Disciplina

Introdução a Métodos Computacionais Aplicados à Física

Prof. Fernando Sato
Departamento de Física – Instituto de Ciências Exatas
Universidade Federal de Juiz de Fora – UFJF
CEP 36036-900

Endereço Eletrônico: fernando.sato@ufjf.br
03/2022

LaTeX

Este curso tem como objetivo proporcionar ao aluno uma primeira aproximação com os métodos computacionais aplicados à Física. O método computacional em si consiste basicamente em transcrever os métodos utilizados na resolução de problemas físicos, com relação ao modelo matemático, em programas escritos em alguma linguagem computacional como a linguagem c, c++, Fortran, entre outras. Neste caso será utilizado a linguagem de programação Fortran dentro das sintaxes da versão do Fortran 90. Uma vez tendo o domínio do Fortran 90 básico será feito uma aplicação para a resolução de problemas, por exemplo, do tipo encontrar raizes de uma equação de segundo grau, derivadas, integrais, etc. O curso tem também como objetivo apresentar um ambiente de trabalho tipo unix, mais especificamente o linux que faz parte do projeto GNU que almeja a difusão dos softwares de código aberto.

Observações: [1] É necessário que o aluno já tenho cursado as disciplinas básicas de Física e Cálculo para ter o máximo de aproveitamento do curso. [2] Este material ainda está sendo construído, portanto poderá ser encontrado alguns erros de digitação, falta de exemplos, entre outros que serão complementados no decorrer do curso ou então que foram ditos em no laboratório e não estão presentes aqui.

Conteúdo

Capítulo 1

Ementa

1. Introdução à linguagem de programação FORTRAN 90;
2. Integração e derivação numérica;
3. Equações diferenciais ordinárias;
4. Noções básicas de Dinâmica Molecular Clássica;
5. Noções básicas do método Monte Carlo Clássico;
6. Complementos.

Capítulo 2

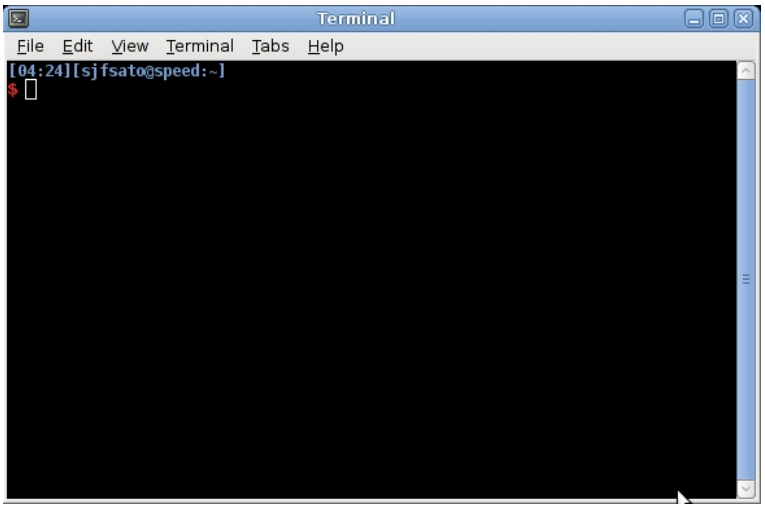
Preliminares

2.1 Conhecendo o Ambiente Linux pelo Shell

2.1.1 Conexão Via SSH

comentário: A infraestrutura para o curso é composta por um laboratório computacional contendo vários computadores com acesso à internet e com o sistema operacional linux. Todas as atividades do laboratório serão realizadas acessando um computador remotamente através de um console (também conhecido como shell) pela linha de comando, não teremos a necessidade de utilizar o ambiente gráfico a não ser quando for necessário visualizar resultados por um gráfico.

No linux temos vários terminais para execução de linhas de comando em função dos diversos ambientes gráficos existentes. Em geral os programas que dão acessos à terminais são: *xterm*, *konsole*, *gnome-terminal* e tem a aparência conforme a figura 2.1.1.



Com o terminal aberto, o comando para se conectar remotamente em uma outra máquina é o *ssh*. A sintaxe do comando é:

ssh usuario@computador.dominio

ou

ssh -l usuario computador.dominio

Em que *ssh* é o comando, *usuario* é o nome do usuário que deve ter uma conta cadastrada na máquina remota e *computador.dominio* é o nome do computador remoto a ser acessado. Após teclado o comando completo e o *enter* será pedido uma senha pessoal inicialmente cadastrada na máquina remota.

comentário: Para trocar a senha da máquina remota basta executar o comando *passwd* e então será solicitado a atual senha, depois a nova senha em seguida a confirmação da nova senha.

comentário: Na figura 2.1.1 o quadrado branco que aparece é chamado de *prompt* de comando onde serão digitados os comandos. O que antecede este quadrado branco é em geral *usuario@computador* o a conta do usuário *usuario* e computador atual. Note que ao abrir o terminal na linha do *prompt* de comando temos um usuario e o nome do computador atual e após conectar no computador remoto teremos um outro usuário e o nome do computador remoto.

2.1.2 Comandos Básicos No Terminal

O terminal ou console é o análogo a um gerenciador de arquivos como o conhecido *Windows Explorer*® do sistema Windows® [1] onde é possível criar diretórios, apagar e mover arquivos, listar diretórios, entro outros possibilidades, no entanto ao invés de utilizar o *mouse* é utilizado a linha de comando.

Os comando mais usuais no terminal é dado na tabela 2.1.2.

Comando	Descrição do Comando
ls	lista o conteúdo do diretório (*ls -l*, *ls -lah*)
cd	troca de diretório (*cd ../../*, *cd /*)
cp	para copiar (sintaxe *cp (opções) (origem) (destino)*)
mkdir	cria diretório (sintaxe *mkdir (nome-do-diretório)*)
rmdir	apaga diretório (sintaxe *rmdir (nome-do-diretório)*)
pwd	mostra o caminho do diretório atual
more	mostra o conteúdo de um arquivo ascii
outros	entre outros comandos veja através do *man* os comandos *cat, rm, mv, clear, date, df, du, ln, find. grep, head, tail, less, sort, time, uname, cat, who, finger, whoami, chmod, chown, nohup, etc*

Um excelente manual pode ser encontrado on-line no próprio *shell* [7], por exemplo, pelo comando *man cp* ou então pelo *Guia Foca GNU/Linux* [6] que possui versões eletrônicas (pdf e html). O *Guia Foca GNU/Linux* [6] é um excelente manual para iniciantes em linux e possui uma parte inicial destinada a periféricos de computadores, organização do sistema de diretórios, entre outros comandos que irão auxiliar na utilização de um *shell*.

Entre a introdução do *shell* e os primeiros passo com a liguagem fortran 90 o tempo é extremamente curto e será necessário um desenvolvimento extra para os alunos que nunca tiveram contato com este tipo de interface computacional. É importante que seja praticado os comandos na *linha de comandos* de um *shell* afim de criar maior intimidade com o novo sistema.

[1] Windows Explorer® e Windows® é marca registrada da empresa *Microsoft Corporation*

2.1.3 Editor de Texto no *shell*

Em breve estaremos escrevendo programas, criando *scripts* e anotando informações e para isso necessitaremos de um editor de texto. Comunmente em um ambiente gráfico utilizamos editores de texto tipo o *openoffice/broffice*. No *shell* temos também bons e poderosos editores de texto como *vim*, *emacs*, *pico*, *joe*, entre outros. Todos eles são ótimos editores e também com determinadas características, no entanto por questões de praticidade utilizaremos o editor *joe*. A utilização do *joe* é simples, para abrir um arquivo basta digitar no *prompt* de comando *joe nome-do-arquivo.txt*. Os comando do editor *joe* estão resumidos na tabela 2.1.

Comando	Descrição do Comando
^kh	Abre menu de ajuda
^kd	Salva e não sai do editor
^kx	Salva e sai do editor
^c	sai do editor
^kf	procura por palavra ou conjunto de letras pelo texto
^ku	vai para o início do arquivo
^kv	vai para o final do arquivo
^kl	vai para a linha desejada
^kb	marca início do bloco
^kk	marca final do bloco
^kc	copia bloco após o *prompt*
^ky	apaga bloco inteiro

Tabela 2.1: Resumo dos comandos do editor de texto *joe*. O comando ^kb significa que deve-se segurar a tecla *ctrl* pressionada e depois teclar as teclas **k** e **b** na seqüência, o semelhante para os outros comando do editor *joe*.

2.1.4 Exportando Aplicativo Gráfico

Uma da habilidades do sistema operacional que estamos utilizando é a possibilidade de utilizar algum aplicativo gráfico da máquina remota como um editor de texto (*gedit* e *kate*), auxiliar para fazer gráficos (*xmgrace* e *gnuplot*), planilha eletrônica (*openoffice-calc* - *oocalc*), navegador de internet (*firefox*), gerenciador de arquivos (*nautilus*). Para que isso seja possível basta adicionarmos a opção *-XA* no comando de conexão *ssh*, como no exemplo abaixo.

ssh -XA usuario@computador.dominio

ou

ssh -XA -l usuario computador.dominio

Depois de conectado na máquina remota é só executar o comando do aplicativo, por exemplo o editor de texto *gedit*. Dependendo da distância e da conexão da internet o aplicativo gráfico pode ficar lento e algumas vezes inviável sendo mais vantajoso a utilização do aplicativo no modo da linha de comandos.

A importância em saber utilizar a linha de comando é que muitas vezes teremos acesso à uma rede de computadores em outros locais onde a presença física é inviável devido à distância e também por ser inviável sentar em cada computador para colocar um computador para fazer as contas, aqui temos a facilidade de poder fazer tudo isso a partir da nossa estação de trabalho. Dentre esses computadores podemos citar por exemplo o Centro Nacional de Computação de Alto Desempenho de Campinas (CENAPAD - Campinas - SP), entro outros centros de computação de alto desempenho existentes no Brasil que nos permite mediante cadastramento de projeto de pesquisa a utilização dos recursos computacionais. Então não temos que ir à cidade de Campinas - SP todas as vezes que desejarmos executar alguns programas da nossa pesquisa científica.

Dentro do que for possível seria muito interessante aprender a utilizar o programa *gnuplot*, dentro do próprio *shell* temos um manual e também no sítio oficial do programa temos disponível o manual completo.

Capítulo 3

Introdução à linguagem de programação FORTRAN 90

3.1 Noções preliminares, conceitos básicos e compilador

Uma das partes importantes do curso inicia-se nesta seção e será uma das nossas bases para a última parte do curso, mais longa, mais trabalhosa, porém mais interessante. Entenda como interessante porque ao chegar na última parte do curso teremos a nossa base completa para aplicação para problemas que antes só vimos na teoria do papel e caneta. O que é referido como base é o conhecimento dos comandos básicos para o *shell*, o conhecimento do Fortran 90[1] [5] e o que já foi visto nas disciplinas dos cálculos e das físicas básicas. O que se pretende depois é unir todo esse conhecimento e construir um programa de computador que seja capaz de auxiliar no entendimento e resolução de problemas matemáticos e físicos.

[1] O Fortran é proveniente das palavras *formula translation*, a história da linguagem Fortran pode ser encontrado em: http://en.wikipedia.org/wiki/Fortran

3.1.1 Compilador

O compilador é o responsável em gerar um programa executável a partir dos comandos que foi colocado no arquivo fonte do programa. A idéia básica consiste em gerarmos um arquivo texto contendo as linhas de comando próprias da linguagem, então utilizamos um compilador para gerar um outro arquivo que pode ser executado no computador. O arquivo executável é um arquivo binário (que não é tipo texto, ascii) com instruções de baixo nível, ou seja, o mais próximo possível da linguagem do computador ou linguagem de máquina. Podemos dizer que o compilador é um programa de computador que faz a tradução entre o arquivo texto com os comandos da linguagem específica (fortran, c, c++) e o arquivo executável, um tradutor.

Outra classe de linguagem de programação consiste em ter seu código interpretado, não necessitando de compilação do código fonte como por exemplo e *bash*, *phyton*, *perl*, entre outros. Aqui no curso não trabalharemos com estes tipos de linguagem interpretadas, limitaremos somente ao Fotran 90. Talvez em um determinado momento faremos uso do *bash* para criarmos scripts afim de facilitarmos o processo de execução de alguns programas, no intuito de automatizar processos repetitivos sem a necessidade de digitar algo por n-vezes consecutivas.

Dentro do GNU/Linux temos um compilador de fortran 90/95 conhecido como *gfortran*. Dentre outros compiladores fotran temos o compilador da Intel (*ifort*) [2], Portland (*pgif90*)[3], NAG Fortran Compiler[4].

Para fazermos uso de um compilador e termos um programa executável, precisamos inicialmente construir um código fonte que contenha as instruções do que desejamos fazer.

comentário: o computador é uma máquina que foi feita para receber ordens e executá-las, no entanto ao perceber que o computador está executando as tarefas de modo estranho ou como você não gostaria, tenha certeza que a ordem foi mal dada. Não adianta ficar nervoso(a), se o computador está dando o resultado errado é porque você está mandando ele fazer a coisa errada, se quiser que faça a coisa certa, mande ele fazer a coisa certa.

O primeiro programa que iremos fazer é fazer um programa que escreva uma frase na tela. Com um editor de texto (comando *joe prog1.f90*) crie um arquivo chamado prog1.f90, com o conteúdo abaixo:

```
1 ! Programa inicial que imprimi na tela
2 ! alguma mensagem desejada.
3 ! DD/MM/AAAA por programador iniciante
4 ! OBS: Tudo que esta a direita do simbolo de exclamacao
5 !       e um comentario e nao faz parte da sintaxe do
```

[2]http://www.intel.com. A Intel fornece uma versão do compilador fortran, c/c++ e bibliotecas matemáticas gratuitamente com restições permitindo somente para uso acadêmico.

[3]http://www.pgroup.com/

[4]http://www.nag.co.uk/nagware/NP.asp

```
6  !       fortran 90.
7
8  ! Inicio do programa
9  program prog1
10
11 ! linha obrigatoria em todos os programas que iremos fazer
12 implicit none
13 !
14         ! comando para imprimir na tela
15         ! tudo que esta entre aspas eh considerado como texto
16         write(*,*) "Boa tarde turma de fiscomp!"
17         !!! Significado do comando write
18         ! write = escreva
19         ! (*,*)
20         !  |----> onde (tela, arquivo, usb, impressora, etc)
21         !    |-->como em relacao seu formato (formato livre)
22         !
23         write(*,*) "Hoje e dia 30/02/20XX. Boa tarde!"
24
25 ! Final do programa
26 stop
27 end program prog1
```

Com o arquivo do programa salvo, compile com o comando:

gfortran -o prog1.x prog1.f90

Estamos dizendo ao compilador para não gerar o objeto[5] e pedindo para ele gerar um executável *prog1.x* a partir do arquivo *prog1.f90*. Execute o programa gerado da forma:

./prog1.x

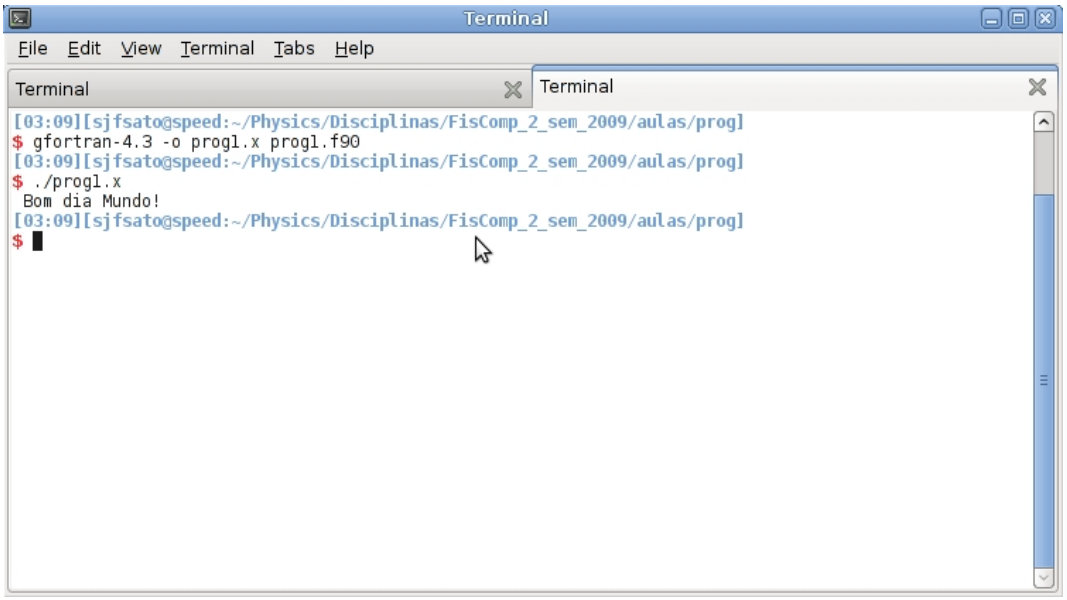
o resultado deve ser como o mostrado na figura 3.1.1.

3.2 Declarações e tipos de variáveis e constantes

Depois dessa primeira experiência temos que conhecer como fica a declaração de constantes e variáveis e qual o formato quanto a ser real, inteiro, complexo, lógico e carácter. Assim temos cinco formatos entre variáveis e constantes que podem ser tratadas dentro do Fortran 90.

- ***integer*** - é um número que não contém casas decimais e podem assumir valores positivos, negativos ou zero. Exemplos 0, -934, 1982649, +13, 1000000. Os

[5]Um passo na tradução entre o código fonte do programa e o executável é a geração do objeto, adiante falaremos sobre o *objeto*.



limites dentro de um computador tipo PC, que é o que estamos utilizando varia entre -2,147,483,648 e 2,147,483,647 que é um número de 32 bits;

- ***real*** - é um número com casas decimais e podem assumir valores positivos, negativos ou zero e pode conter expoentes. Exemplos 8., -934.1, 2.5E3, 0.05E+1, 4.78E-5, 4.78D-5, 6.04D24;
- ***complex*** - número que possui uma parte real e outra imaginária. A declaração é feita por exemplo: "complex :: var1", em que "var1=(parte-real,parte-complexa)";
- ***character*** - uma variável ou constante do tipo caracter pode ser escrita como uma string de uma ou várias letras no limite de 32767 caracteres e sempre de vir entre entre ' ou ";
- ***logical*** - é uma variável do tipo ".TRUE."ou ".FALSE.".

Conjunto de caracteres do Fortran 90/95

- (26) Letras maiúsculas do alfabeto: **A-Z**;
- (26) Letras minúsculas do alfabeto: **a-z**;
- (10) Dígitos: **0 até 9**;
- (01) Caracter *underscore*: _
- (05) Símbolos aritiméticos: **+ - * / ****;
- (17) Outros símbolos: **( ) . = , ' $: ! "% $; < > ? espaço**

Abra o editor de texto e digite o programa abaixo, que chamará *prog2.f90*.

```
1  ! DD/MM/AAAA Programador
2  ! O simbolo de exclamacao significa um comentario, apos o simbolo
3  ! Programa exemplificando alguns os tipos de variaveis.
4  ! kind=4 - simples preciscao - 7 casas decimais (1.175494E-38 3.402823E
       +38)
5  ! kind=8 - dupla precisao - 15 casas decimais (2.225074D-308  1.797693D
       +308)
6  ! kind=16 - quadrupla precisa - 31 casas decimais (2.225074Q-308
       1.797693Q+308)
7  !
8  program variaveis
9  implicit none
10 !
11 ! REAL COM SIMPLES PRECISAO
12 real(kind=4) :: var1
13 ! REAL COM DUPLA PRECISAO
14 real(kind=8) :: var2
15 ! INTEIRO
16 integer :: var3
17 ! CARACTER
18 character(5) :: var4
19 !
20 var1 = 3.14159265358979323846264338327950288419E0
21 var2 = 3.14159265358979323846264338327950288419D0
22 var3 = 5829
23 var4 = "Computacional"
24 !
25 ! ESCREVENDO NA TELA SEM FORMATO
26 write(*,*) var1, var2, var3, var4
27 !
28 stop
29 end program variaveis
```

Uma vez digitado o programa no editor de texto, o programa acima mostra os tipos de declarações mais usuais e como proceder no caso de números reais. Preste atenção que ocorreu algum problema com a exposição da constante/variável character *var4*, foi impresso na tela somente *Compu* e não *Computacional*, isso se deve ao fato da declaração ter sido feita somente para 5 caracteres; altere *character(5)* para *character(30)*, compile e execute o programa novamente.

Vale lembrar que em Fortran os caracteres são *case insensitive*, isto significa que na declaração de constantes/variáveis ou no meio do programa:

$$\text{Var1} = \text{VAR1} = \text{VAr1} = \text{vaR1}$$

então não se preocupe caso no meio do programa tenha utilizado maiúscula ou minúscula escrever uma variável ou constante. Na linguagem c/c++ o exemplo acima

Comando	Descrição do Comando
+	Adição ($a = b + c$)
−	Subtração ($a = b - c$)
*	Multiplicação ($a = b * c$)
/	Divisão ($a = b/c$)
**	Exponenciação ($a = b * *c$)

Tabela 3.1: Operações matemáticas no fortran.

são todos diferentes um dos outros pelo fato de ser *case sensitive*, no linux em geral nomes de arquivos e diretórios são *case sensitive*, ou seja, nomes escritos em maiúsculas são diferentes dos nomes escritos em minúsculas assim como na linguagem c/c++ e diferentemente da linguagem Fortran.

3.3 Expressões

Como o nome da linguagem sugere iremos trabalhar com expressões matemáticas. O primeiro passo para construir uma expressão em Fortran é que a atribuição de uma expressão à uma variável, que sempre fazemos da forma:

nome-da-variável = expressão-matemática

e não o contrário tipo ***expressão-matemática = nome-da-variável***. A expressão é calculada do lado direito da igualdade de atribuída à variável do lado esquerdo.

Dentro das expressões matemáticas podemos realizar as operações:

As operações matemáticas serão utilizadas combinadas em uma expressão. Para isso é necessário estabelecer uma sequência onde qual operação aritmética é realizada primeiro. Em uma expressão o que é realizado primeiro a multiplicação ou a soma? No Fortran a sequência é:

1. O conteúdo dos parênteses são realizados primeiro, partindo do mais interno para o mais externo;
2. Todas as exponenciais são calculadas partindo da direita para a esquerda;
3. Todas as multiplicações e divisões são calculadas partindo da esquerda para a direita;
4. Todas as adições e subtrações são calculadas partindo da esquerda para a direita.

Para testarmos crie o programa *prog3.f90* como dado abaixo:

```
!
program prog3
implicit none
!
real(kind=8) :: distancia1,distancia2,distancia3
real(kind=8) :: aceleracao,tempo
!
aceleracao = 3.0D0
tempo = 12.0D0
!
distancia1 = 0.5 * aceleracao * tempo ** 2.0D0
distancia2 = (0.5 * aceleracao * tempo) ** 2.0D0
distancia3 = 0.5 * aceleracao * (tempo ** 2.0D0)
!
write(*,*) distancia1,distancia2,distancia3
!
stop
end program prog3
```

compile (*gfortran -o prog3.x prog3.f90*) e rode (*./prog3.x*) e veja como qual operação é realizada primeiro.

Para reforçar esta parte, considere os números reais a = 3.0, b = 2.0, c = 5.0, d = 4.0, e = 10.0, f = 2.0 e g = 3.0, faça um programa que calcule a expressões abaixo[6]:

res1 = a*b+c*d+e/f**g

res2 = a*(b+c)*d+(e/f)**g

res3 = a*(b+c)*(d+e)/f**g

Arredondamento na Divisão de Inteiros e Reais

Para desenvolver esta parte, nada melhor do que fazer um programa e entender como o Fortran exprime os resultados de uma divisão.

Declare somente variáveis inteiras e realize as seguintes divisões: 3/4 =?, 4/4 =?, 5/4 =?, 6/4 =?, 7/4 =?, 8/4 =?, 9/4 =?. Para a divisão com reais não teremos problemas e na mistura entre declarações de reais e inteiros, podemos complementar o programa para tirar essas dúvidas, como o programa *prog4,f90.*

```
program prog4
implicit none
!
real(kind=8) :: r1, r2, r3, r4, r5
real(kind=8) :: r_r1, r_r2
integer :: i1, i2, i3, i4, i5, i6, i7
integer :: i_r1, i_r2, i_r3, i_r4, i_r5
```

[6]Os resultados devem ser *res*1 = 27.25, *res*2 = 209.00 e *res*3 = 36.75

```
integer :: i_r6, i_r7, i_r8, i_r9, i_r10
!
i1=3; i2=4; i3=5; i4=6; i5=7; i6=8; i7=9
!
r1=3.0d0; r2=4.0d0; r3=5.0d0; r4=6.0d0; r5=7.0d0
!
i_r1=i1/i2; i_r2=i2/i2; i_r3=i3/i2
i_r4=i4/i2; i_r5=i5/i2; i_r6=i6/i2
i_r7=i7/i2; i_r8=i6/r1; i_r9=i7/r1
i_r10=i1/r1
!
r_r1 = i5 / r3
r_r2 = r1 / i6
!
write(*,*) ':-) Resultado - Inteiro (-:'
write(*,*) '3/4=',i_r1,'4/4=',i_r2,'5/4=',i_r3
write(*,*) '6/4=',i_r4,'7/4=',i_r5,'8/4=',i_r6
write(*,*) '9/4',i_r7
write(*,*) '8/3.0=',i_r8,'9/3.0=',i_r9,'3/3.0=',i_r10
write(*,*) ':-) Resultado - Real (-:'
write(*,*) '7/5.0=',r_r1,'3.0/8=',r_r2
!
stop
end program prog4
```

O resultado deve ser como abaixo:

```
[16:10][sato@df2]~/prog]
$./prog4.x
 :-) Resultado - Inteiro (-:
 3/4=           0 4/4=           1 5/4=           1
 6/4=           1 7/4=           1 8/4=           2
 9/4           2
 8/3.0=           2 9/3.0=           3 3/3.0=           1
 :-) Resultado - Real (-:
 7/5.0=    1.3999999999999999      3.0/8=   0.37500000000000000
[16:10][sato@df2]~/prog]
$
```

Observações

- Por que $3/4 = 0$?

3.3.1 Funções Intrínsecas

Dentro do Fortran 90 podemos simplesmente calcular um *seno* utilizando séries ou utilizar a função intrínseca já existente no Fortan 90. Na tabela 3.2 apresentamos as funções intrínsecas mais usuais.

Nome e Argumento	Valor da Função	Tipo de Argumento	Tipo de Resultado	Comentário
SQRT(X)	$\sqrt{x}$	R	R	Raiz quadrada de x para $x > 0$
ABS(X)	$\|x\|$	R/I	*	Valor absoluto de x
ACHAR(I)		I	CHAR(1)	Retorna o caracter referente a posição 1 na tabela ascii
SIN(X)	$sin(x)$	R	R	Seno de x (x deve estar em radianos)
COS(X)	$cos(x)$	R	R	Cosseno de x (x deve estar em radianos)
TAN(X)	$tan(x)$	R	R	Tangente de x (x deve estar em radianos)
EXP(X)	e^x	R	R	No. neperiano elevado a potência x
LOG(X)	$log_e(x)$	R	R	Logarítmo natural de x, para x>0
LOG10(X)	$log_{10}(x)$	R	R	Logarítmo na base 10 de x, para x>0
IACHAR(C)		CHAR(1)	I	Retorna a posição do carácter C referente a tabela ascii
INT(X)		R	I	Parte inteira de x (x é truncado)
NINT(X)		R	I	Parte inteira de x (x é arredondado)
REAL(X)		I	R	Converte um número inteiro em real
MOD(A,B)		R/I	*	Resto de um divisão de A por B (A/B)
MAX(A,B)		R/I	*	Pega o maior entre A e B
MIN(A,B)		R/I	*	Pega o menor entre A e B
ASIN(X)	$sin^{-1}(x)$	R	R	Arco cujo Seno seja x (resultado em radianos)
ACOS(X)	$cos^{-1}(x)$	R	R	Arco cujo Cosseno seja x (resultado em radianos)
ATAN(X)	$tan^{-1}(x)$	R	R	Arco cuja Tangente seja x (resultado em radianos)

Tabela 3.2: Funções intrínsecas usuais no Fortran 90/95.

Atividade (*prog4a.f90*) para exemplificar algumas funções intrínsecas.

```
1 ! CRIADO EM DD/MM/AAAA - NOME
2 ! PROGRAMA PARA EXEMPLIFICAR UMA FUNCAO INTRINSECA
3 ! --> ANGULO, TRANSFORMACAO DE GRAUS PARA RADIANOS
4 ! --> CONVERSAO DE REAL PARA INTEIRO TRUNCADO OU ARREDONDADO
5 ! --> CONVERSAO DE NUMERO PARA CARACTER DA TABELA ASCii
6 ! --> CONVERSAO DE CARACTER PARA NUMERO DA TABELA ASCii
7 !
```

```
8  program prog4a ! INICIO DO PROGRAMA
9  implicit none  ! SINTAXE OBRIGATORIA NO CURSO
10
11 !!! DECALRACAO DE VARIAVEIS INTEIRAS
12 integer :: h33,h64,h65,h76,h79,i1,i2
13 !!! DECLARACAO DE VARIAVEIS REAIS
14 real(4) :: a,conv_rad,ang_graus,r1
15 real(8), parameter :: pi=3.1415926535897932385
16 !!! DECLARACAO DE VARIAVEIS CARACTER
17 character(1) :: c1,c26
18
19 !!============================ ATRIBUICAO DE VALORES
20 !!! VARIAVEIS CARACTERES
21 c1='A'
22 c26='Z'
23
24 !!! VARIAVEIS INTEIRAS
25 h33=33
26 h64=64
27 h65=65
28 h76=76
29 h79=79
30
31 !!! VARIAVEIS REAIS
32 !a=3.5555
33 a=3.4
34 conv_rad=pi/180.0
35 ang_graus=60.0
36
37 !!! FUNCOES INTRINSECAS
38 r1=sin(ang_graus*conv_rad)
39 i1=INT(a)  !TRUNCADO
40 i2=NINT(a)  !ARREDONDADO
41
42 !!============================ ESCREVENDO NA TELA
43 !!! IMPRIMINDO VARIAVEIS INTEIRAS
44 write(*,*) i1,i2,r1
45 !!!=== IMPRIMINDO OS CARACTERES
46 write(*,*) CHAR(h64),CHAR(h79),CHAR(h76),CHAR(h65),CHAR(h33)
47 !!!=== IMPRIMINDO OS NUMEROS
48 write(*,*) IACHAR(c1),IACHAR(c26)
49
50 !!! FINAL DO PROGRAMA
51 stop
52 end program prog4a ! FECHANDO O PROGRAMA
```

3.4 Programação estruturada

Uma das operações mais notáveis em um computador é a capacidade de tomar decisões previamente estabelecidas e de fato um computador pode realizar muito rápido um conjunto de decisões. Dependendo do nível de programação um programa pode tomar decisões tão complexas quanto a de um ser humano, no entanto essas decisões complexas de um programa de computador são compostas por decisões elementares que por sua vez, em geral, são compostas por comparação entre duas ou três variáveis. Dependendo do resultado da comparação é realizado outras sequências de decisões. Em geral as decisões no Fortran são tomadas pelo comando *if*, que por sua vez estão contidos ou não dentro de outros comandos *do*.

Em conjunto com os camandos *if* e *do*, temos outros comando para controlar a execução que são os comandos *continue*, *pause*, *stop*, *end*, *call* e *return*. O comando *call* é utilizado para transferir o controle de execução para uma subrotina e o comando *return* é utilizado para retornar para o programa principal uma vez estando dentro de uma subrotina[7].

A programação estruturada é um método para combinar as estruturas de controle formadas por blocos de *do* e *if* de maneira organizada e que, principalmente, traduza em linguagem universal de programação o problema a ser resolvido. A estruturação do programa, ou seja, a tradução entre as equações que estão no papel (livros) e o programa propriamente dito deve tornar-se por hábito uma parte muito importante na logística da construção. Um dos fatores de boa logísitca para a construção de um programa é levar em conta que o código computacional seja de execução rápida e também preciso, por outro lado, tempos extras de execução e processamento torna-se em disperdício de recursos.

Essas estruturas de controle são formadas por quatro estruturas básicas de controle:

1. Estrutura Sequencial - vários blocos de comandos podem ser colocados para serem executados seqüencialmente;

2. Estrutura *if-then-else* - um bloco pode ser construído unindo-se dois blocos mediante ha uma condição;

3. Estrutura de laço *do while* - um bloco pode ser construído colocando-se um bloco interno de laço do tipo *do while*;

4. Estrutura de repetição, *repetindo sob condição* - um bloco pode ser construído colocando-se um bloco interno de laço de repetição do tipo repita a operação mediante uma condição.

[7]Uma subrotina é uma parte do programa que é executado na sequência mas fora do programa principal.

A maior dificuldade para programadores iniciantes e já com algum conhecimento é o de expressar de modo organizado a idéia e a lógica estrutural do programa. De modo prático, seria a capacidade que o programado possui para explicar o que um determinado programa executa e como ele executa. Para possibilitar a expressão do funcionamento do programa de forma fácil e organizada utilizaremos o conceito de programação estruturada que é a técnica de construir programas de forma organizada utilizando-se da combinação das formas básicas das estruturas de controle.

Dentro do esquema da programação estruturada, podemos expressar um programa como uma sequência de blocos constituídos pelas estruturas. Uma vez estabelecido esta sequência de blocos veremos que a estrutura permite também estabelecer um **fluxo de sequências lógicas afim de resolver um determinado problema**. Esta idéia de **fluxo de sequências lógicas** baseado nas estruturas básicas de controle é o que conhecemos como um *fluxograma*.

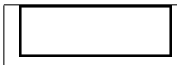
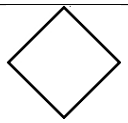
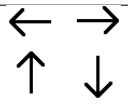
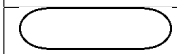
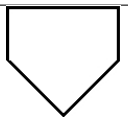
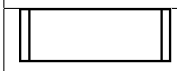
	Processamento: instrução ou conjunto de instruções que indicam processamento de dados.
	Decisão: condição que pode desviar o fluxo do programa para outros pontos do programa.
	Direção de Fluxo: indica a direção do fluxo.
	Terminal: utilizado no início e no final do diagrama de fluxo.
	Entrada/Saída: Entrada ou saída genérica de dados ou resultados.
	Relatório: utilizado principalmente para saída de resultados.
	Conector: conecta uma entrada ou saída para outra parte do programa.
	Conector entre páginas: conecta páginas do diagrama de fluxos.
	Sub-programa externo: referência à um sub-programa externo ao diagrama de fluxo.

Tabela 3.3: Símbolos para fluxograma.

Observe que um fluxograma apresenta algumas características:

- Possui apenas um início (saída de fluxo) e um fim (entrada de fluxo);

- O único símbolo/ação que deriva duas saídas de fluxos (não simultãneos) é o losango;
- Os demais símbolos possuem apenas *uma saída* e uma (ou mais) entrada de fluxo;
- Um fluxo nunca pode terminar em um símbolo/ação que não seja no *fim*;
- Um fluxograma pode ser construído para qualquer tipo de ação, como por exemplo a execução de um dia inteiro da sua rotina.

O exemplo do fluxograma abaixo é o exemplo da estrutura do comando *if* da seção 3.5.2, e ele verifica quais números entre -30 e 30 são divisíveis por 3 (programa 3.5.2). O fluxograma referente ao programa pode ser construído utilizando-se dos símbolos que traduz as estruturas de controle conforme a figura 3.1.

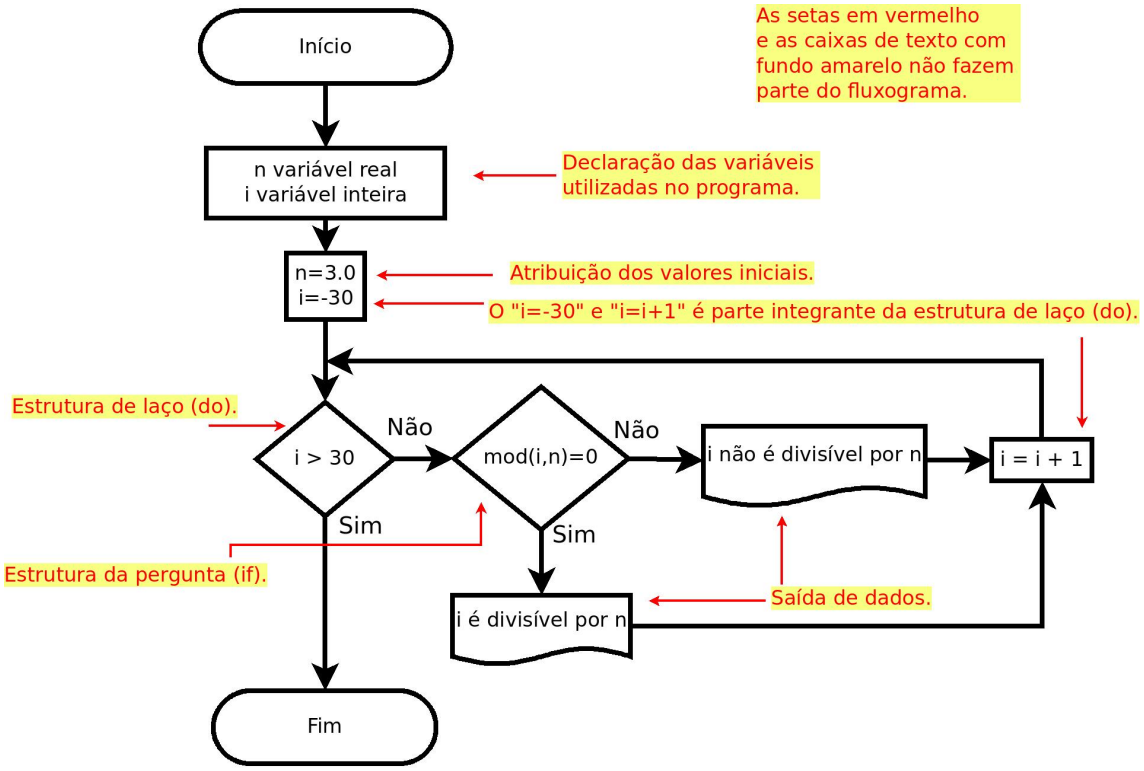


Figura 3.1: Fluxograma referente ao programa que verifica quais números entre −30 e 30 são divisíveis por 3.

A seguir, exemplifico as formas de fluxograma para o *do explícito*, *do infinito* e *do while*.

Um outra forma de construir um código em linguagem universal é aplicando o conceito de psedocódigo. O pseudocódigo é muito semelhante ao código fonte, com todas as indentações, porém as sintaxes são substituídas por palavras de ação, marcações de início, fim e questionamento.

3.5 Comandos de Laço (do, do while) e de condição (if)

Os comandos de laço ou *looping* como o *do/dowhile* e o de condição *if* são um dos maiores propósitos pelo qual vale a pena aprender alguma linguagem de programação, quase todas as linguagens de programa, ou todas elas, possuem esses comandos. Com esses comandos é possível automatizar uma série de processos como, por exemplo, um somatório, avaliar uma expressão em um intervalo de valores, permitir somente a utilização acima/abaixo de um certo valor, etc. É possível utilizar também esses comando um dentro do outro quantas vezes for necessário, por exemplo, dentro de um comando *do* pode ser utilizado um outo comando *do* e um comando *if* e dentro desse comando *if* poderemos ter outro comando *do*, etc.

3.5.1 Comando do explícito

```
1 do variavel=valor-inicial, valor-final, passo
2     (o que se deseja calcular)
3 end do
```

Por exemplo o cálculo de 4 fatorial,

$$4 = 4 \times 3 \times 2 \times 1 = 24 \tag{3.5.1}$$

Escreva um programa que calcule o fatorial de um número. O exemplo acima é simples, mas e se tivéssemos que calcular 1000.0!. Escreva o programa *prog5.f90*

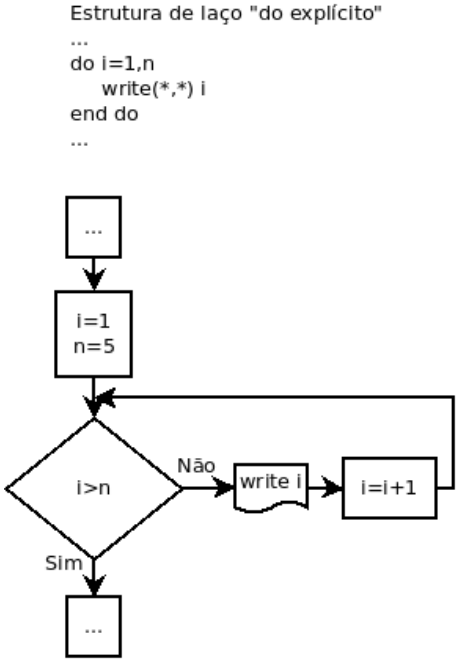
```
1  program prog5
2  implicit none
3  !
4  integer :: i,n,fatorial
5  !
6  fatorial = 1
7  n = 4
8  !
9  do i = 1, n
10     fatorial = fatorial * i
11 end do
12 !
13 write(*,*) 'Fatorial de',n,'=',fatorial
14 !
15 stop
16 end program prog5
```

o programa acima calcula o fatorial da variável n, então esse valor pode ser alterado quantas vez for necessário, lembrando que para cada valor o programa tem que ser compilado novamente.

Para exemplificar um fluxograma de uma estrutura de laço para o *do explícito* veresmo um simples programa que imprime o valor fa variável i na tela:

```
1  program doexplicito
2  implicit none
3  !
4  integer :: i, n
5  !
6  n=5
7  !
8  do i=1,n
9        write(*,*) i
10 end do
11 !
12 stop
13 end program doexplicito
```

Figura 3.2: Exemplo do fluxograma para o laço do *do explícito*.



Observe que não é necessário atribuir valores iniciais para a variável inteira i, na sintaxe do laço já é atribuido o valor para i.

3.5.2 Comando if

Para utilizar o comando *if* será necessário conhecermos os operadores relacionais, para isso vejamos a tabela 3.4:

A idéia do comando *if* é verificar se uma dada condição é verdadeira ou falsa, por exemplo, se para uma determinada conta a utilização fosse somente para números

Operação	Significado
==	Igual a (são dois símbolos de igualdade)
/ =	Diferente de
>	Maior que
>=	Maior ou igual que
<	Menor que
<=	Menor ou igual que

Tabela 3.4: Operador Relacionais utilizados entre duas variáveis/constantes.

menores que zero de uma lista de vários números, assim poderia ser utilizado o comando *if* para filtrar os números menores que zero.

Vamos fazer um programa que verifique quais números entre -30.0 e 30.0 seja divisível por 3.0 e imprima na tela. Neste programa utilizaremos a função intrínseca *MOD(a,b)*, *REAL(a)* também números reais.

```
1  program prog6
2  implicit none
3  !
4  real(kind=8) :: n
5  integer :: i
6  !
7  n = 3.0d0
8  !
9  do i = -30, 30
10       if ( MOD(REAL(i),n) == 0.0d0 ) then
11            write(*,*) i,'    e divisivel por',n
12       else
13            write(*,*) i,'nao e divisivel por',n
14            continue
15       end if
16 end do
17 !
18 stop
19 end program prog6
```

O camando *if* utilizado acima é o *if lógico*, que tem as estruturas e suas variáveis como:

```
1  ...
2  if (a>=b) then
3       write (*,*) 'a maior ou igual a b'
4  end if
5  ...
```

ou

```
1  if (a>=b) then
```

```
2       write(*,*) 'a maior ou igual a b'
3  else
4       write(*,*) 'a menor que b'
5  end if
```

3.5.3 Comando do while

O comando *do while* é um comando de laço (*looping*) que lembra a mistura do comando *do* e *if*. O comando *do while* é muito útil e a função dele é fazer um laço até que uma condição seja satisfeita, ou seja, *faça algo até onde eu mandei*. A sintaxe é *do while (condição)*. Como a idéia deste comando de laço envolve uma condição, é interessante que esta parte seja apresentada após o comando if que é a ideia principal do comando condicional.

Para exemplificar o comando *do while* escreveremos o programa abaixo.

```
1  !
2  ! Exemplo do comando "do while"
3  !
4  program prog6a
5  implicit none
6  !
7  integer :: i,maximo,somatorio
8
9  i=10
10 maximo=1000
11 somatorio=-1000
12 !
13 do while (somatorio < maximo)
14       somatorio = somatorio + i
15       !
16       if(somatorio == maximo)then
17            write(*,*)'somatorio =',maximo
18       end if
19 end do
20 !
21 end program prog6a
22 !
```

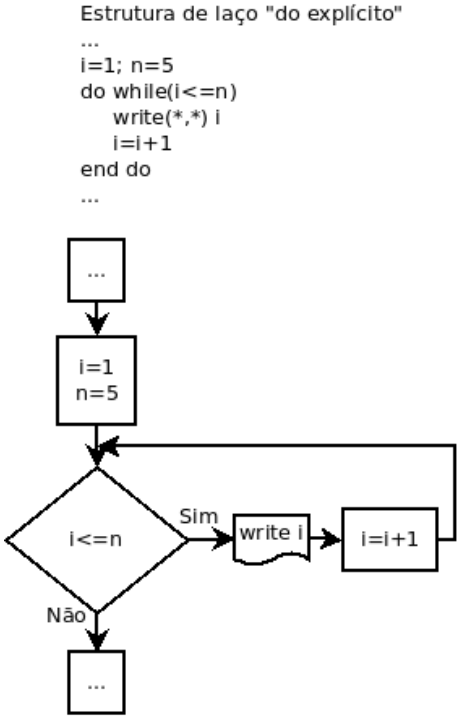
A estrututra do laço condicional *dowhile* apresenta de uma forma ligeiramente diferente em comparação com o *do explícito*. Observerve o programa e em seguida o fluxograma.

```
1  program dw
2  implicit none
3  !
4  integer:: i, n
5  !
```

```
6  i=1; n=5
7  !
8  do while (i<=n)
9        write(*,*) i
10       i=i+1
11 end do
12 !
13 stop
14 end program dw
```

Observe que no caso do *do while* é necessário atribuir valores prévios para i e n e dentro do laço o contador i possui função dupla que é ser impresso na tela e também como condição de parada. Nesta estrutura somente existe a condição de parada, sendo que a condição inicial deve ser previamente estabelecida antes do laço.

Figura 3.3: Exemplo do fluxograma para o laço *do while*.



3.5.4 Comando do infinito

Ideia da construção do bloco do *do* infinito. Como é necessário o uso do comando if, é interessante que esta parte seja apresentada após o comando if.

```
1  !contador inteiro
2  n=0
3  do
4        n = n + 1
5        !(o que se deseja calcular)
6        !(condicao de parada e saida do "do implicito")
```

```
7       if ("condicao satisfeita") exit
8   end do
```

Vamos contruir novamente o programa para o cálculo de 4 fatorial,

$$4 = 4 \times 3 \times 2 \times 1 = 24 \tag{3.5.2}$$

Escreva um programa que calcule o fatorial de um número, agora com o *do* infinito. O exemplo acima é simples, mas e se tivéssemos que calcular 1000.0!, você pode fazer o programa utilizando o *do* infinito. Escreva o programa *prog6b.f90*

```
1   program prog6b
2   implicit none
3   !
4   integer :: i,n,fatorial
5   !
6   fatorial = 1
7   n = 4
8   i=0
9   !
10  do
11        i = i + 1
12        fatorial = fatorial * i
13        if ( i == n ) exit
14  end do
15  !
16  write(*,*) 'Fatorial de',n,'=',fatorial
17  !
18  stop
19  end program prog6b
```

o programa acima calcula o fatorial da variável n, então esse valor pode ser alterado quantas vez for necessário, lembrando que para cada valor o programa tem que ser compilado novamente.

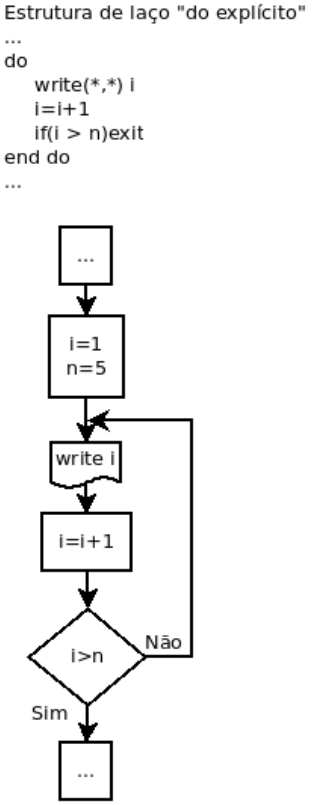
A estrutura do *do infinito* não impõe nenhum condição inicial e/ou final como no caso do *do explícito* (que impõe condição inicial e final) e o *do while* (que impõe somente um condição qeu pode ser atingida a qualquer momento dependendo da estrutura do programa).

```
1   program dw
2   implicit none
3   !
4   integer:: i, n
5   !
6   i=1; n=5
7   !
8   do
9         write(*,*) i
```

```
10        i=i+1
11        if(i > n)exit
12  end do
13  !
14  stop
15  end program dw
```

O fluxograma pode ser construído de mais de uma forma para exemplificar algo mais próximo do *do infinito*, uma forma é a apresentada na figura 3.4

Figura 3.4: Exemplo do fluxograma para o laço *do while*.



3.5.5 Atividade: Comando READ

Comando "**read**´´

Sintaxe: $read(*,*)delta$

é um comando de leitura que atribui um valor, no caso acima, à variável *delta*. A atribuição pode ser um número inteiro, real ou caracter. A variável *delta* deve estar previamente declarada no cabeçalho do programa. No caso do comando *read*, a posição do primeiro asterisco indica de onde vem a atribuição, que poderia ser via teclado ou então via arquivo de dados. O asterisco significa atribuição à variável via teclado. A posição do segundo asterisco significa qual o formato, o asterisco nesta posição indica o formato livre. O significado do formato será visto mais adiante.

Por exemplo:

```
1 write(*,*) 'Digite um valor para o delta'
2 read(*,*) delta
3 write(*,*) 'O valor de delta digitado e',delta
```

suponha que as três linhas acima seja uma parte do seu programa, na execução aparecerá na tela do shell da seguinte maneira:

```
1  Digite um valor para o delta
2 0.5 (digitar um valor teclar enter)
3  O valor de delta digitado e  0.500000000
```

Obs: neste exemplo a variável *delta* foi declarada como variável real de simples precisão.

Construa os programas a seguir:

Programa 1

Dada a equação $y(x) = x^2 - 5x + 6$, faça um programa que leia um valor de x via teclado e imprima o valor de $y(x)$ na tela.

Programa 2

Dada a equação $y(x) = x^2 - 5x + 6$, faça um programa que leia um valor inicial de x_i e um valor final de x_f via teclado. Com o comando de repetição/laço **do** calcule e imprima na tela o valor de $y(x)$ para valores de x iniciando em x_i e terminando em x_f, com o incremento de $\Delta x = 0.2$. Por exemplo $x_i = -1.0$ e $x_f = 4.0$, o primeiro valor será $x = -1.0$, segundo valor será $x = -0.8$, o terceiro valor será $x = -0.6$, ..., o ante-penúltimo valor será $x = 3.6$, o penúltimo valor será $x = 3.8$ e o último valor será $x = 4.0$.

3.5.6 Atividade - O vôo de uma bola

Se assumirmos atrito do ar desprezível e ignorar a curva da Terra, uma bola que é lançada no ar a partir de qualquer ponto na superfície da Terra irá acompanhar uma trajetória parabólica. A altura da bola em qualquer tempo t após ser lançada é dada pela equação abaixo,

$$y(t) = y_0 + v_{0y}t + \frac{1}{2}gt^2$$

em que y_0 é a altura inicial do objeto acima do solo, v_{0y} é a velocidade vertical inicial do objeto e g é a aceleração devido à gravidade da Terra (utilizar $g = 9,8m/s^2$), prestar a atenção no sinal em função do referencial adotado. A distância horizontal (alcance) percorrida pela bola em função do tempo depois de lançada é dada pela equação abaixo

$$x(t) = x_0 + v_{0x}t$$

em que x_0 é a posição horizontal inicial da bola no chão e v_{0x} é a velocidade horizontal inicial da bola.

Se a bola é jogada com alguma velocidade inicial v_0 no ângulo de θ graus em relação à superfície da Terra, então os componentes iniciais horizontal e vertical da velocidade serão:
$v_{x0} = v\cos\theta$ e $v_{y0} = vsen\theta$

Suponha que a bola é lançada a partir da posição inicial $(x_0, y_0) = (0, 0)$ com uma velocidade inicial $|v_0| = 20m/s$ em um ângulo inicial de θ graus. Elabore, escreva e teste um programa que irá determinar a distância horizontal que a bola atingirá a partir do momento em que foi lançada até tocar no chão novamente. O programa deve fazer este cálculo para todos os ângulos θ de 0° a 90° em intervalos de 1° e escreva na tela o ângulo e o alcance atingido pela bola. Após este cálculo é possível fazer um gráfico do ângulo *vs* alcance, e assim determinar o ângulo θ que maximiza o alcance da bola.

Atenção: NÃO utilizar a fórmula pronta para calcular o alcance máximo, o seu programa deve calcular a trajetória parabólica para cada ângulo de lançamento e avaliar quando o objeto atinge o alcance máximo. Se um laço externo para a variação do ângulo e um laço interno para encontrar o alcance máximo a partir das equações dada acima. O código fonte deve conter comentários de tal forma que uma pessoa que não elaborou o código entenda o que foi feito, elaborar também o fluxograma.

Direcionamento de dados

Observação: o direcionamento da saída de dados na tela para um arquivo, de qualquer programa que esteja rodando no terminal, pode ser feito colocando o sinal

"maior que" e o nome de um arquivo no qual será armazenado a informação após o sinal: por exemplo:

```
1 ./prog.x > dados.dat
```

em que *prog.x* é o nome do executável do programa, > é o sinal de direcionamento da saída de dados para o arquivo **ascii** de nome **dados.dat**.

Um outra alternativa de direcionamento de dados pode ser utilizada quando o programa imprime mensagens na tela e/ou inclusão de dados via teclado. Poderá ser utilizado o comando:

```
1 ./prog.x | tee dados.dat
```

assim tudo o que o programa imprimir na tela poderá ser armazenado dentro do arquivo **dados.dat**.

ASCII (do inglês American Standard Code for Information Interchange; Código Padrão Americano para o Intercâmbio de Informação) é um código binário (cadeias de bits: 0s e 1s) que codifica um conjunto de 128 sinais: 95 sinais gráficos (letras do alfabeto latino, sinais de pontuação e sinais matemáticos) e 33 sinais de controle, utilizando portanto apenas 7 bits para representar todos os seus símbolos. Fonte: https://pt.wikipedia.org/wiki/ASCII.

3.6 Vetores, Matrizes e Alocação Dinâmica de Memória

3.6.1 Vetores

Nesta parte veremos que podemos associar valores à uma variável indexada que chamamos de vetores ou como é conhecido internacionalmente de *arrays*. Um *array* pode assumir valores valores inteiros, reais, caracter, etc, como aqueles que estão na seção 3.2.

Um *array* unidimensional (1D) pode ser escrito como:

$$a_i, \text{ sendo } i = 1, n \longrightarrow a_1, a_2, a_3, \cdots, a_n$$

em que cada variável indexada a_i pode assumir um valor.

Em Fortran o *array* pode ser utilizado, por exemplo, para se fazer uma conta de centro de massa de uma distribuição de partículas. Supomos que temos 5 partículas e cada partícula possui as coordenadas e massas abaixo:

Partícula	x (m)	y (m)	z (m)	massa (kg)
1	4.37	1.23	9.55	0.125
2	9.27	-1.23	0.51	0.333
3	3.35	8.23	6.53	0.975
4	7.32	-8.73	4.59	0.229
5	1.22	15.12	5.88	0.434

Tabela 3.5: Coordenadas e massas de 5 partículas.

Como já aprendemos em Física básica a posição do centro de massa para coordenada pode ser dada de acordo com a equação 3.6.1.

$$x_{cm} = \frac{\sum_{i-1}^{n} (x_i * m_i)}{\sum_{i-1}^{n} m_i} \tag{3.6.1}$$

para as outras coordenadas segue a mesma equação trocando apenas x por y ou z.

Em Fortran podemos calcular o centro de massa desse sistema utilizando um conjunto de *arrays* de uma dimensão. Para a coordenada x atribuímos como na tabela 3.6.

O programa Fortran para calcular o centro de massa é dado como se segue abaixo:

Partícula	x (m)	y (m)	z (m)	massa (kg)
1	x(1) = 4.37	y(1) = 1.23	z(1) = 9.55	m(1) = 0.125
2	x(2) = 9.27	y(2) = -1.23	z(2) = 0.51	m(2) = 0.333
3	x(3) = 3.35	y(3) = 8.23	z(3) = 6.53	m(3) = 0.975
4	x(4) = 7.32	y(4) = -8.73	z(4) = 4.59	m(4) = 0.229
5	x(5) = 1.22	y(5) = 15.12	z(5) = 5.88	m(5) = 0.434

Tabela 3.6: Atribuição dos *arrays* dentro do Fortran para as coordenadas e massas de 5 partículas.

```
1  ! PROGRAMA PARA CALCULAR O CENTRO DE MASSA
2  ! COMPOSTA POR 5 PARTICULAS
3  !
4  ! EXEMPLO DA UTILIZACAO DE VARIOS ARRAYS
5  ! UNIDIMENSIONAIS
6  !
7  program rcm
8  implicit none
9  !
10 real(kind=4), dimension(5) :: x, y, z, m
11 real(kind=4) :: xcm, ycm, zcm, mtotal
12 integer :: i
13 !
14 ! ZERANDO AS VARIAVEIS PARA EVITAR LIXO DE MEMORIA
15 xcm = 0
16 ycm = 0
17 zcm = 0
18 mtotal = 0
19 !
20 x(1)=4.37; x(2)=9.27; x(3)=3.35; x(4)=7.23; x(5)=1.22
21 !
22 y(1)=1.23; y(2)=-1.23; y(3)= 8.23; y(4)=-8.73; y(5)=15.12
23 !
24 z(1)=9.55; z(2)=0.51; z(3)=6.53; z(4)=4.59; z(5)=5.88
25 !
26 m(1)=0.125; m(2)=0.333; m(3)=0.975; m(4)=0.229; m(5)=0.434
27 !
28 ! CALCULA O SOMATORIO CENTRO DE MASSA
29 do i=1, 5
30       xcm = xcm + (x(i) * m(i))
31       ycm = ycm + (y(i) * m(i))
32       zcm = zcm + (z(i) * m(i))
33       mtotal = mtotal + m(i)
34 end do
35 !
36 ! POSICOES DO CENTRO DE MASSA
```

```
xcm = xcm / mtotal
ycm = ycm / mtotal
zcm = zcm / mtotal
!
! ESCREVE O RESULTADO DA POSICAO DO CENTRO DE MASSA
write(*,*) xcm, ycm, zcm
!
stop
end program rcm
!
```

3.6.2 Matrizes e *Arrays Multi-dimensionais*

Seguindo o um esquema parecido com o de um *array* unidimensional, uma matriz é entendido como um *array* bi-dimensional, por exemplo, uma matriz quadrada de ordem 3 que tem 3 linhas e 3 colunas, assim com o total de 9 elementos na matriz. No entanto um *array* bi-dimensional não precisa necessariamente ter as dimensões de linhas e colunas iguais, poderia ser, por exemplo uma matriz 3x4 (3 linhas e 4 colunas) com total de 12 elementos.

Pegaremos como exemplo duas matrizes quadradas de ordem 3 e faremos a multiplicação.

$$A = \begin{pmatrix} 5 & 5 & 5 \\ 5 & 5 & 5 \\ 5 & 5 & 5 \end{pmatrix}, B = \begin{pmatrix} 3 & 3 & 3 \\ 3 & 3 & 3 \\ 3 & 3 & 3 \end{pmatrix}, C = A \times B = \begin{pmatrix} 45 & 45 & 45 \\ 45 & 45 & 45 \\ 45 & 45 & 45 \end{pmatrix}$$

```
!
! Array bi-dimensional
!
program prog8
implicit none
!
integer, dimension(3,3) :: a, b, c
integer :: i, j, k
!
do i=1, 3
        do j=1, 3
                a(i,j) = 5
                b(i,j) = 3
                c(i,j) = 0
        end do
end do
!
! Multiplicando as matrizes
do i=1,3
        do j=1,3
                do k=1,3
                        c(i,j) = c(i,j) + ( a(i,k) * b(k,j) )
```

```
23                  end do
24          end do
25  end do
26  !
27  ! Escrevendo a matriz
28  do i=1,3
29          write(*,*) (c(i,j),j=1,3)
30  end do
31  !
32  end program prog8
33  !
```

O próximo programa diferencia do anterior apenas pela declaração dos elementos de matriz diferentes um dos outros.

```
1   !
2   ! Array bi-dimensional
3   !
4   program prog9
5   implicit none
6   !
7   integer, dimension(3,3) :: a, b, c
8   integer :: i, j, k
9   !
10  ! Zerando as variaveis indexadas
11  do i=1, 3
12          do j=1, 3
13                  a(i,j) = 0
14                  b(i,j) = 0
15                  c(i,j) = 0
16          end do
17  end do
18  !
19  ! Atribuindo valores as variaveis indexadas
20  a(1,1)= 5; a(1,2)= 3; a(1,3)= 1
21  a(2,1)=23; a(2,2)=13; a(2,3)=17
22  a(3,1)= 7; a(3,2)=11; a(3,3)=19
23  !
24  b(1,1)= 2; b(1,2)= 8; b(1,3)=14
25  b(2,1)= 4; b(2,2)=10; b(2,3)=16
26  b(3,1)= 6; b(3,2)=12; b(3,3)=18
27  !
28  ! Multiplicando as matrizes
29  do i=1,3
30          do j=1,3
31                  do k=1,3
32                          c(i,j) = c(i,j) + ( a(i,k) * b(k,j) )
33                  end do
34          end do
```

```
35 end do
36 !
37 ! Escrevendo a matriz
38 do i=1,3
39           write(*,*) (c(i,j),j=1,3)
40 end do
41 !
42 end program prog9
43 !
```

Os dois programas anteriores realizam um *looping* para cada variável k, j e i iniciado-se da variável k, depois para variável j e por último para variável i. É importante tem em mente o que o programa está realizando, ou seja, para cada interação qual a operação que ele está fazendo.

i	j	k	c(i,j) = c(i,j) + (a(i,k) * b(k,j))	Somatório
1	1	1	c(1,1) = c(1,1) + (a(1,1) * b(1,1))	$c_{11} = a_{11} * b_{11}$
1	1	2	c(1,1) = c(1,1) + (a(1,2) * b(2,1))	$c_{11} = (a_{11} * b_{11}) + (a_{12} * b_{21})$
1	1	3	c(1,1) = c(1,1) + (a(1,3) * b(3,1))	$c_{11} = (a_{11} * b_{11}) + (a_{12} * b_{21}) + (a_{13} * b_{31})$
1	2	1	c(1,2) = c(1,2) + (a(1,1) * b(1,2))	$c_{12} = a_{11} * b_{12}$
1	2	2	c(1,2) = c(1,2) + (a(1,2) * b(2,2))	$c_{12} = (a_{11} * b_{12}) + (a_{12} * b_{22})$
1	2	3	c(1,2) = c(1,2) + (a(1,3) * b(3,2))	$c_{12} = (a_{11} * b_{12}) + (a_{12} * b_{22}) + (a_{13} * b_{32})$
1	3	1	c(1,3) = c(1,3) + (a(1,1) * b(1,3))	$c_{13} = a_{11} * b_{13}$
1	3	2	c(1,3) = c(1,3) + (a(1,2) * b(2,3))	$c_{13} = (a_{11} * b_{13}) + (a_{12} * b_{23})$
1	3	3	c(1,3) = c(1,3) + (a(1,3) * b(3,3))	$c_{13} = (a_{11} * b_{13}) + (a_{12} * b_{23}) + (a_{13} * b_{33})$
2	1	1	c(2,1) = c(2,1) + (a(2,1) * b(1,1))	$c_{21} = (a_{21} * b_{11})$
2	1	2	c(2,1) = c(2,1) + (a(2,2) * b(2,1))	$c_{21} = (a_{21} * b_{11}) + (a_{22} * b_{21})$
2	1	3	c(2,1) = c(2,1) + (a(2,3) * b(3,1))	$c_{21} = (a_{21} * b_{11}) + (a_{22} * b_{21}) + (a_{23} * b_{31})$
2	2	1	c(2,2) = c(2,2) + (a(2,1) * b(1,2))	$c_{22} = (a_{21} * b_{12})$
2	2	2	c(2,2) = c(2,2) + (a(2,2) * b(2,2))	$c_{22} = (a_{21} * b_{12}) + (a_{22} * b_{22})$
2	2	3	c(2,2) = c(2,2) + (a(2,3) * b(3,2))	$c_{22} = (a_{21} * b_{12}) + (a_{22} * b_{22}) + (a_{23} * b_{32})$
2	3	1	c(2,3) = c(2,3) + (a(2,1) * b(1,3))	$c_{23} = (a_{21} * b_{13})$
2	3	2	c(2,3) = c(2,3) + (a(2,2) * b(2,3))	$c_{23} = (a_{21} * b_{13}) + (a_{22} * b_{23})$
2	3	3	c(2,3) = c(2,3) + (a(2,3) * b(3,3))	$c_{23} = (a_{21} * b_{13}) + (a_{22} * b_{23}) + (a_{23} * b_{33})$
3	1	1	c(3,1) = c(3,1) + (a(3,1) * b(1,1))	$c_{31} = (a_{31} * b_{11})$
3	1	2	c(3,1) = c(3,1) + (a(3,2) * b(2,1))	$c_{31} = (a_{31} * b_{11}) + (a_{32} * b_{21})$
3	1	3	c(3,1) = c(3,1) + (a(3,3) * b(3,1))	$c_{31} = (a_{31} * b_{11}) + (a_{32} * b_{21}) + (a_{33} * b_{31})$
3	2	1	c(3,2) = c(3,2) + (a(3,1) * b(1,2))	$c_{32} = (a_{31} * b_{12})$
3	2	2	c(3,2) = c(3,2) + (a(3,2) * b(2,2))	$c_{32} = (a_{31} * b_{12}) + (a_{32} * b_{22})$
3	2	3	c(3,2) = c(3,2) + (a(3,3) * b(3,2))	$c_{32} = (a_{31} * b_{12}) + (a_{32} * b_{22}) + (a_{33} * b_{32})$
3	3	1	c(3,3) = c(3,3) + (a(3,1) * b(1,3))	$c_{33} = (a_{31} * b_{13})$
3	3	2	c(3,3) = c(3,3) + (a(3,2) * b(2,3))	$c_{33} = (a_{31} * b_{13}) + (a_{32} * b_{23})$
3	3	3	c(3,3) = c(3,3) + (a(3,3) * b(3,3))	$c_{33} = (a_{31} * b_{13}) + (a_{32} * b_{23}) + (a_{33} * b_{33})$

Tabela 3.7: Multiplicação de Matriz.

Observe que a tabela 3.7 apresenta o que a expressão $c(i,j) = c(i,j) + (a(i,k) * b(k,j))$ para k, j, e i, variando de 1 à 3. Para um valor de $i = 1$ e $j = 1$, k irá variar de 1 à 3, depois, para $i = 1$ e $j = 2$, k irá variar de 1 à 3 novamente. É importante perceber finalizado um *do* mais externo o *do* mais interno retorna a contagem inicial.

No que se refere a um *array* multi-dimensional apenas declaramos a variável indexada como tendo várias dimensões. por exemplo:

```
1 !
2 integer , dimension(3,3,3,5) :: a
3 !
```

assim a variável indexa a terá direções em x, y, z e k.

Atividade de reforço em *arrays*

Escreva um programa ou vários programas para o que se pede a seguir:

- Mova um *array* A_i para o *array* B_i, com $i = 1, 2, 3, ..., 10$;
- Armazene as constantes inteiras $i = 1, 2, 3, ..., 22$, em um *array* de nome X;
- Calcule $S = \sum_{i=1}^{N} A_i$, com $N = 50$;
- Calcule $C_i = \sqrt{A_i^2 - B_i^2}$, para $i = 1, 2, 3, ..., 25$;
- Calcule $z = \sum_{j=1}^{m} \sum_{i=1}^{N} a_j x b_i$, para m e n da sua escolha;
- Dado um *array* unidimensional x com 50 elementos, calcular os elementos do *array* y, pela fórmula $y_i = x_{i+1} - x_i$, sendo $i = 1, 2, 3, ..., 49$;
- Dado um *array* unidimensional z com 50 elementos inteiros, substituir cada elemento por ele mesmo multiplicado pela sua posição no conjunto, isto é, substituir m_i por $i.m_i$, para $i = 1, 2, 3, ..., 50$;
- Dado um *array* unidimensional z com 20 elementos, determinar o elemento de maior valor algébrico e sua posição no conjunto armazenando nas variáveis A e N. Utilize o *array* a seguir:
  a(1)=41.2; a(2)=63.0; a(3)=63.5; a(4)=25.3; a(5)=76.2
  a(6)=57.3; a(7)=33.1; a(8)=58.9; a(9)=15.9; a(10)=20.6
  a(11)=90.1; a(12)=77.0; a(13)=67.9; a(14)=49.9; a(15)=34.3
  a(16)=63.6; a(17)=28.4; a(18)=51.4; a(19)=17.3; a(20)=80.5

- Dadom um *array* bi-dimensional B_{ij}, $i = 5$ e $j = 6$, determinar o elemento de maior valor algébrico e armazená-lo em A. Armazenar a posição do *array* bi-dimensional nas variáveis l (linha) e c (coluna). Utilize o *array* abaixo:
  a(1,1)=0.0; a(1,2)=83.2; a(1,3)= 61.5; a(1,4)=90.6; a(1,5)=86.8; a(1,6)=39.9
  a(2,1)=80.5; a(2,2)=53.0; a(2,3)=12.9; a(2,4)=10.0; a(2,5)=45.4; a(2,6)=31.5
  a(3,1)=95.4; a(3,2)=56.9; a(3,3)=67.6; a(3,4)=39.2; a(3,5)=23.2; a(3,6)=84.4
  a(4,1)=31.7; a(4,2)=19.8; a(4,3)=10.3; a(4,4)=31.5; a(4,5)=30.4; a(4,6)=93.9
  a(5,1)=13.1; a(5,2)=26.4; a(5,3)=3.3; a(5,4)=24.4; a(5,5)=37.7; a(5,6)=87.9

- Dados os *arrays* X com 6 componentes e A com 5 linhas e 6 colunas, determine o *array* B pela fórmula: $B_i = \sum_{j=1}^{6} A_{ij} X_j$, com $i = 1, 2, 3, ..., 5$. Utilize os *arrays* dos problemas anteriores;
- Dados p números reais, calcular o somatório dos números negativos.

3.6.3 Alocação Dinâmica de Memória

A idéia da alocação dinâmica de memória é simplesmente a de ser possível a utilização de um *array* sem ser necessário declararmos o tamanho desse *array* no início do programa junto à declaração de variáveis. Inicialmente, como já fizemos em um exemplo anterior, o tamanho do *array* era declarado no início do programa como no programa do cálculo do centro de massa. A declaração tinha a seguinte forma:

real(kind=4), dimension(5) :: x, y, z, m

- **real(kind=4)** indica qual tipo de número irá ser utilizado, se real, inteiro ou caracter. Neste exemplo é um número real com simples precisão;
- **dimension(5)** indica que a variável ou as variáveis serão um *array* unidimensional de tamanho 5, assim estamos dizendo que as variáveis terão tamanho fixo igual a 5;
- **::** um separador, após esse separador virão as variáveis;
- **x, y, z, m** são as variáveis propriamente ditas, podemos entendê-las como x_i, y_i, z_i e m_i onde i assume valores de 1 a 5.

Da maneira como está declarado acima, as variáveis indexadas x, y, z e m terão tamanho fixo igual a 5. O propósito desta seção é fazer com que o tamanho seja dessas variáveis indexadas seja atribuído a partir de um arquivo externo e também que possa alterar de tamanho durante a execução do programa sem a necessidade de uma segunda compilação.

Para utilizarmos a alocação dinâmica de memória, ou seja, atribuir um tamanho para um *array*, precisamos dizer para o compilador que esta operação será realizada. Isto é feito no momento em que se declara uma variável:

real(kind=4), allocatable, dimension(:) :: x, y, z, m

- **real(kind=4)** indica qual tipo de número irá ser utilizado, se real, inteiro ou caracter. Neste exemplo é um número real com simples precisão;
- **allocatable** indica que o tamanho das variáveis terão um tamanho inidicado posteriormente, por exemplo, por uma variável inteira que foi lida de um arquivo de entrada;
- **dimension(:)** indica que a variável ou as variáveis serão um *array* unidimensional de tamanho a ser declarado posteriormente, assim estamos dizendo que as variáveis não terão tamanho fixo e poderão assumir qualquer tamanho dependendo do tamanho desejado ou limitado pela memória RAM do computador;

- **::** um separador, após esse separador virão as variáveis;
- **x, y, z, m** são as variáveis propriamente ditas, podemos entendê-las como x_i, y_i, z_i e m_i onde i assume valor qualquer no futuro.

Em conjunto com esta declaração de variáveis, o *array* deve possuir um tamanho a ser informado poteriorment. O tamanho desse *array* irá depender do problema a ser resolvido e também do tamanho da memória RAM do computador. A variável que indica o tamanho do *array* poderá ser lido a partir de um arquivo ou a partir do teclado. Uma vez que temos o tamanho desse *array* é necessário dizer ao compilador que o *array* terá um tamanho n. No programa isto será feito da seguinte forma:

allocate(x(n),stat=err_x)
allocate(y(n),stat=err_y)
allocate(z(n),stat=err_z)
allocate(m(n),stat=err_m)

A variável n indica o tamanho do *array* e deve possuir algum valor antes da declaração *allocate*. As variáveis *err_x*, *err_y*, *err_z* e *err_m* são variáveis inteiras associadas à alocação das variáveis, no processo de alocação se essas variáveis retornarem 0 (zero) significa que elas foram alocadas com sucesso ou se retornarem qualquer valor diferente de zero significa que houve algum problema com a alocação do *array*. Para exemplificar faremos o programa abaixo:

```
1  !
2  ! PROGRAMA PARA EXEMPLIFICAR A ALOCACAO DINAMICA DE MEMORIA
3  !
4  program allocdyn
5  implicit none
6
7  real(kind=4), allocatable, dimension(:) :: x, y
8  integer :: i, n, err_x, err_y
9  !
10 err_x=0; err_y=0
11 !
12 write(*,*)'Digite o tamanho do seu array (numero inteiro):'
13 read(*,*) n
14 !
15 allocate(x(n),stat=err_x)
16 allocate(y(n),stat=err_y)
17 !
18 if( err_x /= 0 ) then
19         write(*,*)'Problema para allocar o vetor x! Saindo!!!'
20         stop
21 end if
22 if( err_y /= 0 ) then
```

```
23          write(*,*)'Problema para allocar o vetor y! Saindo!!!'
24          stop
25  end if
26  !
27  do i=1, n
28          x(i) = 2.0
29          y(i) = 0.0
30  end do
31  do i=1, n
32          if ( i == 1 ) then
33                  y(i) = x(i)
34          else
35                  y(i) = y(i-1) + 1.0
36          end if
37  end do
38  !
39  do i=1, n
40          write(*,1000) i,x(i),y(i)
41  end do
42  !
43  deallocate(x)
44  deallocate(y)
45  !
46  1000 format(i6,f10.3,f10.3)
47  !
48  stop
49  end program allocdyn
50  !
```

3.7 Comandos de entrada e saída (I/O)

Desde o início, os programas feitos até o momento utilizou uma entrada já estabelecida dentro do próprio programa através de uma variável. No entanto com o Fortran podemos atribuir valores à uma variável através do teclado ou então através de um arquivo. Isto é o que chamamos de *input*.

Após o processamento dos dados pelo programa a saída pode ser basicamente de duas maneiras, uma pelo tela do computador ou então para um arquivo no formato ASCII ou binário armazenado no computador. Isto é o que chamamos de *output*.

Para ler algo e atribuir algum valor para uma variável utilizamos o comando *read(*,*)*. O conteúdo entre parênteses tem como objetivo indicar de qual local será lido, teclado ou arquivo, indicado pelo primeiro asterisco e como será lido, com ou sem formato, indicado pelo segunda asterisco. Na presente forma do comando *read* o primeiro asterisco indica que será lido do teclado e o segundo asterisco indica que será lido no formato livre.

A variável que será atribuída pelo comando *read* deve estar coerente com a declaração de variáveis feitas no início do programa. Se for declarada uma variável real o número a ser lido será um número real, se for caracter então será lido um ou um conjunto de caracter. Faremos um programa parar exemplificar a entrada e saída de dados[8].

```
1  !
2  ! PROGRAMA PARA EXEMPLIFICAR I/O, SENDO A ENTRADA PELO TECLADO
3  ! E A SAIDA PELA TELA DO COMPUTADOR
4  !
5  program io1
6  implicit none
7  !
8  real(kind=4)    :: r_input,r_output
9  integer         :: i_input,i_output
10 character(20)   :: c_input,c_output
11 !
12 write(*,*)'Digite um numero real:'
13 read(*,*) r_input
14 !
15 write(*,*)'Digite um numero inteiro:'
16 read(*,*) i_input
17 !
18 write(*,*)'Digite uma frase com ate 20 digitos:'
19 read(*,*) c_input
20 !
21 r_output = r_input
```

[8]No programa *io1.f90* quando estiver testando o programa ira perceber que ao colocar a seguinte frase *teste do programa* a saída será somente *teste*, para imprimir a frase toda a entrada deve estar entre aspas simples ou duplas como *'teste do programa'*

```
22 i_output = i_input
23 c_output = c_input
24 !
25 write(*,*) 'O que voce digitou como real foi:',r_output
26 write(*,*) 'O que voce digitou como inteiro foi:',i_output
27 write(*,*) 'O que voce digitou como frase foi:',c_output
28 !
29 stop
30 end program io1
31 !
```

No programa *io1.f90* a saída poderia ser simplesmente as variáveis *r_input, i_input e c_input*, não teria a necessidade de utilizar as variáveis *r_output, i_output e c_output*.

As combinações de entrada e saída poderiam ser: (1) teclado-tela, (2) teclado-arquivo, (3) teclado-impressora, (4) arquivo-tela, (5) arquivo-impressora, (6) arquivo-arquivo e (7) programa-tela. Dessas possíveis combinações já utilizamos as combinações (1) e (7), as combinações que envolvem impressoras como *output* dificilmente iremos utilizá-las pois são poucas as vezes onde acertamos os output e deixamos como desejamos, é preferível verificar o arquivo antes de imprimir um trabalho. A combinação mais comum é a (6), pois geralmente montamos um arquivo de entrada, o programa irá processar as informações e depois o resultado será escrito em um arquivo de saída, fazendo com que a execução do trabalho seja mais rápido além da possibilidade de poder interagir com *scripts* construídos em *bash*.

Para exemplificar uma entrada e saída utilizando arquivos, utilizaremos o problema do cálculo do centro de massa de um sistema de partículas. Primeiramente criaremos um arquivo com o nome *entrada.dat*, com o *joe*, com o seguinte conteúdo abaixo, em que a primeira linha indica a coordenada x_1, y_1, z_1 e massa m_1 da partícula 1, ...e a quinta linha indica a coordenada x_5, y_5, z_5 e massa m_5 da partícula 5.

4.37	1.23	9.55	0.125
9.27	-1.23	0.51	0.333
3.35	8.23	6.53	0.975
7.23	-8.73	4.59	0.229
1.22	15.12	5.88	0.434

O programa será:

```
1 !
2 ! PROGRAMA PARA CALCULAR O CENTRO DE MASSA
3 ! COMPOSTA POR 5 PARTICULAS
4 !
5 ! EXEMPLO DA UTILIZACAO DE VARIOS ARRAYS UNIDIMENSIONAIS
6 ! E TAMBEM A ENTRADA COMO SENDO UM ARQUIVO E A SAIDA TAMBEM
```

```
7  ! PARA UM ARQUIVO
8  !
9  program rcm2
10 implicit none
11 !
12 ! DECLARACAO DE VARIAVEIS
13 real(kind=4), dimension(5) :: x, y, z, m
14 real(kind=4) :: xcm, ycm, zcm, mtotal
15 integer :: i, j, k, error_input, error_output
16 !
17 ! ZERANDO AS VARIAVEIS
18 error_input  = 0
19 error_output = 0
20 !
21 ! ARQUIVO DE ENTRADA
22 open (UNIT=20, FILE='entrada.dat', STATUS='OLD', ACTION='READ',  IOSTAT
       =error_input)
23 ! ARQUIVO DE SAIDA
24 open (UNIT=21, FILE='saida.dat', STATUS='REPLACE', ACTION='WRITE',
       IOSTAT=error_output)
25 !
26 ! CONDICAO QUE VERIFICA A ABERTURA DOS ARQUIVOS DE ENTRADA
27 if ( error_input /= 0  ) then
28          write(*,*) 'PROBLEMA COM ARQUIVO DE ENTRADA !!!  '
29 else
30          write(*,*) 'ARQUIVO DE ENTRADA ABERTO COM SUCESSO !!!'
31 end if
32 ! CONDICAO QUE VERIFICA A ABERTURA DOS ARQUIVOS DE SAIDA
33 if ( error_output /= 0 ) then
34          write(*,*) 'PROBLEMA COM ARQUIVO DE SAIDA !!!  '
35 else
36          write(*,*) 'ARQUIVO DE SAIDA ABERTO COM SUCESSO !!!'
37 end if
38 !
39 ! LENDO ARQUIVO DE ENTRADA, UNIDADE 20, NOME ENTRADA.DAT
40 do i=1,5
41          read(20,*) x(i),y(i),z(i),m(i)
42 end do
43 !
44 ! ZERANDO AS VARIAVEIS
45 xcm = 0
46 ycm = 0
47 zcm = 0
48 mtotal = 0
49 !
50 ! CALCULANDO O SOMATORIO DO CENTRO DE MASSA
51 do i=1, 5
52          xcm = xcm + (x(i) * m(i))
```

```
53            ycm = ycm + (y(i) * m(i))
54            zcm = zcm + (z(i) * m(i))
55            mtotal = mtotal + m(i)
56    end do
57    !
58    ! CALCULANDO O CENTRO DE MASSA
59    xcm = xcm / mtotal
60    ycm = ycm / mtotal
61    zcm = zcm / mtotal
62    !
63    ! ESCREVENDO PARA O ARQUIVO SAIDA.DAT, UNIDADE 21
64    write(21,*) xcm, ycm, zcm
65    !
66    close(20)
67    close(21)
68    !
69    stop
70    end program rcm2
71    !
```

Detalhes do programa:

- As linhas que iniciam com o ponto de exclamação (!) são comentários;
- As linhas 13, 14, e 15 são as variáveis do programa;
- As linhas 18 e 20 são as declarações dos arquivos de entrada e saída. Nestas duas linhas *UNIT* é o número a que se refere o arquivo e será utilizado posteriormente, *STATUS* indica qual a situação do arquivo se ele existe (old), se ele é novo (new) ou se ele é desconhecido (unknown). *ACTION* indica qual a ação a ser tomada com o arquivo se ele será lido (read) ou se ele será escrito (write). *IOSTAT* é a variável de controle para saber se houve algum tipo de problema ao abrir o arquivo seja ele de leitura ou escrita;
- Entre as linhas 23 e 33 o programa nos diz se os arquivos foram aberto com sucesso ou se houve algum problema na abertura;
- Entre as linhas 36 e 38 o programa está lendo a partir do arquivo da unidade 20 e atribuindo valores às variáveis x_i, y_i, z_i e m_i;
- Entre as linhas 41 a 46 o programa calcula o somatório conforme a equação 3.6.1;
- As linhas 49, 50 e 51 são os cálculos das componentes do centro de massa (x, y e z);
- Na linha 54 o programa escreve para o arquivo da unidade 21 o resultado do centro de massa.

3.8 Especificações de Formato

Como foi visto na seção anterior (seção 3.7) utilizamos os comandos *read* e *write*. Da forma como foi apresentado, esses comandos leram ou escreveram no formato livre, ou seja, sem formato, no entanto é possível que esses comandos (*read* e *write* leiam e escrevam com um formato, por exemplo, um número real seja escrito com quantas casas decimais desejarmos.

O foco desse curso é basicamente trabalhar com números inteiros e reais e também com caracteres, assim direcionaremos as especificações de formato para os números, reais e inteiros, e também caracter. Outras formas de formatação para conversão de inteiros e decimais, para dados lógicos, binários e hexadecimais também são possíveis mas não o faremos aqui, assim seremos pragmáticos dentro do que se destina esse aprendizado de Fortran. Se em um futuro próximo seja necessário a utilização dos formatos que não será exposto aqui na apostila, nos livros apontados nas referências é possível ser encontrado, e com a base vista no decorrer do curso, seremos capaz de utilizá-los quando necessário.

Formatação de números inteiros, reais e caracteres.

- A formatação mais simples é a do número inteiro, pois basta especificarmos qual o tamanho máximo do número inteiro que desejamos utilizar, queremos dizer quantas posições (colunas) esse número máximo irá utilizar em um linha;
- A formatação para um número real é feita levanto em conta (*i*) um número real sem expoente, (*ii*) um número real com expoente com simples precisão ou (*iii*) um número real com dupla precisão;
- A formatação para um caracter é semelhante a formatação para um inteiro basta especificar quantos caracteres deseja ler ou escrever. Um outro modo de pensar em formatação de caracteres é pensar no tamanho da *string* que esse caracter irá utilizar.

Vamos exemplificar os tipos de formatação a partir de um programa.

```
1  !
2  ! PROGRAMA PARA EXEMPLIFICAR OS TIPOS DE FORMATO
3  ! PARA LEITURA E ESCRITA NO FORTRAN PARA NUMEROS
4  ! INTEIROS E REAIS E TAMBEM PARA CARACTERES
5  !
6  program formato
7  implicit none
8  !
9  ! DECLARACAO DE VARIAVEIS
10 real(kind=4) :: r_1, r_2
11 real(kind=8) :: r_3, r_4
12 integer :: i_1, i_2
```

```
13 character(10) :: c_1, c_2
14 !
15 ! NUMEROS REAIS COM SIMPLES PRECISAO
16 r_1 = 3.55891; r_2 = 12345.67891011
17 !
18 ! NUMEROS REAIS COM DUPLA PRECISAO
19 r_3 = 0.00056789d0; r_4 = 123456789.10111213d0
20 !
21 ! NUMEROS INTEIROS
22 i_1 = 123456789; i_2 = 987
23 !
24 ! VARIAVEL CARACTER
25 c_1 = 'abcdefghij'; c_2 = 'abcdefghijklmn'
26 !
27 ! ESCREVENDO NA TELA
28 write(*,900) r_1,r_2
29 write(*,901) r_1,r_2
30 write(*,902) r_3,r_4
31 write(*,903) i_1,i_2
32 write(*,904) c_1,c_2
33 write(*,905) r_1,r_2,r_3,r_4,i_1,i_2,c_1,c_2
34 !
35 ! FORMATOS
36 900 format(f10.3, 2x, f10.6)
37 901 format(e10.3, 2x, e10.5)
38 902 format(d10.4, 2x, d10.8)
39 903 format(i10, 2x, i4)
40 904 format(a10, 2x, a20)
41 905 format(f10.5,1x,f10.5,1x,d10.6,1x,d10.8,1x,i10,1x,i4,a10,1x,a12)
42 !
43 stop
44 end program formato
45 !
```

Se o programa compilar de primeira, ao executar o resultado que aparecerá na tela será:

```
1 [23:05][sjfsato@speed:~/Physics/FisComp_2_sem_2009/aulas/prog]
2 $ ./formato.x
3       3.559   **********
4  0.356E+01   .12346E+05
5 0.5679D-03   **********
6  123456789    987
7 abcdefghij              abcdefghij
8    3.55891 ********** ********** **********  123456789  987abcdefghij
9 [23:05][sjfsato@speed:~/Physics/FisComp_2_sem_2009/aulas/prog]
10 $
```

Porque ao invés de mostrar os números o que apareceu foram vários asteriscos?

Iremos por partes, podemos perceber que o comando *write* está com números (900, 901, 902, 903, 904, 905) ao invés de asteriscos. Como sabemos o segundo asterisco do comando *write* significa que será escrito no formato livre. Ao colocarmos algum indicador, que deve ser um número inteiro com até 6 posições (por exemplo 999999), que são os números 900, 901, 902, 903, 904 e 905 estamos especificando algum tipo de formato. A instrução com o número dentro do comando *write* necessariamente informa ao compilador que existira uma linha com um número igual ao declarado com as informações do formato, isto é, no primeiro comando *write* informamos o número 900 então em alguma linha abaixo do comando *write(*,900)* existirá uma linha que inicia-se com o número 900.

Na linha que inicia-se com o número 900 indicamos o formato desejado e este formato irá aparecer na tela ou então em um arquivo, percebe-se que após o número 900 aparecerá a declaração de formato *format* e em seguida o formato propriamente dito, por exemplo, *f10.3*. Estes formatos tem os seguinte significados:

- **fm.n** - A letra **f** indica que o número a ser escrito é um número real sem expoente. A letra **m** indica o total de espaços este número irá ocupar e a letra **n** o número de espaços depois do ponto, ou seja, quantas casas decimais. f6.2 = 123.12;

- **em.n** - A letra **e** indica que o número a ser escrito é um número real de simples precisão com expoente. A letra **m** indica o total de espaços que o número irá ocupar e a letra **n** o número de espaços depois do ponto, ou seja, quantas casas decimais e10.2 = 0.12345e±03 = 123.45;

- **dm.n** - A letra **e** indica que o número a ser escrito é um número real de dupla precisão com expoente. A letra **m** indica o total de espaços que o número irá ocupar e a letra **n** o número de espaços depois do ponto, ou seja, quantas casas decimais, d14.6 = 123.456789d±01;

- **im** - A letra **i** indica que o número a ser escrito será um número inteiro, consequentemente sem casas decimais e a letra **m** quantos espaços este número inteiro irá ocupar, i6 = %%%199;

- **am** - A letra **a** indica que será escrito um caracter ou uma sequência de caracteres dependendo do valor da letra **m**, a1=w, a5=abcde.

- *Observação (1)*: Considere o caracter % como um espaço;

- *Observação (2)*: O que aparece como **1x** no formato é simplesmente um espaço entre as formatações.

Para responder a questão inicial porque apareceram os asteriscos ao invés de números, levamos em conta que agora formatamos os números e estamos "mandando" imprimir números que estão fora do que foi formatado.

3.9 Subprogramas - *Funções* e *Sub-rotinas*

Esta parte da exploração sobre o Fortran trata-se da utilização de duas ferramentas de extrema utilidades e frequentemente utilizadas em programas grandes e de um maior grau de complexidade.

3.9.1 Funções

As funções são chamadas no Fortran de *function*, tem como utilidade, criarmos funções específicas e a utilização é semelhante a das funções intrínsecas. No Fortran temos dois tipos de funções que são as funções intrínsecas (tabela 3.2) e as funções definidas pelo programador (que é o assunto deste tópico). Quando uma função é chamada ela nos devolve um valor ou um *array* de valores.

```fortran
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! PROGRAMA QUE CALCULA O VALOR DE UM EQUACAO POLINOMIAL
! DE GRAU 2 (fx = a*x**2 + b*x + c) ATRAVEZ DE UMA FUNCAO.
! EXEMPLO DE UMA "FUNCTION"
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
program pl2
implicit none
!
real(kind=8) :: a, b, c, x, fx
!
write(*,*)'Digite os coeficientes da equacao polinomial de grau 2'
read(*,*) a, b, c
write(*,*)'Entre com o valor de x'
read(*,*) x
!
! AS VARIAVEIS DO ARGUMENTO DE fx PODEM SER DIFERENTES DAQUELAS DO
! ARGUMENTO DA FUNCTION. O VALOR DA VARIAVEL DEPENDE DA POSICAO QUE
! ESTA OCUPA ASSIM:
! a RECEBE a1, b RECEBE b1, c RECEBE c1 E x RECEBE x1.
!
write(*,*)'O valor de fx para x=',x,'e ',fx(a,b,c,x)
!
stop
end program pl2
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! - FUNCTION QUE CALCULA O VALOR DE FX
! - ATENCAO: "FX" NAO ENTRA NA DECLARACAO DE VARIAVEIS DA FUNCAO
!            POIS SERA O OUTPUT DA FUNCAO
! - INTENT(IN): SIGNIFICA QUE AS VARIAVEIS REAIS SERAO INFORMADAS
!               ATRAVAES DO PROGRAMA PRINCIPAL
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
real(kind=8) function fx(a1,b1,c1,x1)
implicit none
!
```

```
real(kind=8), intent(in) :: a1, b1, c1, x1
!
fx = a1*(x1*x1) + b1*x1 + c1
!
end function
!
```

Atividade: Construa um programa que tenha as seguintes características:

- leia de um arquivo os valores dos coeficientes de uma equação de segundo grau do tipo $f(x) = ax^2 + bx + c$, leia os valores de x_i (x inicial), x_f (x final) e o intervalo de Δx;
- calcule os valores de $f(x)$ para cada valor de x;
- os valores de x deve variar de 0.1 ($\Delta x = 0.1$);
- imprimir os valores de x e $f(x)$ formatados em um arquivo de saída;
- gerar um gráfico com o programa *xmgrace* a partir dos resultados.

Dica: o programa irá realizar o cálculo de $f(x)$ no intervalo entre x_i e x_f para um passo de $\Delta x = 0.1$. Calcule o número de intervalos entre x_i e x_f e faça o laço variar entre 1 e o número de intervalos. Assim não será necessário utilizar o comando *if* e o programa ficará mais rápido.

3.9.2 Sub-rotinas

As sub-rotinas também seguem as idéias dos subprogramas que permitem calcular alguma expressão fora do programa principal. As diferenças básicas das funções e das sub-rotinas são a forma de chamada dentro do programa principal e como elas retornam os valores.

Quando desejamos obter um valor de uma função intrínseca ou uma função particular do programador, chamamos a função (que é uma variável) e ela mesma nos retorna um valor.

A utilização de uma sub-rotina consiste inicialmente na chamada que é feita pelo comando *call*, por exemplo *call subroutine pitagoras(ca,co,hip)*, em que *pitagoras* é o nome da subrotina e os argumentos *ca* e *co* são os valores de entrada e *hip* são os valores de saída da subrotina.

Note que em uma sub-rotina o nome não retorna o valor e sim uma ou algumas variáveis que estão no argumento que acompanha o nome da subrotina. Em uma function o próprio nome, que é uma variável, é quem retorna algum valor.

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! PROGRAMA PARA EXEMPLIFICAR UMA SUBROTINA
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
!
program prog_sub_1
implicit none
!
real(kind=4) :: ca, co, hip
!
write(*,*)'---> Triangulo Retangulo <---'
write(*,*)'Digite o tamanho do cateto adjacente:'
read(*,*) ca
write(*,*)'Digite o tamanho do cateto oposto:'
read(*,*) co
!
!
! AS VARIAVEIS DO ARGUMENTO DE hipotenusa PODEM SER DIFERENTES DAQUELAS
! DO ARGUMENTO DA SUBROTINA. O VALOR DA VARIAVEL DEPENDE DA POSICAO QUE
! ESTA OCUPA ASSIM:
! ca RECEBE adjacente, co RECEBE oposto, hip RECEBE hipotenusa
!
call hipotenusa(ca,co,hip)
!
write(*,500) hip
500 format('O tamanho da hipotenusa sera: ',f15.6)
!
stop
end program prog_sub_1
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! SUBROTINA QUE CALCULA O VALOR DA HIPOTENUSA
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine hipotenusa(adjacente,oposto,hipote)
implicit none
!
real(kind=4), intent(in) :: adjacente,oposto
real(kind=4), intent(out) :: hipote
!
hipote = sqrt( adjacente*adjacente + oposto*oposto )
!
! UMA SUBROTINA TERMINA SEMPRE COM O COMANDO RETURN
return
!
end subroutine
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

3.10 Compartilhamento de Variáveis - *Module*

Como vimos nos capítulos e seções anteriores as variáveis são declaradas de forma local, isto é, somente o programa principal ou a sub-rotina tem acesso às suas próprias variáveis. Isto significa que o programa principal acessa somente as variáveis que estão declaradas no início do cabeçalho, assim como as sub-rotinas acessam somente as variáveis declaradas no início do cabeçalho.

Com a declaração do *module* podemos colocar as variáveis de modo que todas as sub-rotinas e o programa principal tenham acesso. Particularmente, gostaria de chamar essas variáveis de variáveis globais onde serão disponibilizadas para qualquer sub-rotina que necessitar utilizar dessas variáveis.

```
1  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
2  ! COMPARTILHAMENTO GLOBAL DE VARIAVEIS "MODULE"
3  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
4  module glob_var
5  implicit none
6  !
7  real(kind=8), allocatable, dimension(:) :: atom,x,y,z,mass
8  real(kind=8) :: xcm, ycm, zcm, mtotal
9  integer :: n_par
10 !
11 end module glob_var
12 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
13 !
14 !
15 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
16 ! PROGRAMA QUE EXEMPLIFICA A UTILIZACAO DO MODULE
17 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
18 program progmod
19 use glob_var
20 implicit none
21 !
22 integer :: err_inp, err_out, err_atom, err_x, err_y, err_z, err_mass, i
23 character(20) :: filename1,filename2
24 !
25 write(*,*)'Digite o nome do arquivo que contem as coordenadas'
26 read(*,'(a20)') filename1
27 !
28 write(*,*)'Digite o nome do arquivo de saida (coordenadas do centro de
       massa)'
29 read(*,'(a20)') filename2
30 !
31 open(unit=20, file=filename1, status='old', action='read', iostat=
       err_inp)
32 open(unit=21, file=filename2, status='replace', action='write', iostat=
       err_out)
```

```
!
read(20,*) n_par
allocate(atom(n_par),stat=err_atom)
allocate(x(n_par),stat=err_x)
allocate(y(n_par),stat=err_y)
allocate(z(n_par),stat=err_z)
allocate(mass(n_par),stat=err_mass)
!
do i=1, n_par
        read(20,*) atom(i),x(i),y(i),z(i)
end do
!
call massa
!
call cm
!
write(21,500) xcm, ycm, zcm
500 format(f15.6,f15.6,f15.6)
!
deallocate(x)
deallocate(y)
deallocate(z)
deallocate(mass)
deallocate(atom)
!
stop
end program progmod
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! SUBROTINA QUE CALCULA O CENTRO DE MASSA
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine cm
use glob_var
implicit none
!
integer :: i,j
!
xcm = 0.0d0
ycm = 0.0d0
zcm = 0.0d0
mtotal = 0.0d0
!
do i=1, n_par
        xcm = xcm + (x(i) * mass(i))
        ycm = ycm + (y(i) * mass(i))
        zcm = zcm + (z(i) * mass(i))
```

```
81          mtotal = mtotal + mass(i)
82  end do
83  !
84  xcm = xcm / mtotal
85  ycm = ycm / mtotal
86  zcm = zcm / mtotal
87  !
88  return
89  end subroutine
90  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
91  !
92  !
93  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
94  ! SUBROTINA QUE ATRIBUI O VALOR DA MASSA A UM ELEMENTO
95  ! QUIMICO
96  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
97  subroutine massa
98  use glob_var
99  implicit none
100 !
101 real(kind=8), dimension(111) :: el
102 integer :: i,j
103 !
104 el(1) =   1.00794000d0; el(2) =   4.002602d0; el(3) =   6.9410000d0;
        el(4) =
105   9.0121820d0; el(5) =  10.8110000d0
106 el(6) =  12.01070000d0; el(7) =  14.006700d0; el(8) =  15.9994000d0;
        el(9) =
107  18.9984032d0; el(10) =  20.1797000d0
108 el(11) =  22.98976928d0; el(12) =  24.305000d0; el(13) =  26.9815386d0;
        el(14) =
109  39.0855000d0; el(15) =  30.9737620d0
110 el(16) =  32.06500000d0; el(17) =  35.453000d0; el(18) =  39.9480000d0;
        el(19) =
111  39.0983000d0; el(20) =  40.0780000d0
112 el(21) =  44.95591200d0; el(22) =  48.867000d0; el(23) =  50.9415000d0;
        el(24) =
113  51.9961000d0; el(25) =  54.9380450d0
114 el(26) =  55.84500000d0; el(27) =  58.933195d0; el(28) =  58.6934000d0;
        el(29) =
115  63.5460000d0; el(30) =  65.4090000d0
116 el(31) =  69.72300000d0; el(32) =  72.640000d0; el(33) =  74.9216000d0;
        el(34) =
117  78.9600000d0; el(35) =  79.9040000d0
118 el(36) =  83.79800000d0; el(37) =  85.467800d0; el(38) =  87.6200000d0;
        el(39) =
119  88.9058500d0; el(40) =  91.2240000d0
120 el(41) =  92.90638000d0; el(42) =  95.940000d0; el(43) =  98.0000000d0;
```

```
    el(44) =
101.0700000d0; el(45) = 102.9055000d0
el(46) = 106.42000000d0; el(47) = 107.868200d0; el(48) = 112.4110000d0;
    el(49) =
114.8180000d0; el(50) = 118.7100000d0
el(51) = 121.76000000d0; el(52) = 127.600000d0; el(53) = 126.9044700d0;
    el(54) =
131.2930000d0; el(55) = 132.9054519d0
el(56) = 137.32700000d0; el(57) = 138.905470d0; el(58) = 140.1160000d0;
    el(59) =
140.9076500d0; el(60) = 144.2420000d0
el(61) = 145.00000000d0; el(62) = 150.360000d0; el(63) = 151.9640000d0;
    el(64) =
157.2500000d0; el(65) = 158.9253500d0
el(66) = 162.50000000d0; el(67) = 164.930320d0; el(68) = 167.2590000d0;
    el(69) =
168.9342100d0; el(70) = 173.0400000d0
el(71) = 174.96700000d0; el(72) = 178.490000d0; el(73) = 180.9478800d0;
    el(74) =
183.8400000d0; el(75) = 186.2070000d0
el(76) = 190.23000000d0; el(77) = 192.217000d0; el(78) = 195.0840000d0;
    el(79) =
196.9665690d0; el(80) = 200.5900000d0
el(81) = 204.38330000d0; el(82) = 207.200000d0; el(83) = 208.9804000d0;
    el(84) =
209.0000000d0; el(85) = 210.0000000d0
el(86) = 222.00000000d0; el(87) = 223.000000d0; el(88) = 226.0000000d0;
    el(89) =
227.0000000d0; el(90) = 232.0380600d0
el(91) = 231.03588000d0; el(92) = 238.028910d0; el(93) = 237.0000000d0;
    el(94) =
244.0000000d0; el(95) = 243.0000000d0
el(96) = 247.00000000d0; el(97) = 247.000000d0; el(98) = 251.0000000d0;
    el(99) =
252.0000000d0; el(100)= 257.0000000d0
el(101)= 258.00000000d0; el(102)= 259.000000d0; el(103)= 262.0000000d0;
    el(104)=
261.0000000d0; el(105)= 262.0000000d0
el(106)= 266.00000000d0; el(107)= 264.000000d0; el(108)= 277.0000000d0;
    el(109)=
268.0000000d0; el(110)= 281.0000000d0
el(111)= 272.00000000d0
!
do i=1,n_par
        do j=1, 111
                if ( atom(i) == j ) then
                        mass(i) = el(j)
                end if
```

```
155           end do
156 end do
157 !
158 return
159 end subroutine
160 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

3.11 Geradores de Números Aleatórios

3.11.1 Um Gerador de Números Aleatórios Simples

Observação Antes de iniciar esta seção, deve ser vista a seção do compartilhamento de variáveis **Module**.

Em vários problemas computacionais é imprescindível a utilização de um gerador de números aleatórios. O propósito em geral é construir uma amostragem que possua algum tipo de ruido. Em outros casos como a técnica de Monte Carlo é muito importante que se utilize um bom gerador de números aleatórios.

Este tema é muito rico em discussões e temos livros específicos sobre geradores de números aleatórios, inclusive especificações e classes de geradores de números aleatórios. De fato o nosso propósito aqui não é estudar a construção de geradores de números aleatórios e sim a utilização deles, no entanto temos que nos atentar ao detalhe de sempre testarmos a qualidade dos geradores de números aleatórios. Um teste simples é verificar a média para uma amostragem dos números aleatórios gerados, em geral geradores de números aleatórios nos devolvem números $0 \leqslant n < 1$ a média de vários números gerados deve ser próximo de 0.5. Outro teste bastante simples de se fazer é construir um histograma do números gerados, quando o número de sorteios tender para o infinito o histograma tem que mostrar um resultado constante.

Na linguagem específica o gerador de números aleatórios que iremos estudar é na verdade um pseudo gerador de números aleatórios pois utilizamos algoritmos, um modelo, para gerar esses números, mas como dito anteriormente sobre essa discussão é melhor conversar com algum especialista sobre o assunto.

Este algoritmo[9] que veremos a seguir utiliza uma *semente*, um número inteiro positivo, para disparar o gerador de números aleatórios. Vale frisar que para uma semente sempre teremos a mesma sequência de números aleatórios. Não temos um números máximo de números aleatórios e isto irá depender do tipo de problema que está estudando ou ira resolver.

```
1  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
2  ! VARIAVEL N SEMPRE SERA COMPARTILHADA ENTRE AS SUBROTINAS
3  ! QUE GERAM OS NUMEROS ALEATORIOS.
4  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
5  module aleatorio
6  implicit none
7  integer, save :: n=6677
8  end module aleatorio
9  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
10 !
```

[9]Este algoritmo foi adaptado da referência [5], páginas 321-325. Podemos encontrar um excelente gerador de números aleatórios no livro da referência [8].

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! PROGRAMA QUE GERAM NUMEROS ALEATORIOS
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
program gna
implicit none
!
real :: media          ! MEDIA DOS NUMEROS ALEATORIOS
integer :: i,j         ! UTILIZADOS NOS DO'S
integer :: isemente    ! SEMENTE INICIAl
real :: nran           ! NUMERO ALEATORIO
real :: soma           ! SOMA DOS NUMEROS ALEATORIOS
!
write(*,*) 'Entre com a semente: '
read(*,*) isemente
!
! CHAMANDO A SUBROTINA SEMENTE
call semente(isemente)
!
write(*,*) '10 numeros randomicos'
do i=1, 10
        ! CHAMADA DO GERADOR DE NUM.ALEAT.
        call num_aleatorio(nran)
        write(*,500)nran
end do
!
!
write(*,*)'Media de 100 numeros aleatorios, de 5 sequencias de 100'
do j=1, 5
        soma = 0.
        do i=1,100
                call num_aleatorio(nran)
                soma = soma + nran
        end do
        media = soma / 100.0
        write(*,500) media
end do
!
500 format(3x,f15.6)
stop
end program gna
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! GERA O NUMERO ALEATORIO
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine num_aleatorio(nran)
use aleatorio
```

```
59 implicit none
60 !
61 real, intent(out) :: nran
62 n = mod( 8127*n+28417 , 134453 )
63 nran = real(n) / 134453.0
64 !
65 end subroutine num_aleatorio
66 !
67 !
68 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
69 ! TRANSFORMA A SEMENTE EM UM NUMERO POSITIVO
70 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
71 subroutine semente(isemente)
72 use aleatorio
73 implicit none
74 integer, intent(in) :: isemente
75 !
76 n=abs(isemente)
77 !
78 end subroutine semente
79 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

A subrotina *num_aleatorio* utiliza a equação $n = MOD(8127 * n + 28417, 134453)$, lembremos que MOD nos dá o resto da divisão de $8127 * n + 28417$ por 134453. O valor de n é atribuído através do número inteiro que digitamos dada pela variável *isemente*, o número n será base e a entrada do novo dará o número aleatório. O número aleatório de fato será por $nran = real(n)/134453.0$. A idéia deste código é que o resto da divisão sempre será um número compreendido entre 0 e 134453 no qual será dividido pelo número 134453, assim o número randômico poderá estar compreendido entre $0 \leqslant n > 1$. Na tabela 3.8 temos a sequencia de 5 números aleatórios tendo como partida um valor da semente igual a 10. Pode ser feito esta sequência no caderno com um auxílio da calculadora, para a primeira linha veja qual o valor do resto para a divisão entre os números $(8127 * 10 + 28417)$ e 134453 que deve ser igual ao valor de n, com o valor de n divida por 134453, assim terá o valor de *nran*.

	n	nran
1	109687	0.815784
2	31276	0.232611
3	92299	0.686462
4	29103	0.216450
5	45671	0.339672

Tabela 3.8: Sequência de números aleatórios (nran) e do número que alimenta a equação dos números aleatórios (n).

3.11.2 Um Gerador de Números Aleatórios Sofisticado

```

!PROGRAMA GERADOR DE NUMEROS ALEATORIOS COM PERIODICIDADE DE 10^22
!PODE-SE ALTERAR OS VALORES k1, k2 e k3 (ix=ieor(ix,ishft(ix,k1)))
!PARA AUMENTAR A PERIODICIDADE.
!
!Extraido das referencias abaixo:
!Marsaglia, G., and Zaman, A. 1994, Computers in Physics, vol. 8, pp. 117-121.
!Marsaglia, G. 1985, Linear Algebra and Its Applications, vol. 67, pp. 147-156.
!
!PRIMEIRA IMPLEMENTACAO: JOSIEL CARLOS DE SOUZA GOMES SEGUNDO SEMESTRE DE 2016
!ALTERACAO: FERNANDO SATO PRIMEIRO SEMESTRE DE 2017
!COMENTARIOS: LEONARDO DA MOTTA DE VASCONCELOS TEIXEIRA 1o. SEMESTRE DE 2017
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
!PROGRAMA MAIN
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
program alea5
!
```

```
24 implicit none
25 !
26 !!VARIAVEL PARA ARAZENAR O NUMERO ALEATORIO
27 real(kind = 8) :: xx
28 !!VARIAVEL SEMENTE INICIAL, VARIAVEL TAMANHO ARRAYS
29 integer(8) :: id,n
30 !!VARIAVEIS DE CONTROLE DOS DO'S
31 integer(8) :: i, j
32 !!CONTADORES PARA CONTROLE
33 integer(8) :: cont, K
34 !!VARIAVEIS DE VERIFICACAO DE ERRO DE ABERTURA DOS ARQUIVOS
35 integer :: error_output, error_output1
36 !!ARRAYS PARA COMPARACAO DE REPETICOES DE NUMEROS GERADOS
37 real(kind = 8),allocatable,dimension (:) :: a,b
38 !
39 !!SEMENTE INICIAL
40 id = 6677
41 !!CONTADOR QUE INDICA O PASSO CORRESPONDENTE AO NUMERO ALEATORIO GERADO
42 cont=0
43 !!QUANTIDADE DE NUMEROS ALEATORIOS SEQUENCIIS A SE VERIFICAR AS
       REPETICOES
44 n=10
45 !
46 !
47 !ARQUIVO QUE ARMAZENA QUANTAS VEZES O PRIMEIRO NUMERO DO ARRAY FOI
       REPETIDO NA
48 !SEQUENCIA DE NUMEROS ALEATORIOS GERADOS
49 open(unit=19,file='contador6.dat',status='replace',action='write',
50 iostat=error_output)
51 !
52 if (error_output /=0 ) then
53   write(*,*) 'PROBLEMA COM ARQUIVO CONTADOR6.DAT'
54 end if
55 !
56 !
57 !ARQUIVO QUE ARMAZENA OS NUMEROS CONSECUTIVOS, NO ARRAY a, AO VALOR
       ARMAZENADO
58 !EM a(1,) QUE SE REPETIRAM APOS CONFIRMADA
59 !A REPETICAO DO PRIMEIRO VALOR
60 open(unit=20,file='result.dat',status='replace',action='write',
61 iostat=error_output1)
62 !
63 if (error_output1 /=0 ) then
64   write(*,*) 'PROBLEMA COM ARQUIVO CONTADOR.DAT'
65 end if
66 !
67 !
68 !ALOCACAO DE UM TAMANHO n AOS ARRAYS DE CONTROLE DE COMPARACAO
```

```
69  allocate(a(n))
70  allocate(b(n))
71  !
72  !
73  !ARMAZENAMENTO DOS n PRIMEIROS NUMEROS ALEATORIOS GERADOS, NO ARRAY a,
        QUE
74  SERAO
75  !UTILIZADOS PARA VERIFICACAO DE REPETICOES DE NUMEROS DO METODO
76  !
77  do i = 1, n
78     call ran1sub(id,xx)
79     a(i) = xx
80     write(*,*) a(i)
81  end do
82  !
83  !
84  !DEFINICAO DE i PARA GARANTIR QUE O LOOP CONTINUE AD INFINITUM ATE A
        CONDICAO
85  DE
86  !REPETICAO SER ATINGIDA
87  i = 1
88  !
89  !
90  !COMANDO DO ONDE CHAMAMOS A SUBROTINA GERADORA DE NUMEROS ALEATORIOS
91  !INDEFINIDAMENTE, E EXECUTAMOS A VERIFICACAO DOS NUMEROS GERADOS PARA
92  !REPETICOES SE OS n NUMEROS INICIAIS ARMAZENADOS FOREM TODOS REPETIDOS,
93  !O PROCESSO E INTERROMPIDO ESTE LOOPING SERVE PARA VERFICARMOS A
94  !FREQUENCIA DESTE CODIGO
95  !
96  do while (i>0)
97            !
98            cont = cont + 1
99            call ran1sub(id,xx)
100 !
101           !VERIFICACAO SE O PRIMEIRO NUMERO ALEATORIO GERADO SE REPETE
102 !
103           if (xx==a(1)) then
104 !
105           !CASO HAJA REPETICAO DO PRIMEIRO NUMERO ALEATORIO GERADO, O
                  NUMERO DO
106           !PASSO EM QUE OCORREU E REGISTRADO NO ARQUIVO 19 E EM SEGUIDA
                  ESCRITO
107           !NA TELA. EM SEGUIDA, ESTE VALOR E ATRIBUIDO AO ARRAY b, QUE
                  SERVIRA
108           !COMO CONTROLE SECUNDARIO PARA A VERIFICACAO DOS  n-1 NUMEROS
109           !ALEATORIOS SEGUINTES
110 !
111                    write(19,*) cont
```

```
                    write(*,*) cont
!
                    b(1) = xx
                    K=0                 !CONTADOR DE CONTROLE QUE INDICA QUANTOS
                                        !NUMEROS ALEATORIOS, ALEM DO PRIMEIRO,
                                        !FORAM REPETIDOS APOS ESTE
!
                    !COMANDO DO ONDE OS n-1 NUMEROS ALEATORIOS SEGUINTES SAO
                    !VERIFICADOS PARA REPETICOES
!
                    do j = 2, n
                              !
                              cont = cont + 1
                              call ran1sub(id,xx)
!
                              b(j) = xx
!
                              !VERIFICANDO SE O j-ESIMO NUMERO SEGUINTE E IGUAL AO
                              !RESPECTIVO NUMERO NO ARRAY a
!
                              if (a(j) == b(j)) then
!
                                        !CASO HAJA REPETICAO, O PROGRAMA REGISTRA O
                                        !NUMERO REPETIDO NO ARQUIVO 20, JUNTAMENTE
                                        !COM SEU RESPECTIVO PASSO
!
                                        write(20,*) b(j), cont
                                        K=K+1
!
                              end if
                              !
                              if (K == n-1) stop
!
                    end do
!
          end if
          !
end do
!
end program alea5
!
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
!SUBROTINA GERADORA DE NUMEROS ALEATORIOS
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!O PRINCIPIO PARA GERAR NUMEROS ALEATORIOS DESTA SUBROTINA SE BASEIA EM
!SUBSTITUIR OS BITS NA REPRESENTACAO BINARIA DA SEMENTE, UTILIZANDO UMA
!SERIE SE NUMEROS QUE FORAM GERADOS DA MESMA FORMA. PARA ISTO, SAO UTILIZADAS
!UMA SERIE DE FUNCOES QUE MUDAM OS BITS UM A UM DO NUMERO SOB
!UMA DADA CONDICAO PRE ESTABELESCIDA. ABAIXO TEMOS A EXPICACAO DAS FUNCOES
!UTILIZADAS
!
!NEAREST(X,Y) => RETORNA O MAIS PROXIMO NUMERO REPRESENTAVEL AO VALOR DE X,
!                COM O SENTIDDO INDICADO PELO SINAL DE Y. ISTO E, SE Y < 0,
!                RETORNA O PRIMEIRO NUMERO REPRESENTAVEL MENOR DO QUE X.
!                SE Y > 0, RETORNA O PRIMEIRO NUMERO REPRESENTAVEL QUE E
!                MAIOR DO QUE X.
!
!IOR(X,Y)     => REALIZA UM OR INCLUSIVO EM CADA PAR DE BITS DAS STRINGS
!                BINARIAS DOS NUMEROS X E Y, EM SEQUENCIA, GERANDO UMA
!                TERCEIRA STRING BINARIA COMO RESULTADO. UM OR INCLUSIVO
!                RETORNA UM VALOR 1 SE PELO MENOS UM DOS DOIS BITS COMPARADOS
!                FOR 1, E 0 CASO CONTRARIO
!
!IEOR(X,Y)    => REALIZA UM OR EXCLUSIVO EM CADA PAR DE BITS DAS STRINGS
!                BINARIAS DOS NUMEROS X E Y, EM SEQUENCIA, GERANDO UMA
!                TERCEIRA STRING BINARIA COMO RESULTADO. UM OR EXCLUSIVO
!                RETORNA UM VALOR 1 SE HOUVER UM NUMERO IMPAR DE 1'S ENTRE OS
!                DOIS BITS COMPARADOS E RETORNA 0 CASO CONTRARIO
!
!ISHIFT(X,Y)  => RETORNA UM NUMERO CORRESPONDENTE A STRING BINARIA DE X COM
```

```
185 !                   TODOS OS BITS MOVIDOS Y POSICOES. SE Y FOR POSITIVO, OS
        BITS
186 !                   SAO MOVIDOS PARA A ESQUERDA. SE Y FOR NEGATIVO, OS BITS
        SERAO
187 !                   MOVIDOS PARA A DIREITA. SE Y FOR ZERO, NAO HA
    MOVIMENTACAO.
188 !                   NOTE NOTE QUE O VALOR DE Y PRECISA SER DO MESMO TAMANHO
        OU
189 !                   MENOR DO QUE A STRING BINARIA QUE REPRESENTA O VALOR DE
        X.
190 !
191 !IAND(X, Y)    => REALIZA UM AND LOGICO EM CADA PAR DE BITS DAS STRINGS
192 !                   BINARIAS DOS NUMEROS X E Y, EM SEQUENCIA, GERANDO UMA
    TERCEIRA
193 !                   STRING BINARIA COMO RESULTADO. UM AND LOGICO RETORNA UM
194 !                   RESULTADO 1 SOMENTE SE OS DOIS BITS COMPARADOS FOREM 1.
195 !
196 !OBSERVACAO: NAS FUNCOES IOR, IEOR E IAND, OS NUMEROS X E Y DEVEM
    POSSUIR
197 !STRINGS BINARIAS CORRESPONDENTES CUJO COMPRIMENTO DE AMBAS E O MESMO
198 !
199 !
200 SUBROUTINE ran1sub(idum,x)
201 !
202 IMPLICIT NONE
203 !
204 !RETORNA O VALOR DE KIND QUE REPRESENTA O MENOR TIPO INTEIRO QUE
    REPRESENTA
205 !TODOS OS VALORES ENTRE 1E-9 E 1E9
206 INTEGER, PARAMETER :: K4B = selected_int_kind(9)
207
208 !SEMENTE DO NUMERO RANDOMICO, PRECISAO K4B
209 INTEGER(K4B), INTENT(INOUT) :: idum
210
211 !NUMERO RANDOMICO FINAL
212 REAL(kind=8) :: x
213
214 !PARAMETROS DE MANIPULACAO, PRECISAO DADA POR K4B
215 INTEGER(K4B), PARAMETER :: IA=16807, IM=2147483647,IQ=127773,IR=2836
216
217 !VARIAVEL DE CONTROLE
218 REAL(kind=8), SAVE :: am
219
220 !VARIAVEIS DE CONTROLE, PRECISAO DADA POR K4B
221 INTEGER(K4B), SAVE :: ix = -1, iy = -1, k
222 !
223 !
224 if (idum <= 0 .or. iy < 0) then
```

```
225             !am RECEBE O PRIMEIRO NUMERO REPRESENTAVEL MENOR QUE 1.0
226             am = nearest(1.0,-1.0)/IM
227
228             !GERA O PRIMEIRO PARAMETRO INTEIRO PSEUDO-ALEATORIO, BASEADO NO
                    VALOR
229             !DE idum
230             iy = ior(ieor(888889999,abs(idum)),1)
231
232             !SEGUNDO PARAMETRO INTEIRO PREUDO-ALEATORIO, DEPENDENTE DO
                    VALOR
233             !DE idum
234             ix = ieor(777755555,abs(idum))
235
236             !VALOR DE idum SE TORNA POSITIVO E INCREMENTADO
237             idum = abs(idum)+1
238     end if
239     !
240     !ATUALIZA O PARAMETRO ix, GERANDO UM PARAMETRO COM UM 'NIVEL DE
            ALEATORIEDADE'
241     !MAIOR, BASEADO NO VALOR ANTERIOR ARMAZENADO E UTILIZANDO OPERACOES NOS
242     !PROPRIOS BITS
243     ix = ieor(ix,ishft(ix,13))
244
245     !REALIZA UMA SEGUNDA ATUALIZACAO NO PARAMETRO ix, BASEADO NO VALOR
            ACIMA
246     !ESTABELECIDO E REALIZANDO MAIS UMA SERIE DE TROCAS DE BITS NO VALOR
247     ix = ieor(ix,ishft(ix,-17))
248
249     !REALIZA UMA ATUALIZACAO FINAL NO NUMERO ix, COM UM RESULTADO FINAL
250     !PSEUDO-ALEATORIO COM POUCA REALACAO PROXIMAL COM A SEMENTE INICIAL
            idum
251     ix = ieor(ix,ishft(ix,5))
252
253     !GERA UM TERCEIRO PARAMETRO PSEUDO-ALEATORIO, BASEADO NO VALOR DO
            PARAMETRO iy
254     k = iy/IQ
255
256     !ATUALIZA O VALOR DE iy, USANDO COMO PARAMETROS O VALOR JA EXISTENTE EM
                iy E O
257     !PARAMETRO k DEFINIDO ACIMA. ESTA OPERACAO DIMINUI A RELACAO COM O
            VALOR DA
258     !SEMENTE idum
259     iy = IA*(iy-k*IQ)-IR*k
260
261     !TRANSFORMA iy NUM VALOR POSITIVO, CASO SEJA NEGATIVO, EM SOMANDO O
            MAIOR
262     !INTEIRO QUE O SISTEMA CONSEGUE REGISTRAR A ELE, DE MODO QUE TORNA O
            RESULTADO
```

```fortran
!FINAL UM POUCO MAIS ALEATORIO
if (iy < 0) iy = iy + IM

!GERA O NUMERO ALEATORIO REAL AO FINAL, UTILIZANDO OS PARAMETROS
!am, ix, iy E IM.
x = am*ior(iand(IM,ieor(ix,iy)),1)

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
END SUBROUTINE ran1sub
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Capítulo 4

Integração e derivação numérica

4.1 Raízes de funções e aproximações numéricas de funções

Dentre os métodos para encontrar raízes de uma equação utilizaremos o método de *Newton-Raphson* (NR). O método de NR é um método interativo recursivo, isto significa que um resultado depende do resultado anterior e assim sucessivamente. Na medida em que se encontra um resultado o método nos leva para uma raiz, assim a raiz que se deseja encontrar depende do chute inicial e do tipo da equação que se gostaria de estudar. O método NR é dado pela equação 4.1.1.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{4.1.1}$$

A representação gráfica pode ser dada pela figura 4.1.

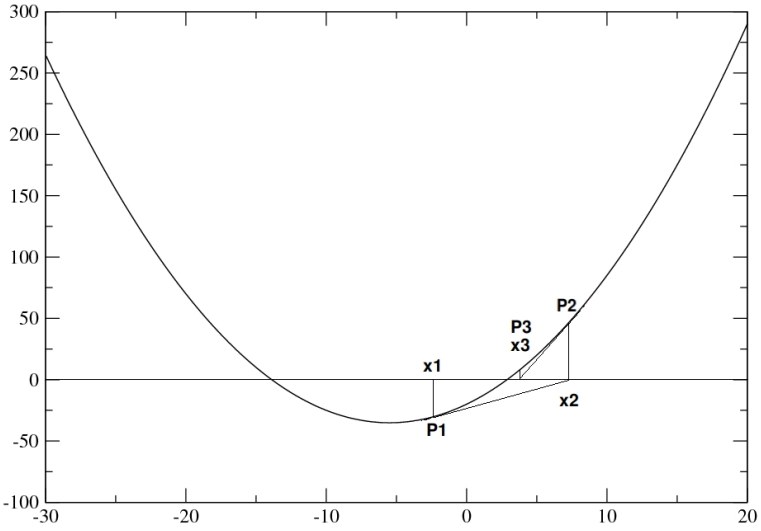


Figura 4.1: Representação gráfica do método NR.

Na figura podemos acompanhar os passos dados pelo método NR. Dado o valor de $i = 1$, temos x_1, $f(x_1)$ e $f'(x_1)$ e assim encontramos o valor de x_2 através da reta

tangente que passa por P_1. Em um segundo passo com o valore de x_2 encontraremos o valor de x_3 e por fim o valor da raiz dado por x_n dentro de um critério de parada. É possível observar que cada passo o valor de x se aproxima da raiz e como podemos observar o chute inicial (o primeiro valor de x é muito importante para o sucesso de se encontra a raiz. Outro detalhe é que para o método de NR é necessário saber a derivada de $f(x_1)$, ou seja, $f'(x_1)$ no ponto de x_1.

A derivada da equação pode ser encontrada basicamente de duas maneiras, uma delas é feita a mão, entende-se analiticamente e a outra forma numericamente. Muitas vezes a equação possui pontos de inflexão e acabam por não ser deriváveis em um ponto específico e até mesmo não sendo analíticas. No entanto um método numérico pode resolver alguns desses problemas, por exemplo o da derivada de um ponto de inflexão. Um desses métodos de derivação numérica em um ponto é o Método das Diferenças Finitas (MDF).

O método é simples e pode ser iniciado da seguinte forma: partindo da definição de derivada e assumindo que entre x e $x + \Delta x$ podemos ter um segmento de reta definimos a derivada à direita de x como:

$$f'(x) = \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{4.1.2}$$

assim a derivada à esquerda de x é dada por:

$$f'(x) = \frac{f(x) - f(x - \Delta x)}{\Delta x} \tag{4.1.3}$$

As equações 4.1.2 e 4.1.3 para $\Delta x \to 0$ leva ao mesmo resultado, no entanto para Δx finito elas levam para aproximações distintas. A forma centradas das equações 4.1.2 e 4.1.3 geralmente é mais utilizada, tem a forma da equação 4.1.4:

$$f'x = \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x} \tag{4.1.4}$$

a forma acima é conhecida como MDF de primeira ordem. Pode ser utilizada também a MDF de segunda ordem como na equação 4.1.5.

$$f''x = \frac{f'(x + \Delta x) - f'(x)}{\Delta x} = \frac{f(x + \Delta x) + f(x - \Delta x) - 2f(x)}{\Delta x^2} \tag{4.1.5}$$

Com a equação 4.1.1 temos o método de NR para encontrar as raízes de uma equação e com a equação 4.1.4 um método para encontrar a derivada de uma equação em um ponto x desejado. Combinando as duas equações temos a forma final para encontrar raízes de uma equação totamlente numérica, dado pela equação 4.1.6.

$$x_{i+1} = x_i - \frac{f(x_i)}{\left(\frac{f(x_i+\Delta x_i)-f(x_i-\Delta x_i)}{2\Delta x_i}\right)} \tag{4.1.6}$$

Atividade: Iremos desenvolver um programa de forma mais genérica possível que encontre as raízes de uma equação utilizando o Método NR juntamente com o método MDF. Utilizaremos as equações:

$$f(x) = x^2 - 5 \tag{4.1.7}$$

$$f(x) = (x^4 - 10x^2)e^{-x} + 1 \tag{4.1.8}$$

Observações

- Como dito anteriormente o método NR é interativo iniciando-se com um dado x_1, encontra-se x_2 e assim por diante. O critério de parada é dado pelo próprio usuário/desenvolvedor fazendo uma verificação entre passos. Quando uma diferença entre passos atingir valores menor que $1,0 \times 10^{-6}$ podemos assumir que o x_n encontrado é um bom valor para a raiz. O valor da precisão é adaptável dependendo do tipo de problema no qual estamos trabalhando;
- Encontre um valor de Δx para o método de MDF que seja coerente com a precisão que está utilizando no método de NR;
- ...

4.2 Integração numérica e transformada de Fourier

Neste tópico estaremos construindo dois programas que calcula uma integral definida entre dois ponto a e b. Inicialmente veremos o cálculo de uma integral numericamente data pela Regra do Trapézio e a seguir o cálculo de uma integral dada pelo método de Monte Carlo. Além desses dois métodos citados temos outros métodos para resolver uma integral numericamente como a Regra de Simpson e também o método da Quadratura de Gauss, no entanto não desenvolveremos pela dificuldade intrínseca e também pela falta de tempo, ficando para os mais curiosos o desenvolvimento extra-classe.

Apesar da existência de outros métodos mais sofisticados, os dois primeiros métodos que iremos desenvolver são razoáveis e produz resultados confiáveis quando utilizados critérios de parada *pequenos o suficiente*. Por fim veremos como é feita uma Transforma de Fourier Discreta.

4.2.1 Regra do Trapézio

Como já sabemos da definição de integral na forma geométrica, o resultado de uma integral definida é a área compreendida abaixo da curva.

Assim da definição temos:

$$I_{ab} = \int_a^b f(x)dx = \lim_{\Delta x_i \to 0} \sum_{i=1}^{N} f(x_i)\Delta x_i \tag{4.2.1}$$

O primeiro termo da direita para a esquerda da equação 4.2.1 representa um somatório de $f(x_i)\Delta x_i$ que são áreas retangulares discretizadas dada pelo largura Δx_i e altura $f(x_i)$. Para um Δx_i muito pequeno conseguimos calcular a área total abaixo de uma curva compreendida entre a e b.

A representação gráfica da equação 4.2.1 é dada pela figura:

A Regra do Trapézio propriamente dita consiste em fazer uma interpolação linear entre os pontos $(x_i,\ f(x_i))$ consecutivos cuja soma fica da seguinte forma.

$$\begin{aligned} I_{ab} = &\ \Delta x \left(\tfrac{1}{2} f(x_1) + f(x_2) + f(x_3) + \cdots + f(x_{n-1}) + f(x_n)\right) \\ I_{ab} = &\ \Delta x \left(\sum_{i=1}^{N} f(x_i) - \tfrac{1}{2}\left(f(x_1) + f(x_N)\right)\right) \end{aligned}$$

Outra forma de calcularmos uma integral numérica é tirarmos uma média entre x_i e x_{i+1} e calcular a altura do retângulo pela média entre pontos utilizando $f((x_{i+1} - x_i)/2)$ e assim multiplicarmos por Δx.

$$
\begin{aligned}
I_{ab} = \quad & f\left(\tfrac{x_2-x_1}{2}\right)(x_2-x_1) + f\left(\tfrac{x_3-x_2}{2}\right)(x_3-x_2) + \cdots + f\left(\tfrac{x_n-x_{n-1}}{2}\right)(x_n-x_{n-1}) \\
I_{ab} = \quad & \sum_{i=1}^{N-1} f\left(\tfrac{x_{i+1}-x_i}{2}\right)(x_{i+1}-x_i) \\
I_{ab} = \quad & \sum_{i=0}^{N-1} f\left(x_1 + i\Delta x + \tfrac{\Delta x}{2}\right)\Delta x \\
I_{ab} = \quad & \sum_{i=0}^{N-1} f\left(x_1 + \tfrac{2i+1}{2}\Delta x\right)\Delta x
\end{aligned}
$$

Para desenvolvermos a nossa atividade utilizaremos as equações 4.2.2 e 4.2.3, em que $a = 0$ e $b = 1$ são os limites de integração para as duas equações.

$$I_{01} = \int_0^1 e^x \, dx = 1,7183, \text{ onde } f(x) = e^x \tag{4.2.2}$$

$$I_{01} = \int_0^1 e^{-x^2} cos(7x) \, dx = 0,0210495, \text{ onde } f(x) = e^{-x^2} cos(7x) \tag{4.2.3}$$

4.2.2 Integração por Monte Carlo

Uma integração de uma equação pelo método de Monte Carlo consiste em analisar uma amosta de uma distribuição de pontos que probabilisticamente está contido na área abaixo da curva da equação de interesse. Pragmaticamente observemos a figura 4.2, temos uma equação quadrática centrada em zero. O que desejamos obter é a integral desta equação nos intervalos x_i e x_f.

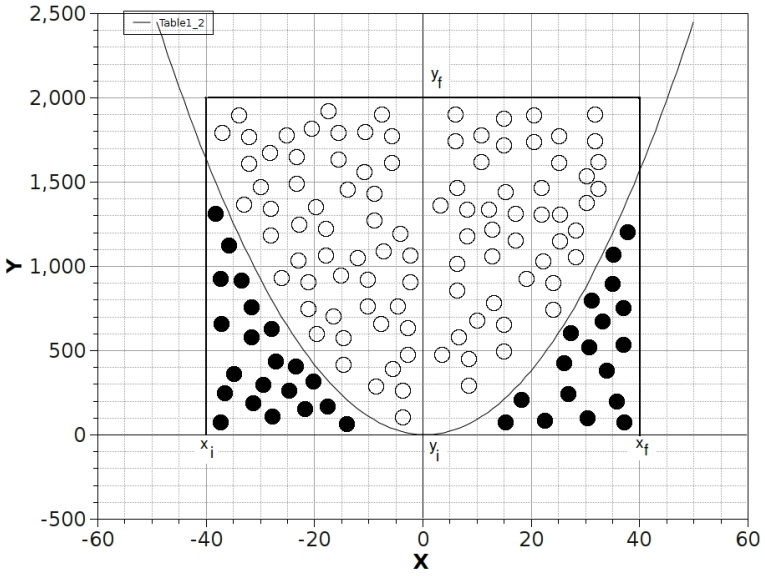


Figura 4.2: Representação gráfica dos sorteios para integração com o método Monte Carlo. Obs.: xi, xf, yi e xf são os limites do quadrado definido e não os valores de f(xi), f(xf), ...

O método de Monte Carlo nos leva à seguinte relação:

$$\frac{\text{área abaixo da curva}}{\text{área do quadrado}} = \frac{I}{(x_f - x_i) * (y_f - y_i)} = \frac{\text{número de acertos}}{\text{tentativas}} \tag{4.2.4}$$

$$\frac{\text{número de acertos}}{\text{tentativas}} = \frac{\text{círculos preenchidos}}{\text{círculos preenchidos + não preenchidos}} \tag{4.2.5}$$

assim

$$I = \frac{\text{número de acertos}}{\text{tentativas}} * (x_f - x_i) * (y_f - y_i) \tag{4.2.6}$$

o que entendemos como *número de acertos* é a quantidade de círculos que foram sorteados abaixo da curva (círculos preenchidos) e as tentativas é a soma dos círculos não preenchidos e preenchidos, ou simplesmente o números total de pontos sorteados.

Cada ponto (x, y) sorteado deve estar contido dentro do quadrado maior onde $x_i \leq x \leq x_f$ e $y_i \leq y \leq y_f$. Para o que chamamos de *número de acertos*, o ponto sorteado além de satisfazer a condição anterior deve satisfazer também a condição de $y \leq f(x)$, este ponto seria representado pelo círculo preenchido. Para todo ponto cujo $y > f(x)$ o ponto estaria sendo representado pelos círculo abertos.

Para aplicar o método de integração de Monte Carlo vamos utilizar a equação 4.2.7.

$$I = \int_0^1 \frac{dx}{1 + x^2} = \frac{\pi}{4} = 0.78540 \tag{4.2.7}$$

Outra possibilidade é estimar o valor de π, considerando um quadrado de lado 2 e dentro desse quadrado uma circunferência de raio igual a 1. Assim considerando a equação 4.2.6 teremos a equação 4.2.8.

$$\frac{A_{circulo}}{A_{quadrado}} = \frac{\pi 1^2}{2 \times 2} = \frac{\pi}{4} \rightarrow \pi = \frac{4 \times N_{acertos}}{N_{tentativas}} \tag{4.2.8}$$

4.2.3 Transformada de Fourier Discreta (TFD)

Antes de iniciarmos a TFD revisaremos alguns itens de matemática e também de Fortran. Revisaremos um pouco de paridade de funções, operações de números complexos e como trabalhar com números complexos com o Fortran 90.

Paridade de uma Função

Uma função é dita par ou ímpar se:

$$\text{par: } f(x) = f(-x) \tag{4.2.9}$$

$$\text{ímpar: } f(x) = -f(-x) \tag{4.2.10}$$

A paridade de uma função define a simetria que é fica melhor representada na figura 4.3a e 4.3b.

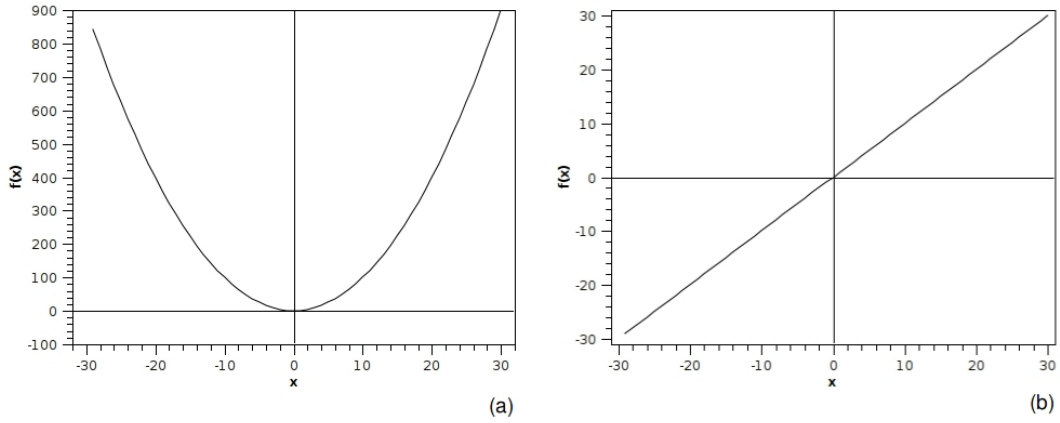


Figura 4.3: Exemplo de uma (a) função par ($f(x) = x^2$) e uma (b) função ímpar ($f(x) = x$).

Essa nomenclatura advem de:

$$f(x) = x^k \begin{cases} \text{será par}, & \text{se k é um número par}, \\ \text{será ímpar}, & \text{se k é um número ímpar}. \end{cases} \tag{4.2.11}$$

Algumas propriedades das funções pares e ímpares:

- Uma função é para e ímpar ao mesmo tempo de $f(x) = 0$;
- Podemos ter uma função sem paridade (?) (gostaria de colocar um exemplo aqui);
- Uma função ímpar definida na origem é nula na origem;
- O resultado da soma de duas funções pares é par;
- O resultado da some de duas funções ímpares é ímpar;
- O resultado da multiplicação de par × par ou ímpar × ímpar é par;
- O resultado da multiplicação de par × ímpar é ímpar;
- O resultado da derivada de uma função par é ímpar;
- O resultado da derivada de uma função ímpar é par;

Números complexos - básico

Os números complexos são elementos do conjunto complexo $\mathbb{C}$ e o conjunto $\mathbb{C}$ está contido no conjunto dos números reais $\mathbb{R}$ [1]. Quando se busca encontrar raízes de uma equação e a raiz apresenta um número $\sqrt{-1}$, esse número é denominado como uma raiz complexa, assim $i = sqsrt{-1}$ e $i^2 = -1$.

Um número complexo é denominado como:

$$K = a + bi \tag{4.2.12}$$

em que K é um números complexo composto pela parte real a e a parte imaginária bi, sendo b um número real.

Algumas operações com números complexos:

- soma: $K + L = (a + bi) + (c + di) = (a + c) + (b + d)i$;
- multiplicação: $K \cdot L = (a + bi) \cdot (c + di) = (ac - bd) + (bc + ad)i$;
- multiplicação: $K \cdot \bar{K} = (a + bi) \cdot (a - bi) = a^2 + b^2$, $\bar{K}$ é o complexo conjugado de K;
- módulo: $|K| = r = \sqrt{a^2 + b^2}$

Na forma polar podemos escrever o número complexo K como:

$$K = r(cos\theta + isen\theta) \tag{4.2.13}$$

Pela igualdade de Euler temos que:

$$e^{\pm i\theta} = cos\theta \pm isen\theta \tag{4.2.14}$$

Assim o número complexo K pode ser escrito na forma:

$$K = re^{\pm i\theta} \tag{4.2.15}$$

Números complexos - no Fortran 90

No Fortran 90 podemos trabalhar com números complexos de uma forma muito parecida como a que trabalhamos no lápis e papel. No programa a seguir temos dois exemplos de números complexos (um vetor complexo e uma variável complexa), como é escrita a parte real e a parte complexa (na verdade são escritas como dois números reais), como atribuir à um número complexo dois números reais, imprimir um número complexo, como utilizar a parte real, imaginária e o módulo do número complexo.

[1] Para relembrar: $\mathbb{Z}$ são números inteiros, $\mathbb{N}$ são números naturais e $\mathbb{Q}$ são números racionais

```fortran
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
! EXEMPLO DA UTILIZACAO DE NUMEROS COMPLEXOS
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> Outubro/2009
program num_complex2
implicit none
! DECLARACAO DE VARIAVEIS >>> INTEIROS
integer :: i, n
! DECLARACAO DE VARIAVEIS >>> REAIS
real(kind=8), parameter :: pi = 3.1415926535897967d0
real(kind=8) :: a, b, theta, soma_a, soma_b, passo
! DECLARACAO DE VARIAVEIS >>> COMPLEXOS
complex(kind=8), allocatable, dimension(:) :: K1
complex(kind=4) :: K2
n = 18
soma_a = 0.d0
soma_b = 0.d0
passo = 0.d0
allocate(K1(n))
write(*,*)'Vetor Complexo K1(i) >>> ( PARTE REAL , PARTE IMAGINARIA )'
do i = 1, n
        if ( i == 1 ) then
                passo = 0.0
        else
                passo = passo + 5.0
        end if
        theta = passo * pi / 180.0
        a = cos(theta)
        b = sin(theta)
        soma_a = soma_a + a
        soma_b = soma_b + b
        ! *** UM VETOR COMPLEXO E ESCRITO COMO DOIS NUMEROS REAIS ***
        !    *** VETOR_COMPLEXO = (PARTE REAL, PARTE IMAGINARIA) ***
        ! escrevendo o numero complexo com os dois numeros reais
        K1(i) = CMPLX(a,b)
        write(*,*)'Vetor Complexo K1(',i,') >>>',K1(i)
end do
soma_a = soma_a / real(n)
soma_b = soma_b / real(n)
! *** UM NUMERO COMPLEXO E ESCRITO COMO DOIS NUMEROS REAIS ***
!    *** NUMERO_COMPLEXO = (PARTE REAL, PARTE IMAGINARIA) ***
K2 = CMPLX(soma_a,soma_b)
write(*,*)'>>>>>>>>>>>>><<<<<<<<<<<<'
write(*,*)'Numero complexo        K2 =',K2
write(*,*)'Parte Real de          K2 =',REAL(K2)
write(*,*)'Parte Imaginaria de    K2 =',AIMAG(K2)
write(*,*)'Complexo Conjugado de K2 =',CONJG(K2)
write(*,*)'Valor absoluto de      K2 =',CABS(K2)
write(*,*)'>>>>>>>>>>>>><<<<<<<<<<<<'
```

```
49 write(*,*) '>>> TERMINOUT <<<'
50 deallocate(K1)
51 !
52 stop
53 end program num_complex2
54 !
```

Transformada de Fourier Discreta - TFD

A partir deste ponto iremos implementar como exercício da linguagem Fortran 90 uma TFD. Os princípios e propriedades de uma Transformada de Fourier não estão descritos aqui nesta apostila, para maiores detalhes sobre a parte matemática é importante que seja consultado o livro da referência [1] e as referências citadas pelo autor. Pode ser consultado também os livros de métodos matemáticos aplicados à Física entre outros bons livros disponíveis. Nesta parte simplesmente nos preocuparemos somente com a implementação e tradução do que é visto teoricamente para a uma linguagem computacional, que é o Fortran 90.

Assumiremos que temos uma função $f(t)$, em que t é uma variável independente e não necessariamente indica o tempo e que a integral 4.2.16 exista.

$$g(\omega) = \int_{-\infty}^{+\infty} e^{-i\omega t} f(t) dt \tag{4.2.16}$$

O resultado da integral dada pela $g(\omega)$ é chamada de Transformada de Fourier de $f(t)$. ω é a variável independente da Transformada de Fourier chamada de frequência angular. Se t é uma variável espacial a variável ω será o número de onda $k = 2\pi/\lambda$, sendo λ o comprimento de onda.

Entre as transformadas existe uma correspondência bi-unívoca entre as funções $f(t)$ e $g(\omega)$ tal que a equação 4.2.17 exista.

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} e^{i\omega t} g(\omega) d\omega \tag{4.2.17}$$

esta equação é chamada de Transformada de Fourier inversa.

Como vimos na seção de integração numérica, podemos resolver a integral que contém $f(t)$ e obter a Transformada de Fourier $g(\omega)$. Neste caso como estamos discretizando o problema da integral para obter a Transformada de Fourier, dizemos que esta Transformada de Fourier numérica é a TFD dentro de um intervalo qualquer $[a, b]$ em que a $f(t)$ é integrável. Assim a equação 4.2.16 toma a forma da equação 4.2.18.

$$g(\omega) = \Delta t \sum_{j=1}^{n} e^{-i\omega_k t_j} f(t_j) \tag{4.2.18}$$

Ao observarem a TFD (ou qualquer outra transformada de fourier) verão que teremos uma dependência de duas variáveis, uma em t_j e outra em ω_k que estão uma no espaço real e outra no espaço dos complexos, respectivamente. A idéia da transformada de fourier é manter a bi-unicidade entre a $f(t_j)$ e $g(\omega_k)$ e para isso a quantidade t_j deve ser igual a quantidade de ω_k, ou vice-verso. Essa igualdade de pontos nos diz que: se entre t_i e t_f temos 100 pontos, então entre ω_i e ω_f também teremos 100 pontos ou intervalos.

Vamos pegar como exemplo $t_i = -30$ e $t_f = 30$, se $\Delta t = 0.6$ então teremos 100 intervalos entre t_i e t_f. Como ω varia sempre de $-m\pi$ até $+m\pi$, então $\Delta\omega = 2m\pi/100$, onde m é um número inteiro. Deve-se sempre explorar os valores de Δt e $\Delta\omega$ para que tenhamos sempre um conjunto de pontos que deixe as curvas $f(t_j)$ e $g(\omega_k)$ suave.

Uma vez definido os intervalos, vemos que é fácil perceber que a TFD é uma multiplicação de matrizes em que $g(\omega_k)$ é uma matriz coluna, o somatório da exponencial é uma matriz bi-dimensional e a $f(t_j)$ é uma matriz coluna, assim definimos a matriz W_{kj} pela expressão 4.2.19.

$$W_{kj} = e^{-i\omega_k t_j} \tag{4.2.19}$$

assim a equação 4.2.18 assume a forma matricial:

$$g = W * f \tag{4.2.20}$$

ou

$$\begin{bmatrix} g(\omega_1) \\ g(\omega_2) \\ \vdots \\ g(\omega_k) \end{bmatrix} = \begin{bmatrix} e^{-i\omega_1 t_1} & e^{-i\omega_1 t_2} & \dots & e^{-i\omega_1 t_j} \\ e^{-i\omega_2 t_1} & e^{-i\omega_2 t_2} & \dots & e^{-i\omega_2 t_j} \\ \vdots & \vdots & \ddots & \vdots \\ e^{-i\omega_k t_1} & e^{-i\omega_k t_2} & \dots & e^{-i\omega_k t_j} \end{bmatrix} \times \begin{bmatrix} f(t_1) \\ f(t_2) \\ \ddots \\ f(t_j) \end{bmatrix} \tag{4.2.21}$$

A matriz representada pela equação 4.2.21 possui elementos que são números complexos, assim pela igualdade de *Euler* podemos escrever a exponencial conforme a equação 4.2.22.

$$e^{-i\omega_k t_j} = cos(\omega_k t_j) - isen(\omega_k t_j) \tag{4.2.22}$$

a equação 4.2.18 e 4.2.21 pode ser escrita na forma da equação 4.2.23

$$g(\omega) = \Delta t \sum_{j=1}^{n} \left[cos(\omega_k t_j) - isen(\omega_k t_j)\right] f(t_j) \tag{4.2.23}$$

sendo a parte real (eq. 4.2.24) e imaginária (eq. 4.2.25) dada por:

$$Real\left(g(\omega)\right) = \Delta t \sum_{j=1}^{n} cos(\omega_k t_j) f(t_j) \tag{4.2.24}$$

$$Imaginario\,(g(\omega)) = \Delta t \sum_{j=1}^{n} sen(\omega_k t_j) f(t_j) \tag{4.2.25}$$

A nossa atividade será calcular a TFD das equações 4.2.26 e 4.2.27.

$$f(t) = e^{-\frac{(t-t_0)^2}{2\sigma^2}} \tag{4.2.26}$$

A equação 4.2.26 é a de uma gaussiana em que t_0 é o centro da gaussiana e σ é a largura da gaussiana. A parte muito interessante aqui é alterar os centros e principalmente as larguras das gaussianas para ver o comportamento da sua TFD. Utilize os intervalos $-30 \leq t \leq 30$ e $-\pi \leq \omega \leq \pi$.

$$f(t) = sen(10t) \cdot e^{-t} \tag{4.2.27}$$

A equação 4.2.27 é a de um oscilador harmônico amortecido, o intervalo no espaço real é $0 \leq t \leq 10$ e espaço complexo $-25\pi \leq \omega \leq 25\pi$.

Atenção: para as duas equações encontre a TFD e faça 4 gráficos com xmgrace, sendo os gráficos da $f(t)$, valor absoluto da $g(\omega)$, da parte real de $g(\omega)$ e da parte imaginária de $g(\omega)$.

Além da TFD apresentada acima, na literatura temos um outro métodos de calcular a TFD de uma maneira mais rápida e eficiente que é conhecida como a *Fast Fourier Transform - FFT*. Um dos algoritmos mais conhecidos é o do *Cooley-Tukey*[2] [9]. Para aprender mais sobre a FFT é bom verificar a referência [1] e também a mais completa/gratuita/on-line a referência [10].

A idéia desta FFT é calcular a TFD mais rápida manipulando os índices do somatório saindo de $k, j = 1, \ldots, n$ e indo para $k, (2j, 2j+1) = 1, \ldots, n/2$. A equação da TFD (eq. 4.2.18) assume a forma da equação 4.2.28.

$$g(\omega_k) = \sum_{j=0}^{(n/2)-1} \left[e^{-\frac{2\pi i}{n}}\right]^{k(2j)} f(t_j) + \sum_{j=0}^{(n/2)-1} \left[e^{-\frac{2\pi i}{n}}\right]^{k(2j+1)} f(t_{2j+1}) \tag{4.2.28}$$

Além de podermos implementar este código de FFT, podemos também utilizar uma sub-rotina já existente de FFT (que no fundo realiza a TFD) já otimizada e

[2]Um overview sobre o método Cooley-Tukey http://en.wikipedia.org/wiki/Cooley-Tukey_FFT_algorithm.

bem testada pelo usuários. Dentro do desenvolvimento computacional chamamos estas sub-rotinas de **bibliotecas matemáticas (ou no inglês *math library*)**. As bibliotecas de FFT podem ser pegas para livre uso no site *http://www.fftw.org*[3]. Para ilustrar a utilização da biblioteca FFTW3 vejamos o programa a seguir:

```
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
! COMO COMPILAR: gfortran -o fftw3f90.x fftw3f90.f90 -lfftw3
!
! NA LINHA: include "/usr/include/fftw3.f" DEVE SER ALTERADO O CAMINHO
! ONDE SE ENCONTRA A BIBLIOTECA/SUBROTINA fftw3.f
!
! ESTE PROGRAMA FOI ADAPTADO DA VERSAO FORTRAN 77 DO SITE
! http://www.psc.edu/general/software/packages/fftw/examples/index.php
! CUJA LICENSA E GERIDA PELA GNU General Public License
!
! ATENCAO: VARIAVEIS REAIS EM KIND=8 E ALGUMAS INTEIRAS COM KIND=8 (8
    BITS)
!
! PROGRAMADOR: Fernando Sato
! E_MAIL: fsato01@gmail.com
! DATA: Outubro/2009
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
program fftw3f90
implicit none

! INCLUINDO O ARQUIVO QUE CONTEM AS DECLARACOES DA FFTW3
include "/usr/include/fftw3.f"

! DECLARACAO DE VARIAVEIS
! UTILIZAR >>> n2 = n/2 + 1
integer, parameter :: n = 60, n2 = 31
real(kind=8), parameter :: pi = 3.14159265358979 67d0
real(kind=8), dimension(n) :: a, b
real(kind=8) :: sigma, a0, abslt, pr, pim
integer(kind=8) :: plan_forward, plan_backward
real(kind=8), dimension(n)      :: in, out2
complex(kind=8), dimension(n2) :: out
integer :: i

! ARQUIVOS DE SAIDA
open(UNIT=30,FILE="fftw_fx_out.dat",STATUS='REPLACE',ACTION='WRITE')
open(UNIT=31,FILE="fftw_tfd_out.dat",STATUS='REPLACE',ACTION='WRITE')
```

[3]As bibliotecas **FFTW** é um Software Livre de caráter técnico definido de acordo com a Free Software Foundation e é distribuído sobre os termos da *GNU General Public License*. Assim como a **FFTW** temos também outras bibliotecas matemáticas como a **LAPACK** (bibliotecas de álgebra linear), **BLAS** (bibliotecas de álgebra linear básica), **ATLAS** (bibliotecas de álgebra linear que inclui a BLAS), **GSL** (o nome se refere à *GNU Scientific Library* e possui uma coleção de sub-rotinas para análise numérica)

```
37 open(UNIT=32,FILE="fftw_ttfd_out.dat",STATUS='REPLACE',ACTION='WRITE')
38
39 ! CONTANTES DA GAUSSIANA
40 sigma = 10.d0
41 a0 = 0.d0
42
43 !>>>>>>>>>>>>>>>>>>> ATRIBUINDO VALORES
44 do i = 1, n
45
46       ! ENTRADA X
47       a(i) = dfloat(i) - dfloat(n2)
48
49       ! VALOR DA F(X)
50       !b(i) = exp( - (a(i)-a0)**2.d0  / (2.d0*sigma*sigma) )
51       b(i) = exp(-a(i)**2.d0)
52
53       ! ESCREVENDO X E F(X)
54       write(30,500) a(i),b(i)
55
56       ! ENTRADA P/ FFT
57       in(i) = b(i)
58 end do
59
60 ! FAZENDO TRANSFORMADA CALCULANDO g(w)
61 !>>>>>>>>>>>>>>>>>>> PASSANDO VARIAVEIS P/ FFT IN:REAL OUT:COMPLEX
62 call dfftw_plan_dft_r2c_1d ( plan_forward, n, in, out, FFTW_ESTIMATE )
63
64 !>>>>>>>>>>>>>>>>>>> EXECUTANDO FFT
65 call dfftw_execute( plan_forward )
66
67 !>>>>>>>>>>>>>>>>>>> SAIDA DA FFT OUT:COMPLEX
68 do i=n2,1,-1
69       pr = REAL(out(i))
70       pim = AIMAG(out(i))
71       abslt = DSQRT(pr*pr + pim*pim)
72       write(31,501) -i, pr/dfloat(n2), pim/dfloat(n2), abslt/dfloat(n2)
73 end do
74 do i = 1,n2
75       pr = REAL(out(i))
76       pim = AIMAG(out(i))
77       abslt = DSQRT(pr*pr + pim*pim)
78       write(31,501) i, pr/dfloat(n2), pim/dfloat(n2), abslt/dfloat(n2)
79 end do
80 !>>>>>>>>>>>>>>>>>>> ENCERRA A TRANSFORMADA DE FOURIER
81 call dfftw_destroy_plan ( plan_forward )
82
83 ! FAZENDO TRANSFORMADA DA TRANSFORMADA CALCULANDO f(t)
84 !>>>>>>>>>>>>>>>>>>> PASSANDO VARIAVEIS P/ FFT IN:COMPLEX OUT:REAL
```

```
85 call dfftw_plan_dft_c2r_1d ( plan_backward, n, out, out2, FFTW_ESTIMATE
       )
86
87 !>>>>>>>>>>>>>>>>>>> EXECUTANDO A TRANSFORMADA DA TRANSFORMADA DE
       FOURIER
88 call dfftw_execute( plan_backward )
89
90 !>>>>>>>>>>>>>>>>>>> SAIDA: TRANFORMADA DA TRANSFORMADA DE FOURIER
91 do i = 1,n
92       write(32,502) i, out2(i)/dfloat(n)
93 end do
94 !>>>>>>>>>>>>>>>>>>> ENCERRA A TRANSFORMADA DE FOURIER
95 call dfftw_destroy_plan ( plan_backward )
96
97 close(30)
98 close(31)
99 close(32)
100
101 !>>>>>>>>>>>>>>>>>>> FORMATOS DE ESCRITA
102 500 format(f18.8,e18.8)
103 501 format(i6,1x,e18.8,e18.8,e18.8)
104 502 format(i6,1x,e18.8)
105 !
106 stop
107 end program fftw3f90
```

Somente lembrando que este código é um exemplo de aplicação das bibliotecas FFTW [11] e não há nenhuma garantia de que este código dará resultados em ambientes de pesquisa ou em qualquer outro ambiente, de qualquer forma o autor não se responsabiliza pelos efeitos colaterais advindos do código acima.

Capítulo 5

Equações diferenciais ordinárias

5.1 Método de Euler

O método de Euler consiste em resolver uma equação diferencial ordinária. Em geral podemos dizer que foi o primeiro método numérico e também que é uma série de Taylor truncada na primeira derivada.

Consideramos que a equação abaixo:

$$\frac{dx}{dt} = f(x,t) \tag{5.1.1}$$

em que x e t são variáveis dependentes e independentes, respectivamente. A $f(x,t)$ é em geral uma função das variáveis dependentes e independentes. A equação 5.1.1 pode ser escrita na forma de limites como na equação 5.1.7.

$$\frac{dx}{dt} = \lim_{\Delta t \to 0} \frac{\Delta x}{\Delta t} \tag{5.1.2}$$

assim para um infinitesimal de t discretizamos a equação 5.1.1 na forma da equação 5.1.7, sendo que a equação 5.1.7 pode ser escrita da forma da equação 5.1.3:

$$\lim_{\Delta t \to 0} \frac{\Delta x}{\Delta t} = \lim_{\Delta t \to 0} \frac{x_{n+1} - x_n}{\Delta t} \tag{5.1.3}$$

combinando as equações 5.1.1 e 5.1.3 teremos:

$$\frac{x_{n+1} - x_n}{\Delta t} = f(x_n, t_n) \tag{5.1.4}$$

ou simplemente:

$$x_{n+1} = x_n + f(x_n, t_n)\Delta t \tag{5.1.5}$$

assim com a equação 5.1.5, dado um conjunto de pontos $f(x_0, t_0)$ é possível obter os valores de $x_1, x_2, \ldots, x_k$, onde $0 \leq k \leq n+1$.

A forma da equação 5.1.5 é muito útil quando conhecemos e não conhecemos a forma analítica de equações do tipo da equação 5.1.1.

Atividade #1

Dada a equação abaixo:

$$y(t) = y_0 + v_0 t + \frac{1}{2} a t^2 \tag{5.1.6}$$

teremos a função será:

$$\frac{dy(t)}{dt} = f(y_n, t_n) = v_0 + at \tag{5.1.7}$$

este movimento pode ser assumido como um lançamento de um projétil com condições iniciais $y_0 = 0$, $|\vec{g}| = 9,8 m/s^2$, $\theta = 30°$ e $v_0 = 25 m/s$, calcule a trajetória da partícula experimentando vários valores de Δt (4 valores está bom). Expresse o seu resultado em um gráfico.

Atividade #2

$$\frac{dy}{dx} = f(y, x) = -xy; y(0) = 1 \tag{5.1.8}$$

Observe que neste caso temos uma condição inicial, para $x = 0$, $y(x) = y(0) = 1$. A partir da condição inicial, o valor de x evolui em passo de δx e y evolui conforme a função data. Para valores entre $0 \leq x \leq 3$, utilize diferentes valores de delta e compare com a solução exata que é $y = exp(-x^2/2)$.

5.2 Método de Runge-Kutta

Ao verificar os diversos tipos de métodos numéricos para resolver equações diferenciais ordinárias verá que existe um pouco de liberdade para desenvolver os algoritmos. De fato, várias delas existem, cada um tendo pontos positivos e negativos, dependendo da aplicação. Um desses métodos bem conhecidos e amplamente utilizados, dentro da classe de métodos de equações diferencias, são os algoritmos de Runge-Kutta [2], que são apresentados em diferentes ordens de precisão. Desenvolveremos a versão de segunda ordem para dar a idéia da aproximação e então apresentaremos as equações para terceira e quarta ordem.

Para obter o algoritmo de Runge-Kutta de segunda ordem, podemos aproximar f da integral da equação 5.2.1 por uma expansão em série de Taylor sobre o ponto médio do intervalo de integração. Assim obteremos a equação 5.2.2, que é a forma simples de Euler (eq. 5.1.5).

$$x_{n+1} = x_n + \int_{t_n}^{t_{n+1}} f(t, x) dt \tag{5.2.1}$$

$$x_{n+1} = x_n + \Delta t \; f(t_{n+1/2}, x_{n+1/2}) + \mathscr{O}(\Delta t^3) \tag{5.2.2}$$

Embora pareça como se nós precisassemos conhecer o valor de $x_{n+1/2}$ que aparece na f da equação 5.2.2, isso não é verdade. Uma vez que o termo do erro já é da ordem de $\mathscr{O}(\Delta t^3)$, uma aproximação para x_{n+1} cujo erro é da ordem de $\mathscr{O}(\Delta t^2)$, já é bom o suficiente. Este é apenas o erro que é fornecido pelo método de Euler simples. Assim, se definirmos k como sendo uma aproximação intermediária ao dobro da diferença entre $x_{n+1/2}$ e x_n, o procedimento dos dois passos seguintes nos dará x_{n+1} em termos de x_n. Assim,

$$k = \Delta t f(t_n, x_n) \tag{5.2.3}$$

$$x_{n+1} = x_n + \Delta t f(t_n + \frac{1}{2}\Delta t, x_n \frac{1}{2}k) + \mathscr{O}(\Delta t^3) \tag{5.2.4}$$

A equação 5.2.4 é o algoritmo de Runge-Kutta de segunda ordem. Esta equação engloba a idéia geral de substituição de aproximações para os valores de x no lado direito das expressões implícitas envolvendo f. São como as acuradas séries de Taylor ou outros métodos implícitos, mas não implica em restrições especiais de f, como fácil diferenciabilidade ou linearidade em x. A equação 5.2.4 exige o cálculo da f duas vezes para cada passo ao longo da evolução.

Os esquemas de Runge-Kutta de altas ordens podem ser obtidos de uma forma relativamente simples utilizando-se alguma de integração por quadratura, não discutidas aqui, mas facilmente obtidas nas referências da apostila. De qualquer forma, qualquer um dos tipos de quadraturas podem ser utilizados para aproximar a equação 5.2.1 através de somas finitas dos valores de f. Assim

$$x_{n+1} = x_n + \frac{\Delta t}{6}\left(f(t_n, x_n) + 4f(t_{n+1/2}, x_{n+1/2}) + f(t_{n+1}, x_{n+1})\right) + \mathscr{O}(\Delta t^5) \tag{5.2.5}$$

O esquema para gerar aproximações sucessivas para x aparecem do lado direito da equação. O algoritmo de terceira ordem com erro local de $\mathscr{O}(\Delta t^2)$ é:

$$k_1 = \Delta t\ f(t_n, x_n)$$

$$k_2 = \Delta t\ f(t_n + \frac{1}{2}\Delta t, x_n + \frac{1}{2}k_1)$$

$$k_3 = \Delta t\ f(t_n + \Delta t, x_n - k_1 + 2k_2)$$

$$x_{n+1} = x_n + \frac{1}{6}(k_1 + k_2 + k_3) + \mathscr{O}(\Delta t^4) \tag{5.2.6}$$

A equação 5.2.6 baseia-se na equação 5.2.5 e necessita calcular f três vezes por passo. Neste esquema, o algoritmo de quarta ordem exigirá que f seja avaliada

quatro vezes em cada passo de integração e possuirá uma precisão local de $\mathscr{O}(\Delta t^5)$, assim temos o algoritmo de Runge-Kutta de quarta ordem:

$$k_1 = \Delta t\ f(t_n, x_n)$$

$$k_2 = \Delta t\ f(t_n + \frac{1}{2}\Delta t, x_n + \frac{1}{2}k_1)$$

$$k_3 = \Delta t\ f(t_n + \frac{1}{2}\Delta t, x_n + \frac{1}{2}k_2)$$

$$k_4 = \Delta t\ f(t_n + \Delta t, x_n + k_3)$$

$$x_{n+1} = x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) + \mathscr{O}(\Delta t^5) \tag{5.2.7}$$

Utilizando os resultados do método de Euler, compare os resultados das atividades #1 e #2 utilizando o método de Runge-Kutta de quarta ordem.

5.3 Método de Verlet

Um outro método de resolver equações diferencias é o Método de Verlet [12]. Antes de entrar no método desenvolvido por Verlet veremos o início considerando as equações de movimento de Newton visto pelo método de Euler. Assim escreveremos duas equações diferenciais acopladas de primeira ordem[1]:

$$\frac{dv}{dt} = a(t) \tag{5.3.1}$$

e

$$\frac{dx}{dt} = v(t) \tag{5.3.2}$$

em que $a(t) = a(x(t), v(t), t)$. Tomamos a quantidade Δt como intervalos entre passos sucessivos e a_n, v_n e x_n como valores da aceleração, velocidade e posição no tempo $t_n = t_0 + n\Delta t$. Utilizando o método de diferenças finitas os valores de x_{n+1} e v_{n+1} para o tempo $t_{n+1} = t_n + \Delta t$. Dentro de um sistema conservativo, tomamos Δt pequeno o suficiente para que não haja grandes flutuações e assim garantimos a estabilidade da energia total do sistema. A expansão em série de Taylor das quantidades $v_{n+1} = v(t + \Delta t)$ e $x_{n+1} = x(t_n + \Delta t)$ nos leva a:

$$x_{n+1} = x_n + v_n\Delta t + \frac{1}{2}a_n\Delta t^2 + \mathscr{O}(\Delta t^2) \tag{5.3.3}$$

$$v_{n+1} = v_n + a_n\Delta t + \mathscr{O}(\Delta t^3) \tag{5.3.4}$$

[1]Para simplificar escrevemos somente um uma dimensão.

assim expandindo agora a para x_{n-1} teremos:

$$x_{n-1} = x_n - v_n\Delta t + \frac{1}{2}a_n\Delta t^2 + \mathscr{O}(\Delta t^2) \tag{5.3.5}$$

somando as equações 5.3.3 e 5.3.5 teremos:

$$x_{n+1} + x_{n-1} = 2x_n + a_n\Delta t^2 + \mathscr{O}(\Delta t^4) \tag{5.3.6}$$

ou simplesmente:

$$x_{n+1} = 2x_n - x_{n-1} + a_n\Delta t^2 \tag{5.3.7}$$

e a subtração entre x_{n+1} (eq. 5.3.3) e x_{n-1} (eq. 5.3.5) nos dá:

$$v_n = \frac{x_{n+1} - x_{n-1}}{2\Delta t} \tag{5.3.8}$$

que é a forma original do algoritmo de Verlet. A forma completa é dada pelas equações a seguir:

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \vec{v}(t)\Delta t + (1/2)\vec{a}(t)\Delta t^2 + (1/6)\frac{d^3\vec{r}(t)}{dt^3}\Delta t^3 + O(\Delta t^4) \tag{5.3.9}$$

$$\vec{r}(t - \Delta t) = \vec{r}(t) - \vec{v}(t)\Delta t + (1/2)\vec{a}(t)\Delta t^2 - (1/6)\frac{d^3\vec{r}(t)}{dt^3}\Delta t^3 + O(\Delta t^4) \tag{5.3.10}$$

$$\vec{r}(t + \Delta t) = 2\vec{r}(t) - \vec{r}(t - \Delta t) + \vec{a}(t)\Delta t^2 + O(\Delta t^4) \tag{5.3.11}$$

Como sabemos as equações 5.3.9 e 5.3.10 são as expansões em série de Taylor até terceira ordem para a expressão da posição [12,13] considerando o avanço e o retorno temporal, respectivamente. O método de Verlet é representado pela equação 5.3.11, que é a soma das equações 5.3.9 e 5.3.10 e possibilita o processo de reversibilidade no tempo. Esse método de Verlet apresenta um problema quando necessitamos dos valores da velocidade, pois, como exposto nas equações 5.3.9, 5.3.10 e 5.3.11, essa primeira versão não apresenta as velocidades. Para estimar as velocidades a partir dessa versão do método de Verlet podemos utilizar a equações 5.3.12 e 5.3.13, para Δt e $\Delta t/2$, repectivamente.

$$\vec{v}(t) = \frac{\vec{r}(t + \Delta t) - \vec{r}(t - \Delta t)}{2\Delta t} \tag{5.3.12}$$

$$\vec{v}(t + \frac{1}{2}\Delta t) = \frac{\vec{r}(t + \Delta t) - \vec{r}(t)}{\Delta t} \tag{5.3.13}$$

Melhorias no algoritmo de Verlet foram feitas para se obter as velocidades, como no caso do algoritmo *leap-frog Verlet*. Porem a implementação mais sofisticada dos métodos de Verlet é o esquema *Velocity Verlet* [14], em que temos a derivação das posições e velocidades no tempo $t+\Delta t$ a partir do tempo t, de acordo com as equações 5.3.14.

$$
\begin{aligned}
\vec{r}(t+\Delta t) &= \vec{r}(t) + \vec{v}(t)\Delta(t) + (1/2)\vec{a}(t)\Delta t^2 \\
\vec{v}(t+\Delta t/2) &= \vec{v}(t) + (1/2)\vec{a}(t)\Delta t \\
\vec{a}(t+\Delta t) &= -(1/m)\Delta\vec{U}(\vec{r}(t+\Delta t)) \\
\vec{v}(t+\Delta t) &= \vec{v}(t+\Delta t/2) + (1/2)\vec{a}(t+\Delta t)\Delta t
\end{aligned}
\tag{5.3.14}
$$

Os algoritmos de Verlet são os mais rápidos para implementação numérica, porém como vimos as expressões são obtidas a partir de expansões e os erros associados para as posições e velocidades são da ordem de Δt^4 e Δt^2, respectivamente, relacionados com os termos mais altos da expansão não incluídos nas equações.

5.4 Atividade #3

Essas duas equações de primeira ordem acopladas

$$\frac{dy}{dt} = p; \frac{dp}{dt} = -4\pi^2 y \tag{5.4.1}$$

$$\text{Lembrando que: } \frac{d^2y}{dt^2} = \frac{d}{dt}\left(\frac{dy}{dt}\right) = \frac{dp}{dt} = -4\pi^2 y$$

define um movimento harmônico simples de período 1. Pela generalização de uma das variáveis das fórmulas acima para esse caso de duas variáveis, integre essas equações com qualquer condição particular inicial de sua escolha (desde que faça sentido, obviamente) e verifique com qual acurácia o sistema retorna para o seu estado inicial para os valores de t. Utilize os modos de Euler e Runge-Kutta e compare os métodos.

Dica

Pelo método de Euler temos:

$$f(t,p) = -4\pi^2 y \Longrightarrow p_{n+1} = p_n + \Delta t\ f(t,p) \tag{5.4.2}$$

encontrando os valores de p podemos calcular os valores de y

$$f(t,y) = p \Longrightarrow y_{n+1} = y_n + \Delta t\ f(t,y) \tag{5.4.3}$$

Agora, utilizando esta idéia, implemente o mesmo sistema com o algoritmo de Runge-Kutta. Deveremos obter gráficos semelhantes aos gráficos das figuras 5.1, 5.2 e 5.3.

Parte do código para o cálculo da eq. 5.4 pelo método de Euler. Lembrando que aqui utilizo vetores. Não é necessário utilizar vetores!!!

```
1 ! EULER
2 do i = 1, n-1
3   t(i+1) = t(i) + dt1
4   p(i+1) = p(i) + dt1 * ( -4.d0 * pi**2 * y(i) )
5   y(i+1) = y(i) + dt1 * p(i)
6 end do
```

Parte do código para o cálculo da eq. 5.4.1 pelo método de Runge-Kutta de quarta ordem. Lembrando que aqui está utilizo vetores. Não é necessário utilizar vetores!!!

```
1 ! RUNGE-KUTTA 4 ORDEM
2 do i = 1, n-1
3   ! CALCULANDO PONTOS Pn
4   ! TERMOS DO RK4
```

```
5    kp1 = dt1 * ( -4.d0 * pi * pi * rk4y(i) )
6    kp2 = dt1 * ( -4.d0 * pi * pi * ( rk4y(i) + 0.5d0 * kp1 ) )
7    kp3 = dt1 * ( -4.d0 * pi * pi * ( rk4y(i) + 0.5d0 * kp2 ) )
8    kp3 = dt1 * ( -4.d0 * pi * pi * ( rk4y(i) + kp3 ) )
9    ! y(n+1) = y(n) + 1.0/6.0 (k1+2*k2+2*k3+k4)
10   ! r6 = 1.d0/6.d0
11   rk4p(i+1) = rk4p(i) + r6 * (kp1 + 2.d0*kp2 + 2.d0*kp3 + kp4)
12
13   ! CALCULANDO PONTOS Yn
14   ky1 = dt1 * ( rk4p(i) )
15   ky2 = dt1 * ( rk4p(i) + 0.5d0 * ky1 )
16   ky3 = dt1 * ( rk4p(i) + 0.5d0 * ky2 )
17   ky4 = dt1 * ( rk4p(i) + ky3 )
18   ! y(n+1) = y(n) + 1.0/6.0 (k1+2*k2+2*k3+k4)
19   ! r6 = 1.d0/6.d0
20   rk4y(i+1) = rk4y(i) + r6 * (ky1 + 2.d0*ky2 + 2.d0ky3 + ky4)
21 end do
```

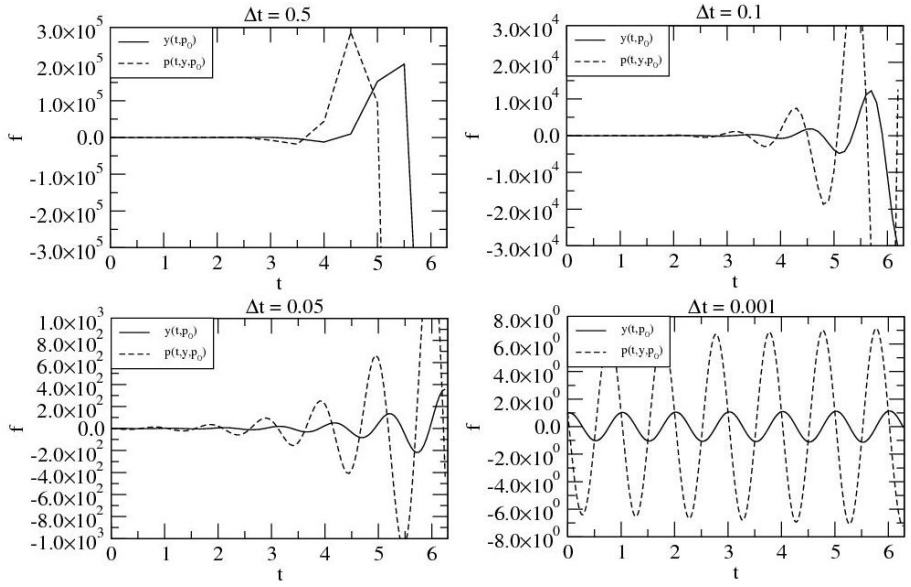


Figura 5.1: Integração das equações 5.4.2 e 5.4.3 para intervalos de tempo iguais a 0.5, 0.1, 0.05 e 0.001

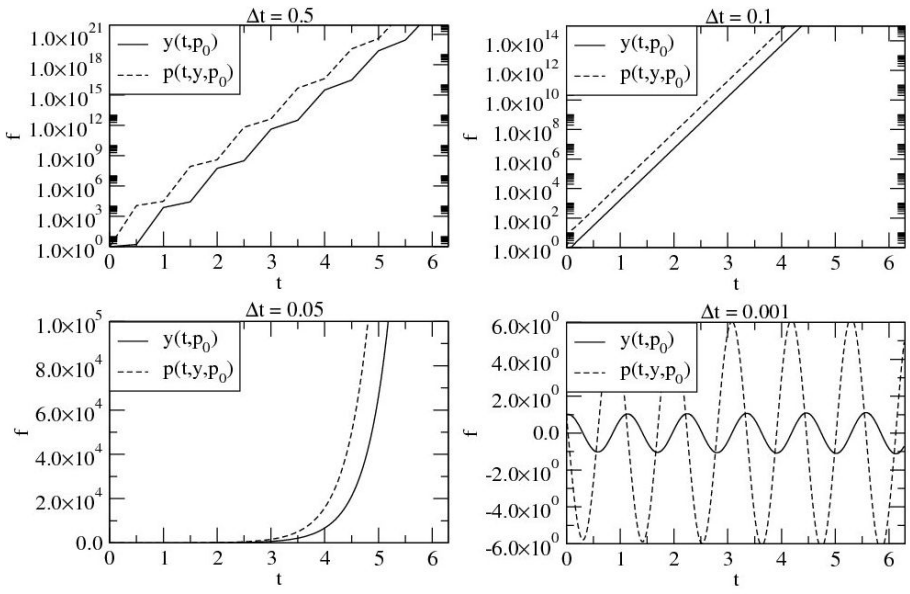


Figura 5.2: Integração da equação 5.4.1 utilizando o método de Runge-Kutta para intervalos de tempo iguais a 0.5, 0.1, 0.05 e 0.001.

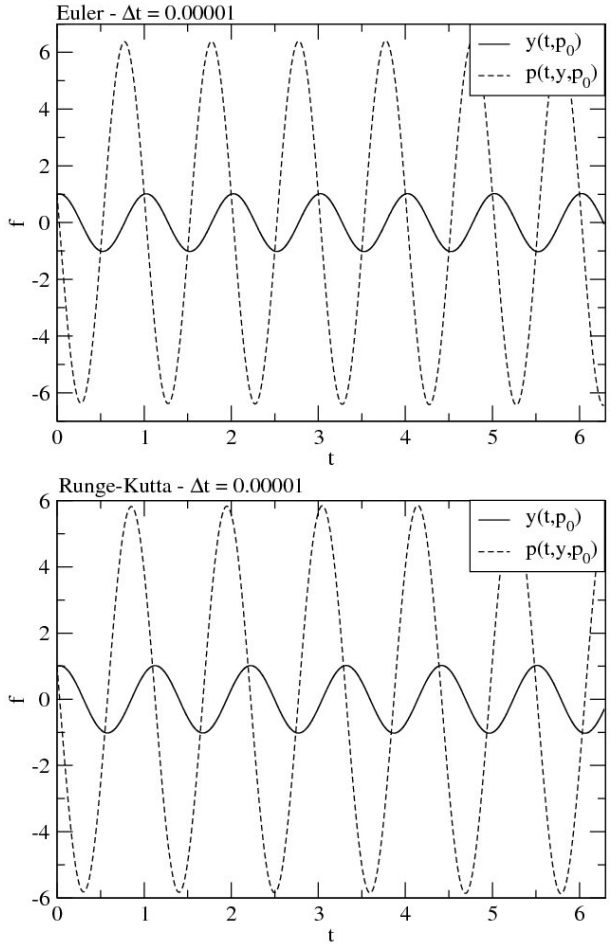


Figura 5.3: Integração utilizando os métodos Euler e Runge-Kutta para o $\Delta t = 1.0 \times 10^{-5}$.

Capítulo 6

Noções básicas de Dinâmica Molecular Clássica

Antes de inicarmos o estudo com a Dinâmica Molecular (DM) veremos o conceito de Mecânica Molecular (MM)[1].

Mecânica [15] Definição: *Estudo das interações entre a matéria e as forças que agem nela. Estática, está amplamente relacionada com a ação das forças quando não há nenhuma mudança do momento, enquanto que a dinâmica trata dos casos em que ocorrem um mudança do momento. Cinemática é o estudo do movimento dos corpos sem referência às forças que afetam o movimento. Ciências clássicas estão relacionadas com os corpos macroscópicos no estado sólido, enquanto que a mecânica dos fluidos é a ciência das interações entre forças e líquidos.*

Baseado na definição de mecânica, no ponto de vista da física, atribuimos a MM como um tratamento estático de otimização das posições de um sistema de átomos ou moléculas, em que as forças são geradas pelos potenciais atômicos[2]. A MM [16] tem um papel muito importante na busca da geometria molecular de sistemas com muitos átomos, por sua simplicidade comparada aos métodos quânticos. Como característica do método clássico de MM não temos a informação da parte eletrônica como no método quântico. Uma referência no uso da MM são os sistemas biológicos de proteínas [17], atualmente um dos limites na simulação atomística de sistemas orgânicos, podendo envolver centenas de milhares (ou milhões) de átomos.

A descrição mais simples do método de MM é considerar a aproximação de Bohr & Oppenheimer. Esta aproximação leva em conta que o movimento dos núcleos é mais lento que o movimento dos elétrons, então podemos separar a informação nuclear e eletrônica em duas partes e resolvê-las separadamente. Dessa aproximação (da mecânica quântica), observamos que em MM a energia total do sistema depende

[1]Parte do texto foi extraído da Tese de Doutorado do Prof. Fernando Sato e pode ser encontrada no formato pdf na biblioteca do Instituto de Física Gleb Wataghin da Universidade Estadual de Campinas.

[2]Um potencial entre átomos ou molécular também é referido na literatura como *campo de força*.

exclusivamente da posição dos átomos do sistema e os efeitos eletrônicos não são computados explicitamente. A energia total então é dada via potencial nuclear dependente das posições, mais conhecido pela denominação de *campo de força* (CF).

O campo de força é uma peça fundamental, se não a mais importante, em MM. Para se ter uma bom resultado de geometria em MM é necessário que o CF esteja adequado ao tipo de sistema no qual deseja tratar. Comunmente os CF são compostos por termos harmômicos para átomos ligados, termos de van der Walls e de Coulomb para átomos não ligados. Os termos dos átomos ligados têm a forma de kx^2, onde x pode assumir valores de distância ou ângulo. Com o CF, a força do sistema é obtida via força central pela derivada da expressão espacial do potencial, que é a derivada da expressão do CF. Por fim, o sistema deve ser conservativo, uma vez que o potencial central impõe a conservação do momento angular total do sistema.

Para exemplificar um CF, na equação 6.0.1 temos um CF que é utilizado para MM também utilizado para DM, este campo de força na literatura é encontrado com o nome de *Universal Force Field*.

$$
E_{\text{potencial}} = \begin{cases}
\sum_r \frac{1}{2} K_{IJ} \left(r - r_{IJ}\right)^2 \\
+ \sum_\theta \begin{cases}
K_{IJK}(1 - cos(\theta)) & : \text{linear} \\
(K_{IJK}/9)(1 - cos(3\theta)) & : \text{trigonal planar} \\
(K_{IJK}/16)(1 - cos(16\theta)) & : \text{quadrático planar} \\
(K_{IJK}/16)(1 - cos(16\theta)) & : \text{octaedrico} \\
K_{IJK}(C_0^\theta + C_1^\theta cos(\theta) + C_2^\theta cos(2\theta)) & : \text{caso geral}
\end{cases} \\
+ \sum_\phi 1/2 V_\phi \left(1 - cos(n\phi_0) cos(n\phi)\right) \\
+ \sum_\omega K_{IJKL} \left(C_0^\omega + C_1^\omega cos(\omega_{IJKL}) + C_2^\omega cos(2\omega_{IJKL})\right) \\
+ \sum_x D_{IJ} \left[-2 \left(x_{IJ}/x\right)^6 + \left(x_{IJ}/x\right)^{12}\right] \\
+ \sum_R Q_I Q_J / \epsilon R_{IJ}
\end{cases}
\tag{6.0.1}
$$

A equação 6.0.1 se refere ao CF *Universal Force Field* (UFF) [18]. A descrição dos termos da equação 6.0.1 é dado por:

- $\sum_r$, $\sum_\theta$, $\sum_\phi$, $\sum_\omega$, $\sum_x$ e $\sum_R$ são os termos de ligação, angular, torção, inversão, van der Waals (vdw) e coulombiano, respectivamente ;
- K_{IJ}, K_{IJK} e K_{IJKL} são constante de força de ligação, angular e torção/inversão, respectivamente;
- r e r_{IJ} são as distâncias calculadas e de equilíbrio entre dois átomos ligados I-J;
- θ e θ_0 são os ângulos calculados e de equilíbrio entre três átomos I-J-K;

- ϕ e ϕ_0 são os ângulos de torção calculados e de equilíbrio entre quatro átomos I-J-K-L;
- ω é o ângulo de inversão do átomo I em relação ao plano formado pelos átomos J-K-L;
- C_0^θ, C_1^θ e C_2^θ são os coeficientes de Fourier para o caso angular geral;
- C_0^ω, C_1^ω e C_2^ω - são os coeficientes de Fourier do termo de inversão;
- V_ϕ e n - são a altura da barreira de torção e a periodicidade do potencial de torsão;
- x e x_{IJ} - são as distâncias calculadas e de equilíbrio entre dois átomos não ligados I e J;
- D_{IJ} - é a profundidade do potencial de Lennard-Jones;
- Q_I e ϵ - são as cargas parciais do átomo I e a constante dielétrica.

Em um sistema molecular dizemos que o sistema está em equilíbrio quando o somatório de todas as forças que atuam em cada átomo tendem para zero. Para completar o processo de MM é necessário um método de comparação entre os passos de otimização. Cada passo de otimização está relacionado com um pequeno movimento do átomo, em geral na direção do ponto de equilíbrio, devido a ação da força proveniente do potencial. Em um sistema multi-atômico modificar as posições dos átomos e calcular a energia total até encontrar um mínimo entre as possíveis configurações pode ser um tanto quanto trabalhoso. Para isso utilizamos outros recursos que nos auxiliam na busca da geometria como o método do *gradiente conjugado* [16]. Além do método do gradiente conjugado, outros métodos matemáticos de minização utilizam a expressão de energia potencial (energia total) como critério de convergência, onde, um sistema com uma boa geometria, apresenta em geral, diferença de energia entre um ciclo e outro menor que $10^{-1} kcal/mol$ Å [16] após os passos de otimização.

Do processo da MM utilizamos o conceito de CF para resolver as equações de movimento no tempo, e a resolução dessas equações é então chamada de DM. A DM [19–21] apresenta conceitualmente algumas características somadas ao processo dinâmico do sistema molecular e inclui o termo de energia cinética na energia total, o que não acontece em MM que pois esta representa a otimização de geometria de um sistema molecular à temperatura de zero Kelvin. As principais características da DM são: resolução das equações de movimento (segunda lei de Newton) com dependência temporal e ajuste da temperatura do sistema (no caso do ensamble seja canônico (NVT) a temperatura é constante).

Como já sabemos se o sistema for conservativo podemos obter a força do sistema a partir da energia potencial do sistema como descrito pela equação 6.0.2.

$$\vec{F} = -\vec{\nabla}U \tag{6.0.2}$$

$$\vec{F} = m\vec{a} = m\frac{d\vec{v}}{dt} = m\frac{d^2\vec{r}}{dt^2} \tag{6.0.3}$$

$$\vec{a} = -\frac{\vec{\nabla}U}{m} \tag{6.0.4}$$

$$\frac{d\vec{v}}{dt} = \vec{a} \tag{6.0.5}$$

$$\frac{d\vec{r}}{dt} = \vec{v} \tag{6.0.6}$$

Pela segunda lei de Newton (equação 6.0.3) podemos encontrar a aceleração (equação 6.0.4) utilizando e derivada da expressão do potencial, ou como chamamos acima de CF, a equação 6.0.2. Para se obter as velocidades e posições é necessário integrar as equações 6.0.5 e 6.0.6. A integração das equações de movimento no processo de dinâmica molecular é realizada numericamente ou por expressões analíticas. Dentre as técnicas de integração uma das mais utilizadas é o algorítmo de Verlet, para o desenvolvimento das nossas atividades utilizaremos o algortimo *Velocity Verlet* (eq. 5.3.14).

Com esse breve histórico de MM e DM, podemos iniciar a construção do nosso programa de dinâmica molecular. No programa a ser construído levaremos em conta os seguintes aspectos:

1. O sistema que iremos simular é um gás de Argônio (gás inerte - Ar) dentro de uma caixa. Assim teremos um número fixo de átomos de Ar dentro de uma caixa de dimensões a ser definida. A inclusão da caixa indica que teremos que utilizar condições periódicas de contorno;

2. Utilizaremos todas as dimensões baseado no sistema internacional de unidades (SI);

3. Utilizaremos somente interação de van der Walls (vdW) como o potencial entre os átomos, a implementação de outros tipos de potenciais demanda muito tempo por isso não faremos aqui. Este tipo de potencial se encaixa na interação entre átomos não ligados;

4. Uma vez definida uma caixa de dimensões $x \times y \times z$ atribuiremos as posições aleatoriamente dos átomos de Ar (x, y, z) dentro das dimensões da caixa.

5. Com as posições definidas, atribuiremos as velocidades aleatoriamente baseado em uma distribuição gaussiana;

6. Com as velocidades teremos uma energia cinética, assim as velocidades deverão ser normalizadas para uma temperatura em Kelvin baseada no Princípío da Equipartição de Energia, onde o pico da distribuição gaussiana das velocidades deve estar relacionado com a temperatura desejada;

7. O banho térmico será simples dado pelo reescalamento das velocidades;

8. Durante a resolução das equações no tempo iremos guardar (escreve em um arquivo) a energia cinética, energia potencial, temperatura, passo, (em um outro arquivo) as posições atômicas.

9. É desejável que o programa da geração das coordendas seja independente do programa de DM.

10. Também é desejável que o programa de DM utilize dois arquivos de entrada um com as coordenadas e outro com os dados de controle da DM, como por exemplo, temperatura, Δt, números de passos, tamanho da caixa, números de passos para escrever os dados, número de passos para escrever a trajetória, etc.

6.1 Geração das Coordenadas

Para gerar as co rdenadas construiremos um programa que leia do teclado a quantidade de átomos desejados, o tamanho dos vértices da caixas para garantir que todos átomos estarão dentro da caixa. Como comentado anteriormente utilizaremos o gerador de números aleatórios para atribuir as coodenadas iniciais.

No programa a seguir, é utilizado o tamanho do raio de van der Walls (vdw) para fazer a distribuição aleatória ou ordenada dos átomos de Argônio (Ar) dentro de uma caixa. Na distribuição aleatória é respeitada a distância entre átomos de Ar de no mínimo 3.76 Åpois o raio de vdw para o átomo de Ar é de 1.88 Å. Na distribuições de posições ordenadas fixa-se as coordenadas y e z e distribui-se ao longo da coordenada x. Quando x é completamente preenchido adiciona-se é acrescentado um delta em y (3.76 Å) e depois distribui-se ao longo de x novamente, primeiro é preenchido as posições nas coordenadas em x, depois em y e por último em z. O número de átomos total é baseado nas dimensões da caixa (x,y,z) e no diâmetro do átomo de Ar ditado pelo raio de vdw. Supondo que as dimensões da caixa seja $lx = 25$, $ly = 35$, $lz = 50$ então teremos $nx = lx/3.76 =$, $ny = ly/3.76$ e $nz = lz/3.76$ com o total de $6 \times 9 \times 13 = 702$ átomos, que é o número máximo que caberia dentro da caixa.

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
2  ! VARIAVEL N SEMPRE SERA COMPARTILHADA ENTRE AS SUBROTINAS
3  ! QUE GERAM OS NUMEROS ALEATORIOS.
4  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
5  module aleatorio
6  implicit none
7  integer, save :: n=6677
8  end module aleatorio
9  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
10
11 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
12 ! PROGRAMA QUE GERAM POSICOES BASEADO NO TAMANHO DA CAIXA
13 ! PARA ENTRADA DO PROGRAMA DE DM
14 ! RAIO DE vdW PARA Ar R_vdw(Ar)= 1.88 Angstron
15 ! http://www.webelements.com/periodicity/van_der_waals_radius/
16 ! VOLUME DA ESFERA = (4.d0/3.d0)*pi*r**3
17 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
18 program gera_xyz
19 implicit none
20
21 ! VARIAVEIS DO GERADOR DE NUMEROS ALEATORIOS
22 real(kind=8) :: nran                ! NUMERO ALEATORIO
23 integer :: isemente                 ! SEMENTE INICIAl
24
25 ! VARIAVEIS DA GERACAO DAS POSICOES ATOMICAS
26 real(kind=8) :: ax,ay,az            ! TAMANHO DA CAIXA
27 real(kind=8) :: rx,ry,rz            ! X(I+1)-X(I), Y(I+1)-Y(I), Z(I+1)-Z(I)
28 real(kind=8) :: rij                 ! DISTANCIA ENTRE DOIS ATOMOS
29 real(kind=8) :: rmin                ! DISTANCIA MINIMA ENTRE ATOMOS
30 real(kind=8), allocatable, dimension(:) :: x,y,z          ! POSICAO DOS
       ATOMOS
31 real(kind=8), parameter :: vdw_ar = 1.88d0
32 integer :: i, j, k, l               ! UTILIZADOS NOS DO'S
33 integer :: natom,natomx,natomy,natomz     ! NUMERO DE ATOMOS
34 integer :: refaca                   ! PARA NO SORTEIO DE POSICOES
35
36 ! VERIFICANDO QUANTIDADE DE ATOMOS DENTRO DA CAIXA
37 real(kind=8) :: diametro
38 integer :: lx,ly,lz
39 integer :: natomos
40 integer :: tipo
41
42 ! ARQUIVO DE SAIDA
43 open(unit=20, file='coord.xyz', status='replace', action='write')
44
45 ! DIMENSOES DA CAIXA
46 write(*,*) 'Entre com o tamanho de x, y e z (em Angstrons):'
47 write(*,*) '(Dimensoes da caixa)'
48 read(*,*) ax, ay, az
```

```
49
50 ! INDICANDO O NUMERO MAXIMO DE ATOMOS QUE CAIBAM DENTRO DA CAIXA
51 ! 1 ATOMO DE Ar OCUPA UM QUADRADO DE LADO 2.d0*1.88d0 Angstrons
52 diametro = vdw_ar * 2.d0
53 lx = int(ax/diametro)
54 ly = int(ay/diametro)
55 lz = int(az/diametro)
56 natomos = lx*ly*lz
57 write(*,*)' '
58 write(*,*)'O numero maximo de atomos que cabem dentro da caixa e:',
       natomos
59 write(*,*)' '
60
61 ! NUMEROS DE ATOMOS DESEJADO
62 write(*,*) 'Entre com o numero de atomos (deve ser inteiro):'
63 write(*,*) 'OBS: Caso seja aleatorio nao colocar mais do que',int(0.9d0
       *dfloat(natomos)),'atomos (90% do total)'
64 read(*,*) natom
65
66 ! CONDICAO VOLUME DO NUMERO DE ATOMOS DESEJADO NAO PODE SER MAIOR
67 ! QUE O VOLUME DA CAIXA DESEJADA
68 if (natom > natomos) then
69   write(*,*) 'ATENCAO >>> O Volume de atomos desejado eh muito'
70   write(*,*) '>>> maior que o volume da caixa desejado !!!'
71   stop
72 end if
73
74 ! CONDICAO ENTRE POSICAO ORDENADA OU ALEATORIA
75 write(*,*)' '
76 write(*,*)'Sorteio das posicoes sera aleatorio (1) ou ordenado (2) ?'
77 read(*,*) tipo
78 write(*,*)' '
79
80 ! ALOCANDO AS POSICOES ATOMICAS
81 allocate(x(natom))
82 allocate(y(natom))
83 allocate(z(natom))
84 ! DISTANCIA MINIMA ENTRE ATOMOS
85 rmin=2.d0*vdw_ar
86 ! ESCREVENDO CABECALHO DO ARQUIVO DE SAIDA
87 write(20,*) natom
88 write(20,500) ax,ay,az
89 500 format('dimensoes da caixa',f10.6,f10.6,f10.6)
90
91 aleat: if ( tipo == 1 ) then
92
93 write(*,*) 'Entre com a semente: '
94 read(*,*) isemente
```

```
95  ! CHAMANDO A SUBROTINA SEED
96  call semente(isemente)
97
98  loop1: do i = 1, natom
99    ! CHAMADA DO GERADOR DE NUM.ALEAT.
100   call num_aleatorio(nran)
101   x(i) = nran * ax
102   call num_aleatorio(nran)
103   y(i) = nran * ay
104   call num_aleatorio(nran)
105   z(i) = nran * az
106   ! VERIFICANDO POSICAO ATOMICA
107   j = i
108   if ( j > 1 ) then
109     loop2: do
110       refaca = 0
111       ! CALCULA RIJ
112       ! SE MENOR QUE RMIN SORTEIA NOVA POSICAO
113       do k=1,j-1
114     rx = x(j) - x(k)
115     ry = y(j) - y(k)
116     rz = z(j) - z(k)
117     rij = dsqrt( rx*rx + ry*ry + rz*rz )
118     if (rij < rmin) then
119       refaca=1
120     end if
121      end do
122       ! SORTEIO DA NOVA POSICAO
123       if ( refaca == 1 ) then
124     call num_aleatorio(nran)
125     x(j) = nran * ax
126     call num_aleatorio(nran)
127     y(j) = nran * ay
128     call num_aleatorio(nran)
129     z(j) = nran * az
130      end if
131       ! CONDICAO DE SAIDA
132       if ( refaca == 0 ) exit
133     end do loop2
134   end if
135   write(20,501) x(i),y(i),z(i)
136 end do loop1
137 endif aleat
138
139 ! POSICOES NO MODO ORDENADO
140 ! NESTE TIPO DE SORTEIO EH PURAMENTE PARA FINS DIDATICOS
141 ! NAO SEI SE TEM SENTIDO FISICO
142 orden: if ( tipo == 2 ) then
```

```
143   natomx = int( ax / rmin )
144   natomy = int( ay / rmin )
145   natomz = int( az / rmin )
146   l = 0
147   do i = 1, natomz
148     ! CONDICAO DE PARADA: SENAO ERRO CRITICO
149     if ( l == natom ) exit
150       do j = 1, natomy
151      ! CONDICAO DE PARADA: SENAO ERRO CRITICO
152      if ( l == natom ) exit
153        do k = 1, natomx
154          l = l + 1
155          if ( k == 1 ) then
156            x(l) = 0.5d0 * rmin
157          else
158            x(l) = x(1) + dfloat(k-1) * rmin
159          end if
160          if ( j == 1 ) then
161            y(l) = 0.5d0 * rmin
162          else
163            y(l) = y(1) + dfloat(j-1) * rmin
164          end if
165          if ( i == 1 ) then
166            z(l) = 0.5d0 * rmin
167          else
168            z(l) = z(1) + dfloat(i-1) * rmin
169          end if
170          ! CONDICAO DE PARADA: SENAO ERRO CRITICO
171          if ( l == natom ) exit
172        end do
173       end do
174   end do
175   do i = 1, natom
176     write(20,501) x(i),y(i),z(i)
177   end do
178 end if orden
179
180 501 format('Ar',2x,f16.6,f16.6,f16.6)
181 stop
182 end program gera_xyz
183
184 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
185 subroutine num_aleatorio(nran)
186 use aleatorio
187 implicit none
188 real(kind=8), intent(out) :: nran
189 n = mod( 8127*n+28417 , 134453 )
190 nran = dfloat(n) / 134453.d0
```

```
191 end subroutine num_aleatorio
192
193
194 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
195 subroutine semente(isemente)
196 use aleatorio
197 implicit none
198 integer, intent(in) :: isemente
199 n=abs(isemente)
200 end subroutine semente
```

6.2 Início do Programa de DM

Em princípio, os programas para realizar algum tipo de simulação necessita de dados de entrada, alguns somente de controle, outros somente coordenadas iniciais ou alguns com informações combinadas. É importante pensar em um programa que tenha um caráter genérico de entrada de dados, dentro dessa generalidade é importante que o programa leia os dados de entrada de um arquivo. O porque desse esquema se baseia no fato de que as simulações podem levar um longo tempo para serem finalizadas ou então curtos intervalos de tempo, mas várias simulações com vários intervalos de tempos. No ambiente *unix-like* (linux) temos uma ferramenta importante e poderosa que nos permite automatizar uma série de tarefas a serem realizadas (programa a serem rodados), mas para isso precisamos de um programa (feito em fortran ou em c/c++) que permite essa automatização.

No caso específico de DM necessitamos das coordendas iniciais e também de um arquivo de controle. Para leitura do arquivo de coordendas utilizaremos o formato *xyz* que é um formato utilizado por vários programas que também realizam DM e que visualizam a trajetória (uma sequência de arquivos *xyz*) como os programas *xmakemol* [22], *vmd* [23], *gopenmol* [24], etc. O arquivo de controle da DM faremos de acordo com as nossas necessidades.

O arquivo *xyz* conforme o programa anterior que gera as coordenadas é composta por três partes. A primeira é o número total de átomos do arquivo que está localizado na primeira linha do arquivo, a segunda é uma linha de comentário localizado na segunda linha do arquivo e a terceira e última é a sequência de posições atômicas em que cada linha possui quatro colunas contendo o rótulo do átomo e as coordenadas x, y, e z. Um exemplo pode ser visto abaixo:

```
1  310
2 linha de comentario
3 Ar    7.167486   26.499595   28.546332
4 Ar   12.382171   16.242033    5.338966
5 Ar   16.117528   13.491108   28.578685
6 Ar    5.317546   22.036176   24.344641
```

```
7 Ar    5.236997   27.417759   20.470945
```

Para ler o arquivo de coordendas, é recomendado que utilizemos o recurso de alocação dinâmica de memória, assim não precisamos compilar o programa todas as vezes que trocarmos a quantidade de átomos que desejarmos estudar. Parte do programa poderia ser feito da seguinte forma:

```
1  ! IMPORTANTE COLOCAR ESSAS VARIAVEIS DENTRO DE UM MODULE
2  integer :: natom
3  character(40) :: comentario
4  character(2), allocatable, dimension(:) :: rotulo
5  real(kind=8), allocatable, dimension(:) :: x, y, z, m
6  !
7  integer :: i
8  !...
9  open(unit=20,file='coord.xyz,status='old',action='read')
10 !...
11 ! LEITURA DO ARQUIVO DAS COORDENADAS
12 read(20,*)natom
13 read(20,*)comentario
14 allocate(x(natom))
15 allocate(y(natom))
16 allocate(z(natom))
17 allocate(rotulo(natom))
18 !
19 ! Massa Ar (kg) = 39.948 uma x 1.6605402d-27
20 !
21 do i = 1, natom
22   read(20,*)rotulo(i),x(i),y(i),z(i)
23   x(i) = x(i) * 1.d-10 ! CONVERTENDO PARA METROS
24   y(i) = y(i) * 1.d-10 ! CONVERTENDO PARA METROS
25   z(i) = z(i) * 1.d-10 ! CONVERTENDO PARA METROS
26   m(i) = 6.633526d-27 ! PESO EM KG
27 end do
28 !...
```

O programa será simples e os controles tambéms serão simples, mas com uma base bastante sólida para um programa mais complexo e destinado à pesquisa científica. Teremos como dados de controle as dimensões da caixa (em angstrons), temperatura desejada (em Kelvin), número de passos total (número inteiro), intervalo de tempo de integração (real), o intervalo que serão escritos os dados de energia e a trajetória (inteiro) e o intervalo em que serão reescaladas as velocidades (inteiro).

O arquivo de controle poderia ser da seguinte maneira:

```
1 30.d0 30.d0 30.d0
2 300.d0
3 1000
4 1.d0
```

```
10
20
13

[1] Dimensoes da caixa lx, ly, lz
[2] Temperatura em Kelvin
[3] Numero de passos da simulacao
[4] dt - intervalo de tempo de integracao em femto segundos
[5] Intervalo para escrita dos dados e trajetoria
[6] Intervalo para reescalar as velocidades
[7] Semente para distribuicao das velocidades
```

Parte do código para leitura desses dados poderia ser da seguinte maneira:

```
! IMPORTANTE COLOCAR ESSAS VARIAVEIS DENTRO DE UM MODULE
! DIMENSOES DA CAIXA
real(kind=8) :: lx, ly, lz
! TEMPERATURA DESEJADA
real(kind=8) :: temperatura
! NUMERO DE PASSOS TOTAL
integer :: ntotal
! INTERVALO DE TEMPO PARA INTEGRACAO DAS EQ. DE MOVIMENTO
real(kind=8) :: dt1
! INTERVALO DE TMEPO PARA ESCRITA DOS DADOS E BANHO TERMICO
integer :: nw, nb
! SEMENTE PARA GERACAO DE NUMEROS ALEATORIOS
integer :: isemente
!...
open(unit=21,file='control.inp,status='old',action='read')
!...
read(21,*) lx, ly, lz
lx = lx * 1.d-10 ! CONVETENDO PARA METROS
ly = ly * 1.d-10 ! CONVETENDO PARA METROS
lz = lz * 1.d-10 ! CONVETENDO PARA METROS
read(21,*) temperatura
read(21,*) ntotal
read(21,*) dt1
dt1 = dt1 * 1.d-15 ! CONVETENDO PARA METROS
read(21,*) nw
read(21,*) nb
read(21,*) isemente
!...
```

Nestes primeiros passos da construção do programa de DM, vimos como gerar e ler os arquivos contendo as coordendas e também como ler o arquivo de controle. Na figura 6.1 é apresentado um esquema básico do funcionamento do programa de DM. É praticamente impossível desenvolver algo desta dimensão se não tivermos pelo menos uma idéia de como irá funcionar o programa. É muito interessante que haja um questionamento para cada pedaço do fluxograma e como poderia ser transformar

em um código computacional.

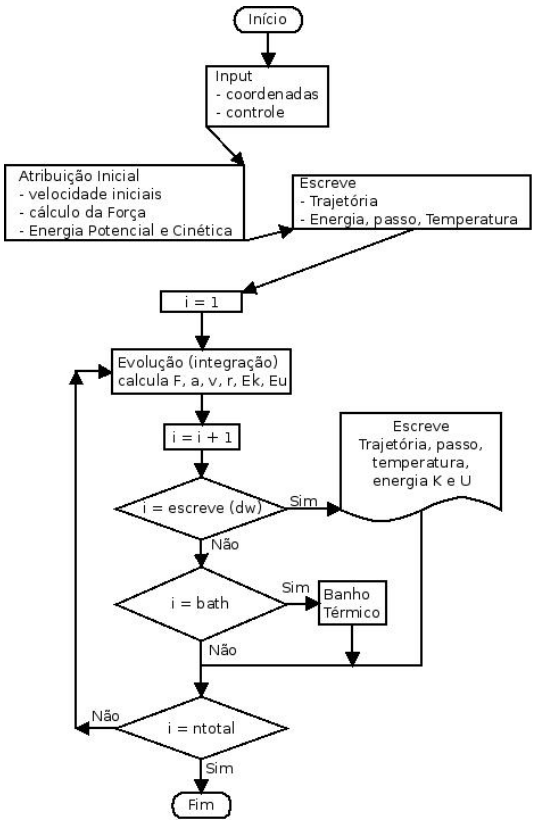


Figura 6.1: Fluxograma básico do programa de Dinêmica Molecular.

Com base no fluxograma a idéia do programa principal de MD poderia assumir a forma abaixo, contendo várias subrotinas, utilizando o compartilhamento de variáveis *module* e a alocação dinâmica de memória:

```
1  !...
2  ! LENDO CONTRLE DA SIMULACAO
3  call r_control
4  ! LENDO COORDENADAS INICIAIS
5  call r_xyz
6  ! INICIANDO AS VELOCIDADES
7  call inicia_vel
8  ! CALCULANDO AS FORCAS
9  call forca
10 ! CALCULANDO A ACELERACAO
11 call acel
12 ! CALCULANDO ENERGIA POTENCIAL
13 call e_potencial
14 ! CALCULANDO ENERGIA CINETICA
15 call e_cinetica
16 ! CALCULANDO TEMPERATURA
17 call temper
18 ! ESCREVENDO POSICOES INICIAIS NA TRAJETORIA
19 call w_trj
```

```
20 ! ESCREVENDO ENERGIA, TEMPERATURA, PASSO, ETC
21 call w_data
22 !
23 banho_termico = 0
24 escreve = 0
25 !
26 do i = 2, ntotal
27      ! CALCULA NOVA POSICAO
28      call evolucao
29      ! CALCULA FORCA, ACELERACAO, ENERGIAS, TEMPERATURA
30      call forca
31      call aceleracao
32      call e_potencial
33      call e_cinetica
34      call temper
35
36      ! CONDICAO DE ESCRITA
37      if ( escreve == nw ) then
38        call w_trj
39        escreve = 0
40      else
41        escreve = escreve + 1
42      end if
43
44      ! CONDICAO DO BANHO TERMICO
45      if ( banho_termico == nb ) then
46        call bt
47        banho_termico = 0
48      else
49        banho_termico = banho_termico + 1
50      end if
51 end do
```

6.3 Atribuição das Velocidades

Inicialmente não possuimos velocidades para os átomos e precisamos de alguma forma atribuir respeitando algumas condições iniciais. Uma delas o tipo de distribuição (distribuição tipo Boltzman) e a outra que o momento linear seja zero. Utilizaremos o gerador de números aleatórios e então atribuiremos valores entre −1 e +1, em seguida zeramos o momento linear.

```
1 ! SOMENTE NESTA SUBROTINA QUE SERA UTILIZADO
2 ! A ROTINA DE NUMEROS RANDOMICOS
3 ! VARIAVEIS RECOMENDADO ESTAR NO MODULE
4 real(kind=8), allocatable, dimension(:) :: vx, vy, vz
5 real(kind=8) :: svx, svy, svz
6 ! VARIAVEL LOCAL
```

```
integer :: i
! SORTEANDO VELOCIDADES INICIAIS, VALORES NO INTERVALOR [-1,1]
svx = 0.d0; svy = 0.d0; svz = 0.d0
do i = 1, natom
   call num_aleatorio(nran)
   vx(i) = (-1.d0)**nint(nran) * nran
   svx = svx + vx(i)
   call num_aleatorio(nran)
   vy(i) = (-1.d0)**nint(nran) * nran
   svy = svy + vy(i)
   call num_aleatorio(nran)
   vz(i) = (-1.d0)**nint(nran) * nran
   svz = svz + vz(i)
end do
! ZERANDO O MOMENTO LINEAR
do i = 1, natom
   vx(i) = vx(i) - (svx/natom)
   vy(i) = vy(i) - (svy/natom)
   vz(i) = vz(i) - (svz/natom)
end do
! ESCALANDO AS VELOCIDADES PARA A TEMPERATURA ALVO
call vel_scal
```

Para escalar a velocidade para a temperatura alvo, precisamos inicialmente calcular a temperatura atual. Para isso chamamos a rotina que calcula a temperatura atual. O calculo da temperatura instantânea é baseado na equação 6.3.1 e 6.3.2.

$$\sum_{i=1}^{N} \frac{1}{2} m_i \vec{v}_i^2 = \frac{3}{2} N k_b T \tag{6.3.1}$$

$$T = \frac{2 \sum_{i=1}^{N} \frac{1}{2} m_i \vec{v}_i^2}{3 N k_b} \tag{6.3.2}$$

Talvez seja desejável criar uma subrotina que calcule a energia cinética, pois iremos utilizar esta informação todas as vezes para calcular a temperatura instantânea e para calcular a energia mecânica total. Rotulamos a energia cinética total como K, assim teremos:

$$T = \frac{2K}{3 N k_b} \tag{6.3.3}$$

e a velocidade escalada para a temperatura desejada é dada pela equação 6.3.4

$$v_{xi}^{nova} = v_{xi}^{antiga} \sqrt{\frac{T_{alvo}}{T_{inst}}} \tag{6.3.4}$$

calculando a energia cinética

```
!..
ek = 0.d0
do i = 1, natom
  v = dsqrt(vx(i)*vx(i) + vy(i)*vy(i) + vz(i)*vz(i))
  ek = ek + m(i)*v
end do
ek = ek*0.5d0
!...
```

escalando as velocidades

```
!...
! kb = 1.3800D-23 J/K = 8.6200D-5 eV/K
real(kind=8), parameter :: kb = 1.3800D-23

t_inst = (2.d0 * ek) / (natom * kb * 3.d0)
fator = dsqrt{temperatura/t_inst}
do i = 1, natom
  vx(i) = vx(i) * fator
  vy(i) = vy(i) * fator
  vz(i) = vz(i) * fator
end do
!...
```

6.4 Cálculo da Força e Energia Potencial

Como vimos no início do capítulo calcularemos a força conforme a equação 6.0.2. O potencial é o conhecido Lennard-Jones 6-12 dado pela equação 6.4.1 e a derivada pela equação 6.4.2. Em geral, como a força e a energia potencial envolve alguns parametros iguais o cálculo é realizado simultaneamente.

$$U(r) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \tag{6.4.1}$$

$$F(r) = -\frac{dU(r)}{dr} = 24\frac{\sigma}{\epsilon}\left[2\left(\frac{\sigma}{r}\right)^{13} - \left(\frac{\sigma}{r}\right)^{7}\right] \tag{6.4.2}$$

- r é a distância entre o átomo i e o átomo j;
- ϵ é o mínimo valor do gráfico $U(r)$ vs r;
- σ é a distância de equilíbrio de vdW no gráfico $U(r)$ vs r;
- $\epsilon = 0.185$ e $\sigma = 3.868$, da referência [18].

O cálculo da energia potencial é realizada da seguinte maneira:

$$u(r) = \sum_{i}^{N} \sum_{i<j}^{N} U(r_{ij}) \tag{6.4.3}$$

o mesmo deve aser aplicado para o cálculo da força. É de se observar que o somatório envolve o átomo i com todos os átomos j, no entando não iremos calcular para todos os átomos j. Iremos truncar o somatório no cálculo do potencial e da força se a distância $r_{ij} > r_c$, em que r_c é um raio de corte. A justificativa para essa truncagem baseia-se no fato de que as contribuições acima do raio de corte são muito pequenas, já que o potencial é de curto alcance. Utilizaremos como raio de corte um valor de 2.5σ [19].

Outro detalhe que devemos nos atentar é que as posições, velocidades e acelerações estão em coordenadas cartesianas e o potencial e a força estão em coordendas polares esféricas (F e U estão em função de r). O fato é que para atribuir os valores para as componentes da aceleração deveremos fazer uma conversão de coordenadas polares esféricas para coordenadas cartesianas. Neste ponto é o único momento em que iremos utilizar a transformação de coordenadas.

```
1  !...
2  ! VARIAVEIS QUE DEVEM ESTA NO MODULE
3  ! E ALOCADAS PREVIAMENTE
4  real(kind=8), parameter :: pi = 3.14159265358979 67d0
5  real(kind=8), parameter :: sigma = 3.868d-10
6  real(kind=8), parameter :: epsilon = 0.185d0
7  real(kind=8), allocatable, dimension(:) :: x, y, z
8  real(kind=8), allocatable, dimension(:) :: fx, fy, fz, f, u
9  real(kind=8) :: eu_total, rc
10 integer :: natom
11 ! VARIAVEIS LOCAIS
12 real(kind=8) :: sigma, epsilon, phi, theta
13 real(kind=8) :: xx, yy, zz, rij, rs, r6, r7, r12, r13
14 integer :: i, j
15 !...
16 rc = sigma * 2.5d0
17 eu_total = 0.d0
18 f(1) = 0.d0; u(1) = 0.d0
19 !
20 loopi: do i = 1, natom
21
22   loopj: do j = 1, natom
23
24     ! SOMENTE PARA J DIFERENTE DE I
25     condji: if ( j /= i ) then
26       xx = x(i) - x(j)
27       yy = y(i) - y(j)
28       zz = z(i) - z(j)
29       rij = dsqrt(xx*xx + yy*yy + zz*zz)
```

```
30          rs  = dsqrt(xx*xx + yy*yy)
31
32          ! CONDICAO DO RAIO DE CORTE
33          condrij: if ( rij <= rc ) then
34
35        ! CALCULO DA ENERGIA POTENCIAL DO ATOMO I
36        r12 = (sigma/rij)**12
37        r6  = (sigma/rij)**6
38        u(i) = u(i) + 4.d0 * epsilon * (r12 - r6)
39
40        ! CALCULO DA FORCA NO ATOMO I
41        r13 = (sigma/rij)**13
42        r7  = (sigma/rij)**7
43        f(i) = f(i) + 24.d0 * (epsilon/sigma) * (2.d0*r13 - r7)
44
45        ! CONVERSAO DE COORDENADAS
46  ! ESTA CONVERSAO PODE SER APLICADA PARA QUALQUER CASO ONDE
47  ! AS COORDENADAS ESTIVEREM ESPALHADAS EM TODOS OS QUADRANTES
48  ! DAS COORDENADAS CARTESIANAS. SEM SOMBRA DE DUVIDA QUE E MAIS
49  ! INTELIGENTE/RAPIDO, SE ANTES DE INICIAR A SIMULACAO DESLOCAR
50  ! O SISTEMA PARA O PRIMEIRO QUADRANTE, ASSIM DIMUNUI O NUMERO
51  ! DE CONDICOES ABAIXO.
52
53        ! ANGULO PHI (COM RELACAO A Z)
54        if ( zz == 0.0d0 ) then
55          phi = pi / 2.0d0
56        else
57          phi = dacos(zz / rij))
58        end if
59
60        ! ANGULO THETA (COM RELACAO A X)
61        if ((xx >= 0.0d0) .and. (yy > 0.0d0)) then
62          theta = dasin(yy/rs)
63        end if
64        if ((xx <  0.0d0) .and. (yy > 0.0d0)) then
65          theta = pi - dasin(yy/rs)
66        end if
67        if ((xx <  0.0d0) .and. (yy < 0.0d0)) then
68          theta = pi + dasin(abs(yy)/rs)
69        end if
70        if ((xx >  0.0d0) .and. (yy < 0.0d0)) then
71          theta = (3.0d0*pi/2.0d0) + dacos(abs(yy)/rs)
72        end if
73        if ((yy == 0.0d0) .and. (xx > 0.0d0)) then
74          theta = 0.0d0
75        end if
76        if ((yy == 0.0d0) .and. (xx < 0.0d0)) then
77          theta = pi
```

```
78        end if
79
80        ! FORCAS CONVERTIDAS PARA COORD CARTESIANAS
81        fx(i) = fx(i) + f(i) * dsin(phi) * dcos(theta)
82        fy(i) = fy(i) + f(i) * dsin(phi) * dsin(theta)
83        fz(i) = fz(i) + f(i) * dcos(phi)
84
85         end if condrij
86
87       end if condij
88
89     end do loopj
90
91     eu_total = eu_total + u(i)
92
93  end do loopi
94  !...
```

6.5 Integrando com *Velocity Verlet*

Após o cálculo das forças, podemos calcular as componentes da aceleração de cada átomos e após calcular as novas posições e velocidades.

Para calcular as componentes da aceleração utilizaremos a equação 6.5.1.

$$a_{xi} = \frac{F_{xi}}{m_i}; a_{yi} = \frac{F_{yi}}{m_i}; a_{zi} = \frac{F_{zi}}{m_i} \tag{6.5.1}$$

```
1  !...
2  ! VARIAVEIS QUE DEVEM ESTAR NO MODULE
3  real(kind=8), allocatable, dimension(:) :: ax, ay, az
4  integer :: natom
5  ! VARIAVEIS LOCAIS
6  integer :: i
7  !...
8  do i = 1, natom
9     ax(i) = fx(i)/m(i)
10    ay(i) = fy(i)/m(i)
11    az(i) = fz(i)/m(i)
12 end do
13 !...
```

Com o procedimento realizado até o momento, temos as posições (x_0, y_0, z_0), as velocidades (vx_0, vy_0, vz_0), as acelerações (ax_0, ay_0, az_0) e as forças (fx_0, fy_0, fz_0), que compõe o conjunto inicial. Para obter o conjunto das posições (x_1, y_1, z_1), das velocidades (vx_1, vy_1, vz_1), das acelerações (ax_1, ay_1, az_1) e das forças (fx_1, fy_1, fz_1) devemos utilizar algum integrador das equaçõe de movimento de Newton,

e a escolha aqui é o método de *Velocity Verlet* pela simplicidade e facilidade de implementação. Assim poderemos evoluir o sistema até o conjunto das posições (x_n, y_n, z_n), das velocidades (vx_n, vy_n, vz_n), das acelerações (ax_n, ay_n, az_n) e das forças (fx_n, fy_n, fz_n).

De acordo com as equações 5.3.14, novamente exposta abaixo, é necessário primeiro calcular as posições para o tempo $(t + \Delta t)$, na sequência calcular as componentes das forças para o tempo $(t + \Delta t)$, para termos então as componentes das velocidades e das acelerações no tempo $(t + \Delta t)$.

$$\begin{aligned}
\vec{r}(t + \Delta t) &= \vec{r}(t) + \vec{v}(t)\Delta(t) + (1/2)\vec{a}(t)\Delta t^2 \\
\vec{v}(t + \Delta t/2) &= \vec{v}(t) + (1/2)\vec{a}(t)\Delta t \\
\vec{a}(t + \Delta t) &= -(1/m)\Delta\vec{U}(\vec{r}(t + \Delta t)) \\
\vec{v}(t + \Delta t) &= \vec{v}(t + \Delta t/2) + (1/2)\vec{a}(t + \Delta t)\Delta t
\end{aligned}$$

```
!...
! VARIAVEIAS QUE DEVEM ESTAR NO MODULE
real(kind=8), allocatable, dimension(:) :: x,y,z
real(kind=8), allocatable, dimension(:) :: vx,vy,vz
real(kind=8), allocatable, dimension(:) :: ax,ay,az
integer :: dt1
integer :: natom

! VARIAVEIS LOCAIS
real(kind=8), allocatable, dimension(:) :: vnx1,vny1,vnz1
real(kind=8) :: vxcm,vycm,vzcm
integer :: i

do i = 1, natom
   ! POSICAO PARA T+DELTAT
   ! NESTE MOMENTO PERDE-SE AS POSICOES NO TEMPO T
   x(i) = x(i) + vx(i)*dt1 + 0.5d0*ax(i)*dt1*dt1
   y(i) = y(i) + vy(i)*dt1 + 0.5d0*ay(i)*dt1*dt1
   z(i) = z(i) + vz(i)*dt1 + 0.5d0*az(i)*dt1*dt1
   ! VELOCIDADE PARA T+DELTAT/2
   vnx1(i) = vx(i) + 0.5d0*ax(i)*dt1
   vny1(i) = vy(i) + 0.5d0*ay(i)*dt1
   vnz1(i) = vz(i) + 0.5d0*az(i)*dt1
end do

! CALCULA FORCA BASEADA NAS NOVAS POSICOES
! NESTE MOMENTO PERDE-SE AS FORCAS NO TEMPO T
call forca

! CALCULA ACELERACAO BASEADA NAS NOVAS FORCAS
! NESTE MOMENTO PERDE-SE AS ACELERACOES NO TEMPO T
call aceleracao
```

```
33
34 ! CALCULANDO NOVAS VELOCIDADES
35 ! NESTE MOMENTO PERDE-SE AS VELOCIDADES NO TEMPO T
36 do i = 1, natom
37    vx(i) = vnx1(i) + 0.5d0*ax(i)*dt1
38    vy(i) = vny1(i) + 0.5d0*ay(i)*dt1
39    vz(i) = vnz1(i) + 0.5d0*az(i)*dt1
40    ! CALCULANDO AS COMPONENTES DA VEL DO CENTRO DE MASSA
41    vxcm = vxcm + vx(i)
42    vycm = vycm + vy(i)
43    vzcm = vzcm + vz(i)
44 end do
45
46 ! ARTIFICIO UTILIZADO PARA ZERAR O MOMENTO LINEAR
47 do i=1, natom
48    vx(i) = vx(i) - (vxcm/natom)
49    vy(i) = vy(i) - (vycm/natom)
50    vz(i) = vz(i) - (vzcm/natom)
51 end do
52 !...
```

6.6 Condição de Contorno

O último passo que finaliza a construção do progama de DM é a inclusão da condição de contorno, que implica em construir um potencial infinito para que a partícula ou átomo ao ir de encontro à parede da caixa com um velocidade $\vec{v}$ reflita retornando com velocidade $-\vec{v}$.

Poderíamos incluir também outro tipo de condição de contorno simulando um sistema infinito. Esta condição consiste em replicar a caixa infinita vezes em todas as direções, assim um átomo ao chegar em um extremo da caixa consegue *sair* da caixa, mas na verdade ao exceder os limites da caixa a partícula/átomos entra novamente na caixa pela parede oposta na qual ela saiu.

Para resolver o problema da condição da parede da caixa ser um potencial infinito repulsivo (como um parede sólida), se uma das coordenadas exceder os limites da caixa trocamos o sinal da componente da velocidade. Esta solução permite que o átomo reflita ao ser chocar com a parede da caixa sofrendo uma colisão completamente elástica. Devemos prestar a atenção que esta solução não é 100% Física, mas resolve o nosso problema. Esta subrotina deve ser chamada sempre que calcular as componentes das velocidades para o tempo $(t + \Delta t)$.

```
1 !...
2 do i = 1, natom
3    if ( (x(i) + 0.5d0*sigma) > lx    ) vx(i)=-vx(i)
4    if ( (x(i) - 0.5d0*sigma) < 0.d0 ) vx(i)=-vx(i)
```

```
5    if ( (y(i) + 0.5d0*sigma) > ly    ) vy(i)=-vy(i)
6    if ( (y(i) - 0.5d0*sigma) < 0.d0  ) vy(i)=-vy(i)
7    if ( (z(i) + 0.5d0*sigma) > lz    ) vz(i)=-vz(i)
8    if ( (z(i) - 0.5d0*sigma) < 0.d0  ) vz(i)=-vz(i)
9  end do
10 !...
```

6.7 Implementação Extra

Dentro do programa de DM que foi exposto acima poderiam ser feitas várias otimizações e implementação de um potencial mais realístico. Uma das otimizações que é de fácil implementação é a lista de vizinhos, que consiste em guardar os vizinhos em uma lista de tal forma que no cálculo da força e da energia potencial seria feita somente sobre os vizinhos contidos nesta lista. A lista de vizinhos deixa o código mais rápido pois o cálculo da força e energia potencial não é feita verificando todos as partículas/átomos do sistema.

A implementação consiste em verificar inicialmente quais são os vizinhos do átomos i, dentro de um raio de corte maior que o raio de corte do potencial, afim de garantir a inclusão dos átomos dentro do raio de corte do potencial. Esta lista é atualizada somente dentro de um intervalo a ser definido. A lista de vizinhos baseia-se na idéia de que a difusão dos átomos é lenta de tal maneira que no intervalo definido a vizinhança não se alterara, ou seja, fica constante. Como iremos trabalhar com um número pequeno de partículas/átomos não será necessário a implementação, mas para sistemas com grandes números de partículas/átomos é necessário a implementação ou se desejar fique a vontade para implementar e tirar dúvidas.

Capítulo 7

Noções básicas do método Monte Carlo Clássico - Modelo de Ising

7.1 Modelo de Ising

O modelo de Ising é um dos mais simples e mais estudado dos modelos de mecânica estatística. Nesta parte olharemos em detalhe o método de Monte Carlo, que têm sido utilizados para investigar as propriedades deste modelo. Durante o desenvolvimento iremos comentar sobre alguns truques utilizados para implementação dos algoritmos de Monte Carlo em códigos computacionais e algumas técnicas utilizadas para analisar os dados gerados por esses programas.

O modelo de Ising é um simples modelo de magnetismo, em que dipolos ou "*spins*" são colocados no sítios de uma rede. Cada *spin* pode assumir qualquer um dos valores: +1 e −1. Se tivermos N sítios na rede, então o sistema pode ser $2N$ estados, e a energia de um estado particular é dada pelo hamiltoniano de Ising:

$$H = -J \sum_{\langle ij \rangle} s_i s_j - B \sum_i s_i \tag{7.1.1}$$

onde J é uma energia de interação entre o primeiro *spin* vizinho $<\boldsymbol{ij}>$, e B é um campo magnético externo. Nós estamos interessados em simular um sistema Ising de tamanho finito usando um método de Monte Carlo, para que possamos estimar os valores das quantidades, como a magnetização média por *spin* $\langle m \rangle = \frac{1}{N}\langle \sum_i s_i \rangle$ ou o calor específico $c = \frac{k\beta^2}{N}(\langle E^2 \rangle - \langle E \rangle^2)$, em qualquer temperatura dada. A maioria das questões interessantes sobre o modelo de Ising podem ser respondidas através de simulações na ausência do campo magnético ($B = 0$), no qual nós iremos nos dedicar.

7.2 Algoritmo de Metropolis

O primeiro algoritmo de Monte Carlo que veremos é o mais famoso e amplamente utilizado algoritmo de todos eles, o algoritmo de Metropolis, que foi introduzido por Nicolas Metropolis e seus colaboradores em um artigo de 1953 em simulações de gases de esferas rígidas (Metropolis et al. 1953). Utilizarremos esse algoritmo para ilustrar alguns conceitos gerais envolvidos em um cálculo de Monte Carlo, incluindo o equilíbrio, medida do valor esperado e o cálculo de erros. Entretanto, vamos detalhar o algoritmo e estudar como implementá-lo em um código computacional.

O algoritmo de Metropolis segue o seguinte esquema: escolhemos um conjunto de probabilidades de seleção $g(\mu \to \nu)$, uma para cada possível transição de um estado para outro, $\mu \to \nu$, então escolhemos um conjunto de probabilidades de aceitação $A(\mu \to \nu)$ tal que

$$\frac{P(\mu \to \nu)}{P(\nu \to \mu)} = \frac{g(\mu \to \nu)A(\mu \to \nu)}{g(\nu \to \mu)A(\nu \to \mu)} \tag{7.2.1}$$

em que
$\frac{A(\mu \to \nu)}{A(\nu \to \mu)}$ pode assumir qualquer valores entre 0 e ∞;
e
$g(\mu \to \nu)$ e $g(\nu \to \mu)$ pode assumir qualquer valor desejado.

A Eq. 2 satisfaz a condição de balanço detalhado dado pela equação

$$\frac{P(\mu \to \nu)}{P(\nu \to \mu)} = \frac{p_\nu}{p_\mu} = e^{-\beta(E_\nu - E_\mu)} \tag{7.2.2}$$

O algoritmo trabalha repetindo a escolha de um novo estado ν aleatoriamente, aceitando ou rejeitando esse novo estado de acordo com a escolha de aceitação probabilística. Se o estado é aceito, o estado que antes era μ passa agora a ser o estado μ, caso contrário permanece como está. Assim o processo é repetido por várias vezes.

As probabilidades de seleção $g(\mu \to \nu)$ devem ser escolhidas de modo que a condição de ergodicidade (exigência de que cada estado estará acessível a partir de todos os outros em um número finito de passos) seja cumprida (*A condição de ergodicidade é a exigência que deve ser possível para os processos de Markov alcançar qualquer estado do sistema a partir de qualquer outro estado, se executado em um tempo suficiente longo. Isso é necessário para alcançar o objetivo de gerar estados com corretas probabilidades Boltzmann. Cada estado aparece ν com alguma probabilidade p_ν, diferente de zero, na distribuição de Boltzmann, e se esse estado estiver inacessível a partir de outro estado μ, não importando quanto tempo ainda continuamos o processo para, em seguida, iniciarmos do estado μ: a probabilidade de encontrar ν nos estados das cadeias de Markov será zero, e não p_ν A condição de ergodicidade nos diz que estamos autorizados a fazer algumas transições de probabilidades a partir do processo de Markov zero, mas deve haver pelo menos um caminho diferente de*

zero nas transições de probabilidades entre dois estados escolhidos. Na prática, a maioria dos algoritmos de Monte Carlo define quase todas as probabilidades de transição como sendo zero, e devemos ter cuidado para que ao fazê-lo não criarmos um algoritmo que viola a ergodicidade. Para os algoritmos desenvolvidos deve-se provar explícitamente que ergodicidade está sendo satisfeita antes de uso do algoritmo na producao de resultados serios). Isto ainda nos deixa uma grande margem sobre a forma de como a passagem de um estado para outro será escolhido e dado um estado inicial μ podemos gerar qualquer número de estados candidatos ν simplesmente girando diferentes grupos de *spins* da rede. As energias dos sistemas em equilíbrio térmico permanecem dentro de um intervalo muito estreito de energia, as flutuações de energia são pequenas em comparação com a energia de todo o sistema. Em outras palavras, o sistema real passa a maior parte de seu tempo em um subconjunto de estados com uma estreita faixa de energias e raramente faz transições que mudam a energia do sistema de forma drástica. Isso diz que provavelmente não queremos gastar muito tempo em nossa simulação considerando as transições para os estados cuja energia é muito diferente da energia do estado atual. A forma mais simples de conseguir isto no modelo de Ising é considerar apenas os estados que diferem de um dos presentes pelo giro de um único *spin*. Um algoritmo que realiza este tipo de giro de *spin* é dito como tendo um dinâmica *single-spin-flip*. O algoritmo que descreveremos tem dinâmica *single-spin-flip*, embora isto não seja realizado no algoritmo de Metropolis. Como discutido abaixo, é a escolha particular da relação de aceitação que caracteriza o algoritmo de Metropolis. Nosso algoritmo continuaria a ser um algoritmo de Metropolis, mesmo que girasse várias vezes os vários *spins* de uma só vez.

Usando a dinâmica *single-spin-flip* garante que o novo estado ν vai ter uma energia E_ν diferenciando-se da energia corrente E_μ de até, no máximo, $2J$ para cada ligação entre o *spin flipado* e seus vizinhos. Por exemplo, em uma rede quadrada em duas dimensões cada *spin* tem quatro vizinhos, assim a máxima diferença de energia seria $8J$. A expressão geral $2zJ$, onde z é o número de coordenação da rede, ou seja, o número de vizinhos em que cada ponto da rede possui (*isto não é a mesma coisa que o "o número de coordenação de spin". O número de coordenação de spin é o número de spins j na vizinhança de i que possui o mesmo valor do spin i:* $s_i = s_j$). Usando uma única dinâmica *single-spin-flip* também garante que o algoritmo obedece a ergodicidade, desde que esteja claro que seja possível pegar um estado a partir do outro na rede finita *flipando* os *spins* um por um, onde há diferença.

No algoritmo de Metropolis as probabilidades de seleção $g(\mu \to \nu)$ para cada um dos possíveis estados μ são escolhidos para serem iguais. As probabilidades de seleção de todos os outros estados são definidos como sendo zero. Suponha que há N *spins* no sistema que desejamos simular. Com uma única dinâmica *single-spin-flip*

teremos então N diferentes *spins* que poderão ser *flipados* e, portanto, N estados ν que podem alcançar um determinado estado μ. Assim, há N probabilidades de seleção $g(\mu \to \nu)$ que serão diferentes de zero, e cada um deles assumirá o valor

$$g(\mu \to \nu) = \frac{1}{N}$$

Com estas probabilidades de seleção, a condição de balanço detalhado, a Eq. 3, assume a forma

$$\frac{P(\mu \to \nu)}{P(\nu \to \mu)} = \frac{g(\mu \to \nu)A(\mu \to \nu)}{g(\nu \to \mu)A(\nu \to \mu)} = \frac{A(\mu \to \nu)}{A(\nu \to \mu)} = e^{-\beta(E_\nu - E_\mu)}$$

(3a)

Agora temos que escolher as taxas de aceitação $A(\mu \to \nu)$ para satisfazer a equação. Uma possibilidade seria escolher

$$A(\mu \to \nu) = A_0 e^{\frac{-1}{2}\beta(E_\nu - E_\mu)} \tag{7.2.3}$$

A constante de proporcionalidade A_0 não está presente na Eq. (4), podemos escolher qualquer valor para esta constante, exceto que $A(\mu \to \nu)$, sendo uma probabilidade, nunca deve se tornar maior do que A_0. A maior diferença de energia $E_\nu - E_\mu$ que podemos ter entre dois estados é $2zJ$, onde z é o número de coordenação da rede. Isso significa que o maior valor de $e^{\frac{-1}{2}\beta(E_\nu - E_\mu)}$ é $e^{\beta zJ}$. Assim, a fim de certificar-se de que $A(\mu \to \nu) \leq 1$, escolhemos

$$A_0 \leq e^{-\beta zJ} \tag{7.2.4}$$

Para tornar o algoritmo mais eficiente possível, fazemos as probabilidades de aceitação tão grande quanto possível, de modo que A_0 seja tão grande quanto é permitido ser, o que nos dá

$$A(\mu \to \nu) = A_0 e^{\frac{-1}{2}\beta(E_\nu - E_\mu + 2zJ)} \tag{7.2.5}$$

Este ainda não é o algoritmo de Metropolis, mas utilizando essa probabilidade de aceitação, podemos realizar uma simulação de Monte Carlo do modelo de Ising, sendo uma amostragem correta da distribuição de Boltzmann. No entanto, a simulação será muito ineficiente, pois a relação de aceitação, a Eq. (6), será muito pequena para quase todos os movimentos.

Na Eq. 4, foi assumido uma forma funcional particular para o índice de aceitação, mas a condição do balanço detalhado, Eq. 3a, na verdade não requer que seja dessa forma. A Eq. 3a especifica apenas a relação entre pares de probabilidades de aceitação, deixando muitas possibilidades de manobra. Quando há uma restrição do tipo da Eq. 3a a maneira de maximizar as taxas de aceitação (e portanto, produzir um algoritmo mais eficiente) é sempre dar para o maior dos dois índices, o maior

valor possível, ou seja, 1, e em seguida, ajustar o outro para satisfazer a restrição. Para ver como isso funciona, suponha que os dois estados μ e ν, μ tenha a menor energia e ν a maior energia: $E_\nu < E_{mu}$. Assim a maior das duas taxas de aceitação $A(\nu \to \mu)$, então ajustamos o conjunto para ser igual a 1. A fim de satisfazer a Eq. 3a, $A(\mu \to \nu)$ deve assumir o valor $e^{-\beta(E_\nu - E_\mu)}$. Assim, o algoritmo otimizado será:

$$A(\mu \to \nu) = \begin{matrix} e^{-\beta(E_\nu - E_\mu)} & \to se E_\nu - E_\mu > 0 \\ 1 & \to \textit{para outros casos} \end{matrix} \tag{7.2.6}$$

Em outras palavras, se selecionar um novo estado que tem uma energia inferior ou igual ao atual, sempre devemos aceitar a transição para esse estado. Se o estado tiver uma energia mais elevada, talvez pode ser aceita com a probabilidade dada acima. Este é o algoritmo de Metropolis para o modelo de Ising com uma única dinâmica *single-spin-flip*. Esta é a parte pioneira que abriu caminho para Metropolis e colaboradores em seu artigo gases de esferas rígidas, podendo ser aplicado a qualquer modelo de acordo com a Eq. 7.

7.3 Implementando o Algoritmo de Metropolis

Veremos agora como escrever um código computacional para realizar simulações do modelo de Ising utilizando o algoritmo de Metropolis. Para facilitar continuaremos utilizando o caso com o campo magnético $B = 0$, embora a generalização para o caso $B \neq 0$ não seja difícil. Vale lembrar que quase todos os estudos anteriores do modelo de Ising, incluindo a solução exata de Onsager em duas dimensões, foram para sistemas com $B = 0$.

Inicialmente, precisamos de uma rede de *spins*, então definimos um conjunto de N variáveis, uma matriz, que pode assumir valores ± 1. Assim construiremos uma matriz que possuem somente números inteiros contendo valores +1 e −1. Normalmente, é aplicada condições periódicas de contorno à matriz, isto é, impõe-se que os *spins* de uma extremidade da rede são vizinhos dos *spins* da outra ponta da rede, semelhante o teorema de Bloch. Isso garante que todas os *spins* têm o mesmo número de vizinhos e uma geometria local, e que não há nenhum *spin* com propriedades diferentes um das dos outros; todos os *spins* são equivalentes e o sistema é completamente invariante sobre translação. Na prática, isso melhora consideravelmente a qualidade dos resultados da simulação.

Uma variação sobre a ideia de condição periódica de contorno é usar a "condição de contorno helicoidal", que é ligeiramente diferente da condição periódica de contorno tradicional, possui todos os mesmos benefícios, é consideravelmente mais simples de implementar e pode tornar o código significativamente mais rápido.

A seguir deveremos decidir em qual temperatura, ou alternativamente, qual o valor de β que desejaremos realizar a simulação, e será necessário atribuir um valor inicial para cada *spin* – que será o estado inicial do sistema. Em muitos casos, o

estado inicial que é escolhido não é particularmente importante, embora, às vezes, uma escolha criteriosa pode reduzir o tempo necessário para chegar ao equilíbrio. Os dois estados iniciais mais utilizados são o estados de temperatura zero e o estado de temperatura infinita. Em $T = 0$ o modelo de Ising estará em seu estado fundamental. Quando a energia de interação J é maior que zero e o campo externo B é zero (como serão nos casos das nossas simulações), há na verdade dois estados fundamentais. Estes estados são os casos em que os *spins* são todos para cima ou todos para baixo. É fácil ver que estes estados devem ser os fundamentais, uma vez que nesses estados cada par de *spins* contribui para a menor energia possível ($-J$) no primeiro termo do Hamiltoniano da Eq. 1. Em qualquer outro estado, haverá pares de *spins* que contribuem com energia $+J$ para o hamiltoniano, de modo que seu valor total será maior. Se $B \neq 0$ haverá apenas um estado fundamental, o campo magnético garante que apenas um dos dois estados fundamentais seja favorecido em relação um ao outro. O outro estado inicialcomumente utilizado é o estado $T = \infty$. Quando $T = \infty$ a energia térmica disponível kT para *flipar* o *spin* é infinitamente maior do que a energia de interação *spin-spin* J, então os *spins* são orientados para cima ou para baixo de maneira randômica de forma a ficarem não correlacionados.

As duas opções do estado inicial são populares porque correspondem a uma temperatura conhecida e bem definida e facilmente de serem geradas. Há no entanto, um outro estado inicial, que às vezes pode ser muito útil. Muitas vezes não basta realizar uma simulação em uma única temperatura, para isto é realizado um conjunto de simulações para diferentes valores de T, no intuito de investigar o comportamento do modelo em função da variação de temperatura. Neste caso leva-se vantagem escolher como o estado inicial do sistema o estado final, para uma simulação a uma temperatura próxima. Por exemplo, suponha que estamos interessados em investigar uma gama de temperaturas entre $T = 1,0$ e $T = 2,0$ em passos de $0,1$. (Aqui referimos-nos à temperatura em unidades de energia de modo que $k = 1$. Assim, quando dizemos que $T = 2,0$ queremos dizer $\beta^{-1} = 2,0$). Então, poderíamos iniciar a simulação em $T = 1,0$ usando o estado da temperatura zero com todos os *spins* alinhados como o estado inicial. Ao final da simulação, o sistema estará em equilíbrio a $T = 1,0$, assim poderemos utilizar o estado final da simulação como a entrada do estado inicial da simulação em $T = 1,1$, e assim por diante.

Iniciando a simulação, o primeiro passo é gerar um novo estado. O novo estado deve ser diferente do atual apenas pelo *flip* de um *spin*, e cada estado deve ser exatamente tão provável como qualquer outro a ser gerado. Esta é uma tarefa fácil de realizar. Pegaremos um único *spin* k aleatoriamente na rede para ser *flipado*. Em seguida calcularemos a diferença de energia $E_\nu - E_\mu$ entre o estado novo e o antigo, a fim de aplicar a Eq. 7. A maneira simples de realizar o cálculo da energia é calcular E_μ diretamente substituindo os valores de *spins* (s_i^μ) do estado μ no Hamiltoniano (Eq. 1), então *flipar* o *spin* k e calcular E_ν, obter a diferença. No entanto, não

é a maneira mais eficiente de realizar esta tarefa. Mesmo com $B = 0$, é necessário realizar a soma do primeiro termo da Eq. 1, que tem tantos termos quantos os termos de conexões da rede, que é $1/2Nz$. Porém, a maioria desses termos não se alteram quando é realizado apenas um *flip* de *spin*. Os únicos termos que se alteram são os que envolvem o *flip* de *spin*. Os outros termos permanecem com o mesmo valor e se cancela quando tomamos a diferença $E_\nu - E_\mu$. A mudança na energia entre dois estados será:

$$E_\nu - E_\mu = -J \sum_{\langle ij \rangle} s_i^\nu s_j^\nu - J \sum_{\langle ij \rangle} s_i^\mu s_j^\mu$$

$$E_\nu - E_\mu = -J \sum_{in.n.para k} s_i^\mu (s_k^\nu - s_k^\mu) \tag{7.3.1}$$

Na segunda linha a soma é apenas para os *spins* i que são os primeiros vizinhos do *spin flipado* k e usamos o fato de que todas esses *spins* não se invertem, de modo que $(s_i^\nu - s_i^\mu)$. Se $s_k^\mu = +1$, então após o *spin* k ter sido *flipado* deveremos ter $s_k^\nu = -1$, de modo que $s_k^\nu - s_k^\mu = -2$. Por outro lado, se $s_k^\mu = -1$ então $s_k^\nu - s_k^\mu = +2$. Assim, podemos escrever

$$s_k^\nu - s_k^\mu = -2 s_k^\mu \tag{7.3.2}$$

e então

$$E_\nu - E_\mu = 2J \sum_{in.n.para k} s_i^\mu s_k^\mu$$

$$E_\nu - E_\mu = 2J s_k^\mu \sum_{in.n.para k} s_i^\mu \tag{7.3.3}$$

Esta expressão envolve apenas a soma sobre termos z, ao invés de $1/2Nz$, e não será necessário realizar multiplicações para os termos na soma, por isso é muito mais eficiente do que avaliar diretamente a variação na energia. Isto envolve apenas os valores dos *spins* no estado μ, avaliando previamente antes de *flipar* realmente o *spin* k.

O algoritmo consiste em calcular $E_\nu - E_\mu$ da Eq. 10 e em seguida pela regra da Eq. 7: se $E_\nu - E_\mu \leq 0$ aceita-se a transição e *flipando* $s_k \to -s_k$. Se $E_\nu - E_\mu > 0$ poderíamos, ou não, *flipar* o *spin*. Com o algoritmo de Metropolis o *flip* do *spin* será dado pela probabilidade $A(\nu \to \mu) = e^{-\beta(E_\nu - E_\mu)}$. Podemos fazer isso da seguinte forma: avaliamos a relação de aceitação $A(\nu \to \mu)$, usando o valor de $E_\nu - E_\mu$ da Eq. 10 , então escolhe-se um número aleatório r entre zero e um. A rigor, o número pode ser igual a zero, mas deve ser inferior a um ($0 \leq r < 1$). Se esse número for menor do que a nossa relação de aceitação, $r < A(\nu \to \mu)$, então o *spin* é *flipado*, caso contrário deixa-se como está.

Essa é sequência completa do algoritmo. Agora basta repetir os mesmos cálculos, escolhendo um *flip* e calculando a variação da energia, então decidindo se ocorrerá ou não o *flip* de acordo com a Eq. 7. Na verdade, há um outro truque que pode deixar o algoritmo um pouco mais rápido. Uma das partes mais lentas do algoritmo é o cálculo da exponencial, pois calcula-se a energia do novo estado, faz-se a comparação de energia e só então *flipa* ou não o *spin*. O cálculo de exponenciais em um computador é feito geralmente utilizando uma aproximação polinomial, que envolve a execução multiplicações de ponto flutuante e somatórios, o que pode levar um intervalo de tempo considerável. É possível contornar este esforço e tornar o código computacional mais rápido ao verificar que a quantidade da Eq. 10 que esta sendo calculada para uma pequena quantidade de valores. Cada termo da soma só pode assumir valores +1 e −1. Então, todo o somatório, que tem z termos, só pode ter valores $-z$, $-z+2$,$-z+4$ e assim por diante até $+z$, um total de $z+1$ valores possíveis. Só precisamos então calcular o exponencial quando a soma for negativa (Eq. 7), de fato temos apenas $1/2z$ valores de $E_\nu - E_\mu$ no qual é necessário o cálculo das exponenciais. Assim, utilizando o bom senso, faz todo o sentido calcular os valores dessas $1/2z$ exponenciais antes de iniciar o cálculo, e armazená-los na memória do computador, em que pode ser feito uma pesquisa durante o andamento da simulação. O cálculo é realizado apenas uma vez no início não sendo necessário calcular novamente a exponencial durante a simulação. O único cálculo necessário de ponto flutuante será na geração do número aleatório r, todos os outros cálculos envolvem apenas números inteiros, que é mais rápido que o tratamento com números reais.

7.4 Equilíbrio

O que fazer o programa de Monte Carlo baseado no modelo de Ising ? Gostaríamos de responder algumas perguntas como “Qual é a magnetização em uma data temperatura ?”, ou “Como é que a energia interna se comporta em função da variação da temperatura ?” Para responder a essas questões que teremos que realizar dois processos: primeiro temos que executar a simulação para um intervalo de tempo suficientemente longo de tempo até o sistema atingir o equilíbrio para a temperatura desejada, este intervalo de tempo é chamado de tempo de equilíbrio τ_{eq}, e então medir a propriedade de interesse para vários intervalos de tempos suficientemente longos tomando a média dessa propriedade. Isso nos leva à várias outras questões. O que exatamente queremos dizer com “ o sistema entrar em equilíbrio ?” E qual a quantidade de tempo para um intervalo de tempo suficientemente longo” para que isso aconteça ? Como mediremos a propriedade de interesse, e quantos intervalos de tempo suficientemente longo necessitamos para calcular a média da propriedade de interesse a fim de obter um resultado com um determinado grau de precisão ? Estas questões gerais devem ser consideradas toda vez que é realizado um cálculo

de Monte Carlo. Embora a discussão destas questões serão sobre simulações com o modelo de Ising, as conclusões são aplicáveis a todos os outros cálculos de equilíbrio de Monte Carlo.

"Equilíbrio" significa que a probabilidade média de encontrar o nosso sistema em um determinado estado qualquer μ é proporcional ao peso de Boltzmann $e^{-\beta E\mu}$ do estado. Se iniciarmos nosso sistema com estados como a $T = 0$ ou $T = \infty$, como descritos anteriormente, levará alguns passo até alcançar o equilíbrio. Lembre-se que um sistema em equilíbrio passa a esmagadora maioria do seu tempo em um pequeno subconjunto de estados em que sua energia interna e outras propriedades assume uma estreita faixa de valores. A fim de obter uma boa estimativa do valor de equilíbrio de qualquer propriedade do sistema será necessário aguardar até que sistema encontre o seu caminho em um dos estados que se encaixam nesta estreita faixa de valores.

Na versão do algoritmo de Metropolis que que foi descrita aqui, o *flip* do *spin* é realizado um de cada vez (escolhido aleatóriamente) o que poderá levar um grande número de passos até obter a sequência correta de *spins* up e down que minimiza a energia. Esperamos utilizar N passos de Monte Carlo até alcançar a energia correta, em que N é números de *spins* da rede, lembrando que devemos dar a chance de *flip* para todos os *spins* da rede. Como exemplo consideremos uma rede de 100×100 *spins*, assumindo $J = 1$, partindo do estado inicial $T = 0$ aquecemos o sistema até a $T = 2,4$, para que o sistema atinja o equilíbrio serão necessários aproximadamente 10^7 passos.

Mesmo com esta quantidade de passos é necessário verificar as propriedades para ver se realmente o sistema está no equilíbrio. Uma maneira fazer um gráfico da magnetização por *spin* m do sistema ou da energia do sistema E, em função dos passos da simulação. A energia de um determinado estado pode ser calculado utilizando todos os valores dos *spins* s_i no Hamiltoniano da Eq. 1. Com o gráfico é fácil perceber quando o perfil da curva alcança a estabilidade, o que significa que a energia e a magnetização não variam além de um certo limite flutuando apenas em torno de um valor médio constante.

Analisando o equilíbrio de um sistema pelo "olhômetro" em um gráfico pode até ser um método razoável, desde que saibamos que o sistema irá entrar em equilíbrio de uma forma suave e previsível. O grande problema é que nem sempre conhecemos o comportamento do sistema até alcançar o equilíbrio. Em muitos casos é possível que o sistema fique preso em algum mínimo local por um certo tempo, assim o gráfico apresenta valores constantes para todas as quantidades que estamos observando parecendo ter atingido o equilíbrio. É possível haver um mínimo de energia local em que o sistema permanece temporariamente, parecendo ter atingido o mínimo global de energia, que é a região do espaço de estado que o sistema de equilíbrio é mais provável. Para evitar essa armadilha, podemos realizar duas simulações diferentes

de um mesmo sistema, partindo de dois estados iniciais um no estado $T = 0$ com todos os *spins* alinhados e outro estado $T = \infty$ com todos os *spins* aleatórios. Outra possibilidade é escolher dois diferentes estados $T = \infty$ (sementes do número aleatório diferentes). Em seguida, observamos o valor da magnetização ou da energia nos dois sistemas, ao estabilizar o perfil da curva e ambas simulações apresentarem valores semelhantes podemos assumir que ambos os sistemas atingiram o equilíbrio.

7.5 Medições

Uma vez que temos a certeza do equilíbrio ter sido atingido, é preciso medir a propriedade que estamos interessado. As propriedades que estamos interessados são a energia e a magnetização do sistema. A energia E_μ do estado μ pode ser calculada diretamente a partir do Hamiltoniano, através dos valores dos *spins* s_i, a partir da matriz de inteiros. A maneira de fazer isso é lembrar que a diferença de energia entre os estados ν e μ é dada por $\Delta E = E_\nu - E_\mu$, de acordo com a equação Eq. 10. Com isso, se sabemos que a energia do atual estado μ, podemos calcular a energia do novo estado ν fazendo a soma:

$$E_\nu = E_\mu + \Delta E \tag{7.5.1}$$

O que pode ser feito é calcular a energia inicial do sistema no início da simulação e em seguida calcular a nova energia quando o *spin* é *flipado* para cada passo da simulação.

O cálculo da magnetização é ainda mais fácil. A magnetização total M_μ de todo o sistema no estado μ, é dada por:

$$M_\mu = \sum_i s_i^\mu \tag{7.5.2}$$

Apesar da magnetização total ser dada pela equação acima, a maneira mais rápida não é utilizando ela. Lembremos que apenas um *spin k flipa* em um passo do algoritmo de Metropolis assim a diferença de magnetização de uma estado μ para um estado ν é dada por:

$$\Delta = M_\nu - M_\mu = \sum_i s_i^\nu - \sum_i s_i^\mu = s_k^\nu - s_k^\mu = 2s_k^\nu \tag{7.5.3}$$

em que o último termo é dada pela Eq. 9. Então calcula-se a magnetização no início da simulação e durante a evolução do sistema (para cada *flip* de *spin*) utiliza-se a equação

$$M_\nu = M_\mu + \Delta M = M_\mu + 2s_k^\nu \tag{7.5.4}$$

Com a energia e a magnetização do sistema no decorrer da evolução da simulação, podemos fazer médias dos valores podemos encontrar a energia interna e a

magnetização. Em seguida, dividindo-se as médias obtidas pelo número de sítios N teremos a energia interna e magnetização por sítio.

Com as quantidades das propriedades é possível calcular também a média dos quadrados da energia e da magnetização para encontrar outras quantidades como o calor específico e susceptibilidade magnética:

$$c = \frac{\beta^2}{N}(\langle E^2\rangle - \langle E\rangle^2) \tag{7.5.5}$$

$$\chi = \beta N(\langle m^2\rangle - \langle m\rangle^2) \tag{7.5.6}$$

Capítulo 8

Complementos

8.1 Tópico avançados em Física

8.2 Implementação avançada em Fortran 90

Capítulo 9

Problemas

9.1 Problema #1

Considerando o *array* unidimensional composto por números inteiros ($a(1) = 1$, $a(2) = 111$, $a(3) = 977$, $a(4) = 41$, $a(5) = 91$, $a(6) = 328$, $a(7) = 7$, $a(8) = 33$, $a(9) = 57$, $a(10) = 271$), crie um programa que leia esses valores de um arquivo de entrada e atribua à variável indexada (a_i) e ordene em ordem crescente os valores das variáveis indexadas e depois escreva em um outro arquivo de saída. A idéia é que as variáveis indexadas e os valores das variáveis indexadas fiquem ordenadas em ordem crescente conforme a tabela 9.1. Este programa é possível de ser construído utilizando um comando *do while*, um comando *do*, um comando *if*, cinco (5) variáveis inteiras e um *array* unidimensional com dez posições. O programa e o arquivo de entrada deve ser entregue via e-mail na data combinada.

O que será considerado na avaliação deste programa: comentários dentro do programa (data de criação, data das alterações realizadas, para que o programa serve, limites do programa, etc.), a utilização dos comandos que foram apresentados até o momento, a utilização do comando *goto* automaticamente zera esta avaliação, o número de linhas utilizadas no programa descontando os comentários, o número de variáveis utilizadas no programa, a existência ou não da indentação, a utilização ou não da formatação para o arquivo de saída.

a(1) = 1	a(1) = 1
a(2) = 111	a(2) = 7
a(3) = 977	a(3) = 33
a(4) = 41	a(4) = 41
a(5) = 91	a(5) = 57
a(6) = 328	a(6) = 91
a(7) = 7	a(7) = 111
a(8) = 33	a(8) = 271
a(9) = 57	a(9) = 328
a(10) = 271	a(10) = 977

Tabela 9.1: *Array* desordenado e ordenado em ordem crescente.

9.2 Problema #2

Podemos calcula os valores de π, *Seno* e *Coseno* de diversas maneiras, para exemplificar temos nas equações 9.2.1, 9.2.2 e 9.2.3 a forma em termos de somatórios que nos permite uma implementação numérica do cálculo dessas quantidades.

Escreva um programa ou programas que calcule o valor de π, *Seno* e *Coseno*. Para π construa o cálculo no corpo do programa principal e para o *Seno* e *Coseno* construa os cálculos como uma *function*. A precisão simples trabalha com 6 casas decimais e a precisão dupla com 15 casas decimais, baseado nesta informação ajuste o valor de n que melhor se adequa às precisões, ou seja, qual o valor de n que deve ser utilizado quando estiver trabalhando com simples ou dupla precisão. Vale lembrar que para o cálculo de *Seno* e *Coseno* o valor de x deve ser em radianos, assim deve-se fazer a conversão $x = x * \pi/180.0d0$ antes do somatório.

$$\pi = (32 * S)^{1/3}, \text{ sendo: } S = \sum_{i=1}^{n} \frac{(-1)^{i+1}}{(2i-1)^3} \tag{9.2.1}$$

$$Seno(x) = \sum_{i=0}^{n} \frac{-1^i}{(2i+1)!} x^{2i+1} \tag{9.2.2}$$

$$Coseno(x) = \sum_{i=0}^{n} \frac{-1^i}{(2i)!} x^{2i} \tag{9.2.3}$$

O critério de correção será baseado no esquema do problema 9.1.

Fernando Sato

Bibliografia

[1] Cláudio Scherer, *Métodos Computacionais da Física*, Editora: Editora Livraria da Física, ISBN: 85-88325-35-7.

[2] Steven E. Koonin and Dawn C. Meredith, Computational Physics - Fortran Fersion, Westview Press, 1990. ISBN: 0201127792, ISBN: 0201386232.

[3] Nicholas J. Giordano, *Computational Physics*, Editora: Prentice Hall, ISBN: 0-13-367723-0.

[4] Ronaldo L. D. Cereda e José Carlos Maldonado, *Introdução ao FORTRAN para microcomputadores*, Editora: McGraw-Hill.

[5] Stephen J. Chapman, *Introduction To Fortran 90/95*, First Edition, WCB McGraw-Hill, 1998. ISBN 0-07-011969-4

[6] Gleydson M. da Silva, Guia Foca GNU/Linux, Vol. 1 (iniciante), Vol. 2 (intermediário) e Vol. 3 (avançado). Sítio *http://focalinux.cipsga.org.br/*.

[7] The Linux Documentation Project, *http://tldp.org*.

[8] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, and Michael Metcalf, Numerical Recipes in Fortran 90, Vol. 2, Cambridge University Press; 2 edition (January 15, 1996). ISBN 0521574390; ISBN-13 978-0521574396.

[9] J. W. Cooley and J. W. Tukey, An algorithm for the machine calculation of complex Fourier series, *Math. Comput.* **19**, 297–301 (1965). doi:10.2307/2003354

[10] C. S. Burrus, M. Frigo, S. G. Johnson, M. Pueschel, I. Selesnick, Fast Fourier Transform, Rice University - Huston - Texas, http://cnx.org/content/col10550/1.18/, livro em pdf.

[11] Ver documentação do sítio oficial da fftw: http://www.fftw.org.

[12] L. Verlet, Computer Experiments on Classical Fluids. I. Thermodinamical Properties of Lennard-Jones Molecules, *Phys. Rev.* **159**, 98-103 (1967)

[13] L. Verlet, *Phys. Rev.* **165**, 201 (1967).

[14] H. C. Andersen, *J. Comp. Phys.* **52**, 24 (1983).

[15] A. Isaacs, *A Dictionary of Physics*, Oxford University Press Inc., New York,2003.

[16] G. Keserü and I. Kolossváry, *Molecular Mechanics and Conformational Analisys in Drug Design*, Blackwell Science Ltd, Osney Mead, Oxford OX2 0EL, 1-168, 1999, www.blackwell-science.com.

[17] L. Kale, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Philips, A. Shinozaki, K. Varadarajan, K. Shulten, *J. Comp. Phys.*, **151**, 283-312(1999)

[18] A.K. Rappé, C.J. Casewit, K.S. Colwell, W.A. Goddard III, and W.M. Skiff, *J. Am. Chem. Soc.*, **114**, 10024-10035 (1992).

[19] J. M. Haile, *Molecular Dynamics Simulation - Elementary Methods*, John Willey & Sons, Inc., New York, 1992.

[20] D. C. Rapaport, *The Art Of Molecular Dynamics Simulation*, Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 2001.

[21] F. Ercolessi, *A Molecular Dynamics Primer*, http://www.ud.infn.it/ ercolessi/md/md/, http://www.freescience.info/books.php?id=225, 1997.

[22] M. P. Hodges, *XMakemol*, Progama Licenciado por GNU General Public License de acordo com a Free Software Foundation - FSF. Pode ser encontrado facilmente na rede muldial de computadores. Nas distribuções Debian e Ubuntu estão inclusos dentro dos pacotes oficiais.

[23] www.ks.uiuc.edu/Research/vmd/ .

[24] L. Laaksonen: A graphics program for the analysis and display of molecular dynamics trajectories. *J. Mol. Graphics* 10 (1992) 33; D.L. Bergman, L. Laaksonen and A. Laaksonen: Visualization of Solvation Structures in Liquid Mixtures. *J. Mol. Graphics & Modelling* 15 (1997) 301. URL: http://www.csc.fi/gopenmol/.