum(time_dot[:k+1])/time_it
```

Por último, temos a função que gera dados a partir das matrizes escolhidas.

É retornada uma tabela com os dados das execuções do agoritmo SOR para 5 ômegas diferentes, e da execução do algoritmo do Gradiente Conjugado. Comentários sobre os dados serão feitos junto das tabelas.

[223]:
```python
def table(file, N, TOL):

    # obtendo os valores para cálculo
    # obtém a matriz esparsa contida em um arquivo .mtx
    # a matriz é retornada no formato CSR
    # obtendo o número de equações
    # vetor b, produto da matriz A com o vetor unitário
    # contando não zeros da matriz
    A = mmread(file).tocsr()
    n = A.get_shape()[0]
    b = A.dot(np.ones(n))
    nz = A.nnz

    # gerando a tabela para receber os dados
    indexes = [1, 1.25, 1.5, 1.75, 2, 'CG']
    columns = ['erro',
               'iterações',
               'segs/iteração',
               'segs/dotproduct',
               'soma_dot/iteração']
    table = pd.DataFrame(index = indexes, columns = columns)
    table = table.rename_axis(index=('w'))

    # preenchendo a tabela
    print('n = ', n, ', não-zeros = ', nz,', densidade = ', nz/(n**2))
    # substituir para rodar com barra de progresso
    #for i, w in enumerate(tqdm_notebook(indexes)):
    for i, w in enumerate(indexes):

        if w == 'CG':
            table.loc[w] = ConjGrad(A, b, N, TOL)
        else:
            table.loc[w] = SOR(A, b, w, N, TOL)

    return table
```

Rodamos todas as matrizes com as mesmas quantidade máxima de iterações e precisão desejada, 1000 e 1e-10, respectivamente. Maiores informações sobre as matrizes de cada experimento podem ser obtidas nos links no final dos parágrafos.

O problema trata de uma pesquisa de leitores de 124 revistas e jornais. Os elementos da matriz representam os grafos de quantidade de leitores entre duas revistas e jornais diferentes.

Temos aqui uma matriz com densidade elevada, de ordem 124X124. No algoritmo SOR, foi observada a melhor performance com w = 1 (equivalente ao Gauss-Seidel), taxa de segundos por iteração alta, taxa de segundos por produto interno alta, e peso da operação mais cara no total moderada. Podemos supor que a matriz possui raio espectral baixo, uma vez que conseguimos atingir a precisão desejada com todos os ômegas(exceto w = 2, quando não se espera convergência). Poderíamos também supor uma relação dos resultados de tempo com a alta densidade, mas temos a seguir, para comparação, uma matriz de ordem maior, mas com menor número de não-zeros, em que se observam alguns resultados de tempo ainda mais elevados.

https://sparse.tamu.edu/Pajek/Journals

http://vlado.fmf.uni-lj.si/pub/networks/data/2mode/journals.htm

[224]:
```
print(table("Journals.mtx", 1000, 1e-10))
```

```
n =  124 , não-zeros =  12068 , densidade =  0.7848595213319459
             erro iterações segs/iteração segs/dotproduct soma_dot/iteração
w
1      7.47711e-11        57      0.0278745     0.000180777          0.818298
1.25   6.90859e-11        88      0.0241484     0.000157264          0.816715
1.5    8.36802e-11       154      0.0276445     0.000181761          0.820586
1.75   9.91082e-11       355       0.023195     0.000152544          0.817797
2          85.9073      1000       0.023262     0.000152073          0.810637
CG     9.81114e-11       214    0.000102141     4.81151e-05          0.473266
```

O problema trata duma rede de energia de ônibus(elétricos, provavelmente). Uma versão interativa do grafo está no segundo dos links abaixo.

Em contraste com a matriz acima, temos densidade bastante baixa, mas ordem um pouco maior: 494X494. No algoritmo SOR, observa-se melhor performance para w = 1, mas a precisão desejada não é atingida. Ainda mais, mesmo para o Gradiente conjugado a precisão não é atingida. O tempo por iteração é bem maior, mesmo tendo a matriz anterior um tempo já alto em relação às outras matrizes testadas. Por outro lado, o tempo no gasto no produto interno é o menor de todos.

https://sparse.tamu.edu/HB/494_bus

http://networkrepository.com/494-bus.php

[227]:
```
print(table("494_bus.mtx", 1000, 1e-10))
```

```
n =  494 , não-zeros =  1666 , densidade =  0.006826861610582045
             erro iterações segs/iteração segs/dotproduct soma_dot/iteração
w
1       0.00144303      1000      0.0902826     0.000149412          0.817536
1.25     0.0020515      1000       0.088875     0.000149024          0.828327
1.5     0.00302569      1000      0.0891975     0.000149531          0.828143
1.75     0.0051681      1000      0.0863588     0.000144522          0.826714
2          26.4534      1000      0.0877452     0.000145076          0.816769
CG     0.000301275       999     6.8013e-05     2.07796e-05           0.30583
```

Apresentamos nas matrizes abaixo outro contraste de esparsidades, dessa vez sendo uma delas densa(não contém zeros), e com tamanhos próximos (48 e 66 linhas). Curiosamente, apesar de ambas coincidirem com melhores ômegas em 1.75, a matriz cheia apresentou melhor desempenho do que a esparsa no Gradiente Conjugado, convergindo com menos iterações. Outro detalhe, dessa vez no SOR, são os segundos por iteração, que dessa vez correspondem à quantidade de não-zeros. Por outro lado, o tempo de execução médio para os produtos internos parece ser o mesmo, assim como o peso dessa operação em relação ao total. Aqui, é certo que a quantidade de linhas influenciou no tempo. Deixamos então observado que este pode ser o motivo do maior tempo de execução por iteração, uma vez que este fenômeno ocorreu nas matrizes anteriores.

https://sparse.tamu.edu/HB/bcsstk01

https://sparse.tamu.edu/HB/bcsstk02

[225]:
```
print(table("bcsstk01.mtx", 1000, 1e-10))
```

```
n =  48 , não-zeros =  400 , densidade =  0.1736111111111111
             erro iterações segs/iteração segs/dotproduct soma_dot/iteração
w
1      0.000999687      1000    0.00967377       0.0001619          0.803324
1.25   0.000312833      1000    0.00875674     0.000148746          0.815348
1.5    1.45816e-05      1000    0.00881797     0.000150182          0.817503
1.75   1.56816e-10      1000    0.00885225     0.000150828          0.817844
2          477.536      1000    0.00895636      0.00015041          0.806098
CG     8.97315e-11       182   6.44673e-05     1.87856e-05          0.292998
```

[226]:
```
print(table("bcsstk02.mtx", 1000, 1e-10))
```

```
n =  66 , não-zeros =  4356 , densidade =  1.0
             erro iterações segs/iteração segs/dotproduct soma_dot/iteração
w
1       0.00114632      1000     0.0125295     0.000153921          0.810785
1.25   0.000301489      1000     0.0123401     0.000153379          0.820336
1.5    1.28507e-05      1000     0.0135371     0.000168988          0.823896
1.75   1.40962e-10      1000     0.0119389     0.000147984           0.81808
2          3.47885      1000     0.0124204     0.000151885          0.807092
CG     6.83338e-11        77   6.45185e-05      2.1476e-05          0.337189
```

Finalmente, apresentamos esta experiência final, que é mais uma 'curiosidade'. Encontramos essas matrizes, de alta ordem, e que demonstram a grande eficiência do Método do Gradiente Conjugado. Nos links abaixo, no final da página, é encontrada uma descrição, onde se observa que a primeira matriz apresenta maior tempo de processamento quando é feita sua fatoração, por causa de denormais. Entendemos essa fatoração ao que o texto se refere como sendo a fatoração LU, e ao executar o Gradiente Conjugado para ambas, obtivemos um comportamento inverso, mesmo que discreto: a matriz que possui os denormais converge mais rápido.

https://sparse.tamu.edu/MaxPlanck/shallow_water1

https://sparse.tamu.edu/MaxPlanck/shallow_water2

[228]:
```
A = mmread("shallow_water1.mtx").tocsr()
n = A.get_shape()[0]
b = A.dot(np.ones(n))
nz = A.nnz
data = (ConjGrad(A, b, 1000, 1e-10))
print('n = ', n,
      '\nnão-zeros = ', nz,
      '\ndensidade = ', nz/(n**2),
      '\nerro = ', data[0],
      '\niterações = ', data[1],
      '\nsegs/iteração = ', data[2],
      '\nsegs/dotproduct = ', data[3],
      '\nproporção = ', data[4])
```

```
n =  81920
não-zeros =  327680
densidade =  4.8828125e-05
erro =  4.769465925917846e-11
iterações =  41
segs/iteração =  0.00201924254254597
segs/dotproduct =  0.0009094703765142532
proporção =  0.46138715938740127
```

[229]:

```python
A = mmread("shallow_water2.mtx").tocsr()
n = A.get_shape()[0]
b = A.dot(np.ones(n))
nz = A.nnz
data = (ConjGrad(A, b, 1000, 1e-10))
print('n = ', n,
      '\nnão-zeros = ', nz,
      '\ndensidade = ', nz/(n**2),
      '\nerro = ', data[0],
      '\niterações = ', data[1],
      '\nsegs/iteração = ', data[2],
      '\nsegs/dotproduct = ', data[3],
      '\nproporção = ', data[4])
```

```
n =  81920
não-zeros =  327680
densidade =  4.8828125e-05
erro =  6.607579722231746e-11
iterações =  77
segs/iteração =  0.003762300912435953
segs/dotproduct =  0.0013638765383989383
proporção =  0.3672192237864565
```