Resolução de Sistemas Lineares Esparsos

André Barreto, Igor Venturini e Vinicius Arruda

Universidade Federal do Espírito Santo
Departamento de Informática

11 de Outubro, 2015

Resumo

Neste trabalho será abordado a resolução de sistemas lineares esparsos, apresentando métodos e algoritmos e, por fim, resultados numéricos através de testes. Para isto, serão utilizados métodos de compactação de matrizes esparsas e algoritmos de resolução de sistemas lineares adaptados para a utilização destas matrizes.

Keywords: sistemas lineares. matriz esparsa.

Introdução

Um sistema linear é uma abstração matemática de forma representar um problema ou situação real de diversas áreas do conhecimento. Em muitos casos, estes sistemas são esparsos e de ordem elevada. Um sistema esparso é tal que seus coeficientes não nulos são em número pequeno. Ou seja, uma matriz deste sistema teria zeros em grande quantidade e poucos coeficientes relevantes ao processo. [4]

Existem vários algoritmos de resolução de sistemas lineares eficientes. Porém, se o sistema é esparso e de grande porte, estes algoritmos, simplesmente utilizados, perdem muito a eficiência, uma vez que são poucos os coeficientes que precisam ser avaliados.

Portanto, trataremos ao longo deste material métodos de resolução de sistemas lineares com a utilização de técnicas de armazenamento de matrizes esparsas. Mais especificamente, serão implementados em linguagem C o algoritmo de eliminação de Gauss e o algoritmo SOR, ambos utilizando destas técnicas.

1 Referencial Teórico

Nesta seção serão abordados os métodos de resolução dos sistemas lineares propostos e os métodos de armazenamento de sistemas lineares esparsos implementados.

Serão apresentados pseudo-códigos em subseções, onde foram tomadas as seguintes convenções:

- A: matriz de coeficientes

- U: matriz triangular superior de coeficientes
- b: vetor de termos independentes
- n: ordem da matriz
- x: vetor solução do sistema
- ω: fator de relaxação

1.1 Eliminação de Gauss

A Eliminação de Gauss é um método direto de resolução que consiste em transformar um sistema linear em um sistema triangular superior equivalente por meio de operações l-elementares. Em seguida, aplica-se o algoritmo de substituições retroativas para encontrar a solução dos sistema.

Operações l-elementares são operações aplicadas a linhas (ou colunas) da matriz que resultam em uma matriz equivalente a anterior. Estas são:

- Trocar uma linha com outra:
  $R_i \leftrightarrow R_j$
- Multiplicar uma linha por uma constante não nula:
  $kR_i \rightarrow R_i,$ where $k \neq 0$
- Trocar uma linha por esta linha somado com um múltiplo de outra:
  $R_i + kR_j \rightarrow R_i,$ where $i \neq j$

De acordo com as definições acima, temos o Algoritmo 1, que corresponde ao Método de Gauss. Como entrada, recebe uma matriz de coeficientes A, um vetor de termos independentes b e a ordem da matriz n. Retorna uma matriz triangular superior U e o vetor de termos independentes modificado b.

Entrada: A, b, n
Saída: U, b
1 **inicio**
2 **para** $k \longleftarrow 1$ **até** $n-1$ **faça**
3 **para** $i \longleftarrow k+1$ **até** n **faça**
4 $m \longleftarrow -\left(\frac{A_{ik}}{A_{kk}}\right)$;
5 **para** $j \longleftarrow k$ **até** n **faça**
6 $A_{ij} \longleftarrow A_{ij} + mA_{kj}$;
7 $b_i \longleftarrow b_i + mb_k$;

Algoritmo 1: Eliminação de Gauss

1.2 Sucessive Over Relaxation

O método de sucessivos sobre-relaxamento (SOR) é uma variante do método de Gauss-Seidel para resolver um sistema de equações lineares, resultando em convergência mais rápida. Um método similar pode ser usado para qualquer processo iterativo lentamente convergente, desta forma podendo acelerar a convergência do sistema linear.

A convergência do método SOR depende do parâmetro ômega (ω) e é necessário que ω seja $0 < \omega < 2$ para garantir a convergência. Usualmente, utiliza-se ω entre 1 e 2, porém não sendo necessariamente uma regra, particularmente para $\omega = 1$, a recorrência para a resolução do sistema de modo iterativo se torna equivalente a de Gauss-Seidel.

Veja o Algoritmo 2 que corresponde ao Método de SOR.

Entrada: A, b, n, ω
Saída: x
1 **inicio**
2 $x \longleftarrow 0$;
3 **repita**
4 **para** $i \longleftarrow 1$ **até** n **faça**
5 $\phi \longleftarrow 0$;
6 **para** $j \longleftarrow 1$ **até** n **faça**
7 **se** $j \neq i$ **então**
8 $\phi \longleftarrow \phi + A_{ij}x_j$;
9 $x_i \longleftarrow (1-\omega)x_i + \left(\frac{\omega}{A_{ii}}\right)(b_i - \phi)$;
10 **até** *convergência*;

Algoritmo 2: Método SOR

1.3 Métodos de Armazenamento

Como explicitado anteriormente, torna-se essencial criar uma forma adequada de armazenamento das informações do sistema linear esparso, de tal forma reduzir a complexidade dos algoritmos de resolução de sistemas.

Serão abordados a seguir dois métodos que foram implementados para alcançar o objetivo deste trabalho.

1.3.1 CSR

O *Compressed Sparse Row* (CSR) é uma forma de armazenamento de sistemas esparsos. Este, como todos os outros, evitam o armazenamento de valores nulos ou zeros, uma vez que são dispensáveis no processo de resolução. [2]

O CSR possui, essencialmente, três vetores que armazenam informações sobre os valores, o índice em que ocorrem e os índices que indicam onde se inicia uma nova linha da matriz. Nesta implementação, foi incluído um vetor com os valores da diagonal principal e um vetor dos valores independentes do sistema. Resultando na seguinte forma:

- *values*: lista encadeada de valores não nulos da matriz
- *column_index*: vetor de índices das colunas para cada valor da lista de valores
- *row_index*: vetor de posições de onde na lista de valores representa uma nova linha
- *diagonal*: vetor de valores da diagonal da matriz de coeficientes.
- *b*: vetor de termos independentes do sistema.

Este método foi adotado para a resolução do algoritmo SOR.

1.3.2 Sparse

Através de tentativas de resolução com o método de Gauss, percebemos que o *CSR* não era apropriado, pois resultava em um algoritmo de complexidade cúbica, chegando a ser pior do que o método direto sem armazenamento esparso. Isto se devia ao fato de não haver uma forma rápida de acessar os elementos não nulos, sendo necessário iterar, de uma forma indireta, em todos os elementos da matriz.

Dado o problema encontrado, foi-se desenvolvido outra forma de armazenamento. Esta nomeamos simplesmente de *Sparse*. Com o objetivo de agilizar o acesso aos elementos relevantes da matriz da forma proposta pelo algoritmo de Gauss, foi criado a seguinte estrutura:

- *matrix*: vetor de ponteiros para listas encadeadas que armazenam os valores não nulos. Cada posição i da *matrix* representa a coluna i da matriz de coeficientes.
- *diagonal*: vetor de valores da diagonal da matriz de coeficientes.
- b: vetor de termos independentes do sistema.
- n: tamanho de cada vetor descrito acima.

A variável *matrix* é um vetor de listas encadeadas que substitui o vetor de valores presente no CSR, visando reduzir o custo computacional da manipulação das informações durante o procedimento da eliminação de Gauss. Estas listas são compostas de valores em uma mesma coluna, facilitando a iteração de busca de valores não nulos para o cálculo de m e combinação de linhas.

2 Implementação

A seguir serão apresentados os códigos relativos à implementação dos métodos considerados. Foi escolhido a linguagem C para a codificação.

2.1 CSR

2.1.1 Estrutura

```
1  typedef struct
2  {
3      size_t n;           // Ordem da matriz
4      size_t nnz;           // Numero de valores nao nulos
5      size_t* column_index;    // Vetor de indices das colunas de A
6      size_t* row_ptr;       // Vetor de posicoes de novas linhas
7      double* A;           // Vetor de valores nao nulos
8      double* d;           // Vetor de valores da diagonal
9      double* b;           // Vetor de termos independentes
10 } CSR;
```

Listing 1 – Estrutura da CSR

2.1.2 Principais funções

Reserva espaço no heap para uma estrutura do tipo CSR e retorna o ponteiro para a memória alocada.

```c
#define get_memory(ptr, size) ((ptr) = malloc(size));if((ptr) == NULL){
    fprintf(stderr, "Out of memory.\n");exit(EXIT_FAILURE);}

CSR* new_csr(size_t n, size_t nnz)
{
  CSR* new;
  get_memory(new, sizeof(CSR));
  new->n = n;
  new->nnz = nnz;
  get_memory(new->column_index, nnz * sizeof(size_t));
  get_memory(new->row_ptr, (n + 1) * sizeof(size_t));
  get_memory(new->A, nnz * sizeof(double));
  new->d = NULL;
  new->b = NULL;

  return new;
}
```

Listing 2 – Função alocar memória CSR

Inicialmente cria uma estrutura temporária do tipo COO e lê do arquivo a matriz, armazenando os índices da linha e da coluna de cada valor e o próprio valor para esta estrutura. Após a leitura, cria-se uma estrutura do tipo CSR e faz um processamento da estrutura COO através da função *coo2csr* que converte a estrutura COO para o formato CSR. A memória ocupado pela COO é liberada e é retornado a estrutura CSR preenchida com os valores da matriz.

```c
CSR* get_csr(const char* file_name)
{
  FILE* stream;
  COO* coo;
  size_t i, j, x, y, n, nnz;
  double temp_A;

  stream  = fopen(file_name, "r");
  if(stream == NULL) {
    fprintf(stderr, "Error while opening file %s.\n", file_name);
    exit(EXIT_FAILURE);
  }
  fscanf(stream, "%lu %lu %lu", &n, &n, &nnz);
  coo = new_coo(n, nnz);
  i = j = 0;
  while(nnz > 0) {
    nnz--;
    fscanf(stream, "%lu %lu %lf", &x, &y, &temp_A);
    if(x == y) coo->d[i++] = temp_A;
    else {
      coo->row_index[j] = x - 1;
      coo->column_index[j] = y - 1;
      coo->A[j++] = temp_A;
    }
  }
  for(i = 0; i < n; i++)
    fscanf(stream, "%lf", &coo->b[i]);

  fclose(stream);
  return coo2csr(coo);
}
```

Listing 3 – Função gerar CSR

2.2 Sparse

2.2.1 Estrutura

```c
typedef struct list
{
  double value;
  size_t row;
  struct list* next;
} List;

typedef struct
{
  size_t n;
  double* diagonal;
  double* b;
  List** matrix;
} Sparse;
```

Listing 4 – Estrutura da Sparse

2.2.2 Principais funções

A função *getSparse* é a responsável por criar a estrutura. Recebendo como parâmetro a nome do documento a ser lido, abre este documento e inicia-se a leitura. Inicialmente lê-se a ordem da matriz e o número de valores não nulos. Com isto, cria-se o espaço necessário para a estrutura. Após isto é iniciado a leitura de acordo com o padrão estabelecido dos arquivos. Cada informação de valor lida é armazenada na estrutura. Se $i = j$, o valor será armazenado na *diagonal*, o vetor de valores da diagonal da matriz. Caso contrário, estes serão inseridos na *matrix*.

```c
Sparse* getSparse(const char* fileName)
{
  FILE* stream;
  Sparse* sparse;
  size_t n, nnz, i, j;
  double temp_A;

  stream = fopen(fileName, "r");
  if(stream == NULL) {
    fprintf(stderr, "Error while opening file %s.\n", fileName);
    exit(EXIT_FAILURE);
  }
  fscanf(stream, "%lu %lu %lu", &n, &n, &nnz);
  get_memory(sparse, sizeof(Sparse));
  sparse->n = n;
  get_memory(sparse->diagonal, n * sizeof(double));
  get_memory(sparse->b, n * sizeof(double));
  get_memory(sparse->matrix, n * sizeof(List *));

  for(i = 0; i < n; i++) sparse->matrix[i] = NULL;

  while(nnz > 0) {
    fscanf(stream, "%lu %lu %lf", &i, &j, &temp_A);
    i--; j--;
    if(i == j) sparse->diagonal[j] = temp_A;
    else insert(&sparse->matrix[j], temp_A, i);
    nnz--;
  }
  for(i = 0; i < n; i++) fscanf(stream, "%lf", &sparse->b[i]);
```

```
30
31    fclose(stream);
32    return sparse;
33  }
```

Listing 5 – Função gerar Sparse

A função *get* recupera o valor dado a linha i e coluna j recebidas como parâmetros. Se $i = j$, será feito um acesso O(1) no vetor diagonal. Caso contrário é realizado uma busca O(n) na estrutura de listas encadeadas.

```
1  double get(Sparse* sparse, size_t i, size_t j)
2  {
3    List* column;
4    if(i == j) return sparse->diagonal[j];
5    else {
6      column = sparse->matrix[j];
7      while(column != NULL) {
8        if(column->row == i)
9          return column->value;
10
11       column = column->next;
12     }
13   }
14   return 0.0;
15 }
```

Listing 6 – Função recuperar valor

2.3 Eliminação de Gauss

A Eliminação de Gauss para matriz esparsa armazenada na estrutura *sparse*, a lógica de procedimento é visualmente similar se comparado ao algoritmo normal. A importante mudança é que não é necessário avaliar todas as linhas abaixo do pivô atual para realizar as combinações lineares. Com a estrutura criada, acessamos diretamente o próximo elemento não nulo na coluna em questão. [3]

2.3.1 Principais funções

A função *elimGauss* recebe como parâmetro um ponteiro para a estrutura *sparse*. São executados as iterações e modificações como indicadas no algoritmo abaixo. A função imprime na tela o tempo de execução do algoritmo e o vetor solução.

2.4 SOR

Para o desenvolvimento do algoritmo SOR através de diversas pesquisas, pode ser constatado que em média a melhor forma de armazenamento com bons resultados para trabalhar com matrizes esparsas para métodos de iterações é o método de armazenamento CSR, porém para a adequação do método de entrada fornecida foi necessário usar um método de armazenamento auxiliar o COO ao qual usa três vetores para armazenar a matriz, logo armazenado no método COO pode ser convertido para o método de armazenamento CSR, ao qual foi desenvolvido o algoritmo SOR para trabalhar com este método de armazenamento.

O algoritmo para a resolução de matriz esparsas por método iterativo SOR, foi construído na função SOR, ao qual recebe como parâmetro de entrada a matriz armazenada

no modo CSR, o fator de relaxação, a tolerância de erro e o número máximo de iterações, após invocada a função, ela constrói o vetor solução recebendo o vetor independente e dividindo pela diagonal correspondente a linha do vetor independente. Após o vetor solução construído é hora de realizar as iterações onde se conta a iteração, e o erro, desta forma as iterações são realizadas através de *loops* que descartam os valores nulos percorrendo através da estrutura CSR para calcular os valores de Xi de modo eficiente, a cada *loop* de iteração é calculado o erro para decidir se a tolerância aceita o erro ou não, e também é calculado a iteração para verificar se a iteração está dentro do limite máximo de iterações. A função SOR tem como saída o vetor solução do sistema fornecido como entrada.

```
Enquanto ERRO > e faca
ERRO=0
NORMAX = 0
Para i=1,...,n
    INICIO = PNTA(i)
    FIM = PNTA(i+1) 1
    Se INICIO FIM faca
        SOMA = b(i)
        Para k = INICIO,...,FIM
            SOMA = SOMA VALA (k) X (INDA(k))
        SOMA = SOMA/ DIAGA(i)
    aux = SOMA + (1 ) X(i)
    Se|aux| > NORMAX, faca NORMAX =|aux|
    Se| a u x X (i)|>ERRO, faca ERRO =| a u x X (i)|
    X(i)=aux
aux = 1
Se NORMAX > 1, faca aux = NORMAX
ERRO=ERRO/aux

ONDE
VALA = VETOR CORRESPONDENTE AOS VALORES N O NULOS E N O DIAGONAIS DA
    MATRIZ
DIAGA = VETOR CORRESPONDENTE AOS VALORES DA DIAGONAL PRINCIPAL DA MATRIZ
PNTA = VETOR QUE ARMAZENA O INDICE ONDE COMEA CADA LINHA DA MATRIZ
INDA = VETOR ONDE ARMAZENA A COLUNA ONDE EST ARMAZENADO CADA ELEMENTO N O
     NULO E N O DIAGONAL DA MATRIZ
```

Listing 7 – Função SOR

2.4.1 Principais funções

```
double *sor(CSR* matrix,double omega,double toler, size_t iterMax)
{
  struct timespec Start,End;
  clock_gettime(CLOCK_MONOTONIC,&Start);

  double *x;
  double soma,normax,aux,Elapsed_Time,erro;
  size_t i,k,iter,inicio,fim;

  x = calloc(matrix->n,sizeof(double));

  iter = 0;
  do{
    erro = 0;
    normax = 0;
    iter++;
    for(i = 0; i < matrix->n; i++) {
      inicio = matrix->row_ptr[i];
```

```
19        fim = matrix->row_ptr[i+1]-1;
20        soma = matrix->b[i];
21        if(inicio <= fim) {
22          for(k = inicio; k <= fim; k++)
23            soma -= matrix->A[k]* x[matrix->column_index[k]];
24        }
25        soma = soma / matrix->d[i];
26        aux = omega * soma + (1 - omega) * x[i];
27
28        if(fabs(aux) > normax)
29          normax = fabs(aux);
30
31        if(fabs(aux - x[i]) > erro)
32          erro = fabs(aux - x[i]);
33
34        x[i] = aux;
35      }
36      aux = 1;
37      if(normax > 1)
38        aux = normax;
39
40      erro = erro/aux;
41
42    } while(erro > toler && iter < iterMax);
43
44    printf("Iter: %lu - IterMax: %lu\n",iter,iterMax);
45
46    clock_gettime(CLOCK_MONOTONIC, &End);
47    Elapsed_Time = End.tv_sec - Start.tv_sec + 1e-9*(End.tv_nsec - Start.
        tv_nsec);
48    printf("O tempo de execucao gasto foi %g segundos\n",Elapsed_Time);
49
50    return x;
51  }
```

Listing 8 – Pseudo código SOR

3 Experimentos Numéricos

Foram realizados uma série de testes para verificar a velocidade e quantidade de memória utilizada para cada uma das estruturas. Oficialmente, foram propostas cinco matrizes diferentes com esforço computacional crescente foram anotados para se estabelecer uma comparação. Estes testes foram executados em uma máquina com as seguintes configurações:

Linux Mint 17.2 Cinnamon 64-bit
Intel Core i7-3770 CPU @ 3.40GHz x 4
8GB de memória

Veja na Tabela 1 os resultados obtidos com os programas apresentados.

Os testes do SOR foram executados com *tolerância de erro* $= 10^{-5}$.

Para a realização de experimentos, testes e alcançar conclusões foram utilizados diversos métodos, a partir de pesquisas em artigos, livros, uso de softwares de *debug* como *gdb* e *valgrind* e softwares matriciais como *octave* e *matlab*.

Experimentos Gauss:
Em respeito ao método de Gauss adaptado a matrizes esparsas, pode-se dizer que

			Eliminação de Gauss	Método SOR		
Matriz	**n**	**nnz**	**Tempo**	**Iterações**	ω	**Tempo**
rdb968	968	5632	1.188885s	132302	0.0000059	3.93555s
rail5177	5177	35185	1.172265s	2374	0.02	1.67600s
aft01	8205	125567	210.353285s	5069	1.9	2.38747s
FEM3D_thermal1	17880	430740	1210.133500s	112	0.5	0.168656s
Dubcova2	65025	1030225	11752.613021s	5111	1	20.4685s

Tabela 1 – Resultados numéricos

houve um ganho significante de velocidade e uma redução de espaço de memória utilizado. Porém não se compara com os métodos iterativos, que são muito facilmente aplicáveis a este tipo de sistema linear. O método de Gauss é apropriado para matrizes densas, por isto pode ser esperado a complexidade elevada.

Infelizmente não foi possível testar todas as matrizes propostas por grande quantidade de tempo necessário.

Experimentos SOR:

Através de testes realizados, pode ser constatado que existem sistemas que convergem para o método de SOR e não para o de Gauss-Seidel, e que segundo o teorema de convergência do raio espectral, pode ser visto que quanto mais próximo de 1 o módulo do maior auto valor da matriz M, mais lento será a convergência do sistema, desta forma muitas vezes o sistema terminando por número máximo de iterações e não de convergência, e que o fator de relaxação pode ajudar em uma convergência mais rápida do sistema, entretanto pode atrapalhar causando muitas vezes uma convergência mais lenta do que a comum equivalente ao método de Gauss-Seidel, também pode ser constatado em testes que mesmo muito próximo o valor do método iterativo do valor real, ele é uma aproximação não exata do resultado da resolução do sistema, porém sendo muito satisfatória o seu tempo de execução comparado ao método direto de gauss, convergindo matrizes extremamente grandes como as fornecidas com uma margem de erro aceitável em um tempo ótimo, comparado a um método direto.

Em nosso método SOR, não foi possível alcançar o valor de alguns testes com precisão devido a lentidão de convergência, ou com a dimensão da divergência de forma ao erro a ser diminuído a cada iteração, mas não o suficiente para convergir antes que alcance o número máximo de iterações, desta forma não conseguindo alcançar com precisão alguns testes, porém em testes menores montados foram realizado de forma eficaz.

Conclusão

Através deste trabalho pode ser constatado na prática como funciona o método direto de gauss e o método iterativo SOR, para o trabalho com matrizes esparsas, pode ser visto que os métodos diretos não foram feitos para trabalhar com matrizes esparsas em seus modos simples de armazenamento como por exemplo o CSR, sendo de forma muito trabalhosa a adaptação deste método para este tipo de trabalho, porém, em caso contrário, o método iterativo SOR trabalha de forma muito satisfatória com os métodos de armazenamento para matriz esparsa, como por exemplo o CSR, logo pode ser visto que os resultados em relação a tempo foram vantajosos para o método iterativo, realizando a resolução de sistemas e tempo aceitável, porém o método direto fornecendo uma precisão

melhor. Outro grande ponto a ser abordado com cuidado é em relação ao método direto existir sempre uma solução em todos os casos, porém já o método iterativo em alguns casos não convergir, desta forma sendo inviável a utilização dele nestes casos, através de todos os pontos observados pode ser concluir que o método a ser usada é relativo aos casos a serem trabalhados, desta forma sendo inviável uma recomendação de melhor modelo sem o estudo de que tipos de matrizes esparsas serão trabalhadas. Mas de modo geral, pode ser destacado que desde que as matrizes sejam convergentes para o método SOR, ele vantagem devido a tempo de resolução do sistema.

Solving Sparse Linear Systems

André Barreto, Igor Venturini e Vinicius Arruda

Universidade Federal do Espírito Santo
Departamento de Informática

11 de Outubro, 2015

Abstract

In this research we will approach solving sparse linear systems, presenting methods and algorithms, and, at last, number results from a series of tests. For this matter, we will implement sparse compactation methods and linear systems resolutions for sparse matrix.

Keywords: sparse matrix. linear system.

Referências

Susan Blackford. *Compressed Row Storage*, November 2000. 3

Xiaoye S. Li. *Sparse Gaussian Elimination on High Performance Computers*, 1996. 7

Wikipedia. *Sparse Matrix*, September 2015. 1