

Capítulo 2

Elementos do Fortran 90

2.1 Introdução

Neste capítulo vamos apresentar os elementos básicos da linguagem de programação Fortran 90. Como exemplo, vamos considerar o seguinte problema: escrever um programa que pergunte ao usuário as coordenadas x e y de três pontos e que calcule a equação do círculo que passa por esses três pontos, a saber:

$$(x - a)^2 + (y - b)^2 = r^2$$

e então mostre as coordenadas (a, b) do centro do círculo e seu raio, r .

Este problema é aparentemente trivial, mas não é tão simples assim! O que ocorre se, por exemplo, os três pontos fornecidos estiverem ao longo de uma reta?

2.1.1 Análise do problema

Basicamente, a análise consiste em criar um *plano estruturado* que envolva níveis sucessivos de refinamento até **atingir um ponto em que o programador possa codificar cada passo individual**. Um exemplo de plano estruturado para o problema acima seria:

- a) Ler os três conjuntos de coordenadas (x_1, y_1) , (x_2, y_2) , (x_3, y_3) ;
- b) Resolver a equação do círculo;
- c) Mostrar as coordenadas (a, b) e o raio r

As partes a) e c) são triviais, mas a parte b) requer uma análise mais aprofundada e será abordada no capítulo 4. No momento, lançaremos mão de um recurso comum (além de útil e importante) em programação, que é mover b) para uma **subrotina** (procedimento externo). Dessa forma, um possível código em Fortran 90 estruturado de acordo com o plano acima seria (arquivo `circulo.f90`)

```

PROGRAM circulo
  IMPLICIT NONE

!Esse programa calcula a equacao de um circulo passando por 3
!pontos fornecidos pelo usuario.
!Utiliza-se da subrotina calcula_circulo

!Declaracao de variaveis
  REAL :: x1, y1, x2, y2, x3, y3, a, b, r

!Passo 1: le coordenadas
  PRINT*,"Entre com a coordenada dos tres pontos"
  PRINT*,"na ordem x1,y1,x2,y2,x3,y3"

  READ*,x1,y1,x2,y2,x3,y3

!Passo 2: chama subrotina calcula_circulo
  CALL calcula_circulo(x1,y1,x2,y2,x3,y3,a,b,r)

!Passo 3: escreve resultado na tela
  PRINT*,"O Centro do circulo que passa por esses &
        &pontos eh (",a,",",b,")"
  PRINT*,"Seu raio eh ",r

END PROGRAM circulo

```

2.2 Estrutura Básica

Um programa em fortran 90 tem a seguinte estrutura básica:

```

PROGRAM nome
  IMPLICIT NONE

  [parte de especificação]
  [parte de execução]
  [parte de subprogramas]

END PROGRAM nome

```

2.2.1 Parte de Especificação

```

REAL :: x1, y1, x2, y2, x3, y3, a, b, r

```

Na parte de especificação, logo após o `IMPLICIT NONE`, é onde se faz a **declaração de variáveis** (e outros tipos de declaração). Variáveis são basicamente endereços lógicos a trechos de memória onde uma determinada informação está armazenada.

Veremos adiante que o computador é capaz de manipular diferentes tipos de variáveis. Os tipos mais comuns são:

```
REAL ::  
DOUBLE PRECISION ::  
INTEGER ::  
CHARACTER ::  
LOGICAL ::  
COMPLEX ::
```

O IMPLICIT NONE deve estar sempre presente! Ele força o programador a declarar explicitamente todas as variáveis, o que é muito importante pois evita uma série de possíveis erros de programação.

2.2.2 Comentários

Linhas cujo primeiro caractere não branco é ! são consideradas comentários e são ignoradas pelo compilador. Comentar bem o código é uma boa prática.

2.2.3 Parte de Execução

```
PRINT *, "Entre com a coordenada dos tres pontos"  
PRINT *, "na ordem x1,y1,x2,y2,x3,y3"  
  
READ *, x1,y1,x2,y2,x3,y3  
  
!Passo 2: chama subrotina calcula_circulo  
CALL calcula_circulo(x1,y1,x2,y2,x3,y3,a,b,r)
```

Na parte de execução estão todos os comandos que devem ser executados pelo computador (PRINT, READ, CALL, etc) e que perfazem o chamado **algoritmo**. Criar um algoritmo numérico que seja representável pelo conjunto de comandos disponíveis em uma determinada linguagem corresponde ao cerne do **método numérico**.

2.2.4 Caractere de continuação

```
PRINT *, "O Centro do circulo que passa por esses &  
      &pontos eh (" ,a, " , " ,b, " )"
```

Se o último caractere de uma linha for &, isso diz ao compilador que o comando continua na próxima linha. Ele é usado em dois contextos:

- Quebrar uma string de caracteres longa. Se o & estiver dentro de uma string de caracteres, então deve haver outro & no início da linha seguinte;
- Quebrar linhas de código, de forma a torná-lo mais legível. Ex:

```
CALL calcula_circulo(x1,y1,x2,y2,x3,y3,&!quebrei a linha  
                    a,b,r)
```

2.3 Compilando o código

Uma vez tendo o código pronto, devemos **compilá-lo**. Quando dizemos “compilar” um código nos referimos, na verdade, a dois procedimentos distintos.

O primeiro (compilação propriamente dita) é a análise do código pelo compilador, que vai ler e analisar cada linha válida de código (i.e., linhas não vazias e que não comecem com !), procurando verificar se há algum **erro de sintaxe**. Quando isso ocorre, o compilador retorna um ou mais **erros de compilação**. Por exemplo, o código abaixo retornará um erro de compilação:

```
PRINT , "Entre com a coordenada dos tres pontos "
```

Os erros de compilação não têm grande consequência, pois são facilmente detectados pelo compilador. Já **erros semânticos** podem resultar em **erros de execução** (divisão por zero, por exemplo) ou simplesmente podem passar despercebidos pelo compilador, gerando um resultado incorreto.

Por exemplo, imaginemos um programa de computador capaz de analisar uma frase e verificar se a gramática está correta. Se lhe fornecermos a frase “O gato está sobre o mesa”, o programa certamente apontará o erro gramatical (“o mesa”). Entretanto, se a frase analisada for “A mesa está sobre o gato”, o programa dirá que a frase está gramaticalmente correta, apesar de semanticamente não fazer sentido.

A segunda parte do processo é chamada **linking**, em inglês. Nela, o compilador vai gerar o **código binário** que poderá ser executado pelo compilador.

2.4 Tipos de dados

Em matemática, assim como em programação, há dois tipos básicos de números: **os que são inteiros e os que não o são**.

Em programação, a diferença entre esses dois tipos é **absolutamente vital**, pois está ligada à maneira como os números são representados e armazenados na memória e manipulados pelo processador. Isso será visto em detalhes mais adiante (capítulo 3), no momento basta considerar que:

- Uma variável **INTEGER** é um número inteiro, sempre **armazenado de forma exata** na memória do computador, e tem um intervalo possível de valores relativamente baixo, tipicamente entre $-2,1 \times 10^9$ e $2,1 \times 10^9$ em um computador de 32 bits;
- Um número **REAL**, por outro lado, é armazenado como uma **aproximação** com um **número fixo de algarismos significativos (7 ou 8)** e tem um intervalo bem maior, tipicamente entre -10^{38} e $+10^{38}$ em um computador com 32 bits;
- Um número **DOUBLE PRECISION** é uma representação usando 64 bits com tipicamente 16 algarismos significativos e um intervalo entre -10^{308} e $+10^{308}$.

Existem várias funções intrínsecas do Fortran que retornam as propriedades dos números que podem ser representados por um dado processador: **EPSILON**, **HUGE**, **TINY**, etc. Isso será visto em detalhes no próximo capítulo, mas o código abaixo mostra como usar tais funções para mostrar os limites da representação numérica em um dado computador (arquivo `funcoes_numericas.f90`).

```

REAL :: x
DOUBLE PRECISION :: y
INTEGER :: i
INTEGER*8 :: j

PRINT*, &
"O menor numero positivo que somado a 1 retorna numero maior&
& que 1: "
PRINT*, EPSILON(x), EPSILON(y)

PRINT*, &
"O maior numero positivo que pode ser representado: "
PRINT*, HUGE(x), HUGE(y), HUGE(i), HUGE(j)

PRINT*, &
"O menor numero positivo que pode ser representado: : "
PRINT*, TINY(x), TINY(y)

```

Saída do programa `funcoes_numericas.f90`:

```

>>O menor numero positivo que somado a 1 retorna numero maior
>>que 1:
>>1.1920929E-07  2.220446049250313E-016

>>O maior numero positivo que pode ser representado:
>>3.4028235E+38  1.797693134862316E+308  2147483647  9223372036854
>>775807

>>O menor numero positivo que pode ser representado:
>>1.1754944E-38  2.225073858507201E-308

```

Saber os limites da representação numérica de um processador é muito importante. Se durante a execução de um programa uma operação resultar em um número fora do limite a ser representado, ocorre a chamada **exceção de ponto flutuante**. O que acontece depois depende da forma como o programa foi compilado.

2.5 Expressões aritméticas

Uma vez declaradas as variáveis, pode-se atribuir valores a elas de várias formas:

- Usando o `READ`;
- Atribuição direta. Ex: `x = 2.2`;
- Através de uma **expressão aritmética**.

Ex: `a = b + c*d/e - f**g/h + i * j + k`

Os operadores aritméticos em Fortran são:

`+` `-` `*` `/` `**` (exponenciação)

Um ponto de fundamental importância é que o Fortran executa as operações acima de acordo com as prioridades abaixo:

1. exponenciação;
2. multiplicação e divisão;
3. adição e subtração.

Dentro do mesmo nível de prioridade, a conta será feita **da esquerda para a direita**, com exceção da exponenciação, que é feita da direita para a esquerda. Por exemplo, vamos estudar como o computador executa a seguinte expressão aritmética:

```
a = b + c*d/e - f**g/h + i * j + k
```

Os passos são:

1. Calcula $f^{**}g$ e salva em temp1
2. Calcula $c*d$ e salva em temp2
3. Calcula $temp2/e$ e salva em temp3
4. Calcula $temp1/h$ e salva em temp4
5. Calcula $i*j$ e salva em temp5
6. Calcula $b+temp3$ e salva em temp6
7. Calcula $temp6-temp4$ e salva em temp7
8. Calcula $temp7+temp5$ e salva em temp8
9. Calcula $temp8+k$ e salva em a.

Em cálculo numérico todo o cuidado é pouco. Os exemplos a seguir ilustram os cuidados que devem ser tomados para se evitar erros semânticos que podem resultar em erros numéricos.

2.5.1 Exemplo 1

Qual o resultado do código abaixo?

```
INTEGER :: i,j  
  
i = 1  
j = 3  
  
PRINT*, "Resultado = ", i/j  
  
>>Resultado = 0
```

Por quê? i e j são inteiros, portanto o que foi feito é uma divisão entre números inteiros, cujo resultado é um inteiro truncado. Outros exemplos:

$5/2 = 2$, $10/3 = 3$, $1999/1000 = 1$, etc.

2.5.2 Exemplo 2

Qual o resultado do código abaixo?

```
REAL :: x
x = 1/3
PRINT*, "Resultado = ", x

>>Resultado = 0.000000
```

Por quê? Para o compilador, números sem o ponto (Ex: 1, 3) são inteiros. Dessa forma o que foi feito foi uma **divisão de inteiros** que resultou em um inteiro que então foi convertido em um número real. **Importante:** o computador sempre faz exatamente o que lhe é dito para fazer...

Para obter o resultado esperado, a sintaxe deveria ser:

```
x = 1./3.    ou x = 1.0/3.0    ou x = 1.E0/3.E0
...
>>Resultado = 0.333333
```

2.5.3 Exemplo 3

Qual o resultado do código abaixo?

```
REAL :: x
x = 1/3.
PRINT*, "Resultado = ", x

>>Resultado = 0.333333
```

Por quê? Quando há diferentes tipos de dados em uma (sub)expressão, o compilador promove um tipo para outro de acordo com a seguinte prioridade:

1 - INTEGER (baixa), 2 - REAL, 3 - DOUBLE, 4 - COMPLEX (alta)

Assim, na expressão acima, o 1, sendo inteiro, foi promovido a real, e a expressão foi calculada entre dois reais.

2.5.4 Exemplo 4

Qual o resultado do código abaixo?

```
REAL :: x,y
INTEGER :: i,j

i = 3.9
x = 3.9
y = 0.1
j = x+y
```

```
PRINT*, "Resultado = ", i,j
>>Resultado = 3 4
```

Porque? Inteiros são sempre truncados em direção ao zero. No caso de j, a operação de truncamento foi feita após a execução da expressão à direita.

Para se obter o inteiro mais próximo, o Fortran tem uma função específica para isso:

```
i = NINT(3.9)
>> Resultado = 4
```

2.5.5 Exemplo 5

Qual o resultado do código abaixo?

```
REAL :: a,b
INTEGER :: c,d

b = 100.
c = 9
d = 10.
a = b*c/d

PRINT*, "Resultado = ", a
a = c/d*b
PRINT*, "Resultado = ", a

>>Resultado = 90.000000
>>Resultado = 0.000000
```

Em cálculo numérico **não necessariamente valem várias propriedades das operações aritméticas** em matemática, tais como associatividade e distributiva (ver capítulo 3). Em outras palavras, **a ordem dos fatores altera o resultado!**

2.6 List-directed input and output

Vimos atrás a função dos comandos READ e PRINT.

```
REAL :: a, b
INTEGER :: c

READ*, a, b, c
PRINT*, a, b, c
```

O “*” significa que os dados impressos ou lidos serão formatados automaticamente de acordo com a lista de variáveis apresentadas (list-directed). O formato atribuído depende do tipo da variável

Ex: se a=1.1, b=2.2 e c=10, o PRINT acima retorna:

```
>>      1.100000      2.200000      10
```

O que ocorre se o input do programa for: 1.1 2.2 10.? Ocorre um erro, pois o programa está esperando que o terceiro número seja um número inteiro.

2.7 Procedimentos, subprogramas e funções

Um procedimento é uma seção especial de um programa que é chamada, de alguma forma, sempre que necessário. Procedimentos podem ser intrínsecos (ou seja, parte do Fortran) ou escritos pelo programador ou por terceiros. Há uma categorização adicional, dependendo da forma como esses procedimentos são usados: funções e subrotinas.

Basicamente, o propósito de uma função é tomar um ou mais valores (ou argumentos) e criar um único resultado. O Fortran tem inúmeras **funções intrínsecas**:

```
!Argumento de funcoes trigonometricas devem estar em radianos!  
SIN(x)  
LOG(x)  
LOG10(x)  
SQRT(x)  
EXP(x)  
etc.
```

Uma lista das funções intrínsecas do Fortran 90 pode ser encontrada em <http://www.nsc.liu.se/boem/f77to90/a5.html>.

2.8 Funções externas

A sintaxe básica para se criar uma função externa no Fortran é:

```
<tipo> FUNCTION nome (arg1, arg2, ..., argn)  
  IMPLICIT NONE  
  
  [parte de especificação]  
  [parte de execução]  
  
END FUNCTION nome
```

Funções são usadas/criadas por vários motivos:

1. Organização do código: programação modular;
2. Evitar duplicação de código (funções podem ser chamadas várias vezes);
3. Uso de códigos de terceiros.

Vale lembrar que a filosofia por trás dos procedimentos é que o programa principal não precisa saber nada sobre os detalhes internos do procedimento: **compartimentalização do código**.

2.8.1 Exemplos

Função que calcula a raiz cúbica de um argumento (arquivo `raiz_cubica.f90`).

```
REAL FUNCTION raiz_cubica(x)
IMPLICIT NONE

!Declaracao do argumento da funcao
  REAL, INTENT(in) :: x

!Variaveis locais
  REAL :: log_x

!Calcula a raiz cúbica usando logaritmos
  log_x = LOG(x)
  raiz_cubica = EXP(log_x/3.)

END FUNCTION raiz_cubica
```

Note que:

- `log_x` é uma variável interna, não é vista pelo programa principal
- o atributo `INTENT(in)` diz ao computador que a variável `x` não pode ser modificada durante a execução.

Este outro exemplo mostra uma função que lê um valor de uma variável inteira. Note que essa função não tem argumento.

```
INTEGER FUNCTION le_valor()
IMPLICIT NONE

!Le numero
  PRINT*, "Por favor entre um numero inteiro: "
  READ*, le_valor

END FUNCTION le_valor
```

Tal função pode ser chamada pelo programa principal da seguinte forma, por exemplo:

```
soma = le_valor()*le_valor()
```

2.9 Subrotinas

A diferença entre uma subrotina e uma função está na forma como o programa se refere a ela e como os resultados (se houver) são retornados.

Subrotinas são acessadas pelo comando `CALL`, da seguinte forma:

```
CALL nome(arg1, arg2, ...)
```

O CALL causa uma **transferência de controle** para a subrotina, de forma que o programa é interrompido temporariamente e as instruções internas da subrotina são executadas.

A lista de argumentos de uma subrotina pode conter argumentos de entrada (input), de saída (output), ou nenhum argumento. A sintaxe básica é:

```
SUBROUTINE nome (arg1, arg2, ..., argn)
  IMPLICIT NONE
  [parte de especificação]
  [parte de execução]
END SUBROUTINE nome
```

No arquivo `raizes.f90` encontra-se o exemplo abaixo, que é uma subrotina que calcula a raiz quadrada, cúbica, quádrupla e quántupla de um número.

```
SUBROUTINE raizes(x,x2,x3,x4,x5)
IMPLICIT NONE

!Declaracao do input
  REAL, INTENT(in) :: x
!Declaracao do output
  REAL, INTENT(out) :: x2,x3,x4,x5
!Variaveis locais
  REAL :: log_x

!Calcula raiz quadrada
  x2 = SQRT(x)

!Calcula as outras raizes usando logaritmos
  log_x = LOG(x)

  x3 = EXP(log_x/3.)
  x4 = EXP(log_x/4.)
  x5 = EXP(log_x/5.)

END SUBROUTINE raizes
```

Exemplo de programa que chama a subrotina raizes:

```
PROGRAM exemplo
IMPLICIT NONE

  REAL :: arg, raiz2, raiz3, raiz4, raiz5

  PRINT*, "Entre com valor do argumento: "
  READ*, arg

!Chama a subrotina
  CALL raizes(arg, raiz2, raiz3, raiz4, raiz5)
```

```
PRINT*, "Raizes = ", raiz2, raiz3, raiz4, raiz5
END PROGRAM
```

Rodando:

```
>>Entre com valor do argumento:
10.
>>Raizes = 3.162278    2.154435    1.778279    1.584893
```

Observação importante sobre subrotinas: *a ordem, tipo e número de argumentos usados no programa que chama a subrotina deve corresponder exatamente ao que foi definido na subrotina*. Coisas imprevistas podem acontecer se isso não for obedecido! Por exemplo, uma subrotina com os seguintes argumentos:

```
SUBROUTINE teste(a,b,c)
...
REAL :: a,b,c
...
```

chamada pelo seguinte programa:

```
...
DOUBLE PRECISION :: a,b,c
...
CALL teste(a,b,c)
...
```

resultará em erro.

Além do `INTENT(in)` e `INTENT(out)`, existe outro atributo que pode ser usado em uma subrotina:

```
dum = INTENT(inout)
```

Esse argumento significa que a variável `dum` pode ser usada tanto para passar quanto receber informações da subrotina.

O uso dos `INTENT` é muito importante, como uma salvaguarda para erros de programação.

2.10 Controlando os passos do programa

O Fortran possui duas construções que podem ser usadas para tomadas de decisão durante o programa. São elas: `IF` e `CASE`. Antes de ver como usá-las, vamos estudar um pouco sobre variáveis `LOGICAL` e também sobre expressões lógicas.

2.10.1 variáveis e expressões lógicas

As variáveis lógicas são variáveis de 1 bit apenas: 0 significa falso e 1, verdadeiro (na verdade 1 bit é efetivamente utilizado, mas elas são armazenadas em palavras de 32 bits).

Declaração:

```
LOGICAL :: var1, var2, etc...
```

Para atribuir valores a elas, pode-se fazê-lo diretamente:

```
var1 = .TRUE.  
var2 = .FALSE.
```

ou através de expressões lógicas, que fazem uso dos chamados operadores relacionais:

```
>      !maior  
>=     !maior ou igual a  
<      !menor  
<=     !menor ou igual a  
==     !igual a  
/=     !diferente de
```

e dos operadores lógicos:

```
.NOT.   !nao logico  
.AND.   !e logico  
.OR.    !ou logico  
.EQV.   !equivalencia logica  
.NEQV.  !nao equivalencia logica
```

Exemplos de expressões lógicas:

```
var1 = 5 > 4  
var2 = 2 > 2  
var3 = SQRT(x) > 1. .AND. SQRT(x) <= 2.  
var4 = a == b .OR. a /= c  
var5 = .NOT. (a < b)  
var6 = a == b .EQV. a /= c
```

2.10.2 A construção IF

As variáveis e expressões lógicas são usadas em conjunto com a construção IF para permitir a tomada de decisões ao longo de um programa. A forma básica dessa construção é

```
IF (expressão lógica 1) THEN  
  comandos 1  
ELSE IF (expressão lógica 2) THEN  
  comandos 2  
ELSE IF (expressão lógica 3) THEN
```

```

                comandos 3
ELSE
    comandos 4
END IF

```

O número de IFs depende da problema em questão, e pode ser apenas um. Nesse caso pode-se usar uma versão simplificada:

```
IF (expressão lógica 1) comando
```

Como exemplo, vamos fazer um programa que:

1. Leia o nome de um aluno;
2. Leia as notas dos 4 EPs de AGA0503;
3. Leia as notas das duas provas;
4. Calcule a média final;
5. Escreva a média final e se o aluno foi aprovado, reprovado ou ficou em recuperação.

Uma possível solução está no programa `media_final.f90`:

```

PROGRAM media_final
!Programa que le o nome do aluno, as notas dos
!EPs e das provas, calcula a media final
!e diz se o aluno foi aprovado ou nao
!Usa a funcao le_valor
IMPLICIT NONE
!Definicao de variaveis

    CHARACTER(LEN=10) :: nome,situacao
    REAL :: EP1, EP2, EP3, EP4
    REAL :: P1, P2
    REAL :: media

    REAL, EXTERNAL :: le_valor

!Le nome do aluno
    PRINT*, "Digite o nome do aluno:"
    READ*, nome

!Le as notas dos EPs
    EP1 = le_valor("EP1")
    EP2 = le_valor("EP2")
    EP3 = le_valor("EP3")
    EP4 = le_valor("EP4")

!Le as notas das provas
    P1 = le_valor("P1 ")

```

```

        P2 = le_valor("P2 ")

!Calcula a media final
        media = 0.4*(EP1+EP2+EP3+EP4)/4. + &
                0.6*(P1+P2)/2.

!Determina situacao
        IF (media < 3.) THEN
                situacao = "reprovado"
        ELSE IF (media < 5.) THEN
                situacao = "recuperacao"
        ELSE
                situacao = "aprovado"
        ENDIF

!Escreve resultado
        PRINT*, "O aluno "//TRIM(nome)//" teve media = ", media
        PRINT*, "Sua situacao: "//situacao

END PROGRAM

REAL FUNCTION le_valor(oque)
IMPLICIT NONE

        CHARACTER(LEN=3), INTENT(in) :: oque

!Le numero
        PRINT*, "Digite a nota do "//oque//":"
        READ*, le_valor

END FUNCTION le_valor

```

2.10.3 SELECT CASE

Uma alternativa ao IF é o CASE:

```

SELECT CASE (expressão)
CASE (seletor 1)
    comandos 1
CASE (seletor 2)
    comandos 2
CASE (seletor 3)
    comandos 3
.....
CASE (seletor n)
    comandos n
CASE DEFAULT
    comandos padrao

```

```
END SELECT
```

Exemplo:

```
INTEGER :: mes

SELECT CASE (mes)
  CASE (1:2)
    PRINT*, "Ferias de verao!"
  CASE (3:6)
    PRINT*, "Aulas do primeiro semestre"
  CASE (7)
    PRINT*, "Ferias de inverno!"
  CASE (8:11)
    PRINT*, "Aulas do segundo semestre"
  CASE DEFAULT
    PRINT*, "Acho que estamos em dezembro..."
END SELECT
```

2.11 Repetindo partes de um programa

Uma grande parte das técnicas matemáticas baseia-se em alguma forma de **processo iterativo** em que a solução é atingida através de passos sucessivos.

Por exemplo, pode-se calcular o valor de π através da série

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots \quad (2.1)$$

Assim, para calcular π usando a série acima (o que, na verdade, seria uma péssima idéia...), temos que somar os termos sucessivamente até atingirmos a precisão desejada.

Além disso, o **processamento de dados** em geral requer que a mesma ação seja aplicada repetidamente para cada dado. Assim, um dos mais importantes conceitos em programação é a habilidade de repetir uma sequência de comandos, seja um número pre-determinado de vezes seja até que determinada condição seja satisfeita.

2.11.1 Loop do DO com contador

```
DO contador = inicial, final, [incremento]

  comandos

END DO
```

Notas:

- O contador e os valores à direita do = devem ser variáveis ou expressões inteiras
- O incremento ao contador é optativo

- Quando o DO é iniciado, o **número de iterações** é calculado usando-se a seguinte expressão:

$$MAX \left(\frac{final - inicial + inc}{inc}, 0 \right) \quad (2.2)$$

Quando o resultado acima é 0, o **loop não é executado**. A tabela abaixo mostra alguns exemplos do comportamento do DO para diferentes argumentos. Note que mesmo no caso em que nada é executado pelo loop (última linha da tabela, em que o número de iterações é 0) é atribuído um valor do contador (i).

DO	Número de iterações	Valores do contador
DO i=1,10	10	1,2,3,4,5,6,7,8,9,10
DO j=20,50,5	7	20,25,30,35,40,45,50
DO j=20,-20,-6	7	20,14,8,2,-4,-10,-16
DO i=4,5,4	1	4
DO i=6,1	0	6

Exemplo de uma função inteira que calcula o fatorial de um número inteiro (arquivo `fatorial.f90`):

```

INTEGER FUNCTION fatorial(arg)
IMPLICIT NONE

  INTEGER, INTENT(in) :: arg
  INTEGER :: i !Contador do loop

!Caso em que arg eh 0
  IF (arg == 0) THEN
    fatorial = 1
  ELSE
    fatorial = 1
    DO i=1,arg
      fatorial = fatorial*i
    ENDDO
  ENDIF
END FUNCTION fatorial

```

2.11.2 Comando EXIT

O EXIT permite terminar o loop quando alguma condição for satisfeita. Esse tipo de construção é muito útil quando se usa métodos iterativos para calcular algo com precisão pré-determinada

```

DO
.
.
  IF (precisao < epsilon) EXIT
.

```

```
END DO
!Após o EXIT o programa segue seu curso normal
```

O uso do código acima é perigoso pois pode-se incorrer num loop infinito, caso a condição do IF nunca seja satisfeita. Para contornar este problema basta impor um número máximo de iterações, como no exemplo abaixo:

```
DO i=1,max_iteracoes
.
.
IF (precisao < epsilon) EXIT
.
END DO
```

2.11.3 Exercício

Escrever um programa que calcule o valor de π usando a série da Eq. (2.1), com a precisão definida pelo usuário e que imprima o resultado e o número de termos que foram somados.

Como definir se a precisão foi atingida?

2.12 Arrays

Em computação científica é frequentemente necessário manipular conjuntos de dados ordenados, tais como vetores e matrizes. Outra tarefa comum é repetir a mesma sequência de operações em um conjunto de dados.

Ambos requerimentos são atendidos em Fortran pelo uso de *arrays*, que permitem que um conjunto de dados seja acessado simultaneamente (como um único objeto).

Há duas formas de se declarar um array:

```
REAL , DIMENSION (50) :: a
REAL :: a(50)
```

Ambas as formas são equivalentes. Para imprimir os elementos 11 a 14 de *a* basta escrever:

```
PRINT*, a(11:14)
```

Por padrão o primeiro índice de um array é 1, mas isso pode ser modificado, como no exemplo abaixo:

```
REAL , DIMENSION (-10:1) :: a
```

Em Fortran, pode ocorrer confusão entre arrays e funções pois a sintaxe é semelhante. Considere o exemplo abaixo:

```

REAL, DIMENSION(10) :: primeiro
REAL :: segundo
INTEGER :: i
REAL :: a, b

...
a = primeiro(i)
b = segundo(i)
...

```

Os dois comandos têm a mesma forma, mas no primeiro o (i) é o índice de um array, então o compilador simplesmente atribui o i-ésimo valor de primeiro em a. Já no segundo comando, como não há o atributo de dimensão na variável segundo, o compilador assumirá que se trata de uma função real externa.

2.12.1 Inicializando um array

Arrays podem ser inicializados de várias formas. Atribuição direta:

```
arr_inteiro = (/1,2,3,4,5,6,7,8,9,10/)
```

Para arrays grandes, usa-se o DO implícito:

```
arr_inteiro = (/ (i,i=1,10) /)
```

Essa é uma construção poderosa, que pode tomar várias formas:

```
arr_real = (/ (SQRT(REAL(i)), i=1,1000) /)
arr_inteiro = (/ ((0,i=1,9), 10*j, j=1,10) /)
```

Exercício: qual o valor do array acima?

2.12.2 Lendo e escrevendo um array

Quando aparecem em expressões sem referências aos índices, então implicitamente está se referindo ao array todo:

```
PRINT*, arr_inteiro
```

Subarrays também podem ser facilmente extraídos:

```
PRINT*, (p(i), i=1,99,2)
PRINT*, p(10:20)
```

Exemplo de uma construção perigosa:

```
READ*, primeiro, ultimo, (arr(i), i=primeiro,ultimo)
```

Por quê é perigosa? Se não tomar cuidado pode-se ter problemas de memória (**segmentation fault**, ou falha de segmentação). Por exemplo, se `arr` for um array de 1 a 10, e o valor de `ultimo` for 11, teremos problemas...

2.12.3 Arrays em expressões aritméticas

Definições:

1. Dois arrays são **conformes** se eles tiverem a mesma **forma** (igual número de dimensões e números de elementos em cada dimensão).
2. Um escalar é sempre conforme com um array.
3. Todas as operações intrínsecas são válidas entre dois arrays conformes.

Exemplos de expressões aritméticas envolvendo arrays:

```
REAL :: a(20),b(20),c(20),maximo, soma, media
REAL :: x(3),y(3)
REAL :: pe, co

a = b*c !Esse comando é equivalente à:
DO i=20
  a(i) = b(i)*c(i)
END DO

a = 10.*c
c = 2.
b = SIN(c)
maximo = MAX(a)           !Retorna o maior valor do array

! Calcula a soma de todos os elementos do array
soma = SUM(a)

! Calcula a média aritmética dos elementos do array
media = SUM(a)/SIZE(a)

! Calcula o produto escalar de dois vetores
pe = SUM(x*y)

! Calcular o comprimento de um vetor:
co = SQRT(SUM(x*y))
```

2.12.4 Arrays em procedimentos

O Fortran permite que arrays em procedimentos tenham um número de elementos indefinido. Esse número é definido ao longo da execução. Este recurso é extremamente útil, e deve ser adotado sempre, como uma boa prática de programação.

Exemplo:

```

SUBROUTINE exemplo(array1,array2)
IMPLICIT NONE

REAL, DIMENSION(:) :: array1, array2
INTEGER :: N
...
!Tamanho do array
N = SIZE(array1)
...

```

Se essa subrotina for chamada de um programa que tem as declarações:

```

REAL, DIMENSION(20) :: a,b
...
CALL exemplo(a,b)

```

então internamente as variáveis array1 e array2 terão dimensão 20.

2.13 Formatos: controlando o input e output

Vimos antes o list-directed input e output. Apesar de práticos, os comandos READ* e PRINT* não são suficientes na maioria das situações, pois frequentemente é necessário mais controle na forma da entrada e saída.

O fortran permite tal controle através de **strings de formatação**. Dessa forma, o READ* e o PRINT* são substituídos por comandos mais gerais, como abaixo:

```

READ(*, formato) arg1, arg2, ...

WRITE(*, formato) arg1, arg2, ...

```

sendo que *formato* se refere a uma string (ou uma variável string) que contém o código de formatação.

Para formatar **números reais**, usa-se os descritores F e E:

- NFw.d: escreve um número real nas próximas w letras com d casas decimais. Esse formato será repetido N vezes.
- NEw.d: escreve um número real em formato exponencial nas próximas w letras com d casas decimais. Lembrar que 4 caracteres são usados pelo expoente. Esse formato será repetido N vezes.

Para formatar caracteres usa-se o descritor A: NAw. Exemplo abaixo ilustra como se deve usar estes formadores (ver arquivo `formatos.f90`)

```

WRITE(6,"(5A16)") "Argumento", "Raiz Quadrada", "Raiz &
&Cubica", "Raiz Quadrupla", "Raiz Quintupla"

WRITE(6,"(5F16.4)") arg,raiz2, raiz3, raiz4, raiz5

WRITE(6,"(5E16.4)") arg,raiz2, raiz3, raiz4, raiz5

```

A saída do código acima é:

```
>> Argumento      Raiz Quadrada      Raiz Cubica      Raiz Quadrupla
>> 10.0000          3.1623           2.1544           1.7783
>>0.1000E+02      0.3162E+01      0.2154E+01      0.1778E+01
>>Raiz Quintupla
>> 1.5849
>> 0.1585E+01
```

Para formatar números inteiros, usa-se o descritor I:

- NIw: escreve um número inteiro nas próximas w letras. Esse formato será repetido N vezes.

Exemplo:

```
WRITE(*, "(A, I1, A, F6.4) ") "x**1/", 2, " = ", raiz2
```

Saída:

```
>>x**1/2 = 4.4721
```

2.14 Usando arquivos

Para poder ler e escrever dados de uma outra unidade lógica além do teclado (tipicamente um arquivo, mas pode ser outro dispositivo), deve-se inicialmente conectar-se a essa unidade. Isso é feito pelo comando OPEN. Ex:

```
OPEN(FILE="tabela.txt", UNIT=7, STATUS="NEW", ACTION="WRITE")
```

A UNIT representa uma unidade lógica, associada a um número inteiro. Usando-se diferentes unidades, pode-se acessar vários arquivos simultaneamente.

O STATUS é útil para se definir certas restrições ao uso de um dado arquivo. Por exemplo, podemos querer garantir que não escreveremos nada por cima de um arquivo. Há três opções principais

- OLD: nesse caso, o arquivo *já deve existir previamente* para poder ser aberto.
- NEW: nesse caso, o arquivo *não deve existir previamente* para poder ser criado.
- UNKNOWN: nesse caso, se o arquivo já existe, ele será tratado como OLD, caso contrário como NEW.

Podemos também especificar que tipo de ações de input/output são permitidas usando o ACTION, que pode tomar três valores:

- READ: nesse caso o arquivo é tratado como sendo *apenas para leitura*. Não é possível escrever nele.
- WRITE: nesse caso o arquivo é *apenas para escrita*. Comandos READ não são permitidos

- READWRITE: tanto input quanto output são permitidos.

Arquivos são acessados usando-se os comandos READ e WRITE. Por exemplo, se houver um arquivo associado à unidade 7, podemos escrever neste arquivo usando o seguinte comando:

```
WRITE (UNIT=7, FMT=<formato>) ...
```

ou simplesmente

```
WRITE (7, <formato>) ...
```

Quando o acesso ao arquivo não é mais necessário, deve-se fechar a conexão:

```
CLOSE (UNIT=7)
```

O exemplo arquivos.f90 usa o código do programa formatos.f90 e escreve o resultado num arquivo cujo nome é fornecido pelo usuário.

2.15 Exercícios

- Fazer um algoritmo 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. Considere, por facilidade, valores de 0 a 100 com um máximo de três casas decimais.
- Desenvolver um programa que informe os números primos entre 1 e 1000.