Introdução à Programação com Matlab/Octave

Saulo P. Oliveira
Departamento de Matemática
Universidade Federal do Paraná

Primeiro Semestre de 2019

Sumário

1 Prefácio

Os aplicativos Matlab (www.mathworks.com) e Octave (www.octave.org) são ambientes integrados de computação científica e visualização (Quarteroni e Saleri, 2007). O Matlab tornou-se padrão na área de matemática aplicada e é amplamente empregado em diversos ramos das ciências e engenharias, enquanto o Octave, na opinião do autor, é o aplicativo gratuito mais compatível com o Matlab.

Enquanto a maioria dos livros e apostilas sobre o Matlab/Octave tratam da sintaxe ou do uso destes aplicativos em cálculo numérico, estas notas de aula tratam de lógica de programação, evidenciando, como Higham e Higham (2002), as vantagens que o Matlab/Octave oferecem para a implementação de algoritmos de computação científica. Estas notas são dirigidas sobretudo a estudantes que tiveram pouco ou nenhum contato com lógica de programação.

2 Utilizando a janela de comando

Ao iniciarmos o Matlab ou o Octave, devem-se abrir telas semelhantes às da Figura 1:

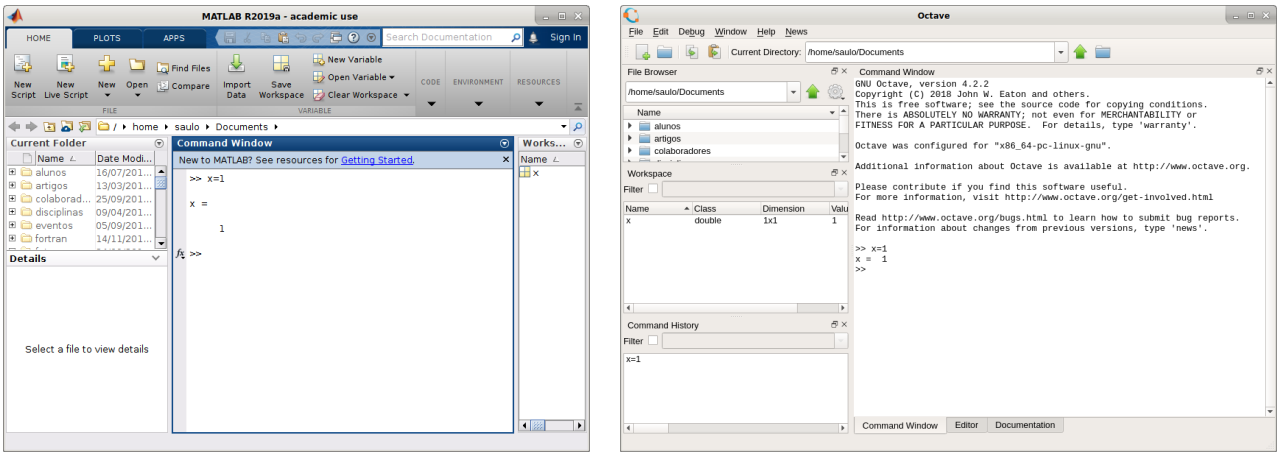

Figura 1: Tela inicial dos aplicativos Matlab (à esquerda) e Octave (à direita).

Entre as vários módulos da tela, vamos nos concentrar na janela de comando (`Comand Window`).Há a alternativa de executar o Matlab e o Octave em linha de comando, que permite abrir apenas a janela de comando. No Matlab, usamos o comando `matlab -nodesktop`, que resulta em uma janela com um conteúdo semelhante ao seguinte:

```
                    < M A T L A B (R) >
          Copyright 1984-2019 The MathWorks, Inc.
          R2019a (9.6.0.1072779) 64-bit (glnxa64)
                          March 8, 2019


To get started, type doc.
For product information, visit www.mathworks.com.

>>
```

Para iniciar o Octave no modo de linha de comando, usamos o comando `octave --no-gui`, que resulta numa tela semelhante a

```
GNU Octave, version 4.2.2
Copyright (C) 2018 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  For details, type 'warranty'.

Octave was configured for "x86_64-pc-linux-gnu".

Additional information about Octave is available at http://www.octave.org.

Please contribute if you find this software useful.
For more information, visit http://www.octave.org/get-involved.html
```

```
Read http://www.octave.org/bugs.html to learn how to submit bug reports.
For information about changes from previous versions, type 'news'.
octave:1>
```

Após clicar no interior da janela de comandos, podemos fazer operações numéricas ou gráficas, executar comandos ou rodar programas. Estas ações devem ser digitadas logo após o >> (ou `octave:1>` no Octave). Esta forma de interação com o usuário é denominada *linha de comando*. Por exemplo, podemos realizar a operação 1+1 na linha de comando, digitando `1+1` e teclando `Enter`:

```
>> 1+1
ans =  2
>>
```

OBS: Por conveniência, daqui em diante o termo "matlab" servirá para se referir tanto ao Matlab quanto ao Octave. Quando quisermos nos referir aos aplicativos, usaremos iniciais maiúsculas. Ou seja, "matlab" significa "Matlab ou Octave", enquanto "Matlab" significa somente "Matlab".

2.1 Operações elementares

Nosso primeiro uso do matlab será transformar o computador em uma "calculadora de luxo". As operações aritiméticas elementares seguem a sintaxe usual:

```
>> 56-3
ans =  53
>> 4*8/3
ans =  10.667
>>
```

Assim como aconteceria em uma calculadora, dízimas periódicas são truncadas[1]. Como os teclados não dispõem da tecla x^y, a exponenciação precisa ser representada por algum caracter. No matlab, o caracter escolhido foi o acento circumflexo:

```
>> 10^3
ans =  1000
>>
```

2.2 Constantes

Outra tecla indisponível é a tecla π. É verdade que as calculadoras em geral também não a possuem, sendo o número π obtido da combinação de mais teclas, e que o mesmo poderia ser feito no teclado de um computador. Porém, ao invés de se recorrer a combinações complicadas de teclas ou mudar a configuração do teclado (exceto quem possuir um notebook grego), o matlab optou por representar o número π simplesmente por `pi`:

```
>> pi
ans =  3.1416
>>
```

[1] Veremos depois como exibir um número maior de casas decimais.

O mesmo acontece com outras constantes clássicas da matemática:

```
>> e
ans =  2.7183
>> i^2
ans = -1
>>
```

OBS: a constante $e = 2.7183...$ é definida pela letra `e` somente no Octave. Entretanto, o número e pode ser obtido em ambos os aplicativos por meio do comando `exp(1)`. Este é um exemplo de uso de uma *função intrínseca*, apresentado a seguir.

2.3 Funções intrínsecas

Assim como as calculadoras científicas, o matlab dispõe das funções trigonométricas e logatítmicas:

```
>> cos(pi)
ans =  -1
>> tan(pi)
ans = -1.2246e-16
>> log(1)
ans = 0
>>
```

OBS: Encontramos $\tan(\pi) \approx 1.2 \times 10^{-16}$ em vez de $\tan(\pi) = 0$. Em muitos cálculos o matlab comete erros em torno de 10^{-16}. Estes erros são relacionados ao *epsilon da máquina*, que discutiremos no final desta seção.

O matlab dispõe de um grande número de funções matemáticas pré-definidas, que denominamos *funções intrínsecas* (*built-in functions*). O leitor pode consultar a disponibilidade de funções em livros e tutoriais, ou por meio dos comandos `help` ou `lookfor`:

```
>> help sqrt
'sqrt' is a built-in function

 -- Mapping Function:  sqrt (X)
     Compute the square root of each element of X.  If X is negative, a
     complex result is returned.  To compute the matrix square root, see
     *note Linear Algebra::.

     See also: realsqrt

Additional help for built-in functions and operators is
available in the on-line version of the manual.  Use the command
'doc <topic>' to search the manual index.

Help and information about Octave is also available on the WWW
at http://www.octave.org and via the help@octave.org
mailing list.
>> sqrt(16)
ans =  4
```

```
>> lookfor cosecant
acsc                Compute the inverse cosecant in radians for each element of X.
acscd               Compute the inverse cosecant in degrees for each element of X.
acsch               Compute the inverse hyperbolic cosecant of each element of X.
csc                 Compute the cosecant for each element of X in radians.
cscd                Compute the cosecant for each element of X in degrees.
csch                Compute the hyperbolic cosecant of each element of X.
>>
```

Enquanto o comando `help` descreve uma função específica, o comando `lookfor` faz uma busca por palavra-chave (em inglês) entre os comandos disponíveis. Cabe observar que o resultado do comando `help` no Matlab é um pouco diferente do octave:

```
>> help sqrt
 SQRT   Square root.
    SQRT(X) is the square root of the elements of X. Complex
    results are produced if X is not positive.

    See also SQRTM, REALSQRT, HYPOT.

    Reference page in Help browser
       doc sqrt
>>
```

Voltaremos às funções intrínsecas na Seção 6.

2.4 Variáveis

Em todas as operações acima apareceu `ans = ...`. Vamos experimentar digitar `ans` depois de alguma operação:

```
>> 10^3
ans =  1000
>> ans
ans =  1000
>>
```

Temos que `ans` é o resultado, ou resposta (answer) da última operação realizada. Podemos usar o `ans` como se fosse um operando:

```
>> ans*7.5
ans =  7500
>> ans
ans =  7500
>>
```

Note que, sendo o resultado da última operação realizada, `ans` passou de 1000 para 7500. Assim como nas calculadoras, podemos usar este recurso para efetuar operações compostas, como $1 + 2 + .. + 5$:

```
>> 1+2
ans =  3
```

```
>> ans + 3
ans =  6
>> ans + 4
ans =  10
>> ans + 5
ans =  15
>>
```

Note que as setas do teclado mostram os comandos anteriormente executados, ajudando na digitação dos cálculos acima. Agora vamos usar um recurso do matlab que não costuma estar disponível nas calculadoras mais simples. Em vez de `ans`, vamos usar uma palavra de nossa preferência, por exemplo, `soma`:

```
>> soma = 1+2
soma =  3
>> soma = soma + 3
soma =  6
>> soma = soma  + 4
soma =  10
>> soma = soma +  5
soma =  15
>>
```

O preço desta comodidade foi ter que digitar `soma =` antes de cada comando para avisar ao matlab que o resultado deve ir para `soma`, e não para `ans`. Certamente seria mais fácil executar

```
>> soma = 1 + 2 + 3 + 4 + 5
soma =  15
>>
```

Posteriormente, teremos exemplos mais complexos em que valerá a pena decompor a operação em vários passos. O importante agora é notar que o resultado de uma operação pode ser guardado no matlab com um nome de nossa escolha, e que ele pode ser empregado como se fosse um operando e seu conteúdo pode ser alterado se for preciso:

```
>> soma_maior = 10*soma
soma_maior =  150
>> soma = soma + soma_maior^2
soma =  22515
>>
```

As quantidades `ans`, `soma` e `soma_maior` são denominadas *variáveis*. Vamos utilizar a seguinte definição de variável:

Definição: uma *variável* é um segmento de memória que é selecionado e identificado durante a execução de um programa para armazenar um conjunto de dados. Os dados armazenados neste segmento de memória podem ser alterados ou liberados para uso de outras variáveis.

O matlab automaticamente faz dois passos importantes da criação de uma variável: a seleção da *estrutura de dados* adequada para delimitar o conjunto de dados (se é um número, uma matriz ou um texto, por exemplo) e a seleção do segmento de memória que a variável ocupará (o que é conhecido como *alocação de memória*). Em particular, **não é necessário declarar ou alocar variáveis no matlab**, embora a alocação seja recomendável em problemas de grande porte, conforme veremos posteriormente.

2.4.1 Regras para identificar variáveis

Quatro regras devem ser levadas em conta no momento de escolher um nome para identificar uma variável:

1. O primeiro caracter deve ser uma letra;
2. Os demais caracteres podem ser letras, números ou o caracter _ ;
3. Maiúsculas e minúsculas são considerados caracteres distintos;
4. O nome não pode coincidir com nenhuma palavra reservada do matlab;

Por exemplo, nomes como `soma1`, `soma_1` e `Soma_1` são válidos, enquanto `soma 1`, `1soma`, e `soma1!` não são válidos.

As palavra reservadas que mais usaremos são `if`, `else`, `for`, `while`, `end`, `switch` e `function`, e a lista completa pode ser obtida com o comando `iskeyword`.

2.4.2 Inicialização

Há um recurso disponível em algumas linguagens de programação que não está disponível no matlab: utilizar uma variável inexistente como se ela já existisse. Por exemplo, no comando abaixo,

```
>> soma_maior = Soma_maior + 3
error: 'Soma_maior' undefined near line 3 column 14
>>
```

a variável `Soma_maior` foi interpretada como uma variável diferente de `soma_maior`, pois **há distinção** entre maiúsculas e minúsculas (Regra 3 acima). O erro foi causado porque tentamos acessar o conteúdo da variável inexistente `Soma_maior`.

Uma variável torna-se existente a partir de seu primeiro uso:

```
>> Soma_maior = 1;
>> Soma_menor = soma_maior;
>>
```

Este processo de criar uma variável atribuindo-lhe um conteúdo (que pode inclusive ser de outra variável existente, como fizemos acima com `Soma_menor`) é denominado *inicialização*.

2.4.3 O comando `clear`

Infelizmente, é possível também inicializar as constantes internas do matlab, pois elas não constam na lista de palavras reservadas. Por exemplo:

```
>> pi = 56
pi =  56
>> pi^2
ans =  3136
>>
```

No exemplo acima, perdemos a constante $\pi \approx 3.1415\ldots$. Para recuperá-la, usamos o comando `clear` para limpar a variável `pi` da memória. Ao constatar que `pi` deixou de ser uma variável, o matlab o redefine como a constante π:

```
>> clear pi
>> pi
ans =  3.1416
>>
```

Por outro lado, quando limpamos da memória (ou no jargão de programação, *desalocamos*) uma variável não-associada a uma constante interna do matlab, esta variável simplesmente deixa de existir:

```
>> Soma_maior
Soma_maior =  1
>> clear Soma_maior
>> Soma_maior
error: 'Soma_maior' undefined near line 4 column 1
>>
```

Finalmente, se quisermos limpar **todas** as variáveis da memória, basta digitar `clear`:

```
>> clear
>> who
>>
```

O comando `who`, que não retornou nenhum resultado, indica todas as variáveis existentes no momento.

2.5 Variáveis do tipo vetor e matriz

Vetores e matrizes são as principais estruturas de dados do matlab (tanto que o nome do pacote Matlab vem de *MATrix LABoratory*). Uma das formas de definir um vetor é digitar suas coordenadas, delimitando-as com colchetes:

```
>> v = [1,2,3]
v =

   1   2   3

>> w = [4 5 6]
w =

   4   5   6
```

Note que as coordenadas podem ser separadas por vírgulas ou por espaços. Um procedimento análogo se aplica às matrizes e vetores-coluna:

```
>> A = [4,2,1
1,4,2
2,1,10]
A =

    4    2    1
    1    4    2
```

```
    2    1   10

>> u = [1
2
1]
u =

   1
   2
   1
```

Como vimos anteriormente, o símbolo `;` permite inserir dois ou mais comandos na mesma linha. Ele nos permite re-escrever `A` e `u` na forma

```
>> A = [4,2,1 ; 1,4,2 ; 2,1,10]
A =

    4    2    1
    1    4    2
    2    1   10

>> u = [1 ; 2 ; 1]
u =

   1
   2
   1
```

Conforme observado em PET-EngComp (2008), depois da criação do vetor/matriz, pode-se alterar um elemento acessando diretamente a sua posição, assim como acessar uma posição inexistente (nesse caso, as posições que não existiam até a posição acessada são automaticamente anuladas). Este acesso é realizado digitando o nome da variável seguido da localização do elemento entre parênteses:

```
>> u(3) = 5
u =

   1
   2
   5

>> u(4) = -1
u =

   1
   2
   5
  -1

>> A(2,3) = 1
```

```
A =

    4    2    1
    1    4    1
    2    1   10
```

Observe que no caso de matrizes, inserimos dois índices entre parenteses e separados por vírgulas; o primeiro corresponde à linha, enquanto o segundo corresponde à coluna. Esta forma de acesso permite tambem extrair valores de uma variável vetorial/matricial:

```
>> A(3,3)
ans =  10
>> beta = u(2)
beta =  2
```

2.6 Gráficos bidimensionais

O *gráfico* da função $y = f(x)$ no intervalo $[a, b]$ é uma representação, no plano cartesiano, do conjunto de pares ordenados $(x, f(x))$, $x \in [a, b]$. Como o intervalo $[a, b]$ é um conjunto não-enumerável, é impossível, mesmo com uma hipotética repetição infinita, marcar sequencialmente no plano cartesiano todos os pares ordenados $(x, f(x))$ do gráfico.

O matlab aproxima o gráfico de $y = f(x)$ marcando em uma janela adicional (que representa o plano cartesiano) um número finito de pares ordenados:

$$(x_i, f(x_i)), \quad 1 \leq i \leq n, \quad \text{com} \quad a = x_1 \leq x_2 \leq \ldots \leq x_n = b.$$

Vamos nos referir aos pontos $x_1, \ldots x_n$ acima como sendo uma *partição* do intervalo $[a, b]$. Quanto maior o número n, mais realista é o gráfico. O comando mais simples do matlab para gerar o gráfico de $y = f(x)$ é o comando `plot(x,y)`. Neste comando, as variáveis `x` e `y` são os vetores $[x_1, x_2, \ldots x_n]$ e $[f(x_1), f(x_2), \ldots, f(x_n)]$, respectivamente. Por *default* (por convenção), o matlab preenche o espaço entre dois pontos consecutivos com uma linha sólida. Isso pode ser visto no seguinte exemplo (ver Fig. 2):

```
>> x = [0,pi/2,pi,3*pi/2,2*pi];
>> y = [sin(0),sin(pi/2),sin(pi),sin(3*pi/2),sin(2*pi)];
>> plot(x,y)
```

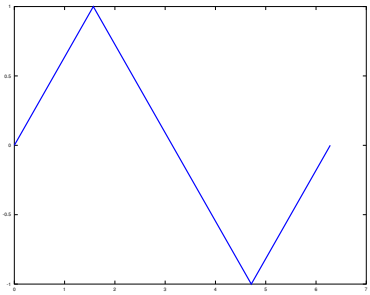

Figura 2: Resultado do comando `plot(x,y)`

O comando `plot` não requer que o vetor `y` seja definido a partir de $y = f(x)$. Por exemplo, poderíamos aproximar o gráfico de $x = \sin(y)$:

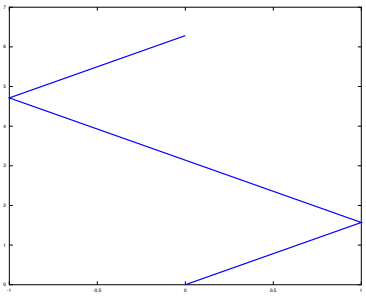

Figura 3: Resultado do comando `plot(y,x)`

```
>> plot(y,x)
```

Este gráfico é exibido na Fig. 3. Em geral, o comando plot permite o traçado do gráfico de funções paramétricas

$$\gamma : \begin{cases} x = x(t), \\ y = y(t), \quad t \in [0, T] \end{cases}$$

Por exemplo, o gráfico das funções paramétricas $x = \cos(t)$ e $y = \sin(t)$ para $t \in [0, 2\pi]$, que corresponde a uma circunferência de raio 1, pode ser aproximada da seguinte forma:

```
>> x = [cos(0),cos(pi/2),cos(pi),cos(3*pi/2),cos(2*pi)];
>> y = [sin(0),sin(pi/2),sin(pi),sin(3*pi/2),sin(2*pi)];
>> plot(x,y)
```

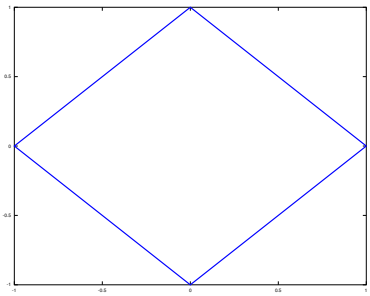

Figura 4: Resultado do comando `plot(x,y)` com funções paramétricas.

2.7 Comandos do sistema operacional

Podemos usar a linha de comando do matlab para executar alguns comandos do sistema operacional (Linux, Windows, Mac OS). Os comandos de consulta e mudança de pastas (diretórios) podem ser utilizados exatamente como seriam em uma janela de terminal:

- `ls`: exibe a lista de arquivos na pasta;
- `dir`: o mesmo que `ls`;
- `pwd`: indica a pasta atual;
- `cd`: muda para outra pasta

Outros comandos, como o de remover um arquivo, necessitam do comando auxiliar `system`. No exemplo a seguir, vamos remover um arquivo e verificar se ele foi removido da pasta atual:

```
>> ls
teste.txt
>> system('rm teste.txt')
ans = 0
>> ls
>>
```

2.8 Exercícios

1. Calcule o produto $3\sqrt{3}$.

2. Calcule (preferencialmente usando variáveis) o resultado da seguinte expressão:

$$\frac{15}{\frac{12}{18*54} + \frac{345}{\frac{1}{8}+\frac{18*54}{13}}}$$

3. De modo análogo ao realizado com a variável `soma`, utilize uma variável para calcular o fatorial de 10:

$$10! = (10)(9)(8)(7)\ldots(3)(2)(1).$$

   Em seguida, use o comando `factorial(10)` para conferir a resposta.

4. Observe os comandos executados abaixo:

```
>> 2
ans = 2
>> ans*ans
ans = 4
>> ans*ans
ans =  16
>> ans*ans
ans =  256
```

   Em geral, se o comando `ans*ans` for executado `n` vezes, qual é o número resultante ?

5. Indique o erro na seguinte sequência de comandos:

```
clear
x = 0
y = 4
z = x + y
y = x + w
```

6. Escreva o comando para gerar a matriz

$$C = \begin{bmatrix} 10 & 9.5 \\ 7 & 12 \end{bmatrix}.$$

7. Sem redefinir toda a matriz C acima, altere o termo $C_{2,2} = 12$ para $C_{2,2} = 10$

2.9 Apêndice: representação computacional dos números reais

Este apêndice, baseado nos livros de Quarteroni e Saleri (2007) e Kincaid e Cheney (2002), discute as limitações da representação computacional dos números reais, presentes no matlab e nas linguagens de programação em geral.

2.9.1 Base binária

Os computadores atuais operam com números em base binária, ou seja, expressos exclusivamente pelos dígitos binários 0 e 1. Por exemplo, o número binário $(100101.11)_2$ (escrevemos o subíndice "2"para distinguí-lo do número decimal $100101.01 = (100101.01)_{10}$) corresponde, na base binária, ao número

$$1\times 2^5+0\times 2^4+0\times 2^3+1\times 2^2+0\times 2^1+1\times 2^0+0\times 2^{-1}+1\times 2^{-2}=32+4+1+0.25=37.25.$$

Como usualmente fornecemos ao computador dados numéricos na base decimal e precisamos receber os resultados nesta mesma base, o computador precisa efetuar conversões tanto de binário para decimal (como no exemplo acima) quanto de decimal para binário.

A conversão de um número inteiro na base decimal para a base binária pode ser feita dividindo-o sucessivamente por dois e preservando o resto das divisões,

$$\begin{aligned} 18 &= \mathbf{2\times 9}+0\\ &= 2\times(\mathbf{2\times 4}+1)+0\\ &= 2\times(2\times(\mathbf{2\times 2}+0)+1)+0\\ &= 2\times(2\times(2\times(\mathbf{2\times 1}+0)+0)+1)+0, \end{aligned}$$

e em seguida distribuindo todas as operações realizadas:

$$\begin{aligned} 18 &= 2\times(2\times(2\times(2\times 1+0)+0)+1)+0\\ &= 2\times(2\times(2\times(2\times 1+2^0\times 0)+2^0\times 0)+2^0\times 1)+2^0\times 0\\ &= 2\times(2\times(2^2\times 1+2^1\times 0)+2^0\times 0)+2^0\times 1)+2^0\times 0\\ &= 2\times(2^3\times 1+2^2\times 0+2^1\times 0)+2^0\times 1)+2^0\times 0\\ &= 2^4\times 1+2^3\times 0+2^2\times 0+2^1\times 1+2^0\times 0=(10010)_2. \end{aligned}$$

A conversão de números decimais entre 0 e 1 utiliza um processo reverso:

$$\begin{aligned} 0.125 &= \mathbf{0.25/2}+0\\ &= (\mathbf{0.5/2}+0)/2+0\\ &= ((\mathbf{1/2}+0)/2+0)/2+0\\ &= (((2^0\times 1)/2+2^0\times 0)/2+2^0\times 0)/2+2^0\times 0\\ &= ((2^{-1}\times 1+2^0\times 0)/2+2^0\times 0)/2+2^0\times 0\\ &= (2^{-2}\times 1+2^{-1}\times 0+2^0\times 0)/2+2^0\times 0\\ &= 2^{-3}\times 1+2^{-2}\times 0+2^{-1}\times 0+2^0\times 0\\ &= 2^0\times 0+2^{-1}\times 0+2^{-2}\times 0+2^{-3}\times 1=(0.001)_2. \end{aligned}$$

Este processo, quando aplicado ao número $0.1 = (0.1)_{10}$, revela uma dificuldade na conversão

de números fracionários:

$$\begin{aligned}
0.1 &= \mathbf{0.2/2} + 0 \\
&= (\mathbf{0.4/2} + 0)/2 + 0 \\
&= ((\mathbf{0.8/2} + 0)/2 + 0)/2 + 0 \\
&= (((\mathbf{1.6/2} + 0)/2 + 0)/2 + 0)/2 + 0 \\
&= ((((\mathbf{0.6 + 1})/2 + 0)/2 + 0)/2 + 0)/2 + 0 = ((((\mathbf{1.2/2} + 1)/2 + 0)/2 + 0)/2 + 0)/2 + 0 \\
&= (((((\mathbf{1 + 0.2)/2} + 1)/2 + 0)/2 + 0)/2 + 0)/2 + 0,
\end{aligned}$$

ou seja,

$$\begin{aligned}
0.1 &= \mathbf{2^{-1} \times 0.2} = 2^{-4} \times ((1 + 0.2)/2 + 1) = 2^{-4} \times (2^{-1} + \mathbf{2^{-1} \times 0.2} + 1) \\
&= 2^{-4} + 2^{-5} + 2^{-4} \times (\mathbf{2^{-1} \times 0.2}).
\end{aligned}$$

A dificuldade reside no fato que, na expressão acima, o termo $2^{-1} \times 0.2$ é recorrente. Substituindo-o em si mesmo, obtemos

$$\begin{aligned}
0.1 &= 2^{-4} + 2^{-5} + 2^{-4} \times (\mathbf{2^{-4} + 2^{-5} + 2^{-4} \times (2^{-1} \times 0.2)}) \\
&= 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-8}(\mathbf{2^{-1} \times 0.2}),
\end{aligned}$$

e a recorrência se repete. Substituindo $2^{-1} \times 0.2 = 2^{-4} + 2^{-5} + 2^{-4} \times (\mathbf{2^{-1} \times 0.2})$ novamente,

$$\begin{aligned}
0.1 &= 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-8}(\mathbf{2^{-4} + 2^{-5} + 2^{-4} \times (2^{-1} \times 0.2)}) \\
&= 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + 2^{-12} \times (\mathbf{2^{-1} \times 0.2})
\end{aligned}$$

e este processo se repete infinitamente, de modo que

$$(0.1)_{10} = (0.000110001100011..)_2, \tag{1}$$

ou seja, o número decimal 0.1 é uma dízima periódica na base binária.

2.9.2 Bits e bytes

A unidade básica de armazenamento é o bit (*binary digit*), um segmento de memória capaz de armazenar o dígito binário **zero** ou o dígito binário **um** (que também pode ser interpretado como as condições **falso** e **verdadeiro**).

Entretanto, é a cadeia de 8 bits, denominada *byte*, que é a principal unidade de medida de memória. Um byte é capaz de armazenar $2^8 = 256$ elementos distintos, por exemplo os números de 0 a 255 ou os índices da tabela ASCII de caracteres.

O byte ainda é insuficiente para a representação de números, mesmo que números naturais. Os dois padrões mais utilizados atualmente para a representação de números reais são os de *precisão simples* (com 32 bits, ou 4 bytes) e de *precisão dupla* (com 64 bits, ou 8 bytes).

Alguns cálculos elementares nos permitem ter uma ideia de quanto espaço de memória um vetor ou matriz pode ocupar. Um vetor do $\mathbb{R}^{10^6}$ (ou uma matriz de ordem 1000) ocupa 4×10^6 bytes, ou 4Mb, em precisão simples, e 8Mb em precisão dupla. Já uma matriz de ordem 10000 ocupa 4×10^8b = 0.4Gb em precisão simples, ou 0.8Gb em precisão dupla. Note que uma dezena (ou menos) de matrizes desta dimensão esgota a memória RAM da maioria dos computadores pessoais atualmente.

OBS: Os computadores com *arquitetura* de 32bits e 64 bits ambos admitem representações de números reais com 32 bits e 64 bits (precisão simples e dupla). A diferença entre estas arquiteturas está no número de bits que podem ser operados simultaneamente e no número máximo de endereços de memória que podem ser mapeados.

2.9.3 Representação de ponto flutuante

Por conveniência, vamos denotar por $(1\ldots1)_{2,k}$ o número na base binária com k algarismos iguais a 1. A cadeia de bits que forma um número real nos formatos dados acima é dividida do seguinte modo:

$$\boxed{\pm}\left(1.\boxed{d_1}\boxed{d_2}\cdots\boxed{d_m}\right)_2 \times 2^{\left(\boxed{d_1}\boxed{d_2}\cdots\boxed{d_e}\right)_2-\left(\boxed{1}\boxed{1}\cdots\boxed{1}\right)_{2,e-1}} \tag{2}$$

Este formato é denominado *representação de ponto flutuante*.

O número $(1.d_1\ldots d_m)_2$ é denominado *mantissa*, enquanto $(1.d_1\ldots d_e)_2-(1\ldots1)_{2,e-1}$ é denominado *expoente*. Note que o número de bits é dividido da seguinte forma:

- 1 bit para o sinal;
- m bits para a mantissa;
- e bits para o expoente.

O padrão mais difundido atualmente (IEEE 754) utiliza

- $m=23$ e $e=8$ em precisão simples;
- $m=52$ e $e=11$ em precisão dupla.

Tanto em precisão simples quanto em precisão dupla, ou mesmo dado um outro par de parâmetros $\{m,e\}$, o conjunto $\mathbb{F}$ de todos os números na forma (2) que temos à disposição é sempre um subconjunto **finito** do conjunto $\mathbb{R}$ dos reais. Este conjunto não necessariamente satisfaz algumas propriedades elementares dos números reais, como a associatividade. Por exemplo,

```
>> a = 1.0e308;
>> b = 1.1e308;
>> c = -1.001e308;
>> a+(b+c)
ans =  1.0990e+308
>> (a+b)+c
ans = Inf
>>
```

O símbolo `Inf`, que denota o infinito, também é armazenado no conjunto $\mathbb{F}$, conforme veremos depois. Além de não conter números arbitrariamente grandes, o conjunto $\mathbb{F}$ também não contem números arbitrariamente pequenos:

```
>> 10^(-10)
ans =  1.0000e-10
>> 10^(-1000)
ans = 0
>>
```

Outra deficiência mais sutil do conjunto $\mathbb{F}$ é com números nem muito grandes nem muito pequenos, mas que necessitam de muitos (ou mesmo infinitos) dígitos para ser representado:

```
>> 1/3
ans =  0.33333
>>
```

Convém observar que o resultado da operação `1/3` tem uma precisão maior do que a exibida acima. Para exibir um resultado com mais casas decimais, usamos o comando `format`:

```
>> format long
>> 1/3
ans =  0.333333333333333
>> format short
>> 1/3
ans =  0.33333
>>
```

Note que `format short` é a opção default do matlab. Use o comando `help format` para ver mais opções de formato. Por outro lado, o comando `sprintf` permite um maior controle sobre o número de casas decimais. Por exemplo, se quisermos exibir `1/3` com 30 caracteres (incluindo o ponto e o sinal), dos quais 20 são casas decimais da parte fracionária, usamos

```
>> sprintf('%30.20f',1/3)
ans =         0.33333333333333331483
>>
```

Note no resultado abaixo que alguns dígitos da representação do número `1/3` no matlab está incorreta. Mesmo números que possuem uma representação decimal finita não escapam de uma representação errônea:

```
>> sprintf('%30.20f',0.125)
ans =         0.12500000000000000000
>> sprintf('%30.20f',0.1)
ans =         0.10000000000000000555
>>
```

O número 0.125 foi representado corretamente, enquanto o número 0.1 não. A falha na representação do número 0.1 se deve ao fato deste número ser uma dízima periódica na base binária, conforme (1).

2.9.4 Maior número e menor número

Afinal, dados $\{m, e\}$, a que limites estão sujeitos a representação de ponto flutuante? Antes de seguir em frente, convém revelar que o matlab utiliza precisão dupla (ou seja, $m = 52$ e $e = 11$), mas que é possível utilizar outros formatos (consulte-os com o comando `help cast`).

Note que o número $(11\ldots1)_{2,k}$ corresponde, na base decimal, à progressão geométrica

$$1 + 2 + 2^2 + \ldots + 2^{k-1} = \frac{2^k - 1}{2 - 1} = 2^k - 1.$$

A princípio, obtemos o maior número do conjunto $\mathbb{F}$ inserindo `1` em todos os bits de (2):

$$\begin{aligned} x_{max} &= (1.1\ldots1)_2 \times 2^{(1\ldots1)_{2,e} - (1\ldots1)_{2,e-1}} \\ &= ((1\ldots1)_{2,m+1} \times 2^{-m}) \times 2^{2^e - 1 - (2^{e-1} - 1)} \\ &= (2^{m+1} - 1) \times 2^{-m} \times 2^{2^e - 2^{e-1}} \\ &= (2 - 2^{-m}) \times 2^{2^{e-1}}. \end{aligned}$$

Entretanto, o expoente 2^{e-1} é reservado para o infinito ($\pm$`Inf`) e indeterminações como 0/0 (`NaN`). Por isso, o maior número corresponde ao número acima com uma unidade a menos no expoente:

$$x_{max} = (2 - 2^{-m}) \times 2^{2^{e-1}-1}.$$

Com $m = 52$ e $e = 11$, temos $x_{max} \approx 1.8 \times 10^{308}$. Analogamente, o menor número é obtido com todos os bits iguais a zero, e excluindo-se o menor expoente[2]:

$$\begin{aligned} x_{min} &= (1.0\ldots0)_2 \times 2^{(0\ldots0)_{2,e}-(1\ldots1)_{2,e-1}+\mathbf{(1)_{10}}} \\ &= 1.0 \times 2^{-(2^{e-1}-1+1)-\mathbf{1}} \\ &= 2^{2^{e-1}+2} \qquad (\approx 2.2 \times 10^{-308} \text{ se } m = 52,\ e = 11). \end{aligned}$$

2.9.5 Epsilon da máquina

A diferença entre dois números consecutivos do conjunto $\mathbb{F}$ pode ser muito maior do que o menor número x_{min}. O exemplo extremo é a diferença entre o maior e o segundo maior número:

$$\Delta x_{max} = (2 - 2^{-m}) \times 2^{2^{e-1}-1} - (2 - 2^{-(m-1)}) \times 2^{2^{e-1}-1} = (2^{-m+1} - 2^{-m}) \times 2^{2^{e-1}-1} = 2^{2^{e-1}-1-m},$$

que em precisão dupla corresponde a $\Delta x_{max} \approx 1.996 \times 10^{292}$. Em outras palavras, há um enorme intervalo entre o maior e o segundo maior número que não é representado em $\mathbb{F}$.

Um número real x que não pertence a $\mathbb{F}$ é arredondado para o elemento mais próximo deste conjunto (detalhes em Goldberg (1991)). Vamos denotar este elemento por $fl(x)$.

Dado $x \in \mathbb{R}$, sejam $fl(x) = (1.d_1 \ldots d_m)_2 \times 2^n$ e $fl(x)_\pm$ os elementos de $\mathbb{F}$ adjacentes a $fl(x)$, ou seja,

$$fl(x)_\pm = (1.d_1 \ldots d_m)_2 \times 2^n \pm (0.0\ldots01)_2 \times 2^n = fl(x) \pm 2^{-m}2^n.$$

Temos que

$$\begin{aligned} fl(x)_- < \quad & x & < fl(x)_+ \\ fl(x)_- - fl(x) < \quad & x - fl(x) & < fl(x)_+ - fl(x) \\ -2^{-m}2^n < \quad & x - fl(x) & < 2^{-m}2^n \\ & |x - fl(x)| & < 2^{n-m}. \end{aligned}$$

Esta é uma estimativa do erro absoluto, que depende de n (e portanto de x). Entretanto, o **erro relativo** pode ser estimado de forma independente de x. Note que:

(i) se $|x| \geq 1.0 \times 2^n$, então

$$\frac{|x - fl(x)|}{|x|} \leq \frac{|x - fl(x)|}{2^n} < \frac{2^{n-m}}{2^n} = 2^{-m}.$$

(ii) se $|x| < 1.0 \times 2^n$, temos dois casos: $0 \leq x < 1.0 \times 2^n$ ou $-1.0 \times 2^n < x \leq 0$. Sem perda de generalidade, vamos analisar apenas o primeiro caso. Lembrando que $fl(x)$ é o elemento de $\mathbb{F}$ mais próximo de x, devemos ter

$$(1.1\ldots1)_2 \times 2^{n-1} < x < (1.0\ldots0)_2 \times 2^n. \tag{3}$$

A desigualdade à direita é consequência da suposição $|x| < 1.0 \times 2^n$. Se a desigualdade à direita fosse falsa, teríamos

$$x < (1.1\ldots1)_2 \times 2^{n-1} < fl(x),$$

[2] reservado ao número zero e aos números desnormalizados (Goldberg, 1991).

ou seja, $(1.1\ldots1)_2\times2^{n-1}$ seria um elemento de $\mathbb{F}$ mais próximo de x, o que contradiz a definição de $fl(x)$. Segue de (3) que $fl(x) = (1.0\ldots0)_2\times2^n$, pois este é o número com expoente 2^n mais próximo de x. Portanto,

$$\begin{array}{rcl} (1.1\ldots1)_2\times2^{n-1} < & x & < (1.0\ldots0)_2\times2^n \\ (1.1\ldots1)_2\times2^{n-1} - (1.0\ldots0)_2\times2^n < & x - (1.0\ldots0)_2\times2^n & < 0 \\ (1.1\ldots1)_2\times2^{n-1} - (1.0\ldots0)_2\times2^n < & x - fl(x) & < 0. \end{array}$$

Segue que $|x - fl(x)| < 2^n - (1.1\ldots1)_2\times2^{n-1}$, e assim,

$$\frac{|x-fl(x)|}{|x|} < \frac{|x-fl(x)|}{(1.1\ldots1)_2\times2^{n-1}} < \frac{2^n}{(1.1\ldots1)_2\times2^{n-1}} - 1 = \frac{2^n}{(2^{m+1}-1)\times2^{-m}\times2^{n-1}} - 1,$$

ou seja,

$$\frac{|x-fl(x)|}{|x|} < \frac{1}{(2^{m+1}-1)\times2^{-(m+1)}} - 1 = \frac{1}{1-2^{-(m+1)}} - 1 = \frac{2^{-(m+1)}}{1-2^{-(m+1)}}.$$

Como $1-2^{-(m+1)} > 1$, temos que $|x-fl(x)|/|x| < 2^{-(m+1)} < 2^{-m}$. Portanto, tanto em (i) quanto em (ii), encontramos

$$\frac{|x-fl(x)|}{|x|} < \epsilon = 2^{-m}.$$

O número ϵ, que em precisão dupla é dado por $\epsilon = 2^{-52} \approx 2.2\times10^{-16}$, é denominado *epsilon da máquina*. Este número é o limitante superior do erro relativo da representação de **qualquer** número real.

Por exemplo, note que o erro relativo da representação do número `0.1`, exibida anteriormente, é

$$\frac{0.1000000000000000555\ldots - 0.1}{0.1} < \frac{5.6\times10^{-18}}{0.1} = 5.6\times10^{-17} < \epsilon.$$

Em geral, o erro relativo esperado para qualquer operação numérica é maior ou igual ao epsilon da máquina (já que, além do erro de aproximar x por $fl(x)$, temos os erros das operações). Neste sentido, algoritmos que fornecem resultados com erro de módulo próximo a ϵ, como neste exemplo visto no início do capítulo,

```
>> tan(pi)
ans = -1.2246e-16
```

são em geral (mas nem sempre) aceitáveis.

3 Scripts

O desenvolvimento dos computadores foi motivado, entre outros fatores, pela necessidade de se automatizar tarefas e/ou cálculos repetitivos.

Vamos ilustrar esta necessidade primeiro de forma rudimentar, voltando ao exemplo da soma $1+\ldots+5$. Vamos fazer um procedimento à primeira vista mais complicado, criando mais uma variável, `i`, que vai armazenar a segunda parcela das somas:

```
>> i = 0; soma = 0
soma =  0
>> i = i + 1; soma = soma + i
soma =  1
>> i = i + 1; soma = soma + i
soma =  3
>> i = i + 1; soma = soma + i
soma =  6
>> i = i + 1; soma = soma + i
soma =  10
>> i = i + 1; soma = soma + i
soma =  15
```

Convém notar o uso do símbolo `;` , que permite inserir dois ou mais comandos na mesma linha. Note também que os resultados das operações que alteravam a variável `i` não foram exibidos. De fato, o símbolo `;` omite a exibição do resultado do comando.

Para ilustrar esta segunda utilidade do símbolo `;` , vamos continuar o cálculo acima, encontrando $1+\ldots+8 = 8(8+1)/2 = 36$ e exibindo o resultado somente no final:

```
>> i = i + 1; soma = soma + i;
>> i = i + 1; soma = soma + i;
>> i = i + 1; soma = soma + i;
>> soma
soma =  36
>> i
i = 8
```

Estes comandos podem ser repetidos rapidamente na linha de comando com o auxílio das setas do teclado, mas até este esforço pode ser poupado se criarmos um arquivo-texto com os comandos acima e usarmos os recursos do editor para repetir as linhas desejadas. Este arquivo pode ser lido a partir da linha de comando, e as operações ali gravadas são executadas.

Crie uma nova pasta, chamada `matlab` (ou outro nome de sua preferência) e, use o editor do matlab ou algum editor de texto simples[3] para criar um arquivo-texto com o seguinte conteúdo (não digite os marcadores >>):

```
i = 0; soma = 0;
i = i + 1; soma = soma + i;
i = i + 1; soma = soma + i;
...
i = i + 1; soma = soma + i;
i
soma
```

[3]Sugestões: WordPad no Windows, gedit no Linux e TextEdit no Mac OS

No lugar do `...` acima, repita a linha `i = i + 1; soma = soma + i;` quantas vezes desejar. Salve este arquivo com o nome `script1.m` na pasta `matlab`. A extensão `.m` não pode ser esquecida: e por meio desta extensão que o matlab reconhece o arquivo. Voltando à linha de comando, acesse a pasta `matlab` e execute o arquivo digitando somente o nome dele **sem a extensão** `.m`:

```
>> cd matlab
>> script1
i =  22
soma = 253
```

No exemplo acima, a linha `i = i + 1; soma = soma + i;` foi repetida 22 vezes, daí o resultado soma = 22(22+1)/2=253. O matlab seguiu o roteiro ("script") do arquivo `script1.m`, executando sequencialmente os comandos digitados neste arquivo. Por este motivo, arquivos como o `script1.m` são denominados *scripts*.

3.1 Estrutura de um script

A estrutura típica de um script é a seguinte:

- Uma ou mais linhas de documentação (opcional)
- Os comandos `clear` e/ou `close all` (opcional)
- entrada de dados
- processamento
- saída de dados

As linhas de documentação (ou comentários) são linhas precedidas pelo símbolo %. Todo o conteúdo das linhas que vem depois do símbolo % não é executado pelo matlab, podendo assim servir para descrever o script. Voltaremos a tocar neste assunto na seção 3.3.

As variáveis de um script compartilham o mesmo espaço da memória de variáveis criadas na janela de comandos. Uma forma de evitar que variáveis já existentes interfiram no funcionamento do script é adicionar, em seu início, o comando `clear`. Este comando, descrito na seção 2.4.3, apaga todas as variáveis da memória. Por outro lado, o comando `clear` não deve ser utilizado se o script tiver como objetivo iteragir com variáveis existentes na memória.

O comando `close all` fecha todas as figuras atualmente abertas. Este comando pode ser útil quando o script prevê a geração de mutas figuras.

O matlab oferece diversos comandos de *I/O* (entrada/saída). Vamos inicialmente utilizar dois comandos básicos que operam a entrada e saída de dados na janela de comandos. O primeiro deles é o comando `input`, que permite preencher uma variável com o conteúdo digitado pelo usuário. Por exemplo, o comando

```
>> a = input('Entre com um numero: ');
```

pede que o usuário digite um número, que será armazenado numa variável com nome `a`. O segundo comando é o `disp`, que exibe uma mensagem na tela, uma variável ou o resultado de uma operação. Por exemplo, o seguinte script pede que o usuário insira dois números, que em seguida são somados e mostrados na tela:

```
a = input('Entre com um numero: ');
b = input('Entre com outro numero: ');
disp('Soma dos dois numeros:');
disp(a+b);
```

OBS: 1) a linha `disp(a+b)` pode ser substuída pela linha `a+b`, sem o símbolo `;` no final.
2) recomenda-se evitar acentos, principalmente em nomes de variáveis e de arquivos.

A etapa de processamento será explorada a partir da Seção 3.2, onde veremos, por exemplo, como executar as operações repetidas sem a necessidade de usar o recurso de "copiar e colar"do editor de texto.

3.2 Fluxo de controle

Fluxo de controle é a ordem na qual um conjunto de instruções deve ser executado em um programa. Uma forma tradicional de representar o fluxo de controle é o *fluxograma*, conforme o exemplo a seguir:

Exemplo: um aluno tirou notas 50, 80 e 70 nas provas. Calcule a média e determine se o aluno foi aprovado sem exame final.

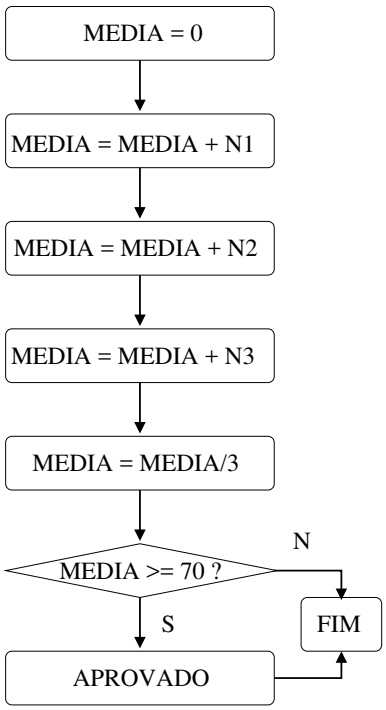

Figura 5: Fluxograma preliminar para cálculo da média das notas.

No fluxograma da figura 5, `N1,N2,N3` representam as notas das três provas. Ele pode ser reorganizado de forma mais compacta utilizando a variável vetorial `NOTA=[N1,N2,N3]` (Fig. 6).

Em geral, duas operações são essenciais para organizar o fluxo de controle:

- Repetição (diagramas 2-3 da figura 6);
- Tomada de decisão (diagrama 3 e diagrama 5 da figura 6);

O matlab possui dois comandos de repetição (`for` e `while`) e dois comandos de tomada de decisão (`if` e `switch`). Vamos começar com os comandos mais populares, `if` e `for`.

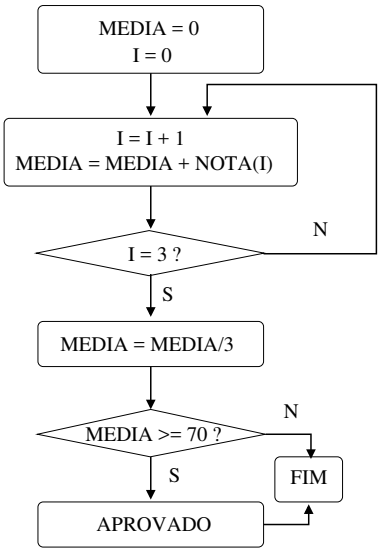

Figura 6: Fluxograma atualizado para cálculo da média das notas.

Nas primeiras linguagens de programação, o procedimento de repetição era implementado com o auxílio de um procedimento de tomada de decisão aliado a um redirecionador para ou passo anterior to programa (*go to*), conforme ilustrado na figura 6. Atualmente a maioria das linguagens emprega comandos semelhantes ao comando `for`.

3.2.1 O comando `if` (versão simplificada)

Em sua forma mais simples, o comando `if` funciona na forma

```
if (condicao)
  comando 1
  comando 2
  ...
end
```

Sendo `condicao` uma condição que queremos verificar se é verdadeira ou falsa (ou seja, uma *expressão lógica*) e `comando 1`, `comando 2`,... são os comandos que desejamos executar caso a condição seja satisfeita. Para o exemplo acima, podemos utilizar

```
if (MEDIA >= 70)
   disp('APROVADO');
end
```

O comando `disp` serve para exibir na tela a mensagem `APROVADO`.

3.2.2 O comando `for`

No exemplo acima, temos uma repetição de somas (que é mais evidente na Figura 5). Podemos escrevê-las no matlab na seguinte forma:

```
for I = 1:3
   MEDIA = MEDIA + NOTA(I);
end
```

Este trecho executa o comando `MEDIA = MEDIA + NOTA(I)` três vezes, sendo que a variável `I` é automaticamente atualizada (de 1 para 2 e depois de 2 para 3). Assim, os comandos acima equivalem às implementações de ambos os fluxogramas:

```
MEDIA = MEDIA + N1;          I = I + 1; MEDIA = MEDIA + NOTA(I);
MEDIA = MEDIA + N2;          I = I + 1; MEDIA = MEDIA + NOTA(I);
MEDIA = MEDIA + N3;          I = I + 1; MEDIA = MEDIA + NOTA(I);
```

Juntando os comandos envolvendo `if` e `for` propostos acima, chegamos ao seguinte script para o cálculo da média (salve o script abaixo com o nome `script2.m` (por exemplo) e execute-o na linha de comando):

```
MEDIA = 0;
NOTA = [50,80,70];
for I = 1:3
   MEDIA = MEDIA + NOTA(I);
end
MEDIA = MEDIA/3;
if (MEDIA >= 70)
   disp('APROVADO');
end
```

A sintaxe geral do comando `for` é a seguinte:

```
for CONTADOR = INICIO:PASSO:FIM
  comando 1
  comando 2
  ...
end
```

Nas linhas acima, devemos escolher o `CONTADOR`, uma variável inteira que, durante a execução, assumirá os valores de `INICIO` até `FIM` aumentando de `PASSO` em `PASSO` (ou diminuindo, se `PASSO`< 0). Por exemplo, podemos calcular a soma dos pares de 2 até 30 da seguinte forma:

```
>> soma = 0; for i = 2:2:40; soma = soma + i; end; soma
soma =  420
```

Este cálculo se torna incoveniente sem o uso do comando `for`:

```
>> soma=2+4+6+8+10+12+14+16+18+20+22+24+26+28+30+32+34+36+38+40
soma =  420
```

3.2.3 O comando `if` (versão geral)

Vamos modificar o script `media.m` proposto acima, adaptando-o às regras atuais da UFPR:

- Média maior ou igual a 70: aprovado;
- Média inferior a 40: reprovado;
- Média maior ou igual a 40 e menor que 70: exame final;
  Seja "Nota Final"a média entre a a média anterior e o exame final.
  - Nota Final maior ou igual a 50: aprovado;

– Nota Final menor que 50: reprovado;

Vamos começar implementando a classificação da nota final. Uma opção é utilizar dois `if`s:

```
if (NotaFinal >= 50)
   disp('APROVADO');
end
if (NotaFinal < 50)
   disp('REPROVADO - Tente no proximo semestre..');
end
```

Note que as duas condições acima são excludentes e esgotam todas as alternativas, ou seja: ambas não podem ser verdadeiras ao mesmo tempo e é impossível que todas sejam falsas. Isto permite usar uma versão mais geral do comando `if`:

```
if (NotaFinal >= 50)
   disp('APROVADO');
else
   disp('REPROVADO - Tente no proximo semestre..');
end
```

Entre os comandos `else` e `end` inserimos os comandos que queremos executar caso a condição `NotaFinal >= 50` não seja satisfeita. Para a classificação anterior ao exame final, poderíamos utilizar três `if`s, mas o fato das três condições serem excludentes e esgotarem todas as alternativas nos permite usar o recurso `elseif`:

```
if (NotaFinal >= 70)
   disp('APROVADO');
elseif ( NotaFinal < 40)
   disp('REPROVADO - Tente no proximo semestre..');
else
   disp('EXAME FINAL');
end
```

3.2.4 Identação

O último trecho acima ficou incompleto, pois faltou considerar o que acontece no caso de um exame final. Para isto, podemos copiar e colar os comandos referentes a esse caso:

```
if (NotaFinal >= 70)
   disp('APROVADO');
elseif ( NotaFinal < 40)
   disp('REPROVADO - Tente no proximo semestre..');
else
if (NotaFinal >= 50)
   disp('APROVADO');
else
   disp('REPROVADO - Tente no proximo semestre..');
end
end
```

Aqui convém alertar sobre o uso (opcional) de espaços em branco no início das linhas, como se fossem parágrafos. A falta destes espaços no trecho que foi colado dá a impressão que o `if` do exame final está fora do primeiro `if`. Para evitar esta impressão, usamos os espaços:

```
if (NotaFinal >= 70)
   disp('APROVADO');
elseif ( NotaFinal < 40)
   disp('REPROVADO - Tente no proximo semestre..');
else
   if (NotaFinal >= 50)
      disp('APROVADO');
   else
      disp('REPROVADO - Tente no proximo semestre..');
   end
end
```

O uso de espaços em branco para indicar que alguns comandos são internos a outros é conhecido como *indentação*.

3.2.5 Operadores relacionais e lógicos

Usamos até agora dois operadores que comparam dois elementos nos comandos `if`: $<$ e $\geq$. Estes operadores são denominados *operadores relacionais*. Além dos operadores relacionais, vamos precisar de *operadores lógicos*, que atuam em uma ou mais condições. Por exemplo, para verificar se a média x de um aluno levaria a exame final, ou seja $40 \leq x < 70$, precisamos do operador lógico *e* para impor simultaneamente as condições $x \geq 40$ e $x < 70$:

```
x = 40;
if (x>=40)&&(x<70)
   disp('Exame final');
end
```

A tabela 1 indica os operadores lógicos e relacionais mais frequentes. Note em particular que o operador de igualdade não é `=`.

Operador	Sintaxe
$x > y$	`x > y`
$x < y$	`x < y`
$x \geq y$	`x >= y`
$x \leq y$	`x <= y`

Operador	Sintaxe
$x = y$	`x == y`
$x \neq y$	`x ~= y`
$x \in y$	`ismember(x,y)`

Operador	Sintaxe
`COND1` *e* `COND2`	`COND1 && COND2`
`COND1` *ou* `COND2`	`COND1 || COND23`
não `COND1`	`~COND1`

Tabela 1: Operadores relacionais e lógicos

OBS: É possível substituir a condição do comando `if` por uma variável escalar. Neste caso, o número zero é interpretado como a condição `falso`, e os demais números, como a condição `verdadeiro`. No exemplo, a seguir, a função intrínseca `mod(m,n)` serve para calcular o resto da divisão entre `m` e `n`.

```
n = 5;
if mod(n,2)
   disp('Numero impar ou nao inteiro');
else
   disp('Numero par');
end
```

3.3 Comentários descritivos

Agora que sabemos automatizar repetições, voltemos ao último exemplo da seção 2.6, em que tentamos desenhar uma circunferência com pontos fornecidos manualmente. Nesta seção vamos gerar os pontos automaticamente, pedindo ao usuário que entre com o número de pontos desejado. Para isto, vamos criar o script `grafico_circ.m` com o seguinte conteúdo:

```
n  = input('Entre com o numero de pontos no grafico: ');
t = linspace(0,2*pi,n);
for i = 1:n
   x(i) = cos(t(i));
   y(i) = sin(t(i));
end
plot(x,y)
```

A função intrínseca `linspace(a,b,n)` gera um vetor de dimensão `n` com primeira coordenada `a`, última coordenada `b`, e coordenadas internas homogeneamente espaçadas. Ao executarmos este script e escolhermos `n=100`, obtemos a imagem mostrada na Fig. 7. Na seção 6 veremos como gerar rapidamente os vetores `x` e `y`.

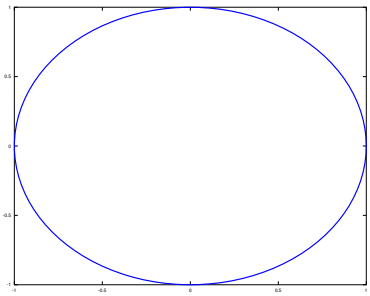

Figura 7: Resultado do script `grafico_circ.m`

Nossa próxima tarefa é *documentar* o script acima, ou seja, inserir informações no arquivo que permitam que outra pessoa compreenda mais rapidamente o programa. A primeira atitude de alguém que acabou de receber o arquivo `grafico_circ.m` poderia ser tentar usar o comando `help`:

```
>> help grafico_circ
error: help: 'grafico_circ' is not documented
```

Vamos alterar o arquivo `grafico_circ.m` de modo que a descrição do arquivo esteja disponível. Para isso, inserimos, na primeira linha do arquivo, o seguinte texto:

```
% grafico_circ: gera o grafico da funcao parametrica
% x = cos(t)
% y = sin(t), t entre 0 e 2*pi
%
% com n pontos, sendo o numero n indicado pelo usuario
```

Usamos o símbolo `%` para introduzir no arquivo um linha que não é executada, ou seja, um *comentário*. Agora temos o seguinte resultado para o comando `help`, no Octave:

```
>> help grafico_circ
'grafico_circ' is a script from the file /home/.../grafico_circ.m

 grafico_circ: gera o grafico da funcao parametrica
 x = cos(t)
 y = sin(t), t entre 0 e 2*pi

 com n pontos, sendo o numero n indicado pelo usuario

Additional help for built-in functions and operators is
available in the on-line version of the manual.  Use the command
'doc <topic>' to search the manual index.

Help and information about Octave is also available on the WWW
at http://www.octave.org and via the help@octave.org
mailing list.
```

No Matlab, o resultado seria o seguinte:

```
>> help grafico_circ
  grafico_circ: gera o grafico da funcao parametrica
  x = cos(t)
  y = sin(t), t entre 0 e 2*pi

  com n pontos, sendo o numero n indicado pelo usuario
```

Podemos usar o símbolo % para introduzir comentários em outros trechos do programa:

```
% grafico_circ: gera o grafico da funcao parametrica
% x = cos(t)
% y = sin(t), t entre 0 e 2*pi
%
% com n pontos, sendo o numero n indicado pelo usuario

n  = input('Entre com o numero de pontos no grafico: ');

t = linspace(0,2*pi,n); % gera valores do parametro t

for i = 1:n             % gera vetores x=cos(t) e y=sin(t)
   x(i) = cos(t(i));
   y(i) = sin(t(i));
end

plot(x,y)               % gera a figura
```

Este símbolo também permite que desativemos temporariamente, e sem apagar, algum comando do script. Por exemplo, se quiséssemos fixar o valor de `n` em 20 sem apagar a linha do comando `input` poderíamos alterar esta linha assim:

```
n = 20; %n  = input('Entre com o numero de pontos no grafico: ');
```

No jargão de programação, dizemos que o trecho `n=input('Entre (...) grafico: ');` foi *comentado*.

3.4 Comandos adicionais de repetição e tomada de decisão

Vimos os dois comandos básicos de repetição e tomada de decisão, o `for` e o `if`. Veremos a seguir dois outros comandos disponíveis no matlab.

3.4.1 O comando `switch`

O comando `switch` é uma versão sofisticada do comando `if` que, ao invés de avaliar uma condição, avalia o conteúdo de uma variável, direcionando o fluxo de controle para comandos distintos. Sua sintaxe geral é a seguinte:

```
switch VAR
   case VALOR1
      comando 1.1
      comando 1.2
      ...
   case VALOR2
      comando 2.1
      ...
   ...
   case VALORN
      comando n.1
      ...
   otherwise
      comando 1
      ...
end
```

Este comando funciona de forma semelhante ao sistema de atendimento automático de um número "0-800". Teste os comandos abaixo salvando-os no arquivo `somas.m`:

```
disp('Tecle 1 para somar 1+1');
disp('Tecle 2 para somar 2+2');
disp('Tecle 3 para somar 3+3');
soma = input('Entre com qualquer outro numero para somar 1+2+3: ');
switch soma
   case 1
      1+1
   case 2
      2+2
   case 3
      3+3
   otherwise
      1+2+3
end
```

Veremos outro exemplo do comando `switch` na Seção 4.5.1.

3.4.2 O comando `while`

O comando `while` é um comando de repetição mais geral que o comando `for` no sentido que o número de repetições não é fixo, mas depende de uma condição se tornar falsa. Sua sintaxe é a seguinte:

```
while (condicao)
  comando 1
  comando 2
  ...
end
```

Por exemplo, a implementação da repetição no cálculo da média poderia ter sido feito da seguinte maneira:

```
MEDIA = 0;
I = 0;
while (I<3)
   I = I + 1;
   MEDIA = MEDIA + NOTA(I);
end
```

Outro exemplo mais sofisticado seria o cálculo aproximado da série $e^x = \sum_{n=0}^{\infty} x^n/n!$:

```
x = input('Entre com o valor de x: ');
serie = 1;
n = 0;
an = 1;
while (abs(an)>1e-10)
   n = n + 1;
   an = an*x/n;
   serie = serie + an;
end
```

OBS: a generalidade do comando `while` traz o risco de travar a execução de um programa pelo que é conhecido como *loop infinito*: uma situação em que a condição avaliada no comando `while` nunca atinge o valor falso. Por exemplo, para evitar um loop infinito no trecho acima, poderíamos impor a condição adicional de que o número de termos nao pode ultrapassar um milhão:

```
nmax = 1000000;
while (abs(an)>1e-10)&&(n<nmax)
   n = n + 1;
   an = x^n/factorial(n);
   serie = serie + an;
end
serie
```

3.5 Exercícios

1. Supondo que a variável x esteja definida, use o comando `if` para calcular $y = \begin{cases} -1, & x < 0 \\ 1, & x \geq 0 \end{cases}$

2. Escreva um script que solicite ao usuário a entrada de um número, armazene este número na variável x, execute a operação acima e exiba o resultado.

3. Escreva um script para solicitar ao usuário dois números, e exibir o produto deles.

4. Use o comando `for` para calcular o fatorial de 20. Compare o resultado com o comando `factorial(20)`.

5. Modifique o script `grafico_circ.m` para gerar o gráfico de $y = x^3$, no intervalo $[-2, 2]$.

6. Modifique o script `grafico_circ.m` para gerar o gráfico de $y = \cos(4\pi x) + x$, primeiro no intervalo $[-0.5, 0.5]$, e depois no intervalo $[-10, 10]$. Note que no primeiro gráfico a parcela $\cos(4\pi x)$ é mais significativa, enquanto no segundo gráfico a parcela x é mais significativa.

7. Faça um script que peça ao usuário para inserir 5 pares ordenados e em seguida faça o gráfico dos pontos correspondentes a estes pares ordenados (não é necessário marcar os pontos isoladamente - pontos consecutivos podem estar ligados por uma linha sólida).

4 Funções

As *funções* são arquivos com extensão `.m`, assim como os *scripts* da seção anterior, mas que funcionam da mesma forma que as funções intrínsecas do matlab. Em nosso primeiro exemplo de função, vamos criar o arquivo `sen.m` com as duas linhas abaixo:

```
function y = sen(x)

   y = sin(x);
```

Por meio deste arquivo, escrever a função seno em português, em vez de utilizar a função intrínseca `sin()`:

```
>> sen(pi/2)
ans =  1
```

4.1 Variáveis de entrada, saída e internas

Em geral, *funções* são arquivos-texto com extensão `.m` que podem ler um conjunto de variáveis (denominadas *argumentos de entrada*) e retornar um outro conjunto de variáveis (denominadas *argumentos de saída*). A principal diferença entre um arquivo *script* e um arquivo *função* é que neste a primeira linha (que não seja um comentário) deve seguir a seguinte sintaxe:

```
function [SAIDA1,SAIDA2,...] = NOME(ENTRADA1,ENTRADA2,...)
```

sendo que `SAIDA1,SAIDA2,...` representa o conjunto de variáveis de saída, `NOME` é o nome da função (que deve coincidir com o nome do arquivo), e `(ENTRADA1,ENTRADA2,...)` representa o conjunto de variáveis de entrada. Para executar no matlab uma função iniciada pela linha acima, devemos entrar, na linha de comando,

```
>> [SAIDA1,SAIDA2,...] = NOME(ENTRADA1,ENTRADA2,...)
```

desde que `ENTRADA1,ENTRADA2,...` estejam bem definidas. É permitido renunciar às variáveis de entrada e/ou saída:

```
function = NOME(ENTRADA1,ENTRADA2,...)
function [SAIDA1,SAIDA2,...] = NOME
function NOME
```

Esta última possibilidade, `function NAME`, levanta uma questão: qual é a diferença entre uma função sem argumentos de entrada/saida e um script ?

A diferença é que as *variáveis internas* da função são eliminadas depois da execução, enquanto as variáveis do script permanecem salvas na memória. Por exemplo, vamos criar o script `teste.m` com apenas uma linha:

```
n = 1
```

Em seguida, vamos executar, na linha de comando,

```
>>clear; teste
n =  1
>> n
n =  1
```

A variável `n` permaneceu na memória. Agora, vamos transformar o script `teste.m` em uma função:

```
function teste
n = 1
```

Agora, o resultado é o seguinte:

```
>>clear; teste
n =  1
>> n
error: 'n' undefined near line 44 column 1
```

Se houver na memória uma variável também chamada `n`, ela não será afetada pela variável interna da função:

```
>>n=3; teste
n =  1
>> n
n =  3
```

No momento da chamada da função, podemos mudar os nomes das variáveis de entrada ou mesmo substituí-las pelos dados de entrada. Podemos também mudar os nomes das variáveis de saída ou omití-las. Por exemplo, considere a seguinte função `mat.m`:

```
function v = mat(x)
if (x>=1)
   for i = 1:x
      v(i) = i;
   end
end
```

Podemos executá-la de diversas maneiras:

```
>> y=5.3;
>> v = mat(y)

v =

     1     2     3     4     5

>> vet = mat(4)

vet =

     1     2     3     4

>> mat(4)

ans =

     1     2     3     4
```

Convém observar que todas as variáveis de saída precisam ser inicializadas na função, ou seja, a função não deve finalizar com variáveis de saída indefinidas. No programa acima, a variável de saída `v` permanecerá indefinida se a variável de saída for menor que 1:

```
>> v=mat(0)
warning: mat: some elements in list of return values are undefined
warning: called from
    mat at line 3 column 1
v = [](0x0)
```

4.2 Funções vs. scripts

Além das variáveis internas, duas diferenças cruciais entre scripts e funções são a forma como cada arquivo é chamado na linha de comando, e a maneira como os dados de entrada e saída são informados.

Por exemplo, vamos rever o exemplo da série $e^x = \sum_{n=0}^{\infty} x^n/n!$ na Seção 3.4.2. Vamos criar o script `script_serie.m` a seguir,

```
% script_serie: calcula a serie associada a exp(x)

x = input('Entre com o valor de x: ');
serie = 1;
n = 0;
an = 10;
nmax = 1000000;
while (abs(an)>1e-10)&&(n<nmax)
   n = n + 1;
   an = x^n/factorial(n);
   serie = serie + an;
end
serie
```

e a função `fun_serie.m` dada por

```
function serie = fun_serie(x)

% fun_serie: calcula a serie associada a exp(x)

%x = input('Entre com o valor de x: ');
serie = 1;
n = 0;
an = 10;
nmax = 1000000;
while (abs(an)>1e-10)&&(n<nmax)
   n = n + 1;
   an = x^n/factorial(n);
   serie = serie + an;
end
```

Vejamos o funcionamento de ambos os programas:

```
>> scr_serie
Entre com o valor de x: 0.2
serie =  1.2214
>> fun_serie(0.2)
ans =  1.2214
```

Os resultados são idênticos, mas note que, para o script `script_serie.m`,

- Executamos na linha de comando chamando `script_serie`;
- Devemos digitar a entrada (o valor de x para o qual devemos calcular e^x);
- Devemos exibir o valor da variável de saída `serie` na última linha do arquivo,

enquanto para a função `fun_serie.m`,

- Executamos na linha de comando chamando `fun_serie(0.2)`, sendo `0.2` o valor de entrada desejado;
- Não é necessário digitar a entrada ou exibir a variável de saída.

Além de valores, poderíamos também inserir variáveis (inicializadas) como argumentos de entradas, e estas variáveis podem ter outros nomes:

```
>> zzz = 0.2;
>> fun_serie(zzz)
ans =  1.2214
```

Aproveitando o exemplo acima, vamos ilustrar a situação em que temos mais argumentos de entrada e de saída. Na função `fun_serie.m` acima, podemos entrar com número máximo de iterações, e retornar quantas iterações efetivamente foram realizadas:

```
function [serie,n] = fun_serie(x,nmax)
% fun_serie: calcula a serie associada a exp(x)

%x = input('Entre com o valor de x: ');
serie = 1;
n = 0;
an = 10;
%nmax = 1000000;
while (abs(an)>1e-10)&&(n<nmax)
   n = n + 1;
   an = x^n/factorial(n);
   serie = serie + an;
end
```

Vejamos o funcionamento da função atualizada:

```
>> fun_serie(0.2,100)
ans =  1.2214
>> [x,n]=fun_serie(0.2,100)
x =  1.2214
n =  8
```

Note que, se omitirmos as variáveis de saída, apenas o primeiro argumento de saída é retornado. Por outro lado, não podemos omitir uma ou mais variáveis de entrada:

```
>> [x,n]=fun_serie(0.2)
error: 'nmax' undefined near line 9 column 26
```

Daqui em diante, vamos nos concentrar no uso de arquivos do tipo função para a criação de funções intrínsecas. Na Seção 2.3 vimos alguns exemplos de funções intrínsecas relacionadas a operações elementares. Vejamos agora mais exemplos de funções intrínsecas, agora associadas a manipulação de matrizes e vetores.

4.3 Extensão das funções elementares para vetores

Na seção 3.3, fizemos um script que gera o gráfico de funções paramétricas a partir de dois vetores. A construção destes vetores, anteriomente realizada com o comando de repetição `for`, torna-se mais simples:[4]

```
n  = input('Entre com o numero de pontos no grafico: ');
t = linspace(0,2*pi,n);
x = cos(t);
y = sin(t);
plot(x,y)
```

Definimos os vetores `x` e `y` simplesmente usando as funções trigonométricas seno e cosseno como se a variável `t` fosse um número. Ao perceber que entramos com o vetor `t` em `x = cos(t)`, o matlab automaticamente gera um vetor `x` com a mesma dimensão do vetor `t`, e cujas coordenadas são dadas por $x_i = \cos(t_i)$, conforme precisamos (analogamente para a função `sin()`). Ou seja: o matlab avalia a função `cos()` componente a componente. No jargão de programação orientada a objetos, dizemos que a função `cos()` foi *sobrecarregada* para operar tanto com números quanto com vetores.

Mais ainda: a função `cos()` funciona também com matrizes:

```
>> A = (pi/4)*[0,1;2,3]
A =
   0.00000   0.78540
   1.57080   2.35619
>> cos(A)
ans =
   1.0000e+00   7.0711e-01
   6.1230e-17  -7.0711e-01
```

A maioria das funções intrínsecas destinadas a números (escalares) também funcionam para matrizes e vetores. Por outro lado, há funções intrínsecas criadas especificamente para vetores e matrizes, como a função `linspace` usada acima. Outras funções serão apresentadas a seguir.

4.4 Funções intrínsecas para vetores e matrizes

O matlab dispõe de diversas funções intrínsecas que extraem informações de vetores. Entre elas, temos a função `length`, que retorna a dimensão de um vetor,

[4] Os comentários foram removidos para evidenciar a simplificação do programa, mas o leitor não deve seguir este mau exemplo.

```
>> length([1,1,2,2,3,3])
ans =  6
```

as funções `max` e `min`, que encontram o máximo e o mínimo do vetor,

```
>> max([1,1,2,2,3,3])
ans =  3
>> min([1,1,2,2,3,3])
ans =  1
```

ou ainda a função `sort`, que ordena o vetor em ordem crescente,

```
>> v = sort([1,3,2,4,3,5])
v =
     1     2     3     3     4     5
```

ou em ordem descrecente:

```
>> v = sort([1,3,2,4,3,5],'descend')
v =
   5   4   3   3   2   1
```

As principais operações típicas da álgebra linear de vetores e matrizes também estão implementadas na forma de funções intrínsecas do matlab. Por exemplo, podemos calcular a norma euclideana (ou norma-2) de um vetor,

```
>> v = [1,2,3]
v =
   1   2   3
>> norm(v)
ans =  3.7417
```

ou se preferirmos, a norma infinito ou norma-1 deste vetor:

```
>> norm(v,inf)
ans =  3
>> norm(v,1)
ans =  6
```

Note que podemos informar na função intrínseca `norm` somente o vetor (neste caso, por *default* a norma euclideana é calculada), ou informar qual tipo de norma desejamos calcular. Este comando permite ainda o cálculo de normas matriciais:

```
>> norm([1,2;3,4],1)
ans =  6
```

Com respeito a matrizes, algumas das funções intrinsecas frequentemente utilizadas são as funções `det(A)` (determinante de A), `inv(A)` (inversa de A) e `eye(n)` (matriz identidade de ordem n). Por exemplo:

```
>> det(4*eye(3))
ans =  64
```

4.4.1 Operações termo a termo

As operações de soma de vetores/matrizes e produto de escalar por vetor/matriz seguem as definições da álgebra linear:

```
>> v = [1,2,3];
>> w = [4 5 6];
>> v+w
ans =
   5   7   9
>> -2*w
ans =
   -8  -10  -12
>> A = [4,2,1 ; 1,4,2 ; 2,1,10];
>> B = eye(3);
>> A + 2*B
ans =
    6    2    1
    1    6    2
    2    1   12
```

Note que estas operações são realizadas termo a termo, o que não ocorre, por exemplo, no produto de matrizes:

```
>> A*B
ans =
    4    2    1
    1    4    2
    2    1   10
>> A^2
ans =
    20    17    18
    12    20    29
    29    18   104
```

Nas operações acima, a expressão `A^2` foi interpretada pelo matlab como o produto `A*A`. Para realizar o produto termo a termo, devemos adicionar o símbolo `.` antes do símbolo do operador:

```
>> A.*B
ans =
    4    0    0
    0    4    0
    0    0   10
>> A.^2
ans =
    16     4     1
     1    16     4
     4     1   100
```

O mesmo vale para vetores:

```
>> v.*w
ans =
    4   10   18
>> v*w
error: operator *: nonconformant arguments (op1 is 1x3, op2 is 1x3)
```

Note que, na segunda operação, o matlab tentou realizar o produto dos vetores `v` e `w` seguindo as regras do produto de matrizes, e detectou dimensões incompatíveis. Esta operação poderia ter sido realizada se antes transformássemos o vetor `v` em um vetor-linha (ou o vetor `v` em um vetor-linha). Isto pode ser feito com a operação de `transposição`, que é realizada com o símbolo `'` e vale tanto para vetores quanto para matrizes:

```
>> v'*w
ans =
    4    5    6
    8   10   12
   12   15   18
>> v*w'
ans =  32
>> A'
ans =
    4    1    2
    2    4    1
    1    2   10
```

Note que a operação `v'*w` gerou a matriz de posto unitário $v^T w$, enquanto a operação `v*w'` realizou o produto interno vw^T.

OBS: Ao atuar em vetores ou matrizes complexas, o símbolo de transposição `'` automaticamente toma o conjugado. Para suprimir o conjugado, devemos utilizar o símbolo composto `.'` , como no exemplo a seguir:

```
>> v = [1,2+i,3*i];
>> v'
ans =
   1 - 0i
   2 - 1i
   0 - 3i
>> v.'
ans =
   1 + 0i
   2 + 1i
   0 + 3i
```

4.4.2 Inicialização de vetores e matrizes com funções intrínsecas

Até agora vimos duas funções intrínsecas que permitem inicializar vetores e matrizes: a função `linspace` e a função `eye`. Outra função intrínseca útil para inicialização de variáveis, e que funciona para vetores e matrizes, é a função `zeros()`:

```
>> A = zeros(4)
A =
   0   0   0   0
```

```
   0   0   0   0
   0   0   0   0
   0   0   0   0
>> B = zeros(4,3)
B =
   0   0   0
   0   0   0
   0   0   0
   0   0   0
>> v = zeros(1,5)
v =
   0   0   0   0   0
```

Note que `zeros(n)` produz uma matriz nula com dimensões $n \times n$, enquanto `zeros(m,n)` produz uma matriz nula com dimensões $m \times n$, ou um vetor linha ou coluna se m=1 ou n=1. Naturalmente, `zeros(1)` e `zeros(1,1)` geram o escalar 0. Existe uma função semelhante à `zeros()` que preenche o vetor/matriz com o número 1 em vez do número 0:

```
>> A=ones(2,3)
A =
   1   1   1
   1   1   1
```

Por outro lado, a função `twos()`, que preencheria o vetor/matriz com o número 2, não é uma função intrínseca. Entretanto, ela pode ser criada pelo usuário, como veremos a seguir

4.5 Funções definidas pelo usuário

Criar a função `twos()` é uma tarefa simples, e semelhante à criação de scripts. Basta gerar o arquivo `twos.m` com o seguinte conteúdo:

```
function A = twos(m,n)

   A = 2*ones(m,n);
```

Ela função vai funcionar *quase* perfeitamente:

```
>> twos(4,3)
ans =
   2   2   2
   2   2   2
   2   2   2
   2   2   2
>> twos(4)
error: 'n' undefined near line 3 column 17
error: evaluating argument list element number 2
error: evaluating argument list element number 1
error: called from:
error:   .../twos.m at line 3, column 6
```

Posteriormente vamos corrigir esta função para que funcione com `twos(4)`.

4.5.1 Extraindo informações das variáveis de entrada

Veremos alguns recursos que permitem extrair informações das variáveis de entrada de uma função, visando atingir os objetivos desta função.

A função `nargin` (*Number of ARGuments of INput*) permite identificar quantos argumentos de entrada foram fornecidos pelo usuário. Ela será usada para corrigir nossa função `twos.m`:

```
function A = twos(m,n)

   if(nargin==2)
      A = 2*ones(m,n);
   else
      A = 2*ones(m);
   end
```

Vamos testar esta nova versão:

```
>> twos(2,3)
ans =
   2   2   2
   2   2   2
>> twos(2)
ans =
   2   2
   2   2
```

Para mais de dois argumentos de entrada, convém usar o comando `switch`:

```
function A = twos(m,n,p)

switch nargin
 case 1
  A = 2*ones(m);
 case 2
  A = 2*ones(m,n);
 case 3
  A = 2*ones(m,n,p);
 end
```

Por outro lado, há a possibilidade do usuário não informar a dimensão da matriz:

```
>> twos
error: 'm' undefined near line 6 column 12
error: evaluating argument list element number 1
error: evaluating argument list element number 1
error: called from:
error:   /home/saulo/twos.m at line 6, column 3
```

Em casos como este, podemos criar nossas próprias mensagens de erro por meio do comando `error`:

```
function A = twos(m,n,p)

switch nargin
 case 1
  A = 2*ones(m);
 case 2
  A = 2*ones(m,n);
 case 3
  A = 2*ones(m,n,p);
 otherwise
  error('Faltaram os argumentos de entrada');
 end
```

Deste modo, podemos informar ao usuário de forma mais precisa o que houve de errado:

```
>> twos
error: Faltaram os argumentos de entrada
error: called from:
error:   /home/.../twos.m at line 11, column 3
```

Para introduzir o próximo recurso, a função `size`, vamos tentar criar uma função que calcule o traço de uma matriz:

$$tr(A) = \sum_{i=1}^{n} A_{ii}.$$

A função `tr.m` terá um argumento de saída, o traço da matriz. Quanto as argumentos de entrada, temos duas opções:

- Entrar com a matriz e sua dimensão;
- Entrar somente com a matriz, e calcular sua dimensão dentro da função.

Podemos facilmente programar o primeiro caso:

```
function x = tr(A,n)

x = 0;
for i = 1:n
   x = x + A(i,i);
end
```

O resultado segue no exemplo abaixo:

```
>> A = [1,2;3,4];
>> tr(A,2)
ans =  5
```

A segundo opção requer o cálculo das dimensões da matriz, que efetuaremos com o comando `size`:

```
function x = tr(A)

[m,n]=size(A);
```

```
x = 0;
for i = 1:m
   x = x + A(i,i);
end
```

Este comando retorna o número de linhas (`m`) e o número de colunas (`n`) da matriz. Na linha de comando, a execução se torna mais confortável para o usuário:

```
>> tr(A)
ans =  5
```

4.6 Funções recursivas

Assim como em várias linguagens de programação, o matlab permite o uso de funções recursivas, ou seja, funções que chamam a si mesmas. Um exemplo típico é o cálculo do fatorial de um número natural:

$$n! = (n)(n-1)(n-1)\cdots(3)(2)(1) = n[(n-1)(n-1)\cdots(3)(2)(1)] = n(n-1)!$$

Observando acima que o fatorial de n pode ser definido em termos do fatorial de $n-1$, podemos implementar a seguinte função:

```
function y = fatorial(n)

if(n==0)
   y = 1;
else
   y = n*fatorial(n-1);
end
```

Funções recursivas dependem de um comando de tomada de decisão (como o `if`) para saber quando parar a recursão. Por exemplo, se o programa acima fosse simplesmente

```
function y = fatorial(n)

y = n*fatorial(n-1);
```

o comando `fatorial(2)` levaria à execução dos comandos `fatorial(1)`, `fatorial(0)`, `fatorial(-1)`,`fatorial(-2)`, ..., até esgotar a memória disponível.

4.7 Exercícios

1. Crie a função `ln(x)`, a versão em português da função `log(x)` (logaritmo natural).
2. Escreva uma função que calcule, a partir de um vetor `x`, o vetor `y` tal que $y_i = x_i^2$ para todo i.
3. Altere o programa `tr.m` acima de modo que ele acuse erro se a matriz não for quadrada.
4. Crie uma função cuja variável de entrada seja um vetor de dimensão arbitrária e cuja saída seja a média e a variância dos componentes do vetor.
5. Altere o programa de cálculo recursivo do fatorial para exibir uma mensagem de erro caso o número `n` não seja natural.

5 Manipulação de arquivos

Esta seção apresenta dois pares de comandos básicos para leitura e gravação de variáveis em arquivos, `load/save` e `dlmread/dlmwrite`.

5.1 load/save

Este primeiro par de comandos é o que permite ler/gravar variáveis da forma mais simples. Por exemplo, vamos executar os seguintes comandos[5]:

```
>> x = linspace(0,2*pi,100);
>> y = sin(x);
>> save
>> exit
```

O penúltimo comando salva todas as variáveis no arquivo binário `matlab.mat`, enquanto o último comando encerra o matlab. Ao iniciar novamente o matlab, as variáveis podem ser recuperadas por meio do comando `load`:

```
>> x(1)
Undefined function 'x' for input arguments of type 'double'.

>> load

Loading from: matlab.mat

>> x(1)

ans =

     0
```

Podemos escolher outro nome de arquivo escrevendo-o logo depois do comando `save`. Também podemos usar estes comandos para ler/gravar uma ou mais variável, escrevendo-a(s) logo após o nome do arquivo:

```
>> clear
>> load matlab.mat y
>> x(1)
Undefined function 'x' for input arguments of type 'double'.

>> y(1)

ans =

     0

>> save y.mat y
```

[5]No Octave, o comando `save` pode não funcionar sem que um nome de arquivo seja dado. Neste caso use, por exemplo, `save vars.mat`, e posteriormente, substitua `load` por `load vars.mat`.

Existe também a opção de salvar o arquivo em formato ASCII (por exemplo, `save y.dat y --ascii`). Embora seu conteúdo seja mais acessível, um arquivo ASCII tem a desvantagem de criar arquivos que ocupam mais espaço do que arquivos binários. Entretanto, quando devemos iteragir com outros aplicativos, a opção de arquivos ASCII pode ser conveniente. Vamos tratar deste caso a seguir.

5.2 dlmread/dlmwrite

Os comandos `load/save` servem sobretudo para uso interno do matlab. Para o intercâmbio de arquivos com outros aplicativos, recomenda-se o segundo par de comandos, `dlmread/dlmwrite`. Por exemplo, o uso do comando `dlmwrite` como se segue,

```
>> x = linspace(0,2*pi,100);
>> y = sin(x);
>> A = [x',y'];
>> dlmwrite('temp.dat',A)
```

cria um arquivo com o seguinte conteúdo:

```
0,0
0.0062895,0.0062894
0.012579,0.012579
...
```

A limitação deste comando, se comparado ao comando `save`, é que ele grava uma única variável – note que foi criada a matriz auxiliar `A` cujas colunas correspondem às variáveis `x` e `y`.

O número de casas decimais pode variar a depender do *default* do matlab. Entretanto, podemos escolher o número decimais, por exemplo dez casas:

```
>> dlmwrite('temp.dat',A,'precision',10);
```

Pode-se também mudar o caracter que separa os dados em uma coluna. Por exemplo, para separar as colunas via tabulação:

```
>> dlmwrite('temp.dat',A,'delimiter','\t','precision',10);
```

A iteração com outros aplicativos pode requerer a introdução de um um cabeçalho. Por exemplo, os comandos

```
>> dlmwrite('temp.dat','xy');
>> dlmwrite('temp.dat',A,'-append','precision',10);
```

geram o arquivo `temp.dat` com o seguinte conteúdo:

```
x,y
0,0
0.06346651825,0.06342391966
0.1269330365,0.1265924536
...
```

O comando correspondente para leitura é o `dlmread` (por exemplo, `B=dlmread('temp.dat');`). Se o arquivo tiver um cabeçalho, como o arquivo `temp.dat` criado no exemplo anterior, devemos indicar o tipo de separador e quantas linhas e colunas devem ser ignoradas (no exemplo anterior, 1 linha e 0 colunas):

```
>> B=dlmread('temp.dat',',',1,0);
```

6 Vetorização de operações

A vetorização de operações permite que um programa se torne mais compacto e, muitas vezes, mais veloz. A vetorização geralmente é realizada com o auxílio do operador `:`, detalhado a seguir.

6.1 O operador :

Usamos o operador `:` pela primeira vez na Seção 3.2.2:

```
for I = 1:3
   MEDIA = MEDIA + NOTA(I);
end
```

Experimente remover o `for` da primeira linha:

```
>> I = 1:3
I =
   1   2   3
```

Assim, `1:3` é um vetor que vai de 1 até 3 com intervalos de 1 a 1, ou seja, é o vetor que contém os valores que o contador `i` assume no comando `for` acima. Podemos substituir `1` e `3` por valores quaisquer, embora os resultados podem não ser aqueles desejados:

```
>> I = 1.4:4.2
I =
    1.4000    2.4000    3.4000

>> I = 4.2:1.4
I =
   Empty matrix: 1-by-0
```

No primeiro exemplo acima, o vetor é formado incrementando o valor 1.4 de 1 em 1 até que que o valor 4.2 seja atingido. Entretanto, este processo produziria o valor $4.4 > 4.2$, que é descartado. Já no segundo exemplo, não é possível atingir 1.4 somando incrementos de 1 unidade a 4.2, logo o vetor gerado é vazio. Em geral,

$$a : b = \begin{cases} [a, a+1, a+2, \ldots, a+\lfloor b-a \rfloor], & a \leq b; \\ \text{vetor vazio}, & a > b, \end{cases}$$

sendo $\lfloor x \rfloor$ a parte inteira do número x, que aliás é obtida no matlab pela função `floor`:

```
>> floor(4.2)
ans =
     4
```

Assim como no comando `for`, podemos criar vetores com um incremento diferente de 1:

```
>> 1:4:23
ans =
     1     5     9    13    17    21

>> 4:-0.5:2
ans =
    4.0000    3.5000    3.0000    2.5000    2.0000
```

Em geral,

$$a:c:b = \begin{cases} [a, a+c, a+2c \ldots, a+\lfloor (b-a)/c \rfloor], & (b-a)c \geq 0 \text{ e } c \neq 0; \\ \text{vetor vazio}, & (b-a)c < 0 \text{ ou } c = 0. \end{cases}$$

6.2 Composição de vetores e matrizes

Dado um vetor `v` de comprimento m, seja `w` um vetor de comprimento n tal que

$$w_i \in \mathbb{N} \text{ e } 1 \leq w_i \leq m \qquad (1 \leq i \leq n). \tag{4}$$

O matlab define `v(w)` como sendo o vetor `u` de comprimento n tal que

$$u_i = v_{w_i}, \quad 1 \leq i \leq n.$$

Por exemplo,

```
>> v = [10,20,30,40];
>> w = [4,4,3,2,1];
>> v(w)
ans =
    40    40    30    20    10

>> v([3,3])
ans =
    30    30

>> v([3,3,5])
Index exceeds matrix dimensions.
```

Note que o vetor $w = [3, 3, 5]$ do último exemplo viola a condição (4), pois $w_3 = 5$, enquanto `v` tem tamanho 4.

Esta operação pode ser interpretada como a função composta $f_u = f_v \circ f_w$ formada pelas seguintes funções associadas aos vetores `v` e `w`:

$$f_v : \begin{array}{ccc} \{1,\ldots,n\} & \to & \mathbb{R} \\ i & \mapsto & v_i, \end{array} \qquad f_w : \begin{array}{ccc} \{1,\ldots,m\} & \to & \{1,\ldots n\} \\ i & \mapsto & w_i. \end{array}$$

Analogamente, podemos compor o vetor `v` com uma matriz `A`, dado que suas entradas safisfaçam $A_{ij} \in \{1, \ldots, n\}$:

```
>> A = [1,2;3,3];
>> v(A)
ans =
    10    20
    30    30
```

No caso em que `v` é uma matriz, podemos fazer composições com os índices das linhas e/ou os índices das colunas. Especificamente, dada uma matriz $n \times m$ `v`, se os vetores `w` e `z` de comprimento p e q satisfazem $w_i \in \{1, \ldots, m\}$ e $z_i \in \{1, \ldots, n\}$, então `A=v(w,z)` produz a matriz

$$A = \begin{bmatrix} v_{w_1,z_1} & v_{w_1,z_2} & \ldots & v_{w_1,z_q} \\ v_{w_2,z_1} & v_{w_2,z_2} & \ldots & v_{w_2,z_q} \\ \vdots & \vdots & \ddots & \vdots \\ v_{w_p,z_1} & v_{w_p,z_2} & \ldots & v_{w_p,z_q} \end{bmatrix}.$$

```
>> v = [1,2;3,3]; w = [1,2,1]; z = [2,1];
>> v(w,z)
ans =
     2     1
     3     3
     2     1
```

Em particular, podemos fazer a composição de uma tabela (matriz/vetor) com os vetores formados pelo operador : com o objetivo de *extrair partes da tabela:*

```
>> A = [1,5,9;2 6 10;3 7 11;4 8 12]
A =
     1     5     9
     2     6    10
     3     7    11
     4     8    12
>> A(1:2,1:3)
ans =
     1     5     9
     2     6    10
>> A(1:2,[1,2,3])
ans =
     1     5     9
     2     6    10
>> A(1:4,2)
ans =
     5
     6
     7
     8
>> A(3,1:3)
ans =
     3     7    11
```

OBS: Quando queremos extrair uma coluna inteira ou uma linha inteira, podemos usar simplesmente :,

```
>> w = A(3,:)
w =
     3     7    11
```

e se quisermos iniciar de um certo número até a maior posição possível, usamos o termo `end`:

```
>> A(2:4,1:2:3)
ans =
     2    10
     3    11
     4    12
>> A(2:end,1:2:end)
ans =
     2    10
     3    11
     4    12
```

6.3 Vetorização de comandos de atribuição

A composição de tabelas com vetores nos permite, além de extrair trechos da tabela, atribuí-los sem a necessidade do comando `for`:

```
>> v = [1 3 2 5 6 1 4 3]
>> clear w
>> for i = 2:5; w(i) = v(i); end; w
w =

     0     3     2     5     6
>> clear w
>> w(2:5) = v(2:5)
w =
     0     3     2     5     6
```

Note que, de modo a eliminar o comando `for` neste exemplo, foi suficiente substituir `i` por `2:5`, que corresponde ao "argumento de entrada"do comando for. A linha

```
>> w(2:5) = v(2:5);
```

corresponde à versão vetorizada do trecho

```
>> for i = 2:5
       w(i) = v(i);
   end
```

OBS: Em geral, o comando `for` admite vetores que não sejam definidos pelo operador `:`

```
>> soma = 0;
>> v = [5 3 8];
>> for i = v; soma = soma + i; end; soma
soma =
    16
```

6.3.1 Funções intrínsecas úteis para vetorização

No último exemplo, o operador `:` não é suficiente para vetorizar o somatório dos elementos do vetor `v`:

```
>> soma = 0;
>> v = [5 3 8];
>> soma = soma + v
soma =
     5     3     8
```

Neste caso, precisamos de uma função intrínseca que nos auxilie na vetorização, a função `sum`:

```
>> soma = sum(v)
soma =
    16
```

Além da função `sum` para somatórios, temos funções para produtórios, médias, máximos e mínimos, entre outros:

```
>> prod(v)
ans =
   120
>> mean(v)
ans =
    5.3333
>> min(v)
ans =
     3
>> max(v)
ans =
     8
```

Quando estes comandos são aplicados a matrizes, a operação é realizada fixando-se o primeiro índice, ou seja, o índice das linhas:

```
>> A = [1,5,9;2 6 10;3 7 11;4 8 12]
A =
     1     5     9
     2     6    10
     3     7    11
     4     8    12
>> sum(A)
ans =
    10    26    42
```

Para realizar a operação fixando-se o segundo índice, ou seja, o índice das colunas, basta incluir o argumento opcional 2.

```
>> sum(A,2)
ans =
    15
    18
    21
    24
>> sum(A,1)
ans =
    10    26    42
```

Note que `sum(A,1)` efetua a soma fixando-se o primeiro índice, que é o *default*. Para efetuar a soma ao longo de ambos os índices, usamos `sum(sum())`:

```
>> sum(sum(A))
ans =
    78
```

6.4 Exercícios

1. Dado $v = [v_1, v_2, \ldots, v_{15}]$, qual é o resultado do comando `v(1:2:5)` ?

2. Vetorize o trecho de programa

```
for i = 1:n
   b(i) = v(2,i);
   for j = 1:2:n
      A(i,2) = 3;
   end
end
```

3. Vetorize o trecho de programa

```
for i = 2:2:10
   A(i,3) = C(i) - d(i+4);
end
```

4. Os operadores lógicos e relacionais também podem ser usados em vetorização:

```
>> v = [-1,2,3-4]
v =
    -1     2    -1
>> v>=0
ans =
     0     1     0
```

Utilize os operadores `>=` e `.*` para gerar, de forma vetorizada, o vetor $\vec{v}^+$ tal que

$$v_i^+ = \begin{cases} v_i, & v_i \geq 0; \\ 0, & v_i < 0. \end{cases}$$

Referências

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys 23*(1), 5–48.

Higham, D. e N. Higham (2002). *MATLAB guide*. Philadelphia: SIAM.

Kincaid, D. e W. Cheney (2002). *Numerical Analysis: Mathematics of Scientific Computing*. Providence: American Mathematical Society.

PET-EngComp (2008). Mini-curso de MATLAB e Octave para Cálculo Numérico. Universidade Federal do Espírito Santo.

Quarteroni, A. e F. Saleri (2007). *Cálculo Científico com MATLAB e Octave*. Milão: Springer-Verlag Itália.