# Métodos Computacionais para a Engenharia Eletrotécnica

## Ficha Laboratorial 2 – Controlo de fluxo de programa

## 1. Objetivo

A presente ficha de trabalho laboratorial tem como objetivo a introdução dos seguintes conceitos:

1. Interação com o utilizador: funções **`input`**, **`disp`** e **`fprintf`**.
2. Estruturas de controlo de fluxo de programa.
3. Depuração (*debugging*) de *scripts.*

Recomenda-se, para além da leitura atenta deste enunciado, a consulta dos *slides* das aulas teóricas e dos apontamentos introdutórios de MATLAB.

## 2. Interação com o utilizador

Na execução de um *script*, a interação com o utilizador pode ser feita quer solicitando a introdução de dados, quer apresentado mensagens e/ou resultados das operações. Para tal, podemos recorrer aos seguintes comandos em MATLAB:

- **`input()`** – solicita informação ao utilizador. O argumento de esta função é uma *string* que serve para passar uma mensagem ao utilizador, ficando a aguardar que este introduza algum valor, seguido da tecla <*enter*>. O utilizador pode passar qualquer valor – numérico ou não numérico – ou mesmo variáveis que estejam no *workspace*. Notar ainda que:
    - se o utilizador premir apenas <*enter*>, sem introduzir qualquer valor, a função `input()` devolve uma matriz de carateres vazia, `[]`.
    - se o utilizador introduzir uma expressão inválida, a função devolve uma mensagem de erro na janela de comando. De seguida, volta a apresentar a mensagem ao utilizador e fica a aguardar novamente pela introdução de dados.
    - se a informação a ser solicitada ao utilizador for uma *string*, então deve ser ainda incluído o argumento `'s'`:

```
price = input('Indique o preço: ');                    % Aguarda por um valor
currency = input('Indique a moeda (Euro/Dólar): ', 's'); % Aguarda por uma string
```

- **`disp()`** – permite fazer a apresentação (*display*) de uma *string* na janela de comandos. Recebe como argumento uma variável *string,* ou, em alternativa, pode passar-se um vetor de carateres com recurso a parêntesis retos, `[]`.

```
dia = 13;
mes = 'fevereiro';
disp(dia);          % Apresenta o conteúdo da variável dia
disp(mes);          % Apresenta o conteúdo da variável mes
```

```
disp('Isto é um exemplo do comando disp()');  % Uso de uma string
disp(['Hoje é ', num2str(dia), ' de ', mes]); % Uso de vetor de carateres
```

Note que, no último exemplo, para construir uma mensagem mais complexa, constituída por uma série de carateres e o valor de duas variáveis, **toda a informação foi concatenada num único vetor de carateres através dos parêntesis retos**. Chama-se à atenção para o facto de, por a variável `dia` ser uma variável numérica, ser necessário a conversão desta variável numa cadeia de carateres ASCII que representem este mesmo número, conversão esta conseguida com a utilização da função **`num2str()`**. Se a variável `dia` for passada neste último exemplo sem usar esta função, a função **`disp()`** iria interpretar o valor 11 como sendo o carater ASCII 13 (*CR –carriage return*).

- **`fprintf()`** – faz a formatação de dados e apresenta o resultado no ecrã. Recebe como argumento uma *string* que define a formatação dos dados, seguido dos valores que se pretendem apresentar. No caso de a *string* de formatação não conter quaisquer referências a valores a apresentar, esta é então tratada apenas como uma mensagem de texto a devolver ao utilizador.

```
dia = 13;
mes = 'fevereiro';
fprintf('%d\n', dia);          % Apresenta o conteúdo da variável dia
fprintf('%s\n', mes);          % Apresenta o conteúdo da variável mes
fprintf ('Isto é um exemplo do comando fprintf()\n');
fprintf('Hoje é %d de %s.\n', dia, mes);
```

Repare que, na última linha do exemplo acima, para passar os valores na mensagem a apresentar no ecrã, deve recorrer ao operador de formatação que começa o sinal de percentagem, **`%`**. A este símbolo podem seguir-se alguns carateres de formatação (que não serão aqui explorados), terminando imperativamente com um carater de conversão. A título de exemplo, na última linha do código atrás apresentado, `%d` é o operador que indica que o valor a passar é do tipo inteiro. Sendo o `%d` o primeiro operador que surge na *string* de formatação, o argumento imediatamente a seguir a esta *string* será então tratado como um valor inteiro, neste caso, o valor inteiro 13.

Note ainda que, neste exemplo, as *strings* de controlo terminam todas com a sequência `'\n'`, isto para garantir que, após apresentada a mensagem, é iniciada uma nova linha (se for essa a intenção, claro). Se omitir esta sequência, podem acontecer algo parecido com o seguinte:

```
>> price = 11.2;
>> fprintf('O preço é %.2f euros', price)  % '%.2f' precisão de duas casas
O preço é 11.20 euros>>
```

Repare que, neste caso, o *prompt* aparece imediatamente a seguir à mensagem que foi apresentada. Esta situação pode facilmente ser corrigida ao colocar a sequência `'\n'` no final da *string*. Também pode notar aqui a utilização da formatação **`'%.2f'`** que fixa o número de casas decimais (duas) imediatamente a seguir ao ponto decimal. Na Tabela 1, estão indicados alguns dos operadores de formatação comummente usados. Mais detalhes sobre formatação de texto a ser passado pela função **`fprintf`** pode ser consultado em https://www.mathworks.com/help/matlab/matlab_prog/formatting-strings.html.

Tabela 1. Alguns operadores de formatação

| Código | Descrição da conversão |
|---|---|
| **%s** | Converte string de caracteres |
| **%d** | Número em notação decimal (com sinal) |
| **%f** | Número em notação de vírgula fixa |
| **%e** | Notação científica (%E também é admissível) |
| **%g** | Notação mais compacta entre os resultados obtidos com %e e %f (omitindo zeros sem significado) |

Uma última nota sobre o comando **`fprintf`**: este comando permite ainda a escrita em ficheiros, sendo que esta vertente não será explorada neste trabalho.

## 3. Controlo de fluxo de programa

Como em qualquer linguagem de programação, MATLAB permite a definição de blocos de código que se podem repetir, repetição esta que pode ser condicionada a um número de ciclos de repetições – **ciclos *for*** – ou condicionadas à verificação de uma condição – **ciclos *while***. Ainda é possível condicionar a execução de determinados blocos de código em função da verificação de determinadas condições – **estruturas *if-then-else*** e **estruturas *switch***.

Estas estruturas são em todo semelhantes àquelas que já foram introduzidas na unidade curricular de Programação de Computadores, sendo apenas necessário ter atenção à sua sintaxe em MATLAB. Não se pretende, por isso, fazer aqui uma apresentação muito exaustiva, mas antes apresentar a sua sintaxe genérica.

Apenas uma breve observação comum a todas estas estruturas: todas elas são sempre limitadas pela respetiva palavra reservada – **`if`**, **`while`**, **`for`**, e **`switch`** – e a palavra reservada **`end`**, não havendo necessidade de limitar blocos de código com chavetas, tal como acontece, por exemplo, na linguagem de programação C.

### 3.1 Ciclo `for`

Repete o bloco de código delimitado pelas palavras reservadas **`for`** e **`end`** um pré-determinado número de vezes. A sua sintaxe é[1]:

```
for var = start:step:end
    % instruções a executar pelo ciclo for ficam
    % entre as palavras reservadas for e end
End
```

Por exemplo, o seguinte bloco permite mostrar uma mensagem informando o índice de cada valor no vetor valores:

```
% definição do vetor valores
valores = randi(10, 1, 5);

% ciclo for para percorrer o vetor valores
for i = 1:length(valores)
    disp(['O valor com índice ', num2str(i), ' é ', num2str(valores(i))])
end
```

[1] Aqui é apresentada a sintaxe mais comum. Para mais detalhes sobre como usar o ciclo *for*, consulte a documentação help do MATLAB.

### 3.2 Ciclo while

Repete o bloco de código delimitado pelas palavras reservadas **`while`** e **`end`** enquanto a condição for verdadeira.

```
while <condição>
    % instruções a executar pelo ciclo while ficam
    % entre as palavras reservadas while e end
end
```

No exemplo seguinte, é usado um bloco while para somar números inteiros a partir do 1, até que a soma ultrapasse o valor 100:

```
sum = 0;
count = 1;

while sum <= 100
    sum = sum + count;
    count = count + 1;
end
```

### 3.3 Estrutura if-then-else

Executa um determinado bloco de código se a condição **`if`** for verdadeira.

```
if <condição1>
    % se se verificar a condição1, executa o bloco de código
    % entre as palavras reservadas if e elseif
elseif <condição2>
    % se se verificar a condição2, executa o bloco de código
    % entre as palavras reservadas elseif e else
else
    % se não se verificar nenhuma das condições, executa
    % o bloco de código entre as palavras reservadas else e end
end
```

Esta estrutura pode incluir novas condições a verificar, caso a condição verificada no **`if`** seja falsa, recorrendo para tal a **`elseif`**. Caso não se verifique nenhuma das condições, é executado o bloco a seguir à palavra reservada **`else`**. Os blocos **`elseif`** e **`else`** são facultativos, podendo um bloco **`if`** incluir vários blocos **`elseif`**.

Considere, por exemplo, o seguinte código que serve para determinar se o valor introduzido pelo utilizador é negativo, zero ou positivo:

```
x = input('Indique um valor para a variável x: ');

if x < 0
    disp('x é negativo')
elseif x == 0
    disp('x é zero')
else
    disp('x é positivo')
end
```

### 3.4 Estrutura switch

Esta estrutura executa condicionalmente um bloco de código de entre várias escolhas disponíveis, sendo cada escolha associada a uma declaração **`case`**. A sintaxe é:

```
switch <expressão>
    case valor1
        % executa este bloco de código
    case valor2
        % executa este bloco de código
    case {valor3, valor4}
        % neste caso, executa este bloco de código caso se
        % verifique qualquer dos valores contidos nas chavetas
    otherwise
        % este bloco é facultativo
end
```

Notar que, ao contrário do que acontece, por exemplo, nas linguagens C, C++ ou C#, o bloco de cada **case** não termina com um **break**. Ainda, o bloco de código associado ao caso **otherwise** é facultativo.

No exemplo que se segue, é usada uma estrutura **switch** para determinar o número de dias do mês que é atribuído à variável **month**:

```
month = 'Abril';

switch month
    case {'Janeiro', 'Março', 'Maio', 'Julho', 'Agosto', 'Outubro', 'Dezembro'}
        disp([month ' tem 31 dias']);
    case {'Abril', 'Junho', 'Setembro', 'Novembro'}
        disp ([month ' tem 30 dias']);
    case 'Fevereiro'
        disp ([month ' tem 28/29 dias']);
    otherwise
        disp('Não foi introduzido nenhum mês válido');
end
```

É importante notar, neste caso, que a variável **month** recebe uma *string*. O valor desta variável deve coincidir exatamente com um dos valores listados nos **cases** que definem o bloco **switch**, caso contrário é devolvida a mensagem de erro definida no bloco **otherwise**, devendo, por isso, dar-se atenção às letras maiúsculas e minúsculas. Dito de outra forma, se o mês de abril tivesse sido escrito como **'ABRIL'** ou **'abril'**, por exemplo, iriamos receber a mensagem correspondente a um mês inválido.

### 3.5 Funções break e continue

A função **break** permite terminar um ciclo **for** ou **while**, não sendo executada qualquer instrução a seguir à declaração do **break**. Considere-se, por exemplo o seguinte ciclo while:

```
count = 1;
sum = 0;

while true            % Este ciclo while é um ciclo infinito
    if count > 10     % count já ultrapassou o valor 10?
        break;        % Sim! Termina o ciclo while
    end
    sum = sum + count;
    count = count + 1;
end
```

Neste bloco de código, o ciclo **while** irá adicionar os números inteiros de 1 a 10. O ciclo, tal como definido, é um ciclo infinito que, a cada ciclo, verifica se a variável **count** já ultrapassou o valor 10. Se não for o caso,

adiciona esse valor à variável **sum**, e incrementa **count**. Se, no entanto, **count** for maior que 10, o ciclo é terminado sem ser executada mais qualquer instrução.

A função **continue**, por outro lado, força o ciclo **for** ou **while** a avançar para o seguinte ciclo de execução, ignorando todas as instruções que se encontrem depois da instrução **continue**. Por exemplo, o seguinte ciclo **for** irá adicionar apenas os números ímpares compreendidos entre 1 e 20:

```
sum = 0;

for count = 1:20
    if rem(count, 2) == 0  % O número é par?
        continue;          % Sim! Passa para o ciclo seguinte
    end
    sum = sum + count;
end
```

## 4. Operações de comparação e operadores lógicos

Regra geral, as operações de comparação são *element-wise,* i.e., elemento a elemento. Se considerarmos um vetor ou uma matriz, as operações de comparação devolverão sempre um vetor ou uma matriz de valores booleanos, respetivamente:

```
>> a = [5 10 20];
>> b = [10 5 40];
>> a > 12
ans =
  1×3 logical array
   0   0   1
>> a <= b
ans =
  1×3 logical array
   1   0   1
```

Em relação aos operadores lógicos, os operadores **e (&),** e **ou (|)** são também elemento a elemento:

```
>> a = [4  6  5  9  9];
>> b = [4  5  7  6  2];
>> (a > 5) & (b < 8)
ans =
  1×5 logical array
   0   1   0   1   1
```

Apesar de os operadores **&** e **|** poderem ser usados como operadores lógicos na verificação de condições de execução em estruturas de controlo de programa, é recomendada a utilização dos operadores lógicos de curto-circuito **e (&&)** e **ou (||)**. A diferença destes operadores, quando comparados com os operadores **&** e **|**, reside no facto de que nem todas as condições de uma expressão booleana são verificadas. Considere-se o seguinte exemplo:

```
x = 10;

if (x < 15) | (x > 20)
    fprintf('Fora do intervalo\n');
end

if (x < 15) || (x > 20)
    fprintf('Fora do intervalo\n');
end
```

As duas estruturas **if** irão devolver o mesmo resultado. A diferença reside em que, no primeiro **if** as duas condições lógicas, `x < 15` e `x > 20`, são avaliadas para a determinação do resultado da operação **ou**. No segundo caso, ao ser avaliada a condição `x < 15`, como esta é verdadeira, o resultado da operação **ou** é independente da segunda condição, pelo que esta nem sequer é avaliada. De modo semelhante, numa operação curto-circuito **&&,** assim que a primeira condição é falsa, o resultado da operação é independente das condições seguintes, pelo que não são avaliadas.

Finalmente, deve notar que os operadores lógicos de curto-circuito apenas operam sobre valores lógicos escalares, não sendo usados sobre vetores e/ou matrizes.

## 5. Depuração de *scripts*

Durante a escrita de um programa, podem ocorrer erros no código que serão naturalmente necessários de corrigir. Estes erros podem ser **erros de sintaxe**, **erros de execução (*runtime errors*)**, e **erros de lógica.**

### 5.1 Erros de sintaxe

Os erros de sintaxe ocorrem quando se escreve de forma incorreta uma função ou declaração. Neste caso, o MATLAB devolve sempre uma mensagem de erro:

```
>> 100 = x
 100 = x
     ↑
Incorrect use of '=' operator. Assign a value to a variable
using '=' and compare values for equality using '=='.

>> a = 0:0.1:
 a = 0:0.1:
           ↑
Error: Invalid expression. Check for missing or extra characters.
```

No primeiro exemplo, é ilegal a utilização do operador "**=**", uma vez que não podemos usar um número como variável. No segundo caso, logo a seguir aos dois pontos, " **:** ", deveria constar um valor para assim definir o vetor `a`. Em qualquer destes casos, é apresentada uma mensagem de erro a explicar qual o problema encontrado, acompanhada de uma pequena seta[2] a apontar o sítio onde provavelmente o erro foi cometido. Noutros casos, esta indicação gráfica da possível localização do erro poderá não ser apresentada:

```
>> A = [1 2 3; 4 5 6 7]
Error using vertcat
Dimensions of arrays being concatenated are not consistent.
```

Enquanto existirem erros de sintaxe no código, o *script* não consegue ser executado.

### 5.2 Erros de execução

Os erros de execução são erros que se verificam quando, apesar da sintaxe estar correta, a instrução está escrita de forma a que é produzido um erro durante a execução do *script*:

```
>> A = [1 2];
>> B = A^2;
Error using  ^  (line 52)
Incorrect dimensions for raising a matrix to a power. Check that the matrix is
square and the power is a scalar. To perform elementwise matrix powers, use '.^'.
```

[2] Em versões anteriores é apenas uma linha vertical, mas o efeito é o mesmo: indicar o local onde provavelmente foi cometido o erro.

Neste exemplo, apesar da instrução `B = A^2;` estar corretamente escrita, é gerado um erro ao executar este código porque a matriz A não é quadrada.

## 5.3 Erros lógicos

Os erros lógicos são os mais complicados – e frustrantes! – de detetar, uma vez que não geram qualquer erro. No entanto, este tipo de erros levam a obtenção de resultados inesperados e, obviamente, incorretos. Considere-se, por exemplo, o seguinte código:

```
n = 1;
while n ~= 12
    fprintf('n = %d\n', n)
    n = n + 2;
end
fprintf('terminou\n')
```

Neste caso, o código é executado sem gerar qualquer erro. No entanto, a condição de paragem – *n* diferente de 12 – nunca se irá verificar, originando um ciclo *while* infinito.

## 5.4 Depuração (*debugging*)

A depuração de um código consiste no processo sistemático de remoção de quaisquer erros que tenham sido detetados. O editor do MATLAB faculta um conjunto de ferramentas que facilitam a realização desta tarefa, muitas vezes fastidiosa. De seguida, será feita uma abordagem sumária da depuração de código em MATLAB.

Considere-se o seguinte script temperatura.m:

*temperatura.m*

```
1  T = input('Temperatura: ');
2  if  T > 37.5
3      fprintf('Estado febril\n');
4  elseif 36 <= T <= 37.5
5      fprintf('Temperatura normal\n');
6  else
7      fprintf('Hipotermia\n');
8  end
```

Ao executar este script na linha de comando podemos verificar:

```
>> temperatura
Temperatura: 35
Temperatura normal
```

o que não corresponde à resposta esperada. Para verificar qual o erro que está a ser cometido, podemos colocar um *breakpoint* na linha 4. O *breakpoint* é um ponto de paragem incondicional, sendo a execução do código suspensa quando chega a esta linha. A maneira mais simples de inserir um *breakpoint* é clicando no número da linha onde se pretende suspender a execução do código (Figura 1).

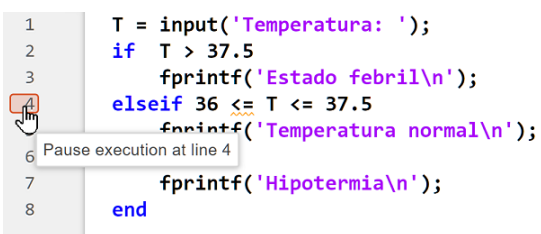


Figura 1 – Definição de um *breakpoint* na linha 4.

O *breakpoint* é assinalado com um retângulo avermelhado a identificar a linha 4[3]. De seguida, podemos executar o *script* e, após indicar o valor da temperatura, verificamos que o programa é suspenso, sendo indicada a linha onde isso ocorreu. Neste momento, o MATLAB está em modo *debug*, o que pode ser confirmado pelo *prompt* **`K>>`**:

```
>> temperatura
Temperatura: 35
4   elseif 36 <= T <= 37.5
K>>
```

A partir daqui podemos, por exemplo, executar o *script* linha a linha, premindo no botão **Step**, localizado no menu RUN da barra de ferramentas do editor (Figura 2), ou premindo simplesmente F10. Podemos ainda retomar a execução normal do *script* premindo **Continue** (ou F5), ou para o programa (botão **Stop** ou shift+F5).


Figura 2. Menu RUN, com destaque ao botão **Step**.

Também podemos usar a linha de comando para tentar determinar o erro que está a ser cometido. Por exemplo, podemos, neste ponto, verificar a condição do `elseif` na linha 4:

```
K>> 36 <= T <= 37.5
ans =
  logical
   1
```

ou seja, a condição está a devolver o valor booleano 1 (verdade)! Porquê? Vamos verificar novamente a condição, desta vez por partes:

```
K>> 36 <= T
ans =
  logical
   0
```

Como seria de esperar 36 < 35, logo obtivemos o valor lógico 0 (falso), e aqui está o problema: a comparação seguinte é 0 < 37.5, o que é verdade. Se olhar com atenção, qualquer valor menor ou igual que 37.5 será sempre classificado como "temperatura normal" porque fica preso nesta condição. O problema está, pois, na forma como esta condição foi escrita. A forma correta seria fazer `elseif (T >= 36) && (T <= 37.5)`.

Para mais informações sobre depuração de código em MATLAB, pesquise a expressão "*Debug MATLAB Code Files*" ou consulte a página https://www.mathworks.com/help/matlab/matlab_prog/debugging-process-and-features.html.

## 6. Atividades

### 6.1 Atividade 1 – Introdução

Nesta primeira atividade pretende-se simplesmente implementar o *script* apresentado e introduzir as alterações solicitadas.

a) Crie um *scritpt* com nome **`atividade1_f2.m`**.

[3] Em versões anteriores, os *breakpoints* são identificados com um pequeno círculo vermelho junto ao respetivo número da linha.

b) Edite o *script*, incluindo as seguintes linhas de código (naturalmente, exclua a numeração das linhas!):

```
1     clc;
2     ano_nasc = input('Indique ano em que nasceu: ');
3     ano = input('Indique o ano presente: ');
4     nome = input('Indique o seu nome: ', 's');
5
6     disp([nome ', se já celebrou o seu aniversário este ano,']);
7     disp(['então já completou ', num2str(ano - ano_nasc), ' anos.']);
```

c) Grave o *script* e execute-o na linha de comando. Verifique se cometeu algum erro ao transcrever o *script* e corrija, se necessário.

d) Altere o *script* nas linhas 6 e 7 de forma a utilizar agora a função **`fprintf`**.

e) Adicione agora no *script* as linhas de código necessárias para o cálculo do Índice de Massa Corporal (IMC). Para tal. O *script* deverá solicitar ao utilizador o peso, em quilogramas, e a altura, em metros. Depois do cálculo do IMC (IMC = peso/(altura)$^2$), deverá apresentar ao utilizador uma mensagem com a indicação do peso e da altura introduzidos, e o valor do IMC, tal como a seguir exemplificado:

```
Indique o seu peso (kg): 85
Indique a sua altura(m): 1.75
Com um peso de 85.0 e uma altura de 1.75 o seu IMC é de 27.8;
```

Use a função `fprintf` para formatar os valores numéricos passados na mensagem ao utilizador (altura, duas casas decimais; peso e IMC, uma casa decimal).

f) De seguida, o *script* deverá indicar a classificação do IMC, segundo a seguinte tabela:

| IMC | Classificação |
|---|---|
| < 18.5 | Peso baixo |
| 18.5 – 25.0 | Peso normal |
| 25.0 – 30.0 | Excesso de peso |
| 30.0 – 35.0 | Obesidade Grau I |
| 35.0 – 40.0 | Obesidade Grau II |
| > **40.0** | Obesidade Grau III |

A mensagem a apresentar deverá o nome do utilizado (guardado na variável `nome`) e a classificação do seu IMC, sendo formatada como no exemplo seguinte:

```
José, o seu IMC indica que apresenta excesso de peso.
```

## 6.2 Atividade 2

Nesta atividade pretendem-se desenvolver um conjunto de pequenos *scripts* que façam uso das estruturas de fluxo de programa, acima introduzidas:

a) Crie um novo *script* (**par_impar.m**) que peça ao utilizador a introdução de um número e que afixe no ecrã a paridade do mesmo (par ou ímpar). O programa deverá continuar a pedir números ao utilizador até que este introduza um valor menor ou igual que zero. (*Dica: recorra à função* `rem` *ou* `mod` *para testar a paridade).*

b) Recorrendo a uma estrutura **`switch`**, escreva um *script* (**clubes.m**) que peça ao utilizador a cor da camisola de um grande clube de futebol português e depois afixe no ecrã o nome do referido clube.

c) A carga num condensador de um circuito RC série é descrita pela expressão:

$$Q(t) = C\,V\,(1 - e^{-t/RC})$$

considerando que a carga do condensador é nula para $t$ = 0. Considerando $V$ = 24 V, $R$ = 10Ω e $C$ = 2 F, escreva, com recurso ao uma estrutura de repetição **`while`**, um *script* (**carga.m**) que apresente o tempo e carga do condensador a cada 0.1 s. Estes valores deverão ser calculados até que a carga do condensador exceda os 3 C (i.e., o último valor apresentado pode ser igual ou superior a 3 C).

### 6.3 Atividade 3

a) Crie um novo *script* com o nome **rectangulo.m**.

b) Escreva o código que permita "desenhar", com recurso ao carater asterisco (*), um retângulo de comprimento e de altura definidos pelo utilizador. O valor do comprimento do retângulo indicado pelo utilizador deverá ser sempre maior ou igual a 5 e menor ou igual a 40. A altura deverá ser maior ou igual a 3 e menor ou igual a 10. Caso o utilizador introduza um valor fora destes intervalos, o *script* volta a fazer o pedido (*dica: recorra a estruturas while para fazer esta verificação*). Um exemplo de funcionamento é apresentado a seguir.

```
Indique o comprimento do rectângulo [5 - 40]: 50
Indique o comprimento do rectângulo [5 - 40]: 20
Indique a altura do rectângulo [3 - 10]: 3
********************
********************
********************
```

## 7. Fora da sala de aula

### 7.1 Exercício 1

Grave o *script* escrito na alínea c) da atividade 2 com o nome **carga2.m**. Altere este script de forma a que ele calculo e apresente o tempo e carga do condensador a cada 0.1 s, sendo que estes valores deverão ser calculados de forma a que o último valor de carga do condenador nunca deverá ser superior aos 3 C (*dica: poderá recorrer a uma função* **`break`**)

### 7.2 Exercício 2

Pretende-se escrever um *script* (**lanca_dado.m**) que simule o lançamento contínuo de um dado. O valor obtido deverá então ser comparado com o número de lançamento. Se o valor obtido no dado for maior ao número do lançamento correspondente, o dado é lançado novamente. Caso contrário o script termina. O script corre de forma automática, sem intervenção do utilizador. A seguir é apresentado um exemplo da informação que é apresentada ao utilizador durante a execução do script proposto:

```
Lançamento n.º 1 <= 2
Lançamento n.º 2 <= 5
Lançamento n.º 3 <= 6
Lançamento n.º 4 <= 2
Fim do script
```

### 7.3 Exercício 3

a) Crie um novo *script* **areas.m**.

b) Escreva um programa que permita calcular a área de uma figura plana a escolher de entre as opções de um pequeno menu de seleção. Este menu deverá ter entradas "Círculo", "Retângulo" e "Hexágono regular"[4]. Após selecionada a figura, o *script* solicita ao utilizador informações sobre as dimensões da mesma (*dica: recorra a uma estrutura switch para fazer a gestão das opções do menu*). Tenha em conta ainda as seguintes caraterísticas de funcionamento do *script*:

   i. Caso seja selecionada uma opção não prevista no menu, é devolvida uma mensagem de alerta, terminando de seguida o *script*.

   ii. Não há necessidade de verificar se as dimensões solicitadas pelo utilizador são positivas.

   iii. O resultado deve ser apresentado na forma de uma mensagem dirigida ao utilizador, e.g., "`A área do retângulo é 200.00 mm^2`".

   iv. O script termina logo após a apresentação do resultado ou caso tenha sido introduzida uma opção não prevista no menu.

Para ilustração considere-se o seguinte exemplo de execução do *script*:

```
>> areas
Menu
1. Círculo
2. Retângulo
3. Hexágono regular
Indique a sua opção: 5
*** Opção inválida! ***

>> areas
Menu
1. Círculo
2. Retângulo
3. Hexágono regular
Indique a sua opção: 1
Indique o valor do raio do círculo: 10

A área do círculo é 314.16 mm^2.
```

c) Faça agora uma cópia do *script* e guarde-o com o nome **areas2.m** (pode fazer gravar o *script* inicialmente criado com "Save as…" no menu do editor).

d) Altere o *script* de maneira a que, no final da execução do *script*, seja perguntado ao utilizador se pretende repetir a execução do *script.* Se o utilizador premir a letra 's' o *script* é repetido. Qualquer outra letra termina o programa. Tenha em atenção que a letra 's' pode ser introduzida como letra maiúscula ou minúscula.

```
>> areas2

Menu
1. Círculo
2. Retângulo
```

[4] Área de um hexágono regular: $\frac{3\sqrt{3}}{2}a^2$, sendo $a$ o lado do hexágono regular.

```
3. Hexágono regular
Indique a sua opção: 2
Indique o valor da base: 10
Indique o valor da altura: 5

A área do retângulo é 50.00 mm^2.

Deseja repetir o programa [S/n]: n
```

## 7.4 Exercício 4

a) A fiabilidade de um equipamento eletrónico é geralmente medida em termos de tempo médio entre falhas (MTBF), onde MTBF é o tempo médio que o equipamento pode operar antes que ocorra uma falha nele. Para grandes sistemas contendo muitos equipamentos eletrónicos, é habitual determinar a MTBF de cada componente, sendo o MTBF geral do sistema calculado a partir das taxas de falha de cada um dos componentes que o constitui. Se o sistema está estruturado de forma semelhante ao apresentado na Figura 3, o MTBF do sistema global pode ser expresso por:

$$\mathrm{MTBF} = \frac{1}{\frac{1}{\mathrm{MTBF}_1} + \frac{1}{\mathrm{MTBF}_2} + \ldots + \frac{1}{\mathrm{MTBF}_n}}$$

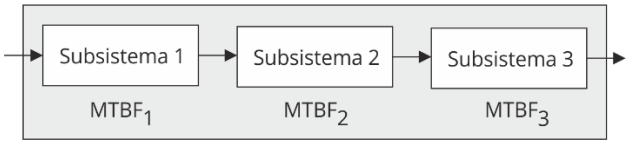


Figura 3. Exercício 3

Escreva um *script* (**calc_MTBF.m**) que solicite ao utilizador o número de subsistemas que compõe o sistema global e, de seguida, solicite o MTBF de cada subsistema. No final, deverá apresentar uma mensagem com o MTBF global, com duas casas decimais de precisão. Por exemplo, considere a seguinte execução de uma possível implementação do código:

```
>> calc_MTBF
Indique o número de subsistemas: 3
MTBF do subsistema 1 (em horas): 4000
MTBF do subsistema 2 (em horas): 2000
MTBF do subsistema 3 (em horas): 800
O MTFB de todo o sistema é de 500.00 horas
```

b) Grave uma cópia o *script* da alínea anterior com o nome **calc_MTBF2.m**, e edite o código de forma a agora respeitar as seguintes condições:

   a. O utilizador não poderá indicar um número de subsistemas superior a 10 e inferior a zero;

   b. O utilizador não poderá introduzir um MTBF igual ou menor que zero.