RelatÃ§rio de estudos individual

PÃľtterson VinÃŋcius Pramiu

21 de abril de 2017

Sumário

Capítulo 1

IntroduÃğÃčo

Este relatÃșrio objetiva registrar de forma detalhada e sistemÃątica as atividades de pesquisa e estudos relacionados Ãă temÃątica de proposta de tese do autor. Nele serÃčo revistos alguns conceitos fundamentais de mecÃćnica de fluidos, mÃľtodos numÃľricos em equaÃğÃțes diferenciais, instalaÃğÃčo e configuraÃğÃčo de bibliotecas para implementaÃğÃčo computacional como PETSc e MPI, entre outros. Exemplos da resoluÃğÃčo de sistemas de equaÃğÃțes lineares e da soluÃğÃčo de algumas EDPs como a equaÃğÃčo de Laplace, de Poisson, da equaÃğÃčo do calor transiente e da equaÃğÃčo de Stokes serÃčo resolvidos detalhadamente em diferenÃğas finitas ou volumes finitos, empregando malhas estruturadas e em programaÃğÃčo serial ou paralela. A linguagem selecionada para implementaÃğÃčo dos cÃșdgidos fonte Ãľ a linguagem C, e alguns exemplos tambÃľm serÃčo ilustrados e implementados em Scilab. Com a realizaÃğÃčo destes exemplos espera-se obter os requisitos necessÃąrios para compreensÃčo de trabalhos e implementaÃğÃčo de cÃșdigo computacional envolvendo simulaÃğÃțes em escoamentos turbulentos empregando o mÃľtodo de SimulaÃğÃčo de Grandes Escalas - *Large Eddy Simulation*.

1.1 Motivação

Dentre as ferramentas e abordagens amplamente utilizadas na engenharia contemporânea em pesquisa e desenvolvimento de obras, máquinas e processos, destacam-se aquelas associadas às simulações numérico-computacionais, que decorrem do vertiginoso desenvolvimento de tecnologias da computação e de métodos numéricos sofisticados. Uma simulação numérico-computacional pode ser definida como aquela resultante da implementação em linguagem de programação de modelos matemáticos e formulações numéricas que representa fenômenos ou eventos de interesse. São, geralmente, soluções de sistemas de equações diferenciais, sistemas de equações estocásticas ou de equações algébricas, entre outros. Com custo relativo inferior à realização de experimentos reais ou de modelos em escala, tais simulações viabilizam o estudo de diversas combinações de condições de

utilização, de modo a auxiliar na especificação de melhorias no projeto, operação e manutenção de equipamentos, estruturas e processos.

Atualmente existem diversas ferramentas computacionais e softwares desenvolvidos especificamente para realização de simulações numérico-computacionais. Softwares ou plataformas como o Ansys, Comsol, OpenFoam, MultiSim, Proteus, DualSPHyics, entre outros, são exemplos de tais ferramentas que são bem sucedidas nos seus campos, sendo amplamente utilizadas na engenharia. Esses softwares, em síntese, implementam modelos e formulações que representam desde o comportamento de determinadas estruturas, o escoamento de fluidos, as trocas de calor, as correntes em circuitos elétricos, entre outros. Além disso, em alguns deles a exemplo do Ansys, OpenFoam e DualSPHysics, é possível a implementação explicita de modelos que não estejam previamente disponibilizados, por meio linguagens de programação próprias ou de outras usuais como C/C++, Fortran e XML.

Uma vasta área das Ciências Aplicadas que emprega simulação é a Dinâmica de Fluidos Computacional, CFD, que trata de processos e fenômenos que apresentam escoamento. As aplicações incluem o cálculo das forças e momentos em aeronaves, a determinação da taxa de fluxo de massa de petróleo em gasodutos, a previsão de condições meteorológicas, a modelagem de detonação de armas nucleares, o escoamento de fluidos em canais e abertos ou condutos forçados. Para muitas delas, um resultado importante das simulações envolvem a transferência de energia, pressão ou velocidade para estruturas que estão em contato com o fluido. Esta análise viabiliza a quantificação dos esforços solicitados pelo escoamento.

Além das equações do momentum, algumas aplicações que envolvem o transporte de quantidades requerem equações específicas para modelar e resolver tal transporte. Exemplos são as equações para a energia, momento, tensões, fluxos e outras quantidades envolvidas no escoamento. Outras quantidades também podem ser transportadas pelo fluido, mesmo que não estejam totalmente dispersas no fluido, a exemplo de partículas suspensas ou arrastadas pelo escoamento. Esse caso, e vários outros usuais nas Engenharias, requerem modelos específicos para a avaliação do transporte das partículas no meio fluido ou gasoso, sendo então necessária a modelagem de diferentes fases, incluindo a fase particulada. Os modelos matemáticos presentes em tais situações provêm formulação às diferentes forças de interação fluido-partícula e para o cálculo das trajetórias das partículas no meio contínuo ou fluido circundante.

A metodologia da CFD e essas concepções podem ser empregadas para contribuir à formulação e desenvolvimento de acuradas simulações à avaliação e predição do desgaste abrasivo-erosivo da superfície de materiais. Tal situação é especificamente importante à Usina Hidroelétrica de Itaipu, cuja superfície das lajes de concreto do vertedouro apresenta certo desgaste superficial decorrente dos mais de 30 anos de operação, requerendo intervenções e manutenções curativas. Dada a atual impossibilidade de predizer matematicamente a evolução espaço-temporal dos danos na referida estrutura, a CFD pode contribuir a tal análise. È, pois, relevante empregá-la para investigar o efeito de distintos mecanismos de desgaste, sobretudo aqueles causados por escoamentos de água com e

sem meio particulado.

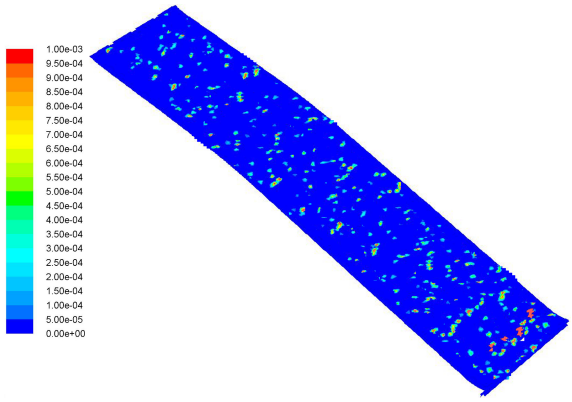
Nesse tipo de estudo além dos modelos hidrodinâmico, de transporte e de partículas, é indispensável à formulação e o emprego de modelagem constitutiva para o material que compõe a estrutura das calhas. Desafortunadamente a complexa natureza do concreto, resultante da sua heterogeneidade, da resposta assimétrica à tração e à compressão, das mudanças nas propriedades mecânicas devidas à evolução da microfissuração, do efeito de escala, entre outros fatores, têm dificultado a formulação de um modelo constitutivo geral e completo para o concreto.

Conhecidos ou calculados os forçantes, as solicitações geradas pelo fluido ou partículas transportadas são inseridas no modelo constitutivo do material, que viabiliza avaliar estados de tensões da estrutura podendo, assim, indicar regiões suscetíveis à ruptura e outros aspectos importantes na concepção do projeto e operação da estrutura. Atualmente um dos modelos constitutivos promissores à modelagem do concreto é o Riedel-Hiermaier-Thoma, RHT, que representa o comportamento de materiais frágeis de acordo com critérios de falha superficial.

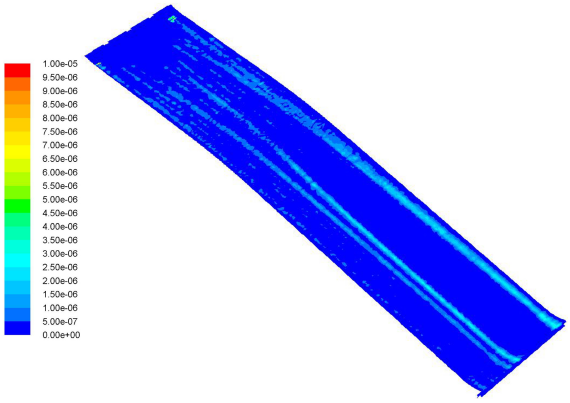
Modelagens via CFD requerem as equações de Navier-Stokes completas, isto é, as equações da continuidade, momentum e transporte. Indispensável nessa abordagem é, então, conhecer e resolver a turbulência, que pode ser caracterizada pela mistura não linear e caótica das partículas e elementos fluidos. Mais especificamente, um campo de fluxo turbulento é caracterizado por flutuações de velocidade em todas as direções e escalas.

Processos de transporte de substâncias num dado ambiente podem ser grandemente acelerados pela presença de turbulência. Com efeito, se considerada apenas o processo difusivo molecular, a propagação de certo material poderia demorar dias para se completar. Mas as forças empuxo ou as correntes de escoamentos forçantes aceleram este processo para minutos, sobretudo devido a ação da turbulência. Assim, dada a relevância da sua modelagem, existem diversas formulações para sua representação, destacando-se à guisa de informação o modelo de zero equações, o modelo $k - \epsilon$, o modelo $k - \omega$, o modelo SST (Shear Stress Transpor), a modelagem RANS (Reynolds Avarage Navier Stokes) e a modelagem TLES (Temporal Large Eddy Scale).

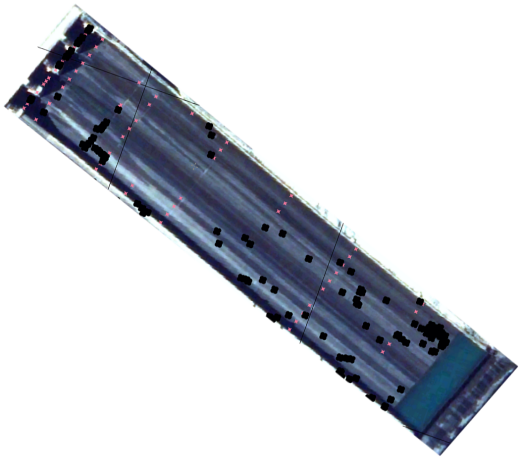
Estudos e resultados preliminares obtidos em Projeto já finalizado no CEASB/PTI indicaram que o emprego de paredes ou contornos com coeficientes médios de rugosidade geram resultados demasiadamente diferentes daqueles em que foram empregadas malhas que mimetizam rugosidades geométricas nas referidas paredes. A Figura 1.1 mostra a diferença entre tais abordagens sobre a distribuição dos pontos de desgaste indicados na superfície da calha do vertedouro da Usina Hidroelétrica de Itaipu. Tais resultados indicam que a modelagem geométrica da rugosidade se faz necessária para que a simulação seja mais realística, no sentido de se aproximar melhor aos dados.



(a) Domínio com rugosidade física.



(b) Domínio com rugosidade numérica.



(c) Alguns pontos de desgaste georreferenciados.

Figura 1.1: Resultados de simulações e pontos amostrais de desgaste na calha direita.

É relevante, pois, ao estudo da interação fluido-estrutura a avaliação da influência da aplicação de diferentes modelos de turbulência no estudo de efeitos abrasivos-erosivos envolvidos no desgaste superficial de estruturas de concreto, visto a associação dos modelos das equações de momentum e de turbulência no cálculo das trajetórias das partículas que incidem sobre a superfície material, dado que esta utiliza a velocidade do fluido e a viscosidade dinâmica do fluido na determinação da velocidade da particula, que por sua vez é empregada na expressão para avaliar a taxa de abrasão-erosão, relacionada com a avaliação do desgaste do concreto.

Assim sendo, a proposta de trabalho aqui apresentada busca a avaliação do emprego de diferentes modelos de turbulência nos resultados de simulações numérico-computacionais para quantificação do desgaste superficial abrasivo-erosivo da calha esquerda do vertedouro. Estas atividades serão realizadas como parte do projeto de pesquisa de doutorado desenvolvido pelo autor, e como parte das atividades a serem desenvolvidas pelo proponente do projeto denominado "Estudos Experimental-Numéricos e Simulação Computacional de Efeitos Abrasivo-Erosivos e Envelhecimento por Radiação Ultravioleta em Reparos e Concretos Assemelhados Àqueles Predominantes na Calha

Esquerda do Vertedouro da Barragem da Usina Hidroelétrica de Itaipu".

Capítulo 2

Turbulência

A turbulência que ocorre nos fluidos está entre os fenômenos mais complexos encontrados na natureza, sendo tridimensional e variando temporalmente. Também é caracterizado por processos não lineares de troca de massa, quantidade de movimento e energia, sendo essas trocas provenientes das interações entre estruturas de escalas variadas tanto no espaço quanto no tempo. De forma geral, existem duas frentes de trabalho em relação aos estudos deste fenômeno: uma devida aos experimentalistas numéricos e os de laboratório.

Esta seção será apresentada a metodologia de Simulação de Grandes Escalas (LES) para escoamentos turbulentos, que é uma das várias abordagens existentes para o tratamento teórico do tema. Tal abordagem teve início em meados da década de 60 com os trabalhos do meteorologista Smagorinsky, cuja motivaçã estava relacionada à simulação das grandes escalas de escoamentos atmosféricos, dada a inviabilidade de simular a totalidade do espectro de escalas. Embora iniciada por Smagorinsky, foi Deardorff que "importou" a metodologia para problemas de engenharia.

Deve-se ressaltar que é razoável o investimento em pesquisas relacionadas à compreensão e ao controle de escoamentos turbulentos, devido à enorme quantidade de implicações de cunho prático decorrentes, que envolvem sistemas de transportes, transmissão e conversão de energia, aplicções geofísicas, entre outras. Dentre as maiores dificuldades para o desenvolvimento da modelagem matemática dos escoamentos que apresentam turbuência, está a sua modelagem, o que será discutido nesta seção.

Uma das principais características dos escoamentos turbulentos é o elevado grau de liberdade associado a este tipo de sistema dinãmico. A maneira mais tradicional ou intuitiva de conduzir uma simulação de escoamento é com auxílio das equações de Navier-Stokes, e se a malha computacional for suficientemente refinada, todos os fenômenos físicos envolvidos serão resolvidos. Essa abordagem é conhecida como Simulação Numérica Direta (DNS), no entanto, por aspectos computacionais ela nem sempre pode ser utilizada, ficando restrita na maioria das vezes à escoamentos com baixos

número de Reynolds. Esse fato torna a DNS inviável para a maior parte dos problemas práticos encontrados. Diante desta limitação e empregando as ideias de decomposição das escalas de Reynolds, que Smagorinsky propôs uma nova concepção de modelagem, na qual é realizada uma separação estruturas que ocorrem em altas frequências daquelas que ocorrem em baixa frequência, ao invés de separar as escalas em um campo médio e nas suas respectivas flutuações. Nesta metodologia, a comprimento característico do filtro (fator que determina a frequ?ncia de corte) é especificado em função do tamanho da malha utilizada na discretização do problema.

A Simulação de Grandes Escalas é considerada como uma metodologia intermediária à DNS e à simulação empregando equações resultantes da técnica de decomposição de Reynolds. Uma característica desta abordagem é que as estruturas turbulentas resposáveis pelo transporte de energia e quantidade de movimento são resolvidas diretamente por meio das equações filtradas, e somenteas as menores estruturas, que ocorrem com maior frequência são modeladas - filtro passa-baixa????. Ao considerar que estruturas menores apresentem uma isotropia e homogeneidade, espera-se que os modelos sejam mais gerais e independentes dos diferentes tipos de escoamento, quando comparados à metodologia que emprega médias.

Ambas metodologias DNS e LES se assemelham, uma vez que ambas permitem a obtençaõ de resultados tridimensionais e transientes das equações de Navier-Stokes. Embora seja necessário uma malha também refinida para LES, nesta abordagem é possível resolver escoamentos com altos número de Reynolds, devido ao processo de separação das escalas utilizado e ao processo de modelagem dos tensores -sub-malha que são obtidos adicionalmente.

2.1 Médias e filtros

No caso de escoamento de fluidos, a turbulência origina-se da instabilidade de camadas de cisalhamento, que surge no momento em que os efeitos não-lineares são dominantes, o que é caracterizado por elevados números de Reynolds. Na maioria das vezes, é de interesse avaliar as propriedades médias dos escoamentos turbulentos. Para isso, decompõe-se cada grandeza física u na soma de um campo médio $\bar{u}$ e uma componente de flutuação turbulenta u', cuja média é considerada nula. Assim os campos de velocidade, pressão e densidade são escritos como em [**?**]:

$$u_i = \bar{u}_i + u_i' \qquad p = \bar{p} + p' \qquad \rho = \bar{\rho} + \rho_0' \tag{2.1}$$

Como apresentado em [**?**] existem diversas formas de definir as grandezas médias e as flutuações, sendo que as propriedades dessas grandezas dependem da sua definição.

2.1.1 Médias de Reynolds

As médias de Reynolds incluem as médias de conjunto, as médias temporais e as médias espaciais. Se o escoamento é quase estacionário, médias com relação ao tempo podem ser utilizadas. No caso da turbulência homogênea, as médias espaciais podem ser tomadas e para os demais casos, pode ser necessário que as médias sejam obtidas sobre um grande número de experimentos possuindo as mesmas condições iniciais e de contorno, o que nos remete ao *valor esperado* ou *esperança matemática* da variável.

Ao tratar um processo estacionário e homogêneo, essas três médias são idênticas, o que é conhecido como hipótese ergótica. Simulações que empregam as variáveis das Equações de Navier-Stokes promediadas por Reynolds são denominadas de RANS (Reynolds-averaged Navier-Stokes).

A média temporal para uma turbulêcia estacionária é como:

$$\bar{U}^t = \lim_{T\to\infty} \frac{1}{2T} \int_{-T}^{T} U(x_0, t)dt \tag{2.2}$$

A média espacial para uma turbulêcia homogênea é expressa por:

$$\bar{U}^s = \lim_{X\to\infty} \frac{1}{2X} \int_{-X}^{X} U(x, t_0)dx \tag{2.3}$$

O valor esperado para uma repetição de N experimentos:

$$\bar{U}^e = \lim_{N\to\infty} \frac{1}{N} \sum_{1}^{N} U_n(x_0, t_0) \tag{2.4}$$

Obviamente os escoamentos de real interesse não são estacionários nem homogêneos. Além disso, por motivos práticos não é possível obter a média para valores infinitos de X, T e N. Deve-se então determinar um intervalo adequado que represente o fenômeno em estudo, ao qual decorrerá a média.

Como discutido em [**?**] a técnica de passagem da média de Reynolds consiste de duas etapas: na primeira, as variáveis presentes nas equações do movimento são decompostas em termos médios e de flutuações; na segunda, deve-se aplicar o operador média temporal sobre um intervalo de tempo finito nos termos resultantes.

As grandezas que caracterizam o campo de um escoamento são escritas para o precesso de decomposição como:

$$\begin{aligned} u_i &= \bar{u}_i + u_i' \\ p &= \bar{p} + p' \\ \rho &= \bar{\rho} + \tilde{\rho} \end{aligned} \tag{2.5}$$

em que u_i representa as componentes da velocidade nas i direções, p é a pressão e ρ é a massa específica. Por notação, as flutuações de densidade foram designadas por ρ'.

Como já apresentado, a equação de conservação de massa ou equação da continuidade para um escoamento incompressível pode ser escrita como:

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x_j}(\rho u_j) = 0 \tag{2.6}$$

Empregando as expressões médias para u e ρ, em (2.6) tem-se:

$$\frac{\partial}{\partial t}(\bar{\rho} + \tilde{\rho}) + \frac{\partial}{\partial x_j}\Big((\bar{\rho} + \tilde{\rho})(\bar{u}_i + u_i')\Big) = 0 \tag{2.7}$$

2.1.2 Separação de escalas

As equações que governam o escoamento de fluidos incompressíveis são especificadas basicamente pelas equações de conservação da massa e da quantidade de movimento, expressas respectivamente, por:

$$\begin{gathered} \frac{\partial u_i}{\partial x_i} = 0 \\ \frac{\partial u_i}{\partial t} + \frac{\partial}{\partial x_j}(u_i u_j) = -\frac{1}{\rho_0}\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left(\nu \frac{\partial u_i}{\partial x_j}\right) \end{gathered} \tag{2.8}$$

em que $i, j = 1, 2, 3$, u é a velocidade t é o tempo, p é a pressão, ρ_0 a densidade e ν a viscosidade do fluido. Como observado, esse sistema apresenta 4 equações e 4 incógnitas, constituindo um sistema de equações fechado. A solução direta deste sistema de equações é possível somente para baixos números de Reynolds, e em escoamentos com elevados números de Reynolds uma alternativa à solução é o processo de filtragem e de separação de escalas. Para isto, as variáveis presentes nas referidas equações governantes devem ser separadas em uma parcela denominada de grandes escalas $\bar{f}(\vec{x}, t)$ e em outra parte denominada sub-malha $f'(\vec{x}, t)$, de forma que:

$$f(\vec{x}, t) = \bar{f}(\vec{x}, t) + f'(\vec{x}, t) \tag{2.9}$$

em que a parte filtrada é definida como:

$$\bar{f}(\vec{x}, t) = \int_D f(\vec{x}, t) G(\vec{x} - \vec{x}') d\vec{x}' \tag{2.10}$$

onde $G(\vec{x})$ é a função filtro e pode ser definida de diversas formas. Uma das mais frequentes funções filtro encontradas na literatura é expressa por:

$$G(\vec{x}) = \begin{cases} 1/\Delta^3 & \text{se } |\vec{x}| \leq \Delta/2 \\ 0 & \text{se } |\vec{x}| > \Delta/2 \end{cases} \tag{2.11}$$

em que Δ é o tamanho característico do filtro, e caracteriza a frequência de corte da filtragem.

Capítulo 3

Instalação e Configuração do PETSc

Esta seção tem como objetivo auxiliar na instalação e configuração do pacote de ferramentas PETSc, seguindo o manual do software. Informações adicionais podem ser obtidas diretamente no site do desenvolvedor da ferramenta: https://www.mcs.anl.gov/petsc/.

Nesta seção serão apresentados também alguns requisitos computacionais necessários à implementação dos métodos numéricos e resolução das equações discretizadas. Também será exibido uma introdução, instalação e configuração da biblioteca PETSc para o sistema Linux Ubuntu 16.04. PETSc (Portable, Extensible Toolkit for Scientific Computation), é um conjunto de estruturas de dados e rotinas para solução paralela de aplicações científicas modeladas por meio de equações diferenciais parciais.

Para instalação do PETSc é necessário a realização do download da distribuição mais atual do pacote, o que pode ser obtido no link https://www.mcs.anl.gov/petsc/download/index.html. Após a obtenção do arquivo `.tar.gz` é necessário extraí-lo em alguma pasta do computador local. Isto pode ser feito por meio do terminal e empregando o comando `tar -xf petsc-3.7.5.tar.gz`. Após a extração, deve-se acessar a pasta que foi criada e que neste tutorial chama-se `/petsc-3.7.5.tar.gz` e executar o comando para configuração do PETSc:

```
./configure -with-cc=gcc -with-cxx=g++ -with-X=1 -with-fc=gfortran
-download-mpich -download-fblaslapack
```

Esse comando, além de configurar o PETSc, também instala algumas ferramentas como compiladores e bibliotecas. Caso a configuração ocorra corretamente uma mensagem semelhante à da Figura 3.1 deve ser exibida no terminal.

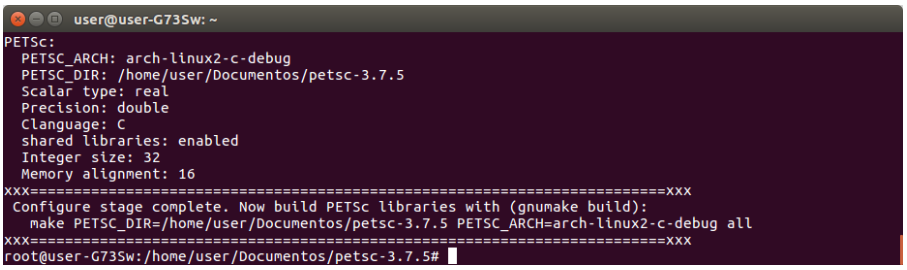


Figura 3.1: Terminal configuração PETSc.

Como indicado, o usuário deve em seguida executar o seguinte comando:

```
make PETSC_DIR=/home/user/Documentos/petsc-3.7.5 PETSC_ARCH=arch-linux2-c-debug all
```

o que resultará na seguinte Figura 3.2.

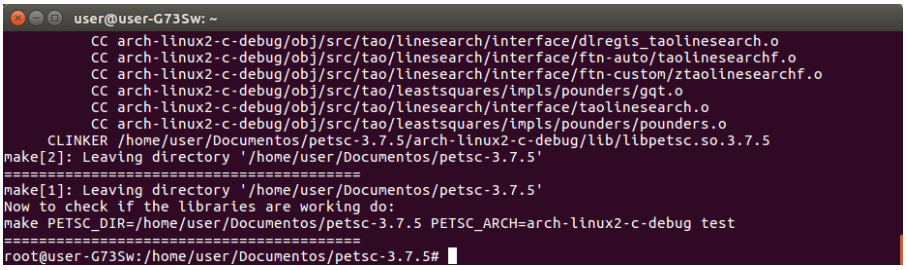


Figura 3.2: Terminal configuração PETSc.

Como indicado, o usuário deve verificar se as bibliotecas estão trabalhando corretamente, por meio do comando:

```
make PETSC_DIR=/home/user/Documentos/petsc-3.7.5 PETSC_ARCH=arch-linux2-c-debug test
```

o que resultará na seguinte Figura 3.3.

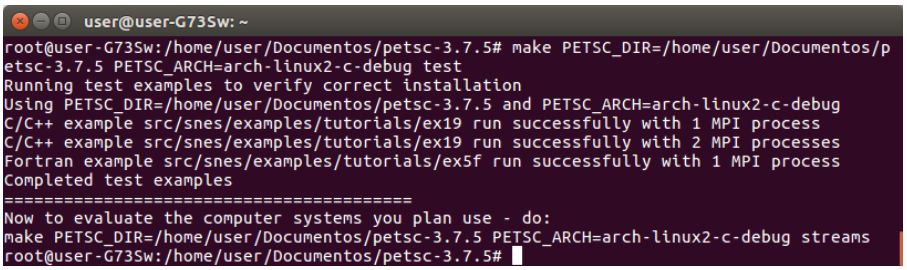


Figura 3.3: Terminal configuração PETSc.

Para finalizar a configuração, o usuário deve o sistema, por meio do comando:

```
make PETSC_DIR=/home/user/Documentos/petsc-3.7.5 PETSC_ARCH=arch-linux2-c-debug streams
```

o que resultará na seguinte Figura 3.4.

Figura 3.4: Terminal configuração PETSc.

Se a instalação foi realizada com êxito uma janela com o speedup em função do número de processadores será exibida.

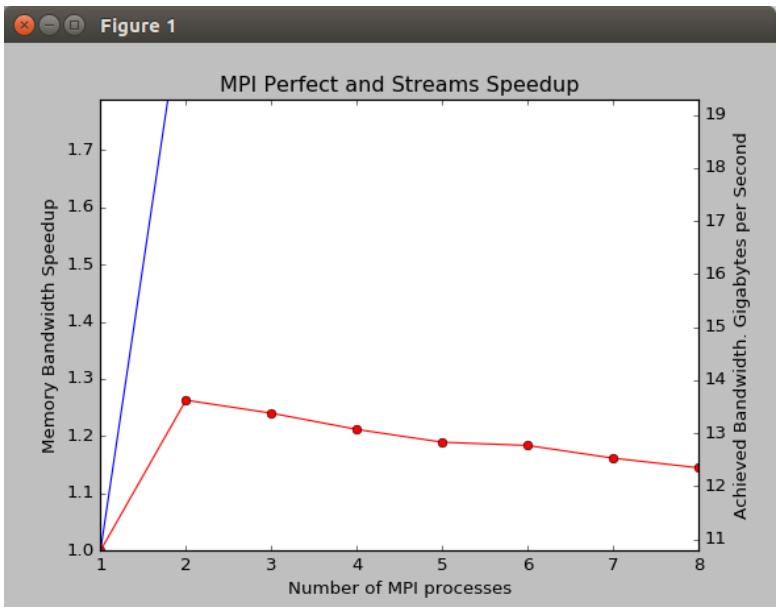


Figura 3.5: Terminal configuração PETSc.

Ao término da instalação deve-se também configurar, via terminal, as variáveis de ambiente

PETSC_DIR=/home/user/Documentos/petsc-3.7.5 e PETSC_ARCH=arch-linux2-c-debug com auxílio do comando export. Ver Figura 3.6.

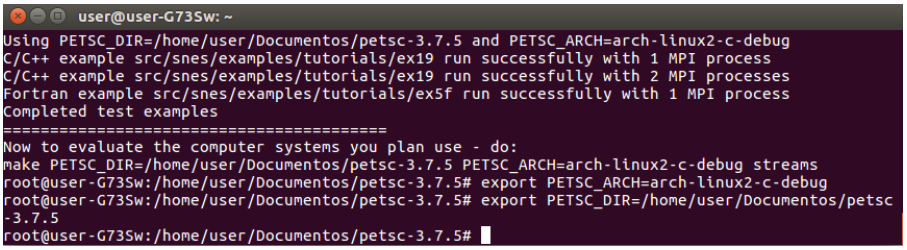


Figura 3.6: Terminal configuração PETSc.

Para verificar se a instalação e configuração foram realizadas com êxito, é possível navegar até a o diretório: `cd /home/user/Documentos/petsc-3.7.5/src/ksp/ksp/examples/tutorials/` e executar os comandos:

```
make ex1
```

o que resultará na Figura 3.7.

Figura 3.7: Terminal exemplo ex1.c.

Para executar o código acima compilado, digita-se `./ex1`, o que deve resultar em algo semelhante à Figura 3.8.

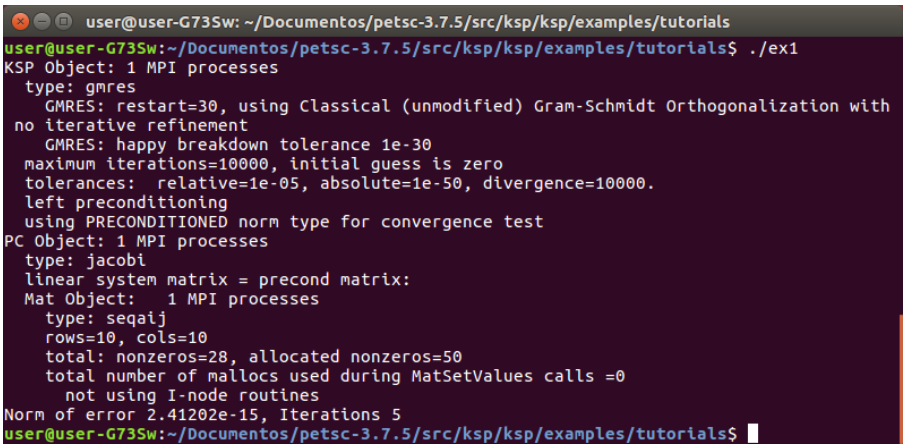


Figura 3.8: Execução do exemplo ./ex1

Para uma configuraçã completa do PETSc, talvez seja necessária a instalação de alguns pacotes adicionais como é o caso do X11 para criação de janelas gráficas. Sua instalação pode ser realizada via terminal com o comando: `apt install libxt-dev`.

3.1 Primeiros exemplos com PETSc

Como já mencionado, o PETSc é uma suite de ferramentas que permite a solução de sistemas de equações em paralelo. Essa biblioteca foi desenvolvida para resolução de Equações Diferenciais Parciais, em que sua resolução conduz à resolução de sistemas de equações de grandes dimensões, o que demanda algoritmos eficientes e programação paralela. Desta forma o propóstico da biblioteca PETSc é auxiliar na solução de problemas científicos e de engenharia em computadores multiprocessados.

O primeiro exemplo aqui ilustrado é apresentado em [4] e aproxima a constante de Euler por meio da série de Maclaurin:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} \approx 2.718281828 \tag{3.1}$$

O programa `code_euler.c` , ilustrado no código 3.1, realiza a computação de cada termo da série infinita em cada processo, resultando numa melhor estimativa de e quando executado em vários processos MPI. Embora seja um exemplo ingênuo do emprego da biblioteca PETSc, ele auxilia na compreensão de algumas ideias envolvidas em computação paralela.

```
1 #include <petsc.h>
2
3 int main(int argc, char **argv) {
4   PetscErrorCode ierr;
5   int            rank, i;
6   double         localval, globalsum;
```

```
7
8   PetscInitialize(&argc,&argv,NULL,"Calcula 'e' em paralelo com PETSc.\n\n");
9
10  ierr = MPI_Comm_rank(PETSC_COMM_WORLD,&rank); CHKERRQ(ierr);
11
12  // calcula  1 / n!  onde n = (rank do processo) + 1
13  localval = 1.0;
14  for (i = 2; i < rank+1; i++)
15      localval /= i;
16
17  // soma as contribuições de cada processo
18  ierr = MPI_Allreduce(&localval, &globalsum, 1, MPI_DOUBLE, MPI_SUM,
19                       PETSC_COMM_WORLD); CHKERRQ(ierr);
20
21  // imprime a estimativa de e
22  ierr = PetscPrintf(PETSC_COMM_WORLD,
23      "O valor da constante 'e' é aproximadamente: %17.15f\n",globalsum);
24  CHKERRQ(ierr);
25
26  ierr = PetscPrintf(PETSC_COMM_SELF,
27      "rank %d did %d flops\n",rank,(rank > 0) ? rank-1 : 0); CHKERRQ(ierr);
28
29  PetscFinalize();
30
31  return 0;
32 }
```

Listing 3.1: Código constante de Euler.

Como qualquer programa escrito em linguagem C, o código é iniciado com uma função chamada `main()` a qual tem os argumentos `argc` e `argv` passados via linha de comando. No exemplo ilustrado esses argumentos serão passados à biblioteca através da função `PetscInitialize()`, e a biblioteca obtém as informações passadas em linha de comando. A função `main()` também tem como retorno um valor inteiro, que é igual a 0 se o programa foi executado corretamente. Além disso, é importante utilizar a função PETSc para vericação de erros associados à sua utilização, `CHKERRQ(ierr)`, a qual retorna um valor inteiro diferente de 0 caso alguma anomalia ocorra na execução de alguma função pertencente à biblioteca.

Como indicado no manual [1] para compilar um arquivo que utiliza PETSc, deve-se ter no mesmo diretório do arquivo fonte, um arquivo `makefile`, cujo conteúdo é exibido no código 3.2.

```
1 include ${PETSC_DIR}/lib/petsc/conf/variables
2 include ${PETSC_DIR}/lib/petsc/conf/rules
3
4 code_euler: code_euler.o  chkopts
5   -${CLINKER} -o code_euler code_euler.o  ${PETSC_LIB}
6   ${RM} code_euler.o
```

```
7
8  build_vector_01: build_vector_01.o  chkopts
9    -${CLINKER} -o build_vector_01 build_vector_01.o  ${PETSC_LIB}
10   ${RM} build_vector_01.o
11
12
13 build_vector: build_vector.o  chkopts
14   -${CLINKER} -o build_vector build_vector.o  ${PETSC_LIB}
15   ${RM} build_vector.o
16
17 build_matrix: build_matrix.o  chkopts
18   -${CLINKER} -o build_matrix build_matrix.o  ${PETSC_LIB}
19   ${RM} build_matrix.o
20
21 solve_linear_ksp: solve_linear_ksp.o  chkopts
22   -${CLINKER} -o solve_linear_ksp solve_linear_ksp.o  ${PETSC_LIB}
23   ${RM} solve_linear_ksp.o
24
25 solve_linear_arbitrary: solve_linear_arbitrary.o  chkopts
26   -${CLINKER} -o solve_linear_arbitrary solve_linear_arbitrary.o  ${PETSC_LIB}
27   ${RM} solve_linear_arbitrary.o
28
29 poisson_fd_2d: poisson_fd_2d.o  chkopts
30   -${CLINKER} -o poisson_fd_2d poisson_fd_2d.o  ${PETSC_LIB}
31   ${RM} poisson_fd_2d.o
```

Listing 3.2: Exemplo de `makefile`.

Após criar o arquivo `makefile` é possível compilar o código programa `code_euler.c` com o seguinte comando:

```
user@user-G73Sw:~$ make code_euler
```

Para executar o código compilado deve-se executar

```
user@user-G73Sw:~$ ./code_euler
O valor da constante 'e' é aproximadamente: 1.000000000000000
rank 0 did 0 flops
```

O valor obtido para $e = 1.0$ é uma estimativa muito ruim, e isso pode ser melhorado com a execução de mais processos MPI, da seguinte forma:

```
user@user-G73Sw:~$ mpiexec -n 5 ./code_euler
O valor da constante 'e' é aproximadamente: 2.708333333333333
```

```
rank 0 did 0 flops
rank 1 did 0 flops
rank 2 did 1 flops
rank 3 did 2 flops
rank 4 did 3 flops
```

Executando o mesmo programa em 10 processos, obtemos uma boa aproximação constante

```
user@user-G73Sw:~$ mpiexec -n 10 ./code_euler
O valor da constante 'e' é aproximadamente: 2.718281525573192
rank 0 did 0 flops
.....
```

Com base na execução dos 10 processos acima, pode-se imaginar que o código tenha sido escrito usando um cluster com no mínimo 10 processadores físicos. Na verdade, esses 5 e 10 processos funcionam muito bem em um computador pessoal com 2 núcleos (processodor i3 2120). Os processos MPI são criados conforme necessário, usando um recurso antigo de sistemas operacionais: multitarefa. Obviamente a aceleração real do paralelismo (speedup) é outra questão.

No exemplo do programa `code_euler.c`, cada processo MPI calcula o termo $1/n!$, onde n é o retorno de `MPI_Comm_rank()`. É importante notar que `PETSC_COMM_WORLD` é um comunicador MPI contendo todos os processos gerados usando `mpiexec -n N` na linha de comando. Uma chamada para `MPI_Allreduce()` calcula a soma parcial de expressão (3.1) e envia o resultado de volta para cada processo. Esses usos diretos da API MPI são uma parte (relativamente pequena) do uso do PETSc, mas ocorrem porque o PETSc geralmente evita a duplicação da funcionalidade MPI.

A estimativa calculada de e é impressa de uma só vez. Além disso, cada processo também imprime seu `rank` e o trabalho que ele fez. O comando de impressão formatado `PetscPrintf()`, semelhante ao `fprintf()` da biblioteca padrão C, é chamado duas vezes no código. Na primeira vez MPI usa o comunicador `PETSC_COMM_WORLD` e a segunda vez `PETSC_COMM_SELF`. O primeiro desses trabalhos de impressão é, portanto, coletivo em todos os processos, e apenas uma linha de saída é produzida, enquanto a segunda é individual para cada processo e obtemos n linhas impressas. As linhas de saída `PETSC_COMM_SELF` podem aparecer em ordem aparentemente aleatória uma vez que a impressão ocorre na ordem que essa classificação encontra o comando `PetscPrintf()` no código.

Todo programa ou parte de programa que utiliza a biblioteca PETSc, deve iniciar e terminar com as funções `PetscInitialize()` e `PetscFinalize()`, respectivamente. Observa-se que o último argumento da função `PetscInitialize(&argc, &argv, NULL, help)` fornece uma string que informa ao usuário uma breve descrição do programa, e pode ser visualizada através do comando:

```
user@user-G73Sw:~$ ./code_euler --help
O valor da constante 'e' é aproximadamente: 2.718281525573192
rank 0 did 0 flops
.....
```

3.1.1 Objetos do tipo vetores e matrizes em PETSc

A maioria dos métodos de resolução numérica de equações diferenciais culmina na solução de sistemas lineares de dimensão finita. Como esses sistemas lineares se tornam representações mais precisas da EDP à medida que seu tamanho vai para o infinito, busca-se resolver os maiores sistemas lineares que a tecnologia disponível possa ser capaz de computar. Resolver tais sistemas lineares, usando algoritmos que têm o potencial de escalar para tamanhos muito grandes - tão grandes, por exemplo, que a solução vetorial do sistema deve ser distribuída através de muitos processadores até mesmo em memória distribuída - representa a biblioteca PETSc.

Uma observação a ser feita, é que muitas das PDEs discretizadas geram sistemas lineares com estrutura explorável, especialmente a esparcidade, o que significa que há poucas entradas diferentes de zero por linha na matriz. Para que os métodos convirjam, também precisa haver outra estrutura no sistema linear, tal como a regularidade das entradas de matrizes que surgem da suavidade dos coeficientes na PDE. A aplicação ingênua de métodos diretos é na maioria das vezes, muito lenta.

O código exibido a seguir ilustra um exemplo da criação de um vetor 10×1 em PETSc.

```
1  static char help[] = "Monta um vetor 10 x 1 usando Vec.\n";
2
3  #include <petsc.h>
4
5  int main(int argc, char **args) {
6    Vec x;
7    int    i[10] = {0, 1, 2, 3,4, 5, 6, 7, 8, 9};
8    double v[10] = {11.0, 7.0, 5.0, 3.0, 6.0,
9              11.0, 7.0, 5.0, 3.0, 6.0};
10
11   PetscInitialize(&argc, &args, NULL, help);
12
13   VecCreate(PETSC_COMM_WORLD,&x);
14   VecSetSizes(x,PETSC_DECIDE,10);
15   VecSetFromOptions(x);
16   VecSetValues(x,10,i,v,INSERT_VALUES);
17   VecAssemblyBegin(x);
18   VecAssemblyEnd(x);
19
20   VecDestroy(&x);
21
```

```
22   PetscFinalize();
23   return 0;
24 }
```

codigo/build_vector.c

Para compilar o código acima, deve-se alterar o arquivo `makefile` para deixá-lo semelhante ao apresentado abaixo:

```
1  include ${PETSC_DIR}/lib/petsc/conf/variables
2  include ${PETSC_DIR}/lib/petsc/conf/rules
3
4  code_euler: code_euler.o  chkopts
5    -${CLINKER} -o code_euler code_euler.o  ${PETSC_LIB}
6    ${RM} code_euler.o
7
8  build_vector_01: build_vector_01.o  chkopts
9    -${CLINKER} -o build_vector_01 build_vector_01.o  ${PETSC_LIB}
10   ${RM} build_vector_01.o
11
12
13 build_vector: build_vector.o  chkopts
14   -${CLINKER} -o build_vector build_vector.o  ${PETSC_LIB}
15   ${RM} build_vector.o
16
17 build_matrix: build_matrix.o  chkopts
18   -${CLINKER} -o build_matrix build_matrix.o  ${PETSC_LIB}
19   ${RM} build_matrix.o
20
21 solve_linear_ksp: solve_linear_ksp.o  chkopts
22   -${CLINKER} -o solve_linear_ksp solve_linear_ksp.o  ${PETSC_LIB}
23   ${RM} solve_linear_ksp.o
24
25 solve_linear_arbitrary: solve_linear_arbitrary.o  chkopts
26   -${CLINKER} -o solve_linear_arbitrary solve_linear_arbitrary.o  ${PETSC_LIB}
27   ${RM} solve_linear_arbitrary.o
28
29 poisson_fd_2d: poisson_fd_2d.o  chkopts
30   -${CLINKER} -o poisson_fd_2d poisson_fd_2d.o  ${PETSC_LIB}
31   ${RM} poisson_fd_2d.o
```

codigo/makefile

Em seguida o código é compilado e executado através dos comandos:

```
user@user-G73Sw:~$ make build_vector
user@user-G73Sw:~$ ./build_vector -vec_view
```

```
Mat Object: 1 MPI processes
type: seqaij
11.
7.
5.
3.
6.
11.
7.
5.
3.
6.
```

A seguir é apresentado um exemplo simples de como preencher uma matriz 4×4, usando um loop 'for' sobre o índice de linha i. O programa é denominado de `build_matrix.c`:

```c
static char help[] = "Monta uma matriz 4x4 usando Mat.\n";

#include <petsc.h>

int main(int argc, char **args) {
  Mat A;
  int   i, j[4] = {0, 1, 2, 3};
  double  aA[4][4] ={{1.0, 2.0, 3.0, 0.0},
      { 2.0, 1.0, -2.0, -3.0},
      {-1.0, 1.0, 1.0, 0.0},
      { 0.0, 1.0, 1.0, -1.0}};

  PetscInitialize(&argc, &args, NULL, help);

  MatCreate(PETSC_COMM_WORLD, &A);
  MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,4,4);
  MatSetFromOptions(A);
  MatSetUp(A);
  for (i=0; i<4; i++) {
    MatSetValues(A,1,&i,4,j,aA[i],INSERT_VALUES);
  }
  MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
  MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);

  MatDestroy(&A);

  PetscFinalize();
  return 0;
```

```
29 }
```

codigo/build_matrix.c

Neste exemplo a matriz é montada por linhas, e em cada iteração do laço 'for', são inseridas quatro colunas em cada linha, a saber as colunas indicadas pelo vetor j. O resultado ou a matriz criada pode ser visualizada de várias formas, e aqui faremos duas visualizações uma no formato esparso e outra mostrando todos os seus elementos.

```
user@user-G73Sw:~$ ./build_matrix -mat_view
Mat Object: 1 MPI processes
type: seqaij
row 0: (0, 1.)  (1, 2.)  (2, 3.)  (3, 0.)
row 1: (0, 2.)  (1, 1.)  (2, -2.)  (3, -3.)
row 2: (0, -1.)  (1, 1.)  (2, 1.)  (3, 0.)
row 3: (0, 0.)  (1, 1.)  (2, 1.)  (3, -1.)
```

```
user@user-G73Sw:~$ ./build_matrix -mat_view ::ascii_dense
Mat Object: 1 MPI processes
type: seqaij
 1.00000e+00  2.00000e+00  3.00000e+00  0.00000e+00
 2.00000e+00  1.00000e+00 -2.00000e+00 -3.00000e+00
-1.00000e+00  1.00000e+00  1.00000e+00  0.00000e+00
 0.00000e+00  1.00000e+00  1.00000e+00 -1.00000e+00
```

É possível também salvar a matriz impressa no terminal através do comando:

```
./build_matrix -mat_view ascii:build_matrix.txt:ascii_dense.
```

3.1.2 Solução de sistema linear

Como descrito em [4], embora PETSc seja escrito em C, e não em C ++, ela é uma biblioteca orientada a objetos. Para construir o nosso primeiro código PETSc para resolver um sistema linear, vamos usar os tipos de dados `Vec` e `Mat`, que são essencialmente objetos, que possuem vetores e matrizes. O exemplo a seguir descreve a solução do sistema linear:

$$\begin{bmatrix} 1 & 2 & 3 & 0 \\ 2 & 1 & -2 & -3 \\ -1 & 1 & 1 & 0 \\ 0 & 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 1 \\ 1 \\ 3 \end{bmatrix} \tag{3.2}$$

Um objeto KSP resolve o sistema linear, com o algoritmo de solução específico escolhido apenas em tempo de execução. O código fonte que contém o programa para resolver o sistema (3.15) é apresentado a seguir:

```
1  static char help[] = "Resolve um sistema linear 4x4 usando Vec, Mat, e KSP.\n";
2
3  #include <petsc.h>
4
5  int main(int argc, char **args) {
6    Vec x, b;
7    Mat A;
8    KSP ksp;
9    int   i, j[4] = {0, 1, 2, 3};
10   double  ab[4] = {7.0, 1.0, 1.0, 3.0};
11   double  aA[4][4] ={{1.0, 2.0, 3.0, 0.0},
12       { 2.0, 1.0, -2.0, -3.0},
13       {-1.0, 1.0, 1.0, 0.0},
14       { 0.0, 1.0, 1.0, -1.0}};
15
16   PetscInitialize(&argc, &args, NULL, help);
17
18   VecCreate(PETSC_COMM_WORLD, &b);
19   VecSetSizes(b,PETSC_DECIDE,4);
20   VecSetFromOptions(b);
21   VecSetValues(b,4,j,ab, INSERT_VALUES);
22   VecAssemblyBegin(b);
23   VecAssemblyEnd(b);
24
25   MatCreate(PETSC_COMM_WORLD, &A);
26   MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,4,4);
27   MatSetFromOptions(A);
28   MatSetUp(A);
29   for (i=0; i<4; i++) {
30     MatSetValues(A,1,&i,4,j,aA[i],INSERT_VALUES);
31   }
32   MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
33   MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
34
35   KSPCreate(PETSC_COMM_WORLD,&ksp);
36   KSPSetOperators(ksp, A, A);
37   KSPSetFromOptions(ksp);
38   VecDuplicate(b,&x);
39   KSPSolve(ksp,b,x);
40   VecView(x,PETSC_VIEWER_STDOUT_WORLD);
41
42   KSPDestroy(&ksp);
43   MatDestroy(&A);
44   VecDestroy(&x);
```

```
45   VecDestroy(&b);
46   PetscFinalize();
47   return 0;
48 }
```

codigo/solve_linear_ksp.c

O resultado final da execução do programa `solve_linear_ksp.c` é:

```
user@user-G73Sw:~$ ./solve_linear_ksp
Vec Object: 1 MPI processes
type: seq
1.
0.
2.
-1.
```

O código `solve_linear_ksp.c` apresentou a solução de um sistema com dimensão fixa, no entanto pode ser necessário, alterar essa dimensão em tempo de execução, como é o caso do próximo exemplo. Ele resovle um sistema de equação com tamanho arbitrário e definido no momento da execução através de um valor inteiro passado na função `PetscOptionsXXX()`. Além disso, nesse exemplo serão vistos algumas formas de manipulação de vetores como soma e determinação da sua norma euclidiana, utilizando `VecAXPY` e `VecNorm`, respectivamente. O programa é ilustrado no Código 3.3.

```
1  //STARTSETUP
2  static char help[] =
3    "Resolve um sistema linear tridiagonal de tamanho definido pelo usuário.\n";
4
5  #include <petsc.h>
6
7  int main(int argc,char **args) {
8    PetscErrorCode ierr;
9    Vec     x, b, xexact;
10   Mat     A;
11   KSP     ksp;
12   int     m = 4, i, Istart, Iend, j[3];
13   double v[3], xval, errnorm;
14
15   PetscInitialize(&argc,&args,NULL,help);
16
17   ierr = PetscOptionsBegin(PETSC_COMM_WORLD,"tri_","Opções para tri","");
18    CHKERRQ(ierr);
19   ierr = PetscOptionsInt("-m","Dimensão do sistema","tri.c",m,&m,NULL);
20    CHKERRQ(ierr);
```

```
21   ierr = PetscOptionsEnd(); CHKERRQ(ierr);
22
23   ierr = VecCreate(PETSC_COMM_WORLD,&x); CHKERRQ(ierr);
24   ierr = VecSetSizes(x,PETSC_DECIDE,m); CHKERRQ(ierr);
25   ierr = VecSetFromOptions(x); CHKERRQ(ierr);
26   ierr = VecDuplicate(x,&b); CHKERRQ(ierr);
27   ierr = VecDuplicate(x,&xexact); CHKERRQ(ierr);
28
29   ierr = MatCreate(PETSC_COMM_WORLD,&A); CHKERRQ(ierr);
30   ierr = MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,m,m); CHKERRQ(ierr);
31   ierr = MatSetOptionsPrefix(A,"a_"); CHKERRQ(ierr);
32   ierr = MatSetFromOptions(A); CHKERRQ(ierr);
33   ierr = MatSetUp(A); CHKERRQ(ierr);
34 //ENDSETUP
35   ierr = MatGetOwnershipRange(A,&Istart,&Iend); CHKERRQ(ierr);
36   for (i=Istart; i<Iend; i++) {
37     if (i == 0) {
38       v[0] = 3.0;  v[1] = -1.0;
39       j[0] = 0;    j[1] = 1;
40       ierr = MatSetValues(A,1,&i,2,j,v,INSERT_VALUES); CHKERRQ(ierr);
41     } else {
42       v[0] = -1.0;  v[1] = 3.0;  v[2] = -1.0;
43       j[0] = i-1;   j[1] = i;    j[2] = i+1;
44       if (i == m-1) {
45         ierr = MatSetValues(A,1,&i,2,j,v,INSERT_VALUES); CHKERRQ(ierr);
46       } else {
47         ierr = MatSetValues(A,1,&i,3,j,v,INSERT_VALUES); CHKERRQ(ierr);
48       }
49     }
50     xval = exp(cos(i));
51     ierr = VecSetValues(xexact,1,&i,&xval,INSERT_VALUES); CHKERRQ(ierr);
52   }
53   ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
54   ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
55   ierr = VecAssemblyBegin(xexact); CHKERRQ(ierr);
56   ierr = VecAssemblyEnd(xexact); CHKERRQ(ierr);
57   ierr = MatMult(A,xexact,b); CHKERRQ(ierr);
58
59   ierr = KSPCreate(PETSC_COMM_WORLD,&ksp); CHKERRQ(ierr);
60   ierr = KSPSetOperators(ksp,A,A); CHKERRQ(ierr);
61   ierr = KSPSetFromOptions(ksp); CHKERRQ(ierr);
62   ierr = KSPSolve(ksp,b,x); CHKERRQ(ierr);
63
64   ierr = VecAXPY(x,-1.0,xexact); CHKERRQ(ierr);
65   ierr = VecNorm(x,NORM_2,&errnorm); CHKERRQ(ierr);
66   ierr = PetscPrintf(PETSC_COMM_WORLD,
67          "Erro para sistema de ordem m = %d é |x-xexact|_2 = %.1e\n",m,errnorm);
68    CHKERRQ(ierr);
```

```
69
70   KSPDestroy(&ksp);   MatDestroy(&A);
71   VecDestroy(&x);   VecDestroy(&b);   VecDestroy(&xexact);
72   PetscFinalize();
73   return 0;
74 }
75 //ENDSOLVE
```

Listing 3.3: Resolução de sistema com tamanho arbitrário.

É importante destacar que embora o tamanho do sistema seja arbitrário, ele sempre terá a forma tridiagonal, com valores 3 na diagonal principal e valor -1 na diagonal superior e inferior.

O primeiro novo recurso usado neste código é `PetscOptionsBegin()` e `PetscOptionsEnd()`, ou seja, a chamada para `PetscOptionsInt()`. O início do método define um prefixo `-tri_` para que a nova opção criada seja distinguida das muitas opções integradas do PETSc que começam por exemplo com `-ksp_` ou `-vec_` ou algo do tipo. Aqui `PetscOptionsInt()` cria a opção `-tri_m` para que o usuário possa definir a variável m e deixa como padrão $m = 4$ inalterado se a opção não for definida na execução.

Após configurar a nova opção em `solve_linear_arbitrary.c` a solução numérica `Vec x` é criada exatamente como realizado no último exemplo. Mas agora também é necessário criar `Vec s`, `Vec b` e `Vec xexact`. O primeiro vetor é o lado direito do sistema linear e este último mantém a solução exata para o sistema linear para que seja possível avaliar o erro associado à solução numérica.

Em seguida, deve-se montar a matriz A, que como mencionado é uma matriz tridiagonal. É importante montá-la de forma eficiente em paralelo, algo que será relevante na resolução de equações diferenciais 2D e 3D posteriormente. No entanto, somente quando o `solve_linear_arbitrary.c` é executado, sabemos quantos processos estão em uso. O método `MatGetOwnershipRange()` informa o programa, executando em um determinado processo (rank), quais linhas ele possui localmente.

Como observado no início do código `solve_linear_arbitrary.c`, chamamos a função `MatGetOwnershipRange(A, &Istart, &Iend)` para obter os índices de linha inicial e final para o processo local. Estes índices são usados como limites no loop 'for' que preenche as linhas da matriz localmente. utiliza-se `MatSetValues()` para realmente definir as entradas da matriz A e `MatAssemblyBegin/End()` para completar a montagem de A.

Adicionalmente, é necessário montar o lado direito do sistema linear e também a solução exacta para o sistema linear $(A\,x_{ex} = b)$. A maneira mais simples de fazer isso, é escolher uma solução exata, e em seguida, calcular b, multiplicando A pela solução exata. Assim, definimos valores para `xexact`. Em seguida calculamos b com auxílio da função `MatMult(A, xexact, b)`.

Como em vecmatksp.c criamos o objeto KSP e então chamamos o solver `KSPSolve()` para resolver aproximadamente $Ax = b$. A opção `-ksp_monitor` imprime a norma residual $||b - Ax||_2$ em tempo de execução. Neste caso, também queremos ver que o erro real $||x - x_{ex}||_2$ é pequeno quando o solver KSP é concluido. Assim, depois de obter x do KSPSolve (), calculamos o erro com os códigos:

`VecAXPY(x,-1.0,xexact)` : $x \leftarrow -1.0x_{ex} + x$
`Vecnorm(x,NORM_2,&errnorm)` : `errnorm` $\leftarrow ||x||_2$

Obviamente o sistema linear resolvido neste exemplo é fácil de resolver por ser tridiagonal, simétrico, diagonal-dominante e positivo definido. Pode-se verificar o tempo necessário para resolução do sistema com auxílio do comando `time`, como segue:

```
user@user-G73Sw:~$ time ./solve_linear_arbitrary -tri_m 1000000
error for m = 1000000 system is (x-xexact)_2 = 4.8e-11
real 0m1.814s
user 0m1.772s
sys 0m0.040s
```

Somente o tempo 'real' deve ser considerado. Note a diferença ao executar o mesmo código com $m = 10000000$.

```
user@user-G73Sw:~$ time ./solve_linear_arbitrary -tri_m 10000000
error for m = 10000000 system is (x-xexact)_2 = 3.5e-10
real 0m17.154s
user 0m17.971s
sys 0m0.140s
```

A princípio um sistema de equações contendo 10^7 variáveis parece ser grande. Mas pensando numa simulação tridimensional da equação de Navier-Stokes em um domínio cúbico de 1 metro de lado e com espaçamento de malha igual a 0.01 m (1cm), isso geraria um sistema de equações dessa ordem de grandeza. Agora, o programa será executado empregando 8 processos.

```
user@user-G73Sw:~$ time mpiexec -n 8./solve_linear_arbitrary -tri_m 10000000
error for m = 10000000 system is (x-xexact)_2 = 9.9e-11
real 0m5.484s
user 0m38.976s
sys 0m2.260s
```

Como esperado, o tempo de execução foi reduzido, neste caso caiu de 17 para 5 segundos.

3.2 Diferenças finitas

O código **??** ilustra uma simples implementação serial de uma equação de diferenças avançadas, atrasadas e centradas, denotadas na saída do console respectivamente por, D+, D- e D0.

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <math.h>
4
5 int main()
6 {
7   float h; //valor do espaçamento
8   double x; //valor para avaliação da função
9   double f,dfav,dfat,dfc; //valor da função
10  int i=0;
11  double vecx[6],vec_dfav[6],vec_dfat[6],vec_dfc[6];
12  FILE *file = popen("gnuplot -persistent","w");
13  h = 0.1;
14  x = 1;
15  f = sin(x);
16
17  printf("Valor de f'(x)=cos(x) ou  f'(%f)=cos(%f)=%f\n",x,x,cos(x));
18  printf("h \t\t D+ \t\t D- \t\t D0 \n");
19  while (h>0.001){
20    f = sin(x);
21          dfav = (sin(x+h)-sin(x))/h; //diferenças finitas avançadas
22          dfat = (sin(x)-sin(x-h))/h; //diferenças finitas atrasadas
23          dfc = (sin(x+h)-sin(x-h))/(2*h); //diferenças finitas centrais
24    printf("%f\t %f \t %f \t %f \n",h,dfav,dfat,dfc);
25    h = h/2;
26  }
27  h = 0.1;
28  printf("Valores do Erro\n");
29  printf("h \t\t D+ \t\t D- \t\t D0 \n");
30        while (h>0.001){
31    f = sin(x);
32          vecx[i]=h;
33    vec_dfav[i] = (sin(x+h)-sin(x))/h; //diferenças finitas avançadas
34          vec_dfat[i] = (sin(x)-sin(x-h))/h; //diferenças finitas atrasadas
35          vec_dfc[i] = (sin(x+h)-sin(x-h))/(2*h); //diferenças finitas centrais
36    printf("%f\t %.4e \t %.4e \t %.4e \n",h,vec_dfav[i]-cos(x),
37      vec_dfat[i]-cos(x),vec_dfc[i]-cos(x));
38    h = h/2;
39    i++;
40  }
41  return 0;
```

```
42 }
```

Listing 3.4: Exemplo de `makefile`.

Cuja saída é apresenta a seguir:

```
Valor de f'(x)=cos(x) ou  f'(1.000000)=cos(1.000000)=0.540302
h   D+   D-   D0
0.100000  0.497364   0.581441   0.539402
0.050000  0.519045   0.561110   0.540077
0.025000  0.529728   0.550764   0.540246
0.012500  0.535029   0.545547   0.540288
0.006250  0.537669   0.542928   0.540299
0.003125  0.538987   0.541616   0.540301
0.001563  0.539645   0.540959   0.540302

Valores do Erro
h   D+   D-   D0
0.100000  -4.2939e-02   4.1138e-02   -9.0005e-04
0.050000  -2.1257e-02   2.0807e-02   -2.2510e-04
0.025000  -1.0574e-02   1.0462e-02   -5.6280e-05
0.012500  -5.2732e-03   5.2451e-03   -1.4070e-05
0.006250  -2.6331e-03   2.6261e-03   -3.5176e-06
0.003125  -1.3157e-03   1.3139e-03   -8.7940e-07
0.001563  -6.5762e-04   6.5718e-04   -2.1985e-07
```

3.3 A equação de Poisson

3.3.1 O caso 1D

Nesta seção será realizada a discretização e implementação de um algoritmo computacional empregando a biblioteca PETSc para resolução da equação de Poisson 1D, por meio do método de diferenças finitas. Embora seja um problema simplificado, ele objetiva introduzir alguns conceitos referentes à utilização da ferramenta PETSc. O referido problema é será resolvido numa reta (0,1) empregando condições de contorno do tipo Dirichlet, em que $u(0) = 0$ e $u(1) = 0$. Mais

especificamente, a equação é representada pelo problema de valor de contorno:

$$\begin{aligned} -\frac{d^2u}{dx^2} &= f(x) \quad \text{em } x \in (0,1) \\ u(0) &= 0 \\ u(1) &= 0 \end{aligned} \tag{3.3}$$

Uma aproximação em diferenças finitas centradas de segunda ordem para a equação (3.3) é como:

$$-\frac{d^2u}{dx^2} \approx -\frac{u_{i+1} - 2u_i + u_{i-1}}{h_x^2} \tag{3.4}$$

Ao substituir a segunda derivada da equação de Poisson por sua aproximação em diferenças finitas, tem-se um esquema para determinação de $u_i \approx u(x_i)$:

$$\begin{aligned} -\frac{u_{i+1} - 2u_i + u_{i-1}}{h_x^2} &= f_i \\ u_0 &= 0 \\ u_{m_x} &= 0 \end{aligned} \tag{3.5}$$

Pode-se escrever a equação (3.4) na forma matricial:

$$A\boldsymbol{u} = \boldsymbol{b} \tag{3.6}$$

em que explicitamente, para uma malha 1D uniforme com um total de nodos $m_x = 5$ e com $h_x = 1/(m_x - 1) = 0,25$:

$$\begin{bmatrix} a & -b & & & \\ -b & a & -b & & \\ & -b & a & -b & \\ & & -b & a & -b \\ & & & -b & a \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} 0 \\ f_1 \\ f_2 \\ f_3 \\ 0 \end{bmatrix} \tag{3.7}$$

em que $a = 2/h_x^2 = 32$ e $b = 1/h_x^2 = 16$. Tomando $u(x) = x^4$, tem-se que $f(x) = 12x^2$, o que será utilizado para realização deste exemplo que é escrito em Octave e apresentado no Código **??** .

3.3.2 O caso 2D

Esta seção é dedicada a resolução numérica do problema de Poisson em um quadrado. Este é um problema que permite compreender partes essenciais da implementação de códigos empregando a biblioteca PETSc. A discretização da EDP gera um sistema linear que é mais interessante do que o sistema tridiagonal que foi resolvido anteriormente. Será construída uma malha estruturada usando um objeto `DMDA`, que será introduzido nesta seção, e depois será montada uma matriz em paralelo

com base nessa malha. Por último o sistema linear resultante será resovido em paralelo usando um objeto `KSP` [2].

Neste exemplo a equação de Poisson será resolvida numa região quadrada, $S, (0,1) \times (0,1)$ cujo contorno é representado por ∂S. O domínio do problema é ilustrado na Figura 3.9 e formulado em (3.8).

$$\begin{aligned} -\nabla^2 u &= f \quad \text{em } S \\ u &= 0 \quad \text{em } \partial S \end{aligned} \tag{3.8}$$

O Laplaciano de $u(x, y)$ é especificado por:

$$\begin{aligned} \nabla^2 u &= \nabla \cdot (\nabla u) \\ &= \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \end{aligned} \tag{3.9}$$

e aparece com frequência em modelos matemáticos que expressam a conservação de alguma quantidade u, juntamente com a suposição de que o fluxo de u é proporcional ao seu gradiente. O problema aqui estudado está sujeito à condições de contorno homogêneas de Dirichlet.

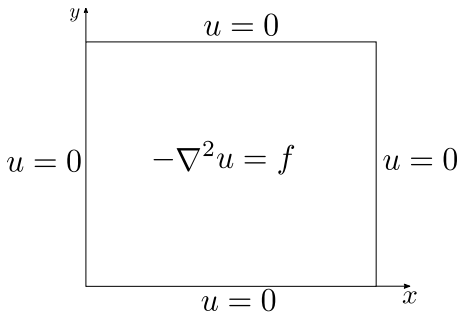


Figura 3.9: Domínio do problema.

O problema de Poisson pode modelar o potencial eletrostático, a distribuição de equilíbrio de certas caminhos aleatórios ou vários outros fenômenos físicos. Para um exemplo, a condução de calor em sólidos segue a lei de Fourier, que diz que o fluxo de calor é $q = -k\nabla u$, onde k é a condutividade térmica. A conservação da energia diz que $c\rho \partial u/\partial t = -\nabla \cdot q + f$ se f representa uma fonte de calor no interior do domínio. O coeficiente $c\rho$ parametriza a capacidade do material para manter o calor por um ganho de temperatura. Se k é constante, então em estado estacionário essas condições resultam na equação de Poisson $0 = k\nabla^2 u + f$. Mantendo a temperatura nula ao longo do limite da região (contorno), tem-se o problema (3.8), que será resolvido numericamente através do método de diferenças finitas.

3.3.3 Geração da malha

O método de diferenças finitias é desenvolvido sobre uma malha composta por $m_x m_y$ pontos igualmente espaçados, como ilustrado na Figura 3.10, cujo espaçamento na direção x é especificado

por $h_x = 1/(m_x - 1)$ e por $h_y = 1/(m_y - 1)$ na direção y.

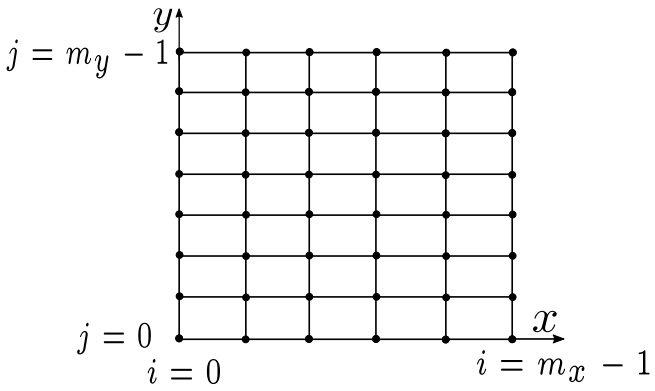


Figura 3.10: Malha computacional.

Considerando a malha da Figura 3.11 em $m_x = 5$ e $m_y = 7$, as coordenadas da malha são $(x_i = ih_x, y_j = jh_y)$, para $i = 0, ..., m_x - 1$ e $j = 0, ..., m_y - 1$.

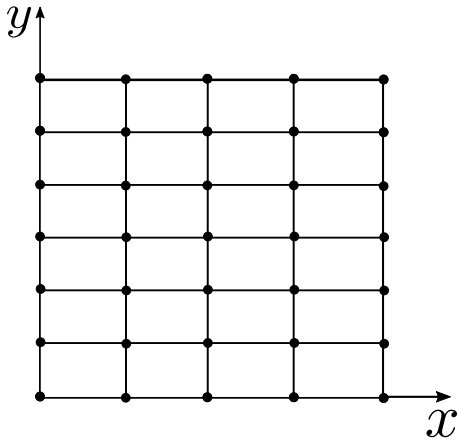


Figura 3.11: Malha computacional 5×7.

Para construção dessa malha 2D de forma distribuida em processos MPI, utiliza-se um novo objeto PETSc que cria uma instância do tipo PETSc `DM` para descrever a topologia (conexão) da malha, a forma como ela é distribuída através de processos MPI e a forma como cada processo pode acessar dados de seus processos vizinhos. Este caso específico aqui é um `DMDA`, que é uma sub-classe de `DM`. A designação `DM` pode significar "distributed mesh" ou "domain management", e `DA` significa "distributed array". Ao executar os comandos abaixo:

```
user@user-G73Sw:~$ make poisson
user@user-G73Sw:~$./poisson -da_grid_x 5 -da_grid_y 7
```

uma malha correspondente à da Figura 3.11 será criada, em que todos os nodos são construídos e pertencentes a um único processo MPI. No entanto, ao digitar o comando:

```
user@user-G73Sw:~$mpiexec -n 4 ./poisson -da_grid_x 5 -da_grid_y 7
```

então a biblioteca PETSc faz o equilíbrio de carga para os pontos da malha entre os processos, com a restrição de que cada processo MPI possui uma sub-malha retangular. Como observado na Figura 3.12, PETSc distribui os quatro processos através dos 35 pontos da malha de forma relativamente uniforme (O processo rank 0 possui 12 pontos e o rank 3 possui 6, enquanto os outros estão entre eles).

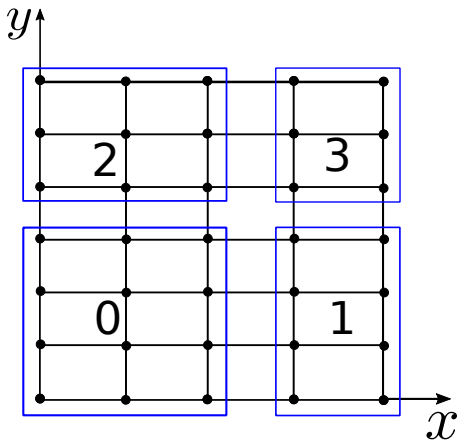


Figura 3.12: Malha computacional 5×7 e 4 processos.

O bloco de código referente ao emprego do objeto `DMDACreate2d` é descrito a seguir:

```
1                &da); CHKERRQ(ierr);
2
3   // Cria matriz do sistema A
4   ierr = DMCreateMatrix(da,&A);CHKERRQ(ierr);
```

Listing 3.5: DMDA()

No Código 3.5, o primeiro argumento é referente ao comunicador. O segundo e terceiro argumentos são do tipo `DM_BOUNDARY_NONE` pois as condições de Dirichlet não precisam de comunicação para o próximo processo. No quarto argumento, escolhe-se o método `DMDA_STENCIL_STAR` porque somente os vizinhos cardeais de um ponto da malha são usados na discretização. A Figura 3.13 mostra seu stencil computacional.

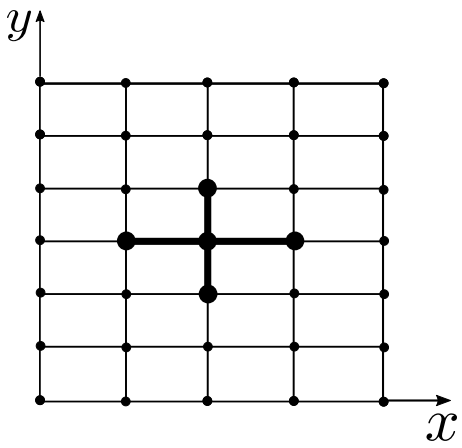


Figura 3.13: Stencil computacional epregado na discretização.

Os dois argumentos `PETSC_DECIDE` depois disso dão ao `PETSc` autonomia para distribuir a malha sobre os processos de acordo com a quantidade de processos no comunicador MPI. Os dois argumentos seguintes, na nona e décima posições, dizem que nossa PDE é escalar, com grau de liberdade igual a 1 (dof $= 1$) e que o método de diferenças finitas só precisará de um vizinho em cada direção (s $= 1$). Os próximos dois argumentos depois disso são NULL porque não estamos dizendo PETSc quaisquer detalhes sobre como distribuir processos sobre a malha. Finalmente, o objeto `DMDA` é retornado através de um argumento ponteiro.

3.3.4 Discretização em diferenças finitas

Considerando uma discretização da equação (3.8) em diferenças finitas, ela pode ser escrita na forma discreta como:

$$-\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h_x^2} - \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h_y^2} = f(x_i, y_j) \tag{3.10}$$

que se aplica a todos os pontos interiores do domínio, ou seja, quando $0 < i < m_x - 1$ e $0 < j < m_j - 1$. As condições de contorno são impostas como:

$$u_{0,j} = 0, \quad u_{m_x-1,j} = 0, \quad u_{i,0} = 0, \quad u_{i,m_y-1} = 0, \tag{3.11}$$

para todo i, j.

Todos os valores $u_{i,j}$ serão tratados como incógnitas, sejam do interior ou da fronteira, resultando num total de $L = m_x m_y$ incógnitas. Isso permite a construção de um sistema de equações linear:

$$A\boldsymbol{u} = \boldsymbol{b} \tag{3.12}$$

em que a matriz A tem dimensão $L \times L$, e os vetores $\boldsymbol{u}$ e $\boldsymbol{b}$, dimensão $L \times 1$. No entanto, para montar as entradas de A e $\boldsymbol{b}$ no sistema linear (3.12), deve-se ordenar as incógnitas. Essa ordenação é implementada dentro do objeto `DMDA`, e o código só precisa usar as coordenadas (i, j).

A ordenação usada em um processo único (serial) executado por um DMDA 2D é mostrada na Figura 3.14. Em uma malha m_x por m_y, pode-se escrever o novo índice global como $k = jm_x + i$ de modo que $u_{i,j}$ seja a k-ésima incógnita do sistema. Esse mapeamento de índice feito pelo objeto DMDA fica transparente ao usuário.

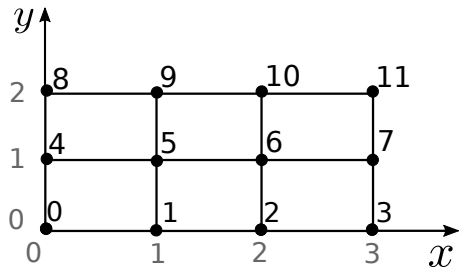


Figura 3.14: Malha computacional 4×3.

Um primeiro exemplo será feito de forma detalhada para auxiliar na compreensão. O tamanho da malha escolhido é de 4×3, ou seja, $m_x = 4$ e $m_y = 3$, o que resulta em $h_x = 1/3$ e $h_y = 1/2$. Desta forma, utiliza-se a expressão 3.10 para cada nodo da malha que não pertença às condições de contorno:
Para o ponto $(1, 1)$ tem-se:

$$\begin{aligned} f(1,1) &= -\frac{u_{2,1} - 2u_{1,1} + u_{0,1}}{h_x^2} - \frac{u_{1,2} - 2u_{1,1} + u_{1,0}}{h_y^2} \\ &= -\frac{u_{2,1}}{h_x^2} - \frac{u_{0,1}}{h_x^2} + 2u_{1,1}\left(\frac{1}{h_x^2} + \frac{1}{h_y^2}\right) - \frac{u_{1,2}}{h_y^2} - \frac{u_{1,0}}{h_y^2} \\ &= -\frac{1}{h_x^2}u_{2,1} - \frac{1}{h_x^2}u_{0,1} + \left(\frac{2}{h_x^2} + \frac{2}{h_y^2}\right)u_{1,1} - \frac{1}{h_y^2}u_{1,2} - \frac{1}{h_y^2}u_{1,0} \end{aligned} \tag{3.13}$$

De forma análoga, para o ponto $(2, 1)$ tem-se:

$$\begin{aligned} f(2,1) &= -\frac{u_{3,1} - 2u_{2,1} + u_{1,1}}{h_x^2} - \frac{u_{2,2} - 2u_{2,1} + u_{2,0}}{h_y^2} \\ &= -\frac{u_{3,1}}{h_x^2} - \frac{u_{1,1}}{h_x^2} + 2u_{2,1}\left(\frac{1}{h_x^2} + \frac{1}{y_x^2}\right) - \frac{u_{2,2}}{h_y^2} - \frac{u_{2,0}}{h_y^2} \\ &= -\frac{1}{h_x^2}u_{3,1} - \frac{1}{h_x^2}u_{1,1} + \left(\frac{2}{h_x^2} + \frac{2}{h_y^2}\right)u_{2,1} - \frac{1}{h_y^2}u_{2,2} - \frac{1}{h_y^2}u_{2,0} \end{aligned} \tag{3.14}$$

Para escrever a equação de diferenças finitas na forma matricial, deve-se separar os coeficientes que multiplicam as velocidades e os valores da função f. A seguir é construído o sistema de equações para o caso da malha representada na Figura 3.14. Somente os pontos com índice global $k = 5$ e

$k = 6$ não são condições de contorno, o que resutla no sistema linear:

$$\begin{bmatrix} 1 & & & & & & & & & & & \\ & 1 & & & & & & & & & & \\ & & 1 & & & & & & & & & \\ & & & 1 & & & & & & & & \\ & & & & 1 & & & & & & & \\ & c & & & b & a & b & & & c & & \\ & & c & & & b & a & b & & & c & \\ & & & & & & & 1 & & & & \\ & & & & & & & & 1 & & & \\ & & & & & & & & & 1 & & \\ & & & & & & & & & & 1 & \\ & & & & & & & & & & & 1 \end{bmatrix} \begin{bmatrix} u_{0,0} \\ u_{1,0} \\ u_{2,0} \\ u_{3,0} \\ u_{0,1} \\ u_{1,1} \\ u_{2,1} \\ u_{3,1} \\ u_{0,2} \\ u_{1,2} \\ u_{2,2} \\ u_{3,2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ f_{1,1} \\ f_{2,1} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.15}$$

em que $a = 2/h_x^2 + 2/h_y^2 = 26$, $b = -1/h_x^2 = -9$ e $c = -1/h_y^2 = -4$. A matriz A não é simétrica e seu número de condição na norma 2 é $k(A) = ||A||_2||A^{-1}||_2 = 43.16$. O código 3.6 escrito em Octave ilustra a implementação desse primeiro exemplo.

```
% Poisson equation -\nablaÂš u = f with dirichlet conditions u = 0

close all;
clear all;

mx = 5;
my = 7;
hx = 1/(mx-1);
hy = 1/(my-1);

x_p = linspace(0,1,(mx));
y_p = linspace(0,1,(my));
f = zeros(1,mx*my);

for i=1:mx
  for j=1:my
  if (i ~= 1 & i ~= mx & j ~= 1 & j ~= my)
    f(1,i + (j-1)*mx) = 2*(1-6*x_p(i)^2)*y_p(j)^2*(1-y_p(j)^2)+...
    2*(1-6*y_p(j)^2)*x_p(i)^2*(1-x_p(i)^2);
    uex(1,i + (j-1)*mx) = (x_p(i)^2-x_p(i)^4)*(y_p(j)^4-y_p(j)^2);
  end
  end
end

A = eye(mx*my);

for i = 1:mx
```

```
28   for j= 1:my
29     if (i ~= 1 & i ~= mx & j ~= 1 & j ~= my)
30       A(i + (j-1)*mx,i + (j-1)*mx) = 2*(1/hy^2 + 1/hx^2);
31       A(i + (j-1)*mx,i + (j-1)*mx-1) = -1/hx^2;
32       A(i + (j-1)*mx,i + (j-1)*mx+1) = -1/hx^2;
33       A(i + (j-1)*mx,i + (j-1)*mx+mx) = -1/hy^2;
34       A(i + (j-1)*mx,i + (j-1)*mx-mx) = -1/hy^2;
35     end
36   end
37 end
38
39 u = A\f';
40
41 spy(A)
```

Listing 3.6: Primeiro exemplo.

Cuja solução é:

```
u =
   0.000000
  -0.000000
  -0.000000
   0.000000
  -0.000000
  -0.012042
  -0.037704
   0.000000
   0.000000
   0.000000
   0.000000
   0.000000
```

Dois detalhes importantes devem ser considerados nessa primeira discretização. O primeiro é que a equação (3.10) tem diferentes escalas ou ordem de grandeza. Por exemplo, ao tomar $m_x = m_y = 1001$, tem-se $h_x = h_y = 0,001$ o que resulta em coeficientes para os nodos internos do domínio da ordem de $4/0.001^2 = x \times 10^6$, enquanto que os coeficientes para os nodos do contorno são iguais a 1. Para mitigar esse problema, pode-se multiplicar (3.10) pela área do elemento de malha, $h_x h_y$, resultando em:

$$2(a+b)u_{i,j} - a(u_{i+1,j} + u_{i-1,j}) - a(u_{i,j+1} + u_{i,j-1}) = h_x h_y f(x_i, y_j) \tag{3.16}$$

em que $a = h_y/h_x$ e $b = h_x/h_y$.

Em segundo, as equações de diferenças finitas podem ser reescritas para formar uma matriz A simétrica. Por exemplo, em um ponto da malha adjacente ao limite esquerdo do domínio, o caso $i = 1$ de (3.16), o valor de localização da incógnita $u_{0,j}$ aparece na equação. A matriz do sistema linear será simétrica ao mover tais valores para o vetor do lado direito, $\boldsymbol{b}$. Isso é possível pois o valor $u_{0,j}$ é explicitado pelas condições de contorno. Isso converte as entradas das sub-diagonais de A para zero nas colunas correspondentes aos valores de contorno conhecidos. Desta forma, pode-se resolver o sistema de equações por métodos mais eficientes, como gradientes conjugados, KSP e pré-condicionadores de Cholesky. Ao realizar estas duas alterações no sistema linear (3.15), tem-se:

$$\begin{bmatrix} 1 & & & & & & & & & & & \\ & 1 & & & & & & & & & & \\ & & 1 & & & & & & & & & \\ & & & 1 & & & & & & & & \\ & & & & 1 & & & & & & & \\ & & & & & \alpha & \beta & & & & & \\ & & & & & \beta & \alpha & & & & & \\ & & & & & & & 1 & & & & \\ & & & & & & & & 1 & & & \\ & & & & & & & & & 1 & & \\ & & & & & & & & & & 1 & \\ & & & & & & & & & & & 1 \end{bmatrix} \begin{bmatrix} u_{0,0} \\ u_{1,0} \\ u_{2,0} \\ u_{3,0} \\ u_{0,1} \\ u_{1,1} \\ u_{2,1} \\ u_{3,1} \\ u_{0,2} \\ u_{1,2} \\ u_{2,2} \\ u_{3,2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ (1/6)f_{1,1} \\ (1/6)f_{2,1} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.17}$$

em que $\alpha = 2(h_x/h_y + h_y/h_x) = 13/3$, $\beta = -h_y/h_x = -3/2$. A nova matriz A é simétrica e positiva definida, está melhor escalada que antes e tem um número de condição $k(A) = 5.83$.

3.3.5 Montagem da matriz A em PETSc

A função denominada `formMatrix()` no Código 3.7 é responsável pela montagem da matriz A do sistema linear (3.17).

```
1  PetscErrorCode formMatrix(DM da, Mat A) {
2    PetscErrorCode ierr;
3    DMDALocalInfo info;
4    MatStencil     row, col[5];
5    double         hx, hy, v[5];
6    int            i, j, ncols;
7
8    ierr = DMDAGetLocalInfo(da,&info); CHKERRQ(ierr);
9    hx = 1.0/(info.mx-1);  hy = 1.0/(info.my-1);
10   for (j = info.ys; j < info.ys+info.ym; j++) {
11     for (i = info.xs; i < info.xs+info.xm; i++) {
12       row.j = j;            // linha de A correspondente ao elemento (x_i,y_j)
```

```
13        row.i = i;
14        col[0].j = j;
15        col[0].i = i;
16        ncols = 1;
17        if (i==0 || i==info.mx-1 || j==0 || j==info.my-1) {
18          v[0] = 1.0;  // Se estiver na diagonal
19        } else {
20          v[0] = 2*(hy/hx + hx/hy); // ... senão monta a linha toda
21          if (i-1 > 0) {
22            col[ncols].j = j;    col[ncols].i = i-1;  v[ncols++] = -hy/hx;  }
23          if (i+1 < info.mx-1) {
24            col[ncols].j = j;    col[ncols].i = i+1;  v[ncols++] = -hy/hx;  }
25          if (j-1 > 0) {
26            col[ncols].j = j-1;  col[ncols].i = i;    v[ncols++] = -hx/hy;  }
27          if (j+1 < info.my-1) {
28            col[ncols].j = j+1;  col[ncols].i = i;    v[ncols++] = -hx/hy;  }
29        }
30    ierr = MatSetValuesStencil(A,1,&row,ncols,col,v,INSERT_VALUES);
31    CHKERRQ(ierr);
32      }
33    }
34    ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
35    ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
36    return 0;
37  }
38  //ENDMATRIX
39
40  //STARTEXACT
```

Listing 3.7: formMatrix()

A variável `DMDALocalInfo info` declarada no Código 3.7 precisa de uma descrição especial. Ela é uma estrutura de inteiros em C, definida pela biblioteca PETSc, e serve para descrever tanto o tamanho da malha global como a extensão das submalhas locais. A extensão da malha global está nos membros `info.mx` e `info.my`. Os processos locais possem sub-malhas retangulares com dimensão `info.xm` e `info.ym`, com um intervalo de índices locais em duas dimensões da forma:

$$\texttt{info.xs} \leq i \leq \texttt{info.xs} + \texttt{info.xm} - 1$$
$$\texttt{info.ys} \leq j \leq \texttt{info.ys} + \texttt{info.ym} - 1$$

Para melhor compreensão, considere a malha da Figura 3.11, em que `info.mx`=5 e `info.my`=7. Ao considerar o caso em que são empregados 4 processos MPI, tem-se que os processos com rank 0 e 2 têm `info.xs`=0 e `info.xm`=3, por outro lado, os processos com rank 1 e 3 têm `info.xs`=3 e `info.xm`=2, respectivamente. Para a coordenada j, os processos com rank 0 e 1 têm `info.ys`=0 e `info.ym`=4, por outro lado, os processos com rank 2 e 3 têm `info.ys`=4 e

`info.xm`=3, respectivamente. Com esses índices, é possível montar a matriz bidimensional em paralelo.

Em particular, os índices locais (i, j) podem ser usados para inserir entradas na `Mat A`. No Código 3.7, vemos um uso de `MatSetValuesStencil()` para cada ponto da malha local. Para um ponto interior genérico, este comando insere cinco coeficientes na matriz, cuja estrutura de dados é do tipo `MatStencil`, que é uma estrutura da biblioteca PETSc que apresenta quatro valores inteiros, k, j, i, c. No caso 2D, com um único grau de liberdade em cada nodo, usamos apenas os membros i e j da estrutura `MatStencil`. A partir da discretização (3.16), as entradas da matriz são $a_{i,i} = 2(h_y/h_x + h_x/h_y)$ na diagonal e $a_{i,j} = -h_y/h_x$ ou $a_{i,j} = -h_x/h_y$ para as diagonais secundárias.

3.3.6 Um problema particular

Neste seção, será especificada uma função $u(x, y)$ para que o problema de Poisson tenha uma solução exata que também satisfaça as condições de contorno de Dirichlet homogêneas ($u = 0$ ao longo de ∂S):

$$u(x, y) = (x^2 - x^4)(y^4 - y^2) \tag{3.18}$$

Cujo Laplaciano é igual a $f = -\nabla^2 u$:

$$f(x, y) = 2(1 - 6x^2)y^2(1 - y^2) + 2(1 - 6y^2)x^2(1 - x^2) \tag{3.19}$$

A expressão (3.18) será referenciada como solução exata, u_{ex} a partir de agora. Este mesmo problema é ilustrado no capítulo 4 [**?**] e em [4],

O Código 3.8 mostra como a solução exata é implementado na função `formExact()`, e mostra como (3.19) é implementada na função `formRHS()`. Os cálculos nestes códigos usam apenas coordenadas da malha local (i, j). Ou seja, a aritmética de ponteiros PETSc permite indexar as arrays que obtemos de `DMDAVecGetArray()` usando as coordenadas da malha i e j, sem conhecer os índices das incógnitas no `Vec u`, considerado como vetor de coluna de comprimento $L = m_x m_y$. Ao terminar de construir os vetores, restaura-se as arrays chamando a função `DMDAVecRestoreArray()`.

```
1  int            i, j;
2  double         hx, hy, x, y, **auexact;
3
4  ierr = DMDAGetLocalInfo(da,&info); CHKERRQ(ierr);
5  hx = 1.0/(info.mx-1);  hy = 1.0/(info.my-1);
6  ierr = DMDAVecGetArray(da, uexact, &auexact);CHKERRQ(ierr);
7  for (j = info.ys; j < info.ys+info.ym; j++) {
8    y = j * hy;
9    for (i = info.xs; i < info.xs+info.xm; i++) {
```

```
10        x = i * hx;
11        auexact[j][i] = x*x * (1.0 - x*x) * y*y * (y*y - 1.0);
12      }
13    }
14    ierr = DMDAVecRestoreArray(da, uexact, &auexact);CHKERRQ(ierr);
15    return 0;
16  }
17
18  PetscErrorCode formRHS(DM da, Vec b) {
19    PetscErrorCode ierr;
20    int            i, j;
21    double         hx, hy, x, y, f, **ab;
22    DMDALocalInfo info;
23
24    ierr = DMDAGetLocalInfo(da,&info); CHKERRQ(ierr);
25    hx = 1.0/(info.mx-1);  hy = 1.0/(info.my-1);
26    ierr = DMDAVecGetArray(da, b, &ab);CHKERRQ(ierr);
27    for (j=info.ys; j<info.ys+info.ym; j++) {
28      y = j * hy;
29      for (i=info.xs; i<info.xs+info.xm; i++) {
30        x = i * hx;
31        if (i==0 || i==info.mx-1 || j==0 || j==info.my-1) {
32          ab[j][i] = 0.0;
33        } else {
34          f = 2.0 * ( (1.0 - 6.0*x*x) * y*y * (1.0 - y*y)
35                      + (1.0 - 6.0*y*y) * x*x * (1.0 - x*x) );
36          ab[j][i] = hx * hy * f;
37        }
38      }
39    }
40    ierr = DMDAVecRestoreArray(da, b, &ab); CHKERRQ(ierr);
41    return 0;
42  }
43  //ENDRHS
44
45  //STARTCREATE
```

Listing 3.8: `formExact()` e `formRHS`

3.3.7 Resolução do sistema de equações

O Código 3.9 mostra a função principal e os objetos necessários para resolver o problema de Poisson: um `DM`, um `Mat` e três `Vec`. Um objeto `DM` determina as dimensões da matriz e dos vetores a partir das dimensões da malha, o que é feito quando chamamos `DMCreateMatrix()` e `DMCreateGlobalVector()` para criar objetos `Mat` e `Vec`, respectivamente. Então invoca-se as

funções mostradas nos Códigos 3.7 e 3.8 para preencher a matriz e os vetores.

```
1   Mat            A;
2   Vec            b,u,uexact;
3   KSP            ksp;
4   double         errnorm;
5   DMDALocalInfo  info;
6
7   PetscInitialize(&argc,&args,(char*)0,help);
8
9   // Tamanho default (9 x 9) pode ser modificado -da_grid_x M -da_grid_y N
10  ierr = DMDACreate2d(PETSC_COMM_WORLD,
11                DM_BOUNDARY_NONE, DM_BOUNDARY_NONE, DMDA_STENCIL_STAR,
12                -9,-9,PETSC_DECIDE,PETSC_DECIDE,1,1,NULL,NULL,
13                &da); CHKERRQ(ierr);
14
15  // Cria matriz do sistema A
16  ierr = DMCreateMatrix(da,&A);CHKERRQ(ierr);
17  ierr = MatSetOptionsPrefix(A,"a_"); CHKERRQ(ierr);
18  ierr = MatSetFromOptions(A); CHKERRQ(ierr);
19
20  // Cria vetor (RHS) 'b', aprox solução 'u', exata solução 'uexact'
21  ierr = DMCreateGlobalVector(da,&b);CHKERRQ(ierr);
22  ierr = VecDuplicate(b,&u); CHKERRQ(ierr);
23  ierr = VecDuplicate(b,&uexact); CHKERRQ(ierr);
24
25  // Preenche vetores e monta matriz
26  ierr = formExact(da,uexact); CHKERRQ(ierr);
27  ierr = formRHS(da,b); CHKERRQ(ierr);
28  ierr = formMatrix(da,A); CHKERRQ(ierr);
29 //ENDCREATE
30 //STARTSOLVE
31  // Resolve sistema linear
32  ierr = KSPCreate(PETSC_COMM_WORLD,&ksp); CHKERRQ(ierr);
33  ierr = KSPSetOperators(ksp,A,A); CHKERRQ(ierr);
34  ierr = KSPSetFromOptions(ksp); CHKERRQ(ierr);
35  ierr = KSPSolve(ksp,b,u); CHKERRQ(ierr);
36
37  // Resultados
38  ierr = VecAXPY(u,-1.0,uexact); CHKERRQ(ierr);    // u <- u + (-1.0) uxact
39  ierr = VecNorm(u,NORM_INFINITY,&errnorm); CHKERRQ(ierr);
40  ierr = DMDAGetLocalInfo(da,&info);CHKERRQ(ierr);
41  ierr = PetscPrintf(PETSC_COMM_WORLD,
42              "Malha %d x %d :  erro |u-uexact|_inf = %g\n",
43              info.mx,info.my,errnorm); CHKERRQ(ierr);
44
45  VecDestroy(&u);  VecDestroy(&uexact);  VecDestroy(&b);
46  MatDestroy(&A);  KSPDestroy(&ksp);  DMDestroy(&da);
```

```
47   PetscFinalize();
48   return 0;
49 }
50 //ENDSOLVE
```

Listing 3.9: formExact()

Como já realizado, o sistema linear é resolvido por um objeto de solução linear KSP. O sistema é resolvido chamando `KSPSolve()`. Então calcula-se e imprimi-se na tela o erro numérico $||u - u_{ex}||$. Finalmente, destruímos objetos com o método `XXXDestroy()` e chamamos `PetscFinalize()` para encerrar. Uma primeira execução da solução pode ser feita através do comando:

```
user@user-G73Sw:~$ make poisson_fd_2d
user@user-G73Sw:~$ ./poisson_fd_2d -ksp_monitor
0 KSP Residual norm 1.020952970432e-01
1 KSP Residual norm 2.656923348626e-02
2 KSP Residual norm 8.679141000397e-03
3 KSP Residual norm 1.557150861763e-03
4 KSP Residual norm 2.239919982542e-04
5 KSP Residual norm 2.519822315367e-05
6 KSP Residual norm 2.152764600588e-06
7 KSP Residual norm 2.650467236964e-07
on 9 x 9 grid:  error |u-uexact|_inf = 0.000763959|
```

Pode-se examinar graficamente a malha para verificar a indexação dos objetos, em tempo de execução Assim, se o sistema X-window estiver corretamente configurado na instalação PETSc, então o comando:

```
user@user-G73Sw:~$ ./poisson_fd_2d -da_grid_x 5 -da_grid_y 7
-dm_view draw -draw_pause 5
```

irá plotar a Figura 3.15.

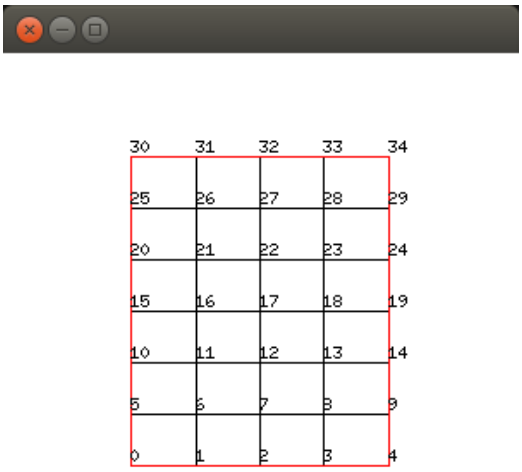


Figura 3.15: Malha computacional.

Outra visualização interessante é obtida com o comando:

```
user@user-G73Sw:~$ ./poisson_fd_2d -da_grid_x 5 -da_grid_y 7
-a_mat_view draw -draw_pause 5
```

que apresenta a estrutura da matriz A, conforme Figura 3.16.

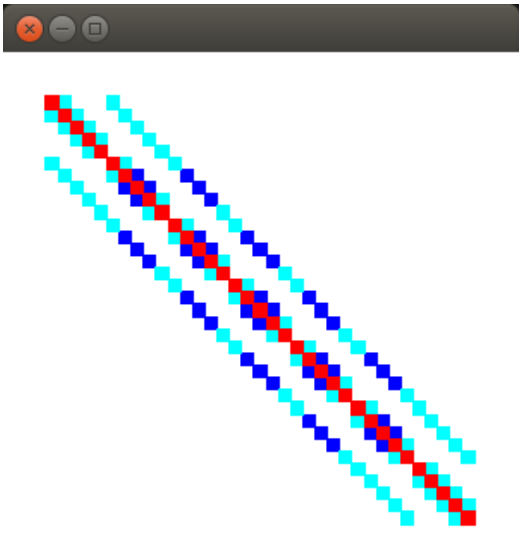


Figura 3.16: Estrutura da matriz A.

É possível também observar a variação do erro conforme o refinamento da malha com o comando:

```
user@user-G73Sw:~$ for K in 0 1 2 3 4 5; do ./poisson_fd_2d -da_refine $K; done
```

```
on 9 x 9 grid:  error u-uexact_inf = 0.000763959
on 17 x 17 grid:  error u-uexact_inf = 0.000196764
on 33 x 33 grid:  error u-uexact_inf = 4.91557e-05
on 65 x 65 grid:  error u-uexact_inf = 1.29719e-05
on 129 x 129 grid:  error u-uexact_inf = 3.76924e-06
on 257 x 257 grid:  error u-uexact_inf = 1.73086e-06
```

Note agora a diferença entre tempos de processamento para o código serial e utilizando 2 processadores.

```
user@user-G73Sw:~$ time ./poisson_fd_2d -da_refine 6
real 0m56.467s
user@user-G73Sw:~$ time mpiexec -n 2 ./poisson_fd_2d -da_refine 6
real 1m0.327s
```

A solução do problema de Poisson na malha 4×3 é:

```
0.
0.
0.
0.
0.
-0.0120422
-0.0377036
0.
0.
0.
0.
0.
```

o que é bem semelhante à obtida com o Código 3.6 implementado em Octave.

3.4 Agenda de atividades

- 28 e 29/03 - Conclusão da seção Poisson 1D;
- 30 e 31/03 - Iniciar redação da seção de revisão em Turbulência;
- 01/04 - 10/04 - Implementação da equação do calor trasiente 1D e 2D + relatório Turbulência;

- 11/04 - 30/04 - Implementação da equação de Stokes 2D e 3D + relatório Turbulência;

AGENDA ABRIL/2017

* FAZER DETALHES DIFERENÇAS FINITAS NO RELATÓRIO
* ALTERAR CONDIÇÕES DE CONTORNO
* TRABALHAR NA VISUALIÇÃO DE RESULTADOS (VTK ou OUTRO)
* IMPLEMENTAR DIFERENTES ORDENS EM PETSC
* CRIAR FUNÇÃO PLOTAR ORDEM ERRO
* IMPLEMENTAR EQUAÇÃO CALOR TRANSIENTE
* IMPLEMENTAR EQUAÇÃO STOKES
* UM EXEMPLO EM MALHA NÃO ESTRUTURADA

MAIO

* IMPLEMENTAR AS EQUAÇÕES EM VOLUMES FINITOS

Reunião Fabricio 07/04/2017 * Empregar técnicas implicitas na equação do calor

- Revisar C
- Estudar MPI e PETSC
- Instalar e configurar MPI e PETSC ('pet-see')
- Primeiros exemplos utilizando PETSC;
- Leituras iniciais sobre turbulência
- Revisão sobre Mecância dos Fluídos
-
- Revisão sobre Volumes Finitos
- Implementações iniciais

Referências Bibliográficas

[1] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc Web page. http://www.mcs.anl.gov/petsc, 2016.

[2] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.7, Argonne National Laboratory, 2016.

[3] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[4] Ed Bueler. *PETSc for Partial Differential Equations*. SIAM, 2018.