

# Apresentação - A partir da Página 19

## Equação de Helmholtz - Métodos Numéricos

## MÉTODOS E IMPLEMENTAÇÃO COMPUTACIONAL

### Discretização MDFC2

- O parâmetro  $N$  representa o número de subintervalos em cada direção do domínio  $[0, 1]^2$ . Testamos valores de  $N = 64, 128, 192, 256$ . O tamanho do passo é dado por  $h = 1/N$ , e temos  $(N + 1) \times (N + 1)$  pontos totais na malha, sendo  $(N - 1)^2$  pontos internos que constituem as incógnitas do sistema. Por exemplo, para  $N = 256$ , temos  $h = 1/256 = 0.00390625$  e  $255 \times 255 = 65.025$  incógnitas.
- A discretização utiliza um stencil de 5 pontos, onde cada ponto interno está conectado ao seu centro e aos quatro vizinhos (norte, sul, leste, oeste). A equação discretizada é:

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2} + k^2 u_{i,j} = 0$$

O primeiro termo discretiza  $\frac{\partial^2 u}{\partial x^2}$ , o segundo discretiza  $\frac{\partial^2 u}{\partial y^2}$ , e o terceiro é o termo reativo  $k^2 u$  avaliado no ponto  $(i, j)$ .

- Esta discretização resulta em um sistema linear esparsa da forma  $A\mathbf{u} = \mathbf{b}$ , onde a matriz  $A = L + k^2 I$  tem dimensão  $(N - 1)^2 \times (N - 1)^2$  e possui apenas 5 diagonais não-nulas devido à estrutura do stencil. O vetor  $\mathbf{b}$  é construído a partir das condições de contorno Dirichlet impostas na fronteira do domínio.

### Implementação com Matrizes Esparsas

- Para  $N = 200$ , teríamos aproximadamente 40.000 incógnitas, e uma matriz densa ocuparia cerca de 12 GB de RAM, tornando a solução inviável em computadores típicos. A solução é o uso de matrizes esparsas. Utilizamos dois formatos principais: o formato LIL (List of Lists) para a montagem eficiente da matriz, que permite inserção rápida de elementos durante a construção e é ideal para montar a matriz Laplaciana. Após a montagem, convertemos para o formato CSC (Compressed Sparse Column), que é otimizado para operações de álgebra linear e é usado pelos solvers do `scipy.sparse`. Esta conversão é automática e resulta em uma redução drástica de memória: de 12 GB para apenas algumas dezenas de MB.

### Sistema de Cache — Otimização Crítica

- Ao testar múltiplos grupos com o mesmo valor de  $N$ , reconstruir as matrizes Laplacianas e as coordenadas da malha para cada caso seria extremamente custoso. Por exemplo, para  $N = 128$  testando 6 grupos e 4 valores diferentes de  $k$ , teríamos 24 casos. Sem cache, isso significaria 24 construções da matriz Laplaciana, todas idênticas. Com cache, fazemos apenas 1 construção e reutilizamos 23 vezes.

- A implementação do cache utiliza dois dicionários globais. O primeiro, `_laplacian_cache`, armazena as matrizes Laplacianas, onde a chave é o valor de  $N$  e o valor é uma tupla  $(L, h)$  contendo a matriz Laplaciana  $L$  já no formato CSC otimizado e o passo  $h$ . O segundo dicionário, `_coords_cache`, armazena as coordenadas da malha, onde a chave é novamente  $N$  e o valor é a tupla  $(x, y, X, Y)$  com as coordenadas, evitando recálculos custosos de `np.linspace` e `np.meshgrid`.
- A lógica de funcionamento é a seguinte: cada função que constrói estruturas recebe o parâmetro  $N$  e uma flag `use_cache`. Primeiro, verifica se  $N$  já existe no dicionário de cache. Se existe, retorna imediatamente do cache com acesso  $O(1)$ , que é praticamente instantâneo. Se não existe, constrói a estrutura, armazena no cache para uso futuro e retorna. Além disso, implementamos um pré-aquecimento do cache: antes de processar qualquer caso, construímos todas as estruturas para todos os valores de  $N$  que serão testados, garantindo que a primeira chamada durante o processamento já utilize o cache.
- Os ganhos de performance são significativos: a construção de matrizes fica aproximadamente 90% mais rápida. Por exemplo, a construção da matriz  $L$  para  $N = 256$  cai de aproximadamente 0.012 segundos para 0.001 segundos após o cache estar aquecido. A mesma matriz  $L$  é reutilizada para todos os grupos com o mesmo  $N$ , independentemente do valor de  $k$  ou da configuração de ângulos. Esta eficiência é especialmente importante ao testar múltiplos grupos e valores de  $k$ .
- Em um cenário completo com 6 grupos, 4 valores de  $k$  e 4 valores de  $N$ , temos um total de 96 casos a processar. Sem cache, isso resultaria em 96 construções da matriz Laplaciana. Com cache, fazemos apenas 4 construções (uma para cada valor de  $N$ ) e 92 reutilizações, resultando em uma economia de aproximadamente 96% de redução no tempo de construção das matrizes.

## Solvers Implementados

- O solver SPLU (Sparse LU) é uma fatoração LU esparsa implementada pela função `scipy.sparse.linalg.splu`. Este é um método direto que fornece solução exata sem iterações. O processo consiste em três etapas: primeiro, fatora a matriz  $A$  em  $A = LU$  através de uma decomposição Lower-Upper; depois, resolve  $Ly = b$  usando substituição progressiva (forward substitution); finalmente, resolve  $Ux = y$  usando substituição regressiva (backward substitution). Este método é eficiente para sistemas menores, especificamente para  $N \leq 192$ , o que corresponde a aproximadamente 40.000 incógnitas. A complexidade é  $O(N^3)$  para a fatoração e  $O(N^2)$  para a resolução, com memória de  $O(N^2)$ . A vantagem é que fornece solução exata e determinística, mas a desvantagem é que o custo cresce rapidamente com  $N$ .
- O solver GMRES+ILU combina o método iterativo GMRES (Generalized Minimal Residual) com um pré-condicionador ILU (Incomplete LU Factorization). O GMRES, implementado por `scipy.sparse.linalg.gmres`, é um método iterativo que encontra a solução no espaço de Krylov e minimiza o resíduo  $\|b - Ax\|_2$  em cada iteração. Utilizamos parâmetros `restart=100` e `maxiter=1000`, com tolerâncias `rtol=10-8` e `atol=10-12`. O ILU, implementado por `scipy.sparse.linalg.spilu`, é um pré-condicionador que fatora  $A \approx LU$  de forma incompleta, sendo mais rápido que uma fatoração completa mas ainda eficaz em acelerar a convergência do GMRES. Utilizamos parâmetros `drop_tol=10-3` e `fill_factor=20`. Este método é eficiente para sistemas grandes, especificamente para  $N \geq 256$ , com complexidade  $O(N^2)$  por iteração. A vantagem é que é escalável para sistemas muito grandes, mas a desvantagem é que pode não convergir se o sistema estiver mal condicionado.
- A seleção automática implementada através do modo "auto" utiliza uma heurística que escolhe o solver baseado no tamanho do sistema: se temos  $n \leq 40.000$  incógnitas, usa SPLU; se  $n > 40.000$ , usa GMRES+ILU. Além disso, implementamos uma estratégia de fallback robusta:

se o ILU falha e  $n \leq 100.000$ , tenta usar SPLU como alternativa; se o ILU falha e  $n > 100.000$ , usa GMRES sem pré-condicionador como último recurso. Esta estratégia garante que sempre obtemos uma solução, mesmo em casos problemáticos.

## Paralelização

- A implementação de paralelização utiliza `multiprocessing.Pool`, que permite processar múltiplos casos simultaneamente em diferentes núcleos do processador. Esta otimização resulta em um ganho de 3 a 4 vezes no tempo total de processamento, sendo especialmente útil ao processar os 96 casos do cenário completo (6 grupos  $\times$  4 valores de  $k$   $\times$  4 valores de  $N$ ).

## RESULTADOS PRINCIPAIS

### Quantificação do Erro

- O erro relativo foi quantificado utilizando a norma  $L^2(\Omega)$  integral, definida como  $\|u\|_{L^2(\Omega)}^2 = \int_{\Omega} |u(x, y)|^2 dx dy$ . O erro relativo é então calculado como  $E_{L^2} = \frac{\|u_{\text{aprox}} - u_{\text{exata}}\|_{L^2(\Omega)}}{\|u_{\text{exata}}\|_{L^2(\Omega)}}$ . Aproximamos esta norma integral utilizando a regra do retângulo em malha uniforme:  $\|u\|_{L^2(\Omega)}^2 \approx h^2 \sum_{i,j} |u_{i,j}|^2 = h^2 \|\mathbf{u}\|_2^2$ . Portanto, temos  $\|u\|_{L^2(\Omega)} \approx h \|\mathbf{u}\|_2$ , e o erro calculado é  $E_{L^2} \approx \frac{h \|\mathbf{U}_{\text{aprox}} - \mathbf{U}_{\text{exata}}\|_2}{h \|\mathbf{U}_{\text{exata}}\|_2}$ . Esta aproximação tem ordem  $O(h^2)$ , sendo consistente com a ordem do método de diferenças finitas de segunda ordem.

### Resultados por Número de Onda $k$

- Para  $k = 1$  (baixa frequência), obtivemos excelente precisão com erros relativos da ordem de  $10^{-7}$  a  $10^{-8}$ . A taxa de convergência observada foi  $p \approx 2.0$ , confirmando a ordem teórica do método. Especificamente, para  $N = 64$  temos erro de  $6.4 \times 10^{-7}$ , para  $N = 128$  o erro cai para  $1.6 \times 10^{-7}$  (4 vezes menor), e para  $N = 256$  o erro é de  $4.1 \times 10^{-8}$  (15 vezes menor que o inicial). Este comportamento demonstra que o método funciona perfeitamente para valores pequenos de  $k$ .
- Para  $k = 20$  (frequência média), observamos uma degradação na precisão, com erro relativo de  $4.2 \times 10^{-2}$  para  $N = 128$ , o que corresponde a 4.2%. A taxa de convergência cai para  $p \approx 1.5 - 1.8$ , indicando o início do efeito de poluição numérica. Para  $N = 64$  temos erro de  $2.1 \times 10^{-1}$  (21%), para  $N = 128$  o erro é  $4.2 \times 10^{-2}$  (4.2%, 5 vezes menor), e para  $N = 256$  o erro cai para  $1.0 \times 10^{-2}$  (1.0%, 21 vezes menor que o inicial). Apesar da degradação, ainda obtemos precisão aceitável com malhas refinadas.
- Para  $k = 40$  (alta frequência), observamos degradação significativa, com erro relativo de  $2.8 \times 10^{-1}$  para  $N = 128$ , o que corresponde a 28%. A taxa de convergência cai para  $p \approx 1.2 - 1.5$ . Para obter precisão aceitável, é necessário utilizar  $N \geq 256$ .
- Para  $k = 100$  (muito alta frequência), observamos poluição numérica severa, com erro relativo superior a 1.0, podendo ultrapassar 300%. A taxa de convergência cai para  $p < 1$ , indicando degradação muito severa. Para  $N = 64$  temos erro de 2.5 (250%), para  $N = 128$  o erro é 1.5 (150%), e surpreendentemente, para  $N = 256$  o erro aumenta para 3.6 (360%), demonstrando que mesmo com refinamento, o erro permanece alto e pode até piorar devido à poluição numérica.

## Regra de Ouro — Pontos por Comprimento de Onda

- A regra de ouro é definida através de  $N_\lambda = \frac{2\pi N}{k}$ , que representa o número de pontos por comprimento de onda. A recomendação é ter  $N_\lambda 10 - 20$  pontos por comprimento de onda para obter precisão aceitável. Ilustrando com exemplos: para  $k = 1$  e  $N = 64$ , temos  $N_\lambda \approx 402$  pontos (excelente); para  $k = 20$  e  $N = 128$ , temos  $N_\lambda \approx 40$  pontos (bom); para  $k = 100$  e  $N = 256$ , temos  $N_\lambda \approx 16$  pontos (marginal). Para casos de alta frequência, recomendamos  $N_\lambda \geq 20 - 30$  pontos por comprimento de onda.

## Desempenho Computacional

- Os tempos de execução são os seguintes: para  $N = 64$  temos 3.969 incógnitas e tempo de solução de 0.008 segundos utilizando SPLU; para  $N = 128$  temos 16.129 incógnitas e tempo de 0.039 segundos (SPLU); para  $N = 192$  temos 36.481 incógnitas e tempo de 0.117 segundos (SPLU); para  $N = 256$  temos 65.025 incógnitas e tempo de 0.423 segundos utilizando GMRES+ILU, que se torna necessário para sistemas grandes. As otimizações implementadas resultam em ganhos significativos: o cache torna a construção aproximadamente 90% mais rápida, e a paralelização acelera o processamento em 3 a 4 vezes.

## ANÁLISE E CONCLUSÕES

- O fenômeno de poluição numérica ocorre quando, mesmo com refinamento da malha, o erro cresce com o aumento do número de onda  $k$ . A causa fundamental é ter poucos pontos por comprimento de onda ( $N_\lambda < 20$ ), o que leva a um erro de fase que se acumula e amplifica, fazendo com que a solução numérica "atrase" espacialmente em relação à solução exata. A consequência é que a taxa de convergência degrada para  $p < 2$ , não atingindo a ordem teórica do método.
- Concluimos que o MDPC2 é adequado para valores moderados do número de onda, especificamente para  $k \leq 20$ , onde obtemos boa precisão com malhas razoáveis e erros aceitáveis menores que 5% com  $N \geq 128$ . A taxa de convergência confirma a ordem teórica de 2 para valores pequenos de  $k$ , com  $p \approx 2.0$  para  $k = 1$ , validando tanto a implementação quanto a análise teórica.
- A eficiência da implementação é notável: o uso de matrizes esparsas resulta em redução drástica de memória, o sistema de cache torna a construção aproximadamente 90% mais rápida, a paralelização acelera o processamento em 3 a 4 vezes, e a seleção automática de solver garante robustez e eficiência para todos os casos. As perspectivas futuras incluem investigação de métodos de ordem superior (4ª ou 6ª ordem) que podem reduzir o efeito de poluição numérica, uso de malhas adaptativas que refinam localmente onde necessário, e desenvolvimento de pré-condicionadores especializados para a equação de Helmholtz.

## PONTOS-CHAVE PARA DESTACAR

- Excelente precisão para  $k = 1$ : erro menor que  $10^{-6}$ , demonstrando que o método funciona perfeitamente para baixas frequências.
- Boa precisão para  $k = 20$  com  $N \geq 128$ : erro menor que 5%, mostrando que o método é adequado para frequências médias com malhas razoáveis.
- Poluição numérica severa para  $k = 100$ : erro superior a 300%, ilustrando as limitações do método para altas frequências.

- Importância do refinamento: ao aumentar de  $N = 64$  para  $N = 256$ , reduzimos o erro em 21 vezes para  $k = 20$ , demonstrando a eficácia do refinamento de malha.
- Cache como otimização crítica: reduz o tempo de construção em aproximadamente 90%, sendo fundamental para viabilizar a análise de múltiplos casos.
- Implementação robusta: a seleção automática de solver garante solução para todos os casos, mesmo em situações problemáticas, através da estratégia de fallback.