Universidade Federal do Espírito Santo
Departamento de Informática

André Barreto Silveira

Método das Diferenças Finitas Aplicado a Problemas Bidimensionais

Trabalho 1 de Algoritmos Numéricos 2

Vitória
2016

1 Introdução

O estudo da equação de transporte, também denominada equação da advecção-difusão-reação, continua sendo um ativo campo de pesquisa, uma vez que essa equação é de fundamental importância nos problemas relacionados a aerodinâmica, meteorologia, oceanografia, hidrologia, engenharia química e de reservatórios. A equação de transporte tem características bastante peculiares que fazem com que sua resolução por meios numéricos seja dificultada em situações onde o problema é fortemente convectivo. Por isso, diversos métodos têm sido desenvolvidos e aplicados, com a intenção de superar as dificuldades numéricas impostas por esta equação.

A equação de transporte bidimensional pode ser definida por:

$$-k\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+\beta_x(x,y)\frac{\partial u}{\partial x}+\beta_y(x,y)\frac{\partial u}{\partial y}+\gamma(x,y)u=f(x,y) \text{ em } \Omega \quad (1)$$

Este trabalho tem como objetivo verificar como as formas de armazenamento das estruturas resultantes pela discretização da equação (1) por diferenças finitas pode impactar no tempo de processamento.

Para isto, será implementado em linguagem C um programa capaz de aplicar a discretização da equação de transporte e resolver o sistema resultante discreto. Quanto ao método de resolução, será aplicado o algoritmo SOR (*Sucessive Over Relaxation*) de duas formas: resolvendo o sistema penta-diagonal armazenado de forma esparsa, utilizando cinto vetores; e completamente livre de matriz.

Nas próximas seções estão descritos em detalhes o desenvolvimento, testes e conceitos do trabalho, sendo que o próximo tópico faz um resumo do método das Diferenças Finitas, descrevendo as técnicas e ordem de aproximação usadas.

2 Método das Diferenças Finitas

Método das Diferenças Finitas é um método numérico para resolver equações diferenciais através de aproximações de derivadas por diferenças finitas. Estas aproximações são obtidas pela série de Taylor:

$$f(x+h)=f(x)+f'(x)h+\frac{f''(x)h^2}{2}+\frac{f'''(x)h^3}{6}+o(h^4)$$
$$f(x-h)=f(x)-f'(x)h+\frac{f''(x)h^2}{2}-\frac{f'''(x)h^3}{6}+o(h^4)$$

Portanto, a primeira e segunda derivadas podem ser escritas das seguintes formas:

$$f'(x) = \frac{f(x+h) - f(x)}{h} + o(h) \text{ , diferença atrasada } (1^{\text{a}} \text{ ordem})$$
$$f'(x) = \frac{f(x) - f(x-h)}{h} + o(h) \text{ , diferença adiantada } (1^{\text{a}} \text{ ordem})$$
$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + o(h^2) \text{ , diferença central } (2^{\text{a}} \text{ ordem})$$
$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + o(h^2)$$

O método das Diferenças Finitas em domínios retangulares consiste em **três** etapas principais:

- **Discretização do domínio**, onde os pontos (x, y) são traduzidos em um domínio linear, de forma que $I = (j-1)n + i$ e $1 \leq I \geq n * m$, onde i e j são representam um ponto bidimensional e I uma posição representativa deste ponto.
- **Aproximar a Equação** (1) **por diferenças finitas**, que consiste em reescrever a equação utilizando as aproximações por diferenças finitas, gerando um sistema linear.
- **Aplicar condições de contorno** que são definidas pelo problema proposto e que impactam diretamente na solução do problema.

Após a aplicação das etapas acima, teremos como resultado um sistema penta-diagonal da seguinte forma:

$$e_I u_{I-n} + c_I u_{I-1} + a_I u_I + b_I u_{I+1} + d_I u_{I+n} = f_I \tag{2}$$

onde

$$e_I = \left[\frac{-1}{h_y^2} - \frac{\beta_y}{2h_y}\right]$$
$$c_I = \left[\frac{-1}{h_x^2} - \frac{\beta_x}{2h_x}\right]$$
$$a_I = \left[\gamma_I + \frac{2}{h_x^2} + \frac{2}{h_y^2}\right]$$
$$b_I = \left[\frac{-1}{h_x^2} + \frac{\beta_x}{2h_x}\right]$$
$$d_I = \left[\frac{-1}{h_x^2} + \frac{\beta_y}{2h_y}\right]$$

e a matriz de coeficientes é penta-diagonal, como ilustrado a seguir:

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

3 Implementação

Nesta seção estão apresentados os códigos em C relevantes do sistema que mostra suas principais funcionalidades necessárias para a resolução do Método das Diferenças Finitas Bidimensional.

O sistema foi desenvolvido com base nas três principais etapas do método das diferenças finitas: discretizar o domínio, realizar a aproximação por diferenças finitas e aplicar condições de contorno. Em todos os experimentos estas etapas são necessárias, porém, devido às diferenças algorítmicas entre os experimentos, o programa seria muito complexo se fosse feito de forma genérica. Por conta disto, e almejando um programa eficiente para a solução dos problemas, o código foi escrito de forma condicional, onde dependendo do experimento indicado para resolver, partes diferentes do programa serão utilizadas. Isto implica em um código de baixa manutenibilidade, devido à grande repetição de códigos, mas ganha em eficiência, como desejado.

3.1 Estruturas

Estão ilustradas nesta seção as estruturas de dados utilizadas para a resolução do método das diferenças finitas.

A estrutura *Dados* é a base de dados primordiais para a aplicação do algoritmo. Esta estrutura existe para facilitar o acesso ao conjunto de dados que são utilizados em diversas funções do sistema. Estes dados são formados pelas informações sobre o domínio bidimensional e os parâmetros do algoritmo SOR.

```
1 typedef struct dados
2 {
3   double inicioX;
4   double fimX;
```

```
  double inicioY;
  double fimY;
  int qtdX;
  int qtdY;
  double omega;
  double tolerancia;
  size_t iterMax;
} Dados;
```

Listing 1: Estrutura de entrada

Ponto é uma simples estrutura que caracteriza um ponto (x, y), que é utilizada para facilitar o processo de discretização, onde deve-se obter um vetor de pontos que será utilizado para as seguintes etapas do método.

```
typedef struct ponto
{
  double x;
  double y;
} Ponto;
```

Listing 2: Estrutura Ponto

A *MatrizPentadiagonal* é a estrutura que armazena os cinco vetores dos coeficientes da matriz penta-diagonal resultante. Esta é utilizada caso seja selecionado o método de resolução pelo SOR "normal", que utiliza uma matriz.

```
typedef struct matrizPentadiagonal
{
  double *e;
  double *c;
  double *a;
  double *b;
  double *d;
    size_t N;       // Ordem da matriz Pentadiagonal
    size_t tamED;   // Tamanho dos vetores *e e *d
} MatrizPentadiagonal;
```

Listing 3: Matriz Pentadiagonal

SistemaLinear é a abstração mais alta que representa o sistema penta-diagonal, contendo a matriz penta-diagonal de coeficientes, o vetor independente do sistema e a ordem do mesmo. Esta estrutura só é utilizada caso seja o SOR "normal" a ser operado.

```
typedef struct sistemaLinear
{
    MatrizPentadiagonal* matriz;
    double* f; // Vetor independente
    size_t N;  // Ordem do sistema
} SistemaLinear;
```

Listing 4: Sistema Linear

3.2 Funções

A função de discretização é a primeira a ser chamada no processo de fato. Esta gera o vetor de pontos que será utilizado nas próximas etapas do sistema.

```
Ponto* discretizaDominio(Dados* dados)
{
  int i, j, n, m, pos;
  double hx, hy;
  Ponto *vetorPontos;

    n = dados->qtdX;
    m = dados->qtdY;
  hx = (dados->fimX - dados->inicioX)/((double)n-1);
  hy = (dados->fimY - dados->inicioY)/((double)m-1);

  vetorPontos = calloc((size_t)(n*m),sizeof(Ponto));

  pos = 0;
  for(i = 1; i <= dados->qtdX; i++)
    for(j = 1; j <= dados->qtdY; j++)
    {
      vetorPontos[pos].x = dados->inicioX + (double)(j - 1)*(hx)
    ;
      vetorPontos[pos].y = dados->inicioY + (double)(i - 1)*(hy)
    ;
      pos++;
    }

    return vetorPontos;
}
```

Listing 5: Função de Discretização

Após a discretização, para um SOR comum, teríamos que gerar a matriz penta-diagonal de coeficientes, criar o vetor independente, criar o sistema linear que contém os dois últimos e por fim aplicar as condições de contorno no sistema gerado. Então aplica-se o algoritmo SOR. Para cada um desses passos, existe uma função correspondente, mas como mencionado previamente, o código é condicional. Ou seja, as funções são interfaces que identificarão qual é o experimento a ser resolvido e chamarão a função correta para resolver tal experimento.

Seria muito extenso mostrar todas as funções. Por conta disto, serão apresentadas a seguir as funções executadas para o experimento Aplicação Física 1 (ver seção 4.3).

```
void montaMatrizA1(MatrizPentadiagonal* matriz, Dados* dados)
{
    size_t i;
    double hx, hy;
```

```
    double c, T;

    // Constantes do problema
    c = 1;
    T = 2;

    // Calculando hx e hy
  hx = (dados->fimX - dados->inicioX)/((double)dados->qtdX-1);
  hy = (dados->fimY - dados->inicioY)/((double)dados->qtdY-1);

    // Montando a matriz pentadiagonal
    for(i=0; i < matriz->N; i++) {
        matriz->e[i] = (-1/(hy*hy));
        matriz->c[i] = (-1/(hx*hx));
        matriz->a[i] = 2*c/T + 2*((1/(hx*hx)) + (1/(hy*hy)));
        matriz->b[i] = (-1/(hx*hx));
        matriz->d[i] = (-1/(hy*hy));
    }
}
```

Listing 6: Função Montar Matriz A1

```
void montaVetorIndependenteA1(double* vetorIndependente, const
    int N)
{
    int i;
    double c, uRef, T;

    // Constantes do problema
    c = 1;
    T = 2;
    uRef = 70;

    for(i=0; i < N; i++)
        vetorIndependente[i] = 2*c*uRef/T;
}
```

Listing 7: Função Cria Vetor Independente A1

```
SistemaLinear* criaSistemaLinear(MatrizPentadiagonal* matriz,
    double* f, const
size_t N)
{
    SistemaLinear* sistema;
    sistema = malloc(sizeof(SistemaLinear));
    sistema->matriz = matriz;
    sistema->f = f;
    sistema->N = N;
    return sistema;
}
```

Listing 8: Função Cria Sistema Linear

```
void aplicaContornoA1(SistemaLinear* sistema, Dados* dados)
{
```

```
3   int i, j, I;
4     double uRef, c, hx;
5     MatrizPentadiagonal* matriz;
6
7     matriz = sistema->matriz;
8   hx = (dados->fimX - dados->inicioX)/((double)dados->qtdX-1);
9     uRef = 70;
10    c = 1;
11
12    for(i=0, j=dados->qtdY-1; j > 0; j--) {
13        I = indiceDiscreto(i, j, dados->qtdX);
14        sistema->f[I] = 200;
15        matriz->e[I]  = 0;
16        matriz->b[I]  = 0;
17        matriz->a[I]  = 1;
18        matriz->c[I]  = 0;
19        matriz->d[I]  = 0;
20    }
21    for(i=dados->qtdX-1, j=1; j < dados->qtdY-1; j++) {
22        I = indiceDiscreto(i, j, dados->qtdX);
23        // Aplica  o da condi  o de contorno mista
24        // k*parcial(u)/parcial(n) = c(uRef-u(L,y)
25        sistema->f[I] = sistema->f[I] - matriz->b[I]*hx*uRef;
26        matriz->a[I]  = matriz->a[I] + matriz->b[I]*(1-hx*c);
27        matriz->b[I]  = 0;
28    }
29    for(i=0, j=0; i < dados->qtdX; i++) {
30        I = indiceDiscreto(i, j, dados->qtdX);
31        sistema->f[I] = 70;
32        matriz->e[I]  = 0;
33        matriz->b[I]  = 0;
34        matriz->a[I]  = 1;
35        matriz->c[I]  = 0;
36        matriz->d[I]  = 0;
37    }
38    for(i=dados->qtdX-1, j=dados->qtdY-1; i >= 0; i--) {
39        I = indiceDiscreto(i, j, dados->qtdX);
40        sistema->f[I] = 70;
41        matriz->e[I]  = 0;
42        matriz->b[I]  = 0;
43        matriz->a[I]  = 1;
44        matriz->c[I]  = 0;
45        matriz->d[I]  = 0;
46    }
47 }
```

Listing 9: Função Aplica Contorno A1

Com isto, temos o sistema pronto para ser solucionado. Então é chamado o algoritmo SOR:

```
1 double *sor(SistemaLinear* sistema, double omega, double toler,
      size_t iterMax)
2 {
```

```
3   double *x; // Vetor solucao
4   double soma, normaX, normaDif, aux, erro;
5   size_t i, iter, dr;
6     MatrizPentadiagonal* matriz;
7
8   x = calloc(sistema->N, sizeof(double));
9     matriz = sistema->matriz;
10    dr = (matriz->N - matriz->tamED);
11
12    // Iterar ate erro aceitavel ou maximo de iteracoes atingido
13  iter = 0;
14  do {
15        normaX = 0;
16        normaDif = 0;
17        i = 0;
18    iter++;
19
20        soma = matriz->b[i] * x[1] +
21               matriz->d[i] * x[dr];
22
23        // Novo valor de x[i] em auxiliar para calcular norma e
      avaliar erro
24        aux = (1-omega)*x[i] + (omega/matriz->a[i]) * (sistema->
      f[i]-soma);
25
26        // Norma de x[i]
27        if(fabs(aux) > normaX)
28            normaX = fabs(aux);
29
30        // Norma da diferen a (x[i] - x[i-1])
31        if(fabs(aux - x[i]) > normaDif)
32            normaDif = fabs(aux - x[i]);
33
34        // Atualizando valor de x[i]
35        x[i] = aux;
36
37        i++;
38
39        for(; i < dr; i++) {
40            soma = matriz->c[i] * x[i-1] +
41                   matriz->b[i] * x[i+1] +
42                   matriz->d[i] * x[i+dr];
43
44            // Novo valor de x[i] em auxiliar para calcular
      norma e avaliar erro
45            aux = (1-omega)*x[i] + (omega/matriz->a[i]) * (
      sistema->f[i]-soma);
46
47            // Norma de x[i]
48            if(fabs(aux) > normaX)
49                normaX = fabs(aux);
50
51            // Norma da diferen a (x[i] - x[i-1])
52            if(fabs(aux - x[i]) > normaDif)
```

```
53                  normaDif = fabs(aux - x[i]);
54
55              // Atualizando valor de x[i]
56              x[i] = aux;
57          }
58
59          for(i=dr; i < matriz->tamED; i++) {
60              soma = matriz->e[i] * x[i-dr] +
61                     matriz->c[i] * x[i-1]  +
62                     matriz->b[i] * x[i+1]  +
63                     matriz->d[i] * x[i+dr];
64
65              // Novo valor de x[i] em auxiliar para calcular
    norma e avaliar erro
66              aux = (1-omega)*x[i] + (omega/matriz->a[i]) * (
    sistema->f[i]-soma);
67
68              // Norma de x[i]
69              if(fabs(aux) > normaX)
70                  normaX = fabs(aux);
71
72              // Norma da diferen a (x[i] - x[i-1])
73              if(fabs(aux - x[i]) > normaDif)
74                  normaDif = fabs(aux - x[i]);
75
76              // Atualizando valor de x[i]
77              x[i] = aux;
78          }
79
80          for(i=matriz->tamED; i < sistema->N-1; i++) {
81              soma = matriz->e[i] * x[i-dr] +
82                     matriz->c[i] * x[i-1]  +
83                     matriz->b[i] * x[i+1];
84
85              // Novo valor de x[i] em auxiliar para calcular
    norma e avaliar erro
86              aux = (1-omega)*x[i] + (omega/matriz->a[i]) * (
    sistema->f[i]-soma);
87
88              // Norma de x[i]
89              if(fabs(aux) > normaX)
90                  normaX = fabs(aux);
91
92              // Norma da diferen a (x[i] - x[i-1])
93              if(fabs(aux - x[i]) > normaDif)
94                  normaDif = fabs(aux - x[i]);
95
96              // Atualizando valor de x[i]
97              x[i] = aux;
98          }
99
100         soma = matriz->e[i] * x[i-dr] +
101                matriz->c[i] * x[i-1];
102
```

```
103         // Novo valor de x[i] em auxiliar para calcular norma e
    avaliar erro
104         aux = (1-omega)*x[i] + (omega/matriz->a[i]) * (sistema->
    f[i]-soma);
105
106         // Norma de x[i]
107         if(fabs(aux) > normaX)
108             normaX = fabs(aux);
109
110         // Norma da diferen a (x[i] - x[i-1])
111         if(fabs(aux - x[i]) > normaDif)
112             normaDif = fabs(aux - x[i]);
113
114         // Atualizando valor de x[i]
115         x[i] = aux;
116
117     erro = normaDif/normaX;
118   }
119     while(erro > toler && iter < iterMax);
120
121   printf("Numero de iteracoes: %lu\n",iter);
122     printf("Erro: %f\n", erro);
123
124   return x;
125 }
```

Listing 10: Função SOR normal

Ao final destas chamadas, temos o vetor solução do problema.

Para o SOR livre de matriz, estes mesmo procedimentos são executados, porém sem o auxílio das estruturas. Isto significa que em cada iteração, os cálculos devem ser refeitos para saber, por exemplo, o coeficiente e_I, a_I e f_I, inclusive todas as aplicações de contorno também estão presentes no algoritmo.

4 Experimentos Numéricos

Para verificar a qualidade dos métodos implementados, foram realizados uma séries de testes a um conjunto de experimentos propostos. Em cada experimento, o programa em questão é aplicado em diferentes ordens de sistema e em ambas as versões do SOR apresentadas, onde o foco é a comparação dos tempos de execução em cada caso.

Em cada experimento a seguir será ilustrado uma tabela com os casos relevantes de teste e o tempo de execução em segundos que cada computação levou. Os tempos de execução apresentados foram recolhidos utilizando o utilitário *time* do Linux que mede o tempo que o processo ficou rodando.

Algumas abreviações foram utilizadas nestas tabelas, a saber:

- n: partições no eixo X;
- m: partições no eixo Y;
- SOR "normal": algoritmo SOR utilizando a estrutura de armazenado da matriz penta-diagonal;
- SOR "livre": algoritmo SOR livre de matriz.

Além disto, alguns testes levam tempo demais para ser viável computar e comparar ou não houve tempo suficiente. Neste caso, seus tempos de processamentos foram marcados com "∞".

Os testes nesta seção foram executados em uma máquina com as seguintes configurações:

Ubuntu 14.04 64-bit
Intel Core i7-3770 CPU @ 3.40GHz x 4
8GB de memória RAM

Quanto aos parâmetros do algoritmo SOR, os testes foram executados com os seguintes:

$$\begin{aligned} \omega &= 1.6 \\ toler &= 0.00001 \\ iterMax &= 1000000 \end{aligned}$$

onde ω é o coeficiente de relaxação, $toler$ é a tolerância do erro de aproximação e $iterMax$ o número máximo de iterações.

4.1 Validação 1 - Problema simples com solução trivial

Este é um experimento simples para testes do sistema onde deve-se determinar a distribuição de calor em uma chapa de metal, com faces termicamente isoladas e com espessura desprezível, sendo que a temperatura é conhecida em todas as faces da chapa. Neste caso, a equação (1) é dada por:

$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = 0 \text{ em } \Omega \tag{3}$$

Sendo T_0 a temperatura nas faces, espera-se que os valores no interior da placa sejam igualmente T_0 em todos os pontos da discretização.

Veja na tabela 1 os tempos de execução deste experimento por valores n e m e versão do algoritmo SOR.

n	m	SOR	Tempo
25	100	normal livre	0m0.034s 0m0.088s
100	100	normal livre	0m0.220s 0m0.526s
500	1000	normal livre	4m16.887s 10m2.833s
1000	10000	normal livre	10m20.053s 209m4.270s

Tabela 1: Testes - Validação 1

Veja na figura 1 o gráfico da solução encontrada ao aplicar a Validação 1 para $\Omega = (0,1) \times (0,1)$, $n = 100$ e $m = 100$.

A constante T_0, neste caso, era 5.1122. Vemos que a solução não é exatamente como esperamos por conta de erros de aproximação.

Para utilizar outro valor para T_0, deve-se editar o *define* da constante, que encontra-se no arquivo "dados.h".

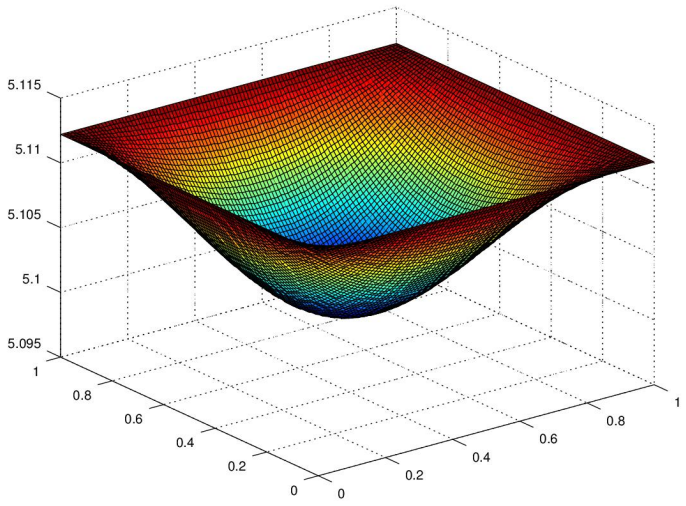

Figura 1: Gráfico solução da Validação 1 para $n = 100$ e $m = 100$

4.2 Validação 2 - Problema com solução conhecida

Neste experimento, deve-se determinar a solução aproximada para $u(x, y)$ em $\Omega = (0, 1) \times (0, 1)$ considerando na Eq. (1):

$$\begin{aligned} k &= 1 \\ \beta_x(x, y) &= 1 \\ \beta_y(x, y) &= 20y \\ \gamma(x, y) &= 1 \\ f(x, y) \text{ tal que } u(x, y) &= 10xy(1 - x)(1 - y)e^{x^{4.5}} \\ &\text{é a solução exata} \end{aligned} \tag{4}$$

e sabendo que $u(x, y) = 0$ no contorno de Ω.

Veja na tabela 2 os tempos de execução deste experimento por valores n e m e versão do algoritmo SOR.

n	m	SOR	Tempo
25	100	normal livre	0m0.041s 0m1.695s
100	100	normal livre	0m0.162s 0m9.440s
500	1000	normal livre	3m34.692s ∞
1000	10000	normal livre	∞ ∞

Tabela 2: Testes - Validação 2

Veja na figura 2 o gráfico da solução encontrada ao aplicar a Validação 2 para $\Omega = (0, 1) \times (0, 1)$, $n = 25$ e $m = 25$, e na figura 3 o gráfico da solução encontrada ao aplicar a Validação 2 para $\Omega = (0, 1) \times (0, 1)$, $n = 80$ e $m = 100$.

4.3 Aplicação Física 1 - Resfriador bidimensional

Este experimento consiste em uma aplicação dos métodos em questão para resfriar uma massa aquecida. Exemplos podem incluir o resfriamento de chips de computadores ou amplificadores elétricos. O modelo matemático que descreve a transferência de calor nas direções x e y é dado por:

$$-k\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + \frac{2c}{T}u = \frac{2c}{T}u_{ref} = 0 \text{ em } \Omega = (0, L) \times (0, W) \tag{5}$$

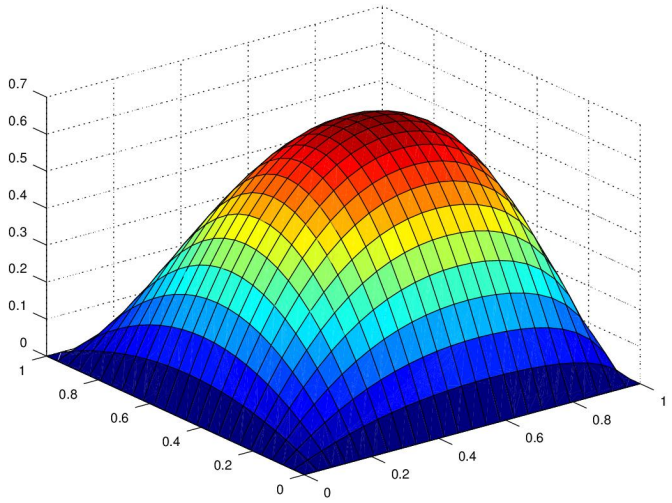

Figura 2: Gráfico solução da Validação 2 para $n = 25$ e $m = 25$

onde k é a condutividade térmica constante, c é o coeficiente de transferência de calor, T é a altura do resfriador e u_{ref} é a temperatura de referência. Deve-se encontrar a temperatura no interior do resfriador considerando as seguintes condições de contorno:

$$\begin{aligned} u(x,0) &= 70 \\ u(x,W) &= 70 \\ u(0,y) &= 200 \\ k\frac{\partial u}{\partial n}(L,y) &= c(u_{ref} - u(L,y)) \end{aligned}$$

Veja na tabela 3 os tempos de execução deste experimento por valores n e m e versão do algoritmo SOR.

Veja na figura 4 o gráfico da solução encontrada ao aplicar a Aplicação Física 1 para $\Omega = (0,1) \times (0,1)$, $n = 25$ e $m = 25$, e na figura 5 o gráfico da solução encontrada ao aplicar a Aplicação Física 1 para $\Omega = (0,1) \times (0,1)$, $n = 25$ e $m = 100$.

Em ambos os testes anteriores, foi-se aplicado a condição de contorno mista. Caso seja aplicado a condição de contorno de valor prescrito $u(L,y) = 70$, e utilizando os dados $\Omega = (0,1) \times (0,1)$, $n = 25$ e $m = 100$, teremos um gráfico de solução ilustrado pela figura 6

Podemos ver que as soluções encontradas para $n = 25$ e $m = 100$ modificando a condição de contorno são muito similares. Inclusive, a norma

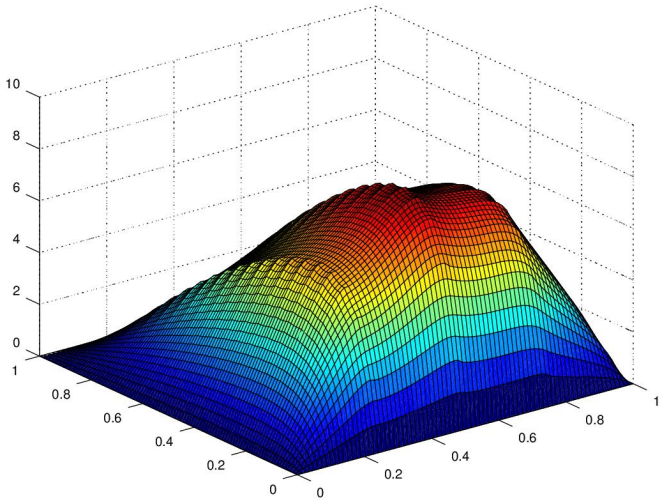

Figura 3: Gráfico solução da Validação 2 para $n = 80$ e $m = 100$

n	**m**	**SOR**	**Tempo**
25	100	normal	0m0.041s
		livre	0m0.095s
100	100	normal	0m0.240s
		livre	0m0.580s
500	1000	normal	2m52.629s
		livre	6m55.552s
1000	10000	normal	56m43.129s
		livre	145m59.439s

Tabela 3: Testes - Aplicação Física 1

da solução é igual à 200.00 em ambos os casos. Um fato importante é notar que o caso de valor prescrito convergiu mais rapidamente (808 contra 1040 iterações do SOR).

4.4 Aplicação Física 2 - Escoamento em Águas Subterrâneas

Infelizmente não foi possível realizar este experimento.

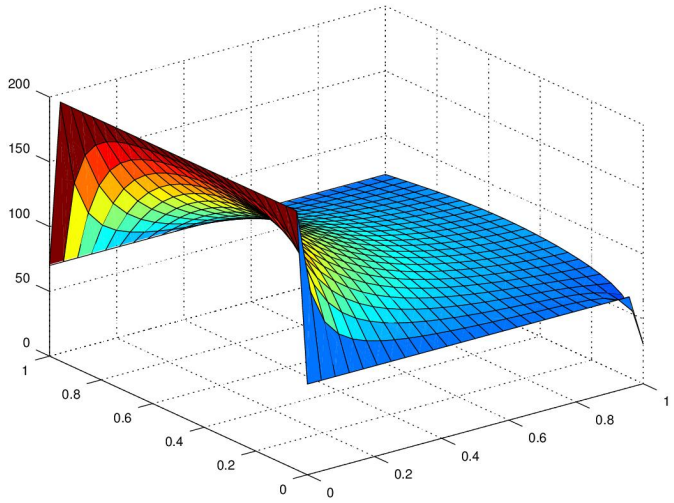

Figura 4: Gráfico solução da Aplicação Física 1 para $n = 25$ e $m = 25$

5 Conclusão

Com os dados coletados dos experimentos realizados, é passível assertar que a forma de armazenamento das estruturas resultantes pela discretização da equação (1) por diferenças finitas tem um impacto muito significativo no tempo de processamento.

Notamos que ao utilizar uma estrutura específica de armazenamento da matriz penta-diagonal com os coeficientes computados de acordo, e igualmente para o vetor independente, temos um ganho em eficiência significativo se avaliado em relação a um programa livre de matriz. Isto é intuitivo, uma vez que o algoritmo SOR livre de matriz deverá realizar exaustivamente operações iguais em cada iteração, enquanto no outro caso tais cálculos são efetuados apenas uma vez e armazenados em uma estrutura apropriada. E desta estrutura, o acesso é direto, o que é um ganho considerável à realizar operações a cada passo.

Porém vale destacar que o algoritmo "perdedor" em tempo de processamento, o SOR livre de matriz, não é descartável. Este é uma opção para casos onde a quantidade de dados a serem processados é de proporções muito grandes. Nestes casos, o método rápido não pode ser aplicado pois necessita de espaço em memória para armazenar seus dados, restando assim a aplicação de um algoritmo livre de matriz eficiente para a solução do problema.

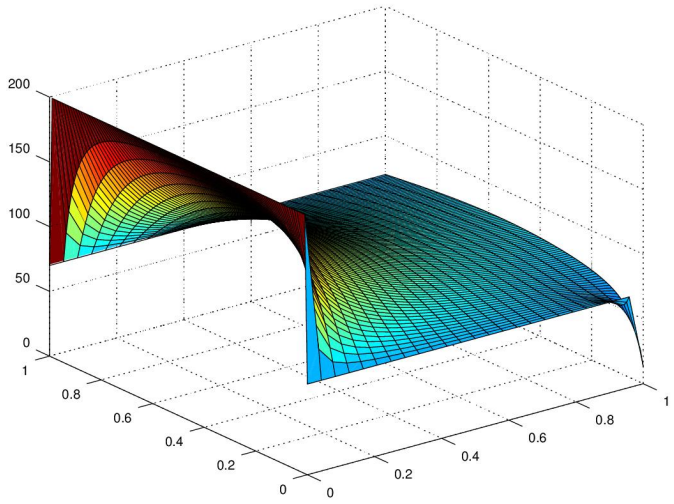

Figura 5: Gráfico solução da Aplicação Física 1 para $n = 25$ e $m = 100$

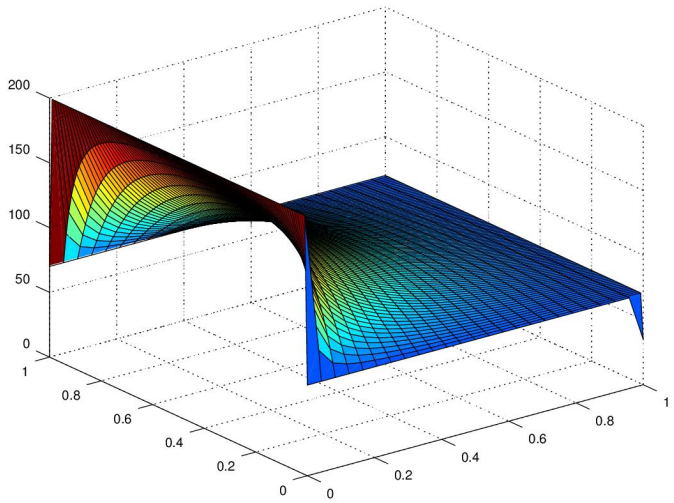

Figura 6: Gráfico solução da Aplicação Física 1 para $n = 25$ e $m = 100$ e $u(L, y) = 70$