Universidade Federal do Espírito Santo
Departamento de Informática

André Barreto

Método das Diferenças Finitas Aplicado a Problemas Bidimensionais

Trabalho 1 de Algoritmos Numéricos 2

Vitória
2016

1 Introdução

Equações diferenciais parciais, aparecem com frequência na solução de problemas em diversas a áreas do conhecimento. Sendo assim, vamos estudar o processo de discretização pelo método das diferenças finitas de equações do tipo:

$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + a\frac{\partial u}{\partial x} + b\frac{\partial u}{\partial y} + cu = f(x,y) \text{ em } \Omega \tag{1}$$

considerando que *a, b, c e f(x,y)* são conhecidas e que $u(x,y)$ é conhecida no contorno de Ω. Deseja-se obter a solução $u(x,y)$ no interior de um domínio retangular de dimensões $(x_0, x_{N+1}) \times (y_0, y_{M+1})$, sendo conhecida a solução no contorno do retângulo (condições de Dirichlet). Considere uma subdivisão do domínio em células retangulares, sendo $N+1$ divisões na horizontal e $M+1$ divisões na vertical, respectivamente, de dimensões h_x e h_y.

Este trabalho tem como objetivo discretizar a equação (1) pelo método das diferenças finitas e resolver o sistema linear resultante através dos métodos de Eliminação Gaussiana e Gauss-Seidel com relaxação sucessiva (SOR). Os algoritmos devem ser implementados tanto em linguagem C como em Octave.

Será feita uma análise comparativa dos métodos de solução considerados e das linguagens utilizadas.

Além disso, serão utilizadas técnicas de armazenamento e resolução de matrizes esparsas, ou seja, salvando apenas os elementos indispensáveis para a resolução de do sistema linear.

2 Referencial Teórico

A discretização da equação (1) resulta em um sistema linear cuja matriz de coeficientes é uma matriz penta-diagonal [**?**], como ilustrado a seguir.

$$\begin{pmatrix}
a_1 & c_1 & & e_1 & & & & & \\
b_2 & a_2 & c_2 & & e_2 & & & & \\
 & \ddots & \ddots & \ddots & & \ddots & & & \\
d_{n+1} & & b_{n+1} & a_{n+1} & c_{n+1} & & e_{n+1} & & \\
 & \ddots & & \ddots & \ddots & \ddots & & \ddots & \\
 & & d_I & & b_I & a_I & c_I & & e_I \\
 & & & \ddots & & \ddots & \ddots & \ddots & \\
 & & & d_{N-1} & & & b_{N-1} & a_{N-1} & c_{N-1} \\
 & & & & d_N & & & b_N & a_N
\end{pmatrix}$$

Para a resolução do sistema linear penta-diagonal resultante após o processamento dos dados de entrada, serão utilizadas os métodos de resolução adaptados à matrizes esparsas já implementados pelo trabalho anterior: Método de Gauss e SOR.

Neste trabalho, foram utilizadas diversas técnicas para a resolução. Em primeiro momento foram criadas funções requisitadas pelo trabalho e uma função de discretização ao qual considera N a quantidade de pontos X na malha e M a quantidade de pontos Y. Após isso serão armazenados $N \times M$ pontos. Uma função de criação da matriz, onde criamos a matriz penta diagonal, onde a,b,c são constantes. De tal forma que a matriz tenha todos os valores de A,B,C,D,E iguais.

Também é criado uma função de construção do vetor independente, ao qual foi utilizado a biblioteca *ae* da linguagem lua e a função *eval* de Octave, para que possa ser interpretada a função passada como parâmetro. Após isso são aplicados os pontos na função, montando o vetor independente. Tendo a matriz e o vetor independente foi substituído da matriz os pontos de contorno fornecidos, construindo assim o sistema esparso, onde, resolvendo ele, conseguimos adquirir os valores de $U(x, y)$.

Para resolver o sistema, foram utilizados o programa adquirido no trabalho passado e o *sor* e *gauss* do Octave vistos em sala. Desta forma foi possível encontrar o resultado de nossa equação em cada ponto discretizado.

3 Implementação

Este trabalho foi desenvolvido tanto em linguagem C quanto em Octave. Além disso, criamos uma interface gráfica em *web* para facilitar a entrada de dados do usuário. Veja nesta seção as partes relevantes dos códigos e a apresentação da interface.

3.1 Linguagem C

Em linguagem C, criamos o seguinte programa principal que chama as principais funções para a resolução do problema:

```
int main(int argc, char **argv)
{
  (...)
  input = readData(&a, &b, &c);
  lPoints = createLPoints(input->amountX, input->amountY);
  discretiza(lPoints,input,&hx, &hy);
  matrix = createMatrix(input,a, b, c);
  vetorIndependent = createVIndependent(lPoints,input);
  insertContourn(matrix, vetorIndependent, input);
  writeMatrix(matrix, vetorIndependent,input);
  (...)
}
```

Listing 1: Função *main*

Após a chamadas destas funções, tem-se como resultado um matriz escrita em um arquivo em formato próprio para manipulação esparsa, que então é lido pelos algoritmos Gauss e SOR.

3.1.1 Estruturas

Para auxiliar na manipulação dos dados de entrada, foram criadas estas duas estruturas de armazenamento, explicitadas no código abaixo.

```
typedef struct data
{
  double beginX;
  double endX;
  double beginY;
  double endY;
  int amountX;
  int amountY;
  int contour;
  Contour *elements;
} Data;

typedef struct contour
{
  int x;
  int y;
  double value;
} Contour;
```

Listing 2: Estruturas de entrada

Além destas, existe a estrutura *Ponto*, que simplesmente armazena as coordenadas *(x,y)*.

3.1.2 Principais funções

A função *discretiza* é responsável por gerar o vetor de pontos discretizados a partir dos intervalos e números de subintervalos recebidos.

```
1  void discretiza(Points *lPoints, Data *input)
2  {
3    int i,j,pos = 0;
4    double hx,hy;
5
6    hx = (input->endX - input->beginX)/(double)input->amountX-1;
7    hy = (input->endY - input->beginY)/(double)input->amountY-1;
8
9    for( i = 1; i <= input->amountX;i++)
10     for( j = 1; j <= input->amountY;j++)
11     {
12       lPoints[pos].x = input->beginX + (double)(j - 1)*(hx);
13       lPoints[pos].y = input->beginY + (double)(i - 1)*(hy);
14       pos++;
15     }
16 }
```

Listing 3: Função Discretiza

A função *createMatrix* gera uma matriz de tamanho $n \times n$ onde n é a quantidade de elementos definidos pelos valores de entrada.

```
1  double **createMatrix(Data *input,double a, double b, double c)
2  {
3    double hx,hy, **matrix;
4    double aI,bI,cI,dI,eI;
5    int qtdElementos, i, j;
6
7    qtdElementos = (input->amountX * input->amountY);
8
9    hx = (input->endX - input->beginX)/(double)input->amountX-1;
10   hy = (input->endY - input->beginY)/(double)input->amountY-1;
11
12   aI = c + 2 *((1/(hx*hx)) + (1/(hy*hy)));
13   bI = (-1/(hx*hx)) - (a/(2*hx));
14   cI = (-1/(hx*hx)) + (a/(2*hx));
15   dI = (-1/(hy*hy)) - (b/(2*hy));
16   eI = (-1/(hy*hy)) + (b/(2*hy));
17
18   matrix = calloc((size_t)qtdElementos,sizeof(double*));
19   for(i = 0; i < qtdElementos; i++)
20     matrix[i] = calloc((size_t)qtdElementos,sizeof(double));
21
22   for(i = 0; i < qtdElementos; i++)
23     for( j = 0; j < qtdElementos; j++)
24     {
25       if( i == j)
26         matrix[i][j] = aI;
27       else if ((j+1) == i)
```

```
28          matrix[i][j] = bI;
29        else if((j-1) == i)
30          matrix[i][j] = cI;
31        else if((j+input->amountX) == i)
32          matrix[i][j] = dI;
33        else if ((j-input->amountX) == i)
34          matrix[i][j] = eI;
35        else
36          matrix[i][j] = 0;
37      }
38
39    return matrix;
40  }
```

Listing 4: Função Cria Matriz

A função *insertContourn* atualiza a matriz penta-diagonal com os valores conhecidos do contorno.

```
1  void insertContourn(double **matrix, double *vetIndependent,
       Data *input)
2  {
3    int i,j, qtdElementos;
4    int index;
5    qtdElementos = (input->amountX * input->amountY);
6
7    for( j = 0; j < input->contour;j++)
8    {
9      index = generatorNewIndex(input->elements[j].x,input->
       elements[j].y,input->amountX);
10     vetIndependent[index] = input->elements[j].value;
11     for(i = 0; i < qtdElementos ; i++)
12     {
13       if(i == index)
14       {
15         matrix[index][i] = 1;
16       }else
17         matrix[index][i] = 0;
18     }
19   }
20 }
```

Listing 5: Função Insere Contorno

Aplicadas as funções anteriores, é chamada a *writeMatrix*, que escreve, em formato esparso, a matriz em um arquivo de saída.

3.2 Octave

Foi traduzido o procedimento da linguagem C apresentado para Octave.

3.2.1 Principais funções

```
hx = (endX - matrixSparseeginX)/(tamX-1);
hy = (endY - matrixSparseeginY)/(tamY-1);
qtdElementos = tamX*tamY;
lPoints = zeros(qtdElementos,2);
pos = 1;
for i=1:tamX
  for j=1:tamY
    lPoints(pos,1) = matrixSparseeginX + (j-1)*hx;
    lPoints(pos,2) = matrixSparseeginY + (i-1)*hy;
    pos++;
  end
end
```

Listing 6: Função Discretiza

```
matrix = zeros(qtdElementos,qtdElementos);
aI = c + 2 *((1/(hx*hx)) + (1/(hy*hy)));
matrixSparseI = (-1/(hx*hx)) - (a/(2*hx));
cI = (-1/(hx*hx)) + (a/(2*hx));
dI = (-1/(hy*hy)) - (matrixSparse/(2*hy));
eI = (-1/(hy*hy)) + (matrixSparse/(2*hy));

for i=1:qtdElementos
  for j=1:qtdElementos
    if i == j
      matrix(i,j) = aI;
    elseif (j+1) == i
      matrix(i,j) = matrixSparseI;
    elseif (j-1) == i
      matrix(i,j) = cI;
    elseif (j+tamX) == i
      matrix(i,j) = dI;
    elseif (j-tamY) == i
      matrix(i,j) = eI;
    else
      matrix(i,j) = 0;
     endif
  end
end
```

Listing 7: Função Cria Matriz

```
for j=1:contour
  index = A(j,1) + tamX * (A(j,2)-1);
  vetIndependent(index) = A(j,3);
  for i=1:qtdElementos
    if i == index
      matrix(index,i) = 1;
    else
      matrix(index,i) = 0;
    endif
  end
```

```
11 end
```

Listing 8: Função Insere Contorno

Após isto, também é chamada uma função para escrever a matriz em um arquivo de saída.

3.3 Interface web

A interface *web* está localizada no endereço http://antrab2.inf.ufes.br/. Esta interface foi criada para facilitar a entrada de dados: permite-se enviar um arquivo ou digitar os dados. *Importante*: deve-se inserir sem espaços a expressão de F(x,y), tanto no arquivo de texto quanto digitando manualmente no formulário. Veja a figura **??** da interface.

Também é possível executar o programa em modo texto, porém em módulos separados da linguagens. *Pré-requisito*: ter o lua instalado. Para instalar o lua, pode-se executar *make linux install* na pasta do lua anexada ao trabalho.

Em C, executa-se da seguinte forma:

./trab2 [Algoritmo] [Metodo de Entrada] [Arquivo?] [Omega?] [Tolerancia?]

Onde em **Algoritmo** insere-se *gauss* ou *sor*; em **Método de Entrada**, *-t* para teclado ou *-f* e o *arquivo* a ser executado; e caso o algoritmo seja o SOR, insere-se os parâmetros de *relaxação* e *tolerância*.

Em Octave, deve-se executar o seguinte:

octave –silent –eval “trab(’[Algoritmo]’,[Omega?])” -qf

Onde o arquivo de entrada deve estar numa pasta um diretório acima e se chamar “*input.txt*”.

4 Experimentos Numéricos

Serão apresentados agora os testes realizados para a validação dos algoritmos desenvolvidos.

4.1 *Hardware* utilizado

Os testes nesta seção 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

4.2 Testes

A seguir serão caraterizados os dois testes realizados. Foram aplicados para as malhas 8×8, 16×16, 32×32, 64×64, 128×128 e 256×256 o teste 2 e apenas as duas primeiras malhas para o teste 1.

4.2.1 *test1*: Equação de Laplace

Um caso simples de aplicação é a condução de calor em uma placa plana com condições de contorno constantes e iguais. Em condições ideias de condutividade a variação da temperatura $T(x, y)$ em uma placa retangular satisfaz a equação de Laplace, um caso particular da equação (1):

$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = 0 \text{ em } (x_0, x_{N+1}) \times (y_0, y_{M+1}) \tag{2}$$

Ou seja, neste caso temos que $a = b = c = 0$ e $f(x, y) = 0$.

Veja na figura **??** o gráfico do teste Laplace com uma malha 8×8 e valores de contorno iguais à 1.

Como esperado, devido à equação de Laplace, todos os valores da malha resultaram em 1.

Em outro teste, também de malha 8×8, valoramos o contorno da malha com valores diferentes: 10 e 20. Veja o gráfico resultante na figura **??**.

4.2.2 *test2*

Para este teste, temos as seguintes configurações: Considere a equação (1) em $\Omega = \{(x, y) : 0 \leq x, y \leq 1\}$ onde $a = 60$, $b = 80$, $c = -40$ e $f(x, y)$ é tal que $u(x, y) = xe^{xy}sen(\pi x)cos(\pi y)$.

Veja na figura **??** o gráfico deste teste com uma malha 8×8 e valores de contorno aplicados na função $u(x, y)$.

4.3 Comparações

Veja as tabelas 1 e 2, dos testes 1 e 2, respectivamente, de comparação entre os algoritmos e as linguagens em questão de eficiência.

Através dos testes completados, vemos que a solução de problemas de valor de contorno bidimensionais é um processo muito custoso mesmo para

Malha	Linguagem	Algoritmo	Tempo de execução
8×8	C	Gauss	0.002s
		SOR	0.002s
	Octave	Gauss	1.158s
		SOR	∞
16×16	C	Gauss	0.017s
		SOR	0.006s
	Octave	Gauss	1m1.133s
		SOR	∞

Tabela 1: Comparações do *test1*

casos aparentemente pequenos. Por exemplo, uma malha 16×16 gera uma matriz esparsa penta-diagonal de ordem *256*, e uma malha 256×256 geraria uma matriz de ordem *65536*. O que remete à importância do tratamento especial esparso.

É possível afirmar também que os algoritmos em C são significantemente mais rápidos devido à complexidade da linguagem se comparado ao Octave.

Infelizmente, não foi possível realizar muitos dos testes propostos por inviabilidade do sistema. Os casos de malha 32 para cima geravam entradas, ao aplicar na $u(x, y)$, inviáveis ao sistema, resultando assim em saídas corrompidas e nada aceitáveis. Porém ainda registramos os tempos de processamento destes casos. Os tempos marcados como ∞ significam que o processo foi longo demais para ser anotado.

5 Conclusão

Podemos ver que encontrar o resultado de equações diferenciais não é uma tarefa tão trivial, porém algo muito utilizado e útil. Os sistemas penta-diagonais com os quais estamos trabalhando, podem ser muito complexos de serem gerados e resolvidos. Em nosso caso, consideramos a, b e c como constantes, o que facilitou a montagem da matriz penta-diagonal, no entanto, nem por isso se tornou um sistema simplório de se resolver. Também pode ser observado que quanto maior a discretização do domínio, terei mais precisão na malha, consequentemente, sabe-se o valor de mais pontos na malha. Pode ser visto, que resolver equações de Laplace são muito mais simples de serem resolvidas.

Para os problemas de contorno é essencial boas técnicas de arrendondamento, e boas formas de otimização para a resolução de sistemas esparsos, devido a malhas muito grandes gerarem sistemas enormes, no qual $N \times M$ é

Malha	Linguagem	Algoritmo	Tempo de execução
8×8	C	Gauss	0.002s
		SOR	0.003s
	Octave	Gauss	1.151ss
		SOR	∞
16×16	C	Gauss	0.022s
		SOR	0.008s
	Octave	Gauss	1m0.799s
		SOR	∞
32×32	C	Gauss	0.346s
		SOR	0.034s
	Octave	Gauss	∞
		SOR	∞
64×64	C	Gauss	19.602s
		SOR	0.172s
	Octave	Gauss	∞
		SOR	∞
128×128	C	Gauss	∞
		SOR	1.603s
	Octave	Gauss	∞
		SOR	∞
256×256	C	Gauss	∞
		SOR	∞
	Octave	Gauss	∞
		SOR	∞

Tabela 2: Comparações do *test2*

a ordem da matriz do sistema, e com isso podendo levar a divergência do sistema muito rápido no *sor*, e várias horas de cálculos sem sucesso no *gauss*. Também pode ser observado, que a linguagem Octave é muito mais simples, sendo assim aconselhável sua utilização para resolver vários problemas, mesmo possuindo execução mais lenta.

Por fim, através de nossos testes, podemos afirmar que a resolução de problemas de contorno de modo iterativo, pode alcançar resultados bons e mais rapidamente, porém da mesma forma, pode não alcançar resultados satisfatórios para outros caso, sendo assim de suma importância uma boa estratégia para a abordagem do problema de resolução de matrizes esparsas.