# Capítulo 11
# Arquivos M de funções

# Considerações iniciais

- Uma função (bem programada) é como uma caixa preta. Não se tem contato com as variáveis e comandos intermediários. Fornece-se os argumentos de entrada e obtém-se o resultado, sendo esta a única forma de comunicação com a função.
- São ferramentas poderosas para execução de comandos que contêm funções matemáticas, ou seqüências de funções que se repetem com alguma freqüência quando se resolve algum problema.
- Para que possamos usufruir das vantagens que as funções, vistas como caixas pretas, fornecem, o **MATLAB**disponibiliza recursos que permitem que possamos construir as nossas próprias funções em um arquivo M de formato especial.

# Arquivos M de funções

- Consulte Help –> **MATLAB**Help –> **MATLAB**–> Programming and... –> M-file programming –> {Function, Subfunction, Private, etc}
- Uma função em um arquivo M é um arquivo de extensão **.m**, com uma sintaxe específica, que diz ao **MATLAB**que o mesmo é um arquivo M de função.
- Este arquivo normalmente é editado em uma janela de edição separada (do próprio **MATLAB**, ou não).

Um arquivo M de função comunica-se com o espaço de trabalho **apenas** através de seus argumentos de entrada e saída. Esta é *a principal diferença* entre um arquivo M de comandos (como vimos antes) e um arquivo M de função.

# Um exemplo: a função *mmempty*

Vejamos um exemplo de arquivo M de função, usando o editor de arquivos M do **MATLAB**, e abrindo o arquivo mm6files/mmempty.m. Note as seguintes características:

- O nome da função é o mesmo que o nome do arquivo, sem a extensão .m.
- A primeira linha do arquivo é a *linha de declaração da função*. Contém a palavra `function` seguida pela sintaxe para utilização da função: seu nome e seus argumentos de entrada e saída.

# Ainda o o exemplo

- A primeira linha de comentário, chamada de linha **H1**, contém o nome da função, seguido de uma descrição sucinta do objetivo da função. Esta é a linha que o comando $lookfor$ exibe quando executa sua busca.
- As linhas de comentários após H1 descrevem sintaxes alternativas de chamadas da função, algoritmos usados e exemplos, se necessário. Estas são as linhas exibidas em resposta aos comandos $help\ mmempty$ e $helpwin\ mmempty$.

# Nomes de funções

- Os nomes de funções estão sujeitos às mesmas regras que os nomes de variáveis.
- Apesar de os nomes de funções poderem ter até 31 caracteres, como estes devem ser os mesmos nomes dos arquivos, este número pode ser limitado pelo sistema operacional em que o **MATLAB**estiver rodando.
- O **MATLAB**diferencia maiúsculas e minúsculas. O Unix/Linux também diferencia, mas as plataformas Windows não. Para evitar dependências de plataformas, convencionou-se o uso apenas de minúsculas para nomes de funções.
- Padronizou-se que os nomes de funções que aparecem no texto de ajuda de uma função estejam em maiúsculas.

# Funcionalidade

- Uma execução de um arquivo M de função pára quando é encontrado, ou o fim do arquivo, ou um comando `return`.
- Os arquivos M de funções podem conter chamadas para arquivos de comandos. Quando um arquivo de comando é encontrado, este é executado no espaço de trabalho da função e não do **MATLAB**.

# Arquivos privados

- Os arquivos M de funções podem chamar arquivos M *privados*. Estes são arquivos M de função que residem no subdiretório *private* do diretório onde reside o arquivo original. Apenas as funções que estão no diretório “pai” do subdiretório privado podem chamar as funções contidas neste último. O escopo de validade dos nomes dessas funções é limitado ao diretório pai.
- Subdiretórios privados normalmente contêm funções úteis (somente) para as funções que estão no diretório pai.
- Os nomes dos arquivos privados seguem as mesmas regras de nomes de arquivos. Convencionou-se que estes nomes possuem o prefixo `private_`.

# Subfunções

- Os arquivos de funções podem conter mais de uma função. A primeira, que leva o mesmo nome do arquivo, é a *função primária*. As outras, *subfunções*, podem aparecer em qualquer ordem após a função primária.
- As subfunções também começam com a palavra reservada `function` e seguem todas as regras de formação das funções primárias. Padronizou-se adicionar ao nome das subfunções o prefixo `local_`.
- As subfunções podem ser chamadas somente pela função primária, e por outras subfunções que estejam no mesmo arquivo.
- Cada subfunção possui sua área de trabalho individual. Isto é, cada uma delas é, por si própria, uma caixa preta.
- O comando $helpwin\ \ func/subfunc$, onde $func$ é o nome da função primária e $subfunc$ é o nome da subfunção procurada, exibe os textos de ajuda desta subfunção.

# Funções úteis

- A função $error$ exibe um texto na tela, interrompe a execução da função e devolve o controle para o teclado. O texto exibido é então passado à função $lasterr$. Exemplo:

```
if length(val) > 1
    error('VAL deve ser um escalar.')
end
```

- A função $warning$ é usada para exibir uma mensagem de alerta na janela de comandos. Neste caso, não ocorre interrupção da execução da função.
  - A diferença entre $warning$ e $disp$ neste caso é que a primeira pode ser ativada ou não com o comando $warning\ \ on$ e $warning\ \ off$.

# Argumentos de entrada e saída

- As funções do **MATLAB**podem ser chamadas com qualquer número de argumentos de entrada e saída, inclusive zero.
- As funções podem ser chamadas com menos (e nunca com mais) argumentos do que os especificados na definição da função.
- Se não forem atribuídos valores a uma ou mais variáveis especificadas como de saída (veremos como a seguir), ou se for usado o comando $clear$, a função não retornará nenhum valor.
- Quando é feita uma chamada a uma função, as variáveis de entrada não são copiadas para o espaço de trabalho da função, mas seus valores podem ser usados pela função. Entretanto, qualquer alteração nos valores força a cópia das variáveis afetadas no espaço de de trabalho da função.

  É a filosofia *delayed copy*. Note que $y = myfunction(x, y)$ ocasiona a cópia imediata da variável $y$ para o espaço de trabalho da função.

# Argumentos $varargin$ e $varargout$

- São vetores de células pré-definidos, utilizados para viabilizar um número variável de argumentos de entrada ($varargin$) e saída ($varargout$).
- A $i$-ésima célula destes vetores representa o $i$-ésimo argumento a partir do ponto em que aparecem.
- Para que uma uma função aceite um número variável de argumentos de entrada (saída), $varargin$ ($varargout$) deve aparecer como último argumento na linha de declaração da função.
- Exemplo:

$$\boxed{function[a, varargout] = myfunction(x, y, varargin)}$$

Neste caso, se executarmos $[a, b, c] = myfunction(x, y, z)$, os argumentos de entrada serão, $x$, $y$, $varargin\{1\}$ e os de saída serão $a$, $varargout\{1\}$, $varargout\{2\}$.

# Funções úteis

- Os números exatos de argumentos de entrada e saída de uma função podem ser determinados por chamadas às funções $nargin$ ($nargout$).
  - Exemplo: A função *mmdigit* (arquivo m de função mmdigit.m) usa $nargin$ para atribuir valores padrão a argumentos de entrada não fornecidos pelo usuário.

# Espaço de trabalho de funções

- Voltando ao conceito de caixa preta, funções recebem dados de entrada, dados, trabalham com estes dados e retornam dados de saída. Todas as variáveis intermediárias permanecem ocultas para o espaço base de trabalho do **MATLAB**.
- Cada função tem seu próprio espaço de trabalho temporário, criado quando a função é chamada e extinto quando termina sua execução.
- Funções do **MATLAB**podem ser chamadas recursivamente, mas cada chamada tem seu próprio espaço de trabalho.
- Existem algumas técnicas de comunicação entre os espaços de trabalho de diferentes funções (além dos parâmetros de entrada e saída). Veremos estas técnicas brevemente a seguir.

# Compartilhamento de variáveis

- Variáveis declaradas como globais podem ser compartilhadas entre diversas funções e com o espaço de trabalho do **MATLAB**.
- Para declarar uma variável como global usamos o comando $global$. A variável global existirá apenas nos espaços de trabalho em que for declarada.
- Exemplo de bom uso de variáveis globais: funções do **MATLAB**$tic$ e $toc$. Veja.
  - Funcionam juntas como um cronômetro para marcar o tempo das operações executadas pelo **MATLAB**.
- O uso de variáveis globais não deve ser encorajado. Entretanto, se for necessário usá-las é aconselhável que estejam sempre em maiúsculas e tenham nomes descritivos da sua f finalidade.

# Variáveis persistentes

- Funções podem compartilhar a mesma variável em diferentes chamadas, recursivas ou não, declarando uma variável como $persistent$.

  `persistent` example

- Variáveis persistentes atuam como globais, em um escopo limitado pela função onde foram declaradas. Isto é, as variáveis persistentes existem enquanto o arquivo M permanecer na memória do **MATLAB**.

- A função $mmclass$ ilustra o uso de variáveis persistentes. Veja o arquivo $mmclass.m$.

  - A variável $clist$ é uma variável persistente. Na primeira vez em que $mmclass$ é chamada, $clist$ está vazia e é então preenchida. Em chamadas futuras, $clist$ existirá e não será criada novamente.

# Funções $evalin$ e $assignin$

- $evalin$ : permite fazer acesso a outro espaço de trabalho, avaliar uma expressão e retornar o resultado para o espaço de trabalho corrente.
  - Permite que a expressão seja avaliada no espaço de trabalho de onde a função foi chamada, ou no espaço base de trabalho da janela de comandos.
  - Possui uma versão de chamada com $catch$.
- $assignin$ : Associa valores de uma expressão no espaço de trabalho corrente a uma variável em outro espaço de trabalho.
- Assim, $assignin('workspace','vname', X)$ faz $vname = X$, onde $X$ está no espaço de trabalho corrente e $vname$ está no espaço de trabalho definido por $'workspace'$, que pode ser igual a $'caller'$ (o espaço de onde feita a chamada) ou $'base'$ (o espaço base do **MATLAB**).

# Função $inputname$

- $inputname$ : Usada para determinar os nomes das variáveis usadas quando uma função é chamada. Por exemplo, suponha que uma função seja chamada com

  ```
  >> y = myfunction(xdot,time,sqrt(2)).
  ```

  Chamando `inputname(2)` dentro de `myfunction` retorna como resultado a string `time`. Já, se colocarmos `inputname(3)`, temos como resultado uma matriz vazia porque `sqrt(2)` não é uma variável e sim uma expressão que produz um resultado temporário.

- A função $mmswap$ ilustra o uso das funções $evalin$, $assignin$ e $inputname$. Veja o arquivo mmswap.m.

- A variável $mfilename$ armazena o nome do arquivo M que está sendo executado.

# Dualidade comando-função

- Assim como criamos funções no **MATLAB**, podemos criar comandos (como $clear$, $whos$, $help$, entre outros).
- Existem basicamente duas diferenças entre comandos e funções:
  - Comandos não possuem argumentos de saída;
  - Os argumentos de entrada não precisam estar entre parêntesis.
- Na verdade, os comandos são chamadas de funções que obedecem às regras acima. Assim,

| help myfunction | é equivalente a | help('myfunction') |
|---|---|---|

*O* **MATLAB***interpreta os argumentos do comando como strings, e a chamada à função é feita colocando estas strings entre parêntesis. Isto vale para qualquer comando do* **MATLAB***.*

# Dualidade comando-função

- Tanto os comandos como as funções podem ser digitados na janela de comando. Isto é, no exemplo anterior, o **MATLAB**aceitaria ambas as versões de chamadas à função help. Usar comandos, normalmente, resulta em menos digitação.
- Considere o exemplo a seguir:

| >> which fname | >> S = which('fname') | >> S = which fname |
|---|---|---|

  - O **MATLAB**aceita as duas primeiras opções sendo que na segunda a saída é armazenada na variável $S$.
  - O último caso não é aceito porque há uma mistura das sintaxes de comandos e de funções. O **MATLAB**assume que após o sinal de '=' existe uma função e interpreta o restante da sentença como uma função. Isto força a necessidade de argumentos separados por vírgulas e entre parêntesis.

# Dualidade comando-função

Resumindo

- Comandos e funções são chamadas a funções.
- Comandos são traduzidos pela interpretação de seus argumentos como strings, colocando-os entre parêntesis, e chamando a função requisitada.
- Qualquer chamada a uma função pode ser feita na forma de um comando se
  - A função não tiver argumentos de saída; e
  - se os seus argumentos de entrada forem apenas strings.

# Função $feval$

- Já falamos rapidamente sobre a função $feval$ quando falamos sobre execução de strings; vamos agora aprofundar-nos um pouco mais. Relembrando:
  - $feval(F, x_1, \ldots, x_k)$ : executa a função $F$ tendo como parâmetros de entrada $x_1, \ldots, x_k$.
  - Além disso, a função $feval$ não recorre a todo o interpretador do **MATLAB**, o que a torna mais eficiente do que $eval$. Eficiência intensificada quando a função precisa ser avaliada muitas vezes.
- Vamos ver duas formas alternativas para o uso de $feval$:
  - com *function handles*;
  - com *in line functions*

# Function handles

- Para melhorar o desempenho de $feval$ é necessário fornecer ao **MATLAB**mais informações sobre uma função que será executada do que apenas o seu nome.
- Ao criar um “function handle object”, o **MATLAB**extrai todas as informações necessárias sobre a função e as oculta dentro deste novo tipo de dado.
- Uma vez criado, este “function handle” simplesmente substitui o nome da string como argumento de $feval$.

```
>> fhan = @sin %@ operador de criação, criação direta de seno(x)
fhan =
    @sin
>>fhan(2) = @cos % criação direta de cos(x)
fhan =
    @sin       @cos
>> disp(feval(fhan(1), pi/4)) % executa sin(pi/4)
    0.7071
```

Note que ’fhan’ é um vetor de células

# Function handles

- Podemos usar $str2func$ para criar uma function handle quando a string que denomina a função é conhecida.

```
>> fhan = str2func('humps'); % cria humps(x)
>> fhan(2) = str2func('cos') % cria cos(x)
fhan =
    @humps    @cos
>> fstr = {'humps','cos'} % cria vetor de células
fstr =
    'humps'    'cos'
>> fhan = str2func(fstr) % cria ambos a partir de células
fhan =
    @humps    @cos
```

- Uma vez criada uma function handle, a função $functions$ fornece informações sobre o seu conteúdo:

```
>> disp(functions(fhan(2)))
    function: 'cos'
        type: 'simple'
    file: 'MATLAB built-in function'
```

# In line functions

- Quando precisamos executar uma string que é a própria função que queremos calcular, não podemos usar $feval$ diretamente porque esta string não é um arquivo M de função:

```
>> myfunc = '100*(y-x^2)^2+(1-x)^2';
>> x = 1.2; y = 2;

>> feval(myfunc)
??? Error using ==> feval
Invalid function name '100*(y-x^2)^2+(1-x)^2'.

>> eval(myfunc) % mas eval funciona
ans =
   31.4000
```

- Para contornar o problema acima, o **MATLAB**criou um objeto chamado *in line function*.

# In line functions

- Criação e manipulação de in-line functions: $inline$, $argnames$, $formula$ e $fcnchk$.

```
>> myfuni=inline(myfunc,'x','y')
myfuni =
     Inline function:
     myfuni(x,y) = 100*(y-x^2)^2+(1-x)^2

>> disp(feval(myfuni,x,y))
   31.4000
>> disp(feval(myfuni,1,0))
   100

>> disp(argnames(myfuni))
    'x'
    'y'
>> disp(formula(myfuni))
100*(y-x^2)^2+(1-x)^2
```

# A função $fcnchk$

- A utilidade das funções in-line vem do fato de $feval$ trabalhar tão bem com todas estas funções como com arquivos M e function handles.
- É preciso que uma função que irá executar uma string de caracteres determine a representação que aquela string está usando (arquivo M, function handle, ou por definição).
- A função $fcnchk$ é usada para receber esta string e prepará-la para que seja usada como argumento de entrada para outra função através da função $feval$.
  - Se o dado de entrada for um arquivo M ou um function handle, $fcnchk$ retorna o nome;
  - Se o dado de entrada for uma definição completa de uma função, $fcnchk$ cria e retorna uma função in-line.

# *fcnchk* - exemplo

```
>> disp(fcnchk(fhan(1))) % function handle
   @humps

>> F = 'cos'; % arquivo M
>> disp(fcnchk(F))
cos

>> F = '100*(y-x^2)^2+(1-x)^2'; % definição
>> fcnchk(F)
ans =
    Inline function:
    ans(x,y) = 100*(y-x^2)^2+(1-x)^2
>> disp(feval(ans,1,0))
  100

>> myfunc = fcnchk(F)
myfunc =
    Inline function:
    myfunc(x,y) = 100*(y-x^2)^2+(1-x)^2
>> disp(formula(myfunc))
100*(y-x^2)^2+(1-x)^2
```

# Considerações finais - $argnames$

- A função $argnames$ retorna os argumentos em uma string.

  É preciso alguma atenção quando os argumentos estão em um vetor.

```
>> disp(F)
100*(y-x^2)^2+(1-x)^2
>> F = strrep(F,'x','x(1)');
>> F = strrep(F,'y','x(2)')
F =
100*(x(2)-x(1)^2)^2+(1-x(1))^2
>> myfunc = fcnchk(F)
myfunc =
    Inline function:
    myfunc(x) = 100*(x(2)-x(1)^2)^2+(1-x(1))^2
>> disp(argnames(myfunc))
   'x'
>> feval(myfunc,[1 0]) % erro se executado com apenas 1 argumento
ans =
  100
```

# Função *tic*

```
function tic
%TIC start a stopwatch timer
%   The sequence of commands
%       TIC, operation, TOC
%   prints the number of seconds required for the operation.
%
%   See also TOC, CLOCK, ETIME, CPUTIME

%   Copyright (c) 1984-1999 The MathWorks, Inc. All Rights Reserved.
%   $Revision: 5.7 $  $Date:1999/01/12  16:43:03 $
% TIC simply strores CLOCK in a global variable
global TICTOC
TICTOC = clock;
```

# Função *toc*

```
function t=toc
% TOC Read the stopwatch timer
%    Toc prints the elapsed time (in seconds) since TIC was used.
%    t = TOC; saves the elapsed time in t, instead print it out.
%
%  See also TIC, CLOCK, ETIME, CPUTIME.

%   Copyright (c) 1984-1999 The MathWorks, Inc. All Rights Reserved.
%   $Revision: 5.8 $  $Date:1999/01/12  16:43:03 $
% TOC uses ETIME and value of CLOCK saved by TIC.
global TICTOC
if isempty(TICTOC)
   error('You must call TIC before calling TOC.');
end
if nargout < 1
   elapsed_time = etime(clock,TICTOC)
else
   t = etime(clock,TICTOC)
```