UNIVERSIDADE FEDERAL DO TRIÂNGULO MINEIRO
CAMPUS UNIVERSITÁRIO DE ITURAMA

Introdução à programação em Fortran 90

João Vitor Teodoro
joao.magda@gmail.com

Este material tem por objetivo apresentar conceitos básicos e introdutórios à programação em linguagem FORTRAN 90. Os códigos e procedimentos descritos representam uma gama de opções a serem adotadas, sendo assim, podem existir outras formas e comandos alternativos que aqui não são citados.

Com este material, é possível capacitar-se à resolução de problemas básicos em computação científica em que, apesar de exemplos e exercícios de grande simplicidade e de fácil entendimento, o leitor pode adaptar seus problemas mais complexos ao que é apresentado. Porém, de acordo com o rumo e necessidade para cada programa, deve-se buscar referências mais ricas e específicas em determinados procedimentos, além da importância em buscar metodologias eficientes.

O texto apresentado é baseado em vários materiais e, pretendeu-se reunir de forma fácil, compreensível e sucinta o que cada um oferece de melhor. Exemplos prontos para execução e descrições dos procedimentos passo-a-passo foram colocados buscando a melhor compreensão desde aqueles que iniciam na área de programação.

Por estar em uma primeira edição, o material pode conter erros e necessitar de melhorias, assim, quaisquer sujestões e comentários são bem-vindos.

João Vitor Teodoro

SUMÁRIO

1 Introdução

Nos primórdios dos computadores, programar era uma tarefa extremamente complicada e, de certa forma, extenuante. Aos programadores era exigido um conhecimento detalhado das instruções, registos e outros aspectos ligados com a unidade de processamento central (CPU) do computador onde era escrito o código. Os programas consistiam numa série de instruções numéricas, denominadas por código binário. Posteriormente, desenvolveram-se algumas mnemónicas (auxiliares de memória) que resultaram no designado assembly (notação legível por humanos para o código de máquina). No período entre 1954-1957 uma equipe de 13 programadores liderados por John Backus desenvolveu uma das primeiras linguagens de alto nível para o computador IBM 704, o FORTRAN (FORmula TRANslation). O objetivo deste projeto era produzir uma linguagem de fácil interpretação mas, ao mesmo tempo, com uma eficiência idêntica à linguagem assembly.

A linguagem FORTRAN foi ao mesmo tempo revolucionária e inovadora. Os programadores libertaram-se assim da tarefa extenuante de usar a linguagem assembler e passaram a ter oportunidade de se concentrar mais na resolução do problema. Mas, talvez mais importante, foi o fato dos computadores passarem a ficar mais acessíveis a qualquer pessoa com vontade de despender um esforço mínimo para conhecer a linguagem FORTRAN. A partir dessa altura, já não era preciso ser um especialista em computadores para escrever programas para computador.

Nos anos seguintes, outras empresas de computadores desenvolveram os seus próprios compiladores de FORTRAN para os seus computadores. Desta forma, programas escritos para uma máquina não podiam ser usados noutras máquinas sem proceder a algumas modificações. Verificou-se assim uma proliferação de diferentes compiladores de FORTRAN. A aquisição de programas em FORTRAN de diferentes procedências, associado à necessidade de converter todos esses programas sempre que estes eram instalados num novo computador, tornou os custos totais proibitivos.

Para ultrapassar estes problemas, passou-se a discutir a necessidade de se proceder a uma normalização da linguagem FORTRAN de forma a que os programas fossem portáveis, isto é, que pudessem ser processados em diferentes máquinas com alterações muito pequenas ou, de preferência, sem qualquer alteração. Em 1966, após quatro anos de trabalho, a Associação Americana de Normalização, posteriormente passou a designar-se Instituto Americano de Normalização Nacional (American National Standards Institute, ANSI), publicou uma versão normalizada designada por FORTRAN IV. Na sua essência, esta versão era um subconjunto comum dos vários dialetos do FORTRAN, de forma que cada dialeto era considerado como uma extensão da versão normalizada. Os utilizadores desta linguagem que pretendessem escrever programas portáveis teriam que ter o cuidado de evitar as extensões referidas.

A proliferação de dialetos continuou a ser um problema após a publicação da versão normalizada em 1966. A primeira dificuldade era a relutância das empresas que desenvolviam os diferentes compiladores a aderirem à normalização. Por outro lado, a implementação de características nos diferentes compiladores, que eram essenciais para programas de longa extensão, foram ignoradas pela versão normalizada.

Esta situação, combinada com a existência de algumas debilidades evidentes na linguagem normalizada, conduziu à introdução de um grande número de pré-processadores. Estes eram programas que liam o código da linguagem de um determinado dialeto do FORTRAN e gerava um segundo texto na versão normalizada. Este procedimento era uma forma de estender as capacidades do FORTRAN usual, mantendo a portabilidade entre diferentes computadores. O aumento do número de pré-processadores registrado nos anos subsequentes, significava não só a grande diversidade de dialetos do FORTRAN, mas também a insuficiência da versão normalizada. Apesar dos programas escritos usando um pré-processador fossem portáveis, o código em FORTRAN gerado desta forma era geralmente de leitura e interpretação muito difícil.

Estas dificuldades foram parcialmente resolvidas pela publicação de uma nova norma-

lização, em 1978, conhecida por FORTRAN 77. Esta versão incluía várias novas características baseadas em dialetos já existentes ou em pré-processadores e, por isso, não era um mero subconjunto dos dialetos existentes, mas sim um novo dialeto. O período de transição entre o FORTRAN IV e o FORTRAN 77 revelou-se, no entanto, extremamente longo, devido ao atraso na avaliação dos novos compiladores baseados na nova versão normalizada e à necessidade das duas versões normalizadas coexistirem por um período de tempo considerável. Na realidade, somente nos meados da década de 80 o FORTRAN IV passou a ter um uso residual.

Após trinta anos de existência o FORTRAN estava longe de ser a única linguagem de programação disponível na maioria dos computadores. As modificações significativas introduzidas no FORTRAN 77 não resolveram todos os problemas que apareceram com a primeira versão normalizada, nem sequer incluía muitas das novas características que, entretanto apareceram com as novas linguagens de programação como o Pascal ou o C. A comunidade de utilizadores do FORTRAN embora com um vasto investimento em códigos de FORTRAN (alguns programas continham mais de 100.000 linhas de instruções) em plena utilização, não estava completamente satisfeita com a linguagem. Por consequência, iniciaram-se trabalhos para rever a versão normalizada. Para o efeito, a ANSI formou um comité técnico, denominado por X3J3, trabalhando como um corpo de desenvolvimento do comitê ISO, designado por ISO/IEC JTC1/SC22/WG5 (que será referido abreviadamente por WG5), na nova versão normalizada, inicialmente referida por FORTRAN 8x que resultou posteriormente no FORTRAN 90.

As motivações para o desenvolvimento da nova versão eram não somente normalizar as diferentes extensões comercializadas, mas também modernizar a linguagem como resposta a outras linguagens de programação como o APL, Algol, Pascal, Ada, C e C++. De forma a preservar o vasto investimento nos códigos anteriormente desenvolvidos, todo o FORTRAN 77 é considerado como um subconjunto da nova versão, embora algumas características sejam desaconselhadas na elaboração de novos programas.

Ao contrário das versões anteriores que resultaram em grande parte de um esforço para normalizar "práticas" já existentes, o FORTRAN 90 é muito mais do que um desenvolvimento da linguagem, introduzindo aspectos que são novidade em FORTRAN e resultam da experiência obtida noutras linguagens. As novas características mais importantes são a facilidade de utilizar variáveis indexadas ("arrays") com uma notação mais concisa e poderosa e a facilidade de definir e manipular diferentes tipos de dados definidos pelo utilizador. O primeiro aspecto permite uma simplificação na programação de problemas matemáticos e torna a linguagem FORTRAN mais eficiente quando se utilizam super-computadores uma vez que se adapta mais convenientemente ao hardware. O segundo aspecto permite aos programadores descrever os seus problemas em termos dos dados-tipo que combinam perfeitamente com as suas necessidades.

Após a publicação da versão normalizada do FORTRAN 90, o WG5 optou por uma nova forma de atuação para revisões futuras. A filosofia atual é a seguinte: se uma nova característica com possibilidade de ser introduzida no futuro não se encontrar suficientemente desenvolvida até uma data pré-estabelecida, então é preferível abandonar essa característica em vez de retardar a nova revisão.

O WG5 passou a ser assim a entidade que decide os vetores de desenvolvimento para as futuras versões do FORTRAN. Entretanto, apareceu o FORTRAN 95 como resultado desta nova filosofia de revisões. Esta versão consiste numa pequena revisão do FORTRAN 90 consistindo unicamente em "correções, clarificações e interpretações", algumas novas características e desaparecimento de outras.

O padrão FORTRAN 2003 dita regras mais precisas sobre a implementação de características inerentes a orientação a objetos (iniciado no FORTRAN 90). Além disso, entre os compiladores, existem aqueles que melhoram a portabilidade da linguagem com vista à programação com máquinas paralelas ampliando as possibilidades de otimização de tempo e memória.

Texto retirado de http://paginas.fe.up.pt/∼aarh/pc/PC-capitulo2.pdf

2 Primeiros algoritmos

Uma das primeiras coisas que se deve aprender, é como resolver problemas com o auxílio do computador, isto é, como montar, logicamente, as instruções para obter a solução de um problema que o computador resolverá utilizando as instruções programadas pelo usuário, ou seja, projetar e escrever um algoritmo, que é uma sequência ordenada de passos executáveis, e precisamente definidos, que manipulam um volume de informações, a fim de obter um resultado. O algoritmo é um método finito, escrito em um vocabulário simbólico fixo, regido por instruções precisas, que se movem em passos discretos

Para que possamos desenvolver um algoritmo, primeiramente devemos entender o problema a ser resolvido e quais serão as informações de entrada e saída. Definindo isso, deve-se esquematizar um processo lógico que o computador entenda.

Fazer um bolo pode ser associado a um algoritmo, definindo os ingredientes como componentes de entrada, o trabalho do cozinheiro como a execução do algoritmo e o bolo (resultado final) como componente de saída. Assim, como fazer um bolo, um algoritimo pode ser feito de várias formas, eficientes ou não, que produzirão um mesmo resultado.

Os computadores só podem executar diretamente os algoritmos expressos em linguagem de máquina. A tradução de um programa escrito em linguagem de alto nível para linguagem de máquina é feita por um programa tradutor denominado *Compilador*. Aqui será discutida uma versão que suporta o FORTRAN 90, apesar de já existirem versões mais novas.

2.1 Caracteres válidos

Nesse ambiente, regras e restrições devem ser seguidas, porém o FORTRAN não é sensível a letras maiúsculas e minúsculas (não é "Case sensitive"), ou seja, cada palavra, independentemente dos caracteres maiúsculos ou minúsculos, tem mesma representação para o programa.

Para que um algoritmo seja executado com sucesso, somente esses caracteres devem ser utilizados na programação:

Tabela 1: Caracteres válidos.

0, ..., 9	Algarismos	+	Sinal de mais	$	Cifrão
A, ..., Z	Letras maiúsculas	−	Sinal de menos	;	Ponto e vírgula
a, ..., z	Letras minúsculas	/	Barra (slash)	<	Menor que
'	Plica ou apóstrofe		Espaço em branco	>	Maior que
"	Aspas	:	Dois pontos	%	Porcentagem
(	Parênteses à esquerda	=	Sinal de igual	?	Ponto de interrogação
)	Parênteses à direita	!	Ponto de exclamação	,	Vírgula
*	Asterisco	&	"E" comercial	.	Ponto

Além disso, é aceitável o máximo de 132 caracteres por linha e nomes com até 31 caracteres.

2.2 Criando um projeto

Este material é baseado na utilização do compilador *Fortran PowerStation 4.0*. Para que se possa iniciar um programa em seu ambiente, é necessário criar novo espaço de trabalho (diretório) do projeto, para isto, após abrir o *Fortran PowerStation 4.0*, deve-se acessar através da barra de ferramentas *File/New.../Project Workspace* e clicar *OK*. No campo *Name:* deve ser inserido um nome para o ambiente de trabalho, e no campo *Location:* o local onde este deve ser salvo.

Após o preenchimento desses campos clicar em *Create*. Após a criação do espaço de trabalho se faz necessário criar também um arquivo de texto onde seu programa será digitado, para sua criação acessar, pela barra de ferramentas *File/New.../Text File* e clicar *OK*. A tela em branco com cursor que aparece é o editor de texto.

Apesar do espaço de trabalho estar criado, o projeto ainda não está associado a ele, portanto para isso é necessário acessar *File/Save* ou atalho equivalente e digitar o nome do programa *.f90* no campo *File Name*. Em *Directories* é possível selecionar o espaço de trabalho em que se deseja salvar o projeto.

Para acessar outro projeto salvo com respectivo espaço de trabalho é necessário acessar *File/Open Workspace...*, selecionar o espaço de trabalho desejado em *Directories* e selecionar o arquivo *mdp* na tela à esquerda antes do *OK*. Clicando no projeto dentro do espaço de trabalho que aparece na janela é possível retomar o projeto.

Um programa em FORTRAN deve ter o seguinte formato:

```
program nome
declarações
comandos
end program nome
```

O nome do programa deve iniciar sempre por letra e não pode conter espaço, caso necessário utilizar "_" e, para finalizar o programa, em vez de escrever "end program nome", pode-se usar apenas "end program" ou "end".

A declaração "program nome" é opcional, porém o seu uso é recomendado. O único campo não opcional na estrutura de um programa, na definição do padrão da linguagem, é a instrução "end", a qual possui dois propósitos: indicar ao compilador que o programa chegou ao fim e, quando da execução do código, provoca a parada do mesmo.

2.2.1 "READ" e "PRINT"

Para que se possa entrar com um valor a ser operacionado no programa, utiliza-se o comando "read*," seguido pelas variáveis a serem lidas, separadas por vírgula e na ordem da leitura.

Para imprimir valores operacionados no programa, utiliza-se o comando "print*," seguido pelas variáveis a serem impressas, separadas por vírgula e na ordem da impressão.

Exemplo 2.1

```
program exemplo
read*,a
print*,a
end program exemplo
```

No exemplo, o programa lê um valor representado pela variável *a*, e em seguida imprime este valor.

2.2.2 Execução do programa

1. Após ter escrito o programa no editor de dexto, acessar pela barra de ferramentas *File/Save*, para salvar seu projeto.

2. Compilar o projeto significa converte-lo em linguagem de máquina e, para isso se faz necessário acessar pela barra de ferramentas *Build/Compile...*. Ao compilar seu programa, informações contendo possíveis erros e avisos serão apresentadas, caso apareça *"0 error(s), 0 warning(s)"* pode prosseguir, caso contrário, deve verificar a(s) causa(s) do(s) problema(s).

3. Após a compilação nenhum e problema encontrado, um executável deve ser gerado, para isso se faz necessário acessar pela barra de ferramentas *Build/Build...*. Novamente serão apresentados possíveis erros e avisos.

4. O programa está pronto para ser executado. Arquivos aparecerão na pasta do espaço de trabalho, assim, se houverem alterações no programa, para que essas sejam atualizadas no projeto os passos 1, 2 e 3 devem ser repetidos.

5. Para executar o programa basta acessar *Build/Execute...* na barra de ferramentas.

Exercício 2.1 *Executar o algoritmo do exemplo 2.1.*

2.3 Declarações

As variáveis podem ser inteiras, reais, complexas, literais ou lógicas. A declaração de uma variável deve vir antes que ela seja usada, se isto não ocorrer o compilador assumirá que as variáveis que começam com as letras I até N como inteiras (INTEGER*4) e todas as outras como reais (REAL*4), ou seja, delaradas implicitamente como ocorrido no *Exemplo 2.1*.

Esta forma de declaração implícita pode ser modificada usando o comando "implicit tipo (a1-a2,b1-b2,...)" sendo a1, a2, b1, b2 quaisquer letras do alfabeto. A vírgula separa os intervalos de letras e o sinal "–" determina o intervalo. As letras que não estiverem em nenhum dos intervalos terá o seu tipo dado pela declaração implícita. O comando seguinte indica que as variáveis que começam com as letras a, b, c e de r até z são do tipo real: implicit real (a, b, c, r-z). Os espaços são usados para dar clareza e são ignorados pelo compilador.

Quando não se deseja que nenhuma variável seja declarada implicitamente usa-se o comando "implicit none". Se este comando for usado e uma variável citada no programa não estiver em nenhuma outra declaração o complilador acusará um erro. **É sempre bom utilizar o comando "implicit none" evitando erros no programa.**

Para se declarar variáveis que sejam matrizes e vetores deve-se indicar suas dimensões logo após o nome da variável, entre parênteses, e separadas umas das outras por vírgula. Por exemplo, "a(4,3)" indica uma matriz a de 4 linhas por 3 colunas.

As variáveis podem receber valores iniciais usando "/valor/", logo após sua declaração. No caso de vetores e matrizes devem ser dados os valores para todos os elementos de cada linha em sequência.

2.3.1 Variáveis inteiras (INTEGER)

As variáveis declaradas como inteiras podem assumir os seguintes valores:

- INTEGER*1: –128 a 127
- INTEGER*2: –32.768 a 32.767
- INTEGER*4 ou INTEGER: –2.147.483.648 a 2.147.483.647

Os números após o "*" indicam quantos bytes a variável ocupa na memória do computador. Esta observação é válida para todos os tipos de variáveis.

Quando uma variável inteira recebe o resultado de uma divisão com resto, este resto é desprezado, ou seja o valor é truncado.

Exemplo 2.2

```
program exemplo
implicit none
integer*1::  c, f
integer::  d, a/6/, b(2,2)/0,1,2,3/
comandos
end
```

Neste caso, o programa identifica c e f como variáveis inteiras que podem assumir valores de –128 a 127, d como variável inteira que pode assumir valores de –2.147.483.648 a 2.147.483.647, a como variável inteira que assume, à priori, o valor 6, mas pode assumir durante a execução do programa através dos comandos valores de –2.147.483.648 a 2.147.483.647 e b como uma matriz 2x2 de variáveis inteiras que assume, à priori,

$$b = \begin{bmatrix} 0 & 2 \\ 1 & 3 \end{bmatrix}$$

, mas pode assumir durante a execução do programa através dos comandos, para cada elemento da matriz, valores de –2.147.483.648 a 2.147.483.647.

O símbolo "::", na maioria dos casos, não precisa ser usado e pode ser trocado por espaço, assim, por exemplo "integer*1:: c, f", pode ser escrito como "integer*1 c, f".

2.3.2 Variáveis reais (REAL)

As variáveis declaradas como reais podem assumir os seguintes valores:

- REAL*4 ou REAL(Precisão simples com 6 casas decimais):

  Varia de $-3,402823 \times 10^{38}$ (-3.402823E+38) a $+3,402823 \times 10^{38}$ (+3.402823E+38) e tem incremento mínimo de $-1,175494 \times 10^{-38}$ a $+1,175494 \times 10^{-38}$, ou seja, números reais neste intervalo são considerados zero.

- REAL*8 ou DOBLE PRECISION (Precisão dupla com 15 casas decimais):

  Varia de $-1,797693134862316 \times 10^{308}$ (-1.797693134862316D+308) a $+1,797693134862316 \times 10^{308}$ (+1.797693134862316D+308) e tem incremento mínimo de $-2,225073858507201 \times 10^{-308}$ a $+2,225073858507201 \times 10^{-308}$. A parte exponencial deve ser separada por um d ou D no lugar do e ou E para real do tipo *8.

Colocando-se "." após um número no programa garantimos que ele seja real, caso contrário somente será reconhecida sua parte inteira.

Exemplo 2.3

```
program exemplo
implicit none
real*4 r1,r2
real*8 r3,r4
integer i1,i2,a
read*,a
r1=a/3.
r2=a/3
r3=a/3.
r4=a/3
i1=a/3.
i2=a/3
print*,r1,r2,r3,r4,i1,i2
end
```

2.3.3 Variáveis complexas (COMPLEX)

- COMPLEX*8 ou COMPLEX (Precisão simples com 6 casas decimais)
- COMPLEX*16 (Precisão dupla com 15 casas decimais)

Os valores que um complexo pode assumir são os mesmos que os reais representado por um par ordenado (*parte real,parte complexa*) de reais.

Exemplo 2.4

```
program exemplo
implicit none
complex c
c = (-1,0)
c = sqrt (c)
print*,c
end
```

2.3.4 Variáveis alfanuméricas ou literais (CHARACTER)

A variável do tipo character é declarada digitando-se "character nome*w" em que nome é a variável alfanuméria e *w* representa o número máximo de caracteres que a variável pode conter dentro do programa. A variável poderá conter até 32.767 caracteres e caso *w* não seja apresentado ("character nome") ele corresponderá a 1.

O tamanho, em bytes, de um tipo character é igual a quantidade de caracteres que a constante ou variável possui mais 1, onde este byte a mais, guarda o tamanho da constante ou variável.

Quando a variável recebe os caracteres na própria execução do programa, ou seja, através da atribuição, ou no momento da impressão, os caracteres atribuídos devem vir entre aspas e contendo apenas caracteres válidos. Quando a variável recebe os caracteres na execução do programa através do "read*," espaços entre palavras não podem ocorrer, neste caso é sugerido o uso de "_".

Exemplo 2.5

```
program exemplo
implicit none
character nome*30
print*, "Qual e seu nome?"
read*, nome
print*, "Eu me chamo ", nome
end
```

2.3.5 Variáveis lógicas (LOGICAL)

A variável lógica é uma variável binária que pode assumir somente os valores *.true.* (verdadeiro) ou *.false.* (falso). Ou até mesmo *0* e *1*. Utiliza 4 bytes de memória.

Exemplo 2.6

```
program exemplo
implicit none
logical log1, log2, log3
log1= 1
log2= .false.
log3= 0
print*, log1, log2, log3
end
```

2.3.6 Variáveis parâmetros (PARAMETER)

As vezes necessitamos, em um programa, de constante ou parâmetro que não deve receber outro valor, uma vez que já foi atribuído um valor no momento de sua declaração. Para casos como este que utilizamos variáveis parâmetros.

Exemplo 2.7

```
program exemplo
implicit none
real raio, pi
parameter (pi = 3.14159)
read*, raio
print*, "A area do círculo é ", pi*raio*raio
end
```

2.4 Operadores

2.4.1 Atribuição

A variável ou identificador que estiver à esquerda do sinal de atribuição "=" recebem o valor da expressão, constante ou variável que estiver à direita.

Identificador = Expressão

Alguns exemplos de atribuições:

```
⋮
nome = "Jonatas Henrique"
curso = "Ciencias Biologicas"
!Não se pode usar acentuação
ano = 1999 !  Não é preciso de ";" no final
prova(1)= 9.6; prova(2)= 8; prova(3)= .8
!  Atribuindo valores a vetores
t= 9.9; data1 = "12/10/10"
nota = (2*prova(1)+3*prova(2)+4*prova(3)+t)/10
!  Casas decimais são separadas por ponto
⋮
```

Estas declarações poderiam estar na mesma linha desde que fossem separadas por ponto e vírgula ";".

2.4.2 Operadores literais

Uma função útil para variáveis literais é a concatenação, ou a junção de duas ou mais palavras. Em FORTRAN a concatenação é feita pelo operador "//".

Por exemplo,

Exemplo 2.8

```
program exemplo
implicit none
character a*3,b*5,c*8
a = "mel"
b = "ancia"
c = a//b
print*, c !c = "melancia"
end
```

2.4.3 Operadores aritméticos

Executam operações aritméticas comuns.

Tabela 2: Operadores aritméticos.

Fortran	Algébrico	Significado
$+$	$+$	Soma
$-$	$-$	Subtração
$*$	$\cdot$	Multiplicação
$/$	$\div$	Divisão
$**$	a^b	Potenciação

2.4.4 Operadores relacionais

Comparam variáveis, constantes ou expressões e retornam ".TRUE.", "T" ou "1" se a comparação for verdadeira, ".FALSE.", "F" ou "0" se a comparação for falsa.

Tabela 3: Operadores relacionais.

Fortran	F90	Algébrico	Significado
.LT.	$<$	$<$	menor que
.LE.	$<=$	$\leq$	menor ou igual que
.EQ.	$==$	$=$	igual a
.NE.	$/=$	$\neq$	diferente de
.GT.	$>$	$>$	maior que
.GE.	$>=$	$\geq$	maior ou igual que

Exemplo 2.9

```
program exemplo
implicit none
logical e, f, g
real a, b, c, d
read*, a, b, c, d
e = (a + b)*0.5 .LT. (c + d)*0.5
f = (a + b)*0.5 == (c + d)*0.5
g = (a + b)*0.5 > (c + d)*0.5
print*, e, f, g
end
```

2.4.5 Operadores lógicos

São usados quando são necessárias mais de uma condição relacional ou quando é preciso inverter seu resultado.

Tabela 4: Operadores lógicos.

Fortran	Significado
.AND.	Verdadeiro se os dois operadores forem verdadeiros
.OR.	Verdadeiro se ao menos um dos dois operadores for verdadeiro
.NOT.	Verdadeiro se o operador for falso (negação)
.NEQV. ou .XOR.	Verdadeiro se somente um dos operadores for verdadeiro
.EQV.	Verdadeiro se os dois operadores forem falsos ou verdadeiros

Por exemplo,

Exemplo 2.10

```
program exemplo
implicit none
logical r1, r2, r3, r4, r5, r6, r7, r8
r1 = 10.GT.5 .AND. 20.GT.25 !  .FALSE.
r2 = 10.GT.5 .OR. 20.GT.25 !  .TRUE.
r3 = .NOT. 20.GT.25 !  .TRUE.
r4 = 10 > 5 .XOR. 20 >= 25 !  .TRUE.
r5 = 10.GT.5 .NEQV. 25.GT.20 !  .FALSE.
r6 = 10 < 5 .EQV. 20 <= 25 !  .FALSE.
r7 = 10.LT.5 .EQV. 25.GT.20 !  .FALSE.
r8 = "José" == "Pedro" !  .FALSE.
print*, r1, r2, r3, r4, r5, r6, r7, r8
end
```

2.4.6 Prioridades

FORTRAN usa a seguinte relação de prioridades:

Tabela 5: Ordem de prioridade.

Prioridade	1°	2°	2°	3°	3°	4°	4°	4°	4°	4°	4°	5°	6°	7°
Operador	**	*	/	+	−	.eq.	.ne.	.gt.	.ge.	.lt.	.le.	.not.	.and.	.or.

O uso de parênteses pode ser feito para trocar a ordem de prioridade.

2.5 Funções intrínsecas

Existem várias funções predefinidas em FORTRAN, que podem ser usadas em qualquer parte do programa. Aqui serão sitadas algumas delas, outras funções intrínsecas podem ser encontradas.

Tabela 6: Funções diversas.

Nome	Definição	Tipo de argumento	Tipo de função
ALOG10(x)	Logaritmo de x na base 10	Real	Real
ALOG(x)	Logaritmo neperiano de x	Real (x > 0)	Real
EXP(x)	O número e elevado a x	Real	Real
ABS(x)	Valor absoluto de x	Real	Real
IABS(x)	Valor absoluto de x	Inteiro	Inteiro
IFIX(x)	Trunca de real para inteiro	Real	Inteiro
FLOAT(x)	Conversão de inteiro para real	Inteiro	Real
DBLE(x)	Converte para dupla precisão	Real	Real*8
CMPLX(x)	Converte para o tipo complexo	Real	Complexo
SIGN(x,y)	Fornece |x| se y $\geq$ 0 e -|x| se y < 0	Real	Real
MOD(x,y)	Resto da divisão de x por y	Inteiro	Inteiro
AMOD(x,y)	Resto da divisão de x por y	Real	Real
SQRT(x)	Raiz quadrada de x	Real (x $\geq$ 0)/Complexo	Real/Complexo
MAX(x,y)	Maior entre x e y	Inteiro/Real	Inteiro/Real
MIN(x,y)	Menor entre x e y	Inteiro/Real	Inteiro/Real

Tabela 7: Funções trigonométricas.

Nome	Definição	Tipo de argumento	Tipo de função
SIN(x)	Seno (radianos)	Real ou complexo	REAL*4
ASIN(x)	Arcoseno (radianos)	Real, $\|x\| \leq 1$	REAL*4
COS(x)	Coseno (radianos)	Real ou complexo	REAL*4
ACOS(x)	Arcocoseno (radianos)	Real, $\|x\| \leq 1$	REAL*4
TAN(x)	Tangente (radianos)	Real	REAL*4
ATAN(x)	Arcotangente (radianos)	Real	REAL*4
SINH(x)	Seno Hiperbólico (radianos)	Real	REAL*4
COSH(x)	Coseno Hiperbólico (radianos)	Real	REAL*4
TANH(x)	Tangente Hiperbólica (radianos)	Real	REAL*4

2.6 Exercícios

Exercício 2.2 *Desenvolver um algoritmo que lê uma amostra com 5 valores reais e imprime média e variância.*

Exercício 2.3 *Dado um número inteiro de quatro algarismos imprimir o número formado pelos 2 primeiros dígitos, o número formado pelos 2 últimos dígitos e a soma desses dois resultados.*

Exercício 2.4 *Fazer um algoritmo que calcule as raízes da função $f(x) = ax^2 + bx + c$, em que a, b e c são informações fornecidas no momento da execução do programa.*

Exercício 2.5 *Fazer um algoritmo que imprima o quociente e o resto da divisão entre dois números fornecidos no momento da execução do programa.*

Exercício 2.6 *O número 3025 tem a seguinte característica:*

$$30 + 25 = 55 \text{ e } 55 \times 55 = 3025$$

Fazer um algoritmo que lê um número inteiro de quatro algarismos e testa se ele tem ou não a mesma característica do 3025.

Exercício 2.7 *São dados dois números inteiros positivos p e q, sendo que p e q têm respectivamente 2 e 7 dígitos. Verificar se p é um subnúmero de q.*

Exemplo:

$p = 23$, $q = 5723835$, p é subnúmero de q.

$p = 23$, $q = 2583479$, p não é subnúmero de q.

3 Controle de execução

Os programas em FORTRAN devem conter os comandos escritos na ordem em que serão executados, com exceção das funções, subrotinas e laços de repetição. O comando "end" indica o fim do programa. Se o programador preferir pode finalizar o programa prematuramente usando os comandos "stop" ou "call exit".

Por exemplo:

Exemplo 3.1

```
program exemplo
implicit none
integer a, b
read*, a, b
print*, a, b
stop !ou "call exit"
a = a**b
print*, a
end
```

Neste caso, o programa encerrará na linha de comando em que aparece "stop" ou, se preferir pode colocar "call exit".

3.1 "GOTO"

Quando se deseja que o comando do programa avance ou recue em sua estrutura de forma não sequencial, usa-se o comando "goto".

```
...
goto n
...
n comando
...
```

Em que n é um número inteiro positivo que rotula uma linha que possua ou não um comando, na qual o programa, ao executar a linha "goto n" redireciona-se à linha "n comando", podendo esta estar antes ou depois de "goto n". Como uma linha rotulada não pode estar em branco e pode não conter comando, pode-se usar a palavra chave "continue".

Por exemplo:

Exemplo 3.2

```
program exemplo
implicit none
integer a, b
read*, a, b
goto 12
print*, a, b
a = a**b
12 print*, a
end
```

3.2 Estrutura condicional

Toda linguagem de programação estruturada necessita de artifícios que possibilitem a execução condicional de comandos. Esses comandos normalmente alteram o fluxo de execução de um programa.

Uma das características da linguagem FORTRAN é o processamento de cima para baixo, linha por linha. Entretanto, essa ordem pode ser alterada quando utilizamos algumas condições para que os cálculos sejam realizados. Isso pode ser feito utilizando o comando IF.

3.2.1 Estrutura condicional simples

A sintaxe do comando de uma estrutura condicional simples é descrita por:

```
...
if (condição)
comando
...
```

ou

```
...
if (condição) then
bloco de comandos
end if
...
```

Quando a condição for verdadeira o comando ou bloco de comandos será executado, quando for falsa o programa segue para o próximo comando logo abaixo da estrutura condicional. A primeira opção só é válida quando for executado um único comando. Este comando pode ser de qualquer tipo, atribuição, escrita, leitura, "goto" ou interrupção do programa.

Por exemplo,

Exemplo 3.3

```
program exemplo
implicit none
integer a,b
read*, a,b
if (mod(a,b)==0) then
print*, a," e multiplo de", b
end if
end
```

Obs.: É permitido o uso de estruturas condicionais umas dento das outras.

3.2.2 Estrutura condicional composta

Na estrutura condicional composta, se a condição for verdadeira, o *bloco1* é executado, se não o *bloco2* é que será executado. Mesmo quando só há um comando no *bloco1*, não se pode omitir a palavra chave "then". A sintaxe do comando é:

```
...
if (condição) then
bloco1
else
bloco2
end if
...
```

Por exemplo,

Exemplo 3.4

```
program exemplo
implicit none
integer a,b
read*, a,b
if (mod(a,b)==0) then
print*, a," e multiplo de", b
else
print*, a," nao e multiplo de", b
end if
end
```

3.2.3 Estrutura condicional composta expandida

Na estrutura condicional composta expandida várias condições são testadas. A estrutura da sintaxe é:

```
...
if (condição1) then
bloco1
else if (condição2) then
bloco2
...
else if (condiçãon) then
blocon
else
bloco(n+1)
end if
...
```

Dessa forma, se a *condiçãok* for satisfeita, as seguintes são ignoradas.
Por exemplo:

Exemplo 3.5

```
program exemplo
implicit none
real nota
print*, "Qual foi sua nota?"
read*, nota
if (nota .ge.  9.00) then
print*, "Muito bem!  Conceito A"
else if (nota .ge.  7.00) then
print*, "Voce foi bem!  Conceito B"
else if (nota .ge.  5.00) then
print*, "Voce nao foi tao bem!  Conceito C"
else
print*, "REPROVADO!!!!!!"
end if
end
```

3.2.4 Estrutura condicional composta simplificada

Uma outra forma de se usar uma estrutura condicional composta é usando o comando "case". A sintaxe de comando é:

```
...
nome_case:  select case (exp.  case)
case (lista de seleção 1)
comandos1
case (lista de seleção 2)
comandos2
...
case (lista de seleção n)
comandosn
case default
comandosd
end select nome_case
...
```

Em que "exp. case" é uma expressão ou constante inteira, lógica ou literal (somente um caractere "character*1"). Caso o valor de "exp. case" estiver na "lista de seleção 1", os "comandos1" serão executados. Se o valor não estiver na "lista de seleção 1" o computador irá avaliar a "lista de seleção 2", se for verdadeira serão executados os "comandos2" e assim até terminar os comandos "case (lista de seleção n)". O comando "case default" é opcional, e faz com que os "comandosd" sejam executados caso nenhuma das outras avaliações sejam verdadeiras. "nome_case" é opcional e deve seguir as mesmas regras usadas para dar nomes as variáveis. A sua utilidade é apenas de dar maior clareza ao programa.

É importante lembrar que somente uma das condições (lista de seleção) deve ser satisfeita.

As listas de seleção podem ser da seguinte forma:

Tabela 8: Listas de seleção.

Estrutura	Condição para ser verdadeira
case (valor)	exp. case igual ao valor
case (:valor)	exp. case menor ou igual ao valor
case (valor:)	exp. case maior ou igual ao valor
case (valor1:valor2)	exp. case entre valor1 e valor2
case (valor1,valor2,...,valorn)	exp. case igual ao valor1 ou igual ao valor2 ou ... valorn

Por exemplo, utilizando uma constante literal:

Exemplo 3.6

```
program exemplo
implicit none
character i*1
i="h"
valor_i:  select case (i)
case ("a","b","c")
print*, "i=a ou b ou c"
case ("d":"m")
print*, "i esta entre d e m"
case ("D":"M")
print*, "i esta entre D e M"
end select valor_i
end
```

Por exemplo, utilizando uma constante inteira:

Exemplo 3.7

```
program exemplo
implicit none
integer a
read*, a
select case (a)
case (:-2)
print*, a, " menor ou igual a -2"
case (0)
print*, a, " igual a zero"
case (2:7)
print*, a, " entre 2 e 7"
case default
print*, "nenhuma das condicoes foi satisfeita"
end select
end
```

3.3 Estrutura de repetição

Quando o mesmo comando precisa ser executado várias vezes até que se atinja uma certa condição ou um número certo de repetições, o melhor é usar as estruturas de repetição. Estas estruturas são bem simples e podem economizar várias linhas de comando.

3.3.1 "LOOP" condicional

"Loop" consiste de um bloco de comandos que são executados ciclicamente, infinitamente. É necessário um mecanismo condicional para sair do "loop". O bloco de comandos que é executado ciclicamente é delimitado pelo comando "do ... end do" e o comando exit determina a saída do "loop". A sintaxe de comando é:

```
...
do
...
if (expressão lógica) exit
...
end do
...
```

Por exemplo:

Exemplo 3.8

```
program exemplo
implicit none
integer i
i=0
do
i = i + 1
if (i .GT. 100) exit
print*, "i vale", i
end do
print*, "Fim do loop.  i = ", i
end
```

3.3.2 "LOOP" ciclo condicional

"Loop" cíclico consiste de um mecanismo condicional para sair e iniciar o "loop" novamente. O comando "cycle" determina, novamente, o início imediato do "loop". A sintaxe de comando é:

```
...
do
...
if (expressão lógica) cycle
if (expressão lógica) exit
...
end do
...
```

Por exemplo:

Exemplo 3.9

```
program exemplo
implicit none
integer i
i=0
do
i = i + 1
if (i >= 50 .AND. i <= 59) cycle
if (i .GT. 100) exit
print*, "i vale", i
end do
print*, "Fim do loop.  i = ", i
end
```

3.3.3 "DO" iterativo

O "DO" iterativo consiste num "loop" que possui um número fixo de ciclos. A sintaxe de comando é:

```
...
do variável = expressão1,expressão2,expressão3
...
end do
...
```

Em que, *expressão1* é o valor inicial, *expressão2* é o valor final e *expressão3* é o valor de incremento. Podemos interpretar "*do variável = expressão1,expressão2,expressão3* ... end do" como "Para *variável*, de *expressão1* até *expressão2* passo *expressão3* faça ... fim para".

Por exemplo:

Exemplo 3.10

```
program exemplo
implicit none
integer i
do i = 2,100,2
print*, "i vale", i
end do
end
```

3.3.4 "DO-WHILE"

O "DO-WHILE" consiste num "loop" que condiciona a sua execução antes de executar o bloco de comandos, pois a condição é testada no topo do "loop". A sintaxe de comando é:

```
...
do while (expressão lógica)
...
end do
...
```

Podemos interpretar "do while (expressão lógica) ... end do" como "Faça enquanto *expressão lógica* ... fim para".

Por exemplo:

Exemplo 3.11

```
program exemplo
implicit none
integer I
I = 1
do while (I<25)
if (I==1) then
print*, "Passei", I, " vez"
else
print*, "Passei" , I, " vezes"
endif
I=I+1
enddo
end
```

3.4 Exercícios

Exercício 3.1 *Fazer um algoritmo que gere os n primeiros elementos da sequência de Fibonacci, dada por: 1, 1, 2, 3, 5, 8, 13, 21, ...*

Exercício 3.2 *Desenvolver um programa que calcule n! para um n inteiro não negativo qualquer.*

Exercício 3.3 *Desenvolver um programa que solicita que se digite três valores e informa se eles podem corresponder a três lados de um triângulo equilátero, isósceles, escaleno ou não formam um triângulo.*

Exercício 3.4 *Desenvolver um programa que informe os números primos entre 1 e 1000.*

Exercício 3.5 *Desenvolver um programa que escreva um número natural n como produto de números primos.*

Exercício 3.6 *O número 3025 tem a seguinte característica:*

$$30 + 25 = 55 \text{ e } 55 \times 55 = 3025$$

Fazer um algoritmo que apresente todos os números naturais de quatro dígitos que têm a mesma característica do 3025.

Exercício 3.7 *Sabendo que o número e dos logarítimos naturais neperianos é aproximadamente 2,718281 e sabendo que este número é representado numericamente pela soma abaixo, em que quanto maior o valor de x, melhor a aproximação. Crie um programa que calcule o valor de x que nos dará quantos elementos da série serão somados para que se obtenha um valor maior ou igual ao apresentado.*

$$e \cong \sum_{n=0}^{x} \frac{1}{n!}$$

Exercício 3.8 *Escreva um programa para identificar os números amigáveis menores que 2000. Dois números são amigáveis quando cada um é igual à soma dos divisores do outro número (excluindo apenas o próprio número). Exemplo: 220 e 284 são números amigáveis pois a soma dos divisores de 220 (1, 2, 4, 5, 10, 11, 20, 22, 44, 55, 110) é igual a 284 e a soma dos divisores de 284 (1, 2, 4, 71, 142) é igual a 220.*

Exercício 3.9 *Escreva um programa para calcular uma aproximação para $sen(x)$, onde x é um valor inteiro lido da unidade padrão de entrada. A aproximação pode ser obtida de: $sen(x) = x - x/3! + x/5! - x/7! + ...$ O programa deve encerrar o processamento quando a variação no valor calculado for inferior a 0.001.*

4 Matrizes

Matrizes ou "Arrays" são uma coleção de dados armazenados na memória e acessados, individualmente, de acordo com a sua posição espacial, definida pelas dimensões da matriz.

O FORTRAN armazena os elementos de matrizes em espaços contíguos de memória. Este ordenamento é obtido variando-se inicialmente o índice da primeira dimensão da matriz, depois variando-se o índice da segunda dimensão e assim por diante. Em uma matriz com 2 dimensões isto é obtido variando-se inicialmente as linhas e depois as colunas, ou seja, os elemenos são distribuídos por colunas.

4.1 Declarações

Para declarar matrizes existem as seguntes formas:

```
tipodevariável, dimension (a1, a2, ..., ak) ::  mat1, ..., mats
```

ou

```
tipodevariável mat1(a1, a2, ..., ak), ..., mats(b1, b2, ..., bl)
```

ou

```
dimension mat1(a1, a2, ..., ak), ..., mats(b1, b2, ..., bl)
tipodevariável mat1, ..., mats
```

Em que "tipodevariável" determina o tipo de variável para cada elemento da matriz, podendo ser inteiro, real, complexo, alfanumérico ou lógico.

O FORTRAN 90 trabalha com vertores de até 7 dimensões. O limite inferior e superior de níveis por dimensão ficam separados pelo caractere ":". Caso não exista esse caractere, o limite inferior será sempre 1 e o limite superior, o informado na definição da matriz. Se esse limite inferior não for informado, então a alocação de memória será dinâmica, ou seja, durante a execução do programa. Para leitura e impressão, a ordem por coluna, será utilizada.

Por *Default* cada elemento das matrizes recebe, inicialmente, valor zero. Porém, pode-se iniciar os elementos da matriz utilizando "/.../".

Por exemplo,

Exemplo 4.1

```
program exemplo
implicit none
integer, dimension(4) ::  A = (/2,3,4,5/)
integer B(2,2)/1,2,3,4/
print*, A
print*, B
end
```

4.2 Operações

- Para a matriz toda.

Exemplo 4.2

```
program exemplo
implicit none
integer A(3,2), B(3,2)/1,2,3,4,5,6/
A = 1
B = 2*B*A + 1
A = B + A
print*, A
print*, B
end
```

Neste caso, a matriz A é declarada de ordem 3×2 formada por elementos inteiros, por *Default* é nula, e a matriz B declarada de ordem 3×2 formada por elementos inteiros, à priori:

$$B = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Depois, a matriz A recebe valor 1 para todos seus elementos. Cada elemento $B(i,j)$ da matriz B recebe $2 \times B(i,j) \times A(i,j) + 1$, ou seja, quando apresentado "A*B" é calculado o produto elemento a elemento, valendo também para as outras operações matemáticas. E´no comando seguinte cada elemento $A(i,j)$ da matriz A recebe $B(i,j) + A(i,j)$ e, por fim, A e B são impressas. Observa-se que quando dimensão ou ordem de matrizes são diferentes essas operações não podem ser feitas entre elas.

- Para alguns elementos da matriz

Exemplo 4.3

```
program exemplo
implicit none
integer A(3,2), B(3,3)/1,2,3,4,5,6,7,8,9/
A(1,1) = 80
A(3,2) = B(1,1) + B(3,3)
print*, A
print*, A(1,1)
end
```

Neste caso, a matriz A é declarada de ordem 3×2 formada por elementos inteiros, por *Default* é nula, e a matriz B declarada de ordem 3×3 formada por elementos inteiros, à priori:

$$B = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

Depois, os elementos $A(1,1)$ e $A(3,2)$ da matriz A recebem respectivamente valores 80 e $B(1,1) + B(3,3)$ e, por fim, a matriz A e seu elemento $A(1,1)$ são impressos. Observa-se que elementos de matrizes de dimensão ou ordem diferentes podem ser operacionados.

- Para algumas seções de elementos da matriz

  Para se operacionar somente uma seção da matriz, podemos escrever:

  matriz(liminf 1:limsup 1:incremento 1, ..., liminf n:limsup n:incremento n)

  Em que *liminf k*, *limsup k* e *imcremento k* representam respectivamente limite inferior, limite superior e incremento da dimensão k.

  Se um dos limites, inferior ou superior (ou ambos) for omitido, então o limite ausente é assumido como o limite inferior ou superior da correspondente dimensão da matriz da qual a seção está sendo extraída e se o incremento for omitido, então assume-se que valerá um.

  Dessa forma, por exemplo, dada a matriz:

$$X = \begin{bmatrix} 1 & 6 & 11 & 16 & 21 \\ 2 & 7 & 12 & 17 & 22 \\ 3 & 8 & 13 & 18 & 23 \\ 4 & 9 & 14 & 19 & 24 \\ 5 & 10 & 15 & 20 & 25 \end{bmatrix}$$

$X(:,:)$ ou X representa todos os elementos da matriz:

$$\begin{bmatrix} \mathbf{1} & \mathbf{6} & \mathbf{11} & \mathbf{16} & \mathbf{21} \\ \mathbf{2} & \mathbf{7} & \mathbf{12} & \mathbf{17} & \mathbf{22} \\ \mathbf{3} & \mathbf{8} & \mathbf{13} & \mathbf{18} & \mathbf{23} \\ \mathbf{4} & \mathbf{9} & \mathbf{14} & \mathbf{19} & \mathbf{24} \\ \mathbf{5} & \mathbf{10} & \mathbf{15} & \mathbf{20} & \mathbf{25} \end{bmatrix}$$

$X(2{:}2,2{:}2)$ ou $X(2,2)$ representa o elemento da linha 2 e coluna 2.

$$\begin{bmatrix} 1 & 6 & 11 & 16 & 21 \\ 2 & \mathbf{7} & 12 & 17 & 22 \\ 3 & 8 & 13 & 18 & 23 \\ 4 & 9 & 14 & 19 & 24 \\ 5 & 10 & 15 & 20 & 25 \end{bmatrix}$$

$X(3,3{:}5)$ representa os elementos de linha 3 que variam da coluna 3 à 5:

$$\begin{bmatrix} 1 & 6 & 11 & 16 & 21 \\ 2 & 7 & 12 & 17 & 22 \\ 3 & 8 & \mathbf{13} & \mathbf{18} & \mathbf{23} \\ 4 & 9 & 14 & 19 & 24 \\ 5 & 10 & 15 & 20 & 25 \end{bmatrix}$$

$X(2{:}3,2{:}5{:}2)$ representa os elementos das linhas 2 e 3 e das colunas 2 à 5 e incremento 2:

$$\begin{bmatrix} 1 & 6 & 11 & 16 & 21 \\ 2 & \mathbf{7} & 12 & \mathbf{17} & 22 \\ 3 & \mathbf{8} & 13 & \mathbf{18} & 23 \\ 4 & 9 & 14 & 19 & 24 \\ 5 & 10 & 15 & 20 & 25 \end{bmatrix}$$

$X(1{::}2,2{:}4)$ representa os elementos da linha 1 até terminar as linhas com incremento 2 e das colunas 2 à 4:

$$\begin{bmatrix} 1 & \mathbf{6} & \mathbf{11} & \mathbf{16} & 21 \\ 2 & 7 & 12 & 17 & 22 \\ 3 & \mathbf{8} & \mathbf{13} & \mathbf{18} & 23 \\ 4 & 9 & 14 & 19 & 24 \\ 5 & \mathbf{10} & \mathbf{15} & \mathbf{20} & 25 \end{bmatrix}$$

Exemplo 4.4

```
program exemplo
implicit none
integer X(5,5),i,j
do i= 1,5,1
do j= 1,5,1
X(i,j) = (j-1)*5+i
enddo
enddo
print*, X(:,:)
print*, X(2:2,2:2)
print*, X(3,3:5)
print*, X(2:3,2:5:2)
print*, X(1::2,2:4)
end
```

4.3 Leitura e impressão

Há várias formas de leitura e impressão de matrizes, entre elas podemos apenas utilizar o comando "read*, A" e "print*, A" para ler e imprimir uma matriz A. No entanto, os elementos serão lidos e impressos no vídeo em uma linha contínua na ordem por colunas.

Outra forma de leitura e impressão de matrízes muito útil é utilizando um processo iterativo. Por exemplo, caso queiramos entrar com os valores da matriz A por linhas e depois imprimir também nesta mesma ordem:

Exemplo 4.5

```
program exemplo
implicit none
integer A(3,3),i,j
do i= 1,3,1
do j= 1,3,1
read*, A(i,j)
enddo
enddo
do i= 1,3,1
do j= 1,3,1
print*, A(i,j)
enddo
enddo
end
```

Porém, apesar de apresentar funcionalidade matemática, ainda há deficiência estética. Para que a matriz seja apresentada em uma melhor forma utiliza-se:

$$(\ \textbf{\textit{expressão}} \ , \mathbf{i} = \textbf{\textit{a}}, \ \textbf{\textit{n}}, \ \textbf{\textit{b}} \ )$$

Em que, ocorre uma repetição de *expressão* para i de a até b passo n. Ou seja, é equivalente a:

```
...
do i = a, n, b
expressão
end do
...
```

Porém, este tipo de repetição, além representar muito bem o "DO" iterativo, muda de linha na tela sempre que termina. Assim, por exemplo:

Exemplo 4.6

```
program exemplo
implicit none
integer A(3,3),i,j
do i= 1,3,1
read*, ( A(i,j) , j=1, 3, 1 )
enddo
do i= 1,3,1
print*, ( A(i,j) , j=1, 3, 1 )
enddo
end
```

Apresenta o vetor A em linhas e colunas.
Outras formatações serão vistas nas sessões posteriores.

4.4 Funções

Algumas funções podem ser utilizadas para apresentar informações e tributos destas.

Tabela 9: Funções de matrizes.

Função	Significado
LBOUND	Limite inferior das dimensões da matriz
UBOUND	Limite superior das dimensões da matriz
SHAPE	Ordem da matriz
SIZE	Número de elementos da matriz
TRANSPOSE	Transposta da matriz

Para obter as informações para uma determinada dimensão, basta inserir vírgula e a dimensão após o nome da matriz. Por exemplo:

Exemplo 4.7

```
program exemplo
implicit none
integer A(2,3)/1,2,3,4,5,6/,i,j,B(3,2)
B = TRANSPOSE(A)
do i= 1,2,1
print*, ( A(i,j) , j=1, 3, 1 )
enddo
do i= 1,3,1
print*, ( B(i,j) , j=1, 2, 1 )
enddo
print*, LBOUND(A), LBOUND(A,1), LBOUND(A,2)
print*, UBOUND(A), UBOUND(A,1), UBOUND(A,2)
print*, SHAPE(A)
print*, SIZE(A), SIZE(A,1), SIZE(A,2)
end
```

4.5 Alocação

Uma novidade importante introduzida no FORTRAN 90 é a habilidade de se declarar variáveis dinâmicas e em particular, matrizes dinâmicas. O FORTRAN 90 fornece tanto matrizes alocáveis quanto matrizes automáticas, ambos os tipos sendo matrizes dinâmicas. Usando

matrizes alocáveis, é possível alocar e de-alocar espaço de memória conforme necessário. O recurso de matrizes automáticas permite que matrizes locais em uma função ou subrotina tenham forma e tamanho diferentes cada vez que a rotina é invocada.

Matrizes alocáveis permitem que grandes frações da memória do computador sejam usadas somente quando requerido e, posteriormente, liberadas, quando não mais necessárias. Este recurso oferece um uso de memória muito mais eficiente que o FORTRAN 77, o qual oferecia somente alocação estática (fixa) de memória. Além disso, o código torna-se muito mais robusto, pois a forma e o tamanho das matrizes podem ser decididos durante o processamento do código.

Uma matriz alocável é declarada na linha de declaração de tipo de variável com o atributo "ALLOCATABLE". O posto da matriz deve também ser declarado com a inclusão dos símbolos de dois pontos ":", um para cada dimensão da matriz. Por exemplo, a matriz de duas dimensões A é declarada como alocável através da declaração:

```
REAL, DIMENSION(:,:), ALLOCATABLE :: A
```

sta forma de declaração não aloca espaço de memória imediatamente à matriz, como acontece com as declarações usuais de matrizes. O status da matriz nesta situação é not currently allocated, isto é, correntemente não alocada. Espaço de memória é dinamicamente alocado durante a execução do programa, logo antes da matriz ser utilizada, usando-se o comando "ALLOCATE". Este comando especifica os limites da matriz. Por exemplo:

```
ALLOCATE (A(0:N,M))
```

O espaço alocado à matriz com o comando "ALLOCATE" pode, mais tarde, ser liberado com o comando "DEALLOCATE". Este comando requer somente nome da matriz previamente alocada. Por exemplo, para liberar o espaço na memória reservado para a matriz A:

```
DEALLOCATE (A)
```

Tanto os comandos "ALLOCATE" e "DEALLOCATE" possuem o especificador opcional "STAT", o qual retorna o status do comando de alocação ou de-alocação. Neste caso, a forma geral do comando é:

```
ALLOCATE (lista de objetos alocados , STAT= status)
DEALLOCATE (lista de objetos alocados , STAT= status)
```

Em que "*status*" é uma variável inteira escalar. Se STAT= está presente no comando, "*status*" recebe o valor zero se o procedimento do comando ALLOCATE/DEALLOCATE foi bem sucedido ou um valor positivo se houve um erro no processo. Se o especificador STAT= não estiver presente e ocorra um erro no processo, o programa é abortado. Finalmente, é possível alocar-se ou de-alocar-se mais de uma matriz simultaneamente, como indica "*lista de objetos alocados*".

Matrizes alocáveis tornam possível o requerimento freqüente de declarar uma matriz tendo um número variável de elementos. Por exemplo, pode ser necessário ler variáveis, digamos *tam1* e *tam2* e então declarar uma matriz com *tam1* $\times$ *tam2* elementos.

É possível verificar se uma matriz está ou não correntemente alocada usando-se a função intrínseca ALLOCATED. Esta é uma função lógica com um argumento, o qual deve ser o nome de uma matriz alocável. Usando-se esta função, comandos como os seguintes são possíveis:

```
IF (ALLOCATED(A)) DEALLOCATE (A)
IF (.NOT. ALLOCATED(A)) ALLOCATE (A(5,20))
```

Por exemplo:

Exemplo 4.8

```
program exemplo
implicit none
real, dimension(:), allocatable ::  A
real, dimension(:,:), allocatable ::  B
integer N, I, J, ERRO
read*, N
allocate (A(N), B(N,N), STAT=ERRO)
if (ERRO/=0) print*, "Problemas de alocacao"
do I = 1,N,1
A(I)=N**(1./I)
enddo
do I = 1,N,1
B(I,:)= I*A
enddo
do I = 1,N,1
print*, ( B(I,J) , J=1, N, 1 )
enddo
deallocate (A, B)
end
```

4.6 Exercícios

Exercício 4.1 *Desenvolver um algoritmo que construa uma matriz identidade $n \times n$.*

Exercício 4.2 *Desenvolver um algoritmo que lê uma amostra com n valores reais e imprime média e variância.*

Exercício 4.3 *Calcular $C{=}A{\cdot}B$ para A e B matrizes quaisquer tal que o produto seja possível.*

Exercício 4.4 *Quadrado Mágico é uma tabela quadrada de lado n, onde a soma dos números das linhas, das colunas e das diagonais é constante, sendo que nenhum destes números se repete. Fazer um algorítmo que verifica se uma tabela $n \times n$ é um quadrado mágico.*

Exercício 4.5 *Fazer um algoritmo que imprime as n primeiras linhas do Triângulo de Pascal.*

Exercício 4.6 *Dado um experimento com delineamento inteiramente casualizado que testa* t *tratamentos com r repetições cada, apresentar o valor do* F *calculado para tratamento.*

5 Subprogramas e módulos

Quando um algoritmo tem muitas linhas de comandos começa a ser de difícil manipulação. Normalmente alguns programas apresentam esta catacterística e algumas instruções são muitas vezes repetidas. Assim, é possível dividir o programa em unidades menores que executam as instruções repetidas separadamente com dados ou parâmetros diferentes, ou seja, em subprogramas.

É possível escrever um programa completo em um único arquivo, ou como uma unidade simples. Contudo, se o código é suficientemente complexo, pode ser necessário que um determinado conjunto de instruções seja realizado repetidas vezes, em pontos distintos do programa.

Cada uma das unidades de programa corresponde a um conjunto completo e consistente de tarefas que podem ser, idealmente, escritas, compiladas e testadas individualmente, sendo posteriormente incluídas no programa principal para gerar um arquivo executável. Em FORTRAN há dois tipos de estruturas que se encaixam nesta categoria: subrotinas e funções (externas ou extrínsecas).

Um código executável é criado a partir de um programa principal, que pode invocar rotinas externas e usar módulos também. A única unidade de programa que deve necessariamente existir sempre é o programa principal.

Quaisquer das três unidades de programas podem também invocar rotinas internas, as quais têm estrutura semelhante às rotinas externas, porém não podem ser testadas isoladamente.

5.1 Programa principal

Todo código executável deve ser composto a partir de um, e somente um, programa principal. Opcionalmente, este pode invocar subprogramas. Um programa principal possui a seguinte estrutura:

```
program nome
declarações
comandos
contains
subprogramas internos
end program nome
```

A declaração "contains" indica a presença de subprogramas internos (funções ou subrotinas).

5.2 Funções

Uma função retorna um único valor (matriz ou escalar), e esta usualmente não altera os valores de seus argumentos. Neste sentido, uma função em FORTRAN age como uma função em análise matemática.

O FORTRAN tem dois tipos de funções, intrínsecas e definidas pelo usuário.

Funções intrínsecas são próprias (latentes) da linguagem FORTRAN, tais como sin(x), cos(x), sqrt(x), entre outras. Estas já foram abordadas em 2.5.

As funções definidas pelo usuário são funções que o programador cria para executar uma tarefa específica. Exceto pela declaração inicial, as funções apresentam uma forma semelhante a de um programa principal:

```
...
contains
function nome (argumentos)
declarações
comandos
contains
subprogramas internos
end function nome
...
```

O efeito do comando "end" em um subprograma consiste em retornar o controle à unidade que o chamou, ao invés de interromper a execução do programa. Recomenda-se o uso da forma completa do comando para deixar claro ao compilador e ao programador qual parte do programa está sendo terminada.

Nas declarações dentro da função devem aparecer: a própria função, os argumentos e variáveis auxiliares da função. Estas declarações não são declaradas novamente junto às declarações do programa principal.

O comando "contains" inserido dentro da função só é necessário se houverem subprograms internos à função.

Uma função é ativada ou chamada de forma semelhante como se usa uma função em análise matemática. Por exemplo, dada uma função "func(n,x)", esta pode ser chamada para atribuir seu valor a uma variável escalar ou a um elemento de matriz:

$$y = \text{func(n,x)}$$

Uma função pode fazer operações em uma expressão, por exemplo: y = func(n,x) + 5*func(n,x**3), ou ainda servir de argumento para uma outra rotina.

Por exemplo:

Exemplo 5.1

```
program exemplo
implicit none
real y1, y2, y3, y4
read*, y1, y2, y3, y4
print*, func(y1,y2)
print*, func(y3,y4)
contains
function func(x,y)
real func, a, b, x, y
a=1
b=2
func = a*x+b*y
end function func
end
```

5.3 Subrotinas

Uma subrotina pode executar uma tarefa mais complexa que a função e retornar diversos valores através de seus argumentos, os quais podem ser modificados ao longo da computação da subrotina.

Exceto pela declaração inicial, as subrotinas apresentam uma forma semelhante a de um programa principal:

```
...
contains
subroutine nome (argumentos)
declarações
comandos
contains
subprogramas internos
end subroutine nome
...
```

Nas declarações dentro da subrotina, devem aparecer as variáveis auxiliares e os argumentos da subrotina. Estas declarações não são declaradas novamente junto às declarações do programa principal.

Uma subrotina, devido ao fato de retornar, em geral, mais de um valor em cada chamada, não pode ser operada como uma função em análise matemática e deve ser chamada através da instrução "call". Qualquer unidade de programa pode chamar uma subrotina, até mesmo outra subrotina. Por exemplo, se existe uma sub-rotina "subrot", será obtida através da chamada:

call subrot(x1, x2, ..., xn)

A ordem e tipo dos argumentos na lista de argumentos devem corresponder à ordem e tipo dos argumentos declarados na subrotina. A subrotina finaliza sua execução quando encontra um "return" ou um "end subroutine" e, retorna ao programa que a requisitou na linha seguinte ao "call".

Por exemplo:

Exemplo 5.2

```
program exemplo
implicit none
real y1, y2, y3, y4, r1, r2
read*, y1, y2, y3, y4
call subrot(y1,y2,r1,r2)
print*, r1, r2
call subrot(y3,y4,r1,r2)
print*, r1, r2
contains
subroutine subrot(x,y,z1,z2)
real a, b, x, y, z1, z2
a=1
b=2
z1 = a*x+b*y
z2 = a*(x**2)+b*(y**2)
end subroutine subrot
end
```

5.4 Módulos

No que diz respeito a modularização de programas, a linguagem FORTRAN oferece facilidades através de subrotinas e funções, o que torna possível a implementação de programas modulares e estruturados. No FORTRAN 90, esta modularização teve um avanço significativo através das declarações e procedimentos "module", tanto que esta declaração tem status de programa.

A declaração "module" (ou módulo) pode conter dados, procedimentos, ou ambos, que podemos compartilhar entre unidades de programas (programa principal, subprograma e em outros módulos). A estrutura de um módulo é idêntica à de um programa, porém deve-se trocar "program" por "module". Os dados e procedimentos estarão disponíveis para uso na unidade de programa através da declaração "use", seguida do nome do módulo.

Por exemplo:

Exemplo 5.3

```
module funcao
implicit none
contains
subroutine raizes (a1,b1,c1,raiz1,raiz2)
real a1, b1, c1
complex a, b, c, raiz1, raiz2, delta
a = cmplx(a1)
b = cmplx(b1)
c = cmplx(c1)
delta = b*b - 4*a*c
raiz1 = (-b + sqrt(delta))/(2*a)
raiz2 = (-b - sqrt(delta))/(2*a)
end subroutine raizes
end module funcao
```

Que deve ser compilado como se fosse um programa principal. Assim, é possível criar programas no mesmo espaço de trabalho (diretório) que o utilize:

Exemplo 5.4

```
program exemplo
use funcao
implicit none
real a, b, c
complex raizum, raizdois
read*, a, b, c
call raizes(a,b,c,raizum,raizdois)
print*, raizum, raizdois
end
```

5.5 Exercícios

Exercício 5.1 *Faça uma subrotina que receba uma matriz M(10,10), o número de uma linha L, o número de uma coluna C e retorne a matriz N(9,9) resultante da remoção da linha L e da coluna C.*

Exercício 5.2 *Escreva um programa que jogue o jogo da velha com o usuário, o qual deve ter a seguinte estrutura: inicializar a matriz 3×3 com zeros; pedir para o jogador escolher o seu símbolo (X ou O); pedir a jogada do usuário; gerar a jogada do computador (que simplesmente deve preencher com o seu símbolo o primeiro espaço vazio que encontrar (ele não é muito inteligente!); mostrar a matriz alterada na tela; verificar se há vencedor (linhas, colunas ou diagonais com um mesmo símbolo) e anunciá-lo; caso contrário, pedir nova jogada ao usuário, etc. Depois que um jogador vencer, o programa deve perguntar se o usuário quer jogar novamente. Se a resposta for negativa, terminar o programa.*

Exercício 5.3 *O número de cadastro de pessoas físicas do Ministério da Fazenda (CPF) tem 9 dígitos seguidos de dois dígitos verificadores, os quais servem como teste para erros de digitação*

na sequência. Dada a sequência dos 9 dígitos $(n1, ..., n9)$ *o primeiro dígito verificador* $(dv1)$ *é gerado seguindo-se a regra: a) calcula-se a soma* $s1 = 10 \times n1 + 9 \times n2 + ... + 3 \times n8 + 2 \times n9$*; b) calcula-se o resto* $r1$ *da divisão de* $s1$ *por 11; c) subtrai-se* $r1$ *de 11; d) se* $dv1$ *resultar 10 ou 11, transforme para 0. O segundo dígito verificador* $(dv2)$ *é gerado usando-se o* $dv1$*: calcula-se a soma* $s2 = 11 \times n1 + 10 \times n2 + ... + 4 \times n8 + 3 \times n9 + 2 \times dv1$ *e seguem-se os demais passos de forma semelhante. Escreva um programa para verificar se um CPF, dado numa sequência de 11 dígitos, sem pontos ou hífen, é válido, ou seja, não contém erros de digitação. (Usar módulo).*

Exercício 5.4 *Escreva um programa que lê um número não determinado de valores m, todos inteiros e positivos, um valor de cada vez, e, se* $m < 10$ *utiliza um sub-programa do tipo função que calcula o fatorial de m, e caso contrário, utiliza um sub-programa do tipo função para obter o número de divisores de m (quantos divisores m possui). Escrever cada m lido e seu fatorial ou seu número de divisores com uma mensagem adequada. Neste caso, temos um programa principal e dois sub-programas.*

6 Entrada e saída de dados

Entrada e saída de dados é uma parte de fundamental importância na programação. O FORTRAN 90 possui uma grande variedade de opções de I/O (input/output), que permitem diferentes tipos de arquivos se conectarem ao programa principal para leitura e gravação.

6.1 I/O simples

Neste material foram utilizados, até agora, para leitura e impressão de dados no ecrã (tela do computador), respectivamente os comandos "read*," e "print*," porém, outras formas mais elaboradas podem ser utilizadas.

Leitura: read (*n° unidade* , *n° formato / código do formato*) *variáveis*

Impressão: write (*n° unidade* , *n° formato / código do formato*) *variáveis*

Em que, *n° unidade* corresponde ao número que representa um ficheiro (arquivo) ou um ecrã, sua utilização será descrita à frente. E *n° formato* corresponde ao número que representa um rótulo para declaração de formato ou pode ser substituído pelo próprio código do formato sem uso de declaração de formato.

Para simplificação destas declarações, utiliza-se "*" em vez de números, isto é utilizado quando se deseja ler e imprimir variáveis ou textos no ecrã e não há preocupação com formatação:

Leitura: read (*,*) *variáveis*

Impressão: write (*,*) *variáveis*

Para utilização de formatação dos dados, utiliza-se a sintaxe:

```
...
write (*, '(código do formato)') variáveis
...
```

ou

```
...
write (*, n° formato) variáveis
...
n° formato format (código do formato)
...
```

Os códigos de formatação mais utilizados são:

Tabela 10: Códigos de formatação mais utilizados.

Código	Representação
Iw	Dado inteiro com largura total de campo w
Iw.m	Dado inteiro com largura total de campo w e numero mínimo de caracteres m
Fw.d	Dado real com largura total de campo w e d casas decimais
nX	n espaços horizontais
Ew.d	Dado real em notação exponencial com largura total de campo w e d casas decimais
Aw	Dado de caractere com largura de campo
pEw.d	Dado real com p números antes da vírgula, em notação exponencial com largura total de campo w e d casas decimais
n/	n espaços verticais (saltar linha)

Se um número ou texto não preencher o tamanho do campo declarado, serão somados espaços. Normalmente o texto será ajustado à direita, mas as regras variam pra formatações diferentes.

Exemplo 6.1

```
program exemplo
implicit none
real x
integer n
character*19 c
x = 13.760
n = 276
c = "Programa em Fortran"
write(*,23) c
write(*,900) x
write(*,1) n
write(*,'(5/)')
write(*,'(10x,A19)') c
write(*,'(10x,F8.4)') x
write(*,'(10x,E8.3,10x,I10.2)') x, n
write(*,'(10x,I10.2)') n
write(*,'(10x,I8)') n
write(*,'(5/)')
23 format (15x,A8)
1 format (15x,I6)
900 format (15x,F8.1)
end
```

6.2 Ficheiros

É possível, através de comandos em FORTRAN, ler ou imprimir em ficheiros, que são armazenados em dispositivos de armazenamento de dados. Inicialmente, é necessário efetuar a abertura de um arquivo, já existente ou não.

Um programa pode gerar tantos dados, que todos eles não caberiam na tela de uma só vez, e ainda seriam perdidos ao finalizar o programa. Os dados salvos em arquivos podem ser usados pelo próprio programa ou exportados para serem processados de outra forma. Arquivos de leitura economizam um tempo precioso para o usuário do programa, pois ele não vai precisar enviar dados via teclado, e com arquivos milhares de dados podem ser lidos em segundos. Além de que, não é necessário executar o programa sempre que se desejar consultar dados de saída em arquivos,

A sintaxe básica para abertura, criação ou substituição de um ficheiro é:

```
...
open(n° unidade, file="nome do arquivo")
...
close(n° unidade)
...
```

Em que "open" faz a chamada, se já existente, ou a criação do arquivo "*nome do arquivo*" e atribui a este um inteiro "*n° unidade*" que o representará nos comandos do algoritmo. Quando não for utilizado o comando "open", o programa emitirá uma mensagem na tela pedindo o seu nome, podendo o usuário escolher um nome diferente a cada vez que o programa for executado. Todos os arquivos devem estar ou serão criados no mesmo diretório em que estiver

o programa. Outros comandos opcionais podem ser inseridos em "open" de acordo com cada compilador.

E, "close" efetua o fechamento do arquivo representado por "*n° unidade*". Outros comandos opcionais podem ser inseridos em "close" dacordo com cada compilador.

Para impressão em um arquivo representado por "*n° unidade*", é utilizado o comando "write (*n° unidade* , *n° formato / código do formato*) *variáveis*" após a abertura ("open") e antes do fechamento ("close") do arquivo.

Por exemplo:

Exemplo 6.2

```
program exemplo
implicit none
integer n, i
read*, n
open(503,file="arquivo.txt")
do i=1,n,1
write (503,'(3x,I5)') i
enddo
close(503)
end
```

Exemplo 6.3

```
program exemplo
implicit none
integer n, j
real i
read*, n
open(10, file="arquivo.dat")
do i=1,n,1
write (10, '(10000(1x,f5.0))') ( i , j=1,n,1 )
print*, i
enddo
close (10)
end
```

Alguns comandos extras podem ser inseridos caso necessário:

Tabela 11: Comandos extras de formatação.

Comando	Representação
rewind (n° unidade)	Recuo total
backspace (n° unidade)	Recuo de um campo

Apesar de se poder usar qualquer extensão de arquivo ou até omití-la, as extensões .dat para leitura e .out para saída são mais comumente encontradas.

Por exemplo:

Exemplo 6.4

```
program exemplo
implicit none
open(10, file="arquivo.txt")
write(10,*) "Texto"
write (10, '(1x,f6.3)') 12.234
write (10, '(1x,f6.3)') 23.876
rewind (10)
write (10, '(1x,f6.3)') 56.789
write (10, '(1x,f6.3)') 67.123
backspace (10)
write (10, '(1x,f6.3)') 77.153
close (10)
end
```

6.3 Exercícios

Exercício 6.1 *Fazer um algorítmo que leia 10 valores reais e imprima uma tabela cuja primeira coluna seja formada por estes números, a segunda coluna apresente a parte inteira desses valores e a terceira coluna apresente estes valores em notação científica. NOTA: Considerar, por facilidade, valores de 0 a 100 com um máximo de três casas decimais.*

Exercício 6.2 *Criar um arquivo txt contendo uma matriz identidade 500×500.*

Exercício 6.3 *Desenvolver um programa que cria um arquivo txt com uma sequência de 1000 números aleatórios. Cada número deve estar entre 0 e 1, conter duas casas decimais e os números devem ser impressos numa mesma linha com três espaços horizontais entre eles. NOTA: Para geração de números aleatórios utilizar o comando "ran(n)", em que n é um inteiro qualquer informado que serve como semente.*

Exercício 6.4 *Desenvolver um programa que leia os dados do arquivo criado no exercício anterior e apresente na tela de execução a porcentagem de elementos menores que 0,5.*

Exercício 6.5 *Desenvolver um programa que resolva um SUDOKU espresso por um arquivo com extensão txt e imprima a resolução em outro arquivo txt.*

Exercício 6.6 *Desenvolver um programa que gere, aleatoriamente, um SUDOKU não resolvido de solução única e o imprima em um arquivo txt.*

7 Algoritmos dos exercícios

Serão apresentados alguns dos algorítmos em FORTRAN exigidos nos exercícios. É importante salientar que será apresentada apenas uma forma de resolver cada problema, assim, não existe um gabarito para os exercícios e cada programador terá, provavelmente um programa diferente que resolva o mesmo problema. Espera-se do leitor a compreensão dos procedimentos realizados de modo a ampliar seus horizontes às próprias criações e a resolver os problemas em sua determinada área.

```
program exercicio_2_2
implicit none
real a1, a2, a3, a4, a5, m, v
read *, a1, a2, a3, a4, a5
m = (a1 + a2 + a3 + a4 + a5)/5
v=(((a1-m)**2)+((a2-m)**2)+((a3-m)**2)+((a4-m)**2)+((a5-m)**2))/4
print*, "media", m
print*, "variancia", v
end
```

```
program exercicio_2_3
implicit none
integer m,r,s,t
read *, m
r = m/100
s = m-(r*100)
t = r + s
print*, r, s, t
end
```

```
program exercicio_2_4
implicit none
real a1, b1, c1
complex a, b, c, raiz1, raiz2, delta
read*, a1, b1, c1
a = CMPLX(a1)
b = CMPLX(b1)
c = CMPLX(c1)
delta = b*b - 4*a*c
raiz1 = (-b + sqrt(delta))/(2*a)
raiz2 = (-b - sqrt(delta))/(2*a)
print*, raiz1, raiz2
end
```

```
program exercicio_2_5
implicit none
real a, b
integer q, r
read*, a, b
q = a / b
r = a-(b*q)
print*, "quociente:  ", q
print*, "resto:  ", r
end
```

```
program exercicio_2_6
implicit none
integer m,r,s
logical resp
read*, m
r = m/100
s = m-(r*100)
resp = ((r+s)**2) == m
print*, resp
end
```

```
program exercicio_2_7
implicit none
integer p, q, a, b, c, d, e, f, g
logical r
read*, p, q
a=(q/1000000)
b=(q/100000)-(a*10)
c=(q/10000)-(a*100)-(b*10)
d=(q/1000)-(a*1000)-(b*100)-(c*10)
e=(q/100)-(a*10000)-(b*1000)-(c*100)-(d*10)
f=(q/10)-(a*100000)-(b*10000)-(c*1000)-(d*100)-(e*10)
g=q-(a*1000000)-(b*100000)-(c*10000)-(d*1000)-(e*100)-(f*10)
r=10*a+b==p.or.10*b+c==p.or.10*c+d==p.or.10*d+e==p.or.10*e+f==p.or.10*f+g==p
print*, r
end
```

```
program exercicio_3_1
implicit none
integer n, a, b, c, i
a=1
b=1
read*, n
if (n==1) then
print*, a
else
print*, a
print*, a
do i=3,n,1
print*, a+b
c=b
b=a+b
a=c
enddo
endif
end
```

```
program exercicio_3_2
implicit none
integer n, nfat, i
read*, n
nfat=1
if (n>0) then
do i=1,n,1
nfat=nfat*i
enddo
endif
if (n>=0) then
print*, nfat
endif
end
```

```
program exercicio_3_3
implicit none
real a, b, c
read*, a, b, c
if (min(a,b,c)>0 .and.  a+b>c .and.  a+c>b .and.  b+c>a) then
if (a==b .and.  b==c) then
print*, "O triangulo e equilatero"
elseif (a==b .or.  b==c .or.  a==c) then
print*, "O triangulo e isosceles"
else
print*, "O triangulo e escaleno"
endif
else
print*, "Nao pode ser um triangulo"
endif
end
```

```
program exercicio_3_4
implicit none
integer n, i, cont
do n=1,1000,1
cont=0
do i=1,n,1
if (mod(n,i)==0) then
cont=cont+1
endif
enddo
if (cont==2) then
print*, n
endif
enddo
end
```

```
program exercicio_3_5
implicit none
integer m, n, i
read*, n
m=n
print*, n, " pode ser escrito como produto de:"
20 do i=2,m,1
if (mod(n,i)==0) then
print*, i
n=n/i
goto 20
endif
enddo
end
```

```
program exercicio_3_6
implicit none
integer m, r, s
do m=1000,9999,1
r = m/100
s = m-(r*100)
if (((r+s)**2) == m) then
print*, m
endif
enddo
end
```

```
program exercicio_3_7
implicit none
real soma, nfat
integer n, i
n=0
soma=0
do while (soma<2.718281)
nfat=1
if (n>0) then
do i=1,n,1
nfat=nfat*i
enddo
endif
soma=soma+(1/nfat)
n=n+1
enddo
print*, "Valor somado", soma
print*, "x =", n-1
end
```

8 Bibliografia consultada

- http://paginas.fe.up.pt/~aarh/pc/PC-apontamentos.htm
- http://www.orengonline.com/arquivos/mcf90.pdf
- http://www.fisica.uece.br/graduacao/caf/sites/default/files/80_exercicios.doc
- http://www.inf.ufes.br/~thomas/fortran/tutorials/inpe_fortran.pdf
- http://www.cenapad.unicamp.br/servicos/treinamentos/apostilas/apostila_fortran90.pdf
- http://www.inf.ufes.br/~thomas/fortran/tutorials/helder/fortran.pdf
- http://minerva.ufpel.edu.br/~rudi/grad/ModComp/Apostila/
- http://astro.uesc.br/~apaula/Tutorial_fortran.pdf
- http://www.fis.ufba.br/~edmar/fortran/abel/HOME%20MAT045%20-%20HOME%20PAGE/APOSTILAS%20FORTRAN%20FORMATO%20WORD
- http://pt.scribd.com/doc/28356002/VisuAlg-Ref
- http://www.orengonline.com/computacao_fortran.html
- http://www.nr.com/oldverswitcher.html