

Relatório de estudos individual

Péttersen Vinícius Pramiu

26 de março de 2017

Sumário

1	Introdução	2
2	Instalação e Configuração do PETSC	3
2.1	Primeiros exemplos com PETSC	7
2.1.1	Objetos do tipo vetores e matrizes em PETSC	11
2.2	A equação de Poisson em uma malha estruturada	19
2.3	Geração da malha	20
2.4	Diferenças finitas	21

Capítulo 1

Introdução

Este relatório objetiva registrar de forma detalhada e sistemática as atividades de pesquisa e estudos relacionados à temática de proposta de tese do autor. Nele serão revistos alguns conceitos fundamentais de mecânica de fluidos, métodos numéricos em equações diferenciais, instalação e configuração de bibliotecas para implementação computacional como PETSC e MPI, entre outros. Exemplos da resolução de sistemas de equações lineares e da solução de algumas EDPs como a equação de Laplace, de Poisson, da equação do calor transiente e da equação de Stokes serão resolvidos detalhadamente em diferenças finitas ou volumes finitos, empregando malhas estruturadas e em programação serial ou paralela. A linguagem selecionada para implementação dos códigos fonte é a linguagem C, e alguns exemplos também serão ilustrados e implementados em Scilab. Com a realização destes exemplos espera-se obter os requisitos necessários para compreensão de trabalhos e implementação de código computacional envolvendo simulações em escoamentos turbulentos empregando o método de Simulação de Grandes Escalas - *Large Eddy Simulation*.

Capítulo 2

Instalação e Configuração do PETSC

Esta seção é breve e 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 2.1 deve ser exibida no terminal.

```
user@user-G73Sw: ~  
PETSc:  
PETSC_ARCH: arch-linux2-c-debug  
PETSC_DIR: /home/user/Documentos/petsc-3.7.5  
Scalar type: real  
Precision: double  
Language: C  
shared libraries: enabled  
Integer size: 32  
Memory alignment: 16  
XXX=====XXX  
Configure stage complete. Now build PETSc libraries with (gnumake build):  
make PETSC_DIR=/home/user/Documentos/petsc-3.7.5 PETSC_ARCH=arch-linux2-c-debug all  
XXX=====XXX  
root@user-G73Sw:/home/user/Documentos/petsc-3.7.5#
```

Figura 2.1: 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 2.2.

```
user@user-G73Sw: ~  
CC arch-linux2-c-debug/obj/src/tao/linesearch/interface/dlregis_tao_linesearch.o  
CC arch-linux2-c-debug/obj/src/tao/linesearch/interface/ftn-auto/tao_linesearchf.o  
CC arch-linux2-c-debug/obj/src/tao/linesearch/interface/ftn-custom/ztao_linesearchf.o  
CC arch-linux2-c-debug/obj/src/tao/least_squares/impls/pounders/gqt.o  
CC arch-linux2-c-debug/obj/src/tao/linesearch/interface/tao_linesearch.o  
CC arch-linux2-c-debug/obj/src/tao/least_squares/impls/pounders/pounders.o  
LINKER /home/user/Documentos/petsc-3.7.5/arch-linux2-c-debug/lib/libpetsc.so.3.7.5  
make[2]: Leaving directory '/home/user/Documentos/petsc-3.7.5'  
=====  
make[1]: Leaving directory '/home/user/Documentos/petsc-3.7.5'  
Now to check if the libraries are working do:  
make PETSC_DIR=/home/user/Documentos/petsc-3.7.5 PETSC_ARCH=arch-linux2-c-debug test  
=====  
root@user-G73Sw:/home/user/Documentos/petsc-3.7.5#
```

Figura 2.2: 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 2.3.

```
user@user-G73Sw: ~  
root@user-G73Sw:/home/user/Documentos/petsc-3.7.5# make PETSC_DIR=/home/user/Documentos/petsc-3.7.5 PETSC_ARCH=arch-linux2-c-debug test  
Running test examples to verify correct installation  
Using PETSC_DIR=/home/user/Documentos/petsc-3.7.5 and PETSC_ARCH=arch-linux2-c-debug  
C/C++ example src/snes/examples/tutorials/ex19 run successfully with 1 MPI process  
C/C++ example src/snes/examples/tutorials/ex19 run successfully with 2 MPI processes  
Fortran example src/snes/examples/tutorials/ex5f run successfully with 1 MPI process  
Completed test examples  
=====  
Now to evaluate the computer systems you plan use - do:  
make PETSC_DIR=/home/user/Documentos/petsc-3.7.5 PETSC_ARCH=arch-linux2-c-debug streams  
root@user-G73Sw:/home/user/Documentos/petsc-3.7.5#
```

Figura 2.3: 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 2.4.

```
user@user-G73Sw: ~  
user-G73Sw user-G73Sw  
Triad: 12829.9201 Rate (MB/s)  
Number of MPI processes 6 Processor names user-G73Sw user-G73Sw user-G73Sw  
user-G73Sw user-G73Sw user-G73Sw  
Triad: 12768.2064 Rate (MB/s)  
Number of MPI processes 7 Processor names user-G73Sw user-G73Sw user-G73Sw  
user-G73Sw user-G73Sw user-G73Sw user-G73Sw  
Triad: 12528.9329 Rate (MB/s)  
Number of MPI processes 8 Processor names user-G73Sw user-G73Sw user-G73Sw  
user-G73Sw user-G73Sw user-G73Sw user-G73Sw  
Triad: 12347.7850 Rate (MB/s)  
-----  
np speedup  
1 1.0  
2 1.26  
3 1.24  
4 1.21  
5 1.19  
6 1.18  
7 1.16  
8 1.14  
Estimation of possible speedup of MPI programs based on Streams benchmark.  
It appears you have 1 node(s)  
See graph in the file src/benchmarks/streams/scaling.png
```

Figura 2.4: Terminal configuração PETSC.

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

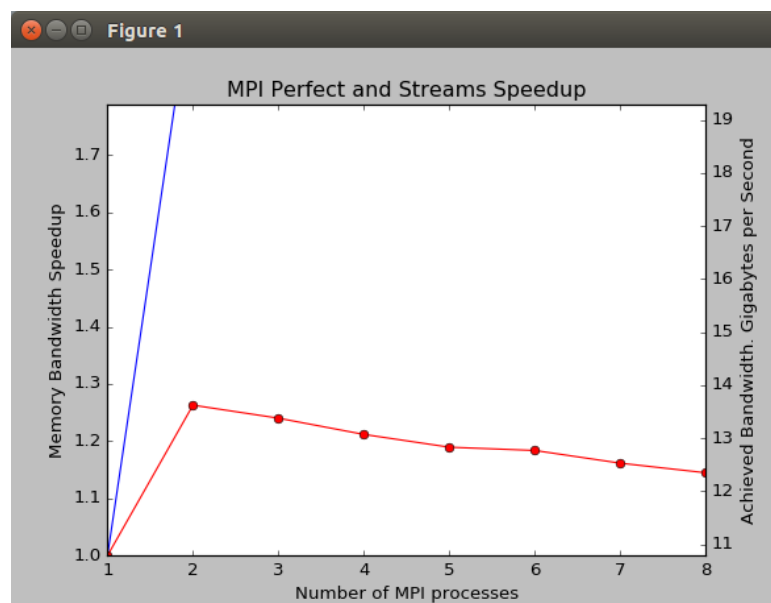


Figura 2.5: Terminal configuração PETSC.

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 2.6.

```

user@user-G73Sw: ~
Using PETSC_DIR=/home/user/Documentos/petsc-3.7.5 and PETSC_ARCH=arch-linux2-c-debug
C/C++ example src/snes/examples/tutorials/ex19 run successfully with 1 MPI process
C/C++ example src/snes/examples/tutorials/ex19 run successfully with 2 MPI processes
Fortran example src/snes/examples/tutorials/ex5f run successfully with 1 MPI process
Completed test examples
=====
Now to evaluate the computer systems you plan use - do:
make PETSC_DIR=/home/user/Documentos/petsc-3.7.5 PETSC_ARCH=arch-linux2-c-debug streams
root@user-G73Sw: /home/user/Documentos/petsc-3.7.5# export PETSC_ARCH=arch-linux2-c-debug
root@user-G73Sw: /home/user/Documentos/petsc-3.7.5# export PETSC_DIR=/home/user/Documentos/petsc-3.7.5
root@user-G73Sw: /home/user/Documentos/petsc-3.7.5#

```

Figura 2.6: 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 2.7.

```

user@user-G73Sw: ~/Documentos/petsc-3.7.5/src/ksp/ksp/examples/tutorials
user@user-G73Sw:~$ cd /home/user/Documentos/petsc-3.7.5/src/ksp/ksp/examples/tutorials/
user@user-G73Sw:~/Documentos/petsc-3.7.5/src/ksp/ksp/examples/tutorials$ make ex1
/home/user/Documentos/petsc-3.7.5/arch-linux2-c-debug/bin/mpicc -o ex1.o -c -Wall -Wwr
te-strings -Wno-strict-aliasing -Wno-unknown-pragmas -fvisibility=hidden -g3 -I/home/
user/Documentos/petsc-3.7.5/include -I/home/user/Documentos/petsc-3.7.5/arch-linux2-c-d
ebug/include -pwd' /ex1.c
/home/user/Documentos/petsc-3.7.5/arch-linux2-c-debug/bin/mpicc -Wall -Wwrite-strings -
Wno-strict-aliasing -Wno-unknown-pragmas -fvisibility=hidden -g3 -o ex1 ex1.o -WL,-rpa
th,/home/user/Documentos/petsc-3.7.5/arch-linux2-c-debug/lib -L/home/user/Documentos/
petsc-3.7.5/arch-linux2-c-debug/lib -lpetsc -L,-rpath,/home/user/Documentos/petsc-3.7
.5/arch-linux2-c-debug/lib -lfblas -lX11 -lhwloc -lpthread -lm -L,-rpath,/usr/
lib/gcc/x86_64-linux-gnu/5 -L,-rpath,/usr/lib/gcc/x86_64-linux-gnu/5 -WL,-rpath,/usr/lib/x86_
64-linux-gnu -L/usr/lib/x86_64-linux-gnu -WL,-rpath,/lib/x86_64-linux-gnu -L/lib/x86_64
-linux-gnu -Lmpi4fort -lgfortran -lm -lgfortran -lm -lquadmath -lm -Lmpicxx -lstdc++ -Wl,
-rpath,/home/user/Documentos/petsc-3.7.5/arch-linux2-c-debug/lib -L/home/user/Document
os/petsc-3.7.5/arch-linux2-c-debug/lib -L,-rpath,/usr/lib/gcc/x86_64-linux-gnu/5 -L/us
r/lib/gcc/x86_64-linux-gnu/5 -WL,-rpath,/usr/lib/x86_64-linux-gnu -L/usr/lib/x86_64-lin
ux-gnu -WL,-rpath,/lib/x86_64-linux-gnu -L/lib/x86_64-linux-gnu -WL,-rpath,/usr/lib/x86_
64-linux-gnu -L/usr/lib/x86_64-linux-gnu -ldl -WL,-rpath,/home/user/Documentos/petsc-3
.7.5/arch-linux2-c-debug/lib -lmpi -lgcc_s -ldl
/bin/rm -f ex1.o
user@user-G73Sw:~/Documentos/petsc-3.7.5/src/ksp/ksp/examples/tutorials$

```

Figura 2.7: Terminal exemplo ex1.c.

Para executar o código acima compilado, digita-se `./ex1`, o que deve resultar em algo semelhante à Figura 2.8.

```

user@user-G73Sw: ~/Documentos/petsc-3.7.5/src/ksp/ksp/examples/tutorials
user@user-G73Sw:~/Documentos/petsc-3.7.5/src/ksp/ksp/exanples/tutorials$ ./ex1
KSP Object: 1 MPI processes
  type: gmres
    GMRES: restart=30, using Classical (unmodified) Gram-Schmidt Orthogonalization with
no iterative refinement
    GMRES: happy breakdown tolerance 1e-30
    maximum iterations=10000, initial guess is zero
    tolerances: relative=1e-05, absolute=1e-50, divergence=10000.
    left preconditioning
    using PRECONDITIONED norm type for convergence test
PC Object: 1 MPI processes
  type: jacobi
  linear system matrix = precondition matrix:
  Mat Object: 1 MPI processes
    type: seqaij
    rows=10, cols=10
    total: nonzeros=28, allocated nonzeros=50
    total number of mallocs used during MatSetValues calls =0
    not using I-node routines
  Norm of error 2.41202e-15, Iterations 5
user@user-G73Sw:~/Documentos/petsc-3.7.5/src/ksp/ksp/exanples/tutorials$

```

Figura 2.8: Execução do exemplo ./ex1

Para uma configuração 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`.

2.1 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 suite 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 propósito da biblioteca PETSC é auxiliar na solução de problemas científicos e de engenharia em computadores multiprocessados. O primeiro exemplo aqui ilustrado é apresentado em [Bueler(2018)] e aproxima a constante de Euler por meio da série de Maclaurin:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} \approx 2.718281828 \quad (2.1)$$

O programa `code_euler.c`, ilustrado no código 2.1, 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.

Listing 2.1: Código constante de Euler.

```

#include <petsc.h>

int main(int argc, char **argv) {
    PetscErrorCode ierr;
    int rank, i;
    double localval, globalsum;

```



```

PetscInitialize(&argc,&argv,NULL,"Calcula 'e' em paralelo com PETSc.\n\n");

ierr = MPI_Comm_rank(PETSC_COMM_WORLD,&rank); CHKERRQ(ierr);

// calcula 1 / n! onde n = (rank do processo) + 1
localval = 1.0;
for (i = 2; i < rank+1; i++)
    localval /= i;

// soma as contribuições de cada processo
ierr = MPI_Allreduce(&localval, &globalsum, 1, MPI_DOUBLE, MPI_SUM,
                    PETSC_COMM_WORLD); CHKERRQ(ierr);

// imprime a estimativa de e
ierr = PetscPrintf(PETSC_COMM_WORLD,
    "O valor da constante 'e' é aproximadamente: %17.15f\n", globalsum); CHKERRQ(ierr);
ierr = PetscPrintf(PETSC_COMM_SELF,
    "rank %d did %d flops\n", rank, (rank > 0) ? rank-1 : 0); CHKERRQ(ierr);

PetscFinalize();

return 0;
}

```

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 verificação 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 [?] 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 ??.

```

include ${PETSC_DIR}/lib/petsc/conf/variables
include ${PETSC_DIR}/lib/petsc/conf/rules

code_euler: code_euler.o  chkopts
    -${CLINKER} -o code_euler code_euler.o  ${PETSC_LIB}
    ${RM} code_euler.o

build_vector_01: build_vector_01.o  chkopts
    -${CLINKER} -o build_vector_01 build_vector_01.o  ${PETSC_LIB}

```

```
    ${RM} build_vector_01.o
```

```
build_vector: build_vector.o  chkopts
    -${CLINKER} -o build_vector build_vector.o  ${PETSC_LIB}
    ${RM} build_vector.o
```

```
build_matrix: build_matrix.o  chkopts
    -${CLINKER} -o build_matrix build_matrix.o  ${PETSC_LIB}
    ${RM} build_matrix.o
```

```
solve_linear_ksp: solve_linear_ksp.o  chkopts
    -${CLINKER} -o solve_linear_ksp solve_linear_ksp.o  ${PETSC_LIB}
    ${RM} solve_linear_ksp.o
```

```
solve_linear_arbitrary: solve_linear_arbitrary.o  chkopts
    -${CLINKER} -o solve_linear_arbitrary solve_linear_arbitrary.o  ${PETSC_LIB}
    ${RM} solve_linear_arbitrary.o
```

Após ter criado 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 basta digitar

```
user@user-G73Sw:~$ ./code_euler
O valor da constante 'e' é aproximadamente: 1.0000000000000000
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
0 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 (processador 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 (2.1) 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 comando 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
0 valor da constante 'e' é aproximadamente: 2.718281525573192
rank 0 did 0 flops
.....

```

2.1.1 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 do PDE à medida que seu tamanho vai para o infinito, busca-se resolver os maiores sistemas lineares que a tecnologia de computação disponível possa ser capaz de suportar. 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 se encaixar na memória - representa a tecnologia de núcleo em PETSc.

Uma observação a ser feita, é que muitas das PDEs discretizadas geram sistemas lineares com estrutura explorável, especialmente a esparsidade, o que significa que há poucas entradas diferentes de zero por linha na matriz. Para que os métodos converjam, 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.

```
static char help[] = "Monta um vetor 10x1 usando Vec.\n";

#include <petsc.h>

int main(int argc, char **args) {
    Vec      x;
    int      i[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    double   v[10] = {11.0, 7.0, 5.0, 3.0, 6.0,
                      11.0, 7.0, 5.0, 3.0, 6.0};

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

    VecCreate(PETSC_COMM_WORLD, &x);
    VecSetSizes(x, PETSC_DECIDE, 10);
    VecSetFromOptions(x);
    VecSetValues(x, 10, i, v, INSERT_VALUES);
    VecAssemblyBegin(x);
    VecAssemblyEnd(x);

    VecDestroy(&x);

    PetscFinalize();
    return 0;
}
```

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

```

include ${PETSC_DIR}/lib/petsc/conf/variables
include ${PETSC_DIR}/lib/petsc/conf/rules

code_euler: code_euler.o  chkopts
    -${CLINKER} -o code_euler code_euler.o  ${PETSC_LIB}
    ${RM} code_euler.o

build_vector_01: build_vector_01.o  chkopts
    -${CLINKER} -o build_vector_01 build_vector_01.o  ${PETSC_LIB}
    ${RM} build_vector_01.o

build_vector: build_vector.o  chkopts
    -${CLINKER} -o build_vector build_vector.o  ${PETSC_LIB}
    ${RM} build_vector.o

build_matrix: build_matrix.o  chkopts
    -${CLINKER} -o build_matrix build_matrix.o  ${PETSC_LIB}
    ${RM} build_matrix.o

solve_linear_ksp: solve_linear_ksp.o  chkopts
    -${CLINKER} -o solve_linear_ksp solve_linear_ksp.o  ${PETSC_LIB}
    ${RM} solve_linear_ksp.o

solve_linear_arbitrary: solve_linear_arbitrary.o  chkopts
    -${CLINKER} -o solve_linear_arbitrary solve_linear_arbitrary.o  ${PETSC_LIB}
    ${RM} solve_linear_arbitrary.o

```

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`:

```
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;
}
```

O resultado ou a matriz criada pode ser visualizada de várias formas, e aqui faremos duas visualizações uma no formato esparsa 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.
```

Como descrito em [?], 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:

$$\begin{bmatrix} 1 & 2 & 3 & 0 \\ 2 & 1 & -2 & -3 \\ -1 & 1 & 1 & 0 \\ 0 & 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 1 \\ 1 \\ 3 \end{bmatrix} \quad (2.2)$$

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 (2.11) é apresentado a seguir:

```

static char help[] = "Resolve um sistema linear 4x4 usando Vec, Mat, and KSP.\n";

#include <petsc.h>

int main(int argc, char **args) {
    Vec      x, b;
    Mat      A;
    KSP      ksp;
    int      i, j[4] = {0, 1, 2, 3};
    double   ab[4] = {7.0, 1.0, 1.0, 3.0};
    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);

```

```

VecCreate(PETSC_COMM_WORLD, &b);
VecSetSizes(b,PETSC_DECIDE,4);
VecSetFromOptions(b);
VecSetValues(b,4,j,ab, INSERT_VALUES);
VecAssemblyBegin(b);
VecAssemblyEnd(b);

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);

KSPCreate(PETSC_COMM_WORLD,&ksp);
KSPSetOperators(ksp, A, A);
KSPSetFromOptions(ksp);
VecDuplicate(b,&x);
KSPSolve(ksp,b,x);
VecView(x,PETSC_VIEWER_STDOUT_WORLD);

KSPDestroy(&ksp);
MatDestroy(&A);
VecDestroy(&x);
VecDestroy(&b);
PetscFinalize();
return 0;
}

```

O resulta 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 resolve 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.....

```
//STARTSETUP
static char help[] =
    "Solve a tridiagonal system of arbitrary size. Option prefix = tri_.\n";

#include <petsc.h>

int main(int argc, char **args) {
    PetscErrorCode ierr;
    Vec      x, b, xexact;
    Mat      A;
    KSP      ksp;
    int      m = 4, i, lstart, lend, j[3];
    double   v[3], xval, ernorm;

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

    ierr = PetscOptionsBegin(PETSC_COMM_WORLD, "tri_", "options for tri_", ""); CHKERRQ(ierr);
    ierr = PetscOptionsInt("-m", "dimension of linear system", "tri.c", m, &m, NULL); CHKERRQ(ierr);
    ierr = PetscOptionsEnd(); CHKERRQ(ierr);

    ierr = VecCreate(PETSC_COMM_WORLD, &x); CHKERRQ(ierr);
    ierr = VecSetSizes(x, PETSC_DECIDE, m); CHKERRQ(ierr);
    ierr = VecSetFromOptions(x); CHKERRQ(ierr);
    ierr = VecDuplicate(x, &b); CHKERRQ(ierr);
    ierr = VecDuplicate(x, &xexact); CHKERRQ(ierr);

    ierr = MatCreate(PETSC_COMM_WORLD, &A); CHKERRQ(ierr);
    ierr = MatSetSizes(A, PETSC_DECIDE, PETSC_DECIDE, m, m); CHKERRQ(ierr);
    ierr = MatSetOptionsPrefix(A, "a_"); CHKERRQ(ierr);
    ierr = MatSetFromOptions(A); CHKERRQ(ierr);
    ierr = MatSetUp(A); CHKERRQ(ierr);
//ENDSETUP
    ierr = MatGetOwnershipRange(A, &lstart, &lend); CHKERRQ(ierr);
    for (i = lstart; i < lend; i++) {
        if (i == 0) {
            v[0] = 3.0; v[1] = -1.0;
            j[0] = 0; j[1] = 1;
            ierr = MatSetValues(A, 1, &i, 2, j, v, INSERT_VALUES); CHKERRQ(ierr);
        } else {
            v[0] = -1.0; v[1] = 3.0; v[2] = -1.0;
            j[0] = i - 1; j[1] = i; j[2] = i + 1;
            if (i == m - 1) {
                ierr = MatSetValues(A, 1, &i, 2, j, v, INSERT_VALUES); CHKERRQ(ierr);
            }
        }
    }
}
```

```

    } else {
        ierr = MatSetValues(A,1,&i,3,j,v,INSERT_VALUES); CHKERRQ(ierr);
    }
}
xval = exp(cos(i));
ierr = VecSetValues(xexact,1,&i,&xval,INSERT_VALUES); CHKERRQ(ierr);
}
ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
ierr = VecAssemblyBegin(xexact); CHKERRQ(ierr);
ierr = VecAssemblyEnd(xexact); CHKERRQ(ierr);
ierr = MatMult(A,xexact,b); CHKERRQ(ierr);

ierr = KSPCreate(PETSC_COMM_WORLD,&ksp); CHKERRQ(ierr);
ierr = KSPSetOperators(ksp,A,A); CHKERRQ(ierr);
ierr = KSPSetFromOptions(ksp); CHKERRQ(ierr);
ierr = KSPSolve(ksp,b,x); CHKERRQ(ierr);

ierr = VecAXPY(x,-1.0,xexact); CHKERRQ(ierr);
ierr = VecNorm(x,NORM_2,&errnorm); CHKERRQ(ierr);
ierr = PetscPrintf(PETSC_COMM_WORLD,
    "error_for_m=%d_system_is_|x-xexact|_2=%%.1e\n",m,errnorm); CHKERRQ(ierr);

KSPDestroy(&ksp); MatDestroy(&A);
VecDestroy(&x); VecDestroy(&b); VecDestroy(&xexact);
PetscFinalize();
return 0;
}
//ENDSOLVE

```

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 fizemos 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 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 exata para o sistema linear ($Ax_{exact} = b$). 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 concluído. 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 caiu de 17 para 5 segundos, indicando um speedup de:

$$546546 \quad (2.3)$$

2.2 A equação de Poisson em uma malha estruturada

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 resolvido em paralelo usando um objeto KSP [Balay et al.(2016)Balay, Abhyankar, and Smith].

Neste exemplo a equação de Poisson será resolvida numa região quadrada aberta, $S, (0, 1) \times (0, 1)$ cujo contorno é representado por ∂S . O problema é ilustrado na Figura ?? e formulado em (2.4).

$$\begin{aligned} -\nabla^2 u &= f & \text{em } S \\ u &= 0 & \text{em } \partial S \end{aligned} \quad (2.4)$$

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

$$\begin{aligned} \nabla^2 u &= \nabla \cdot (\nabla u) \\ &= \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \end{aligned} \quad (2.5)$$

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 às condições de contorno homogêneas de Dirichlet.

O problema de Poisson pode modelar o potencial eletrostático, a distribuição de equilíbrio de certas camadas aleatórias 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 (2.4), que será resolvido numericamente através do método de diferenças finitas.

2.3 Geração da malha

O método de diferenças finitas é desenvolvido sobre uma malha composta por $m_x m_y$ pontos igualmente espaçados, como ilustrado na Figura ??, 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 . Considerando a malha da Figura ?? em $m_x = 5$ e $m_y = 7$, as coordenadas da malha são $(x_i = ih_x, y_j = jh_y)$, para $i = 0, \dots, m_x - 1$ e $j = 0, \dots, m_y - 1$.

Para construção dessa malha 2D de forma distribuída 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 ?? será criada, em que todos os nós 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

??, 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). O bloco de código referente ao emprego do objeto DMDACreate2d é descrito a seguir:

Listing 2.2: DMDA()

```
ierr = DMDACreate2d(PETSC_COMM_WORLD,
                    DM_BOUNDARY_NONE, DM_BOUNDARY_NONE, DMDA_STENCIL_STAR,
                    -9,-9,PETSC_DECIDE,PETSC_DECIDE,1,1,NULL,NULL,
                    &da); CHKERRQ(ierr);
```

No Código 2.2, 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 ?? mostra seu stencil computacional. 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 usando a lógica interna PETSc. 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.

2.4 Diferenças finitas

Considerando uma discretização da equação (2.4) em diferenças finitas, ela pode ser escrita como:

$$-\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h_x^2} - \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h_x^2} = f(x_i, y_j) \quad (2.6)$$

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

$$u_{0,j} = 0, \quad u_{m_x-1,j} = 0, \quad u_{i,0} = 0, \quad u_{i,m_y-1} = 0, \quad (2.7)$$

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:

$$A\mathbf{u} = \mathbf{b} \quad (2.8)$$

em que a matriz A tem dimensão $L \times L$, e os vetores \mathbf{u} e \mathbf{b} , dimensão $L \times 1$. No entanto, para montar as entradas de A e \mathbf{b} no sistema linear (2.8), 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 ???. 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.

Como exemplo, a seguir será construído o sistema de equações para o caso da malha representada na Figura ???, em que $m_x = 4$ e $m_y = 3$, o que resulta em $h_x = 1/3$ e $h_y = 1/2$. Somente os pontos com índice global $k = 5$ e $k = 6$ não são condições de contorno e o sistema linear é dado por:

$$\begin{bmatrix} 1 & & & & & & & & \\ & 1 & & & & & & & \\ & & 1 & & & & & & \\ & & & 1 & & & & & \\ & & & & 1 & & & & \\ & & c & & b & a & b & & c \\ & & & c & & b & a & b & c \\ & & & & & & 1 & & \\ & & & & & & & 1 & \\ & & & & & & & & 1 \\ & & & & & & & & & 1 \end{bmatrix} \begin{bmatrix} u_{0,0} \\ u_{1,0} \\ u_{2,0} \\ u_{3,0} \\ u_{0,1} \\ u_{1,1} \\ u_{2,1} \\ u_{3,1} \\ u_{0,2} \\ u_{1,2} \\ u_{2,2} \\ u_{3,2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ f_{1,1} \\ f_{2,1} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad (2.9)$$

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 $k(A) = \|A\|_2 \|A^{-1}\|_2 = 43.16$. Dois detalhes importantes devem ser considerados nessa primeira discretização.

O primeiro é que a equação (2.6) 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 multiplicar (2.6) pela área do elemento de malha, $h_x h_y$.

$$2(a + b)u_{i,j} - a(u_{i+1,j} + u_{i-1,j}) - a(u_{i,j+1} + u_{i,j-1}) = h_x h_y f(x_i, y_j) \quad (2.10)$$

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 simétrica A . Por exemplo, em um ponto de malha adjacente ao limite esquerdo do domínio, o caso $i = 1$ de (2.10), 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 \mathbf{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 meio de álgebra linear mais eficientes, como gradientes conjugados, KSP e pré-condicionadores de Cholesky. Ao realizar estas duas alterações no sistema linear (2.11), tem-se:

$$\begin{bmatrix}
 1 & & & & & & & & & \\
 & 1 & & & & & & & & \\
 & & 1 & & & & & & & \\
 & & & 1 & & & & & & \\
 & & & & 1 & & & & & \\
 & & & & & \alpha & \beta & & & \\
 & & & & & \beta & \alpha & & & \\
 & & & & & & & 1 & & \\
 & & & & & & & & 1 & \\
 & & & & & & & & & 1 \\
 & & & & & & & & & & 1
 \end{bmatrix}
 \begin{bmatrix}
 u_{0,0} \\
 u_{1,0} \\
 u_{2,0} \\
 u_{3,0} \\
 u_{0,1} \\
 u_{1,1} \\
 u_{2,1} \\
 u_{3,1} \\
 u_{0,2} \\
 u_{1,2} \\
 u_{2,2} \\
 u_{3,2}
 \end{bmatrix}
 =
 \begin{bmatrix}
 0 \\
 0 \\
 0 \\
 0 \\
 0 \\
 (1/6)f_{1,1} \\
 (1/6)f_{2,1} \\
 0 \\
 0 \\
 0 \\
 0 \\
 0
 \end{bmatrix}
 \quad (2.11)$$

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 $k(A) = 5.83$.

Referências Bibliográficas

- [Balay et al.(2016)Balay, Abhyankar, and Smith] Balay, S., Abhyankar, S., Smith, B. F., 2016. PETSc users manual. Tech. Rep. ANL-95/11 - Revision 3.7, Argonne National Laboratory.
URL <http://www.mcs.anl.gov/petsc>
- [Bueler(2018)] Bueler, E., 2018. PETSc for Partial Differential Equations. SIAM.