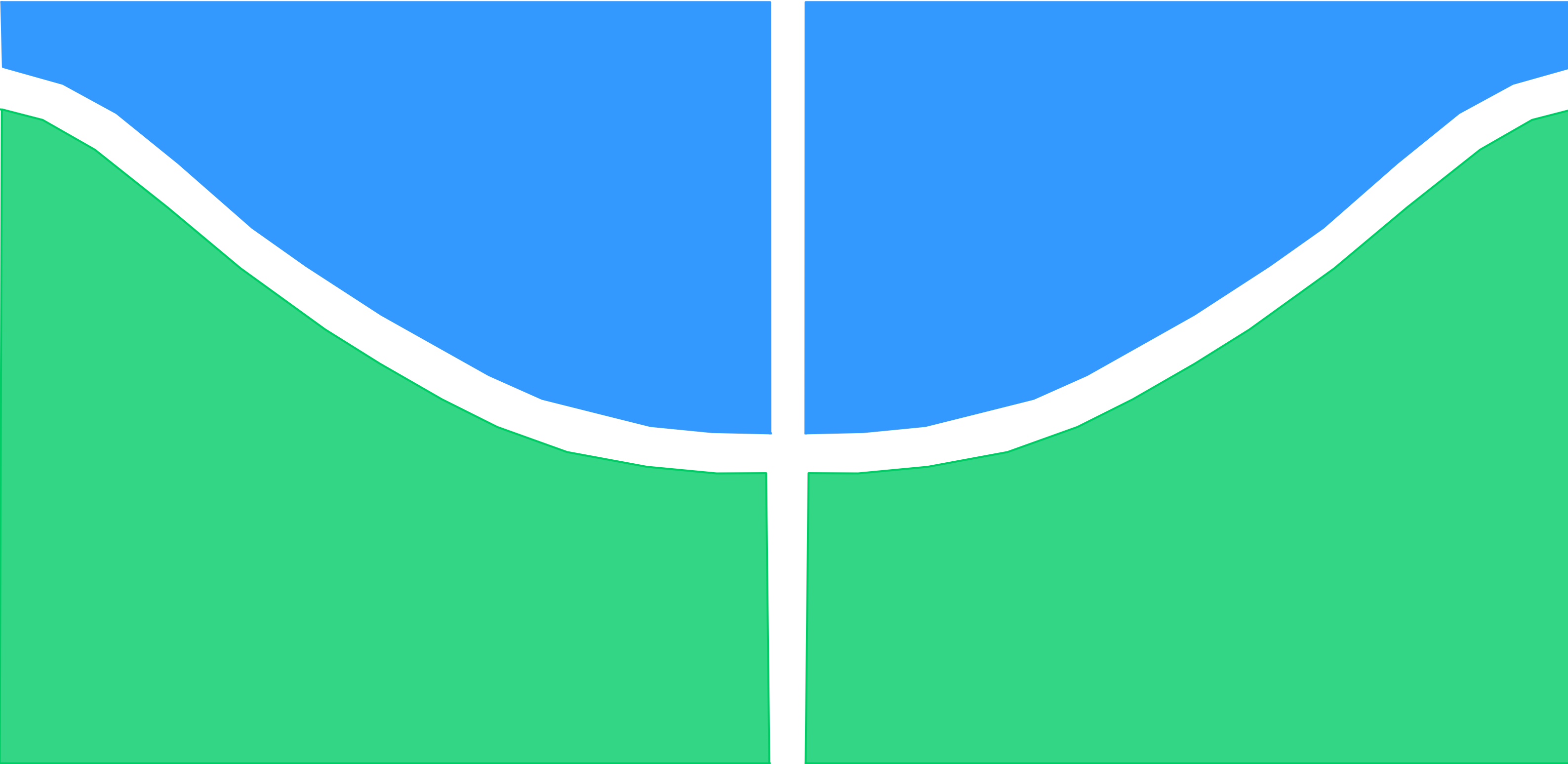


**PROJETO DE GRADUAÇÃO**

# IMPLEMENTAÇÃO DE DIFERENTES MÉTODOS ITERATIVOS PARA A RESOLUÇÃO DE SISTEMAS LINEARES COM APLICAÇÃO EM MECÂNICA DOS FLUIDOS

**Guilherme Mesquita Corrêa**

**Brasília, Novembro de 2021**

**UNIVERSIDADE DE BRASÍLIA**

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia

# PROJETO DE GRADUAÇÃO

# IMPLEMENTAÇÃO DE DIFERENTES MÉTODOS ITERATIVOS PARA A RESOLUÇÃO DE SISTEMAS LINEARES COM APLICAÇÃO EM MECÂNICA DOS FLUIDOS

**Guilherme Mesquita Corrêa**

*Relatório submetido ao Departamento de Engenharia Mecânica como requisito parcial para obtenção do grau de Bacharel em Engenharia Mecânica*

Banca Examinadora

Prof. Adriano P. Rosa, ENM/UnB ____________________
*Orientador*

Prof. Bráulio G. Pimenta, ENM/UnB ____________________
*Examinador*

Prof. Taygoara F. de Oliveira, ENM/UnB ____________________
*Examinador*

**Dedicatória**

*Aos meus tão amados e queridos pais, Auxiliadora Mesquita e Osvaldir Corrêa.*

*Guilherme Mesquita Corrêa*

## Agradecimentos

*Primeiramente, agradeço aos meus pais, a quem devo tudo que sou e que sei, que sempre me deram condições de me dedicar de maneira exclusiva aos estudos, que sempre me incentivaram e encorajaram a seguir em frente aprendendo cada vez mais, que me deram condições de ter aquilo que eles não tiveram e nunca me cobraram nada em troca de tamanha riqueza que me proveram, que é o meu conhecimento. Agradeço também aos meus irmãos, que sempre estiveram comigo e me ajudaram a descontrair nos momentos de lazer. Por fim, agradeço ao meu orientador por todo conhecimento que me passou, sempre de maneira muito didática, nas várias disciplinas que o tive como professor, com quem fiz, inclusive, a primeira disciplina da cadeira de Mecânica dos Fluidos na minha graduação, onde me encantei por essa área do conhecimento, que é tão bonita quanto complexa.*

*Guilherme Mesquita Corrêa*

## RESUMO

O presente trabalho estuda e compara diversos métodos iterativos de resolução de sistemas de equações lineares aplicados na solução do problema da cavidade na mecânica dos fluidos. Os métodos abordados vão desde métodos simples como Jacobi e Gauss-Seidel, até métodos mais avançados como o Gradiente Conjugado Pré-condicionado e o Multigrid. Os diversos métodos iterativos são aplicados para várias malhas com diferentes graus de refinamento e são comparados em termos de número de iterações necessárias para resolver o problema e tempos de execução dos códigos. As equações governantes do problema da cavidade são apresentadas, bem como a transformação das mesmas, que são originalmente equações diferenciais parciais, em um sistema de equações lineares através do método de projeção e do método de diferenças finitas. A implementação dos métodos iterativos é feita através do Python, uma linguagem de programação muitas vezes considerada lenta para esse propósito. Por ser lenta, uma otimização é implementada à linguagem para acelerar a velocidade de execução dos códigos através da biblioteca Numba. Uma comparação entre os tempos de execução obtidos no Python otimizado será feita com códigos equivalentes executados em Fortran, que é considerada uma linguagem rápida para esse tipo de aplicação. De uma forma geral, o método do Gradiente Conjugado Pré-condicionado se mostrou o mais rápido dentre todos os métodos testados nesse trabalho, seguido do Multigrid e do Gradiente Conjugado sem pré-condicionamento. Em relação à linguagem de programação utilizada, o Python, conseguiu-se acelerar consideravelmente seu tempo de execução dos códigos com a utilização da biblioteca Numba, tornando-o comparável com o Fortran, especialmente no sistema operacional Windows.

## ABSTRACT

In this work we study and compare several iterative methods for solving linear system of equations applied to the cavity problem of the fluid mechanics. The methods considered here goes from simple methods like the Jacobi and Gauss-Seidel method to more advanced methods like Preconditioned Conjugated Gradient and Multigrid method. The iterative methods are applied to several grids with diferent levels of refinement and are compared in terms of number of iterations necessary to solve the problem and execution time of the codes. The governing equations of the cavity problem are presented, as well as the transformation of the governing equations, which are partial diferential equations, into a system of linear equations through the projection method and the finite diferences method. The implementation of the iterative methods is done in Python, a programming language often considered slow for such purpose. For this reason, a optimization is implemented to the language in order to accelerate the execution time of the codes using the Numba library. A comparison will be done between execution times got in optimized Python codes and equivalent Fortran codes, which is a language considered fast for this kind of application. In a general way, the

Preconditioned Conjugate Gradient was the fastest method compared to the other ones, followed by Multigrid and Conjugate Gradient without preconditioning. When it comes to the programing language used, the Python, it was possible to considerably accelerate the execution time of the codes using the library Numba, making Python comparable to Fortran, specially on operational system Windows.

# SUMÁRIO


**1 Introdução** ..... **1**

1.1 Contextualização ..... 1

1.2 A relevância da Mecânica dos Fluidos ..... 1

1.3 A relevância dos métodos numéricos ..... 2

1.4 Python como linguagem de programação científica ..... 3

1.5 Objetivos do projeto ..... 4

1.6 Possíveis aplicações ..... 4

1.7 Apresentação do manuscrito ..... 5

**2 Fundamentação Teórica** ..... **6**

2.1 Introdução ..... 6

2.2 O problema da cavidade em mecânica dos fluidos ..... 6

2.2.1 Equações governantes ..... 7

2.2.2 Condições de contorno e condição inicial ..... 8

2.3 O método de projeção ..... 9

2.3.1 Discretização das equações governantes ..... 11

2.4 Sistema de equações do problema da cavidade ..... 14

2.5 Métodos de solução de sistemas de equações lineares ..... 17

2.6 Métodos diretos ..... 18

2.6.1 Eliminação Gaussiana ..... 18

2.6.2 Decomposição LU ..... 19

2.7 Métodos iterativos ..... 20

2.7.1 Critério de parada ..... 21

2.7.2 Método de Jacobi ..... 21

2.7.3 Método de Gauss-Seidel ..... 22

2.7.4 Método de Sobre-Relaxação Sucessiva (SOR) ..... 23

2.7.5 Método do Gradiente Conjugado ..... 24

2.7.6 Método do Multigrid ..... 31

2.7.7 Critérios de Convergência ..... 42

**3 Resultados** ..... **45**

3.1 Validação ..... 45

3.2 A importância de se implementar uma otimização para o Python ..... 46

3.3 FORTRAN vs Python+Numba ..... 47
3.4 A instabilidade do método de Jacobi ..... 48
3.4.1 Modificando sistema original ..... 48
3.4.2 Método de Jacobi Amortecido ..... 50
3.4.3 O motivo da instabilidade do método de Jacobi ..... 51
3.5 Métodos de Multigrid ..... 52
3.6 Comparações entre diversos métodos ..... 53
3.6.1 Resultados ..... 53
3.6.2 Análise final acerca dos resultados ..... 57

**4 Conclusões ..... 59**

**REFERÊNCIAS BIBLIOGRÁFICAS ..... 61**

**Anexos ..... 63**

**I Forma alternativa dos métodos estacionários ..... 64**

**II Códigos ..... 66**
II.1 Código do método Gauss-Seidel ..... 66
II.2 Código do método Jacobi Amortecido ..... 72
II.3 Código do método SOR ..... 79
II.4 Código do método Gradiente Conjugado ..... 86
II.5 Código do Método GCP-SSOR ..... 93
II.6 Código do método Multigrid (MG-GC-2M) ..... 101

# LISTA DE FIGURAS

2.1 O problema da cavidade . . . . . 7
2.2 Malha defasada 5x5. Triângulo: componente u. Quadrado: componente v. Círculo: pressão p. . . . . . 12
2.3 Malha defasada 3x3. Triângulo: componente u. Quadrado: componente v. Círculo: pressão p. . . . . . 15
2.4 Comparação entre vários modos de Fourier com diferentes números de onda k. . . . . . 32
2.5 Comportamento do erro em função do número de iterações no método de Jacobi Amortecido com $\omega = 2/3$ para vários números de onda k. . . . . . 33
2.6 Comportamento do erro em função do número de iterações para o método de Jacobi Amortecido com $\omega = 2/3$ para um chute inicial $\mathbf{x}^{(0)} = (\mathbf{x}_1^{(0)} + \mathbf{x}_6^{(0)} + \mathbf{x}_{32}^{(0)})/3$ . . . . . 34
2.7 Comparação entre o chute inicial e a aproximação após 10 iterações no método de Jacobi Amortecido com $\omega = 2/3$ para diferentes modos de Fourier. . . . . . 37
2.8 Modo de Fourier com número de onda k=4 representado em uma malha com 12 e 6 pontos, respectivamente. A malha mais grossa tem uma representação mais oscilatória do mesmo modo. . . . . . 38
2.9 Operação de interpolação da malha $\Omega^{2h}$ para malha $\Omega^h$. . . . . . 39
2.10 Operação de restrição da malha $\Omega^h$ para malha $\Omega^{2h}$. . . . . . 40
2.11 Diferentes possibilidades de transitar entre as malhas no método do Multigrid. . . . . . 41

3.1 Comparação dos resultados desse trabalho com os resultados de Marchi et al. [1] para validação. . . . . . 45
3.2 Linhas de corrente para Re=100. . . . . . 46
3.3 Comportamento do número de iterações para os 100 primeiros passos de tempo para os métodos de SOR e Gauss-Seidel. . . . . . 49
3.4 Comportamento do número de iterações para os 100 primeiros passos de tempo para o método de Jacobi. . . . . . 50
3.5 Gráfico para determinar o $\omega$ ótimo para o método de Jacobi Amortecido para N=100. 51
3.6 Comportamento do número de iterações para os 100 primeiros passos de tempo. . . . . . 51
3.7 Comparação de diversos métodos Multigrid e Singlegrid em termos de tempo de execução para $N = 512$, $Re = 3000$, $\Delta t = 0,0005$, $t_f = 0,05$ e $\epsilon = 10^{-6}$ . . . . . 53
3.8 Gráfico para determinar $\omega$ ótimo para o método de SOR para $N = 128$. . . . . . 55
3.9 Gráfico para determinar $\omega$ ótimo para o método de GCP-SSOR para $N = 128$. . . . . . 56
3.10 Comparação entre os tempos de execução dos diversos métodos para N=1024, Re=3000, $t_f = 0,05$, $\epsilon = 10^{-6}$ e $= 0,0005$. . . . . . 56

3.11 Comparação entre os tempos de execução dos diversos métodos para N=1024, Re=3000, $t_f$=0,05, $\epsilon$=$10^{-6}$ e $\Delta$t=0,0005. .......... 57
3.12 Comparação de tempo de execução entre os métodos de SOR, GC e GCP-SSOR para malhas de diversos tamanhos. .......... 58

# LISTA DE TABELAS

3.1 Resultados para N=25, Re=100, $t_f$=0,1, $\epsilon$=$10^{-6}$ e $\Delta$ t=0,001. .............................. 46
3.2 Resultados para N=100, Re=1000, $t_f$=1.0, $\epsilon$=$10^{-7}$ e $\Delta$ t=0,001. .......................... 47
3.3 Resultados para N=128, Re=3000, $t_f$=0,05, $\epsilon$=$10^{-6}$ e $\Delta$t=0,0005. ........................ 54

# LISTA DE SÍMBOLOS

### Símbolos Gregos

| | | |
|---|---|---|
| $\Delta$ | Variação entre duas grandezas similares | |
| $\epsilon$ | Tolerância | |
| $\rho$ | Densidade | $[\mathrm{m}^3/\mathrm{kg}]$ |
| $\mu$ | Viscosidade dinâmica | $[\mathrm{N.s/m}s^2]$ |
| $\omega$ | Fator de relaxação | |
| $\Omega$ | Domínio da cavidade | |
| $\kappa$ | Número de condição | |

### Símbolos Romanos

| | |
|---|---|
| P | Pressão adimensional |
| t | Tempo adimensional |
| U | Velocidade da tampa da cavidade |
| L | Comprimento adimensional das paredes da cavidade |
| $\mathbf{u}$ | Vetor velocidade adimensional |
| u | Componente horizontal da velocidade adimensional |
| v | Componente vertical da velocidade adimensional |
| N | Número de intervalos da malha em uma direção |
| $\hat{\mathbf{T}}$ | Vetor tangente à parede da cavidade |
| $\hat{\mathbf{n}}$ | Vetor normal à parede da cavidade |

### Grupos Adimensionais

| | |
|---|---|
| $Re$ | Número de Reynolds |

**Siglas**

| | |
|---|---|
| SOR | Sobre Relaxação Sucessiva |
| GC | Gradiente Conjugado |
| GCP | Gradiente Conjugado Pré-condicionado |
| GCP-D | Gradiente Conjugado Pré-condicionado pela matriz Diagonal |
| GCP-SSOR | Gradiente Conjugado Pré-condicionado pela matriz $M_{SSOR}$ |
| MG-SOR-2M | Multigrid com SOR em esquema com 2 malhas |
| MG-SOR-3M | Multigrid com SOR em esquema com 3 malhas |
| MG-GC-2M | Multigrid com Gradiente Conjugado em esquema com 2 malhas |
| MG-GC-3M | Multigrid com Gradiente Conjugado em esquema com 3 malhas |
| FMG-GC | Full-Multigrid com Gradiente Conjugado em esquema com 2 malhas |
| MG-GCP-2M | Multigrid com Gradiente Conjugado em esquema com 3 malhas |

# Capítulo 1

# Introdução

## 1.1 Contextualização

Em diversas áreas da ciência (pra não dizer quase todas), seja na física, na química, na biologia, na economia ou estatística, surgem sistemas de equações lineares a serem resolvidos. As diversas formas de solução desses sistemas já são conhecidas há muito tempo, seja de forma direta ou de forma iterativa. O fato dessas formas de resolução já serem conhecidas há muito tempo nos faz pensar (de maneira equivocada) que talvez esse seja um tópico de estudo ultrapassado [2, 3]. A necessidade de se estudar a solução de sistemas de equações lineares ainda nos dias atuais surge do fato de que os sistemas lineares em aplicações reais são muito grandes, com milhares, frequentemente milhões, de equações envolvidas, o que torna extremamente demorado de se resolver mesmo com os computadores mais rápidos existentes na atualidade. Dentro desse contexto, faz-se necessário a utilização de formas cada vez mais rápidas de solução de equações lineares, seja por meio de novos métodos de resolução ou por otimização dos métodos já conhecidos. Os métodos diretos de resolução de sistemas de equações lineares, apesar de nos fornecer uma solução exata, apresentam um custo computacional bastante elevado quando comparado com os métodos iterativos para sistemas esparsos, fazendo com que seja muito demorado implementar os métodos diretos na resolução de sistemas muito grandes.

O presente trabalho tem como objetivo comparar diferentes métodos iterativos conhecidos atualmente, a saber, o método de Jacobi, método de Gauss-Seidel, método do Gradiente Conjugado e o método do Multigrid, bem como tentar otimizá-los, dentro da linguagem Python, ao máximo de forma a reduzir o tempo computacional despendido na aplicação. Essa comparação será feita através da resolução do problema da cavidade na mecânica dos fluidos.

## 1.2 A relevância da Mecânica dos Fluidos

O estudo de escoamentos de fluidos é uma área de importância crítica em inúmeras aplicações de engenharia. Prever o comportamento de um fluido submetido a determinadas condições de escoamento é importante porque nos permite estimar forças que o fluido exerce sobre uma estrutura

sólida (como a força que o vento exerce sobre um prédio ou que a água exerce sobre o casco de um navio, por exemplo), ou prever zonas de alta ou baixa pressão, zonas de recirculação, entre outras coisas. Ao levantar essas informações, é possível tomar decisões de projeto mais assertivas e simultaneamente compreender melhor os fenômenos físicos envolvidos. Resolver problemas de escoamentos é tão complexo quanto importante. O estudo desses fenômenos envolve equações diferenciais parciais que, muitas vezes, sequer possuem solução analítica. Nesses casos, se torna necessário utilizar os chamados métodos numéricos para obtenção da solução desses escoamentos. Os métodos numéricos possuem fundamental importância no estudo de diversas equações com as quais os engenheiros precisam lidar pois permitem encontrar solução para equações que não podem ser resolvidas analiticamente. Nesse contexto, muitos estudos vêm sendo desenvolvidos nas últimas décadas na área de simulação de escoamentos através de métodos numéricos, com aplicações na engenharia [4, 5], na ciência e tecnologia de uma forma geral [6], mas também na indústria do entretenimento, como na produção de animações e de jogos eletrônicos mais realistas, como os trabalhos de Kui Wu et al. [7], Jos Stam [8] e Ives [9], por exemplo.

## 1.3 A relevância dos métodos numéricos

O estudo de métodos numéricos tem fundamental importância no desenvolvimento da ciência e da tecnologia, especialmente na engenharia. Os métodos numéricos permitem que importantes equações físicas, que descrevem fenômenos vistos na natureza, sejam solucionadas, ainda que não se conheça uma solução analítica para tais equações, como já foi dito anteriormente. Como exemplos desses fenômenos, podemos citar a transferência de calor, a propagação de ondas mecânicas e eletromagnéticas, o comportamento de fluidos, o crescimento de populações de seres vivos em geral, a contaminação de uma determinada população por uma doença, além de fenômenos em áreas das ciências humanas como na economia [10, 11, 12], por exemplo.

Uma classe especial de equações que podem ser solucionadas numericamente são as chamadas equações diferenciais parciais, ou EDPs, para os mais íntimos. Existem diversos métodos numéricos para resolução de EDPs. Entre esses métodos se destacam o método das diferenças finitas, o método de elementos finitos e o método dos volumes finitos [13]. Os métodos numéricos consistem, de maneira geral, em transformar uma equação diferencial parcial em várias equações algébricas, que são, normalmente, muito mais fáceis de resolver do que as equações diferenciais. Um inconveniente dos métodos numéricos é o fato de que se gera, inevitavelmente, um número muito grande de equações algébricas para resolver (às vezes na ordem de milhões de equações), de maneira que torna o método numérico impossível de ser feito à mão. Nesse contexto, o avanço da informática, especialmente dos processadores, tem fundamental importância para a implementação dos métodos numéricos, uma vez que um computador é capaz de realizar cálculos milhões de vezes mais rápido do que nós humanos. Através de algoritmos, um computador pode executar uma sequência de comandos imposta pelo programador que leve à solução numérica de uma EDP em poucos minutos, às vezes até em milésimos de segundo.

Além dos métodos numéricos aplicados à resolução de EDPs, uma classe especial de métodos

numéricos são os voltados para resolver sistemas de equações lineares. Ao se discretizar a EDP em várias equações algébricas através de uma discretização numérica, recaímos sobre um sistema de equações lineares que podem, por sua vez, serem resolvidas de maneira direta ou iterativa. Os métodos diretos de resolução de sistemas de equações lineares, apesar de fornecerem a solução exata do sistema, são bastante custosos do ponto de vista computacional quando aplicados a sistemas muito grandes. Já os métodos iterativos, apesar de fornecerem uma solução aproximada, quando aplicados da maneira correta, são muito mais rápidos do que os métodos diretos para obter uma solução para sistemas muito grandes.

Nesse sentido, o objetivo no presente trabalho é estudar diversos métodos iterativos de resolução de sistemas de equações lineares do ponto de vista velocidade de resolução de problemas e também de tempo de execução dos códigos em Python de cada método. O estudo de vários métodos diferentes é importante para que se conheça as vantagens e desvantagens de cada um e, assim, se possa aplicar de maneira mais consciente e otimizada um método numérico na resolução de um problema complexo específico.

## 1.4 Python como linguagem de programação científica

A computação moderna permite em diversos aspectos a expansão das fronteiras do conhecimento humano, em especial através da análise de grandes conjuntos de dados e na resolução de grandes sistemas de equações. Isso é possível porque, diferente dos seres humanos, os computadores têm a capacidade de realizar operações matemáticas de maneira extremamente rápida, na ordem de milhões de operações em segundos. Para fazer tais operações, é necessária a utilização de uma linguagem de programação para implementar os algoritmos que detalham o passo a passo de como as operações devem ser feitas. A escolha da linguagem de programação é fundamental, já que as diferentes opções de linguagem oferecem diferentes características em termos de curva de aprendizado, de velocidade de execução e várias outras particularidades.

Nesse contexto, o Python se destaca por ser uma linguagem com uma curva de aprendizado bastante acelerada, sendo possível aprender em pouco tempo, além de ser bastante popular, o que torna bastante vasto o material disponível sobre a linguagem, seja na internet ou em livros. Apesar de ser fácil de aprender, o Python possui um problema que o torna quase impeditivo para resolver problemas com um número exageradamente grande de operações, que é o tempo de execução dos seus códigos, em especial quando se trata de laços (ou ciclos) dentro de laços. Outras linguagens, como o Fortran, C e C++ são opções bastante mais rápidas do que o Python, entretanto são mais complicadas e com curvas de aprendizado mais lentas. Uma forma de contornar o problema da lentidão do Python em executar algoritmos com muitas operações é a implementação do Numba, uma biblioteca de Python desenvolvida com a contribuição de grandes empresas de tecnologia como a AMD, Intel, NVidia e várias outras. O Numba é capaz de aumentar dramaticamente a velocidade de execução de códigos no Python, tornando a linguagem comparável com Fortran, C e C++ em termos de velocidade de execução e de maneira muito fácil. Quem disse que não existe almoço grátis?!

## 1.5 Objetivos do projeto

Esse projeto tem como principal objetivo a comparação entre os diversos métodos iterativos de resolução de sistemas de equações lineares dentro do contexto de mecânica dos fluidos, mais especificamente no problema da cavidade, que é um problema clássico bastante explorado. A comparação será feita em termos de tempo de execução de códigos e número de iterações para convergência. Os métodos iterativos explorados e comparados diretamente serão:

- Método de Jacobi;
- Método de Gauss-Seidel;
- Método de SOR;
- Método do Gradiente Conjugado;
- Método do Multigrid.

Para a implementação desses métodos, será utilizado o Python como linguagem de programação. Além disso, será utilizado a biblioteca Numba, dentro do Python, para otimização do código.

## 1.6 Possíveis aplicações

Além de criar um material de consulta e de estudo para o problema da cavidade e para os diversos métodos numéricos de solução de sistemas de equações lineares, incluindo métodos refinados como Gradiente Conjugado Pré-condicionado e o Multigrid, os resultados desse trabalho podem ajudar a expandir a aplicação do método das diferenças finitas para domínios com malhas cada vez mais refinadas. Isso será possível porque o presente trabalho busca maneiras mais rápidas de resolver sistemas de equações lineares, ou pelo menos mais rápidas do que velocidades alcançadas normalmente com os métodos iterativos mais tradicionais e comumente ensinados em disciplinas básicas de métodos numéricos em cursos de graduação em engenharia, como o método de Jacobi e o método de Gauss-Seidel por exemplo. Atualmente, é muito difícil, por exemplo, simular uma cavidade tridimensional pelo método das diferenças finitas, ou qualquer outro método de discretização de domínios, para malhas muito refinadas pois o sistema de equações lineares gerado nesse tipo de problema é demasiadamente grande e cresce com $N^3$, o que é um problema enorme em termos de tempo computacional, mesmo para computadores modernos e com códigos bem otimizados. Esse foi um problema encontrado, por exemplo, no Projeto de Graduação de Abicalil [14]. Além do estudo de métodos que convirjam mais rapidamente, o presente trabalho se propõe a fazer a implementação desses métodos mais eficientes usando o Python como linguagem de programação, o que é muito útil uma vez que linguagens mais rápidas, como Fortran e C++, por serem mais difíceis de se aprender, costumam atrasar ou até impedir que novos alunos ataquem o problema por encontrarem uma barreira na programação. Usando o Python de forma otimizada, que é

muito mais fácil de aprender e está bem difundido tanto no meio acadêmico quanto na indústria, a programação se tornaria uma barreira menor e os alunos vão poder focar naquilo que realmente importa, que é resolver problemas.

## 1.7 Apresentação do manuscrito

No capítulo 2 é feita uma discussão aprofundada sobre o problema da cavidade, sobre como partir das EDPs governantes e chegar no sistema de equações lineares a ser resolvido. Ainda no capítulo 2, é feita uma revisão sobre os diversos métodos diretos e iterativos de resolução de sistemas de equações lineares. A implementação e a comparação entre esses métodos são desenvolvidas no capítulo 3. No capítulo 4 são apresentadas as conclusões finais do trabalho. Os anexos contêm os códigos utilizados para implementar cada um dos métodos iterativos de resolução de sistemas de equações lineares tratados nesse trabalho.

# Capítulo 2

# Fundamentação Teórica

## 2.1 Introdução

O presente capítulo tem como objetivo apresentar o problema da cavidade na mecânica dos fluidos, descrever as equações governantes e apresentar discretização dessas equações por meio do método de projeção e da técnica de diferenças finitas, o que nos levará a um sistema de equações lineares. Em seguida, então, serão revistos os principais métodos diretos e iterativos de resolução de sistemas lineares. Dessa forma, todo o arcabouço teórico necessário para o problema o qual esse trabalho se propõe a resolver pode ser encontrado nesse capítulo.

## 2.2 O problema da cavidade em mecânica dos fluidos

O escoamento a ser resolvido é o de um fluido confinado em uma cavidade quadrada bidimensional com uma velocidade U constante na parede superior da cavidade, como mostra a figura 2.1. Esse é um problema clássico da mecânica dos fluidos computacional [15] e é comumente utilizado para validar metodologias numéricas. O interior da cavidade é o domínio $\Omega$ na qual as equações governantes devem ser resolvidas e as paredes da cavidade compõem o bordo $\partial\Omega$ do domínio e, nesse bordo, a velocidade é conhecida para qualquer instante $t>0$ através das chamadas condições de contorno do problema.

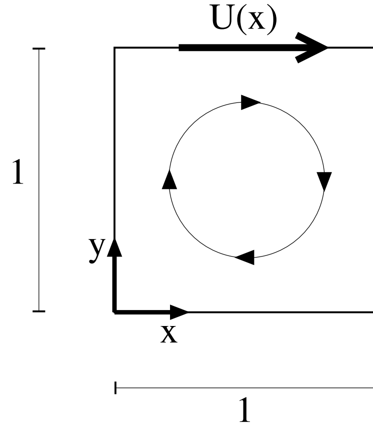



Figura 2.1: O problema da cavidade

### 2.2.1 Equações governantes

O escoamento tratado nesse trabalho é admitido como incompressível e newtoniano. Para essas condições, segundo White [16], as equações governantes, na forma adimensional, são dadas por

$$\nabla \cdot \mathbf{u} = 0 \tag{2.1}$$

e

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + \frac{1}{Re}\nabla^2 \mathbf{u}, \tag{2.2}$$

onde

$$Re = \frac{\rho U_0 L}{\mu} \tag{2.3}$$

é o número de Reynolds. A equação 2.1, chamada de equação da continuidade, impõe a conservação de massa ao sistema considerando o escoamento como incompressível. A equação 2.2, chamada de equação de Navier-Stokes, é uma equação que impõe o balanço da quantidade de movimento ao sistema e nada mais é do que a aplicação da segunda lei de Newton ($\mathbf{F} = m\mathbf{a}$) ao sistema. Na equação 2.2, o segundo termo do lado esquerdo da equação ($\mathbf{u} \cdot \nabla \mathbf{u}$) é um termo de convecção e é responsável pelo transporte de informação no escoamento através do próprio escoamento, ou seja, trata-se das partículas de fluido carregando consigo informação enquanto viajam pelo espaço. Já o segundo termo do lado direito ($\frac{1}{Re}\nabla^2 \mathbf{u}$) é um termo de difusão, ou seja, ele é responsável por difundir informação pelo escoamento sem ser através do movimento das partículas. Uma forma de exemplificar a difusão é através do calor, que se propaga em uma estrutura mesmo que cada partícula da estrutura esteja parada. Já na convecção, o movimento do fluido é quem carrega a informação, assim como um rio carrega as folhas das árvores. O termo $\frac{\partial \mathbf{u}}{\partial t}$ diz respeito à variação da velocidade $\mathbf{u}$ no tempo e o termo e $\nabla p$ é o gradiente de pressão, que quantifica o quanto a pressão varia ao longo do espaço. O parâmetro Re (equação 2.3) nos dá informação sobre o regime de escoamento do problema. Um escoamento com número de Reynolds alto significa que

o escoamento sofre pouca influência da viscosidade do fluido e pode ser tratado como invíscido quando longe das paredes, já que perto das mesmas existe a influência da camada limite, onde os efeitos da viscosidade nunca podem ser desprezados. Quando Reynolds tem um valor baixo, significa que o escoamento é afetado majoritariamente pela viscosidade do fluido e, portanto, o escoamento não pode ser tratado como invíscido.

Como o escoamento considerado nesse trabalho é bidimensional, temos que

$$\mathbf{u} = u(x,y,t)\hat{\mathbf{e}}_{\mathbf{x}} + v(x,y,t)\hat{\mathbf{e}}_{\mathbf{y}}. \tag{2.4}$$

A pressão é dada por

$$p = p(x,y,t). \tag{2.5}$$

Particularizando as equações 2.1 e 2.2 para duas dimensões, podemos reescrevê-las como

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{2.6}$$

e

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{\partial p}{\partial x} + \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right), \tag{2.7}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -\frac{\partial p}{\partial y} + \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right). \tag{2.8}$$

As equações 2.6, 2.7 e 2.8 então são as equações governantes do problema.

### 2.2.2 Condições de contorno e condição inicial

As condições de contorno para o problema são

- Condição de não deslizamento em todas as paredes: $\mathbf{u}(x,y,t) \cdot \hat{\mathbf{T}} = 0 \ \forall (x,y) \in \partial\Omega$, onde $\hat{\mathbf{T}}$ é o vetor tangente à superfície $\partial\Omega$ e $\Omega$ é o domínio do escoamento;
- Condição de impenetrabilidade em todas as paredes: $\mathbf{u}(x,y,t) \cdot \hat{\mathbf{n}} = 0 \ \forall (x,y) \in \partial\Omega$, onde $\hat{\mathbf{n}}$ é o vetor normal à superfície $\partial\Omega$ e $\Omega$ é o domínio do escoamento.

Expressando as condições de contorno de maneira detalhada para cada uma das componentes u e v de velocidade para o caso da cavidade bidimensional, temos

$$u(0,y,t) = 0,\ 0 < y < 1, t > 0, \tag{2.9}$$

$$u(1,y,t) = 0,\ 0 < y < 1,\ t > 0, \tag{2.10}$$

$$u(x,0,t) = 0,\ 0 < x < 1,\ t > 0, \tag{2.11}$$

$$u(x,1,t) = 1,\ 0 < x < 1,\ t > 0, \tag{2.12}$$

$$v(0,y,t) = 0,\ 0 < y < 1, t > 0, \tag{2.13}$$

$$v(1,y,t) = 0,\ 0 < y < 1, t > 0, \tag{2.14}$$

$$v(x,0,t) = 0,\ 0 < x < 1, t > 0, \tag{2.15}$$

e

$$v(x,1,t) = 0,\ 0 < x < 1, t > 0. \tag{2.16}$$

Já a condição inicial é dada por

- Condição de velocidade inicial nula no interior do domínio: $\mathbf{u}(x,y,0) = 0\ \forall (x,y) \in \Omega$.

Ou seja, de maneira individual para cada uma das componentes, temos que

$$u(x,y,0) = 0,\ 0 < x < 1,\ 0 < y < 1. \tag{2.17}$$

e

$$v(x,y,0) = 0,\ 0 < x < 1,\ 0 < y < 1. \tag{2.18}$$

## 2.3 O método de projeção

Na solução da equação 2.2, o termo de pressão e de velocidade estão acoplados entre si e isso dificulta encontrar a solução da equação [17]. Para contornar esse problema, vamos apelar para o método de projeção, que basicamente desacopla os termos de pressão e velocidade da equação 2.2 [15, 18]. Dessa forma, vamos primeiro calcular uma velocidade intermediária $\mathbf{u}^*$, em seguida a pressão e o resultados usaremos para calcular a velocidade. Para aplicar o método de projeção, teremos que fazer uma discretização da derivada temporal da equação 2.2. Discretizando então a derivada temporal usando uma aproximação de primeira ordem de Euler da equação da equação 2.2, temos que

$$\frac{\mathbf{u}^{k+1} - \mathbf{u}^k}{\Delta t} + \mathbf{u}^k \cdot \nabla \mathbf{u}^k = -\nabla p^{k+1} + \frac{1}{Re}\nabla^2 \mathbf{u}^k, \tag{2.19}$$

onde o índice k indica o passo de tempo correspondente. Como o campo de velocidade $\mathbf{u}$ é conhecido no instante inicial t=0, podemos isolar o termo $\mathbf{u}^{k+1}$ para calculá-lo. O problema é que o termo de pressão $\nabla p^{k+1}$ não é conhecido a priori e seria necessário encontrar um termo de pressão que satisfaça simultaneamente as equações da continuidade e de Navier-Stokes (Equações 2.1 e 2.2). Para fazer o desacoplamento de $\mathbf{u}^{k+1}$ e $p^{k+1}$, reescreveremos a equação 2.19 da seguinte maneira:

$$\frac{\mathbf{u}^{k+1} + \Delta t \nabla p^{k+1} - \mathbf{u}^k}{\Delta t} = -\mathbf{u}^k \cdot \nabla \mathbf{u}^k + \frac{1}{Re} \nabla^2 \mathbf{u}^k. \tag{2.20}$$

Agora, definindo uma nova variável $\mathbf{u}^*$ como sendo

$$\mathbf{u}^* = \mathbf{u}^{k+1} + \Delta t \nabla p^{k+1} \tag{2.21}$$

e substituindo na equação 2.20, temos

$$\frac{\mathbf{u}^* - \mathbf{u}^k}{\Delta t} = -\mathbf{u}^k \cdot \nabla \mathbf{u}^k + \frac{1}{Re} \nabla^2 \mathbf{u}^k. \tag{2.22}$$

Agora é possível isolar $\mathbf{u}^*$ na equação 2.22 e determinar o valor dessa variável. O próximo passo é determinar o valor de $p^{k+1}$. Para isso, aplica-se o operador divergente em ambos os lados da equação 2.21, o que nos leva à equação

$$\nabla \cdot \mathbf{u}^* = \nabla \cdot \mathbf{u}^{k+1} + \nabla \cdot \Delta t \nabla p^{k+1}. \tag{2.23}$$

Como se trata de escoamento incompressível, devemos garantir que o termo $\nabla \cdot \mathbf{u}^{k+1}$ seja identicamente nulo, de acordo com a equação 2.1. Portanto, podemos escrever

$$\nabla^2 p^{k+1} = \frac{1}{\Delta t} \nabla \cdot \mathbf{u}^*. \tag{2.24}$$

A equação 2.24 é o grande foco desse trabalho. Na seção 2.3.1, essa equação será discretizada através do método das diferenças finitas e será transformada em um sistema de equações lineares e é exatamente esse sistema o foco do atual trabalho. O sistema de equações lineares gerado pela discretização da equação 2.24 será resolvido através de diversos métodos iterativos diferentes no capítulo 3 e, com a solução do sistema, ou seja, tendo o valor de pressão para cada ponto do domínio, podemos seguir para o próximo passo, que é calcular o termo $\mathbf{u}^{k+1}$ através da equação 2.21.

Dessa forma, temos que o algoritmo pra calcular a velocidade para cada passo de tempo é da seguinte forma:

- Calcular $\mathbf{u}^*$ usando a equação 2.22;
- Calcular $p^{k+1}$ usando a equação 2.24;
- Calcular $\mathbf{u}^{k+1}$ usando a equação 2.21.

Todas as equações descritas nesse algoritmo são equações diferenciais parciais e, para resolvelas, elas serão aproximadas por diferenças finitas, o que nos levará a um sistema de equações lineares, como já foi dito anteriormente. A discretização das equações 2.21, 2.22 e 2.24 será feita termo a termo na seção 2.3.1. Uma vez que a pressão é desacoplada da velocidade pelo método de projeção, é necessário definir uma condição de contorno para a pressão. Será usada a condição

de contorno que define um gradiente normal nulo para a pressão nas paredes da cavidade. Dessa forma, temos que

$$\nabla p^{k+1} \cdot \hat{\mathbf{n}} = 0 \quad \forall (x, y) \in \partial\Omega. \tag{2.25}$$

Portanto, a equação 2.25, que define uma condição de parede rígida, será a condição de contorno para a equação 2.24. A condição de contorno adotada para a pressão nos provê um erro $O(\Delta x)$ nos valores de pressão próximos das paredes, porém não afeta a pressão no interior da cavidade [17]. Essa condição de contorno foi escolhida por apresentar uma implementação mais simples e por gerar a mesma matriz de coeficientes do sistema de equações, que será apresentado na seção 2.4, que condições de contorno mais sofisticadas. É importante destacar que maneiras mais sofisticadas de definir a condição de contorno da pressão nos levam a sistemas parecidos a serem resolvidos [19, 20].

### 2.3.1 Discretização das equações governantes

Para encontrar solução numérica do problema da cavidade, é necessário discretizar o domínio $\Omega$ no qual as equações serão resolvidas. É necessário discretizar não somente o domínio espacial $\Omega$, mas também o domínio temporal. A discretização do domínio espacial e temporal nos permite escrever as equações 2.21, 2.22 e 2.24 como equações algébricas ao invés de equações diferenciais parciais. Essa discretização, na verdade, é a ideia principal do método das diferenças finitas para resolução de EDPs. Para fazer a discretização espacial das equações, será utilizada uma malha defasada, ou seja, os locais de cálculo de pressão, velocidade $u$ e velocidade $v$ não serão nos mesmos pontos. Essa estratégia é vantajosa pois produz soluções mais acuradas do que os resultados obtidos em uma malha não defasada, que gera oscilações na solução que não são físicas [17]. Seja $N$ o número de divisões da malha, figura 2.2 mostra o esquema de discretização do domínio $\Omega$ para $N = 5$.

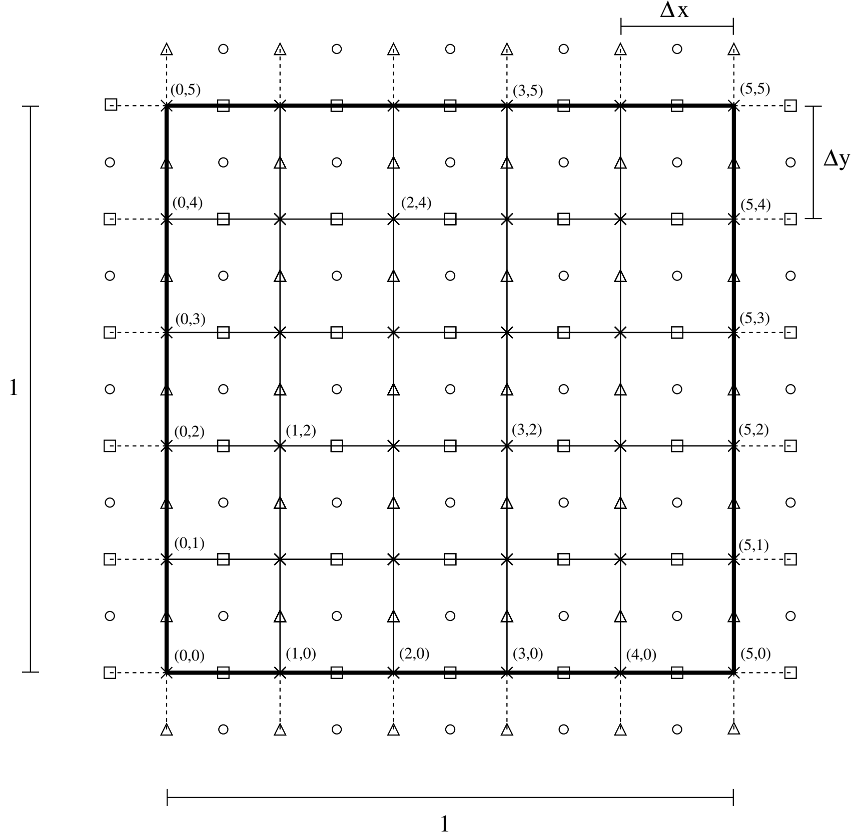



Figura 2.2: Malha defasada 5x5. Triângulo: componente u. Quadrado: componente v. Círculo: pressão p.

Partindo finalmente para discretização das derivadas espaciais do problema, isolando o termo $\mathbf{u}^*$ na equação 2.22, escrevendo cada componente $u^*$ e $v^*$ individualmente e fazendo aproximação das derivadas por diferenças finitas através do truncamento da expansão em série de Taylor, podemos escrever

$$\begin{aligned} u^*_{i,j} = u^k_{i,j} - \Delta t \left[ u^k_{i,j} \left( \frac{u^k_{i+1,j} - u^k_{i-1,j}}{2\Delta x} \right) + v^k_{i,j} \left( \frac{u^k_{i,j+1} - u^k_{i,j-1}}{2\Delta y} \right) \right] + \\ \frac{\Delta t}{Re} \left[ \left( \frac{u^k_{i+1,j} - 2u^k_{i,j} + u^k_{i-1,j}}{\Delta x^2} \right) + \left( \frac{u^k_{i,j+1} - 2u^k_{i,j} + u^k_{i,j-1}}{\Delta y^2} \right) \right] \end{aligned} \tag{2.26}$$

e

$$\begin{aligned} v^*_{i,j} = v^k_{i,j} - \Delta t \left[ u^k_{i,j} \left( \frac{v^k_{i+1,j} - v^k_{i-1,j}}{2\Delta x} \right) + v^k_{i,j} \left( \frac{v^k_{i,j+1} - v^k_{i,j-1}}{2\Delta y} \right) \right] + \\ \frac{\Delta t}{Re} \left[ \left( \frac{v^k_{i+1,j} - 2v^k_{i,j} + v^k_{i-1,j}}{\Delta x^2} \right) + \left( \frac{v^k_{i,j+1} - 2v^k_{i,j} + v^k_{i,j-1}}{\Delta y^2} \right) \right]. \end{aligned} \tag{2.27}$$

Agora, abrindo a equação 2.24 e fazendo aproximação das derivadas por diferenças finitas,

podemos escrevê-la como

$$\frac{p_{i+1,j}^{k+1} - 2p_{i,j}^{k+1} + p_{i-1,j}^{k+1}}{\Delta x^2} + \frac{p_{i,j+1}^{k+1} - 2p_{i,j}^{k+1} + p_{i,j-1}^{k+1}}{\Delta y^2} = \frac{1}{\Delta t}\left(\frac{u_{i+1,j}^{*} + u_{i,j}^{*}}{\Delta x} + \frac{v_{i,j+1}^{*} + v_{i,j}^{*}}{\Delta y}\right). \tag{2.28}$$

Além da própria equação da pressão, é necessário também discretizar a equação 2.25, que descreve as condições de contorno do problema para a pressão. O termo $\hat{\mathbf{n}}$ na equação 2.25 depende da parede na qual a equação é aplicada. Dessa forma, então a discretização da equação 2.25 será feita para cada uma das paredes da cavidade. Fazendo então a discretização da equação 2.25 por diferenças finitas para a parede de baixo, temos que

$$\frac{p_{i,0}^{k+1} - p_{i,-1}^{k+1}}{\Delta y} = 0 \tag{2.29}$$

$$\Rightarrow p_{i,0}^{k+1} = p_{i,-1}^{k+1}. \tag{2.30}$$

Fazendo agora a discretização da equação 2.25 por diferenças finitas para a parede de cima, temos que

$$\frac{p_{i,N}^{k+1} - p_{i,N-1}^{k+1}}{\Delta y} = 0 \tag{2.31}$$

$$\Rightarrow p_{i,N}^{k+1} = p_{i,N-1}^{k+1}, \tag{2.32}$$

onde N é o número de divisões da malha da figura 2.2.

Fazendo agora a discretização da equação 2.25 por diferenças finitas para a parede esquerda, temos que

$$\frac{p_{0,j}^{k+1} - p_{-1,j}^{k+1}}{\Delta y} = 0 \tag{2.33}$$

$$\Rightarrow p_{0,j}^{k+1} = p_{-1,j}^{k+1}. \tag{2.34}$$

Fazendo agora a discretização da equação 2.25 por diferenças finitas para a parede direita, temos que

$$\frac{p_{N,j}^{k+1} - p_{N-1,j}^{k+1}}{\Delta y} = 0 \tag{2.35}$$

$$\Rightarrow p_{N,j}^{k+1} = p_{N-1,j}^{k+1}. \tag{2.36}$$

Por último, isolando o termo $\mathbf{u}^{k+1}$ na equação 2.21, escrevendo cada componente $u^{k+1}$ e $v^{k+1}$ individualmente e fazendo a aproximação das derivadas por diferenças finitas, temos que

$$u_{i,j}^{k+1} = u^* - \Delta t \left( \frac{p_{i,j}^{k+1} - p_{i-1,j}^{k+1}}{\Delta x} \right) \tag{2.37}$$

e

$$v_{i,j}^{k+1} = v^* - \Delta t \left( \frac{p_{i,j}^{k+1} - p_{i,j-1}^{k+1}}{\Delta y} \right). \tag{2.38}$$

Para garantir a estabilidade numérica do método das diferenças finitas para o problema da cavidade, algumas restrições devem ser obedecidas ao determinar os valores de Δt, Δx e Δy. As condições de estabilidade são dadas por

$$\Delta t < \Delta x \tag{2.39}$$

e

$$\Delta t < \frac{1}{4} Re \Delta x^2. \tag{2.40}$$

A restrição 2.39 é a chamada condição de CFL e existe para limitar o passo de tempo a um valor menor do que aquele em que uma informação é advectada através de uma célula da malha. Fisicamente falando, isso significa que o fluido não pode percorrer uma distância maior do que o comprimento de uma célula da malha a cada passo de tempo [21]. A equação 2.40 é uma restrição que impõe que o passo de tempo seja menor do que o tempo característico de difusão do problema [17]. Aqui vale dizer que os critérios para $\Delta x$ estabelecidos nas equações de 2.39 a 2.40 são necessários também para $\Delta y$, uma vez que ao longo de todo trabalho temos que $\Delta x = \Delta y$. Todos os resultados apresentados no capítulo 3 respeitam as condições de estabilidade apresentadas nesta seção.

## 2.4 Sistema de equações do problema da cavidade

Agora será montado de fato o sistema de equações oriundo da equação 2.28. O sistema será montado para $N = 3$, conforme ilustra a figura 2.3, somente para critérios de ilustração, uma vez que o sistema a ser resolvido de fato terá ordem muito maior que $N = 3$. Para montar o sistema, a equação 2.28 deve ser aplicada a cada um dos pontos internos da malha da figura 2.3 e, nas paredes, as equações 2.30, 2.32, 2.34 e 2.36 deverão ser também aplicadas.

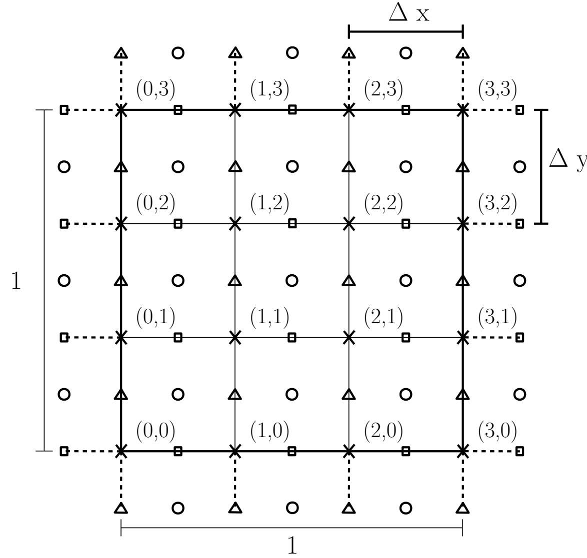



Figura 2.3: Malha defasada 3x3. Triângulo: componente u. Quadrado: componente v. Círculo: pressão p.

Aplicando então para o ponto (0,0) e considerando que $\Delta x = \Delta y$ e passando esse termo para o lado direito da equação, temos

$$\text{Para o ponto (0,0), temos: } p_{1,0} - 2p_{0,0} + p_{0,1} = b_{0,0} \tag{2.41}$$

$$\text{Para o ponto (1,0), temos: } p_{2,0} - 3p_{1,0} + p_{0,0} + p_{1,1} = b_{1,0} \tag{2.42}$$

$$\text{Para o ponto (2,0), temos: } -2p_{2,0} + p_{1,0} + p_{2,1} = b_{2,0} \tag{2.43}$$

$$\text{Para o ponto (0,1), temos: } p_{1,1} - 3p_{0,1} + p_{0,2} + p_{0,0} = b_{0,1} \tag{2.44}$$

$$\text{Para o ponto (1,1), temos: } p_{2,1} - 4p_{1,1} + p_{0,1} + p_{1,2} + P_{1,0} = b_{1,1} \tag{2.45}$$

$$\text{Para o ponto (2,1), temos: } -3p_{2,1} + p_{1,1} + p_{1,0} + p_{1,2} = b_{2,1} \tag{2.46}$$

$$\text{Para o ponto (0,2), temos: } p_{1,2} - 2p_{0,2} + p_{0,1} = b_{0,2} \tag{2.47}$$

$$\text{Para o ponto (1,2), temos: } p_{2,2} - 3p_{1,2} + p_{0,2} + p_{1,1} = b_{1,2} \tag{2.48}$$

$$\text{Para o ponto (2,2), temos:} \quad -2p_{2,2} + p_{1,2} + p_{2,1} = b_{2,2} \tag{2.49}$$

Escrevendo o sistema em forma matricial $\mathbf{Ax} = \mathbf{b}$, temos

$$\begin{pmatrix} -2 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -3 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -3 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -3 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} P_{0,0} \\ P_{1,0} \\ P_{2,0} \\ P_{0,1} \\ P_{1,1} \\ P_{2,1} \\ P_{0,2} \\ P_{1,2} \\ P_{2,2} \end{pmatrix} = \begin{pmatrix} b_{0,0} \\ b_{1,0} \\ b_{2,0} \\ b_{0,1} \\ b_{1,1} \\ b_{2,1} \\ b_{0,2} \\ b_{1,2} \\ b_{2,2} \end{pmatrix}, \tag{2.50}$$

onde $b_{i,j}$ é dado como

$$b_{i,j} = \frac{\Delta x}{\Delta t} \left( u^*_{i+1,j} - u^*_{i,j} + v^*_{i,j+1} - v^*_{i,j} \right). \tag{2.51}$$

Como se pode perceber pela equação 2.50, a matriz $\mathbf{A}$ dos coeficientes é uma matriz simétrica e esparsa, ou seja, a maioria dos termos são nulos e isso pode e deve ser usado a nosso favor. De fato, ao se multiplicar a matriz $\mathbf{A}$ por um vetor qualquer, o fato da matriz ser esparsa faz com que o produto seja previsível. Para ilustrar isso, vamos observar as equações de 2.41 até 2.49. Para cada ponto (i,j), a pressão no ponto (i,j) é somada à pressão no ponto da direita (i+1,j), da esquerda (i-1,j), de cima (i,j+1) e de baixo (i,j-1), com cada uma dessas 5 pressões multiplicadas por um fator específico. Podemos, portanto, escrever de forma genérica que o produto cada elemento do vetor resultante do produto $\mathbf{Ap} = \mathbf{b}$ pode ser escrito como

$$r_{i,j}p_{i+1,j} + c_{i,j}p_{i,j} + l_{i,j}p_{i-1,j} + u_{i,j}p_{i,j+1} + d_{i,j}p_{i,j-1} = b_{i,j} \tag{2.52}$$

onde os fatores $r_{i,j}$, $c_{i,j}$, $l_{i,j}$, $u_{i,j}$ e $d_{i,j}$ dependem de cada posição (i,j) da malha. Para o caso simplificado de $N = 3$, esses fatores serão da seguinte forma

$$r = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}, \tag{2.53}$$

$$c = \begin{pmatrix} -2 & -3 & -2 \\ -3 & -4 & -3 \\ -2 & -3 & -2 \end{pmatrix}, \tag{2.54}$$

$$l = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}, \tag{2.55}$$

$$u = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix}, \tag{2.56}$$

e

$$r = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}. \tag{2.57}$$

Os fatores $r_{i,j}$, $c_{i,j}$, $l_{i,j}$, $u_{i,j}$ e $d_{i,j}$ serão chamados daqui pra frente de coeficientes do estêncil da matriz $\mathbf{A}$. Para exemplificar a aplicação do uso dos coeficientes do estêncil para determinar o resultado de um produto entre a matriz $\mathbf{A}$ por um vetor $\mathbf{p}$, vamos considerar o ponto (1,1) da malha apresentada na figura 2.3. Usando a equação 2.52 para o ponto (1,1), temos que

$$r_{1,1}p_{2,1} + c_{1,1}p_{1,1} + l_{1,1}p_{0,1} + u_{1,1}p_{1,2} + d_{1,1}p_{1,0} = b_{1,1}. \tag{2.58}$$

Das matrizes dos coeficientes do estêncil (equações 2.53, 2.54, 2.55, 2.56 e 2.57), temos que $r_{1,1} = 1$, $c_{1,1} = -4$, $l_{1,1} = 1$, $u_{1,1} = 1$, e $d_{1,1} = 1$ e, portanto, a equação 2.58 se torna

$$p_{2,1} - 4p_{1,1} + p_{0,1} + p_{1,2} + p_{1,0} = b_{1,1}, \tag{2.59}$$

que é exatamente igual à equação 2.45. Essa estratégia de se utilizar os coeficientes do estêncil pra determinar o resultado de um produto da matriz $\mathbf{A}$ por um vetor é muito útil especialmente para sistemas de dimensão elevada, uma vez que não é necessário armazenar a matriz $\mathbf{A}$, que tem dimensão $N^2 \times N^2$. As dimensões da matriz $\mathbf{A}$ crescem com $N^2$, enquanto que as matrizes dos coeficientes do estêncil crescem com $N$. Sendo assim, para sistemas de dimensão muito alta, é muito mais vantajoso trabalhar com as 5 matrizes de coeficientes do estêncil, cada uma com dimensões $N \times N$, do que trabalhar apenas com a matriz $\mathbf{A}$, que tem ordem $N^2 \times N^2$. Além do fato das matrizes dos coeficientes do estêncil ocuparem menos memória do que a matriz $\mathbf{A}$ para sistemas com ordem alta, os algoritmos que usam a abordagem do estêncil são muito mais rápidos do que um algoritmo que ignora a esparsidade da matriz $\mathbf{A}$ e faz todas as multiplicações com os termos nulos da matriz $\mathbf{A}$. A abordagem através dos coeficientes do estêncil ignora quase que completamente os termos nulos da matriz $\mathbf{A}$ e por isso é muito mais rápida, já que maior parte da matriz $\mathbf{A}$ é nula para ordens mais altas. Quanto maior for a matriz $\mathbf{A}$ para esse caso particular da cavidade, maior será a quantidade de termos nulos em comparação com o tamanho da matriz. Podemos usufruir dos benefícios da aplicação dessa abordagem dos coeficientes dos estêncil porque, em todos os métodos iterativos de resolução de sistemas de equações lineares que serão tratados nesse trabalho, a matriz $\mathbf{A}$ não precisa estar explicitada, mas somente o produto de $\mathbf{A}$ por um vetor qualquer de dimensões apropriadas.

## 2.5 Métodos de solução de sistemas de equações lineares

A resolução de sistemas de equações lineares pode ser feita, de maneira geral, de duas formas: através de métodos diretos, o que nos leva a uma solução exata, ou através de métodos iterativos,

o que nos leva a uma solução não exata, mas tão próxima quanto se queira da solução exata, dada uma tolerância [22, 23]. Para sistemas grandes, os métodos iterativos, ainda que não forneçam uma solução exata, são melhores do que os métodos diretos porque excluem a necessidade de trabalhar com a matriz do sistema em sua forma completa, tornando-se métodos muito mais rápidos. Dentre os métodos diretos, temos como destaque a Eliminação Gaussiana e a Decomposição LU. Entre os métodos iterativos mais comuns, temos o método de Jacobi, o método de Gauss-Seidel, o método do Gradiente Conjugado e o método do Multigrid. Nas seções subsequentes se faz a revisão de cada um desses métodos diretos e iterativos.

De uma forma genérica, um sistema de equações lineares pode ser representado de forma compacta como

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{2.60}$$

onde $\mathbf{A}$ é a matriz dos coeficientes, $\mathbf{b}$ é um vetor coluna e $\mathbf{x}$ é o vetor de incógnitas. A equação 2.60 pode ser reescrita na sua forma explícita da seguinte maneira:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{pmatrix}. \tag{2.61}$$

Encontrar a solução do sistema 2.60 consiste em encontrar os valores das componentes de $\mathbf{x}$ para uma matriz $\mathbf{A}$ e um vetor $\mathbf{b}$ dados. As seções 2.6 e 2.7 se comprometem a fazer uma revisão das formas diretas e iterativas, respectivamente, de encontrar o vetor de incógnitas $\mathbf{x}$ para o caso em que $\mathbf{A}$ é uma matriz quadrada $N \times N$.

## 2.6 Métodos diretos

Os métodos diretos de resolução de sistemas de equações lineares não são o foco do trabalho atual, entretanto serão apresentados nas seções 2.6.1 e 2.6.2 dois métodos diretos bastante populares simplesmente para exemplificar como são os métodos diretos. Os métodos diretos apresentados serão o método da Eliminação Gaussiana e o método da Decomposição LU.

### 2.6.1 Eliminação Gaussiana

O método da Eliminação Gaussiana tem como estratégia transformar a matriz $\mathbf{A}$ da equação 2.61 em uma matriz triangular superior através de operações elementares entres as linhas da matriz expandida $(\mathbf{A}|\mathbf{b})$ [22]. As operações elementares entre linhas da matriz $(\mathbf{A}|\mathbf{b})$ nos levam a um novo sistema que é equivalente ao sistema original, ou seja, ambos possuem a mesma solução. Uma vez que a matriz $\mathbf{A}$ esteja já transformada em uma matriz triangular superior, a última componente do vetor $\mathbf{x}$, que é $x_N$, pode ser obtida de forma direta. Esse resultado obtido para $x_N$ é substituído

na equação $n-1$ do sistema novo sistema e obtém-se então o valor de $x_{N-1}$, que será, por sua vez, substituído na equação $n-2$ para se obter o valor de $x_{N-2}$. Assim se segue até se obter o valor de todas as incógnitas $x_i$ do vetor $\mathbf{x}$. As operações elementares possíveis entre as linhas da matriz expandida $(\mathbf{A}|\mathbf{b})$ são:

- Multiplicação de linha por uma constante;
- Subtração de um múltiplo de uma linha por outra;
- Troca entre duas linhas.

### 2.6.2 Decomposição LU

O método da Decomposição LU consiste em reescrever a matriz $\mathbf{A}$ da equação 2.60 como o produto entre duas matrizes $\mathbf{L}$ e $\mathbf{U}$, onde $\mathbf{L}$ é uma matriz triangular inferior e $\mathbf{U}$ é uma matriz triangular superior, conforme explica Ruggiero e Lopes [22]. Dessa forma, temos que a equação 2.60 é reescrita como

$$(\mathbf{LU})\mathbf{x} = \mathbf{b}. \tag{2.62}$$

Se fizermos $\mathbf{Ux} = \mathbf{y}$, temos então que a equação 2.62 pode ser reescrita como $\mathbf{Ly} = \mathbf{b}$. Dessa forma, a solução do sistema original 2.60 pode ser encontrada através da solução dos sistemas lineares triangulares

$$\mathbf{Ux} = \mathbf{y} \tag{2.63}$$

e

$$\mathbf{Ly} = \mathbf{b}. \tag{2.64}$$

A vantagem desse método está no fato de que as equações 2.63 e 2.64 são facilmente resolvidas por se tratarem de sistemas cujas matrizes dos coeficientes são triangulares.

Para se obter as matrizes $\mathbf{L}$ e $\mathbf{U}$, deve-se transformar a matriz $\mathbf{A}$ em uma matriz triangular superior, assim como no método da Eliminação Gaussiana. Ao fazer essa transformação, uma série de operações elementares são executadas na matriz $\mathbf{A}$. Por exemplo, ao zerar todos os termos abaixo do termo $a_{11}$ da matriz $\mathbf{A}$, isso equivale a pré-multiplicar a matriz $\mathbf{A}$ por uma matriz $\mathbf{M}^{(\mathbf{1})}$ tal que

$$\mathbf{M}^{(\mathbf{1})} = \begin{pmatrix} 1 & & & & \\ m_{21} & 1 & & & \\ m_{31} & & 1 & & \\ \vdots & & & \ddots & \\ m_{N1} & & & & 1 \end{pmatrix}, \tag{2.65}$$

onde todos os espaços vazios são, na verdade, zeros. Continuando, ao zerar todos os termos abaixo do termo $a_{22}$ da matriz $\mathbf{A}$, isso equivale a pré-multiplicar a matriz $\mathbf{A}$ por uma matriz $\mathbf{M^{(2)}}$ tal que

$$\mathbf{M^{(2)}} = \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & m_{32} & 1 & & \\ & \vdots & & \ddots & \\ & m_{N2} & & & 1 \end{pmatrix}. \tag{2.66}$$

O processo continua até que tenhamos transformado a matriz $\mathbf{A}$ em uma matriz triangular superior. Temos então que, ao terminar a triangularização da matriz $\mathbf{A}$, teremos a equação

$$\mathbf{A^{(N-1)}} = \mathbf{M^{(1)}}\mathbf{M^{(2)}}...\mathbf{M^{(N-1)}}\mathbf{A}, \tag{2.67}$$

onde $\mathbf{A^{(N-1)}}$ é a matriz triangular superior originada a partir da transformação de $\mathbf{A}$ por operações elementares. Isolando a matriz $\mathbf{A}$ na equação 2.67, temos então

$$\mathbf{A} = [\mathbf{M^{(1)}}]^{-1}[\mathbf{M^{(2)}}]^{-1}...[\mathbf{M^{(N-1)}}]^{-1}\mathbf{A^{(N-1)}}. \tag{2.68}$$

Como todas as matrizes $\mathbf{M^{(i)}}$ na equação 2.68 são matrizes triangulares inferiores, então o produto $[\mathbf{M^{(1)}}]^{-1}[\mathbf{M^{(2)}}]^{-1}...[\mathbf{M^{(N-1)}}]^{-1}$ também é uma matriz triangular inferior. Portanto, podemos escrever que

$$\mathbf{L} = [\mathbf{M^{(1)}}]^{-1}[\mathbf{M^{(2)}}]^{-1}...[\mathbf{M^{(N-1)}}]^{-1} \tag{2.69}$$

e

$$\mathbf{U} = \mathbf{A^{(N-1)}} \tag{2.70}$$

e então, finalmente, a equação 2.68 pode ser reescrita como

$$\mathbf{A} = \mathbf{LU}. \tag{2.71}$$

## 2.7 Métodos iterativos

A ideia básica dos métodos iterativos de resolução de sistemas de equações lineares é, a partir de uma aproximação inicial dada para a solução $\mathbf{x}$ do sistema $\mathbf{Ax} = \mathbf{b}$, ir refinando cada vez mais essa aproximação a cada iteração até que a solução aproximada obtida esteja suficientemente próxima da solução exata, conforme explica Ascher [23].

Para se obter um algoritmo que nos permita chegar a uma solução aproximada para o sistema a partir de uma aproximação inicial, iremos reescrever o sistema $\mathbf{Ax} = \mathbf{b}$ num sistema do tipo [1]

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{M}^{-1}\mathbf{r}_k, \tag{2.72}$$

[1] Há outras formas de escrever a equação 2.72. O anexo I mostra a equivalência da forma da equação 2.72 com outra forma comumente encontrada em livros texto.

onde $\mathbf{r}_k = \mathbf{b} - \mathbf{A}\mathbf{x_k}$ é o resíduo. Dessa forma, temos uma função de iteração. Partindo de uma aproximação inicial $\mathbf{x_0}$, as aproximações seguintes são dadas por

$$\mathbf{x_1} = \mathbf{x}_0 + \mathbf{M}^{-1}\mathbf{r_0},$$
$$\mathbf{x_2} = \mathbf{x}_1 + \mathbf{M}^{-1}\mathbf{r_1},$$
$$\vdots$$
$$\mathbf{x_{k+1}} = \mathbf{x}_k + \mathbf{M}^{-1}\mathbf{r_k},$$

A forma como o sistema linear $\mathbf{Ax} = \mathbf{b}$ é transformado em uma função de iteração $\mathbf{x}_{k+1} = \mathbf{x}_k + M^{-1}\mathbf{r}_k$ depende da escolha da matriz $\mathbf{M}$, que varia com cada método iterativo utilizado. Quando um método iterativo pode ser escrito na forma dada pela equação 2.72, o método iterativo é chamado de estacionário ou de relaxação [23]. Os métodos iterativos estacionários tratados nesse trabalho serão o Método de Jacobi, o Método de Gauss-Seidel e o Método SOR. Já os métodos iterativos não estacionários a serem explorados serão o Método do Gradiente Conjugado e o Método do Multigrid.

### 2.7.1 Critério de parada

Ao se adotar um método iterativo para solucionar um sistema de equações lineares, assumimos que não teremos a solução exata do sistema, mas teremos uma solução aproximada tão próxima quanto se queira da solução exata para uma dada tolerância $\epsilon$. Essa tolerância $\epsilon$ define o critério de parada das iterações, ou seja, a solução aproximada será iterada até que o erro definido de uma forma conveniente seja menor que a tolerância. A forma como esse erro é definido pode variar a depender de como queremos abordar o problema. O resíduo relativo é uma escolha comum de critério de parada [23]. Dessa forma, para o presente trabalho, esse erro será tomado como o resíduo relativo, definido pela equação

$$r^k_{rel} = \frac{||r_k||}{||b||}, \tag{2.73}$$

onde $r^k_{rel}$ é o resíduo relativo da k-ésima iteração, $||r_k||$ é a norma infinito do resíduo oriundo da k-ésima iteração e $||b||$ é a norma infinito do vetor coluna do sistema. A norma infinito de um vetor $\mathbf{x}$ é definida como

$$||\mathbf{x}|| = max(|x_1|, |x_2|, ..., |x_N|). \tag{2.74}$$

Assim, nos métodos, uma solução aproximada $\mathbf{x_k}$ será tomada como solução final se $r^k_{rel} < \epsilon$.

### 2.7.2 Método de Jacobi

No método de Jacobi, a matriz M do função de iteração 2.72 é dada por

$$\mathbf{M} = \mathbf{D} = \begin{pmatrix} a_{11} & 0 & \dots & 0 \\ 0 & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{NN} \end{pmatrix}. \tag{2.75}$$

Substituindo a matriz **M** na equação 2.72, temos que essa equação pode ser escrita na sua forma de componentes como

$$x_i^{k+1} = \frac{1}{a_{ii}} \left[ b_i - \sum_{j=1, j \neq i}^{N} a_{ij} x_j^k \right]. \tag{2.76}$$

A equação 2.76 não é a forma como foi implementada computacionalmente nesse trabalho. Ao invés disso, ela então será reescrita na forma

$$x_i^{k+1} = x_i^k + R_i^k, \tag{2.77}$$

onde, $R_i^k$ é dado por

$$R_i^k = \frac{1}{a_{ii}} \left[ b_i - \sum_{j=1}^{N} a_{ij} x_j^k \right]. \tag{2.78}$$

Dessa forma, o método de Jacobi consiste em, a partir de uma aproximação inicial $\mathbf{x_0}$, obter as aproximações seguintes $\mathbf{x_1}$, $\mathbf{x_2}$, ...,$\mathbf{x_k}$ até que $erro_k < \epsilon$. O método de Jacobi, como será visto, é o mais lento entre todos os métodos iterativos, porém é o mais fácil de ser paralelizado.

### 2.7.3 Método de Gauss-Seidel

No método de Gauss-Seidel, a matriz **M** é escolhida de tal maneira que

$$\mathbf{M} = \mathbf{E} = \begin{pmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \dots & a_{NN} \end{pmatrix}. \tag{2.79}$$

Escrevendo $\mathbf{x^{(k+1)}}$ na forma de componentes, temos que

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[ b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{N} a_{ij} x_j^{(k)} \right]. \tag{2.80}$$

A equação 2.80 não é a forma como foi implementada computacionalmente nesse trabalho. Para implementar o método de Gauss-Seidel, a equação2.80 foi reescrita na forma

$$x_i^k = x_i^k + R_i^k, \tag{2.81}$$

onde $R_i^k$ é dado por

$$R_i^k = \frac{1}{a_{ii}} \left[ b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i}^{N} a_{ij} x_j^k \right]. \quad (2.82)$$

A vantagem do método de Gauss-Seidel sobre o método de Jacobi está no fato de que no cálculo de $x_i^{k+1}$, usa-se já os valores de $x_{i-s}^{k+1}$, com $0 < s < i$. Ou seja, na k-ésima iteração, já se usa valores calculados de $x_i$ nessa mesma iteração nos cálculos das componentes seguintes. Isso faz com que o processo iterativo convirja muito mais rapidamente do que quando comparado com o método de Jacobi.

### 2.7.4 Método de Sobre-Relaxação Sucessiva (SOR)

É possível modificar o método de Gauss-Seidel através da implementação de um fator $\omega$ ao método e obter um ganho considerável de velocidade de convergência do método. De fato, o método de Jacobi também é passível de ser modificado da mesma maneira, porém os ganhos em termos de velocidade de convergência não são tão expressivos quanto aqueles observados no método de Gauss-Seidel [23]. Para o método de SOR, a matriz $\mathbf{M}$ é dada por

$$\mathbf{M} = \frac{1-\omega}{\omega}\mathbf{D} + \mathbf{E}. \quad (2.83)$$

O parâmetro $\omega$ é chamado de fator de relaxação e, para o método de Gauss-Seidel, ele é implementado na função de iteração como

$$x_i^{k+1} = (1-\omega)x_i^k + \frac{\omega}{a_{ii}} \left[ b_i - \sum_{j<i} a_{ij} x_j^{k+1} - \sum_{j>1} a_{ij} x_j^k \right]. \quad (2.84)$$

A equação 2.84 não é a forma como foi implementada computacionalmente nesse trabalho. Para implementar o método de SOR, convém reescrever a equação 2.84 na forma

$$x_i^k = x_i^k + \omega R_i^k, \quad (2.85)$$

onde $R_i^k$ é dado por

$$R_i^k = \frac{1}{a_{ii}} \left[ b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i}^{N} a_{ij} x_j^k \right]. \quad (2.86)$$

Portanto, a equação 2.85 é que será implementada numericamente para o método de SOR.

Nesse momento, vale ressaltar que, para matrizes oriundas da discretização do operador laplaciano, que é o caso desse trabalho, existe uma equação que descreve o $\omega$ ótimo em função do raio espectral $\rho$ da matriz de iteração do método de Jacobi [23]. Essa equação é dada por

$$\omega = \frac{2}{1+\sqrt{1-\rho^2}}. \quad (2.87)$$

O raio espectral $\rho$ e a matriz do método de Jacobi são definidos na seção 2.7.7, no Teorema 05.

### 2.7.5 Método do Gradiente Conjugado

O método do Gradiente Conjugado é o primeiro método iterativo não estacionário a ser apresentado. O elegante método do Gradiente Conjugado consiste em transformar o problema de encontrar a solução do sistema de equações lineares em um problema de encontrar o ponto de mínimo de uma função, ou seja, torna-se um problema de otimização [23]. Para fazer essa transformação, consideremos a seguinte função

$$\phi(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x}. \tag{2.88}$$

Para encontrarmos um ponto de crítico da função $\phi(\mathbf{x})$, devemos calcular o gradiente de $\phi(\mathbf{x})$ e igualar a zero. Assumindo que $\mathbf{A}$ seja simétrica, isso nos leva à equação

$$\nabla\phi(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0}. \tag{2.89}$$

Se observamos atentamente a equação 2.89, percebemos que é igual à equação 2.60. Se a matriz $\mathbf{A}$ for definida positiva, então garantimos que a segunda derivada de 2.88 é positiva, portanto a função tem concavidade positiva e possui um ponto de mínimo, o que garante que a solução $\mathbf{x}$ do sistema original é um ponto de mínimo da função 2.88.

A função de iteração utilizada no método do gradiente conjugado é dada por

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{p}_k. \tag{2.90}$$

Antes de entrar especificamente no método do Gradiente Conjugado, será brevemente apresentado o método da Descida mais Íngreme, que precede o Método do Gradiente Conjugado em termos de construção de raciocínio.

A função de iteração do método da Descida mais Íngreme é dada por

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{r}_k, \tag{2.91}$$

onde $\mathbf{r}_k$ é o resíduo dado por $\mathbf{r}_k = \mathbf{b} - \mathbf{A}\mathbf{x_k}$, e estamos interessados em obter o escalar $\alpha_k$ tal que a função $\phi(\mathbf{x}_k + \alpha\mathbf{r}_k)$ seja minimizada em relação a $\alpha$. Substituindo a equação 2.91 na equação 2.88, derivando em relação a $\alpha$ e isolando, obtemos que

$$\alpha_k = \frac{\langle\mathbf{r}_k, \mathbf{r}_k\rangle}{\langle\mathbf{r}_k, \mathbf{A}\mathbf{r}_k\rangle}, \tag{2.92}$$

onde o operador $\langle\cdot,\cdot\rangle$ é o produto escalar. Com a equação 2.92 e 2.91, temos então condições de criar um laço de iteração. Esse laço é o chamado Método da Descida Mais Íngrime.

O método do Gradiente Conjugado consiste em, ao invés de usar $\mathbf{p}_k$ igual a $\mathbf{r}_k$, usa-se $\mathbf{p}_k$ dado pela equação

$$\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \frac{\delta_{k+1}}{\delta_k}\mathbf{p}_k \tag{2.93}$$

onde $\delta_k = \langle r_k, r_k \rangle$. Além disso, o termo $\alpha_k$ é calculado como sendo

$$\alpha_k = \frac{\delta_k}{\langle \mathbf{p}_k, \mathbf{s}_k \rangle}. \tag{2.94}$$

O algoritmo para o Método do Gradiente Conjugado é fornecido a seguir.

**Algoritmo: Método do Gradiente Conjugado** Dado um chute inicial $\mathbf{x}_0$ e uma tolerância $\epsilon$, estabeleça inicialmente $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$, $\delta_0 = \mathbf{r}_0 \cdot \mathbf{r}_0$, k=0 e $\mathbf{p}_0 = \mathbf{r}_0$.

Enquanto erro>$\epsilon$, faça:

- $\mathbf{s}_k = \mathbf{A}\mathbf{p}_k$
- $\alpha_k = \frac{\delta_k}{\mathbf{p}_k \cdot \mathbf{s_k}}$
- $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$.
- $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{r}_k$.
- $\delta_{k+1} = \mathbf{r}_{k+1} \cdot \mathbf{r}_{k+1}$
- $\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \frac{\delta_{k+1}}{\delta_k} \mathbf{p}_k$.
- k=k+1

Na maioria dos casos quando a matriz $\mathbf{A}$ é grande, é vantajoso usar o método do Gradiente Conjugado com o que chamamos de pré-condicionamento [24]. O processo consiste em multiplicar o sistema $\mathbf{A}\mathbf{x} = \mathbf{b}$ pelo inverso de uma matriz condicionadora $\mathbf{P}$, nos levando a

$$\mathbf{P}^{-1}\mathbf{A}\mathbf{x} = \mathbf{P}^{-1}\mathbf{b}. \tag{2.95}$$

Como resultado dessa operação, teremos um novo sistema que estará mais bem condicionado, ou seja, um sistema em que a matriz associada tem um número de condição $\kappa$ menor. Sendo um novo sistema mais bem condicionado, então a solução aproximada desse novo sistema tenderá a convergir de maneira mais rápida para a solução do que o sistema original. O grau de condicionamento de uma matriz $\mathbf{A}$ é calculado através da expressão [23]

$$\kappa(\mathbf{A}) = \parallel \mathbf{A}^{-1} \parallel \parallel \mathbf{A} \parallel, \tag{2.96}$$

onde a norma induzida da matriz $\parallel \mathbf{A} \parallel$ pode ser livremente escolhida. Dessa forma, para que seja vantajoso fazer o pré-condicionamento com uma matriz $\mathbf{P}$, é necessário que $\kappa(\mathbf{P}^{-1}\mathbf{A}) \ll \kappa(\mathbf{A})$.

A escolha da matriz condicionadora ótima para cada caso é um tópico de estudo e existem opções bastante sofisticadas, mas algumas matrizes são escolhas comuns, como por exemplo a matriz $\mathbf{D} = diag(\mathbf{A})$. Assim como a matriz diagonal $\mathbf{D}$ é uma escolha comum de matriz pré-condicionadora, a matriz $\mathbf{M_{SSOR}}$, definida como

$$\mathbf{M_{SSOR}} = \frac{1}{\omega(2-\omega)}(\mathbf{D} - \omega\mathbf{L})\mathbf{D}^{-1}(\mathbf{D} - \omega\mathbf{U}), \tag{2.97}$$

também uma escolha comum de pré-condicionamento [24].

Na equação 2.97, a matriz $\mathbf{L}$ é uma matriz triangular inferior formada pelas componentes abaixo da diagonal da matriz $\mathbf{A}$ e a matriz $\mathbf{U}$ é uma matriz triangular superior formada pelos elementos acima da diagonal da matriz $\mathbf{A}$ de tal maneira que $\mathbf{A} = \mathbf{D} - \mathbf{L} - \mathbf{U}$.

O algoritmo para o Método do Gradiente Conjugado Pré-condicionado por uma matriz $\mathbf{P}$ é fornecido a seguir.

**Algoritmo: Método do Gradiente Conjugado Pré-condicionado** Dado um chute inicial $\mathbf{x}_0$ e uma tolerância tol, estabeleça inicialmente $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$, $\mathbf{h}_0 = \mathbf{P}^{-1}\mathbf{r}_0$, $\delta_0 = \mathbf{r}_0 \cdot \mathbf{h}_0$, k=0 e $\mathbf{p}_0 = \mathbf{h}_0$.

Enquanto erro>tol, faça:

- $\mathbf{s}_k = \mathbf{A}\mathbf{p}_k$
- $\alpha_k = \frac{\delta_k}{\mathbf{p}_k \cdot \mathbf{s_k}}$
- $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$.
- $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{s}_k$.
- $\mathbf{h}_{k+1} = \mathbf{P}^{-1}\mathbf{r}_{k+1}$
- $\delta_{k+1} = \mathbf{r}_{k+1} \cdot \mathbf{h}_{k+1}$
- $\mathbf{p}_{k+1} = \mathbf{h}_{k+1} + \frac{\delta_{k+1}}{\delta_k}\mathbf{p}_k$.
- k=k+1

É necessário dar uma atenção especial para o termo $\mathbf{h}_{k+1} = \mathbf{P}^{-1}\mathbf{r}_{k+1}$ no algoritmo do Gradiente Conjugado Pré-condicionado. Apesar de parecer, não é necessário calcular a inversa de $\mathbf{P}$ para se obter, $\mathbf{h}_{k+1}$. Para mostrar isso, será utilizada a matriz $\mathbf{M_{SSOR}}$, definida pela equação 2.97. Seja a matriz $\mathbf{A}$ dada pela equação 2.50, fatora-se a matriz $\mathbf{A}$ como

$$\mathbf{A} = \mathbf{D} - \mathbf{L} - \mathbf{U}, \tag{2.98}$$

onde, para o caso em que $N = 3$,

$$\mathbf{D} = \begin{pmatrix} -2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2 \end{pmatrix}, \tag{2.99}$$

$$\mathbf{L} = -\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \tag{2.100}$$

e

$$\mathbf{U} = -\begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{2.101}$$

Então temos que o termo $\mathbf{D}^{-1}(\mathbf{D} - \omega\mathbf{U})$ da matriz $\mathbf{M_{SSOR}}$ é triangular superior e dado por

$$\mathbf{D}^{-1}(\mathbf{D} - \omega\mathbf{U}) = \begin{pmatrix} 1 & \frac{\omega}{-2} & 0 & \frac{\omega}{-2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & \frac{\omega}{-3} & 0 & \frac{\omega}{-3} & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & \frac{\omega}{-2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & \frac{\omega}{-3} & 0 & \frac{\omega}{-3} & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \frac{\omega}{-4} & 0 & \frac{\omega}{-4} & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \frac{\omega}{-3} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & \frac{\omega}{-2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \frac{\omega}{-3} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.102}$$

o termo $(\mathbf{D} - \omega\mathbf{L})$ é triangular inferior, dado por

$$(\mathbf{D} - \omega\mathbf{L}) = \begin{pmatrix} -2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \omega & -3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \omega & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\ \omega & 0 & 0 & -3 & 0 & 0 & 0 & 0 & 0 \\ 0 & \omega & 0 & \omega & -4 & 0 & 0 & 0 & 0 \\ 0 & 0 & \omega & 0 & \omega & -3 & 0 & 0 & 0 \\ 0 & 0 & 0 & \omega & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & \omega & 0 & \omega & -3 & 0 \\ 0 & 0 & 0 & 0 & 0 & \omega & 0 & \omega & -2 \end{pmatrix} \tag{2.103}$$

Com essas configurações triangulares das matrizes $\mathbf{D}^{-1}(\mathbf{D} - \omega\mathbf{L})$ e $(\mathbf{D} - \omega\mathbf{L})$, torna-se conveniente reescrever a matriz $\mathbf{M_{SSOR}}$ da forma

$$\mathbf{M_{SSOR}} = \mathbf{WY}, \tag{2.104}$$

onde

$$\mathbf{W} = \frac{1}{\omega(2-\omega)}(\mathbf{D} - \omega\mathbf{L}) \tag{2.105}$$

e

$$\mathbf{Y} = \mathbf{D}^{-1}(\mathbf{D} - \omega\mathbf{U}). \tag{2.106}$$

Agora, voltando para a equação $\mathbf{h}_{k+1} = \mathbf{P}^{-1}\mathbf{r}_{k+1}$ do algoritmo do Gradiente Conjugado Pré-condicionado, pode-se reescrever a expressão como

$$\mathbf{P}\mathbf{h}_{k+1} = \mathbf{r}_{k+1}. \tag{2.107}$$

Nesse momento, vale ressaltar que a equação 2.107 nada mais é do que um sistema de equações lineares do tipo $\mathbf{Ax} = \mathbf{b}$, com $\mathbf{P}$ sendo a matriz dos coeficientes, $\mathbf{h}_{k+1}$ sendo o vetor incógnita e $\mathbf{r}_{k+1}$ o vetor resposta. Dessa forma, para calcular $\mathbf{h}_{k+1}$, é necessário apenas resolver o sistema da equação 2.107 e, com a configuração triangular das matrizes $\mathbf{W}$ e $\mathbf{Y}$, será visto agora como isso será simples. Substituindo $\mathbf{P}$ por $\mathbf{M_{SSOR}}$ na equação 2.107, temos que

$$\mathbf{M_{SSOR}}\mathbf{h}_{k+1} = \mathbf{r}_{k+1}. \tag{2.108}$$

Substituindo a equação 2.104 na equação 2.108, tem-se que

$$\mathbf{WY}\mathbf{h}_{k+1} = \mathbf{r}_{k+1}. \tag{2.109}$$

Reescrevendo o produto $\mathbf{Y}\mathbf{h}_{k+1}$ como sendo $\mathbf{z}$, então a equação 2.109 se torna

$$\mathbf{Wz} = \mathbf{r}_{k+1}, \tag{2.110}$$

onde

$$\mathbf{Y}\mathbf{h}_{k+1} = \mathbf{z}. \tag{2.111}$$

Dessa forma, tem-se agora dois sistemas a se resolver, o sistema da equação 2.110 e o sistema da equação 2.111. Assim, para encontrar a solução do sistema 2.109, primeiro e encontra-se a solução do sistema 2.110 para determinar o valor de $\mathbf{z}$ e então substituir a solução para $\mathbf{z}$ na equação 2.111 e determinar, finalmente, o $\mathbf{h}_{k+1}$. Essa fatoração que faz dividir o sistema da equação 2.108 em dois sistemas de equações é vantajosa porque tanto a matriz $\mathbf{W}$ quanto a $\mathbf{Y}$ são triangulares e isso torna as soluções dos sistemas atribuídos a essas matrizes muito mais fáceis de serem obtidas do que obter a solução do sistema original dado pela equação 2.108. Para demonstrar isso, considere o sistema da equação 2.110 aberta

$$\mathbf{W}\mathbf{z} = \mathbf{r}_{k+1}$$

$$\Rightarrow \alpha \begin{pmatrix} -2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \omega & -3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \omega & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\ \omega & 0 & 0 & -3 & 0 & 0 & 0 & 0 & 0 \\ 0 & \omega & 0 & \omega & -4 & 0 & 0 & 0 & 0 \\ 0 & 0 & \omega & 0 & \omega & -3 & 0 & 0 & 0 \\ 0 & 0 & 0 & \omega & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & \omega & 0 & \omega & -3 & 0 \\ 0 & 0 & 0 & 0 & 0 & \omega & 0 & \omega & -2 \end{pmatrix} \begin{pmatrix} z_{0,0} \\ z_{1,0} \\ z_{2,0} \\ z_{0,1} \\ z_{1,1} \\ z_{2,1} \\ z_{0,2} \\ z_{1,2} \\ z_{2,2} \end{pmatrix} = \begin{pmatrix} r_{0,0} \\ r_{1,0} \\ r_{2,0} \\ r_{0,1} \\ r_{1,1} \\ r_{2,1} \\ r_{0,2} \\ r_{1,2} \\ r_{2,2} \end{pmatrix}, \tag{2.112}$$

onde $\alpha = \frac{1}{\omega(2-\omega)}$.

Como a matriz $\mathbf{W}$ é triangular inferior, o primeiro termo do vetor $\mathbf{z}$ pode ser calculado prontamente e é dado por

$$z_{0,0} = \frac{r_{0,0}/\alpha}{-2}.$$

O segundo termo do vetor $\mathbf{z}$ pode ser calculado em seguida, uma vez que já temos o valor do primeiro termo, e é dado por

$$z_{1,0} = \frac{r_{1,0}/\alpha - \omega z_{0,0}}{-3}.$$

Os termos seguintes do vetor $\mathbf{z}$ então são dados por

$$z_{2,0} = \frac{r_{2,0}/\alpha - \omega z_{1,0}}{-2},$$

$$z_{0,1} = \frac{r_{0,1}/\alpha - \omega z_{0,0}}{-3},$$

$$z_{1,1} = \frac{r_{1,1}/\alpha - \omega z_{0,1} - \omega z_{1,0}}{-4}.$$

De uma forma geral, observa-se que os termos $z_{i,j}$ podem ser calculados pela fórmula de recorrência

$$z_{i,j} = \frac{r_{i,j}/\alpha - \omega l_{i,j} z_{i-1,j} - \omega d_{i,j} z_{i,j-1}}{c_{i,j}}, \tag{2.113}$$

onde os termos $l_{i,j}$, $d_{i,j}$ e $c_{i,j}$ são os coeficientes do estêncil, definidos pelas equações 2.55, 2.57 e 2.54, respectivamente. Vale a pena observar que o cálculo de cada $z_{i,j}$ envolve o termo $z$ que está à esquerda do ponto (i,j) na malha (figura 2.3), representado pelo termo $z_{i-1,j}$ e abaixo do ponto (i,j), representado pelo termo $z_{i,j-1}$, quando os mesmos existem. O coeficiente do estêncil $l_{i,j}$ vale 1 quando o ponto à esquerda de (i,j) existe e vale 0 quando não existe o ponto à esquerda do ponto (i,j). O mesmo raciocínio vale para o coeficiente $d_{i,j}$, ou seja, quando existe o ponto abaixo de (i,j), então $d_{i,j} = 1$, se não existir ponto abaixo de (i,j), então $d_{i,j} = 0$. Uma vez calculado o vetor $\mathbf{z}$ através da equação 2.113, substitui-se o resultado na equação 2.111 para se calcular o $\mathbf{h}_{k+1}$.

Abrindo então a equação 2.111, temos que

$$\mathbf{Y}\mathbf{h}_{k+1} = \mathbf{z}$$

$$\Rightarrow \begin{pmatrix} 1 & \frac{\omega}{-2} & 0 & \frac{\omega}{-2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & \frac{\omega}{-3} & 0 & \frac{\omega}{-3} & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & \frac{\omega}{-2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & \frac{\omega}{-3} & 0 & \frac{\omega}{-3} & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \frac{\omega}{-4} & 0 & \frac{\omega}{-4} & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \frac{\omega}{-3} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & \frac{\omega}{-2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \frac{\omega}{-3} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} h_{0,0} \\ h_{1,0} \\ h_{2,0} \\ h_{0,1} \\ h_{1,1} \\ h_{2,1} \\ h_{0,2} \\ h_{1,2} \\ h_{2,2} \end{pmatrix} = \begin{pmatrix} z_{0,0} \\ z_{1,0} \\ z_{2,0} \\ z_{0,1} \\ z_{1,1} \\ z_{2,1} \\ z_{0,2} \\ z_{1,2} \\ z_{2,2} \end{pmatrix}. \qquad (2.114)$$

De forma análoga ao que foi feito anteriormente, porém agora começando do último termo do vetor incógnita, temos que, pelo fato da matriz $\mathbf{Y}$ ser triangular superior, o último termo do vetor $\mathbf{h}_{k+1}$ pode ser calculado prontamente e é dado por

$$h_{2,2} = z_{2,2}.$$

Os termos acima então são dados por

$$h_{1,2} = z_{1,2} - \frac{\omega}{-3} h_{2,2},$$

$$h_{0,2} = z_{0,2} - \frac{\omega}{-2} h_{1,2},$$

$$h_{2,1} = z_{2,1} - \frac{\omega}{-3} h_{2,2},$$

$$h_{1,1} = z_{1,1} - \frac{\omega}{-4} h_{2,1} - \frac{\omega}{-4} h_{1,2}.$$

De uma forma geral, pode-se calcular o termo $h_{i,j}$ pela fórmula de recorrência

$$h_{i,j} = z_{i,j} - \frac{\omega}{c_{i,j}} r_{i,j} h_{i+1,j} - \frac{\omega}{c_{i,j}} u_{i,j} h_{i,j+1}, \qquad (2.115)$$

onde os termos $r_{i,j}$, $u_{i,j}$ e $c_{i,j}$ são os coeficientes do estêncil definidos nas equações 2.53, 2.56 e 2.54, respectivamente. De forma parecida com o que aconteceu no cálculo do vetor $\mathbf{z}$, vale a pena observar que o cálculo de cada $h_{i,j}$ envolve o termo $h$ que está à direita do ponto (i,j) na malha (figura 2.3), representado pelo termo $h_{i+1,j}$ e acima do ponto (i,j), representado pelo termo $h_{i,j+1}$, quando os mesmos existem. O coeficiente do estêncil $r_{i,j}$ vale 1 quando o ponto à direita de (i,j) existe e vale 0 quando não existe o ponto à direita do ponto (i,j). O mesmo raciocínio vale para o coeficiente $u_{i,j}$, ou seja, quando existe o ponto acima de (i,j), então $u_{i,j} = 1$, se não existir ponto acima de (i,j), então $u_{i,j} = 0$. Portanto, através da equação 2.115, é possível se calcular $\mathbf{h}_{k+1}$ utilizando apenas os coeficientes do estêncil, sem ter que calcular a inversa da matriz $\mathbf{M_{SSOR}}$, o que seria muito mais demorado, certamente.

### 2.7.6 Método do Multigrid

O sofisticado método do Multigrid se constrói em cima de dois princípios básicos, conforme explicam Wienands e Joppich [25]:

- **Princípio da suavização**: métodos iterativos simples, como o método de Jacobi e o método de Gauss-Seidel possuem uma acentuada propriedade de suavizar as oscilações de alta frequência dos erros;
- **Princípio da correção de malha grosseira**: Um termo de erro suavizado pode ser bem representado em uma malha mais grosseira, onde as operações são muito menos custosas computacionalmente do que em uma malha mais refinada.

Sendo assim, o método do Multigrid, como o próprio nome sugere, consiste em resolver o problema de encontrar a solução do sistema $\mathbf{Ax} = \mathbf{b}$ usando mais de uma malha. Na malha mais refinada, onde as operações são mais custosas, um determinado número $\nu_1$ de iterações é executado de forma a suavizar o erro, eliminando as componentes de alta frequência e gerando a solução aproximada $\mathbf{x}_k$. Depois disso, calcula-se o resíduo oriundo dessas iterações através da equação $\mathbf{r}_k = \mathbf{b} - \mathbf{Ax}_k$ e o transfere para uma malha mais grosseira, onde as operações são menos custosas do que na malha mais fina. Na malha mais grossa, usando o resíduo que foi transferido da malha mais fina, calcula-se o erro $\mathbf{e}_k$ dado pela equação

$$\mathbf{Ae}_k = \mathbf{r}_k. \tag{2.116}$$

Uma vez calculado o erro através da equação 2.116, projeta-se o erro de volta para a malha mais fina através de uma interpolação e calcula-se a nova solução para $\mathbf{x}$ usando a equação

$$\mathbf{x}_k^{novo} = \mathbf{x}_k + \mathbf{e}_k. \tag{2.117}$$

Por fim, efetua-se novamente um número $\nu_2$ de iterações na malha mais fina.

Todo esse processo pode ser repetido recursivamente e, para o cálculo do erro na equação 2.116, o processo pode ser replicado para uma malha ainda mais grosseira, conforme explica Ascher [23].

No Multigrid existe um número grande de variáveis a serem estudadas: o número de malhas a se utilizar, o procedimento de suavização em cada malha, o número de iterações em cada suavização, a forma a se definir as malhas mais grossas e vários outros detalhes [26].

Agora que as linhas gerais do método do Multigrid já estão expostas, será explicado em mais detalhes cada etapa desse método. A começar por uma melhor descrição dos que seriam as componentes de alta e baixa frequência do erro. Para explorar esse assunto, considere o seguinte sistema de equações lineares com n=64 subintervalos e n-1=63 pontos internos:

$$\begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \cdots & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{62} \\ x_{63} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix} \tag{2.118}$$

.

O motivo para utilizar o sistema da equação 2.118 está no fato de que a solução do sistema é conhecida ($\mathbf{x} = \mathbf{0}$) e, portanto, é possível calcular o erro para uma dada aproximação $\mathbf{x^{(m)}}$, que é simplesmente $-\mathbf{x^{(m)}}$. Utilizando uma aproximação inicial da forma

$$x_i^{(0)} = \text{sen}\left(\frac{ik\pi}{n}\right),\ 0 \leqslant i \leqslant n,\ 1 \leqslant k \leqslant n-1, \tag{2.119}$$

onde o índice i indica a componente do vetor $\mathbf{x}$ e o escalar k é chamado de número de onda ou frequência e indica o número de metades de ondas de seno que constituem o vetor $\mathbf{x}$ no domínio do problema. A forma da equação 2.119 é chamada de modo de Fourier. Aqui, será utilizada a notação $\mathbf{x_k}$ para denotar o vetor $\mathbf{x}$ com número de onda k. Nesse ponto é importante salientar que, quanto maior o valor de k, mais ondas completas terá o vetor $\mathbf{x_k}$ dentro do domínio, enquanto que, quanto menor for o valor de k, mais suave será o comportamento do vetor $\mathbf{x_k}$. A figura 2.4 exemplifica o que foi dito.

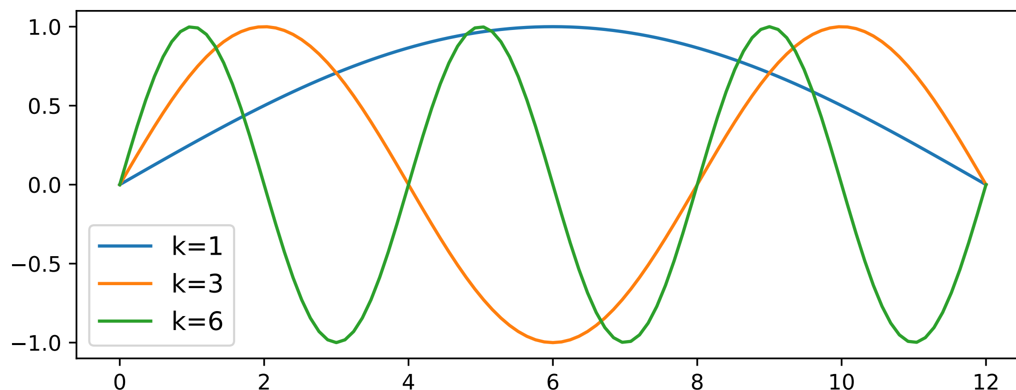



Figura 2.4: Comparação entre vários modos de Fourier com diferentes números de onda k.

Considere 3 chutes iniciais dados em modos de Fourier $\mathbf{x_k^{(0)}}$ dados por $\mathbf{x_1^{(0)}}$, $\mathbf{x_3^{(0)}}$ e $\mathbf{x_6^{(0)}}$, ou seja, com números de onda de 1, 3 e 6, respectivamente. A figura 2.5 mostra o comportamento erro em função das iterações para cada valor de k para o método de Jacobi amortecido com $\omega = 2/3$.

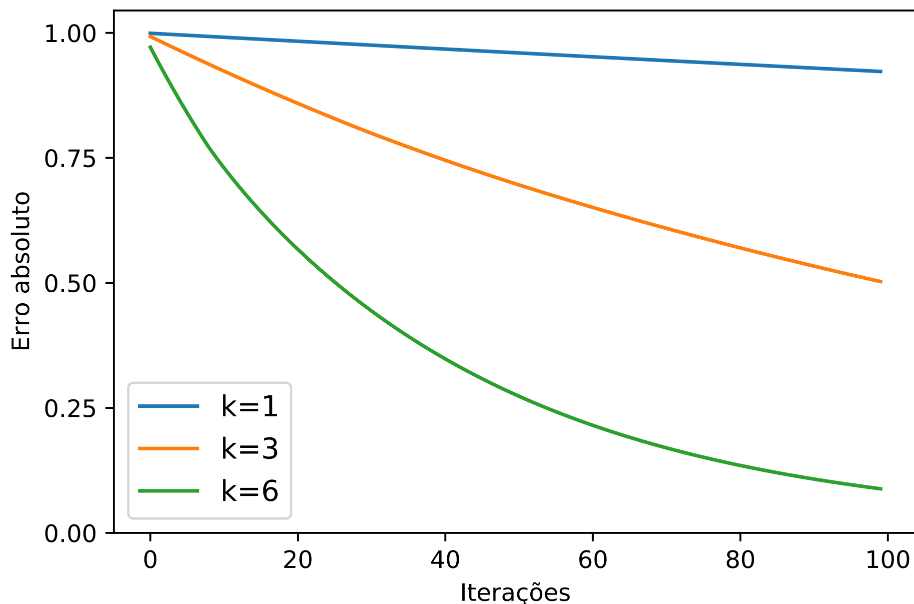



Figura 2.5: Comportamento do erro em função do número de iterações no método de Jacobi Amortecido com $\omega = 2/3$ para vários números de onda k.

Vale notar como o método citado (Jacobi Amortecido) é efetivo em diminuir rapidamente o erro para valores maiores de k, enquanto é pouco útil para diminuir os erros associados a baixas frequências. A figura 2.5 reforça o que vem sendo dito: os métodos de relaxação, ou estacionários, são muito efetivos em eliminar rapidamente as componentes de alta frequência do erro.

Na prática, o erro é constituído de várias componentes com variadas frequências, desde mais baixas até mais altas. Utilizando agora um chute inicial um pouco mais realista, constituído de 3 modos (ou frequências) dado por

$$x_i^{(0)} = \frac{1}{3}\left[\text{sen}\left(\frac{i\pi}{n}\right) + \text{sen}\left(\frac{6i\pi}{n}\right) + \text{sen}\left(\frac{32i\pi}{n}\right)\right]. \qquad (2.120)$$

A figura 2.6 mostra o comportamento do erro em função do número de iterações para o chute inicial dado pela equação 2.120. Perceba como o erro decai rapidamente nas primeiras iterações, mas desacelera rapidamente após isso. O decaimento rápido nas primeiras iterações é devido à eliminação das componentes de alta frequência do erro. Uma vez que essas componentes são praticamente zeradas, resta somente as componentes de baixa frequência.

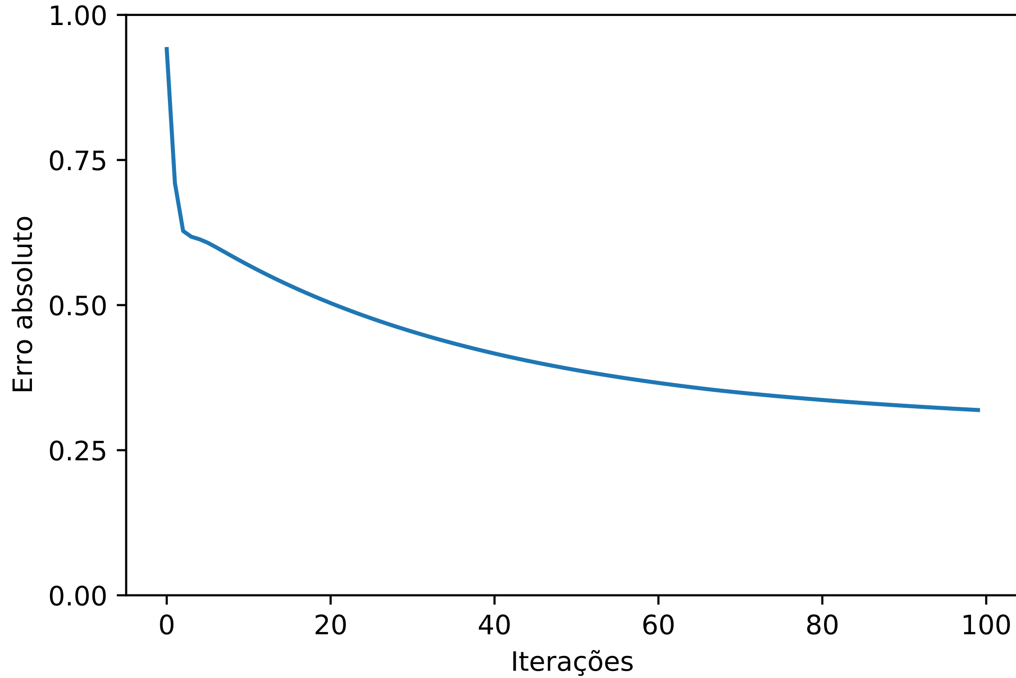



Figura 2.6: Comportamento do erro em função do número de iterações para o método de Jacobi Amortecido com $\omega = 2/3$ para um chute inicial $\mathbf{x}^{(0)} = (\mathbf{x}_1^{(0)} + \mathbf{x}_6^{(0)} + \mathbf{x}_{32}^{(0)})/3$

Como será explicado na seção 2.7.7, a velocidade de convergência de um dado método está relacionado com o raio espectral da matriz do método, e o raio espectral, por sua vez, é dado como o maior autovalor em módulo da matriz do método. Sendo assim, agora estamos interessados em encontrar os autovalores da matriz do método de Jacobi Amortecido. Recuperando que a matriz de iteração do método de Jacobi Amortecido é dada por $\mathbf{T}_\omega = (1-\omega)\mathbf{I} + \omega\mathbf{T_J}$, temos que

$$\mathbf{T}_\omega = \mathbf{I} - \frac{\omega}{2}\begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \cdots & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix}, \tag{2.121}$$

onde vê-se que o último termo da equação 2.121 é exatamente a matriz $\mathbf{A}$ da equação 2.118. Portanto, nesse caso, calcular os autovalores da matriz $\mathbf{T}_\omega$ recai em um problema de calcular os autovalores da matriz $\mathbf{A}$ da seguinte forma

$$\lambda(\mathbf{T}_\omega) = 1 - \frac{\omega}{2}\lambda(\mathbf{A}). \tag{2.122}$$

Os autovalores da matriz $\mathbf{A}$ são dados por

$$\lambda_k(\mathbf{A}) = 4\text{sen}^2\left(\frac{k\pi}{2n}\right),\ 1 \leqslant k \leqslant n-1. \tag{2.123}$$

Já os autovetores da matriz $\mathbf{A}$ são dados por

$$w_{k,i} = \text{sen}\left(\frac{ik\pi}{n}\right),\ 1 \leqslant k \leqslant n-1,\ 0 \leqslant i \leqslant n, \tag{2.124}$$

onde $w_{k,i}$ se refere à i-ésima componente do k-ésimo autovetor associado à matriz $\mathbf{A}$.

Portanto, substituindo a equação 2.123 na equação 2.122, temos que os autovalores da matriz $\mathbf{T}_\omega$ são dados por

$$\lambda_k(\mathbf{T}_\omega) = 1 - 2\omega\text{sen}^2\left(\frac{k\pi}{2n}\right),\ 1 \leqslant k \leqslant n-1. \tag{2.125}$$

Os autovetores da matriz $\mathbf{T}_\omega$ são os mesmos da matriz $\mathbf{A}$, dados pela equação 2.124. Uma vez que os autovetores de uma matriz são linearmente independentes entre si, eles formam uma base de um espaço vetorial. Dessa forma, é possível escrever o erro $\mathbf{e}^{(0)}$ como uma combinação linear dos autovetores da matriz $\mathbf{A}$, dado como

$$\mathbf{e}^{(0)} = \sum_{k=1}^{N-1} c_k \mathbf{w}_k. \tag{2.126}$$

De fato, é possível demonstrar que o erro do método em cada passo é proporcional à uma norma da matriz $\mathbf{T}_\omega$, conforme é demonstrado por Ascher [23]. Sendo assim, o erro depois de $m$ iterações é dado por

$$\mathbf{e}^{(m)} = \mathbf{T}_\omega^m \mathbf{e}^{(0)}. \tag{2.127}$$

Substituindo a equação 2.126 na equação 2.127, temos que

$$\mathbf{e}^{(m)} = \sum_{k=1}^{N-1} c_k \mathbf{T}_\omega^m \mathbf{w}_k. \tag{2.128}$$

Uma vez que as matrizes $\mathbf{A}$ e $\mathbf{T}_\omega$ possuem os mesmos autovetores, a equação 2.128 pode ser reescrita como

$$\mathbf{e}^{(m)} = \sum_{k=1}^{N-1} c_k \lambda_k^m (\mathbf{T}_\omega^m) \mathbf{w}_k. \tag{2.129}$$

O que essa manipulação nos trás de interessante é o fato de que a equação 2.129 nos mostra que, depois de m iterações, o k-ésimo modo do vetor erro é reduzido por um fator de $\lambda_k^m(\mathbf{T}_\omega)$. Aqui é importante salientar que, para que método convirja, todos os autovalores associados à matriz $\mathbf{T}$ do método são menores do que 1 em módulo, conforme estipula o Teorema 05 da seção 2.7.7. Dessa maneira, todas as componentes do erro são diminuídos a cada iteração, porém as componentes associadas aos menores autovalores são mais rapidamente reduzidas. A figura 2.7 compara a aproximação após 10 iterações com o chute inicial para diferentes chutes iniciais com diferentes modos individualmente, bem como para um chute inicial sendo uma composição de

modos. Na figura 2.7 é importante notar que, para o modo $\mathbf{x}_3$, a aproximação após 10 iterações é praticamente igual ao chute inicial, com uma redução mínima. Já no caso do chute inicial dado pelo modo $\mathbf{x}_{16}$, a aproximação após 10 iterações é notoriamente menor do que o chute inicial. Novamente fica claro aqui o quanto o erro diminui rapidamente para modos de alta frequência. O caso mais interessante está no chute inicial composto de dois modos, um de baixa frequência $\mathbf{x}_3$ e um de frequência mais alta $\mathbf{x}_{16}$. Perceba como após 10 iterações o modo de alta frequência foi consideravelmente reduzido enquanto que o modo de baixa frequência praticamente se manteve inalterado. Essa redução do modo de alta frequência mantendo praticamente inalterado o modo de baixa frequência deu um aspecto mais suavizado à solução e é a isso que se deve a propriedade de suavização dos métodos de relaxação.

Uma vez demonstrada a capacidade dos métodos de relaxação de eliminar as componentes de alta frequência do erro, vem a pergunta: como então lidar com as componentes de baixa frequência? Nesse contexto que o Multigrid entra em cena. De fato, componentes de baixa frequência do erro passam a ser componentes de alta frequência quando transferidas para malhas mais grosseiras. Portanto, utilizar malhas de diferentes graus de refinamento é uma forma de driblar a capacidade pobre de lidar com as componentes de baixa frequência dos métodos de relaxação, enquanto aproveita o melhor que esses métodos têm a oferecer: a excelente capacidade de eliminar as componentes de alta frequência do erro.

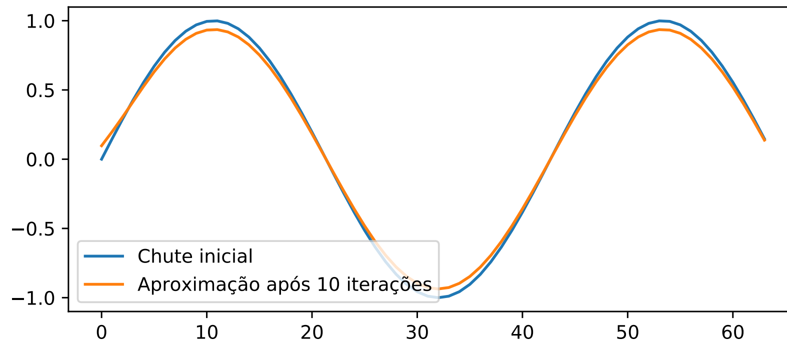
Para demonstrar o que foi dito anteriormente sobre componentes de baixa frequência do erro serem representadas como componentes de alta frequência em malhas mais grosseiras, considere o vetor $\mathbf{w}_k$, com frequência k, avaliado em uma malha $\Omega^h$ unidimensional com n=12 pontos, espaçamento h entre os pontos, dado por

$$w^h_{k,i} = \text{sen}\left(\frac{ik\pi}{n}\right). \tag{2.130}$$

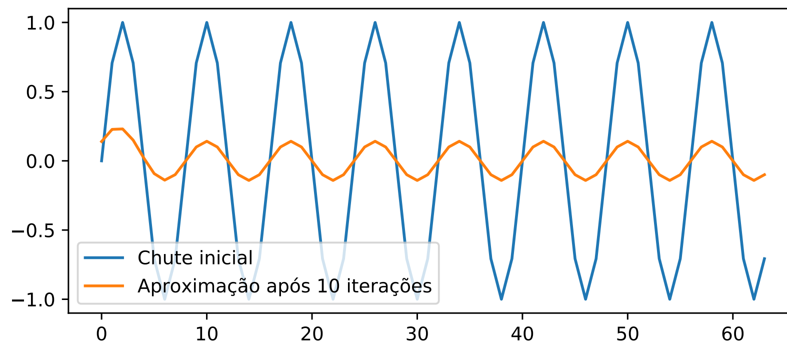
Considere agora o mesmo vetor, com frequência k, avaliado numa malha unidimensional $\Omega^{2h}$, mas agora com n=6, ou seja, uma malha mais grosseira, com apenas metade dos pontos da malha original e espaçamento 2h entre os pontos. Nessa nova malha $\Omega^{2h}$, o vetor $\mathbf{w}^{2h}_k$ é dado como

$$w^{2h}_{k,i} = \text{sen}\left(\frac{ik\pi}{n/2}\right) = \text{sen}\left(\frac{i(2k)\pi}{n}\right). \tag{2.131}$$

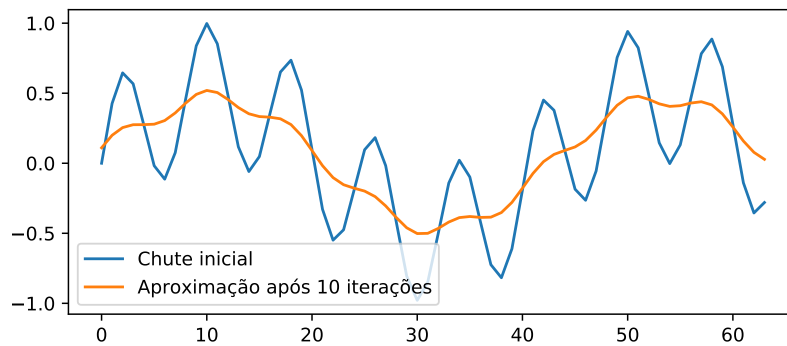
A figura 2.8 mostra, para número de onda k=4, como o mesmo modo se apresenta mais oscilatório na malha mais grosseira. Repare como, na malha mais grosseira, o modo tem dois comprimentos de onda, ou ciclos, completos em apenas 6 pontos, enquanto que na malha mais fina o mesmo modo leva 6 pontos pra completar apenas um ciclo. Dessa forma, fica evidente que, ao passar um vetor de uma malha fina para uma malha grossa, o vetor se torna mais oscilatório, o que é ótimo para os métodos de relaxação, que são mais eficazes em eliminar modos oscilatórios do erro. Sendo assim, a estratégia do método do Multigrid consiste em, depois de algumas iterações em uma determinada malha usando um método de relaxação, os modos mais oscilatórios do erro já foram eliminados e os modos mais suaves são as únicas componentes restantes do erro, então, nesse momento, troca-se para uma malha mais grosseira, onde os modos se tornam mais oscilatórios e




(a) Chute inicial dado por modo de Fourier $\mathbf{x}_3$.




(b) Chute inicial dado por modo de Fourier $\mathbf{x}_{16}$.




(c) Chute inicial dado por composição de modos Fourier $(\mathbf{x}_3 + \mathbf{x}_{16})/2$.

Figura 2.7: Comparação entre o chute inicial e a aproximação após 10 iterações no método de Jacobi Amortecido com $\omega = 2/3$ para diferentes modos de Fourier.

serão facilmente eliminados pelos métodos de relaxação. A explicação feita nesse parágrafo resume o raciocínio básico por trás do método do Multigrid.

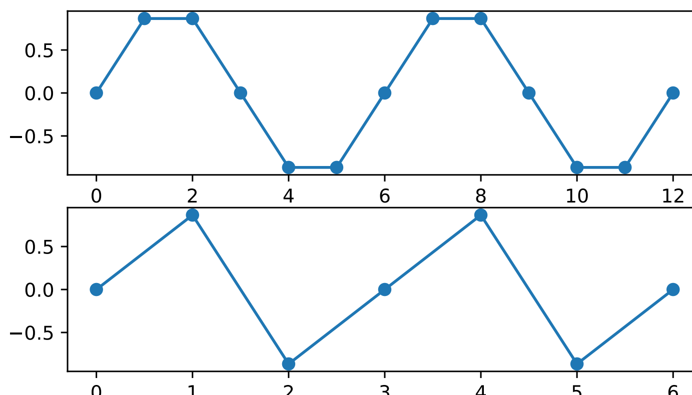



Figura 2.8: Modo de Fourier com número de onda k=4 representado em uma malha com 12 e 6 pontos, respectivamente. A malha mais grossa tem uma representação mais oscilatória do mesmo modo.

Recapitulando então o método do Multigrid, temos os seguintes passos:

- Realizar iterações no sistema $\mathbf{Ax} = \mathbf{b}$ na malha $\Omega^h$ para obter uma aproximação $\mathbf{x}^h$;
- Calcular o resíduo, dado por $\mathbf{r} = \mathbf{b} - \mathbf{Ax}^h$;
- Transferir o resíduo da malha $\Omega^h$ para a malha $\Omega^{2h}$;
- Realizar iterações no sistema $\mathbf{Ae} = \mathbf{r}$ na malha $\Omega^{2h}$ para obter uma aproximação para o erro $\mathbf{e}^{2h}$;
- Transferir o erro da malha $\Omega^{2h}$ para a malha $\Omega^h$;
- Corrigir a aproximação obtida na malha $\Omega^h$ com a estimativa para erro obtida na malha $\Omega^{2h}$ através da equação $\mathbf{x}^h \leftarrow \mathbf{x}^h + \mathbf{e}^h$.

Uma vez demonstrado o potencial de melhora dos métodos de relaxação com a transição entre malhas com diferentes níveis de refinamento, vem a pergunta: como fazer essa transição entre as malhas? Essa é uma questão fundamental do método do Multigrid e será discutida em detalhes agora. Nesse trabalho, será considerado apenas o caso em que cada malha tem intervalos entre os pontos duas vezes maiores que a malha imediatamente mais fina. Segundo Briggs [27], essa é uma prática comumente adotada porque, em geral, não há vantagens em usar outros padrões de malhas mais grossas. As duas operações usadas para fazer a transição entre as malhas serão a **interpolação** e a **restrição**. Quando um vetor é passado de uma malha mais grossa para uma mais fina, é utilizando a interpolação. Em contrapartida, quando um vetor é transferido de uma malha mais fina para uma malha mais grossa, se trata da restrição.

Vamos primeiramente considerar o caso em que o vetor $\mathbf{x}^h$ será transferido da malha mais fina $\Omega^h$ para a malha mais grossa $\Omega^{2h}$ através da interpolação. Nesse trabalho, será utilizada a forma mais simples e direta de interpolação, a interpolação linear. Em termos de notação, o operador interpolação será denotado como $\mathbf{I}_{2h}^{h}$. O operador interpolação leva vetores de uma malha grossa para uma malha fina através da regra

$$\mathbf{x}^h = \mathbf{I}_{2h}^{h}\mathbf{x}^{2h} \tag{2.132}$$

$$x_{2i,2j}^{h} = \frac{9x_{i,j}^{2h} + 3x_{i-1,j}^{2h} + 3x_{i,j-1}^{2h} + x_{i-1,j-1}^{2h}}{16} \tag{2.133}$$

$$x_{2i,2j+1}^{h} = \frac{9x_{i,j}^{2h} + 3x_{i-1,j}^{2h} + 3x_{i,j+1}^{2h} + x_{i-1,j+1}^{2h}}{16} \tag{2.134}$$

$$x_{2i+1,2j}^{h} = \frac{9x_{i,j}^{2h} + 3x_{i,j-1}^{2h} + 3x_{i+1,j}^{2h} + x_{i+1,j-1}^{2h}}{16} \tag{2.135}$$

$$x_{2i+1,2j+1}^{h} = \frac{9x_{i,j}^{2h} + 3x_{i,j+1}^{2h} + 3x_{i+1,j}^{2h} + x_{i+1,j+1}^{2h}}{16} \tag{2.136}$$

Nesse ponto, é importante salientar que o processo de interpolação será bom sempre que o vetor a ser interpolado for suave, ou seja, com apenas componentes de baixa frequência. A figura 2.9 ilustra como a interpolação é feita na malha defasada.

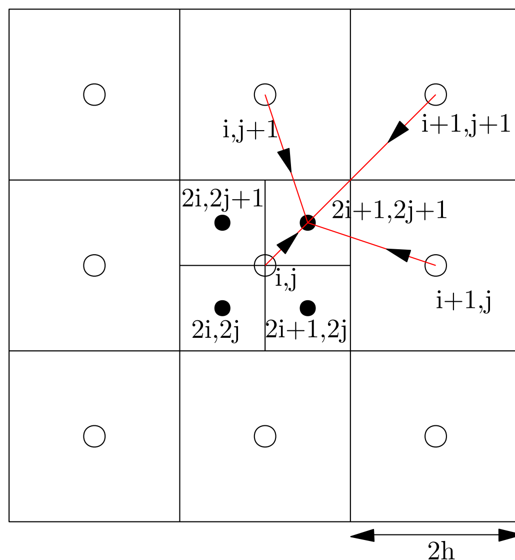



Figura 2.9: Operação de interpolação da malha $\Omega^{2h}$ para malha $\Omega^h$.

Partindo agora para a operação responsável por levar um vetor de uma malha mais fina para uma malha mais grossa, estamos falando do operador restrição, denotado por $\mathbf{I}_h^{2h}$. Assim como na interpolação, a restrição feita de maneira linear também apresenta bons resultados [28].

Nesse trabalho, a restrição será feita através da regra

$$\mathbf{x}^{2h} = \mathbf{I}_h^{2h}\mathbf{x}^h \tag{2.137}$$

ou

$$x_{i,j}^{2h} = \frac{x_{2i+1,2j+1}^h + x_{2i,2j+1}^h + x_{2i,2j}^h + x_{2i+1,2j}^h}{4}. \quad (2.138)$$

A figura 2.10 ilustra como a restrição é feita na malha defasada.

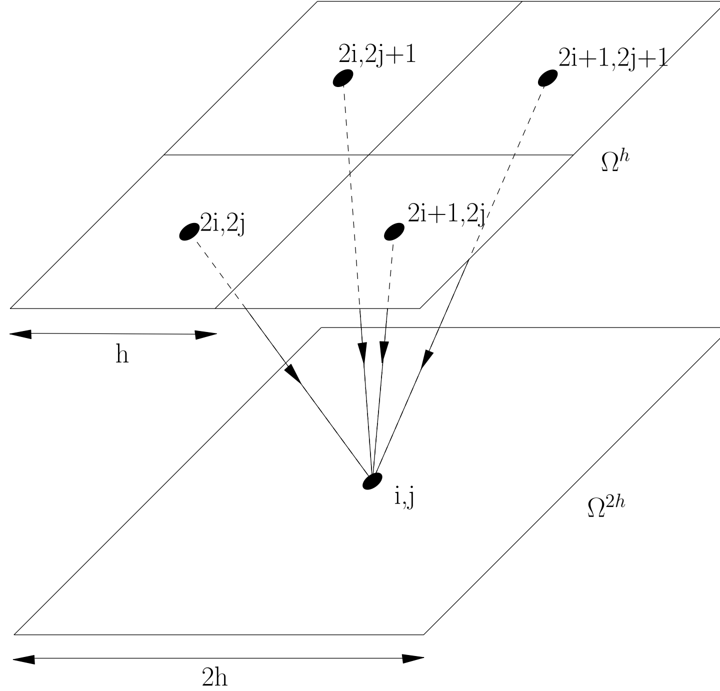



Figura 2.10: Operação de restrição da malha $\Omega^h$ para malha $\Omega^{2h}$.

Agora que já temos uma forma bem definida de transferir vetores entre as diversas malhas, podemos retornar ao método do Multigrid. Para tornar mais fácil a descrição dos algoritimos, algumas mudanças na notação serão necessárias. A partir daqui, na equação do resíduo ($\mathbf{Ae} = \mathbf{r}$), o termo do lado direito da equação será chamado apenas de $\mathbf{b}$ ao invés de $\mathbf{r}$. Além disso, o termo $\mathbf{e}$ será chamado de $\mathbf{x}$, por se tratar apenas de um vetor solução. Essas mudanças facilitam a descrição dos algorítimos do Multigrid, mas é necessário se ter sempre em mente o significado de cada termo. Veremos primeiramente o chamado esquema V-Ciclo, definido pelo algorítimo a seguir.

**Esquema V-Ciclo**

$$\mathbf{x}^h \leftarrow V^h(\mathbf{x}^h, \mathbf{b}^h).$$

1. Realizar $\nu_1$ iterações no sistema $\mathbf{A}^h\mathbf{x}^h = \mathbf{b}^h$ com chute inicial $\mathbf{x}^h$;

2. Se $\Omega^h$ for a malha mais grossa, ir para o passo 4
   Caso contrário:

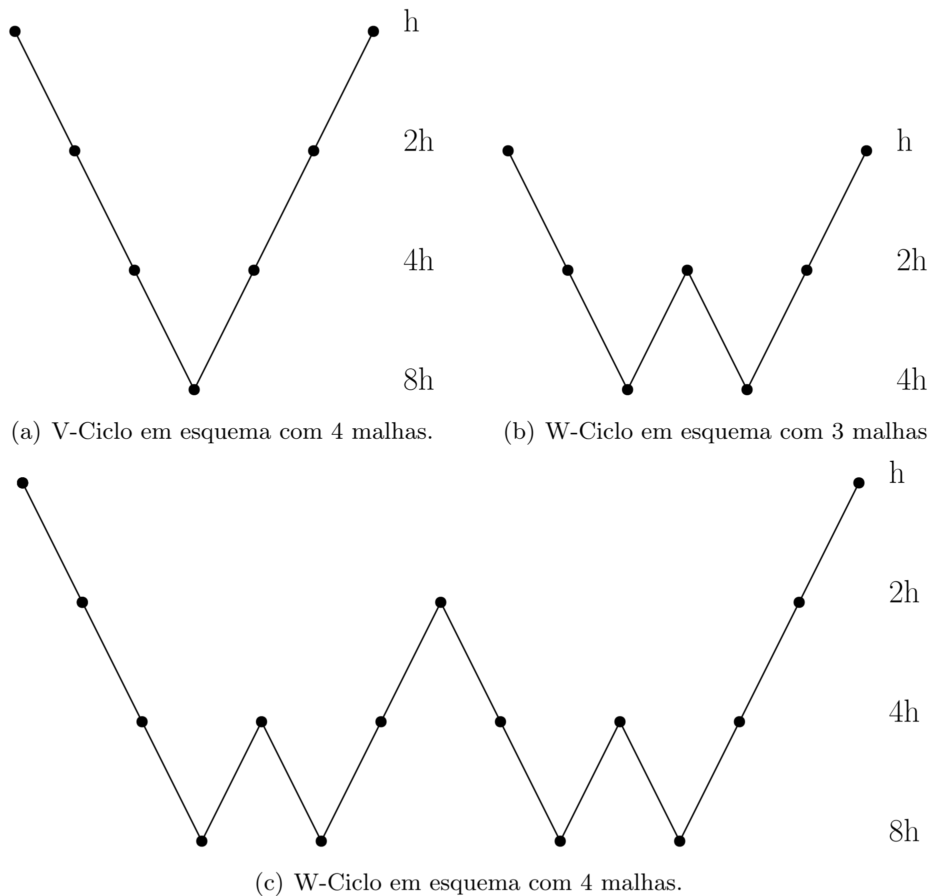
   - Calcular $\mathbf{b}^{2h} = \mathbf{I}_h^{2h}\mathbf{r}^h$;

- Fazer $\mathbf{x}^{2h} \leftarrow \mathbf{0}$;
- Executar $\mathbf{x}^{2h} \leftarrow V^{2h}(\mathbf{x}^{2h}, \mathbf{b}^{2h})$.

3. Fazer a correção $\mathbf{x}^h \leftarrow \mathbf{x}^h + \mathbf{I}_{2h}^h \mathbf{x}^{2h}$;

4. Realizar $\nu_2$ iterações no sistema $\mathbf{A}^h \mathbf{x}^h = \mathbf{b}^h$.

De uma forma qualitativa, o esquema V-Ciclo realiza $\nu_1$ iterações na malha $\Omega^h$ e então desce para uma malha menos refinada $\Omega^{2h}$, onde realiza novamente $\nu_1$ iterações e desce para uma malha menos refinada $\Omega^{4h}$ e assim sucessivamente até atingir a malha mais grossa para só então voltar a subir para malhas mais finas realizando $\nu_2$ iterações em cada malha mais grossa antes de subir para a mais fina. Isso nos leva ao esquema mostrado na figura 2.11 (a). O esquema mostrado na figura 2.11 (a) justifica o nome V-Ciclo. Alternativamente, pode-se implementar o Multigrid fazendo descidas e subidas entre as malhas de formas diversas.




(a) V-Ciclo em esquema com 4 malhas. (b) W-Ciclo em esquema com 3 malhas

(c) W-Ciclo em esquema com 4 malhas.

Figura 2.11: Diferentes possibilidades de transitar entre as malhas no método do Multigrid.

O V-Ciclo é, na verdade, apenas um único tipo de ciclo de toda uma família de ciclos que o multigrid pode oferecer. A forma mais geral possível é o chamado $\mu$-Ciclo e é definida como se segue:

**Esquema $\mu$-Ciclo**

$$\mathbf{x}^h \leftarrow M\mu^h(\mathbf{x}^h, \mathbf{b}^h).$$

1. Realizar $\nu_1$ iterações no sistema $\mathbf{A}^h\mathbf{x}^h = \mathbf{b}^h$ com chute inicial $\mathbf{x}^h$;
2. Se $\Omega^h$ for a malha mais grossa, ir para o passo 4
   Caso contrário:
   - Calcular $\mathbf{b}^{2h} = \mathbf{I}_h^{2h}\mathbf{r}^h$;
   - Fazer $\mathbf{x}^{2h} \leftarrow \mathbf{0}$;
   - Executar $\mathbf{x}^{2h} \leftarrow M\mu^{2h}(\mathbf{x}^{2h}, \mathbf{b}^{2h})$ $\mu$ vezes.
3. Fazer a correção $\mathbf{x}^h \leftarrow \mathbf{x}^h + \mathbf{I}_{2h}^h\mathbf{x}^{2h}$;
4. Realizar $\nu_2$ iterações no sistema $\mathbf{A}^h\mathbf{x}^h = \mathbf{b}^h$.

De fato, o V-Ciclo nada mais é do que um $\mu$-Ciclo com $\mu$=1. Quando $\mu = 2$, temos o chamado W-Ciclo. Segundo Saad [29], os esquemas $\mu = 1$ e $\mu = 2$ são os mais usados na prática e raramente $\mu = 3$ é utilizado.

Um outro esquema de Multigrid é o chamado Full-Multigrid. Nesse esquema, a ideia central é usar malhas mais grosseiras para gerar uma chute inicial para uma malha mais refinada. Uma vez que os métodos estacionários são fortemente dependentes de bons chutes iniciais [27], o esquema de Full Multigrid é uma estratégia interessante. O algorítmo do Full-Multigrid é dado como

**Full Multigrid em V**

$$\mathbf{x}^h \leftarrow FMG^h(\mathbf{b}^h).$$

1. Se $\Omega^h$ for a malha mais grossa, determine $\mathbf{x} \leftarrow \mathbf{0}$ e vá para o passo 2;
   Caso contrário:
   - Calcular $\mathbf{b}^{2h} \leftarrow \mathbf{I}_h^{2h}\mathbf{b}^h$;
   - Fazer $\mathbf{x}^{2h} \leftarrow FMG(\mathbf{f}^{2h})$.
2. Fazer $\mathbf{x}^h \leftarrow \mathbf{I}_{2h}^h\mathbf{x}^{2h}$
3. Fazer $\mathbf{x}^h \leftarrow V^h(\mathbf{x}^h, \mathbf{b}^h)$ $\mu_0$ vezes.

### 2.7.7 Critérios de Convergência

E possível saber a priori se um determinado método iterativo irá convergir antes mesmo de aplicar o método de maneira propriamente dita [30]. Para cada um dos métodos, existem algumas condições suficientes pra garantir a convergência, mas em geral não são necessários, ou seja, se o critério se aplica, então o método converge, mas se ele não se aplica, o método pode convergir ou não. A seguir serão apresentados alguns critérios suficientes de convergência para os vários métodos

iterativos e, ao final, será apresentado um critério necessário e suficiente que vale para todos os métodos ditos estacionários, que são os métodos de Jacobi, Gauss-Seidel e SOR. Os critérios suficientes são apresentados antes pois em geral é muito mais fácil de verificar tais critérios. O último critério, que é geral, suficiente e necessário, é muito mais complicado de ser testado por necessitar do cálculo de auto-valores da matriz de iteração $\mathbf{T}$ de cada método, que será definida a seguir.

#### 2.7.7.1 Critério Das Linhas

O critério das linhas é um critério que, se atendido, garante a convergência para o método de Jacobi e Gauss-Seidel [22]. O teorema 01 estabelece o Critério das Linhas para convergência dos métodos de Jacobi e Gauss-Seidel.

Teorema 01: Seja o sistema de equações lineares $\mathbf{Ax} = \mathbf{b}$ de ordem n, então os métodos de Jacobi e Gauss-Seidel convergem se

$$|a_{ii}| > \sum_{j=1}^{N} |a_{ij}|, \text{ com } j \neq i \ \ \forall i = 1, ..., n \tag{2.139}$$

O que a equação 2.139 nos diz é que, caso o módulo do elemento da diagonal de cada linha da matriz $\mathbf{A}$ seja maior do que a soma do módulo de todos os outros termos da mesma linha, então o método de Jacobi e de Gauss-Seidel convergem.

Vale ressaltar que o critério das linhas para os métodos de Jacobi e Gauss-Seidel é um critério suficiente mas não necessário para a convergência dos métodos citados.

#### 2.7.7.2 Critério de Sassenfeld

O critério de Sassenfeld, caso atendido, garante a convergência do método de Gauss-Seidel [22]. O teorema 02 estabelece o Critério de Sassenfeld.

Teorema 02: Seja o sistema de equações lineares dado por $\mathbf{Ax} = \mathbf{b}$, então o método de Gauss-Seidel converge se

$$\beta_i = \frac{1}{|a_{ii}|} \left( \sum_{j=1}^{i-1} |a_{ij}| \beta_j + \sum_{j=i+1}^{N} |a_{ij}| \right) < 1 \ \forall i = 1, ..., n. \tag{2.140}$$

Novamente, vale ressaltar que o critério de Sassenfeld é um critério suficiente mas não necessário para convergência do método de Gauss-Seidel.

#### 2.7.7.3 Critério de Ostrowski-Reich

O critério de Ostrowski-Reich estabelece uma condição suficiente mas não necessária para a convergência do método de SOR [31]. O teorema 03 estabelece o Critério de Ostrowski-Reich.

Teorema 03: Seja o sistema de equações lineares dado por $\mathbf{Ax} = \mathbf{b}$, então o método de SOR converge $\forall\, \omega \in [1, 2[$ se a matriz $\mathbf{A}$ for estritamente diagonal dominante, isto é, se a equação 2.139 é satisfeita.

#### 2.7.7.4 Critério de convergência para o método do Gradiente Conjugado

O método do gradiente conjugado tem sua convergência garantida quando a matriz $\mathbf{A}$ é simétrica e definida positiva [23]. O teorema 04 estabelece esse critério, que é suficiente, mas não necessário. Para convergência do método do Gradiente Conjugado.

Teorema 04: Seja o sistema de equações lineares dado por $\mathbf{Ax} = \mathbf{b}$, então se a matriz $\mathbf{A}$ é definida positiva, isto é, se

$$\mathbf{x^T A x} > 0\ \forall \mathbf{x} \neq 0, \tag{2.141}$$

então o método do Gradiente Conjugado converge.

#### 2.7.7.5 Condição suficiente e necessária para convergência de métodos iterativos estacionários

Todos os critérios de convergência mencionados nos parágrafos anteriores são condições suficientes mas não necessárias para a convergência dos diversos métodos. Sendo assim, existe a possibilidade de que um método convirja para um determinado sistema do tipo $\mathbf{Ax} = \mathbf{b}$ mesmo que as condições citadas não sejam satisfeitas. Para resolver isso, o critério estabelecido no teorema 5 estipula uma condição necessária e suficiente para a convergência do método estacionário iterativo, seja ele qual for [23].

Teorema 05: Seja o sistema de equações lineares dado por $\mathbf{Ax} = \mathbf{b}$, definindo a matriz de iteração do método iterativo como sendo

$$\mathbf{T} = \mathbf{I} - \mathbf{M^{-1}A}, \tag{2.142}$$

então o método converge se, e somente se

$$\rho(\mathbf{T}) < 1, \tag{2.143}$$

onde o $\rho(\mathbf{T})$ é o raio espectral da matriz $\mathbf{T}$, definido como

$$\rho(\mathbf{B}) = max\{|\lambda_i|; \text{onde } \lambda_i \text{ são os auto-valores associados à matriz } \mathbf{T}.\} \tag{2.144}$$

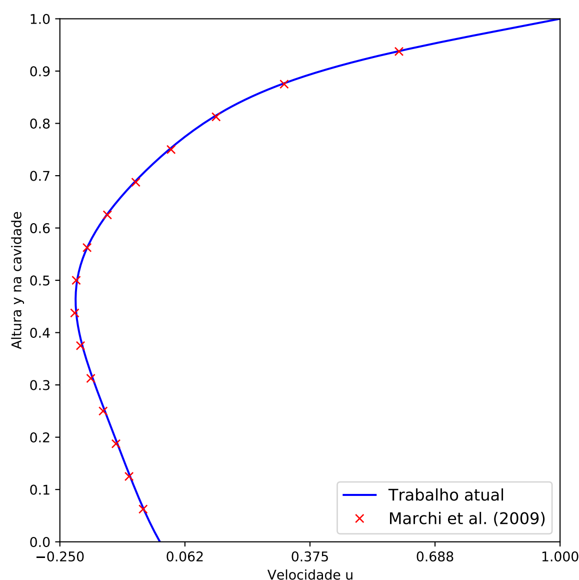
A matriz $\mathbf{M}$ na equação 2.142 depende do método iterativo e é única para cada método. Na seção específica de cada método, a matriz $\mathbf{M}$ é definida.

# Capítulo 3

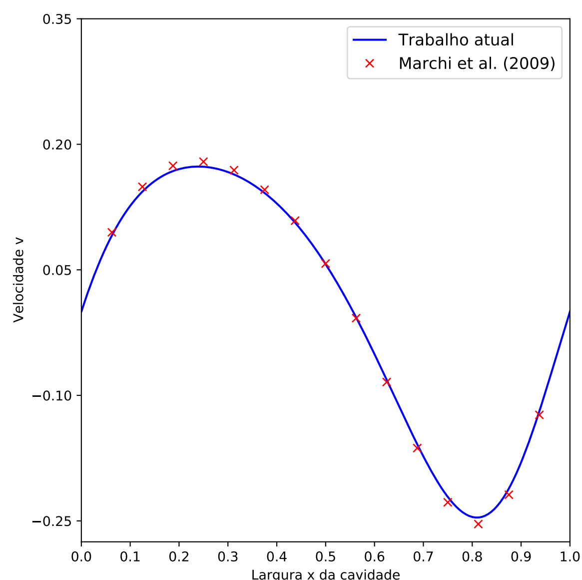
# Resultados

## 3.1 Validação

Para validar os resultados desse trabalho, os resultados obtidos para o campo de velocidade para Re=100, $\Delta t$=0,001, N=155 e em regime permanente ($t_f$=7,639) serão comparados com os resultados de Marchi et al. [1]. Para se chegar no regime permanente, a solução foi evoluída no tempo até que a diferença relativa de velocidade entre dois passos de tempo consecutivos em um ponto no centro na malha fosse menor que $10^{-6}$. A figura 3.1 compara os resultados do presente trabalho com os resultados obtidos por Marchi et al. [1]. Conforme observa-se, os resultados são bastante coerentes.




(a) Velocidade u medida em x=0,5.




(b) Velocidade v medida em y=0,5.

Figura 3.1: Comparação dos resultados desse trabalho com os resultados de Marchi et al. [1] para validação.

A figura 3.2 mostra as linhas de corrente na cavidade em regime permanente.

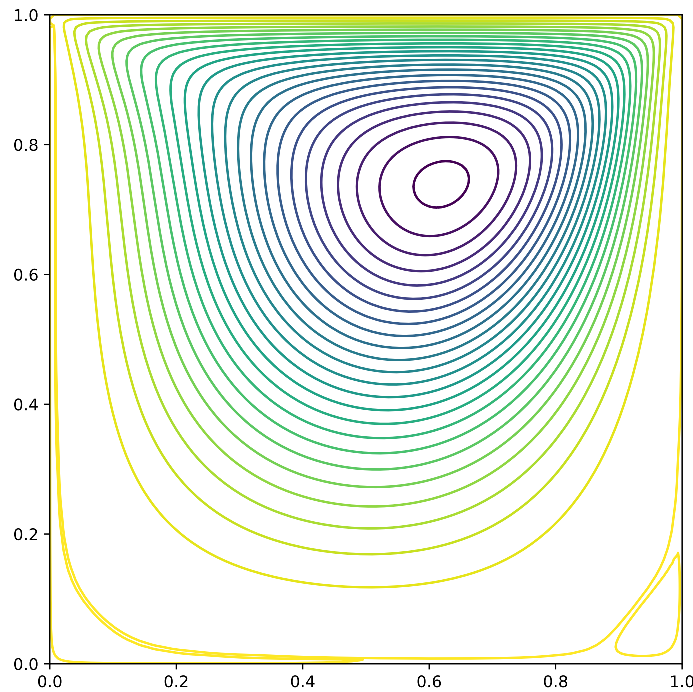



Figura 3.2: Linhas de corrente para Re=100.

## 3.2 A importância de se implementar uma otimização para o Python

Como já foi discutido no capítulo 1, o Python de maneira pura é uma linguagem extremamente lenta quando comparada com outras linguagens como Fortran e C++. Para contornar esse problema e tornar a linguagem mais rápida e sem perder a sua facilidade de implementação, o presente trabalho faz a utilização da biblioteca Numba. Para se ter uma noção do quanto o Numba é capaz de acelerar o Python, a tabela 3.1 compara os resultados de tempo de execução de alguns métodos para uma malha de $25 \times 25$ para os métodos de SOR, Gauss-Seidel e Jacobi Amortecido[1] com o Numba e sem o Numba. Todos os resultados apresentados nesse trabalho, incluindo os mostrados na tabela 3.1, com exceção dos resultados apresentados para Linux na tabela 3.2, foram extraídos de uma máquina equipada com processador portátil AMD Ryzen™ 5 3500U com placa de vídeo integrada Radeon™ Vega 8, com 8 GB de memória RAM em Dual Channel operando a 2400 MHz.

Tabela 3.1: Resultados para N=25, Re=100, $t_f$=0,1, $\epsilon$=$10^{-6}$ e $\Delta$ t=0,001.

| | | **Tempo de execução [s]** | |
|---|---|---|---|
| **Método** | **Omega** | **Com Numba** | **Sem Numba** |
| **SOR** | 1,85 | 2,4172 | 26,8388 |
| **Gauss-Seidel** | 1 | 2,5262 | 130,802 |
| **Jacobi Amortecido** | 0,95 | 2,6994 | 229,078 |

Pela tabela 3.1, vê-se que, para o método de Jacobi Amortecido, houve uma redução de 98,82%

[1]O método de Jacobi Amortecido é explicado na seção 3.4.2.

no tempo de execução! Essa é só uma demonstração do poder absurdo de acelerar códigos que o Numba possui. Depois que foram implementados com o Numba, os métodos passaram a ter velocidade de execução parecidas, de aproximadamente 2,5 s, conforme se vê na tabela 3.1. Isso ocorre porque, o Numba primeiro compila as funções para só depois rodar o código e essa compilação leva algum tempo, então o tempo de aproximadamente 2,5 s que observamos para o tempo de execução com Numba para os diversos métodos na tabela 3.1 é praticamente somente o tempo que o Numba leva para compilar a função. Para códigos com N maiores, a diferença entre o tempo de execução para os diferentes métodos aplicados com Numba tende a aumentar, já que os 2 segundos que a biblioteca leva para compilar as funções começa a perder importância relativa. A implementação do Numba no código é feita de maneira extremamente simples, porém alguns cuidados são necessários para que tudo funcione. Para implementar a aceleração do código pelo Numba, é necessário escrever as partes do código que demandam mais tempo para serem executadas dentro de uma função. Uma vez que a parte mais custosa do código está escrita dentro de uma função, basta usar o decorador *@njit* imediatamente antes da declaração da função para a mágica acontecer. Alguns cuidados devem ser tomados como sempre usar variáveis do mesmo tipo em um vetor dentro da função, sempre declarar o tipo das variáveis antes de usá-las e também declarar matrizes somente fora das funções e usá-las como argumento de entrada para função. Daqui em diante, nesse trabalho, todos os códigos serão executados com o Numba e, portanto, os tempos de execução vistos daqui em diante serão sempre para códigos com a biblioteca implementada.

## 3.3 FORTRAN vs Python+Numba

Uma vez demonstrada a capacidade de acelerar os códigos em Python que a biblioteca Numba possui, o próximo passo natural é comparar então a velocidade de execução dos códigos obtidos em Python+Numba com outras linguagens de programação consideradas mais rápidas para esse tipo de propósito. Nesse trabalho, a linguagem que será comparada em termos de tempo de execução será o FORTRAN, uma vez que essa é uma opção comumente utilizada no meio acadêmico de simulação de comportamentos de fluidos através de métodos numéricos. Para a comparação, foram construídos dois códigos equivalentes, um em FORTRAN e outro em Phython+Numba para resolver o problema da cavidade no método de SOR usando um método de projeção de primeira ordem, com tempo final $t = 1,0$, $Re = 1000$ e $\Delta t = 0,001$. Os resultados obtidos estão expostos na tabela 3.2, que mostra os tempos obtidos tanto em sistema operacional Windows quanto Linux. No caso do Windows, o Python+Numba se mostrou ligeiramente mais rápido do que o FORTRAN. Os resultados exibidos na tabela 3.2 para Windows e Linux foram obtidos em diferentes máquinas, com diferentes configurações e, portanto, não podem ser comparados diretamente.

Tabela 3.2: Resultados para N=100, Re=1000, $t_f$=1.0, $\epsilon$=$10^{-7}$ e $\Delta$ t=0,001.

| | **Tempo de execução [s]** | |
|---|---|---|
| **Linguagem** | **Windows** | **Linux** |
| **Python+Numba** | 22,38 | 15,98 |
| **FORTRAN** | 23,44 | 13,17 |

A situação, entretanto, se inverte no caso do Linux, onde o FORTRAN leva a vantagem e é aproximadamente 18% mais rápido do que a dupla Python+Numba. Mesmo sendo mais lento que o FORTRAN no Linux, o código executado em Python se aproxima bastante do seu concorrente graças à otimização feita na linguagem com o Numba. Aqui é importante ressaltar que os resultados apresentados na tabela 3.2 para o FORTRAN foram utilizando a opção Ofast de compilação, que torna a execução do código mais rápida. Apesar disso, os resultados ainda são passíveis de otimização, em especial para o FORTRAN.

## 3.4 A instabilidade do método de Jacobi

Para o sistema da forma original como foi concebido na seção 2.4, o método de Jacobi não foi capaz de apresentar estabilidade suficiente para fazer a solução aproximada convergir e ficar dentro da tolerância do erro. O erro diminui até chegar a um valor mínimo e então para de diminuir, quando deveria tender a 0. Apesar de não convergir para abaixo da tolerância definida, o método de Jacobi apresentou uma solução que não divergiu. Isso é bastante curioso e ocorre pelo fato do método de Jacobi apresentar oscilações no resíduo. De fato, depois de um determinado número de iterações, o resíduo a cada iteração oscila com amplitude sempre igual e em torno do zero, e, por causa disso, a resposta não converge, mas também não diverge, uma vez que o resíduo de cada passo subtrai da solução exatamente o que o resíduo do passo anterior adicionou. Uma vez que o erro é definido como o módulo do maior termo do vetor diferença entre duas iterações, então ele fica sempre igual depois de um certo número de iterações. Para contornar esse problema de instabilidade do método de Jacobi, duas opções são possíveis: podemos modificar o sistema original de tal maneira que tenhamos um sistema que convirja para a solução exata dentro da tolerância definida, ou podemos aplicar um fator de sub relaxação ou de amortecimento $\omega$ ao método de Jacobi original, obtendo o método de Jacobi Amortecido. A seção 3.4.1 e a seção 3.4.2 descrevem e implementam cada uma dessas duas abordagens possíveis para implementação do método de Jacobi.

### 3.4.1 Modificando sistema original

Para a primeira opção, uma forma de modificar o sistema original é definindo o valor da pressão em um ponto do sistema e calculando para o resto dos pontos. Isso é possível porque, para o nosso problema, o valor da pressão, de fato, não é o mais importante, e sim o gradiente de pressão dentro da cavidade, ou seja, mais importante do que o valor da pressão em cada ponto é a forma como a pressão varia dentro do domínio. Sendo assim, podemos definir um valor de pressão arbitrário para um ponto e todos os outros pontos devem ser calculados de maneira a satisfazer a equação 2.24. Sendo assim, definindo que a pressão no ponto (0,0) seja igual a zero, a matriz do sistema, para N=3, passa a ser dada por

$$
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -3 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & -2 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & -3 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & -3 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & -2 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & -3 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -2
\end{pmatrix}
\begin{pmatrix}
P_{0,0} \\ P_{1,0} \\ P_{2,0} \\ P_{0,1} \\ P_{1,1} \\ P_{2,1} \\ P_{0,2} \\ P_{1,2} \\ P_{2,2}
\end{pmatrix}
=
\begin{pmatrix}
b_{0,0} \\ b_{1,0} \\ b_{2,0} \\ b_{0,1} \\ b_{1,1} \\ b_{2,1} \\ b_{0,2} \\ b_{1,2} \\ b_{2,2}
\end{pmatrix},
\tag{3.1}
$$

com $b_{0,0} = 0$. Essa simples modificação é capaz de agora fazer o sistema convergir para dentro da tolerância adotada. Apesar disso, a convergência ainda apresenta instabilidades na forma de oscilação do resíduo a cada iteração. Em geral, os métodos tendem a diminuir o número de iterações necessárias a cada passo de tempo uma vez que, para cada passo, o chute inicial para resposta do sistema é o resultado do próprio sistema para o passo de tempo anterior. A figura 3.3 mostra essa tendência da redução do número de iterações em função do passo de tempo para o método de SOR e Gauss-Seidel. Apesar disso, para o método de Jacobi aplicado para o sistema modificado da equação 3.1, o resíduo apresenta oscilações fortes ao longo das iterações e isso faz com que o número de iterações necessárias para convergência em cada passo de tempo tende a oscilar bastante também. A figura 3.4 mostra o número de iterações para os 100 primeiros passos de tempo para o método de Jacobi.

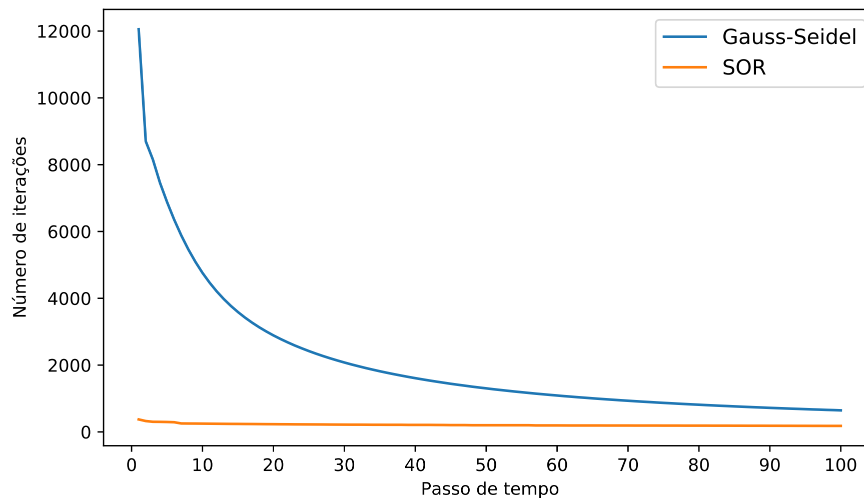



Figura 3.3: Comportamento do número de iterações para os 100 primeiros passos de tempo para os métodos de SOR e Gauss-Seidel.

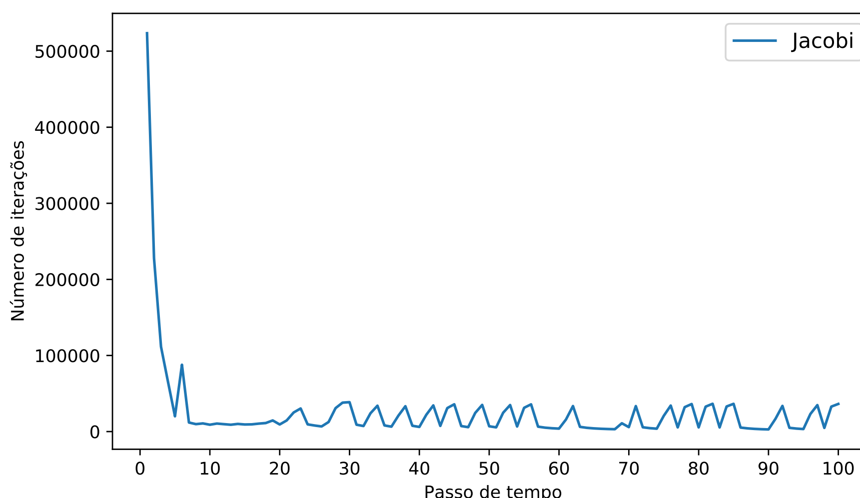



Figura 3.4: Comportamento do número de iterações para os 100 primeiros passos de tempo para o método de Jacobi.

Apesar da modificação do sistema original permitir a convergência do sistema, essas oscilações do resíduo tornam o método bastante demorado. Uma forma de contornar esse problema é resolvendo o sistema original com o método de Jacobi Amortecido. A descrição e implementação desse método são feitas na seção seguinte.

### 3.4.2 Método de Jacobi Amortecido

Uma forma de diminuir a oscilação do resíduo para o método de Jacobi para o sistema original, sem a modificação explicada na seção 3.4.1, é multiplicar o termo R na equação 2.77 por um fator de sub relaxação ou de amortecimento $\omega$, parecido com o que é feito no método de SOR (equação 2.85), nos levando ao método de Jacobi Amortecido, conforme explica Ascher [23]. A diferença é que, no método de Jacobi Amortecido, o fator $\omega$ é tal que $0 < \omega \leqslant 1$, por isso é chamado de fator de **sub** relaxação, enquanto que, no método de SOR, $\omega$ é tal que $1 \leqslant \omega < 2$, por isso se chama fator de **sobre** relaxação. A aplicação desse fator $\omega$ no método de Jacobi Amortecido é capaz de reduzir drasticamente as oscilações do resíduo e permitir que o sistema convirja mesmo para o sistema original. Assim como ocorre no método de SOR, o fator $\omega$ não é conhecido a priori para o método de Jacobi Amortecido. Dessa forma, deve-se fazer um estudo do impacto desse fator no número de iterações necessárias para fazer o sistema convergir de maneira a se determinar o $\omega$ ótimo que minimiza o número de iterações. A figura 3.5 mostra a curva de iterações em função de $\omega$ para o primeiro passo de tempo para N=100. Pela figura 3.5, é possível determinar que o $\omega$ ótimo é $\omega = 0,999$ e em $\omega = 1$ existe uma descontinuidade no gráfico, onde se tem, na verdade, o método de Jacobi original, sem amortecimento e para o sistema sem modificação, que não converge, e o número de iterações tende ao infinito.

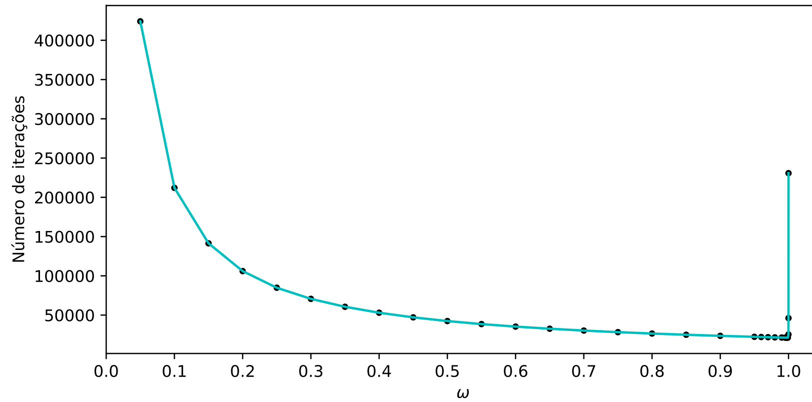



Figura 3.5: Gráfico para determinar o $\omega$ ótimo para o método de Jacobi Amortecido para N=100.

A figura 3.6 mostra o resultado da aplicação do fator $\omega$ no método. Pela comparação da figura 3.6 com a figura 3.4, vê-se que aplicação do fator $\omega$ não somente reduz a oscilação, como também reduz dramaticamente o número de iterações para convergência necessário em cada passo de tempo, tornando o método muito mais rápido. Na seção 3.6, a tabela 3.3 compara diversos métodos, inclusive o método de Jacobi e Jacobi Amortecido, onde vê-se que, enquanto o método de Jacobi levou 1 h e 16 min e 2,55 milhões de iterações para chegar no tempo final $t_f = 0,1$, o método de Jacobi Amortecido levou 58,13 s e 317 mil iterações!

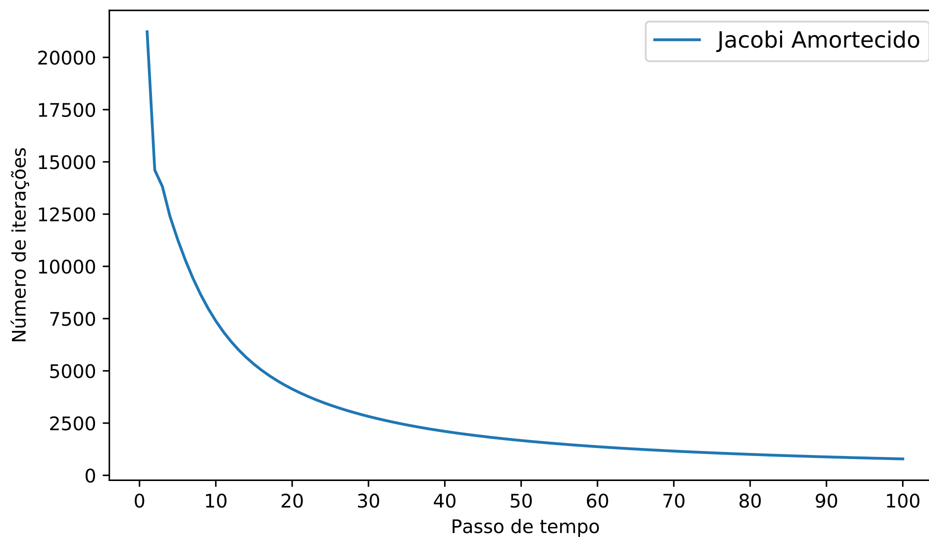



Figura 3.6: Comportamento do número de iterações para os 100 primeiros passos de tempo.

### 3.4.3 O motivo da instabilidade do método de Jacobi

Para o método de Jacobi aplicado ao sistema original, calculando o raio espectral $\rho(\mathbf{T})$ da matriz de iteração do método, vê-se que $\rho(\mathbf{T}) = 1$. Pelo teorema 05, o método converge se $\rho(\mathbf{T}) < 1$. De fato, é possível demonstrar que o erro do método em cada passo é proporcional à

uma norma da matriz **T**, definida na seção 2.7.7, conforme é demonstrado por Ascher [23]. O raio espectral da matriz é um limitante inferior para qualquer norma de uma matriz. Ou seja, o raio espectral é igual à menor norma possível de uma matriz. Dessa forma, se o erro a cada passo é multiplicado por um fator maior que 1, então o erro cresce a cada passo e o método diverge. Mas se o erro é multiplicado por fator menor que 1, então o erro a cada iteração diminui e o método converge. E se o erro for multiplicado por um fator igual a 1? Isso mesmo, nesse caso, observamos exatamente o mesmo comportamento que vemos no resíduo do método de Jacobi aplicado para o sistema original, o erro não converge e nem diverge, mas tende a estabilizar em um valor.

Uma vez esclarecidas as particularidades da convergência do método de Jacobi, partiremos agora para a comparação entre os vários métodos.

## 3.5 Métodos de Multigrid

O método do Multigrid oferece muita liberdade para quem o implementa. Existem inúmeras variáveis e decisões a serem tomadas na hora de codificá-lo, como o método que será aplicado junto com o Multigrid, o número de iterações em cada malha, o esquema em V ou W, entre outras decisões. Dessa maneira, essa seção se dedica a fazer uma comparação geral de várias implementações diferentes do Multigrid com os respectivos métodos singlegrids associados. A figura 3.7 compara os tempos de execução obtido para os métodos de SOR, Gradiente Conjugado (GC), Gradiente Conjugado Precondicionado (GCP) e métodos de Multigrid associados. Foi implementado o Multigrid com o SOR em um esquema em V com 2 malhas (MG-SOR-2M) e 3 malhas (MG-SOR-3M). Nesses casos, os valores de $\nu_1$ e $\nu_2$ foram de 3. Comparando o tempo obtido com os métodos Multigrid com o SOR e o método SOR em si, vê-se que não houve melhora real. Uma vez que os métodos mais óbvios a serem aplicados com o Multigrid, que são os métodos estacionários, não apresentaram melhoras em relação aos métodos originais, testou-se também o Multigrid em conjunto com o método do Gradiente Conjugado. Ainda na figura 3.7 podemos comparar o método de Multigrid com o Gradiente Conjugado em esquema em V para 2 malhas (MG-GC-2M) e 3 malhas (MG-GC-3M). Testou-se também o método de Full-Multigrid com o Gradiente Conjugado em um esquema de 2 malhas (FMG-GC). Percebe-se que o método MG-GC-2M foi o único que apresentou uma ligeira melhora em relação ao método original. Por fim, testou-se também o Multigrid com o Gradiente Conjugado Precondicionado com um esquema em V com 2 malhas (MG-GCP-2M). Assim como no caso do SOR, o GCP não apresentou melhoras quando implementado com o Multigrid. Como o método MG-GC-2M foi o único método Multigrid que apresentou melhoras em relação ao método original, os resultados apresentados nas seções seguintes conterão apenas esse método de Multigrid.

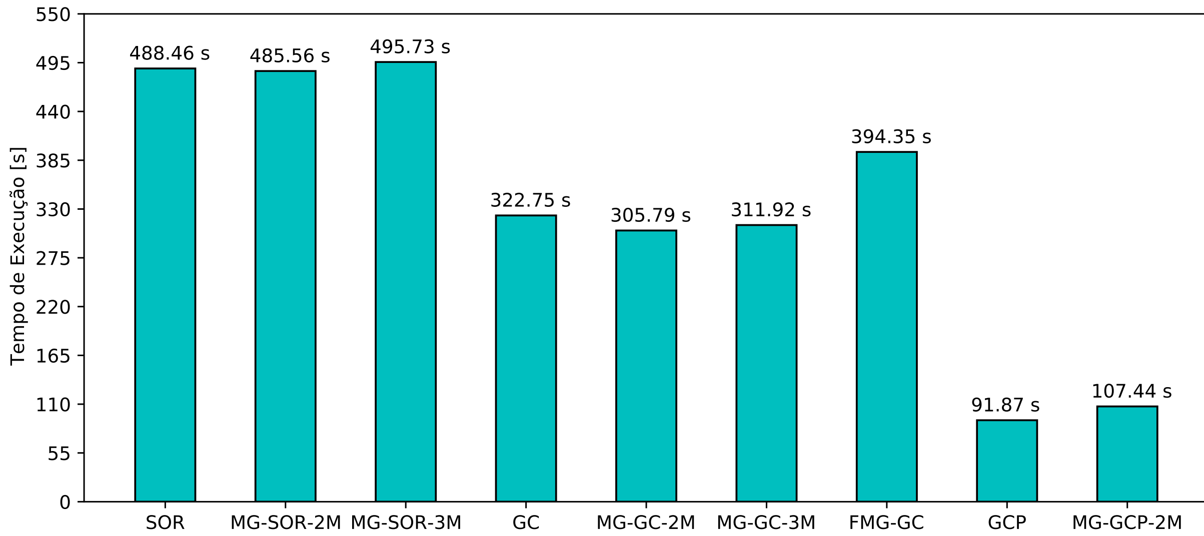



Figura 3.7: Comparação de diversos métodos Multigrid e Singlegrid em termos de tempo de execução para $N = 512$, $Re = 3000$, $\Delta t = 0,0005$, $t_f = 0,05$ e $\epsilon = 10^{-6}$.

## 3.6 Comparações entre diversos métodos

Para se estudar a diferenças entre os diferentes métodos, serão feitas análises em sistemas com diferentes números de discretização N para se avaliar como cada método se comporta em relação aos outros em função do tamanho do sistema de equações lineares a ser resolvido. A seção 3.6.1 discute em detalhes os resultados para $N = 128$ e $N = 512$, enquanto que a seção 3.6.2 mostra e discute os resultados para $N = 128$, $N = 256$, $N = 512$ e $N = 1024$.

### 3.6.1 Resultados

A tabela 3.3 compara o número total de iterações, o número de iterações somente no primeiro passo de tempo e o tempo de execução total de vários métodos para $Re = 3000$, $N = 128$, $\epsilon$=$10^{-6}$, $t_f = 0,05$ e $\Delta t = 0,0005$. Nessa tabela, o método GCP-SSOR é o método do Gradiente Conjugado Pré-Condicionado pela matriz $\mathbf{M_{SSOR}}$, definida pela equação 2.97. É importante salientar que os tempos de execução exibidos na tabela 3.3 não incluem o tempo de compilação da função pelo Numba e que esses tempos apresentam variações de alguns segundos a cada vez que se executa os códigos. Dessa forma, os dados mostrados na tabela 3.3 mostram o menor tempo de compilação obtido para cada método após executar cada código algumas vezes.

Tabela 3.3: Resultados para N=128, Re=3000, $t_f$=0,05, $\epsilon$=$10^{-6}$ e $\Delta$t=0,0005.

| **Método** | **Nº total de iterações** | **Nº de iterações no 1º passo de tempo** | **Tempo de execução [s]** |
|---|---|---|---|
| **Jacobi Amortecido** | 2590748 | 63357 | 434,91 |
| **Gauss-Seidel** | 1268964 | 31711 | 242,49 |
| **SOR** | 38971 | 535 | 10,77 |
| **GC** | 30315 | 385 | 7,21 |
| **GCP-SSOR** | 4885 | 77 | 7,45 |
| **MG-GC-2M** | 30329 | 386 | 20,67 |

Um resultado interessante que pode ser retirado da tabela 3.3 é que o método GCP-SSOR foi feliz na escolha da matriz condicionadora **P**, uma vez que se verifica que há melhora significativa no número de iterações para se resolver o problema no método GCP-SSOR em relação ao método GC original, seja no primeiro passo ou no total de iterações. De fato, a estratégia de pré-condicionamento do método do Gradiente Conjugado só faz sentido se for aplicado uma matriz pré-condicionadora que diminua o número de condição da matriz do sistema, conforme foi explicado na seção 2.7.5, ou seja, somente se $\kappa(\mathbf{P^{-1}A}) \ll \kappa(\mathbf{A})$. Para o nosso caso específico, não é possível calcular o número de condição $\kappa$ da matriz **A** pois essa matriz possui determinante nulo e, portanto, não pode ser invertida. Dessa forma, o número de condição $\kappa(\mathbf{A})$ não está definido. Pelo fato do determinante da matriz do sistema ser nulo, isso implica que o sistema ou não possui solução ou possui infinitas soluções. É sabido que o sistema possui solução, portanto trata-se de um caso com infinitas soluções. Isso ocorre porque, ao resolver o sistema, estamos procurando uma solução para o gradiente de pressão e não para o valor da pressão em si em cada ponto. Dessa forma, existem infinitos de valores de pressão possíveis para um dado ponto do sistema de forma a satisfazer o gradiente de pressão correto. Uma forma de contornar esse problema é arbitrando um valor de pressão para um ponto da malha, como foi feito na seção 2.76, o que nos leva a um sistema de equações diferente, mas com solução única.

Pela tabela 3.3, é possível perceber que o ganho de um método em relação a outro em número de iterações para convergência não se replica necessariamente no tempo de execução dos códigos. Para exemplificar isso, compara-se o número total de iterações para o método de GC e GCP-SSOR vistos na tabela 3.3. O método de GC teve 30315 iterações para resolver o problema enquanto que o método GCP-SSOR precisou executar apenas 4885 iterações, ou seja, o método de GC executou quase $6,2\times$ mais iterações do que o método GCP-SSOR, porém quando compara-se os tempos de execução, os métodos tiveram praticamente o mesmo tempo de execução. Isso ocorre porque, a cada iteração, os diferentes métodos exigem que diferentes operações sejam feitas e alguns métodos precisam executar mais operações a cada iteração. Dessa forma, métodos que convergem com menos iterações costumam exigir mais operações por iteração do que outros métodos, tornando cada iteração mais lenta e isso explica porque métodos que convergem com menos iterações não necessariamente são mais rápidos do que métodos que precisam de mais iterações para convergir. Já o método do Multigrid se mostra muito mais lento do que o GC, mesmo tendo um número de iterações compatível com esse método. Isso ocorre porque, para o rodar o Multigrid, muitas

funções são necessárias e esse tempo de chamar todas as funções se torna muito significativo para sistemas pequenos. Para malhas maiores, esse custo relativo começa a desaparecer e então o método MG-GC-2M se torna mais rápido do que o GC, conforme será visto na figura 3.12.

Nos resultados da tabela 3.3, para o método de SOR, foi utilizado o $\omega$ ótimo para $N = 128$. Para determinar o $\omega$ ótimo, foi feito um estudo do número de iterações necessárias para convergência em função do $\omega$ para $N = 128$ para o primeiro passo de tempo do método SOR. A figura 3.8 mostra os resultados obtidos, que determina $\omega$=1,97 como o $\omega$ ótimo.

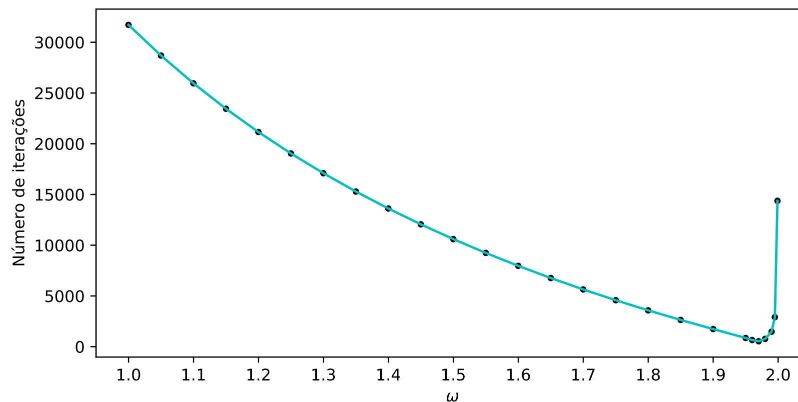



Figura 3.8: Gráfico para determinar $\omega$ ótimo para o método de SOR para $N = 128$.

Conforme foi visto na seção 2.7, para matrizes oriundas da discretização do operador laplaciano, que é o caso desse trabalho, existe uma equação que descreve o $\omega$ ótimo em função do raio espectral $\rho$ da matriz de iteração do método de Jacobi dada por

$$\omega = \frac{2}{1 + \sqrt{1 - \rho^2}}. \tag{3.2}$$

Porém, nesse trabalho, não é possível aplicar tal equação pois, conforme foi visto na seção 3.4, o raio espectral $\rho$ da matriz de iteração do método de Jacobi é 1, o que nos leva a um valor de $\omega = 2$, o que está fora dos valores aplicáveis de $\omega$, que vai de 1 a 2 não inclusos 1 e 2. Portanto, o estudo feito na figura 3.8 se torna necessário.

Assim como foi feito para o método de SOR, o método de GCP-SSOR também foi executado utilizando-se um valor ótimo de $\omega$, que, por sua vez, foi determinado também fazendo uma análise do número de iterações necessário para convergência no primeiro passo de tempo do método de GCP-SSOR para vários valores de $\omega$. A figura 3.9 mostra os resultados obtidos e determina $\omega = 1,85$ como ótimo. Para o método de Jacobi Amortecido, o valor de $\omega$ ótimo foi determinado pela figura 3.5, na seção 3.4.2.

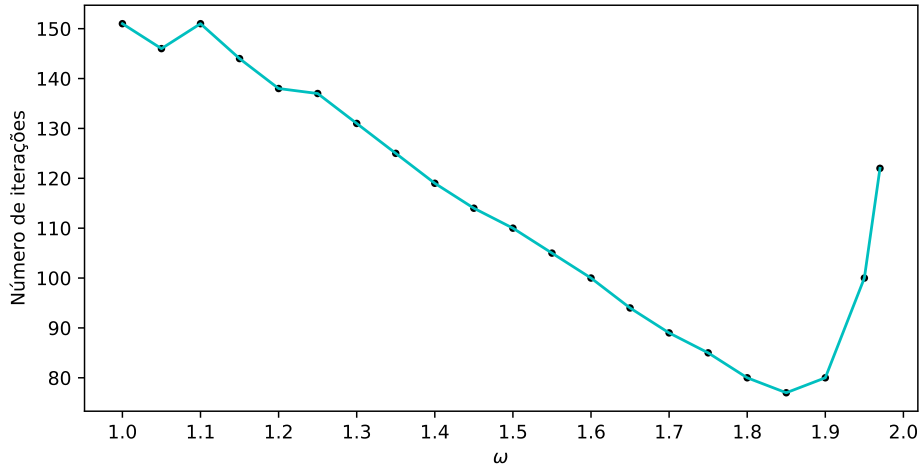



Figura 3.9: Gráfico para determinar $\omega$ ótimo para o método de GCP-SSOR para $N = 128$.

A figura 3.10 compara os erros em função de cada iteração para os métodos de SOR, GC, MG-GC-2M e GCP-SSOR para $N = 1024$, onde fica evidente o quanto o método GCP é melhor que os demais em reduzir rapidamente o resíduo relativo em poucas iterações. A figura 3.11 compara os tempos de execução desses métodos também para $N = 1024$. Mais uma vez fica claro em como o método SOR fica bem atrás dos métodos de Gradiente Conjugado, enquanto que o Multigrid MG-GC-2M fica próximo do GC.

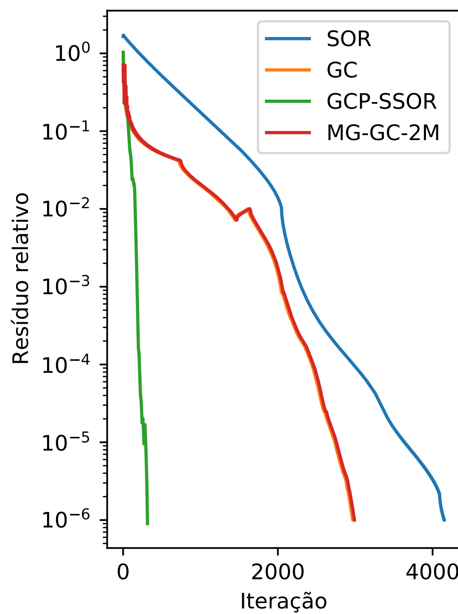



Figura 3.10: Comparação entre os tempos de execução dos diversos métodos para N=1024, Re=3000, $t_f = 0,05$, $\epsilon = 10^{-6}$ e $= 0,0005$.

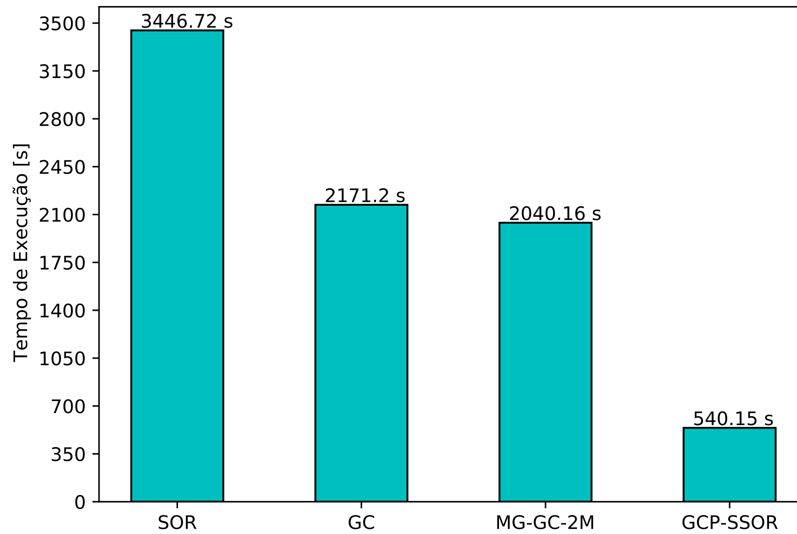



Figura 3.11: Comparação entre os tempos de execução dos diversos métodos para N=1024, Re=3000, $t_f$=0,05, $\epsilon$=$10^{-6}$ e $\Delta$t=0,0005.

Estudos foram feitos para malhas mais refinadas, com $N = 256$, $N = 512$ e $N = 1024$ e os resultados em termos de tempo de execução são mostrados na seção 3.6.2.

### 3.6.2 Análise final acerca dos resultados

Os tempos de execução dos códigos de diferentes métodos para diferentes tamanhos de sistema podem ser vistos na figura 3.12, onde os métodos de Jacobi, Jacobi Amortecido e Gauss-Seidel foram omitidos por apresentarem tempos de convergência muito maiores que os demais métodos. Para sistemas pequenos, o método de SOR se mostra uma boa opção visto que é de simples implementação e o tempo de execução é comparável ao de métodos mais refinados, como o GC. Para sistemas grandes, o método de SOR possui um agravante: o cálculo do fator $\omega$ ótimo. De fato, nesse trabalho, para se calcular o $\omega$ de qualquer método que tenha esse fator, é necessário rodar o código para o primeiro passo de tempo do problema para vários valores de $\omega$. O problema é que o primeiro passo de tempo de qualquer método é normalmente o mais demorado de se calcular, uma vez que o chute inicial para o campo de pressão é a pressão nula em todos os pontos e, nos passos seguintes, o chute inicial é o próprio campo de pressão obtido no passo de tempo anterior, que está mais próximo da solução exata do que um campo todo nulo. Outro agravante está no fato do quão sensível o método de SOR é ao fator $\omega$. Conforme pode ser visto na figura 3.8, o número de iterações necessárias para valores mais baixos de $\omega$ no método SOR é extremamente alto, fazendo com que se torne bastante demorado calcular essas iterações para cada um dos valores de $\omega$ investigados no método de SOR. Para o método de GCP-SSOR, o número de iterações não varia de forma tão dramática em função do fator $\omega$, fazendo com que seja muito mais rápido calcular o $\omega$ ótimo do método do GCP-SSOR do que do método SOR. Na execução desse trabalho, o cálculo do $\omega$ ótimo levou várias horas para o método do SOR para as malhas mais refinadas, enquanto que o método do GCP-SSOR levou apenas alguns minutos até para as malhas mais refinadas. A figura 3.12

compara os tempos de execução para os métodos mais rápidos tratados até aqui e evidencia o que já vem sendo discutido: o método de SOR fica bem atrás dos demais métodos, o GC fica no meio termo e o método de MG-GC-2M se sai ligeiramente melhor que o GC, enquanto que o GCP-SSOR se destaca dos demais métodos, sendo um mais rápido na execução de seus códigos. Em relação ao Multigrid, constata-se, pelos resultados, que o método, da maneira como foi implementada neste trabalho, não é vantajoso pois apresenta uma complexidade de implementação muito maior que o GC, por exemplo, e não melhora muito os resultados obtidos. Mesmo no melhor dos cenários, que foi o caso do MG-GC-2M, o método é somente um pouco mais rápido que o método original do GC, o que não justifica a complexidade adicional de se implementar o Multigrid, uma vez que o GCP-SSOR é muito mais rápido e simples de implementar.

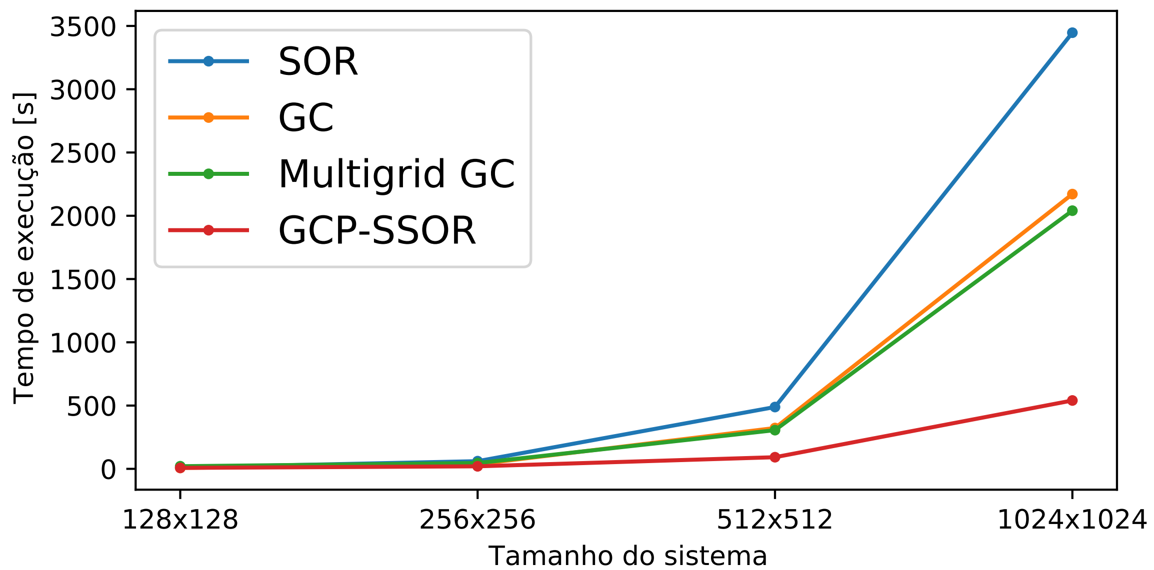



Figura 3.12: Comparação de tempo de execução entre os métodos de SOR, GC e GCP-SSOR para malhas de diversos tamanhos.

# Capítulo 4

# Conclusões

Dentro do campo da mecânica dos fluidos, frequentemente é necessário resolver sistemas de equações lineares com dimensões demasiadamente grandes, o que é um problema porque o tempo de execução dos algoritmos que executam os métodos tende a crescer muito. Nesse contexto, o presente trabalho estudou diversos métodos iterativos de solução de equações lineares que não somente convirjam com menos iterações, mas também sejam mais rápidos de serem executados computacionalmente através do Python, que é uma linguagem de programação relativamente fácil de aprender e bastante difundida na indústria e na academia, mas é considerada lenta quando comparada com outras linguagens mais comumente utilizadas para esse propósito. Através da implementação da biblioteca Numba no Python, foi possível, de maneira fácil, acelerar significativamente o tempo de execução dos códigos, tornando a linguagem comparável a linguagens mais rápidas disponíveis para resolver esse tipo de problema, como o Fortran e C++, com a vantagem de ser muito mais fácil de aprender e implementar. Vimos que, para o caso do FORTRAN executado no Windows, o Python com o Numba se mostrou equiparável, o que foi uma grata surpresa.

Em relação aos métodos iterativos, os resultados encontrados nesse trabalho até aqui mostram que, para malhas não muito refinadas, os métodos, apesar de apresentarem uma diferença considerável no número de iterações para convergência, não são muito diferentes no tempo de execução, com exceção do método de Jacobi para o sistema modificado, que se mostrou muito pior do que todos os outros métodos em todos os aspectos. Sendo assim, para sistemas relativamente pequenos, não faz muita diferença o uso de qualquer um dos métodos, com exceção do método de Jacobi para o sistema modificado. Esse cenário tende a mudar para malhas mais refinadas, onde o método do GCP-SSOR se destaca e apresenta convergência com um tempo menor de execução do que todos os demais métodos. Em malhas mais grosseiras, o método de SOR apresenta um tempo de execução parecido com o tempo de execução do método GC, apesar de precisar de mais iterações para convergência. Dessa forma, para malhas mais grossas, conclui-se que é preferível implementar o método de SOR ao método do Gradiente Conjugado por apresentar um tempo de execução comparável e ser mais fácil de implementar. Isso não é válido, porém, para malhas muito mais refinadas, onde o cálculo do fator $\omega$ do método SOR se mostra muito custoso de ser calculado, além do método em si ser mais lento que o GC. O método de GCP-SSOR é o método mais rápido estudado nesse trabalho, ficando na frente até do Multigrid. Outra vantagem conta

para o método GCP-SSOR: para malhas mais refinadas e o cálculo do fator $\omega$ para esse método se mostrou bastante rápido de ser feito mesmo para malhas muito mais refinadas. Para o Multigrid, nas formas como foi implementado nesse trabalho, conclui-se que não há qualquer vantagem em usar o método para resolver o problema da cavidade. Isso porque o Multigrid apresentou uma maior complexidade de implementação e, mesmo no melhor dos casos, não melhorou muito o método original ao qual foi aplicado, que foi o Gradiente Conjugado para esquema em V em duas malhas. Dessa forma, o método do GCP-SSOR é muito mais simples de implementar e apresenta resultados muitos melhores do que aqueles apresentados pelo MG-GC-M2 em termos de tempo de computação.

O autor desse trabalho reconhece que é necessário testar novas formas de implementar o Multigrid, em diferentes esquemas e em mais níveis de malha, de forma a se extrair o máximo de performance desse método, que sabidamente tem muito potencial, o que certamente teria sido feito nesse trabalho se houvesse mais tempo hábil para tal. De qualquer forma, o estudo detalhado dos diversos métodos e a implementação dos mesmos feitos ao longo desse Projeto de Graduação certamente servirá de degrau para que novos estudos sejam feitos partindo de onde esse trabalho parou.

## Trabalhos futuros

Esse trabalho e todos os resultados aqui apresentados apenas começam a explorar os estudos acerca dos assuntos tratados. Muitos estudos ainda são necessários pra melhor compreender formas mais rápidas e menos custosas computacionalmente de se resolver sistemas de equações lineares. Como principais sugestões de estudos futos, podem ser citados

- Uso de GPUs para execução dos códigos;
- Paralelização dos códigos;
- Implementação de diferentes esquemas Multigrid, com mais níveis de malhas e para sistemas maiores que $1024 \times 1024$;
- Testar métodos do subespaço de Krylov, como o método do Resíduo Mínimo Generalizado (GMRES), e o método do Resíduo Mínimo (MINRES);
- Testar o método Gradiente Biconjugado Estabilizado (BCGSTA);
- Avaliação do consumo de memória de cada método;
- Comparação entre o Python+Numba com diferentes linguagens além do FORTRAN.

# REFERÊNCIAS BIBLIOGRÁFICAS


[1] MARCHI, R. S. e. L. K. A. C. H. The lid-driven square cavity flow: Numerical solution with a 1024 x 1024 grid. Journal of the Brazilian Society of Mechanical Sciences and Engineering, 2009.

[2] SRIVASTAVA PREDRAG S. STANIMIROVIć, V. N. K. S.; GUPTA, D. K. A family of iterative methods with accelerated convergence for restricted linear system of equations. *Mediterranean Journal of Mathematics*, n. 222, 2017.

[3] EZZATI, S. K. R.; YOUSEFZADEH, A. Solving fully fuzzy linear system of equations in general form. *Journal of Fuzzy Set Valued Analysis*, n. 222, 2012.

[4] KENWAY CHARLES A. MADER, P. H. G. K.; MARTINS, J. R. Effective adjoint approaches for computational fluid dynamics. *Progress in Aerospace Sciences*, 2019.

[5] LYE, S. M. K. O.; RAY, D. Deep learning observables in computational fluid dynamics. *Journal of Computational Physics*, 2020.

[6] KUIPOU, A. M. W. D.; KENGNE, E. Cellular transport through nonlinear mechanical waves in fibrous and absorbing biological tissues. *Chaos, Solitons  Fractals*, 2021.

[7] WU NGHIA TRUONG, C. Y. K.; HOETZLEIN, R. Fast fluid simulations with sparse volumes on the gpu. John Wiley  Sons Ltd, 2018.

[8] STAM, J. Real-time fluid dynamics for games. Proceedings of the Game Developer Conference, 2003.

[9] JúNIOR, I. J. de A. M. Masther thesis: On the simulation of fluids for computer graphics. Instituto Nacional de Matemática Pura e Aplicada, 2007.

[10] RAMíREZ-RAMíREZ E. FLORES-OLMEDO, G. B. E. S. F.; MéNDEZ-SáNCHEZ, R. A. Emulating tightly bound electrons in crystalline solids using mechanical waves. *Scientific Reports*, 2020.

[11] NASUTION HERLINA JUSUF, E. R. H.; HUSEIN, I. Model of spread of infectious disease. *Systematic Reviews in Pharmacy*, 2020.

[12] IVORRA M.R. FERRáNDEZ, M. V.-P. B.; RAMOSA, A. Mathematical modeling of the spread of the coronavirus disease 2019 (covid-19) taking into account the undetected infections. the case of china. *Communications in Nonlinear Science and Numerical Simulation*, 2020.

[13] HOFFMAN, J. D. *Numerical Methods for Engineers*. New York: Marcel Dekker, 2001.

[14] GUINANCIO, V.; ABICALIL. Projeto de graduação: Modelagem numérica de escoamentos multifásicos tridimensionais. Departamento de Engenharia Mecânica da Faculdade de Tecnologia da Universidade de Brasília, p. 84, 2018.

[15] CHORIN, A. J. Numerical solution of the Navier-Stokes equations. *Math Comput*, n. 22, p. 745–762, 1968.

[16] WHITE, F. M. *Mecânica dos Fluidos*. sexta edição. New York: Mc Graw Hill, 2011.

[17] HINCH, E. J. *Think Before You Compute: A Prelude to Computational Fluid Dynamics*. Cambridge, United Kingdom: Cambridge Texts in Applied Mathematics, 2020.

[18] TEMAM, R. Sur l'approximation de la solution des equations de Navier-Stokes par la methode des pas fractionnaries ii. *Math Comput*, n. 33, p. 377–385, 1969.

[19] BROWN, R. C. D. L.; MINION, M. L. Accurate projection methods for the incompressible Navier-Stokes equations. *Journal of Computational Physics*, n. 168, p. 464–499, 2001.

[20] BELL, P. C. J. B.; GLAZ, H. M. A second-order projection method for the incompressible Navier-Stokes equations. *Journal of Computational Physics*, n. 85, p. 257–283, 1989.

[21] FORTUNA, A. de O. *Técnicas Computacionais para Dinâmica dos Fluidos*. São Paulo: Edusp, 2020.

[22] RUGGIERO, M. A. G.; LOPES, V. L. D. R. *Cálculo Numérico. Aspectos Teóricos e Computacionais*. segunda edição. São Paulo: Pearson, 2000.

[23] ASCHER, U. M.; GREIF, C. *A First Course in Numerical Methods*. Vancouver, British Columbia: Society for Industrial and Applied Mathematics, 2011.

[24] SHEWCHUK, J. R. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. sexta edição. Pittsburgh, Pennsylvania: Carnegie-Mellon University. Department of Computer Science, 1994.

[25] WIENANDS, R.; JOPPICH, W. *Practical Fourier Analysis For Multigrid Methods*. Boca Raton, Florida: Chapman  Hall/CRC Press, 2005.

[26] TROTTENBERG, C. O. U.; SCHULLER, A. *Multigrid*. London: Academic Press, 2001.

[27] BRIGGS, B. *A Multigrid Tutorial*. Philadelphia, Pennsylvania: Academic Press, 1999.

[28] VILLAR, M. M. *Análise numérica detalhada de escoamentos multifásicos bidimensionais*. Tese (Doutorado) — Universidade Federal de Uberlândia, 2007.

[29] SAAD, Y. *Iterative Methods for Sparse Linear Systems*. second edition. Minneapolis, Minnesota: Yousef Saad, 2000.

[30] CAMPOS, F. F. *Algoritmos Numéricos*. Rio de Janeiro: LTC, Segunda edição.

[31] FRANCO, N. B. *Cálculo Numérico*. São Paulo: Pearson Prentice Hall, 2006.

# ANEXOS

# I. FORMA ALTERNATIVA DOS MÉTODOS ESTACIONÁRIOS

A equação 2.72 é uma forma possível de se escrever a função de iteração dos métodos estacionários, mas não a única. Uma outra forma muito comum de ser vista em livros texto é dada por

$$\mathbf{x}_{k+1} = \mathbf{N}\mathbf{x}_k + \mathbf{c}, \tag{I.1}$$

onde a matriz $\mathbf{N}$ e o vetor $\mathbf{c}$ dependem do método estacionário a ser utilizado.

A seguir, será demonstrada a dedução da matriz $\mathbf{N}_J$ para o método de Jacobi, bem como da matriz $\mathbf{N}_{GS}$ para o método de Gauss-Seidel. Para o método de Jacobi, partindo da equação do sistema $\mathbf{Ax} = \mathbf{b}$ e decompondo a matriz $\mathbf{A}$ na soma $(\mathbf{L} + \mathbf{D} + \mathbf{U})$, onde $\mathbf{L}$ é a matriz triangular inferior de $\mathbf{A}$, $\mathbf{D}$ é a matriz diagonal de $\mathbf{A}$ e $\mathbf{U}$ é a matriz triangular superior de $\mathbf{A}$, temos que

$$\mathbf{Ax} = \mathbf{b} \tag{I.2}$$

$$(\mathbf{L} + \mathbf{D} + \mathbf{U})\mathbf{x} = \mathbf{b} \tag{I.3}$$

$$\mathbf{Dx} = -(\mathbf{L} + \mathbf{U})\mathbf{x} + \mathbf{b} \tag{I.4}$$

$$\mathbf{x} = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{x} + \mathbf{D}^{-1}\mathbf{b} \tag{I.5}$$

Nesse momento, vale notar que a equação I.5 ainda é exata e muito parecida com a forma I.1. O método de Jacobi então propõe, na equação I.5, que

$$\mathbf{N}_J = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U}) \tag{I.6}$$

e

$$\mathbf{c}_J = \mathbf{D}^{-1}\mathbf{b}. \tag{I.7}$$

Substituindo então as equações I.6 e I.7 na equação I.1, temos

$$\mathbf{x}_{k+1} = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{x}_k + \mathbf{D}^{-1}\mathbf{b}. \tag{I.8}$$

A equação I.8 define o método de Jacobi.

Agora, para o método de Gauss-Seidel, partindo novamente da equação do sistema $\mathbf{Ax} = \mathbf{b}$ e fazendo a mesma decomposição $(\mathbf{L} + \mathbf{D} + \mathbf{U})$ para a matriz $\mathbf{A}$, temos que

$$\mathbf{Ax} = \mathbf{b} \tag{I.9}$$

$$(\mathbf{L} + \mathbf{D} + \mathbf{U})\mathbf{x} = \mathbf{b} \tag{I.10}$$

$$(\mathbf{L} + \mathbf{D})\mathbf{x} = -\mathbf{Ux} + \mathbf{b} \tag{I.11}$$

$$\mathbf{x} = -(\mathbf{L} + \mathbf{D})^{-1}\mathbf{Ux} + (\mathbf{L} + \mathbf{D})^{-1}\mathbf{b} \tag{I.12}$$

Novamente, a equação I.12 é exata e muito parecida com a forma I.1. O método de Gauss-Seidel então propõe, na equação I.12, que

$$\mathbf{N}_{GS} = -(\mathbf{L}+\mathbf{D})^{-1}\mathbf{U} \tag{I.13}$$

e

$$\mathbf{c}_{GS} = (\mathbf{L}+\mathbf{D})^{-1}\mathbf{b}. \tag{I.14}$$

Substituindo então as equações I.13 e I.14 na equação I.1, temos

$$\mathbf{x}_{k+1} = -(\mathbf{L}+\mathbf{D})^{-1}\mathbf{U}\mathbf{x}_k + (\mathbf{L}+\mathbf{D})^{-1}\mathbf{b}. \tag{I.15}$$

A equação I.15 define o método de Gauss-Seidel.

Agora, recapitulando a equação 2.72 abaixo, temos

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{M}^{-1}\mathbf{r}_k. \tag{I.16}$$

O objetivo agora é mostrar a equivalência da equação I.16 com a equação I.1 para os métodos de Jacobi e Gauss-Seidel. Começando com o método de Jacobi, conforme é mostrado na seção 2.7.2, temos que $\mathbf{M}_J = \mathbf{D}$. Substituindo então $\mathbf{M}_J = \mathbf{D}$ na equação I.16, temos

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{D}^{-1}\mathbf{r}_k \tag{I.17}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{D}^{-1}(\mathbf{b} - \mathbf{A}\mathbf{x}_k) \tag{I.18}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{D}^{-1}\left[\mathbf{b} - (\mathbf{L}+\mathbf{D}+\mathbf{U})\mathbf{x}_k\right] \tag{I.19}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{D}^{-1}\mathbf{b} - \mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})\mathbf{x}_k - \mathbf{D}^{-1}\mathbf{D}\mathbf{x}_k \tag{I.20}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{D}^{-1}\mathbf{b} - \mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})\mathbf{x}_k - \mathbf{x}_k \tag{I.21}$$

$$\mathbf{x}_{k+1} = \mathbf{D}^{-1}\mathbf{b} - \mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})\mathbf{x}_k \tag{I.22}$$

A equação I.22 é exatamente igual à equação I.8, o que prova a equivalência das equações I.1 e I.16 para o método de Jacobi.

Agora, partindo para o método de Gauss-Seidel, conforme é mostrado na seção 2.7.2, temos que $\mathbf{M}_{GS} = \mathbf{E} = \mathbf{L}+\mathbf{D}$. Substituindo então $\mathbf{M}_{GS} = \mathbf{E}$ na equação I.16, temos

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{E}^{-1}\mathbf{r}_k \tag{I.23}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + (\mathbf{L}+\mathbf{D})^{-1}(\mathbf{b} - \mathbf{A}\mathbf{x}_k) \tag{I.24}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + (\mathbf{L}+\mathbf{D})^{-1}\left[\mathbf{b} - (\mathbf{L}+\mathbf{D}+\mathbf{U})\mathbf{x}_k\right] \tag{I.25}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + (\mathbf{L}+\mathbf{D})^{-1}\mathbf{b} - (\mathbf{L}+\mathbf{D})^{-1}(\mathbf{L}+\mathbf{D}+\mathbf{U})\mathbf{x}_k \tag{I.26}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + (\mathbf{L}+\mathbf{D})^{-1}\mathbf{b} - (\mathbf{L}+\mathbf{D})^{-1}(\mathbf{L}+\mathbf{D})\mathbf{x}_k - (\mathbf{L}+\mathbf{D})^{-1}\mathbf{U}\mathbf{x}_k \tag{I.27}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + (\mathbf{L}+\mathbf{D})^{-1}\mathbf{b} - \mathbf{x}_k - (\mathbf{L}+\mathbf{D})^{-1}\mathbf{U}\mathbf{x}_k \tag{I.28}$$

$$\mathbf{x}_{k+1} = (\mathbf{L}+\mathbf{D})^{-1}\mathbf{b} - (\mathbf{L}+\mathbf{D})^{-1}\mathbf{U}\mathbf{x}_k \tag{I.29}$$

A equação I.29 é exatamente igual à equação I.15, o que prova a equivalência das equações I.1 e I.16 para o método de Gauss-Seidel.

# II. CÓDIGOS

## II.1 Código do método Gauss-Seidel

```
import numpy as np
import timeit
from numba import njit

#DEFINIÇÃO DAS PRINCIPAIS VARIÁVEIS DO PROBLEMA

N=128                   #índice do último ponto na discretização
L=1.0                   #largura da cavidade
H=L                     #altura da cavidade
Re=3000                  #número de Reynolds
tempo_final=0.05         #tempo final da simulação
U_topo=1.0              #velocidade da parede superior da cavidade

dx=L/N
dy=dx
dt=0.0005

u_star=np.zeros([N+1,N+2],float)        #compontente horizontal de vel. estrela
v_star=np.zeros([N+2,N+1],float)        #componente vertical de vel. estrela
p=np.zeros([N+2,N+2],float)             #pressão
u=np.zeros([N+1,N+2],float)             #componente horizontal de velocidade
v=np.zeros([N+2,N+1],float)             #componente vertical de velocidade
b=np.zeros([N,N],float)                 #vetor resposta do sistema Ax=b
itera_por_passo_graph=np.zeros(100)     #vetor gerado para criar gráficos
erro_graph=np.zeros([0],float)          #vetor gerado para criar gráficos
Res=np.zeros([N+2,N+2],float)             #vetor resíduo

#DEFINIÇÃO DAS CONDIÇÕES DE CONTORNO E CONDIÇÃO INICIAL DO PROBLEMA

for i in range(1,N):
    u[i,N]=2.0*U_topo-u[i,N-1]

u_star=np.copy(u)

#VERIFICANDO CONDIÇÕES DE ESTABILIDADE DAS DIFERENÇAS FINITAS
```

```
if dt>=(dx) or dt>=(dy):
    print('CONDIÇÃO DE ESTABILIDADE 1 NÃO ATENDIDA')
    exit()
if dx>=(1/(Re**(1/2.0))) or dy>=(1/(Re**(1/2.0))):
    print('CONDIÇÃO DE ESTABILIDADE 2 NÃO ATENDIDA')
    exit()
if dt>=(1/4.0*Re*dx**2) or dt>=(1/4.0*Re*dy**2) :
    print('CONDIÇÃO DE ESTABILIDADE 3 NÃO ATENDIDA')
    exit()
#CÁLCULO DOS COEFICIENTES DO ESTÊNCIL DO PROBLEMA DA CAVIDADE BIDIMENSIONAL

r_e=np.zeros([N, N],float)
c_e=np.zeros([N, N],float)
l_e=np.zeros([N, N],float)
u_e=np.zeros([N, N],float)
d_e=np.zeros([N, N],float)

for i in range(0,N-1):
    for j in range(0,N):
        r_e[i,j]=1

for i in range(0,N):
    for j in range(0,N):
        if i==0:
            c_e[i,j]=-3
        elif i==N-1:
            c_e[i,j]=-3
        elif j==0:
            c_e[i,j]=-3
        elif j==N-1:
            c_e[i,j]=-3
        else:
            c_e[i,j]=-4
c_e[0,0]=-2
c_e[N-1,0]=-2
c_e[0,N-1]=-2
c_e[N-1,N-1]=-2

for i in range(1,N):
    for j in range(0,N):
        l_e[i,j]=1
```

```
for i in range(0,N):
    for j in range(0,N-1):
        u_e[i,j]=1

for i in range(0,N):
    for j in range(1,N):
        d_e[i,j]=1

##############################

start = timeit.time.time()
tempo=0.0
passo_de_tempo=0
iteracao_total=0

#############################
#DEFININDO FUNÇÕES

@njit
def calcula_pressao_gs(p, u_star, v_star,b, N, dx, dy, dt, tol,erro_graph,Res):
    erro=100
    iteracao_pressao=0
    omega=1
    if passo_de_tempo==0:
        def calcula_delta(R):
            delta=0
            for i in range(N+2):
                for j in range(N+2):
                    delta=delta+R[i,j]*R[i,j]
            return delta
    while erro>tol:
        R_max=0.0
        for i in range(0,N):
            for j in range(0,N):
                # if i == N//2 and j == N/2:
                #     # p[i,j]=-0.00868782
                #     p[i,j]=1
                #     # p[i,j]=-0.194239
                # else:
                a_ii=c_e[i,j]
                R=(1.0/a_ii)*(b[i,j]-(r_e[i,j]*p[i+1,j]+c_e[i,j]*p[i,j]+l_e[i,j]*p[i-1,j]+
```

```
                                        u_e[i,j]*p[i,j+1]+d_e[i,j]*p[i,j-1]))
                p[i,j]=p[i,j]+omega*R
                # p[N//2,N//2]=-0.00868782
                if np.abs(R)>R_max:
                    R_max=np.abs(R)
        for i in range(N):
            for j in range(N):
                Res[i,j]=b[i,j]-(r_e[i,j]*p[i+1,j]+c_e[i,j]*p[i,j]+l_e[i,j]*p[i-1,j]+
                                            u_e[i,j]*p[i,j+1]+d_e[i,j]*p[i,j-1])

        delta=calcula_delta(Res)
        soma=0
        for i in range(N):
            for j in range(N):
                soma=soma+b[i,j]*b[i,j]
        b_delta=soma
        erro=np.sqrt(delta/b_delta)
        # erro=R_max

        iteracao_pressao=iteracao_pressao+1

        #GERANDO OS VETORES PARA OS GRAFICOS DE ERRO X Nº DE ITERACOES
        if passo_de_tempo==0:
            erro_graph=np.append(erro_graph,erro)

    #ATUALIZANDO CONTORNO DA PRESSAO

    for i in range(N):
        p[-1,i]=p[0,i]
        p[N,i]=p[N-1,i]
        p[i,-1]=p[i,0]
        p[i,N]=p[i,N-1]

    p[-1,-1]=p[0,0]
    p[-1,N]=p[0,N-1]
    p[N,-1]=p[N-1,0]
    p[N,N]=p[N-1,N-1]
    return p, iteracao_pressao, erro_graph
@njit
def calcula_u_star(N,u,v,u_star):
    for i in range(1,N):
        for j in range(0,N):
```

```
                C1=0.25*(v[i-1,j]+v[i,j]+v[i,j+1]+v[i-1,j+1])
                a1=u[i,j]*(u[i+1,j]-u[i-1,j])/(2*dx)+C1*(u[i,j+1]-u[i,j-1])/(2*dy)
                a2=(u[i+1,j]-2*u[i,j]+u[i-1,j])/(dx*dx)+(u[i,j+1]-2*u[i,j]+u[i,j-1])/(dy*dy)
                R=dt*(-a1+(1.0/Re)*a2)
                u_star[i,j]=u[i,j]+R
    #ATUALIZANDO CONTORNO DAS VELOCIDADES STAR
    for i in range(1,N):
        u_star[i,-1]=-u_star[i,0]

    for i in range(1,N):
        u_star[i,N]=2*U_topo-u_star[i,N-1]

    return u_star
@njit
def calcula_v_star(N,u,v,v_star):
    for i in range(0,N):
        for j in range(1,N):
            C2=0.25*(u[i,j]+u[i+1,j]+u[i+1,j-1]+u[i,j-1])
            a1=C2*(v[i+1,j]-v[i-1,j])/(2*dx)+v[i,j]*(v[i,j+1]-v[i,j-1])/(2*dy)
            a2=(v[i+1,j]-2*v[i,j]+v[i-1,j])/(dx*dx)+(v[i,j+1]-2*v[i,j]+v[i,j-1])/(dy*dy)
            R=dt*(-a1+(1.0/Re)*a2)
            v_star[i,j]=v[i,j]+R
    #ATUALIZANDO CONTORNO DAS VELOCIDADES STAR
    for j in range(1,N):
        v_star[-1,j]=-v_star[0,j]

    for j in range(1,N):
        v_star[N,j]=-v_star[N-1,j]
    return v_star

@njit
def calcula_u_e_v(u,v,u_star,v_star, p,dt,dx,dy,N):
    for i in range(0,N):
        for j in range(0,N):
            u[i,j]=u_star[i,j]-dt*(p[i,j]-p[i-1,j])/(dx)
            v[i,j]=v_star[i,j]-dt*(p[i,j]-p[i,j-1])/(dy)
    #ATUALIZANDO CONTORNO DAS VELOCIDADES
    for i in range(1,N):
        u[i,-1]=-u[i,0]
    for i in range(1,N):
        u[i,N]=2*U_topo-u[i,N-1]
```

```
    for j in range(1,N):
        v[-1,j]=-v[0,j]
    for j in range(1,N):
        v[N,j]=-v[N-1,j]
    return u,v

@njit
def calcula_b(N,dt,dx,u_star,v_star,b):
    for i in range(0,N):
        for j in range(0,N):
            b[i,j]=(dx/dt)*(u_star[i+1,j]-u_star[i,j]+v_star[i,j+1]-v_star[i,j])
    return b

#################################

#ENTRANDO NO LOOP PRINCIPAL
while tempo<tempo_final:

    calcula_u_star(N,u,v,u_star)
    calcula_v_star(N,u,v,v_star)

    ############################

    #CALCULANDO O VETOR B

    b=calcula_b(N, dt, dx, u_star, v_star,b)

    #CALCULANDO A PRESSÃO MÉTODO DE SOR
    tol=10**(-6)
    if passo_de_tempo==0:
        erro_graph=np.zeros([0],float)

# Nas linhas abaixo, a variável funcao_calcula recebe os valores de pressao,
# iteracões feitas naquele passo de tempo e os valores de erro para cada
# iteração no primeiro passo de tempo para gerar gráficos posteriormente.
# nas linhas abaixo, cada valor da variável funcao_calcula é passada para uma
# variável adequada, como a pressão é passada para a variável p, por exemplo.

    funcao_calcula=calcula_pressao_gs(p, u_star, v_star, b, N, dx, dy, dt, tol,erro_graph,Res)
    p=funcao_calcula[0]
    iteracao_pressao=funcao_calcula[1]
    iteracao_total=iteracao_total+iteracao_pressao
```

```
    if passo_de_tempo==0:
        erro_graph=funcao_calcula[2]
        iteracao_primeiro_tempo=iteracao_pressao

    #CALCULANDO A NOVA VELOCIDADE

    u_e_v=calcula_u_e_v(u,v,u_star,v_star, p,dt,dx,dy,N)
    u=u_e_v[0]
    v=u_e_v[1]

    tempo=tempo+dt
    itera_por_passo_graph[passo_de_tempo]=iteracao_pressao
    passo_de_tempo=passo_de_tempo+1
    print('No passo de tempo ',passo_de_tempo,'foram realizadas ',
          iteracao_pressao, 'iteracoes na pressao')

finish = timeit.time.time()
tempo_execucao=finish-start
print('Tempo de execução:',finish-start)
print('Numero total de iteracoes: ',iteracao_total)
# exit()

#SALVANDO INFORMAÇÕES EM FORMATO TXT
info=np.array([Re,N,tol, dt,tempo_final,tempo_execucao,iteracao_primeiro_tempo,
               iteracao_total])
np.savetxt('info_gc.txt', info,fmt='%1.6f' , header=
           'Re,N,tol,passo_de_tempo,tempo_final,tempo_execucao,'
           'iteracao_primeiro_tempo,iteracao_total')
np.savetxt('erro_graph_gc.txt',erro_graph, fmt='%1.7f')
np.savetxt('campo_pressao_gc.txt', p, fmt='%1.7f')
np.savetxt('campo_u_gc.txt', u, fmt='%1.7f')
np.savetxt('campo_v_gc.txt', v, fmt='%1.7f')
np.savetxt('itera_por_passo_graph_gc.txt', itera_por_passo_graph,fmt='%1.1f')
```

## II.2 Código do método Jacobi Amortecido

```
import numpy as np
import timeit
from numba import njit

#DEFINIÇÃO DAS PRINCIPAIS VARIÁVEIS DO PROBLEMA
```

```
N=128                      #índice do último ponto na discretização
L=1.0                     #largura da cavidade
H=L                       #altura da cavidade
Re=3000                    #número de Reynolds
tempo_final=0.05           #tempo final da simulação
U_topo=1.0                #velocidade da parede superior da cavidade

dx=L/N
dy=dx
dt=0.0005

u_star=np.zeros([N+1,N+2],float)        #compontente horizontal de vel. estrela
v_star=np.zeros([N+2,N+1],float)        #componente vertical de vel. estrela
p=np.zeros([N+2,N+2],float)             #pressão
p_new=np.zeros([N+2,N+2],float)         #vetor pressão auxiliar
u=np.zeros([N+1,N+2],float)             #componente horizontal de velocidade
v=np.zeros([N+2,N+1],float)             #componente vertical de velocidade
b=np.zeros([N,N],float)                 #vetor resposta do sistema Ax=b
itera_por_passo_graph=np.zeros(100)     #vetor gerado para criar gráficos
erro_graph=np.zeros([0],float)          #vetor gerado para criar gráficos
Res=np.zeros([N+2,N+2],float)             #vetor resíduo

#DEFINIÇÃO DAS CONDIÇÕES DE CONTORNO E CONDIÇÃO INICIAL DO PROBLEMA

for i in range(1,N):
    u[i,N]=2*U_topo-u[i,N-1]

u_star=np.copy(u)

#VERIFICANDO CONDIÇÕES DE ESTABILIDADE DAS DIFERENÇAS FINITAS

if dt>=(dx) or dt>=(dy):
    print('CONDIÇÃO DE ESTABILIDADE 1 NÃO ATENDIDA')
    exit()
if dx>=(1/(Re**(1/2.0))) or dy>=(1/(Re**(1/2.0))):
    print('CONDIÇÃO DE ESTABILIDADE 2 NÃO ATENDIDA')
    exit()
if dt>=(1/4.0*Re*dx**2) or dt>=(1/4.0*Re*dy**2) :
    print('CONDIÇÃO DE ESTABILIDADE 3 NÃO ATENDIDA')
    exit()
#CÁLCULO DOS COEFICIENTES DO ESTÊNCIL DO PROBLEMA DA CAVIDADE BIDIMENSIONAL
```

```
r_e=np.zeros([N, N],float)
c_e=np.zeros([N, N],float)
l_e=np.zeros([N, N],float)
u_e=np.zeros([N, N],float)
d_e=np.zeros([N, N],float)

for i in range(0,N-1):
    for j in range(0,N):
        r_e[i,j]=1

for i in range(0,N):
    for j in range(0,N):
        if i==0:
            c_e[i,j]=-3
        elif i==N-1:
            c_e[i,j]=-3
        elif j==0:
            c_e[i,j]=-3
        elif j==N-1:
            c_e[i,j]=-3
        else:
            c_e[i,j]=-4
c_e[0,0]=-2
c_e[N-1,0]=-2
c_e[0,N-1]=-2
c_e[N-1,N-1]=-2

for i in range(1,N):
    for j in range(0,N):
        l_e[i,j]=1

for i in range(0,N):
    for j in range(0,N-1):
        u_e[i,j]=1

for i in range(0,N):
    for j in range(1,N):
        d_e[i,j]=1

##############################
```

```
start = timeit.time.time()
tempo=0.0
passo_de_tempo=0
iteracao_total=0

######################
#DEFININDO FUNÇÕES

@njit
def calcula_pressao_jacobi_sr(p, u_star, v_star,b, N, dx, dy, dt, tol,
                              erro_graph,p_new,Res):
    erro=100
    iteracao_pressao=0
    omega_jacobi=0.9999
    if passo_de_tempo==0:
        def calcula_delta(R):
            delta=0
            for i in range(N+2):
                for j in range(N+2):
                    delta=delta+R[i,j]*R[i,j]
            return delta

    while erro>tol:
        R_max=0.0
        for i in range(0,N):
            for j in range(0,N):
                a_ii=c_e[i,j]
                R=(1.0/a_ii)*(b[i,j]-(r_e[i,j]*p[i+1,j]+c_e[i,j]*p[i,j]+
                                      l_e[i,j]*p[i-1,j]+u_e[i,j]*p[i,j+1]+
                                      d_e[i,j]*p[i,j-1]))
                p_new[i,j]=p[i,j]+omega_jacobi*R
                if np.abs(R)>R_max:
                    R_max=np.abs(R)
        for i in range(0,N):
            for j in range(0,N):
                p[i,j]=p_new[i,j]
        for i in range(N):
            for j in range(N):
                Res[i,j]=b[i,j]-(r_e[i,j]*p[i+1,j]+c_e[i,j]*p[i,j]+l_e[i,j]*p[i-1,j]+
                                      u_e[i,j]*p[i,j+1]+d_e[i,j]*p[i,j-1])

        delta=calcula_delta(Res)
```

```
        soma=0
        for i in range(N):
            for j in range(N):
                soma=soma+b[i,j]*b[i,j]
        b_delta=soma
        erro=np.sqrt(delta/b_delta)
        # erro=R_max
        # print(erro)
        iteracao_pressao=iteracao_pressao+1

        #GERANDO OS VETORES PARA OS GRAFICOS DE ERRO X Nº DE ITERACOES
        if passo_de_tempo==0:
            erro_graph=np.append(erro_graph,erro)

    #ATUALIZANDO CONTORNO DA PRESSAO

    for i in range(N):
        p[-1,i]=p[0,i]
        p[N,i]=p[N-1,i]
        p[i,-1]=p[i,0]
        p[i,N]=p[i,N-1]

    p[-1,-1]=p[0,0]
    p[-1,N]=p[0,N-1]
    p[N,-1]=p[N-1,0]
    p[N,N]=p[N-1,N-1]
    return p,iteracao_pressao,erro_graph
@njit
def calcula_u_star(N,u,v,u_star):
    for i in range(1,N):
        for j in range(0,N):
            C1=0.25*(v[i-1,j]+v[i,j]+v[i,j+1]+v[i-1,j+1])
            a1=u[i,j]*(u[i+1,j]-u[i-1,j])/(2*dx)+C1*(u[i,j+1]-u[i,j-1])/(2*dy)
            a2=(u[i+1,j]-2*u[i,j]+u[i-1,j])/(dx*dx)+(u[i,j+1]-2*u[i,j]+u[i,j-1])/(dy*dy)
            R=dt*(-a1+(1.0/Re)*a2)
            u_star[i,j]=u[i,j]+R
    #ATUALIZANDO CONTORNO DAS VELOCIDADES STAR
    for i in range(1,N):
        u_star[i,-1]=-u_star[i,0]

    for i in range(1,N):
        u_star[i,N]=2*U_topo-u_star[i,N-1]
```

```
    return u_star
@njit
def calcula_v_star(N,u,v,v_star):
    for i in range(0,N):
        for j in range(1,N):
            C2=0.25*(u[i,j]+u[i+1,j]+u[i+1,j-1]+u[i,j-1])
            a1=C2*(v[i+1,j]-v[i-1,j])/(2*dx)+v[i,j]*(v[i,j+1]-v[i,j-1])/(2*dy)
            a2=(v[i+1,j]-2*v[i,j]+v[i-1,j])/(dx*dx)+(v[i,j+1]-2*v[i,j]+v[i,j-1])/(dy*dy)
            R=dt*(-a1+(1.0/Re)*a2)
            v_star[i,j]=v[i,j]+R
    #ATUALIZANDO CONTORNO DAS VELOCIDADES STAR
    for j in range(1,N):
        v_star[-1,j]=-v_star[0,j]

    for j in range(1,N):
        v_star[N,j]=-v_star[N-1,j]
    return v_star

@njit
def calcula_u_e_v(u,v,u_star,v_star, p,dt,dx,dy,N):
    for i in range(0,N):
        for j in range(0,N):
            u[i,j]=u_star[i,j]-dt*(p[i,j]-p[i-1,j])/(dx)
            v[i,j]=v_star[i,j]-dt*(p[i,j]-p[i,j-1])/(dy)
    #ATUALIZANDO CONTORNO DAS VELOCIDADES
    for i in range(1,N):
        u[i,-1]=-u[i,0]
    for i in range(1,N):
        u[i,N]=2*U_topo-u[i,N-1]

    for j in range(1,N):
        v[-1,j]=-v[0,j]
    for j in range(1,N):
        v[N,j]=-v[N-1,j]
    return u,v

@njit
def calcula_b(N,dt,dx,u_star,v_star,b):
    for i in range(0,N):
        for j in range(0,N):
            b[i,j]=(dx/dt)*(u_star[i+1,j]-u_star[i,j]+v_star[i,j+1]-v_star[i,j])
```

```
    return b
###########################

#ENTRANDO NO LOOP PRINCIPAL
while tempo<tempo_final:

    calcula_u_star(N,u,v,u_star)
    calcula_v_star(N,u,v,v_star)

    ##############################

    #CALCULANDO O VETOR B

    b=calcula_b(N, dt, dx, u_star, v_star,b)

    #CALCULANDO A PRESSÃO MÉTODO DE GAUSS-SEIDEL
    tol=10**(-6)
    if passo_de_tempo==0:
        erro_graph=np.zeros([0],float)


    funcao_calcula=calcula_pressao_jacobi_sr(p,u_star, v_star, b, N, dx, dy, dt,
                                             tol,erro_graph,p_new,Res)
    p=funcao_calcula[0]
    iteracao_pressao=funcao_calcula[1]
    iteracao_total=iteracao_total+iteracao_pressao
    if passo_de_tempo==0:
        erro_graph=funcao_calcula[2]
        iteracao_primeiro_tempo=iteracao_pressao

    #CALCULANDO A NOVA VELOCIDADE

    u_e_v=calcula_u_e_v(u,v,u_star,v_star, p,dt,dx,dy,N)
    u=u_e_v[0]
    v=u_e_v[1]

    tempo=tempo+dt
    itera_por_passo_graph[passo_de_tempo]=iteracao_pressao
    passo_de_tempo=passo_de_tempo+1

    print('No passo de tempo ',passo_de_tempo,'foram realizadas ',
           iteracao_pressao, 'iteracoes na pressao')
```

```
finish = timeit.time.time()
tempo_execucao=finish-start
print('Tempo de execução:',finish-start)
print('Numero total de iteracoes: ',iteracao_total)

#SALVANDO INFORMAÇÕES EM FORMATO TXT
info=np.array([Re,N,tol,dt,tempo_final,tempo_execucao,iteracao_primeiro_tempo,
               iteracao_total])
np.savetxt('info_jacobi_sr.txt', info,fmt='%1.6f' , header=
           'Re,N,tol,passo_de_tempo,tempo_final,tempo_execucao,'
           'iteracao_primeiro_tempo,iteracao_total')
np.savetxt('erro_graph_jacobi_sr.txt',erro_graph, fmt='%1.7f')
np.savetxt('campo_pressao_jacobi_sr.txt', p, fmt='%1.7f')
np.savetxt('campo_u_jacobi_sr.txt', u, fmt='%1.7f')
np.savetxt('campo_v_jacobi_srl.txt', v, fmt='%1.7f')
np.savetxt('itera_por_passo_graph_sr.txt', itera_por_passo_graph,fmt='%1.1f')
```

## II.3 Código do método SOR

```
import numpy as np
import timeit
from numba import njit

#DEFINIÇÃO DAS PRINCIPAIS VARIÁVEIS DO PROBLEMA

N=128                   #índice do último ponto na discretização
L=1.0                   #largura da cavidade
H=L                     #altura da cavidade
Re=3000                  #número de Reynolds
tempo_final=0.05         #tempo final da simulação
U_topo=1.0              #velocidade da parede superior da cavidade

dx=L/N
dy=dx
dt=0.0005

u_star=np.zeros([N+1,N+2],float)        #compontente horizontal de vel. estrela
v_star=np.zeros([N+2,N+1],float)        #componente vertical de vel. estrela
p=np.zeros([N+2,N+2],float)             #pressão
u=np.zeros([N+1,N+2],float)             #componente horizontal de velocidade
v=np.zeros([N+2,N+1],float)             #componente vertical de velocidade
```

```
b=np.zeros([N,N],float)                  #vetor resposta do sistema Ax=b
itera_por_passo_graph=np.zeros(100)      #vetor gerado para criar gráficos
erro_graph=np.zeros([0],float)           #vetor gerado para criar gráficos
Res=np.zeros([N+2,N+2],float)              #vetor resíduo

#DEFINIÇÃO DAS CONDIÇÕES DE CONTORNO E CONDIÇÃO INICIAL DO PROBLEMA

for i in range(1,N):
    u[i,N]=2.0*U_topo-u[i,N-1]

u_star=np.copy(u)

#VERIFICANDO CONDIÇÕES DE ESTABILIDADE DAS DIFERENÇAS FINITAS

if dt>=(dx) or dt>=(dy):
    print('CONDIÇÃO DE ESTABILIDADE 1 NÃO ATENDIDA')
    exit()
if dx>=(1/(Re**(1/2.0))) or dy>=(1/(Re**(1/2.0))):
    print('CONDIÇÃO DE ESTABILIDADE 2 NÃO ATENDIDA')
    exit()
if dt>=(1/4.0*Re*dx**2) or dt>=(1/4.0*Re*dy**2) :
    print('CONDIÇÃO DE ESTABILIDADE 3 NÃO ATENDIDA')
    exit()
#CÁLCULO DOS COEFICIENTES DO ESTÊNCIL DO PROBLEMA DA CAVIDADE BIDIMENSIONAL

r_e=np.zeros([N, N],float)
c_e=np.zeros([N, N],float)
l_e=np.zeros([N, N],float)
u_e=np.zeros([N, N],float)
d_e=np.zeros([N, N],float)

for i in range(0,N-1):
    for j in range(0,N):
        r_e[i,j]=1

for i in range(0,N):
    for j in range(0,N):
        if i==0:
            c_e[i,j]=-3
        elif i==N-1:
            c_e[i,j]=-3
        elif j==0:
```

```
            c_e[i,j]=-3
        elif j==N-1:
            c_e[i,j]=-3
        else:
            c_e[i,j]=-4
c_e[0,0]=-2
c_e[N-1,0]=-2
c_e[0,N-1]=-2
c_e[N-1,N-1]=-2

for i in range(1,N):
    for j in range(0,N):
        l_e[i,j]=1

for i in range(0,N):
    for j in range(0,N-1):
        u_e[i,j]=1

for i in range(0,N):
    for j in range(1,N):
        d_e[i,j]=1

##############################

start = timeit.time.time()
tempo=0.0
passo_de_tempo=0
iteracao_total=0

#############################
#DEFININDO FUNÇÕES

@njit
def calcula_pressao_sor(p, u_star, v_star,b, N, dx, dy, dt, tol,erro_graph,Res):
    erro=100
    iteracao_pressao=0
    omega=1.97
    if passo_de_tempo==0:
        def calcula_delta(R):
            delta=0
            for i in range(N+2):
                for j in range(N+2):
```

```
                delta=delta+R[i,j]*R[i,j]
            return delta
    while erro>tol:
        R_max=0.0
        for i in range(0,N):
            for j in range(0,N):
                # if i == N//2 and j == N/2:
                #     # p[i,j]=-0.00868782
                #     p[i,j]=1
                #     # p[i,j]=-0.194239
                # else:
                a_ii=c_e[i,j]
                R=(1.0/a_ii)*(b[i,j]-(r_e[i,j]*p[i+1,j]+c_e[i,j]*p[i,j]+l_e[i,j]*p[i-1,j]+
                                    u_e[i,j]*p[i,j+1]+d_e[i,j]*p[i,j-1]))
                p[i,j]=p[i,j]+omega*R
                # p[N//2,N//2]=-0.00868782
                if np.abs(R)>R_max:
                    R_max=np.abs(R)
        for i in range(N):
            for j in range(N):
                Res[i,j]=b[i,j]-(r_e[i,j]*p[i+1,j]+c_e[i,j]*p[i,j]+l_e[i,j]*p[i-1,j]+
                                        u_e[i,j]*p[i,j+1]+d_e[i,j]*p[i,j-1])

        delta=calcula_delta(Res)
        soma=0
        for i in range(N):
            for j in range(N):
                soma=soma+b[i,j]*b[i,j]
        b_delta=soma
        erro=np.sqrt(delta/b_delta)
        # erro=R_max

        iteracao_pressao=iteracao_pressao+1

        #GERANDO OS VETORES PARA OS GRAFICOS DE ERRO X Nº DE ITERACOES
        if passo_de_tempo==0:
            erro_graph=np.append(erro_graph,erro)

    #ATUALIZANDO CONTORNO DA PRESSAO

    for i in range(N):
        p[-1,i]=p[0,i]
```

```
        p[N,i]=p[N-1,i]
        p[i,-1]=p[i,0]
        p[i,N]=p[i,N-1]

    p[-1,-1]=p[0,0]
    p[-1,N]=p[0,N-1]
    p[N,-1]=p[N-1,0]
    p[N,N]=p[N-1,N-1]
    return p, iteracao_pressao, erro_graph
@njit
def calcula_u_star(N,u,v,u_star):
    for i in range(1,N):
        for j in range(0,N):
            C1=0.25*(v[i-1,j]+v[i,j]+v[i,j+1]+v[i-1,j+1])
            a1=u[i,j]*(u[i+1,j]-u[i-1,j])/(2*dx)+C1*(u[i,j+1]-u[i,j-1])/(2*dy)
            a2=(u[i+1,j]-2*u[i,j]+u[i-1,j])/(dx*dx)+(u[i,j+1]-2*u[i,j]+u[i,j-1])/(dy*dy)
            R=dt*(-a1+(1.0/Re)*a2)
            u_star[i,j]=u[i,j]+R
    #ATUALIZANDO CONTORNO DAS VELOCIDADES STAR
    for i in range(1,N):
        u_star[i,-1]=-u_star[i,0]

    for i in range(1,N):
        u_star[i,N]=2*U_topo-u_star[i,N-1]

    return u_star
@njit
def calcula_v_star(N,u,v,v_star):
    for i in range(0,N):
        for j in range(1,N):
            C2=0.25*(u[i,j]+u[i+1,j]+u[i+1,j-1]+u[i,j-1])
            a1=C2*(v[i+1,j]-v[i-1,j])/(2*dx)+v[i,j]*(v[i,j+1]-v[i,j-1])/(2*dy)
            a2=(v[i+1,j]-2*v[i,j]+v[i-1,j])/(dx*dx)+(v[i,j+1]-2*v[i,j]+v[i,j-1])/(dy*dy)
            R=dt*(-a1+(1.0/Re)*a2)
            v_star[i,j]=v[i,j]+R
    #ATUALIZANDO CONTORNO DAS VELOCIDADES STAR
    for j in range(1,N):
        v_star[-1,j]=-v_star[0,j]

    for j in range(1,N):
        v_star[N,j]=-v_star[N-1,j]
    return v_star
```

```
@njit
def calcula_u_e_v(u,v,u_star,v_star, p,dt,dx,dy,N):
    for i in range(0,N):
        for j in range(0,N):
            u[i,j]=u_star[i,j]-dt*(p[i,j]-p[i-1,j])/(dx)
            v[i,j]=v_star[i,j]-dt*(p[i,j]-p[i,j-1])/(dy)
    #ATUALIZANDO CONTORNO DAS VELOCIDADES
    for i in range(1,N):
        u[i,-1]=-u[i,0]
    for i in range(1,N):
        u[i,N]=2*U_topo-u[i,N-1]

    for j in range(1,N):
        v[-1,j]=-v[0,j]
    for j in range(1,N):
        v[N,j]=-v[N-1,j]
    return u,v

@njit
def calcula_b(N,dt,dx,u_star,v_star,b):
    for i in range(0,N):
        for j in range(0,N):
            b[i,j]=(dx/dt)*(u_star[i+1,j]-u_star[i,j]+v_star[i,j+1]-v_star[i,j])
    return b

#################################

#ENTRANDO NO LOOP PRINCIPAL
while tempo<tempo_final:

    calcula_u_star(N,u,v,u_star)
    calcula_v_star(N,u,v,v_star)

    #############################

    #CALCULANDO O VETOR B

    b=calcula_b(N, dt, dx, u_star, v_star,b)

    #CALCULANDO A PRESSÃO MÉTODO DE SOR
    tol=10**(-6)
```

```
    if passo_de_tempo==0:
        erro_graph=np.zeros([0],float)

# Nas linhas abaixo, a variável funcao_calcula recebe os valores de pressao,
# iteracões feitas naquele passo de tempo e os valores de erro para cada
# iteração no primeiro passo de tempo para gerar gráficos posteriormente.
# nas linhas abaixo, cada valor da variável funcao_calcula é passada para uma
# variável adequada, como a pressão é passada para a variável p, por exemplo.

    funcao_calcula=calcula_pressao_sor(p, u_star, v_star, b, N, dx, dy, dt, tol,erro_graph,Res
    p=funcao_calcula[0]
    iteracao_pressao=funcao_calcula[1]
    iteracao_total=iteracao_total+iteracao_pressao
    if passo_de_tempo==0:
        erro_graph=funcao_calcula[2]
        iteracao_primeiro_tempo=iteracao_pressao

    #CALCULANDO A NOVA VELOCIDADE

    u_e_v=calcula_u_e_v(u,v,u_star,v_star, p,dt,dx,dy,N)
    u=u_e_v[0]
    v=u_e_v[1]

    tempo=tempo+dt
    itera_por_passo_graph[passo_de_tempo]=iteracao_pressao
    passo_de_tempo=passo_de_tempo+1
    print('No passo de tempo ',passo_de_tempo,'foram realizadas ',
          iteracao_pressao, 'iteracoes na pressao')

finish = timeit.time.time()
tempo_execucao=finish-start
print('Tempo de execução:',finish-start)
print('Numero total de iteracoes: ',iteracao_total)
# exit()

#SALVANDO INFORMAÇÕES EM FORMATO TXT
info=np.array([Re,N,tol, dt,tempo_final,tempo_execucao,iteracao_primeiro_tempo,
               iteracao_total])
np.savetxt('info_sor.txt', info,fmt='%1.6f' , header=
           'Re,N,tol,passo_de_tempo,tempo_final,tempo_execucao,'
           'iteracao_primeiro_tempo,iteracao_total')
np.savetxt('erro_graph_sor.txt',erro_graph, fmt='%1.7f')
```

```
np.savetxt('campo_pressao_sor.txt', p, fmt='%1.7f')
np.savetxt('campo_u_sor.txt', u, fmt='%1.7f')
np.savetxt('campo_v_sor.txt', v, fmt='%1.7f')
np.savetxt('itera_por_passo_graph_sor.txt', itera_por_passo_graph,fmt='%1.1f')
```

## II.4 Código do método Gradiente Conjugado

```
import numpy as np
import timeit
from numba import njit

#DEFINIÇÃO DAS PRINCIPAIS VARIÁVEIS DO PROBLEMA

N=1024                        #índice do último ponto na discretização
L=1.0                        #largura da cavidade
H=L                          #altura da cavidade
Re=3000                       #número de Reynolds
tempo_final=0.05              #tempo final da simulação
U_topo=1.0                   #velocidade da parede superior da cavidade

dx=L/N
dy=dx
dt=0.0005

u_star=np.zeros([N+1,N+2],float)        #compontente horizontal de vel. estrela
v_star=np.zeros([N+2,N+1],float)        #componente vertical de vel. estrela
p=np.zeros([N+2,N+2],float)             #pressão
u=np.zeros([N+1,N+2],float)             #componente horizontal de velocidade
v=np.zeros([N+2,N+1],float)             #componente vertical de velocidade
b=np.zeros([N,N],float)                 #vetor resposta do sistema Ax=b
itera_por_passo_graph=np.zeros(100)     #vetor gerado para criar gráficos
erro_graph=np.zeros([0],float)          #vetor gerado para criar gráficos
R=np.zeros([N+2,N+2],float)             #vetor resíduo

#DEFINIÇÃO DAS CONDIÇÕES DE CONTORNO E CONDIÇÃO INICIAL DO PROBLEMA

for i in range(1,N):
    u[i,N]=2*U_topo-u[i,N-1]

u_star=np.copy(u)
```

```python
#VERIFICANDO CONDIÇÕES DE ESTABILIDADE DAS DIFERENÇAS FINITAS

if dt>=(dx) or dt>=(dy):
    print('CONDIÇÃO DE ESTABILIDADE 1 NÃO ATENDIDA')
    exit()
if dx>=(1/(Re**(1/2.0))) or dy>=(1/(Re**(1/2.0))):
    print('CONDIÇÃO DE ESTABILIDADE 2 NÃO ATENDIDA')
    exit()
if dt>=(1/4.0*Re*dx**2) or dt>=(1/4.0*Re*dy**2) :
    print('CONDIÇÃO DE ESTABILIDADE 3 NÃO ATENDIDA')
    exit()
#CÁLCULO DOS COEFICIENTES DO ESTÊNCIL DO PROBLEMA DA CAVIDADE BIDIMENSIONAL

r_e=np.zeros([N, N],float)
c_e=np.zeros([N, N],float)
l_e=np.zeros([N, N],float)
u_e=np.zeros([N, N],float)
d_e=np.zeros([N, N],float)

for i in range(0,N-1):
    for j in range(0,N):
        r_e[i,j]=1

for i in range(0,N):
    for j in range(0,N):
        if i==0:
            c_e[i,j]=-3
        elif i==N-1:
            c_e[i,j]=-3
        elif j==0:
            c_e[i,j]=-3
        elif j==N-1:
            c_e[i,j]=-3
        else:
            c_e[i,j]=-4
c_e[0,0]=-2
c_e[N-1,0]=-2
c_e[0,N-1]=-2
c_e[N-1,N-1]=-2

for i in range(1,N):
    for j in range(0,N):
```

```
        l_e[i,j]=1

for i in range(0,N):
    for j in range(0,N-1):
        u_e[i,j]=1

for i in range(0,N):
    for j in range(1,N):
        d_e[i,j]=1

##############################

start = timeit.time.time()
tempo=0.0
passo_de_tempo=0
iteracao_total=0

##############################

tol=10**(-6)
d=np.zeros([N+2,N+2],float)         #matriz gerada para armazenar variáveis do GC
s=np.zeros([N+2,N++2])              #matriz gerada para armazenar variáveis do GC
@njit
def calcula_pressao_gc(p,u_star, v_star, b, N, dx, dy, dt, tol,erro_graph,R,d,
                       passo_de_tempo,s):

    if passo_de_tempo==0:
        print('As funcões necessárias para rodar o algoritmo do GC foram definidas aqui')
        def calcula_R(R,alpha,s,N):
            for i in range(N):
                for j in range(N):
                    R[i,j]=R[i,j]-alpha*s[i,j]
            return R

        def calcula_s(s,r_e,c_e,l_e,u_e,d_e,d):
            for i in range(N):
                    for j in range(N):
                        s[i,j]=(r_e[i,j]*d[i+1,j]+c_e[i,j]*d[i,j]+l_e[i,j]*d[i-1,j]+
                                                            u_e[i,j]*d[i,j+1]+d_e[i,j]*d[i,j-1])
            return s
        def calcula_delta(R):
            delta=0
```

```
        for i in range(N+2):
            for j in range(N+2):
                delta=delta+R[i,j]*R[i,j]
        return delta
    def calcula_alpha(delta,d,s):
        soma=0
        for i in range(N+2):
            for j in range(N+2):
                soma=soma+d[i,j]*s[i,j]
        alpha=delta/soma
        return alpha
    def calcula_p(p,alpha,d):
        for i in range(N+2):
            for j in range(N+2):
                p[i,j]=p[i,j]+alpha*d[i,j]
        return p

    def calcula_d(delta_new,delta,d):
        for i in range(N+2):
            for j in range(N+2):
                d[i,j]=R[i,j]+delta_new/delta*d[i,j]
        return d
erro=100
k=0
iteracao_pressao=0
for i in range(N):
    for j in range(N):
        R[i,j]=b[i,j]-(r_e[i,j]*p[i+1,j]+c_e[i,j]*p[i,j]+l_e[i,j]*p[i-1,j]+
                                    u_e[i,j]*p[i,j+1]+d_e[i,j]*p[i,j-1])
d=np.copy(R)
# soma=0
# for i in range(N):
#     for j in range(N):
#         soma=soma+b[i,j]*b[i,j]
# b_delta=soma
delta=calcula_delta(R)
soma=0
for i in range(N):
    for j in range(N):
        soma=soma+b[i,j]*b[i,j]
b_delta=soma
```

```
    while delta>((tol**2)*b_delta):
        s=calcula_s(s,r_e,c_e,l_e,u_e,d_e,d)
        delta=calcula_delta(R)
        alpha=calcula_alpha(delta,d,s)
        p=calcula_p(p,alpha,d)
        R=calcula_R(R,alpha,s,N)
        delta_new=calcula_delta(R)
        d=calcula_d(delta_new,delta,d)

        k=k+1
        erro=np.sqrt(delta/b_delta)
        # erro=np.sqrt(delta_new/b_delta)
        iteracao_pressao=iteracao_pressao+1
        soma=0
        for i in range(N):
            for j in range(N):
                soma=soma+b[i,j]*b[i,j]
        b_delta=soma

        #GERANDO OS VETORES PARA OS GRAFICOS DE ERRO X Nº DE ITERACOES
        if passo_de_tempo==0:
            erro_graph=np.append(erro_graph,erro)


    #ATUALIZANDO CONTORNO DA PRESSAO

    for i in range(N):
        p[-1,i]=p[0,i]
        p[N,i]=p[N-1,i]
        p[i,-1]=p[i,0]
        p[i,N]=p[i,N-1]

    p[-1,-1]=p[0,0]
    p[-1,N]=p[0,N-1]
    p[N,-1]=p[N-1,0]
    p[N,N]=p[N-1,N-1]
    return p,iteracao_pressao,erro_graph

@njit
def calcula_u_star(N,u,v,u_star):
    for i in range(1,N):
        for j in range(0,N):
```

```
            C1=0.25*(v[i-1,j]+v[i,j]+v[i,j+1]+v[i-1,j+1])
            a1=u[i,j]*(u[i+1,j]-u[i-1,j])/(2*dx)+C1*(u[i,j+1]-u[i,j-1])/(2*dy)
            a2=(u[i+1,j]-2*u[i,j]+u[i-1,j])/(dx*dx)+(u[i,j+1]-2*u[i,j]+u[i,j-1])/(dy*dy)
            R=dt*(-a1+(1.0/Re)*a2)
            u_star[i,j]=u[i,j]+R
    #ATUALIZANDO CONTORNO DAS VELOCIDADES STAR
    for i in range(1,N):
        u_star[i,-1]=-u_star[i,0]

    for i in range(1,N):
        u_star[i,N]=2*U_topo-u_star[i,N-1]

    return u_star
@njit
def calcula_v_star(N,u,v,v_star):
    for i in range(0,N):
        for j in range(1,N):
            C2=0.25*(u[i,j]+u[i+1,j]+u[i+1,j-1]+u[i,j-1])
            a1=C2*(v[i+1,j]-v[i-1,j])/(2*dx)+v[i,j]*(v[i,j+1]-v[i,j-1])/(2*dy)
            a2=(v[i+1,j]-2*v[i,j]+v[i-1,j])/(dx*dx)+(v[i,j+1]-2*v[i,j]+v[i,j-1])/(dy*dy)
            R=dt*(-a1+(1.0/Re)*a2)
            v_star[i,j]=v[i,j]+R

    #ATUALIZANDO CONTORNO DAS VELOCIDADES STAR
    for j in range(1,N):
        v_star[-1,j]=-v_star[0,j]

    for j in range(1,N):
        v_star[N,j]=-v_star[N-1,j]
    return v_star
@njit
def calcula_u_e_v(u,v,u_star,v_star, p,dt,dx,dy,N):
    for i in range(0,N):
        for j in range(0,N):
            u[i,j]=u_star[i,j]-dt*(p[i,j]-p[i-1,j])/(dx)
            v[i,j]=v_star[i,j]-dt*(p[i,j]-p[i,j-1])/(dy)
    #ATUALIZANDO CONTORNO DAS VELOCIDADES
    for i in range(1,N):
        u[i,-1]=-u[i,0]
    for i in range(1,N):
        u[i,N]=2*U_topo-u[i,N-1]
```

```
    for j in range(1,N):
        v[-1,j]=-v[0,j]
    for j in range(1,N):
        v[N,j]=-v[N-1,j]
    return u,v
@njit
def calcula_b(N,dt,dx,u_star,v_star,b):
    for i in range(0,N):
        for j in range(0,N):
            b[i,j]=(dx/dt)*(u_star[i+1,j]-u_star[i,j]+v_star[i,j+1]-v_star[i,j])
    return b
###########################

#ENTRANDO NO LOOP PRINCIPAL
while tempo<tempo_final:

    calcula_u_star(N,u,v,u_star)
    calcula_v_star(N,u,v,v_star)

    ##############################

    b=calcula_b(N, dt, dx, u_star, v_star,b)

    #CALCULANDO A PRESSÃO MÉTODO DO GRADIENTE CONJUGADO
    tol=10**(-6)
    if passo_de_tempo==0:
        erro_graph=np.zeros([0],float)

# Nas linhas abaixo, a variável funcao_calcula recebe os valores de pressao,
# iteracões feitas naquele passo de tempo e os valores de erro para cada
# iteração no primeiro passo de tempo para gerar gráficos posteriormente.
# nas linhas abaixo, cada valor da variável funcao_calcula é passada para uma
# variável adequada, como a pressão é passada para a variável p, por exemplo.

    funcao_calcula=calcula_pressao_gc(p,u_star, v_star, b, N, dx, dy, dt, tol,
                                  erro_graph,R,d,passo_de_tempo,s)
    p=funcao_calcula[0]
    iteracao_pressao=funcao_calcula[1]
    iteracao_total=iteracao_total+iteracao_pressao
    if passo_de_tempo==0:
        erro_graph=funcao_calcula[2]
        iteracao_primeiro_tempo=iteracao_pressao
```

```
    #CALCULANDO A NOVA VELOCIDADE

    u_e_v=calcula_u_e_v(u,v,u_star,v_star, p,dt,dx,dy,N)
    u=u_e_v[0]
    v=u_e_v[1]

    tempo=tempo+dt
    itera_por_passo_graph[passo_de_tempo]=iteracao_pressao
    passo_de_tempo=passo_de_tempo+1
    print('No passo de tempo ',passo_de_tempo,'foram realizadas ',
          iteracao_pressao, 'iteracoes na pressao')
finish = timeit.time.time()
tempo_execucao=finish-start
print('Tempo de execução:',finish-start)
print('Numero total de iteracoes: ',iteracao_total)

#SALVANDO INFORMAÇÕES EM FORMATO TXT
info=np.array([Re,N,tol,dt,tempo_final,tempo_execucao,iteracao_primeiro_tempo,
               iteracao_total])
np.savetxt('info_gradiente_conjugado.txt', info,fmt='%1.6f' , header=
           'Re,N,tol,passo_de_tempo,tempo_final,tempo_execucao,'
           'iteracao_primeiro_tempo,iteracao_total')
np.savetxt('erro_graph_gradiente_conjugado.txt',erro_graph, fmt='%1.7f')
np.savetxt('campo_pressao_gradiente_conjugado.txt', p, fmt='%1.7f')
np.savetxt('campo_u_gradiente_conjugado.txt', u, fmt='%1.7f')
np.savetxt('campo_v_gradiente_conjugado.txt', v, fmt='%1.7f')
np.savetxt('itera_por_passo_graph_gradiente_conjugado.txt', itera_por_passo_graph,fmt='%1.1f')
```

## II.5 Código do Método GCP-SSOR

```
import numpy as np
import timeit
from numba import njit

#DEFINIÇÃO DAS PRINCIPAIS VARIÁVEIS DO PROBLEMA

N=1024                    #índice do último ponto na discretização
L=1.0                    #largura da cavidade
H=L                      #altura da cavidade
Re=3000                   #número de Reynolds
```

```
tempo_final=0.05          #tempo final da simulação
U_topo=1.0               #velocidade da parede superior da cavidade

dx=L/N
dy=dx
dt=0.0005

u_star=np.zeros([N+1,N+2],float)        #compontente horizontal de vel. estrela
v_star=np.zeros([N+2,N+1],float)        #componente vertical de vel. estrela
p=np.zeros([N+2,N+2],float)             #pressão
u=np.zeros([N+1,N+2],float)             #componente horizontal de velocidade
v=np.zeros([N+2,N+1],float)             #componente vertical de velocidade
b=np.zeros([N,N],float)                 #vetor resposta do sistema Ax=b
itera_por_passo_graph=np.zeros(100)     #vetor gerado para criar gráficos
erro_graph=np.zeros([0],float)          #vetor gerado para criar gráficos
R=np.zeros([N+2,N+2],float)             #vetor resíduo

#DEFINIÇÃO DAS CONDIÇÕES DE CONTORNO E CONDIÇÃO INICIAL DO PROBLEMA

for i in range(1,N):
    u[i,N]=2*U_topo-u[i,N-1]

u_star=np.copy(u)

#VERIFICANDO CONDIÇÕES DE ESTABILIDADE DAS DIFERENÇAS FINITAS

if dt>=(dx) or dt>=(dy):
    print('CONDIÇÃO DE ESTABILIDADE 1 NÃO ATENDIDA')
    exit()
if dx>=(1/(Re**(1/2.0))) or dy>=(1/(Re**(1/2.0))):
    print('CONDIÇÃO DE ESTABILIDADE 2 NÃO ATENDIDA')
    exit()
if dt>=(1/4.0*Re*dx**2) or dt>=(1/4.0*Re*dy**2) :
    print('CONDIÇÃO DE ESTABILIDADE 3 NÃO ATENDIDA')
    exit()
#CÁLCULO DOS COEFICIENTES DO ESTÊNCIL DO PROBLEMA DA CAVIDADE BIDIMENSIONAL

r_e=np.zeros([N, N],float)
c_e=np.zeros([N, N],float)
l_e=np.zeros([N, N],float)
u_e=np.zeros([N, N],float)
d_e=np.zeros([N, N],float)
```

```
for i in range(0,N-1):
    for j in range(0,N):
        r_e[i,j]=1

for i in range(0,N):
    for j in range(0,N):
        if i==0:
            c_e[i,j]=-3
        elif i==N-1:
            c_e[i,j]=-3
        elif j==0:
            c_e[i,j]=-3
        elif j==N-1:
            c_e[i,j]=-3
        else:
            c_e[i,j]=-4
c_e[0,0]=-2
c_e[N-1,0]=-2
c_e[0,N-1]=-2
c_e[N-1,N-1]=-2

for i in range(1,N):
    for j in range(0,N):
        l_e[i,j]=1

for i in range(0,N):
    for j in range(0,N-1):
        u_e[i,j]=1

for i in range(0,N):
    for j in range(1,N):
        d_e[i,j]=1

##############################

start = timeit.time.time()
tempo=0.0
passo_de_tempo=0
iteracao_total=0

###########################
```

```
tol=10**(-6)

d=np.zeros([N+2,N+2],float)         #matriz gerada para armazenar variáveis do GCP
y=np.zeros([N+2,N+2],float)         #matriz gerada para armazenar variáveis do GCP
h=np.zeros([N+2,N+2],float)         #matriz gerada para armazenar variáveis do GCP
s=np.zeros([N+2,N++2])              #matriz gerada para armazenar variáveis do GCP

@njit
def calcula_pressao_gcpc_ssor(p,u_star, v_star, b, N, dx, dy, dt, tol,erro_graph,
                              R,d, y, h,passo_de_tempo,s):

    if passo_de_tempo==0:
        print('As funcões necessárias para rodar o algoritmo do GCP foram definidas aqui')
    # @njit
        def calcula_h(y,h,omega_ssor,N):
            for i in range(N):
                for j in range(N):
                    coef=(1/(omega_ssor*(2-omega_ssor)))
                    y[i,j]=(R[i,j]/coef-omega_ssor*l_e[i,j]*y[i-1,j]-omega_ssor*d_e[i,j]*y[i,j
            for i in range(N-1,-1,-1):
                for j in range(N-1,-1,-1):
                    h[i,j]=y[i,j]-omega_ssor/(c_e[i,j])*(r_e[i,j]*h[i+1,j]+u_e[i,j]*h[i,j+1])
            return h
        def calcula_R(R,alpha,s):
            for i in range(N):
                for j in range(N):
                    R[i,j]=R[i,j]-alpha*s[i,j]
            return R
        def calcula_s(s,r_e,c_e,l_e,u_e,d_e,d):
            for i in range(N):
                    for j in range(N):
                        s[i,j]=(r_e[i,j]*d[i+1,j]+c_e[i,j]*d[i,j]+l_e[i,j]*d[i-1,j]+
                                                            u_e[i,j]*d[i,j+1]+d_e[i,j]*d[i,j-1])
            return s
        def calcula_delta(R,h):
            delta=0
            for i in range(N+2):
                for j in range(N+2):
                    delta=delta+R[i,j]*h[i,j]
            return delta
        def calcula_alpha(delta,d,s):
```

```
        soma=0
        for i in range(N+2):
            for j in range(N+2):
                soma=soma+d[i,j]*s[i,j]
        alpha=delta/soma
        return alpha
    def calcula_p(p,alpha,d):
        for i in range(N+2):
            for j in range(N+2):
                p[i,j]=p[i,j]+alpha*d[i,j]
        return p
    def calcula_d(h,delta_new,delta,d):
        for i in range(N+2):
            for j in range(N+2):
                d[i,j]=h[i,j]+delta_new/delta*d[i,j]
        return d

erro=100
k=0
iteracao_pressao=0

for i in range(N):
    for j in range(N):
        R[i,j]=b[i,j]-(r_e[i,j]*p[i+1,j]+c_e[i,j]*p[i,j]+l_e[i,j]*p[i-1,j]+
                                    u_e[i,j]*p[i,j+1]+d_e[i,j]*p[i,j-1])
R_dot_R=np.sum(R*R)
soma=0
for i in range(N):
    for j in range(N):
        soma=soma+b[i,j]*b[i,j]
b_delta=soma
omega_ssor=1.95
h=calcula_h(y,h,omega_ssor,N)
d=np.copy(h)

while erro>tol:

    s=calcula_s(s,r_e,c_e,l_e,u_e,d_e,d)
    delta=calcula_delta(R,h)
    alpha=calcula_alpha(delta,d,s)
    p=calcula_p(p,alpha,d)
    R=calcula_R(R,alpha,s)
```

```
        h=calcula_h(y,h,omega_ssor,N)
        delta_new=calcula_delta(R,h)
        d=calcula_d(h,delta_new,delta,d)
        k=k+1
        # erro=np.max(np.abs(alpha*d))
        R_dot_R=np.sum(R*R)
        soma=0
        for i in range(N):
            for j in range(N):
                soma=soma+b[i,j]*b[i,j]
        b_delta=soma
        erro=np.sqrt(np.abs(R_dot_R/b_delta))
        iteracao_pressao=iteracao_pressao+1

        #GERANDO OS VETORES PARA OS GRAFICOS DE ERRO X Nº DE ITERACOES
        if passo_de_tempo==0:
            erro_graph=np.append(erro_graph,erro)

    #ATUALIZANDO CONTORNO DA PRESSAO

    for i in range(N):
        p[-1,i]=p[0,i]
        p[N,i]=p[N-1,i]
        p[i,-1]=p[i,0]
        p[i,N]=p[i,N-1]

    p[-1,-1]=p[0,0]
    p[-1,N]=p[0,N-1]
    p[N,-1]=p[N-1,0]
    p[N,N]=p[N-1,N-1]
    return p,iteracao_pressao,erro_graph
@njit
def calcula_u_star(N,u,v,u_star):
    for i in range(1,N):
        for j in range(0,N):
            C1=0.25*(v[i-1,j]+v[i,j]+v[i,j+1]+v[i-1,j+1])
            a1=u[i,j]*(u[i+1,j]-u[i-1,j])/(2*dx)+C1*(u[i,j+1]-u[i,j-1])/(2*dy)
            a2=(u[i+1,j]-2*u[i,j]+u[i-1,j])/(dx*dx)+(u[i,j+1]-2*u[i,j]+u[i,j-1])/(dy*dy)
            R=dt*(-a1+(1.0/Re)*a2)
            u_star[i,j]=u[i,j]+R
    #ATUALIZANDO CONTORNO DAS VELOCIDADES STAR
    for i in range(1,N):
```

```
        u_star[i,-1]=-u_star[i,0]

    for i in range(1,N):
        u_star[i,N]=2*U_topo-u_star[i,N-1]

    return u_star
@njit
def calcula_v_star(N,u,v,v_star):
    for i in range(0,N):
        for j in range(1,N):
            C2=0.25*(u[i,j]+u[i+1,j]+u[i+1,j-1]+u[i,j-1])
            a1=C2*(v[i+1,j]-v[i-1,j])/(2*dx)+v[i,j]*(v[i,j+1]-v[i,j-1])/(2*dy)
            a2=(v[i+1,j]-2*v[i,j]+v[i-1,j])/(dx*dx)+(v[i,j+1]-2*v[i,j]+v[i,j-1])/(dy*dy)
            R=dt*(-a1+(1.0/Re)*a2)
            v_star[i,j]=v[i,j]+R
    #ATUALIZANDO CONTORNO DAS VELOCIDADES STAR
    for j in range(1,N):
        v_star[-1,j]=-v_star[0,j]

    for j in range(1,N):
        v_star[N,j]=-v_star[N-1,j]
    return v_star

@njit
def calcula_u_e_v(u,v,u_star,v_star, p,dt,dx,dy,N):
    for i in range(0,N):
        for j in range(0,N):
            u[i,j]=u_star[i,j]-dt*(p[i,j]-p[i-1,j])/(dx)
            v[i,j]=v_star[i,j]-dt*(p[i,j]-p[i,j-1])/(dy)
    #ATUALIZANDO CONTORNO DAS VELOCIDADES
    for i in range(1,N):
        u[i,-1]=-u[i,0]
    for i in range(1,N):
        u[i,N]=2*U_topo-u[i,N-1]

    for j in range(1,N):
        v[-1,j]=-v[0,j]
    for j in range(1,N):
        v[N,j]=-v[N-1,j]
    return u,v

@njit
```

```
def calcula_b(N,dt,dx,u_star,v_star,b):
    for i in range(0,N):
        for j in range(0,N):
            b[i,j]=(dx/dt)*(u_star[i+1,j]-u_star[i,j]+v_star[i,j+1]-v_star[i,j])
    return b

###########################

#ENTRANDO NO LOOP PRINCIPAL
while tempo<tempo_final:

    calcula_u_star(N,u,v,u_star)
    calcula_v_star(N,u,v,v_star)

    ##############################

    #CALCULANDO O VETOR B
    b=calcula_b(N, dt, dx, u_star, v_star,b)

    #CALCULANDO A PRESSÃO MÉTODO DO GRADIENTE CONJUGADO PRECONDICIONADO
    tol=10**(-6)
    if passo_de_tempo==0:
        erro_graph=np.zeros([0],float)

# Nas linhas abaixo, a variável funcao_calcula recebe os valores de pressao,
# iteracões feitas naquele passo de tempo e os valores de erro para cada
# iteração no primeiro passo de tempo para gerar gráficos posteriormente.
# nas linhas abaixo, cada valor da variável funcao_calcula é passada para uma
# variável adequada, como a pressão é passada para a variável p, por exemplo.

    funcao_calcula=calcula_pressao_gcpc_ssor(p,u_star, v_star, b, N, dx, dy, dt,tol,
                                            erro_graph,R,d, y, h,passo_de_tempo,s)
    p=funcao_calcula[0]
    iteracao_pressao=funcao_calcula[1]
    iteracao_total=iteracao_total+iteracao_pressao
    if passo_de_tempo==0:
        erro_graph=funcao_calcula[2]
        iteracao_primeiro_tempo=iteracao_pressao

    #CALCULANDO A NOVA VELOCIDADE

    u_e_v=calcula_u_e_v(u,v,u_star,v_star, p,dt,dx,dy,N)
```

```
    u=u_e_v[0]
    v=u_e_v[1]

    tempo=tempo+dt
    itera_por_passo_graph[passo_de_tempo]=iteracao_pressao
    passo_de_tempo=passo_de_tempo+1
    print('No passo de tempo ',passo_de_tempo,'foram realizadas ',
          iteracao_pressao, 'iteracoes na pressao')
finish = timeit.time.time()
tempo_execucao=finish-start
print('Tempo de execução:',finish-start)
print('Numero total de iteracoes: ',iteracao_total)

#SALVANDO INFORMAÇÕES EM FORMATO TXT
info=np.array([Re,N,tol,dt,tempo_final,tempo_execucao,iteracao_primeiro_tempo,
               iteracao_total])
np.savetxt('info_gradiente_conjugado_precondicionado_ssor.txt', info,fmt='%1.6f' ,
           header='Re,N,tol,passo_de_tempo,tempo_final,tempo_execucao,'
           'iteracao_primeiro_tempo,iteracao_total')
np.savetxt('erro_graph_gradiente_conjugado_precondicionado_ssor.txt',erro_graph, fmt='%1.7f')
np.savetxt('campo_pressao_gradiente_conjugado_precondicionado_ssor.txt', p, fmt='%1.7f')
np.savetxt('campo_u_gradiente_conjugado_precondicionado_ssor.txt', u, fmt='%1.7f')
np.savetxt('campo_v_gradiente_conjugado_precondicionado_ssor.txt', v, fmt='%1.7f')
np.savetxt('itera_por_passo_graph_gradiente_conjugado_precondicionado_ssor.txt',
           itera_por_passo_graph,fmt='%1.1f')
```

## II.6 Código do método Multigrid (MG-GC-2M)

```
import numpy as np
import timeit
from numba import njit

#DEFINIÇÃO DAS PRINCIPAIS VARIÁVEIS DO PROBLEMA

N=128                    #índice do último ponto na discretização
L=1.0                    #largura da cavidade
H=L                      #altura da cavidade
Re=3000                   #número de Reynolds
tempo_final=0.05          #tempo final da simulação
U_topo=1.0               #velocidade da parede superior da cavidade
```

```
dx=L/N
dy=dx
dt=0.0005

u_star=np.zeros([N+1,N+2],float)        #compontente horizontal de vel. estrela
v_star=np.zeros([N+2,N+1],float)        #componente vertical de vel. estrela
p=np.zeros([N+2,N+2],float)             #pressão
p_new=np.zeros([N+2,N+2],float)         #vetor pressão auxiliar 1
p_aux=np.zeros([N+2,N+2],float)         #vetor pressão auxiliar 2
u=np.zeros([N+1,N+2],float)             #componente horizontal de velocidade
v=np.zeros([N+2,N+1],float)             #componente vertical de velocidade
b=np.zeros([N,N],float)                 #vetor resposta do sistema Ax=b
itera_por_passo_graph=np.zeros(100)     #vetor gerado para criar gráficos
erro_graph=np.zeros([0],float)          #vetor gerado para criar gráficos
Res=np.zeros([N,N],float)
R=np.zeros([N+2,N+2],float)             #vetor resíduo
Res_2h=np.zeros([N//2+1,N//2+1],float)
R_2h=np.zeros([(N+2)//2+1,(N+2)//2+1],float)            #vetor resíduo
Res_aux=np.zeros([(N)//2+1,(N)//2+1],float)
e=np.zeros([(N)+2,(N)+2],float)
e_aux=np.zeros([(N),(N)], float)
e_new=np.zeros([(N)//2+1,(N)//2+1],float)
e_2h=np.zeros([(N)//2+2,(N)//2+2],float)
#DEFINIÇÃO DAS CONDIÇÕES DE CONTORNO E CONDIÇÃO INICIAL DO PROBLEMA

for i in range(1,N):
    u[i,N]=2*U_topo-u[i,N-1]

u_star=np.copy(u)

#VERIFICANDO CONDIÇÕES DE ESTABILIDADE DAS DIFERENÇAS FINITAS

if dt>=(dx) or dt>=(dy):
    print('CONDIÇÃO DE ESTABILIDADE 1 NÃO ATENDIDA')
    exit()
if dx>=(1/(Re**(1/2.0))) or dy>=(1/(Re**(1/2.0))):
    print('CONDIÇÃO DE ESTABILIDADE 2 NÃO ATENDIDA')
    exit()
if dt>=(1/4.0*Re*dx**2) or dt>=(1/4.0*Re*dy**2) :
    print('CONDIÇÃO DE ESTABILIDADE 3 NÃO ATENDIDA')
    exit()
#CÁLCULO DOS COEFICIENTES DO ESTÊNCIL OMEGA_h DO PROBLEMA DA CAVIDADE BIDIMENSIONAL
```

```
r_e=np.zeros([N, N],float)
c_e=np.zeros([N, N],float)
l_e=np.zeros([N, N],float)
u_e=np.zeros([N, N],float)
d_e=np.zeros([N, N],float)

for i in range(0,N-1):
    for j in range(0,N):
        r_e[i,j]=1

for i in range(0,N):
    for j in range(0,N):
        if i==0:
            c_e[i,j]=-3
        elif i==N-1:
            c_e[i,j]=-3
        elif j==0:
            c_e[i,j]=-3
        elif j==N-1:
            c_e[i,j]=-3
        else:
            c_e[i,j]=-4
c_e[0,0]=-2
c_e[N-1,0]=-2
c_e[0,N-1]=-2
c_e[N-1,N-1]=-2

for i in range(1,N):
    for j in range(0,N):
        l_e[i,j]=1

for i in range(0,N):
    for j in range(0,N-1):
        u_e[i,j]=1

for i in range(0,N):
    for j in range(1,N):
        d_e[i,j]=1

#CÁLCULO DOS COEFICIENTES DO ESTÊNCIL OMEGA_2h DO PROBLEMA DA CAVIDADE BIDIMENSIONAL
r_e_2h=np.zeros([N//2, N//2],float)
```

```
c_e_2h=np.zeros([N//2, N//2],float)
l_e_2h=np.zeros([N//2, N//2],float)
u_e_2h=np.zeros([N//2, N//2],float)
d_e_2h=np.zeros([N//2, N//2],float)

for i in range(0,N//2-1):
    for j in range(0,N//2):
        r_e_2h[i,j]=1

for i in range(0,N//2):
    for j in range(0,N//2):
        if i==0:
            c_e_2h[i,j]=-3
        elif i==N//2-1:
            c_e_2h[i,j]=-3
        elif j==0:
            c_e_2h[i,j]=-3
        elif j==N//2-1:
            c_e_2h[i,j]=-3
        else:
            c_e_2h[i,j]=-4
c_e_2h[0,0]=-2
c_e_2h[N//2-1,0]=-2
c_e_2h[0,N//2-1]=-2
c_e_2h[N//2-1,N//2-1]=-2

for i in range(1,N//2):
    for j in range(0,N//2):
        l_e_2h[i,j]=1

for i in range(0,N//2):
    for j in range(0,N//2-1):
        u_e_2h[i,j]=1

for i in range(0,N//2):
    for j in range(1,N//2):
        d_e_2h[i,j]=1



##############################
```

```
##############################

start = timeit.time.time()
tempo=0.0
passo_de_tempo=0
iteracao_total=0

########################

tol=10**(-6)

d=np.zeros([N+2,N+2],float)          #matriz gerada para armazenar variáveis do GCP
y=np.zeros([N+2,N+2],float)          #matriz gerada para armazenar variáveis do GCP
h=np.zeros([N+2,N+2],float)          #matriz gerada para armazenar variáveis do GCP
s=np.zeros([N+2,N++2])               #matriz gerada para armazenar variáveis do GCP

d_2h=np.zeros([(N+2)//2+1,(N+2)//2+1],float)         #matriz gerada para armazenar variáveis d
y_2h=np.zeros([(N+2)//2+1,(N+2)//2+1],float)         #matriz gerada para armazenar variáveis d
h_2h=np.zeros([(N+2)//2+1,(N+2)//2+1],float)         #matriz gerada para armazenar variáveis d
s_2h=np.zeros([(N+2)//2+1,(N+2)//2+1],float)              #matriz gerada para armazenar variáv

#DEFININDO FUNÇÕES



@njit
def calcula_u_star(N,u,v,u_star):
    for i in range(1,N):
        for j in range(0,N):
            C1=0.25*(v[i-1,j]+v[i,j]+v[i,j+1]+v[i-1,j+1])
            a1=u[i,j]*(u[i+1,j]-u[i-1,j])/(2*dx)+C1*(u[i,j+1]-u[i,j-1])/(2*dy)
            a2=(u[i+1,j]-2*u[i,j]+u[i-1,j])/(dx*dx)+(u[i,j+1]-2*u[i,j]+u[i,j-1])/(dy*dy)
            R=dt*(-a1+(1.0/Re)*a2)
            u_star[i,j]=u[i,j]+R
    #ATUALIZANDO CONTORNO DAS VELOCIDADES STAR
    for i in range(1,N):
        u_star[i,-1]=-u_star[i,0]

    for i in range(1,N):
        u_star[i,N]=2*U_topo-u_star[i,N-1]
```

```
    return u_star
@njit
def calcula_v_star(N,u,v,v_star):
    for i in range(0,N):
        for j in range(1,N):
            C2=0.25*(u[i,j]+u[i+1,j]+u[i+1,j-1]+u[i,j-1])
            a1=C2*(v[i+1,j]-v[i-1,j])/(2*dx)+v[i,j]*(v[i,j+1]-v[i,j-1])/(2*dy)
            a2=(v[i+1,j]-2*v[i,j]+v[i-1,j])/(dx*dx)+(v[i,j+1]-2*v[i,j]+v[i,j-1])/(dy*dy)
            R=dt*(-a1+(1.0/Re)*a2)
            v_star[i,j]=v[i,j]+R
    #ATUALIZANDO CONTORNO DAS VELOCIDADES STAR
    for j in range(1,N):
        v_star[-1,j]=-v_star[0,j]

    for j in range(1,N):
        v_star[N,j]=-v_star[N-1,j]
    return v_star

@njit
def calcula_u_e_v(u,v,u_star,v_star, p,dt,dx,dy,N):
    for i in range(0,N):
        for j in range(0,N):
            u[i,j]=u_star[i,j]-dt*(p[i,j]-p[i-1,j])/(dx)
            v[i,j]=v_star[i,j]-dt*(p[i,j]-p[i,j-1])/(dy)
    #ATUALIZANDO CONTORNO DAS VELOCIDADES
    for i in range(1,N):
        u[i,-1]=-u[i,0]
    for i in range(1,N):
        u[i,N]=2*U_topo-u[i,N-1]

    for j in range(1,N):
        v[-1,j]=-v[0,j]
    for j in range(1,N):
        v[N,j]=-v[N-1,j]
    return u,v

@njit
def calcula_b(N,dt,dx,u_star,v_star,b):
    for i in range(0,N):
        for j in range(0,N):
            b[i,j]=(dx/dt)*(u_star[i+1,j]-u_star[i,j]+v_star[i,j+1]-v_star[i,j])
    return b
```

```
###########################

@njit
def calcula_pressao_multigrid_2h(p,u_star, v_star, b, N, dx, dy, dt,tol,
                                  erro_graph,p_new,iteracao_total,
                                  e,e_aux,e_new,Res,Res_aux,p_aux,h,s,R,d,y,Res_2h,e_2h,
                                  h_2h,s_2h,R_2h,d_2h,y_2h,passo_de_tempo):
    if passo_de_tempo==0:
        print("Funções definidas aqui")
        def restricao(Res,Res_2h,Res_aux,N):
            for i in range(0,(N)//2):
                for j in range(0,(N)//2):
                    Res_aux[i,j]=(Res[2*i,2*j]+Res[2*i+1,2*j]+Res[2*i,2*j+1]+Res[2*i+1,2*j+1])
            for i in range(0,(N)//2):
                for j in range(0,(N)//2):
                    Res_2h[i,j]=Res_aux[i,j]
            return Res_2h
        def interpolacao(e,e_2h,e_aux,N):
            for i in range(0,(N)//2):
                for j in range(0,(N)//2):
                    e_aux[2*i,2*j]=(9*e_2h[i,j]+3*e_2h[i-1,j]+3*e_2h[i,j-1]+e_2h[i-1,j-1])/16
                    e_aux[2*i,2*j+1]=(9*e_2h[i,j]+3*e_2h[i-1,j]+3*e_2h[i,j+1]+e_2h[i-1,j+1])/1
                    e_aux[2*i+1,2*j]=(9*e_2h[i,j]+3*e_2h[i,j-1]+3*e_2h[i+1,j]+e_2h[i+1,j-1])/1
                    e_aux[2*i+1,2*j+1]=(9*e_2h[i,j]+3*e_2h[i,j+1]+3*e_2h[i+1,j]+e_2h[i+1,j+1])
            for i in range(0,(N)):
                for j in range(0,(N)):
                    e[i,j]=e_aux[i,j]
            return e

        def calcula_equacao_do_residuo(e_2h, u_star, v_star,Res_2h, N, dx, dy, dt, tol,
                                       erro_graph,e_new,R_2h,h_2h,s_2h,d_2h,y_2h,R,h,s,d,y,e):

            if passo_de_tempo==0:

                def calcula_R(R,alpha,s,N):
                    for i in range(N):
                        for j in range(N):
                            R[i,j]=R[i,j]-alpha*s[i,j]
                    return R
                # @njit
                def calcula_s(s,r_e,c_e,l_e,u_e,d_e,d,N):
                    for i in range(N):
```

```
                for j in range(N):
                    s[i,j]=(r_e[i,j]*d[i+1,j]+c_e[i,j]*d[i,j]+l_e[i,j]*d[i-1,j]+
                                                            u_e[i,j]*d[i,j+1]+d_e[i,j]*d[i,j-1
            return s
        # @njit
        def calcula_delta(R,N):
            delta=0
            for i in range(N):
                for j in range(N):
                    delta=delta+R[i,j]*R[i,j]
            return delta
        # @njit
        def calcula_alpha(delta,d,s,N):
            soma=0
            for i in range(N):
                for j in range(N):
                    soma=soma+d[i,j]*s[i,j]
            alpha=delta/soma
            return alpha
        # @njit
        def calcula_p(p,alpha,d,N):
            for i in range(N):
                for j in range(N):
                    p[i,j]=p[i,j]+alpha*d[i,j]
            return p
        # @njit
        def calcula_d(R,delta_new,delta,d,N):
            for i in range(N):
                for j in range(N):
                    d[i,j]=R[i,j]+delta_new/delta*d[i,j]
            return d
    erro=100
    k=0
    iteracao_pressao=0
    ni_1=3
    cont=1
    #zerando chute inicial
    for i in range(N//2+2):
        for j in range(N//2+2):
            e_2h[i,j]=0
    for i in range(N//2):
        for j in range(N//2):
```

```
            R[i,j]=Res_2h[i,j]-(r_e_2h[i,j]*e_2h[i+1,j]+c_e_2h[i,j]*e_2h[i,j]+l_e_2h[i
                                        u_e_2h[i,j]*e_2h[i,j+1]+d_e_2h[i,j]*e_2h[i,j-1

    d=np.copy(R)

    while cont<=ni_1:
    # while erro>(tol):
        s=calcula_s(s,r_e_2h,c_e_2h,l_e_2h,u_e_2h,d_e_2h,d,N//2)
        delta=calcula_delta(R,N//2)
        alpha=calcula_alpha(delta,d,s,N//2)
        e_2h=calcula_p(e_2h,alpha,d,N//2)
        R=calcula_R(R,alpha,s,N//2)
        delta_new=calcula_delta(R,N//2)
        d=calcula_d(R,delta_new,delta,d,N//2)
        k=k+1
        erro=np.max(np.abs(alpha*d))
        iteracao_pressao=iteracao_pressao+1
        # GERANDO OS VETORES PARA OS GRAFICOS DE ERRO X Nº DE ITERACOES
        if passo_de_tempo==0:
            erro_graph=np.append(erro_graph,erro)
        cont=cont+1

    for i in range(N//2):
        e_2h[-1,i]=e_2h[0,i]
        e_2h[N//2,i]=e_2h[N//2-1,i]
        e_2h[i,-1]=e_2h[i,0]
        e_2h[i,N//2]=e_2h[i,N//2-1]

    e_2h[-1,-1]=e_2h[0,0]
    e_2h[-1,N//2]=e_2h[0,N//2-1]
    e_2h[N//2,-1]=e_2h[N//2-1,0]
    e_2h[N//2,N//2]=e_2h[N//2-1,N//2-1]

    return e_2h,iteracao_pressao,erro_graph,erro

def calcula_pressao_gc(p, u_star, v_star,b, N, dx, dy, dt, tol,
                              erro_graph,p_new,sentido,R,h,s,Res,d,y):

    if passo_de_tempo==0:
        def calcula_R(R,alpha,s,N):
            for i in range(N):
                for j in range(N):
```

```
                R[i,j]=R[i,j]-alpha*s[i,j]
        return R
    # @njit
    def calcula_s(s,r_e,c_e,l_e,u_e,d_e,d,N):
        for i in range(N):
                for j in range(N):
                    s[i,j]=(r_e[i,j]*d[i+1,j]+c_e[i,j]*d[i,j]+l_e[i,j]*d[i-1,j]+
                                                        u_e[i,j]*d[i,j+1]+d_e[i,j]*d[i,j-1
        return s
    # @njit
    def calcula_delta(R,N):
        delta=0
        for i in range(N):
            for j in range(N):
                delta=delta+R[i,j]*R[i,j]
        return delta
    # @njit
    def calcula_alpha(delta,d,s,N):
        soma=0
        for i in range(N):
            for j in range(N):
                soma=soma+d[i,j]*s[i,j]
        alpha=delta/soma
        return alpha
    # @njit
    def calcula_p(p,alpha,d,N):
        for i in range(N):
            for j in range(N):
                p[i,j]=p[i,j]+alpha*d[i,j]
        return p
    # @njit
    def calcula_d(R,delta_new,delta,d,N):
        for i in range(N):
            for j in range(N):
                d[i,j]=R[i,j]+delta_new/delta*d[i,j]
        return d

b_delta=0
for i in range(N):
    for j in range(N):
        b_delta=b_delta+b[i,j]*b[i,j]
```

```
erro=100
k=0
iteracao_pressao=0
ni_1=5
cont=0
for i in range(N):
    for j in range(N):
        R[i,j]=b[i,j]-(r_e[i,j]*p[i+1,j]+c_e[i,j]*p[i,j]+l_e[i,j]*p[i-1,j]+
                                            u_e[i,j]*p[i,j+1]+d_e[i,j]*p[i,j-1])

d=np.copy(R)

if sentido==0:
    while cont<=ni_1:
        s=calcula_s(s,r_e,c_e,l_e,u_e,d_e,d,N)
        delta=calcula_delta(R,N)
        alpha=calcula_alpha(delta,d,s,N)
        p=calcula_p(p,alpha,d,N)
        R=calcula_R(R,alpha,s,N)
        delta_new=calcula_delta(R,N)
        d=calcula_d(R,delta_new,delta,d,N)
        k=k+1
        erro=np.max(np.abs(alpha*d))
        iteracao_pressao=iteracao_pressao+1

        #GERANDO OS VETORES PARA OS GRAFICOS DE ERRO X Nº DE ITERACOES
        if passo_de_tempo==0:
            erro_graph=np.append(erro_graph,erro)
            # print(erro_graph)
        cont=cont+1

elif sentido == 1:
    delta=calcula_delta(R,N)
    soma=0
    for i in range(N):
        for j in range(N):
            soma=soma+b[i,j]*b[i,j]
    b_delta=soma
    while erro>tol:
        s=calcula_s(s,r_e,c_e,l_e,u_e,d_e,d,N)
        delta=calcula_delta(R,N)
        alpha=calcula_alpha(delta,d,s,N)
```

```
                p=calcula_p(p,alpha,d,N)
                R=calcula_R(R,alpha,s,N)
                delta_new=calcula_delta(R,N)
                d=calcula_d(R,delta_new,delta,d,N)
                k=k+1
                erro=np.sqrt(delta/b_delta)
                iteracao_pressao=iteracao_pressao+1

                #GERANDO OS VETORES PARA OS GRAFICOS DE ERRO X Nº DE ITERACOES
                if passo_de_tempo==0:
                    erro_graph=np.append(erro_graph,erro)
                cont=cont+1

        # Res=np.copy(R)
        for i in range(N):
            for j in range(N):
                Res[i,j]=b[i,j]-(r_e[i,j]*p[i+1,j]+c_e[i,j]*p[i,j]+l_e[i,j]*p[i-1,j]+
                                            u_e[i,j]*p[i,j+1]+d_e[i,j]*p[i,j-1])

        return p,iteracao_pressao,erro_graph,erro,Res


erro=100
ciclos=0
sentido=0
funcao_calcula=calcula_pressao_gc(p,u_star, v_star, b, N, dx, dy, dt,
                                           tol,erro_graph,p_new,sentido,R,h,s,Res,d,y)
p=funcao_calcula[0]
iteracao_pressao=funcao_calcula[1]
# print(iteracao_pressao)
erro=funcao_calcula[3]
if passo_de_tempo==0:
    erro_graph=funcao_calcula[2]
Res=funcao_calcula[4]



Res_2h=restricao(Res,Res_2h,Res_aux,N)

funcao_calcula=calcula_equacao_do_residuo(e_2h,u_star, v_star, Res_2h, N, dx, dy, dt,
                                           tol,erro_graph,e_new,R_2h,h_2h,s_2h,d_2h,y_2h,
                                           R,h,s,d,y,e)
e_2h=funcao_calcula[0]
```

```
    if passo_de_tempo==0:
        erro_graph=np.append(erro_graph,funcao_calcula[2])
    iteracao_pressao=funcao_calcula[1]+iteracao_pressao
    sentido=1
    e=interpolacao(e,e_2h,e_aux,N)

    for i in range(0,N):
        for j in range(0,N):
            p_aux[i,j]=p[i,j]+e[i,j]
            p[i,j]=p_aux[i,j]

    funcao_calcula=calcula_pressao_gc(p,u_star, v_star, b, N, dx, dy, dt,
                                          tol,erro_graph,p_new,sentido,R,h,s,Res,d,y)
    p=funcao_calcula[0]
    iteracao_pressao=funcao_calcula[1]+iteracao_pressao
    if passo_de_tempo==0:
        erro_graph=np.append(erro_graph,funcao_calcula[2])
    erro=funcao_calcula[3]
    iteracao_total=iteracao_total+iteracao_pressao
    ciclos=ciclos+1
    #ATUALIZANDO CONTORNO DA PRESSAO

    for i in range(N):
        p[-1,i]=p[0,i]
        p[N,i]=p[N-1,i]
        p[i,-1]=p[i,0]
        p[i,N]=p[i,N-1]

    p[-1,-1]=p[0,0]
    p[-1,N]=p[0,N-1]
    p[N,-1]=p[N-1,0]
    p[N,N]=p[N-1,N-1]
    return p,iteracao_pressao,erro_graph,erro,ciclos,iteracao_total

#ENTRANDO NO LOOP PRINCIPAL

while tempo<tempo_final:

    calcula_u_star(N,u,v,u_star)
    calcula_v_star(N,u,v,v_star)
```

```
    #CALCULANDO O VETOR B

    b=calcula_b(N, dt, dx, u_star, v_star,b)

    #CALCULANDO A PRESSÃO MÉTODO DE GAUSS-SEIDEL
    tol=10**(-6)
    if passo_de_tempo==0:
        erro_graph=np.zeros([0],float)

    funcao_calcula_p=calcula_pressao_multigrid_2h(p,u_star, v_star, b, N, dx, dy, dt,tol,
                                erro_graph,p_new,iteracao_total,
                                e,e_aux,e_new,Res,Res_aux,p_aux,h,s,R,d,y,Res_2h,e_2h,
                                h_2h,s_2h,R_2h,d_2h,y_2h,passo_de_tempo)
    p=funcao_calcula_p[0]
    iteracao_pressao=funcao_calcula_p[1]
    erro=funcao_calcula_p[3]
    ciclos=funcao_calcula_p[4]
    iteracao_total=funcao_calcula_p[5]

    if passo_de_tempo==0:
        erro_graph=funcao_calcula_p[2]

        iteracao_primeiro_tempo=iteracao_pressao*ciclos


    #CALCULANDO A NOVA VELOCIDADE

    u_e_v=calcula_u_e_v(u,v,u_star,v_star, p,dt,dx,dy,N)
    u=u_e_v[0]
    v=u_e_v[1]

    tempo=tempo+dt
    itera_por_passo_graph[passo_de_tempo]=iteracao_pressao
    passo_de_tempo=passo_de_tempo+1

    print('No passo de tempo ',passo_de_tempo,'foram realizadas ',
           ciclos*iteracao_pressao, 'iteracoes na pressao')
finish = timeit.time.time()
tempo_execucao=finish-start
print('Tempo de execução:',finish-start)
print('Numero total de iteracoes: ',iteracao_total)
```

```
#SALVANDO INFORMAÇÕES EM FORMATO TXT
info=np.array([Re,N,tol,dt,tempo_final,tempo_execucao,iteracao_primeiro_tempo,
               iteracao_total])
np.savetxt('info_multigrid_GC.txt', info,fmt='%1.6f' , header=
           'Re,N,tol,passo_de_tempo,tempo_final,tempo_execucao,'
           'iteracao_primeiro_tempo,iteracao_total')
np.savetxt('erro_graph_multigrid_GC.txt',erro_graph, fmt='%1.7f')
np.savetxt('campo_pressao_multigrid_GC.txt', p, fmt='%1.7f')
np.savetxt('campo_u_multigrid_GC.txt', u, fmt='%1.7f')
np.savetxt('campo_v_multigrid_GC.txt', v, fmt='%1.7f')
np.savetxt('itera_por_passo_multigrid_GC.txt', itera_por_passo_graph,fmt='%1.1f')
```