

# Školení MATLAB II


## NÁSTROJE PRO ROZSÁHLEJŠÍ APLIKACE A POKROČILÁ PRÁCE S GRAFIKOU


### Pokročilé datové typy

- **Základní datové typy**
  - číselné datové typy
    - *double, single*
    - *int8, int16, int32, int64*
    - *uint8, uint16, uint32, uint64*
    - *logical*
  - znaky a řetězce
    - *char*, znak „, '“
    - *string*, znak „, "“
- **Pole buněk**
  - *cell, { }*
    - může obsahovat data různých datových typů a velikostí
    - data jsou v buňkách
      - odkazujeme se na ně pozicí buňky = numerické indexování
        - řádek-sloupec, lineární, logické
    - může být "děravé"

[1 2 ... 158]	$\begin{bmatrix} 0,81 & 0,91 & 0,27 \\ 0,90 & 0,63 & 0,55 \\ 0,12 & 0,09 & 0,96 \end{bmatrix}$	-126
'Ahoj'	$\begin{Bmatrix} .. & .. & .. \\ .. & .. & .. \\ .. & .. & .. \end{Bmatrix}$	

- zadání výčtem prvků
  - `>> h = {[4 5 3 1 4], [2 1 5]}`
- výpis buněk
  - `>> h(2) ... vrátí buňku`

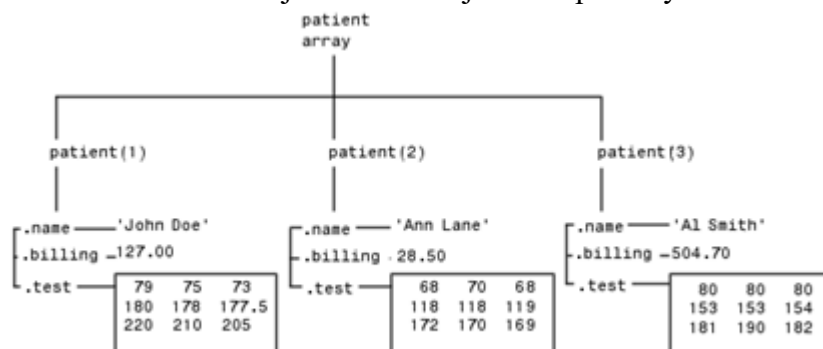

  - `>> h{2} ... vrátí obsah buňky`


  - `>> h{2}(3) ... vypíše 3. prvek vektoru v 2. buňce`
  - Grafické zobrazení – dvojklik v okně *Workspace*
- dynamická tvorba
  - `>> h{3} = 'nezadáno'`
  - `>> h{4} = [4 1 6 1]`
- pole buněk může být jedno-, dvou- i vícerozměrné
- předdefinování prázdného pole dané velikosti *cell(m,n)*

- pole buněk je skládanka
  - `>> info = 'hody kostkou'`
  - `>> vse_pb = {info, h}`
  - `>> vse_pb{2}{4}(3)` ... vypíše 3. prvek vektoru v 4. buňce pole z druhé buňky, tedy hodnotu 6
- pole buněk se často používají při práci s řetězci
  - popisky os grafů, ...
  - dvojklikem importovat NejvetsiMestaCR.xls > Column vectors > jmena mest => Cell Array (od R2017a nastavit v Text Options)
  - označit Mesto a Pocetobyvatel v okně *Workspace* > *PLOTS* > *Pie*

## • Struktura

- *struct*, znak "."
  - může obsahovat data různých datových typů a velikostí
  - data jsou v položkách
    - odkazujeme se na ně jménem položky



- dynamická tvorba
  - `>> kruh.stred = [2,3]`
  - `>> kruh.polomer = 6`
    - ⇒ struktura se dvěma položkami
    - postupné naplňování struktury
- grafické zobrazení - dvojklik v okně *Workspace*
  - možnost editace hodnot, jmen položek
- dotaz na položky
  - `>> S = pi*kruh.polomer^2`
    - *Tab* ... doplňuje jména položek
- přidání dalších "záznamů" ⇒ pole struktur
  - `>> kruh(2).stred = [1,1]`
  - `>> kruh(2).polomer = 3`
  - pole struktur může být jedno-, dvou- i vícerozměrné
- dotaz na položky v poli struktur
  - `>> kruh(1).polomer` ... dotaz na 1. strukturu a její položku polomer
  - `>> kruh(1)` ... dotaz na 1. strukturu a všechny její položky
- pole struktur můžeme spojit, mají-li stejná jména položek
  - `>> [kruh(2), kruh(1)]`
- získání dané položky od všech záznamů (struktur)
  - `>> kruh.polomer` %výstup je tzv. "comma separated list"
  - `>> p = [kruh.polomer]` ... sloučení hodnot vedle sebe
  - `>> s = cat(1,kruh.stred)` ... sloučení hodnot pod sebe
- předdefinování struktury nebo pole struktur
  - funkce *struct*

- jména položek lze zadávat i dynamicky, pomocí proměnných s řetězcí
  - *struktura.(promenna)*
    - `>> s = 'polomer'`
    - `>> kruh(1).(s)`
- jména položek lze získat příkazem *polozky = fieldnames(struktura)*
  - výstup je cell array se jmény položek
- **struktura vs. pole buněk**
  - dává-li smysl se na data odkazovat jménem položek, použijeme strukturu
  - lze kombinovat, např. položka struktury může obsahovat pole buněk:
    - `>> h %pole buněk se záznamy hodů kostkou`
    - `>> info %text s informacema`
    - `>> vse_s.data = h`
    - `>> vse_s.informace = info`
    - `>> vse_s.data{4}(3) ... vypíše 3. prvek vektoru ve 4. buňce pole buněk obsaženého v položce data, tedy hodnotu 6`
- **Tabulky**
  - *table*
    - datový typ pro snadnou manipulaci se sloupcově orientovanými nebo tabulkovými daty
  - různé sloupce mohou obsahovat různé datové typy
  - typickým použitím pro *table* je držení experimentálních dat, kde sloupce představují různé měřené proměnné a řádky představují jednotlivá pozorování
  - vytváření
    - výčtem prvků po jednotlivých proměnných (= po sloupcích)
      - `T = table(promenna1,..., promennaN)`
    - import z tabulek ve formátu Microsoft Excel a textových souborů
      - v GUI Import vybrat *Table*
      - `T = readtable(jmeno_datoveho_souboru)`
    - dynamická tvorba – přidávání položek k existující tabulce
  - proměnné (= sloupce) jsou pojmenovány
  - řádkům lze přiřadit jména volitelně
  - přístup k datům:
    - pomocí číselných indexů
    - pomocí pojmenování řádků a sloupců

Typ indexování	Výsledek	Syntaxe	Výběr řádků	Výběr proměnné
kulaté závorky	tabulka	<code>T(rows,vars)</code>	jeden nebo více	jedna nebo více
složené závorky	extrahovaná data	<code>T{rows,vars}</code>	jeden nebo více	jedna nebo více
tečková notace	extrahovaná data	<code>T.var</code>	všechny	jedna
tečková notace	extrahovaná data	<code>T.var(rows)</code>	jeden nebo více	jedna

- grafické zobrazení - dvojklik v okně *Workspace*
  - možnosti editace hodnot, řazení řádků

⇒ P04

- otevřít soubor *tabulky.mlx*
- projít skript

**Import tabulky z XLS souboru**

Dvojklik na NejvetsiMestaCR.xls v okně current folder, v nástroji pro import zvolit datový typ table.

**Import tabulky pomocí funkce readtable**

Načtení dat do tabulky T.

```
T = readtable('NejvetsiMestaCR.xls')
```

**Práce s tabulkou**

Výběr podtabulky pomocí indexování kulatými závorkami.

```
T1 = T(1:5, [2 3])
```

**Vytažení dat z tabulky**

Vytažení dat z tabulky do vektoru nebo matice lze několika způsoby.

1) složenými závorkami s číselným indexem řádků i sloupců

```
M1 = T{1:5, 3}
```

2) složenými závorkami s číselným indexem řádků a pojmenováním proměnné (sloupce)

```
M2 = T{1:5, 'PocetObyvatel'}
```

3) tečkovou notací s pojmenováním proměnné - vybere celý sloupec

```
M3 = T.PocetObyvatel
```

4) tečkovou notací s pojmenováním proměnné a číselným indexem řádků

```
M4 = T.PocetObyvatel(1:5)
```

**Práce s položkami**

Výpočty - procentuální velikost populace jednotlivých měst vůči celé ČR.

```
T.PocetObyvatel = T.PocetObyvatel/sum(T.PocetObyvatel)*100
```

Vizualizace.

```
bar(T.PocetObyvatel(1:end-1))
```

**Souhrnné informace o tabulce**

```
summary(T)
```

**Vytvoření tabulky z jednotlivých proměnných**

```
Prijmeni = {'Smith'; 'Johnson'; 'Williams'; 'Jones'; 'Brown'};  
Vek = [38; 43; 38; 40; 49];  
Vyska = [71; 69; 64; 67; 64];  
Vaha = [176; 163; 131; 133; 119];  
KrevniTlak = [124 93; 109 77; 125 83; 117 75; 122 80];
```

Tabulka zadaná z jednotlivých sloupců

```
T2 = table(Prijmeni,Vek,Vyska,Vaha,KrevniTlak)
```

- jména sloupců podle jmen proměnných, ze kterých je vytvořena
- v jedné položce může být i vektor, viz. dvojprvkový vektor KrevniTlak

### Pojmenování řádků

Volitelně mohou být pojmenovány i řádky.

```
T3 = table(Vek,Vyska,Vaha,KrevniTlak,'RowNames',Prijmeni)
```

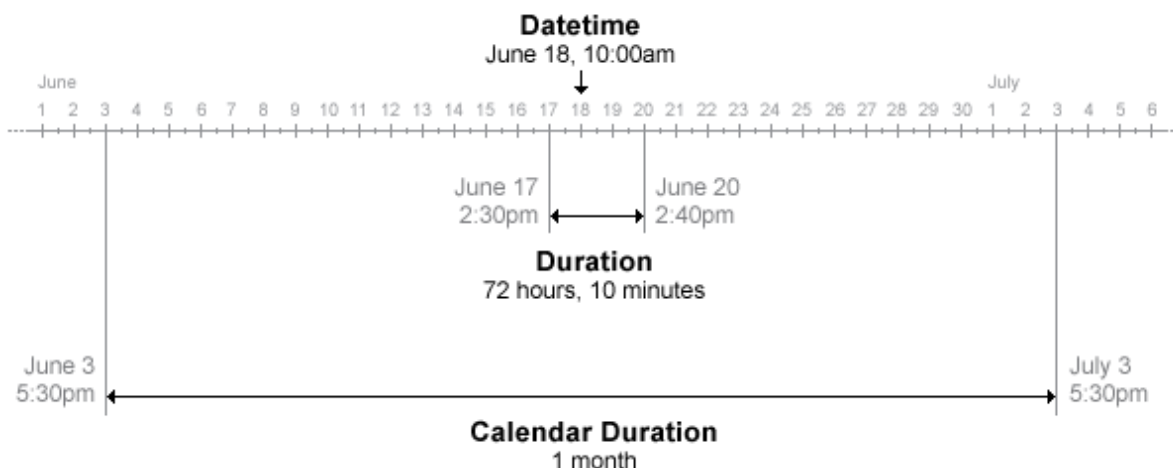
Výběr jednořádkové podtabulky tvořené Jonesovými údaji.

```
T3('Jones',:)
```

### Další možnosti práce s tabulkami

```
doc table
```



- **Datum a čas - základní reprezentace vektorem, číslem nebo řetězcem**
  - vektor 6-ti prvků [*rok měsíc den hodina minuta sekunda*]
    - `>> clock ...` aktuální datum a čas ve formátu vektoru
    - výpočet časového rozdílu
      - *etime*
  - jediná hodnota, desetinné číslo vyjadřující počet dní od 0. ledna 0000
    - `>> now ...` aktuální datum a čas ve formátu čísla
  - datum jako řetězec ve formátu den-měsíc-rok
    - `>> date ...` aktuální datum ve formátu řetězce
  - převodní funkce mezi reprezentacemi
    - *datestr, datenum, datevec*
    - `>> datestr(now) ...` lze nastavit různý formát data a času
- **Datum a čas - pokročilá reprezentace data a času**
  - speciální datové typy
    - *datetime* ... časový okamžik
    - *duration, calendarDuration* ... trvání
  - přesnost na nanosekundy
  - zohledňují časová pásma, letní čas, přestupné roky a přestupné sekundy



- **Datetime**
  - aktuální datum a čas
    - `>> t = datetime`
  - datetime můžeme vytvořit jako jeden časový údaj nebo celé pole časových položek
    - `>> t2 = datetime(2016,9,20,[8 9],0,0)`
  - datetime má možnost zobrazit jednotlivé složky položkami:
    - *Year, Month, Day, Hour, Minute, Second*
    - `>> t.Day`
  - položky lze měnit
    - `>> t.Day = [18 19]`
  - formát zobrazení lze zobrazit i měnit
    - `>> t.Format`
  - datetime definuje časové pásmo
    - položka *TimeZone*
- **Duration**
  - rozdíl dvou proměnných datetime je duration, časové trvání
    - `>> d = t - t2`
  - formát zobrazení lze zobrazit i měnit přes položku
    - *Format*
  - hodnotu duration lze extrahovat v různých jednotkách pomocí funkcí:
    - *years, days, hours, minutes, seconds, milliseconds*
    - `>> days(d)`
    - `>> hours(d)`
  - duration lze vytvářet zadáním hodnoty trvání pomocí funkcí:
    - *years, days, hours, minutes, seconds, milliseconds*
  - novou hodnotu datetime lze spočítat přičtením duration ke stávající hodnotě datetime
    - `>> t2`
    - `>> t3 = t2 + days(2)`
- **CalendarDuration**
  - calendarDuration lze vytvářet zadáním hodnoty trvání pomocí funkcí
    - *calyears, calquarters, calmonths, calweeks, caldays*
  - novou hodnotu datetime lze spočítat přičtením calendarDuration ke stávající hodnotě datetime
- Vizualizace
  - vektor typu *datetime* jako x souřadnice grafu
  - při zoomování automatická změna popisků roky > měsíce > dny ...
- **Časové tabulky**
  - *timetable, readtimetable*
  - od R2016b
  - obdoba *table* s asociovaným časovým údajem ke každému řádku
    - usnadnění operací vázaných na čas, jako je výběr řádků v časovém rozsahu, slučování a synchronizace tabulek, převzorkování, časový posun, ...
  - `>> doc "Create Timetables"`
- **Kategoriální pole**
  - *categorical*
    - datový typ pro data, jejichž hodnoty tvoří uzavřenou množinu prvků – kategorie, např.: dny v týdnu

- alternativa k poli buněk s řetězcí - snazší třídění, řazení, možnost přiřadit hodnotám pořadí, ochrana proti neúmyslnému přidání nové kategorie
- `>> C = {'a', 'b', 'a', 'c'}`
- `>> CT = categorical(C)`
- `>> categories(CT)`
- `>> nnz(CT == 'a')`
- **Řídké matice**
  - *sparse*
  - efektivní využití paměti pro matice, které mají velké množství nulových prvků
    - v paměti jsou uloženy pouze hodnoty a pozice nenulových prvků
    - normální matice (*full*) mají v paměti uloženy všechny hodnoty
  - s řídkými maticemi lze využít
    - aritmetické operace, logické operace, indexování, speciální funkce
  - `>> M = eye(5)`
  - `>> S = sparse(M)`
  - `>> spy(S)`
  - `>> full(S)`

## Funkce v MATLABu

- **Skript vs. funkce**
  - Skript je sekvence po sobě jdoucích příkazů umístěných v souboru
    - *skript.m*
    - volá se samotným jménem
    - do volání nelze předávat hodnoty
    - operuje nad workspace, ze kterého se volá
      - hlavní workspace při volání z *Command Window* nebo tlačítkem *Run*
  - $\Rightarrow$  je nutné umět vytvářet vlastní funkce, které se chovají stejně, jako funkce předdefinované (  $y = \sin(x)$  )
- **Vytvoření funkce**
  - funkce se od skriptu liší klíčovým slovem *function* v prvním řádku
  - vytvoření funkce ze skriptu  $\Rightarrow$  definice hlavičky v prvním řádku
    - *function [vystupni parametry] = jmeno\_funkce(vstupni parametry)*
      - hlavička definuje vstupní a výstupní proměnné
      - výstupní proměnné musí být ve funkci vyčísleny (musí jim být přiřazeny hodnoty)
      - na konci *end* (u samostatné funkce být nemusí)
  - New  >  Function
    - vytvoří prázdný soubor s předdefinovanou strukturou funkce

$\Rightarrow$  Př.: *moje\_funkce.m*

```
function [A,B,C] = moje_funkce(a,b,c)

%MOJE_FUNKCE   Pokusna funkce
%   MOJE_FUNKCE(a,b,c)

x1 = a;
x2 = b;

A = x1+x2;
B = c;
```

```
C = A+B;
```

```
end
```

- **Dokumentace:**

- komentáře od prvního řádku až do prvního vynechaného řádku
  - nápověda, která se zobrazí po zadání `>> doc jmeno_funkce`
- `>> doc moje_funkce`

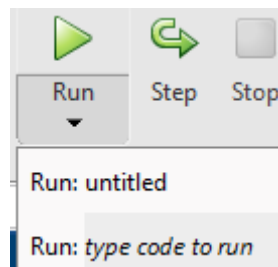
- **Soubor musí být uložen pod stejným jménem, jako je jméno funkce**

- **Funkce má lokální workspace**

- předané vstupní proměnné a proměnné vytvořené uvnitř funkce nejsou viditelné zvenčí a po skončení funkce zanikají
  - `x1, x2` v příkladu jsou vnitřní proměnné, zvenku nedostupné
- uvnitř funkce nejsou vidět proměnné z hlavního workspace
- parametry dovnitř a ven z funkce se předávají pomocí hlavičky
- hlavička definuje jednoznačné rozhraní funkce  $\Rightarrow$  přehlednost

- **Volání funkce z MATLABu**


- stejně jako vestavěné funkce
- `[out1, out2, ...] = jmeno_funkce(in1, in2, ...)`
  - vstupní proměnné se při volání funkce přiřazují v daném pořadí
  - výstupní proměnné se při volání funkce přiřazují v daném pořadí
  - `>> [m1, m2, m3] = moje_funkce(1, 2, 3)`
    - 1 do *a*, 2 do *b*, 3 do *c*
    - *A* do *m1*, *B* do *m2*, *C* do *m3*
- funkce nelze spouštět přímo z editoru stejným způsobem jako skript
  - nemáme zadání vstupních parametrů
  - $\Rightarrow$  z menu lze vyvolat nabídku pro zadání parametrů







- **Code Analyzer**

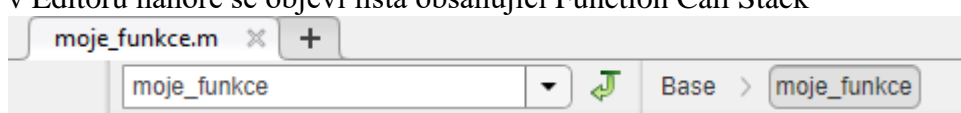
- v pravém horním rohu symbol
  - při psaní bílý
  - syntaktická chyba ... červený
  - varování ... oranžový
  - vše OK ... zelený
- pod symbolem barevné čárky podél okraje funkce = odkazy na místa skriptu s daným problémem
  - klepnutí = umístění kursoru do místa problému
  - pozastavení = info
- v textu funkce je daný problém zvýrazněn podtržením vlnovkou

- **Debugging**

- zadat breakpoint
  - kliknutí na číslo před řádkem funkce přidá breakpoint `6 >` 
    - místo, kde se funkce při vykonávání zastaví



- můžeme jich přidat několik
- zrušíme opětovným kliknutím
- podmíněný *breakpoint* – přes kontextové menu tlačítka *Run*
- **zavolání funkce** (např. z příkazové řádky)
  - ⇒ spustí se a zastaví na prvním breakpointu
  - šipka  u aktuálního řádku funkce
  - v příkazové řádce kurzor v podobě: *K>>*
- v toolstripu se objeví ikony pro procházení skriptem
  - Step  ... po řádcích
  - Continue  ... přejezd na další breakpoint
  - další tlačítka ... skákání dovnitř volaných funkcí a ven
  - Quit Debugging  ... přerušování vykonávání skriptu
    - nebo funkce *K>>dbquit*
- v Editoru nahoře se objeví lišta obsahující Function Call Stack
 



  - Base ... proměnné v základním workspace
  - *moje\_funkce* ... proměnné vnitřního workspace funkce *moje\_funkce*
- v Editoru se při pozastavení nad proměnnou zobrazí její aktuální hodnota
- v průběhu ladění můžeme z příkazové řádky volat příkazy, například měnit hodnoty proměnných a sledovat vliv těchto změn.
- ukončení debugging režimu
  - odstranění všech breakpointů z funkce, *Breakpoints > Clear All*

⇒ Př.: *funkce\_s\_chybou.m*

```
function y = funkce_s_chybou(a,b,c,x)

%FUNKCE_S_CHYBOU je kvadratická funkce zadaná parametry a bodem
% Y = FUNKCE_S_CHYBOU(A,B,C,X) vypočte hodnoty rovnice
% Y = AX^2+BX+C pro hodnoty X. Vykreslí křivku, která spojuje
% body s x-ovými souřadnicemi danými vektorem X a y-ovými
% souřadnicemi danými vektorem Y a přidá název grafu ve tvaru
% 'y = ax^2 + bx + c', kde 'a', 'b' a 'c' jsou dány vstupními
% parametry.
% Příklad: Nalezněte dvě chyby, které FUNKCE_S_CHYBOU obsahuje.
% Po zavolání Y = FUNKCE_S_CHYBOU(5,2,1,1:10) by funkce měla
% vykreslit graf křivky spojující body o souřadnicích [x,y],
% pro x = 1:10 a k nim odpovídající y=5x^2+2x+1, a přidat ke grafu
% název 'y=5x^2+2x+1'.

%% Výpočet
y=a*x^2+b*x+c;

%% Vykreslení grafu
plot(x,y)



%% Přidání názvu grafu
s1 = 'y = ';
s2 = 'x^2 + ';
s3 = 'x + ';
sa = num2str(a);
sb = num2str(b);
```

```
sc = num2str(a);
str = [s1, sa, s2, b, s3, sc];
title(str)

end
```

### Pozn.:

V případě, že má funkce jediný výstup, nemusí být v hlavičce v [ ].

- >> `y = funkce_s_chybou(5,2,1,1:10)`
  - **problém:** v *Command Window* se vypíše chybové hlášení
  - **nápověda:** chybové hlášení
    - ... To perform elementwise matrix powers, use `'.^'`...
    - ...Error in `funkce_s_chybou` (line 14)...
  - **řešení:** na řádku 14 nahradit (^) mocninou po prvcích (`.^`) a uložit
- >> `y = funkce_s_chybou(5,2,1,1:10)` (využít *Command History*: [↑])
  - **problém:** v názvu grafu je místo jednoho čísla čtvereček.
  - **nápověda:** *Code Analyzer* oranžově podtrhl proměnnou `sb`.
    - najet myši na podtržení
    - 'The value assigned to variable 'sb' might be unused'.
  - **řešení:** na řádku 26 místo proměnné 'b' dát 'sb' a uložit
- >> `y = funkce_s_chybou(5,2,1,1:10)`
  - **problém:** v názvu grafu je na konci místo očekávané jedničky pětka.
  - **nápověda:** debugging
    - vložit breakpoint na řádek 26.
    - >> `y = funkce_s_chybou(5,2,1,1:10)`
    - najetím myši zkoumáme hodnoty proměnných
    - proměnná `sc` nabývá hodnotu '5' (char)
    - vybereme ji a označí se nám všechna místa, kde se používá.
    - na řádku 25 je `sc = num2str(a)`
    - změníme hodnotu proměnné `sc`: K>> `sc = num2str(c);`
    -  **Step**, proměnná `str` již nabývá správné hodnoty,  **Continue**
    - **řešení:** opravit řádek 25 na `sc = num2str(c)` a uložit
- **Zpracování chyb**
  - funkce *error*
    - >> `error('Nastala chyba.')`
    - do *Command Window* vypíše chybovou zprávu
    - ukončí funkci – další příkazy se neprovedou
  - funkce *warning*
    - >> `warning('Nezadán vstup. Použita defaultní hodnota.')`
    - do *Command Window* vypíše varování
    - pokračuje ve funkci – další příkazy se provedou
  - blok *try* a *catch*
    - když je příkaz v sekci *try* úspěšný, provede se
    - když se v sekci *try* vyskytne chyba, přejde do sekce *catch*

<pre>try     příkazy; catch     příkazy; end</pre>	<pre>try     příkazy; catch err     příkazy; % pro zpracování err     rethrow(err) end</pre>
--	--

⇒ Př.: >> funkce\_s\_chybou

- vrátí chybu, neboť voláme bez zadání vstupů
- chování v případě chyby změníme pomocí try - catch bloku
- upravíme sekci Výpočet u funkce\_s\_chybou:

```
%% Výpočet
try
    y=a*x.^2+b*x+c;
catch
    error('Něco se pokazilo. Zadal jsi vstupy?')
end

>> funkce_s_chybou
```

### Počet vstupů a výstupů

- **Výstupní proměnné** se při volání funkce přiřazují v daném pořadí
  - >> [m1,m2,m3] = moje\_funkce(1,2,3) ... A do m1, B do m2, C do m
- **Není nutné přiřadit všechny výstupní proměnné**
  - >> m1 = moje\_funkce(1,2,3) ... přiřadí pouze A do m1
  - >> [m1,m2] = moje\_funkce(1,2,3) ... přiřadí A do m1 a B do m2
  - >> moje\_funkce(1,2,3) ... přiřadí pouze A do ans
  - >> 1+moje\_funkce(1,2,3) ... přičte k A hodnotu 1 a výsledek uloží do ans
- **Vynechání zvoleného výstupu** – znak "~"
  - >> [m1,~,m3] = moje\_funkce(1,2,3) ... přiřadí A do m1 a C do m3, B se nepřihadí
    - příklad využití: funkce min (>> doc min) zajímá mne pouze index minima, ne jeho hodnota: [~,I] = min(A)
- **Vstupní proměnné** se při volání funkce přiřazují v daném pořadí
  - >> moje\_funkce(1,2,3) ... 1 do a, 2 do b, 3 do c
  - pokud vstup nezádáme, vstupní proměnná uvnitř funkce nebude vytvořena
    - >> moje\_funkce(1,2) ... 1 do a, 2 do b, c není vytvořeno = neexistuje
- **Počet vstupů a výstupů funkce definovaný v hlavičce**
  - nargin('jmeno\_funkce') ... počet vstupů funkce
  - nargout('jmeno\_funkce') ... počet výstupů funkce
- **Použití uvnitř funkce**
  - nargin ... vrátí počet parametrů, kolik jich bylo při volání funkce zadáno
  - nargout ... počet výstupů, kolik požadují po funkci při jejím volání
  - varargin ... uvede se v hlavičce, pro libovolný počet vstupů při volání funkce
  - varargout ... v hlavičce, pro libovolný počet výstupů při volání funkce
- ⇒ různý počet zadaných vstupů a požadovaných výstupů mohou ošetřit sérií if příkazů

⇒ Př.: kresli.m

Vytvoření funkce, která bude vykreslovat přímku definovanou rovnicí  $y = ax + b$ , kde ošetříme 2 věci: zda uživatel nezadal některé vstupy (`nargin`) a zda uživatel požaduje nějaké výstupy (`nargout`).

```
function [x0,y0] = kresli(x,a,b)

if nargin < 3
    b = 0;
    warning('Neni zadano b: b=0')
end

if nargin < 2
    a = 1;
    warning('Neni zadano a: a=1')
end

if nargin < 1
    error('Neni zadan zadny vstup')
end

y = a*x+b;

if nargout == 0
    plot(x,y,'-o');
else
    x0 = x;
    y0 = y;
end

end
```

- Co vrací funkce

- `>> kresli(0:5,2,3) ... nargout = 0`
- `>> [x,y] = kresli(0:5,2,3) ... nargout = 2`
- `>> kresli(0:5,2) ... nargin = 2`
- `>> kresli(0:5) ... nargin = 1`
- `>> kresli ... nargin = 0`

### Pozn.:

Proč jsou ve funkci `kresli` definovány výstupní proměnné `x0` a `y0` a nepoužijí se rovnou `x` a `y`? Hodnoty `x0` a `y0` nejsou v případě, že nepožadují výstup, vůbec vyčísleny (neexistují). Kdyby byly výstupní proměnné vyčísleny (jako je tomu v případě `x` a `y`), předal by se při volání funkce `kresli(0:5,2,3)` první výstup (`x`) do proměnné `ans`. Použití `x0` a `y0` vytváření `ans` zabrání.

- Použití `varargin` = proměnný počet vstupů funkce
  - `function [vystupni parametry] = jmeno_funkce(varargin)`
  - do proměnné `varargin` se zapíší vstupy ve formátu pole buněk (cell array)
  - na vstupy se odkážeme indexováním do proměnné `varargin`
    - `x(i) = varargin{i};`
    - `x(i) = varargin{i}(1);` ... pro výběr konkrétních hodnot pole

⇒ Př.: `body.m`

Vykreslení různého počtu bodů zadaných uživatelem.

```
function body(varargin)

for k = 1:length(varargin)
    x(k) = varargin{k}(1);
    y(k) = varargin{k}(2);
end

plot(x,y,'r*')

end
```

- >> A = [6,5];
- >> B = [5,8];
- >> C = [6,9];
- >> D = [0,4];
- >> body(A,B,C,D)

- *varargin* lze použít i jako dodatečný argument za standardními vstupy pro předání hodnot doplňujících parametrů
  - *function [vystupni parametry] = jmeno\_funkce(vstupni parametry, varargin)*
  - např. když bychom měly funkci danou předpisem *function fun(a,b, varargin)* a zavolali ji příkazem *fun(1,2,3,4,5)*, tak se předá 1 do *a*, 2 do *b* a hodnoty 3,4,5 jako pole buněk do proměnné *varargin*
- Při opačném problému – neznámém počtu výstupů – lze použít proměnnou *varargout*
  - *function [varargout] = jmeno\_funkce(vstupni parametry)*

## Validace vstupů

- Blok *arguments*
  - omezení na vstupní parametry
  - možnost definovat defaultní hodnoty vstupních parametrů
  - v případě použití se musí uvést před samotný kód funkce

```
function myFunction(inputArg)
    arguments
        inputArg (dim1,dim2,...) ClassName {fcn1,fcn2,...} = defaultValue
    end
    % Function code
end
```

Size
Class
Functions

⇒ Př.: kresli2.m

```
function [x0,y0] = kresli2(x,a,b)

arguments
    x (1,:) double      % (1,:) přijímá řádkové i sloupcové vektory
    a (1,1) {mustBeNumeric} = 1
    b (1,1) {mustBeNumeric} = 0
end

y = a*x+b;

if nargin == 0
    plot(x,y,'-o');
else
    x0 = x;
    y0 = y;
end

end
```

- pro validaci lze využít předdefinované validační funkce i definovat vlastní validační funkce
- umožňuje definovat opakující se vstupy i vstupy typu „Name-Value“ (viz. dokumentace)
- od verze R2019b
- alternativně lze použít např. funkci *validateattributes*

## Rozsah proměnných

- Po konci funkce se lokální proměnná ztratí a její hodnota se zapomene
- **Persistentní proměnné**
  - chci-li zapamatovat hodnotu proměnných do příštího volání funkce
  - možnosti použití:
    - držení hodnoty, kterou stačí zjistit jednou, např. typ OS
    - akumulace, např. „kolikrát jsem funkci volal?“
  - obdoba lokální statické proměnné v jazyce C
  - definice persistentní proměnné
    - *persistent seznam\_proměnných*

⇒ Př.: stopky.m

```
function stopky

persistent CAS

if isempty(CAS)
    CAS = datetime;
    disp(['start v ', datestr(CAS)])
else
    t = datetime - CAS;
    disp(t)
end

end
```

- >> stopky
- >> stopky
- >> clear stopky
- použitá funkce
  - *isempty* ... je proměnná prázdná? 1 = ano, 0 = ne

- při prvním zavolání funkce *stopky* se v paměti vytvoří obraz funkce s jejími persistentními proměnnými. Příkaz *clear stopky* toto smaže.

Pozn.:

Funkce nemusí mít ani vstupní ani výstupní parametry.

- **Globální proměnné**
  - definice globální proměnné
    - *global seznam\_proměnných*
  - tyto proměnné se berou globálně tam, kde je na začátku klíčové slovo *global* se seznamem proměnných, včetně Command Window
  - globální proměnné mají vlastní workspace oddělený od hlavního i všech lokálních
    - kdekoliv proměnnou deklaruji jako globální, tam můžu zobrazit i měnit její hodnotu uloženou v tomto „globálním workspace“
  - **upozornění:**
    - Využívání globálních proměnných se z důvodů spolehlivosti a robustnosti programů nedoporučuje.
- **Automatické vytváření příkazů**
  - využití funkce *eval*
    - *eval('řetězec')* ... vyhodnotí řetězec, jako by byl zadán do Command Window
    - *eval('1+1')*
  - **upozornění:**
    - Využívání funkce *eval* se z důvodů spolehlivosti a robustnosti programů nedoporučuje.
    - Alternativy k funkci *eval*: `>> doc "Alternatives to the eval Function"`

Lokální funkce (subfunkce)

- Soubor *f1.m*

```
function f1
    příkazy;
end
function f2
    příkazy;
end
function f3
    příkazy;
end
```

- *f1* = hlavní funkce
- *f2* a *f3* = lokální funkce
  - nejsou viditelné z vnějšku
  - lze je volat jen v hlavní funkci (*f1*) a lokálních funkcích definovaných ve stejném souboru (*f2*, *f3*)
  - každá lokální funkce má svůj lokální workspace

⇒ Př.: *stat.m*

Funkce stat bude počítat aritmetický a geometrický průměr. Máme 1 hlavní funkci a 2 lokální funkce.

```
function [ap, gp] = stat(x)
n = length(x);
ap = aprumer(x, n);
gp = gprumer(x, n);
end

function ya = aprumer(xa, na)
ya = sum(xa)/na;
end

function yg = gprumer(xg, ng)
yg = prod(xg)^(1/ng);
end
```

- >> v = rand(1,100);
- >> [AP, GP] = stat(v)

- **Debugging**

- když dám breakpoint do lokální funkce, můžu si po zavolání prohlédnout *Function Call Stack* pro Base, hlavní funkci i danou lokální funkci

## Vnořené funkce

- **"nested functions"**
- Vnořené funkce jsou alternativou k lokálním funkcím, ale vykazují několik odlišností
- Viditelnost proměnných
  - hlavní rozdíl oproti lokálním funkcím
  - ve vnořených funkcích jsou vidět proměnné z vnější funkce a lze jim přímo přiřazovat hodnoty
    - Př.: x definováno ve vnější funkci, viditelné a měnitelné ve vnitřní
  - ve vnější funkci jsou vidět proměnné z vnořených funkcí, ale musí být definovány i ve vnější funkci
    - Př.: c definováno ve vnitřní funkci, viditelné ve vnější
  - v editoru barevně odlišeny – [zelenomodře](#)

⇒ Př.: fvn.m            % všímáme si barevného vybarvování jednotlivých proměnných

```
function fvn
x = 5;
vnorena(1)
disp(x)
vnorena(2)
disp(x)

function vnorena(a)
```



```

        x = x+a;
    end
end

```

- *fyn* je vnější funkce
- *vnorena* je vnořená funkce
- Vnořené funkce na stejné úrovni
  - mezi sebou navzájem workspace nesdílí
- **Vstupní a výstupní parametry vnořených funkcí**
  - vstupní parametry vnořené funkce jsou její lokální proměnné (nejsou viditelné zvenčí)
  - výstupní proměnné jsou pro vnořenou funkci lokální  $\Rightarrow$  musí se předat
- Funkce mohou být vnořeny ve více úrovních
- `>> doc nested functions`

## Function handle


- Existují funkce, do kterých je potřeba předat funkci jako jeden z parametrů
  - *fplot*, *fminsearch*, řešiče *ode\** a podobně
  - předávaná funkce se nemá v hlavičce vyčíslit, ale předat "dovnitř"
- $\Rightarrow$  předání odkazu na funkci pomocí *function handle* - znak @
  - `>> sin(pi/4)`
  - `>> hs = @sin`
  - `>> hs(pi/4)` ... zavolání funkce pomocí *function handle*
  - `>> fplot(hs,[0 pi])`
    - graf funkce předané v *hs* v mezích od 0 do  $\pi$
    - lze zadat přímo: `fplot(@sin,[0 pi])`
- **Více funkcí sloučíme do pole buněk**
  - `>> fun = {@sin, @cos, @log}`
  - `>> fplot(fun{2},[0 pi])`


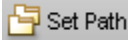
## Handle anonymní funkce

- *function handle* nemusí pouze odkazovat na existující funkci ale může i definovat funkci novou  $\Rightarrow$  anonymní funkce
- **Anonymní funkce je funkce, která existuje pouze v paměti**
  - `>> H = @(x,y) x.^2 + y.^2;`
  - `>> a = 1:5;`
  - `>> b = 2:6;`
  - `>> H(a,b)`
- **Anonymní funkce pevně zafixuje všechny konstanty v době vytvoření** – hodnoty mění pouze vstupní proměnné
  - `>> k = 100;`
  - `>> H1 = @(x,y) x.^2 + y.^2 + k;`
  - `>> k = 200;`
  - `>> H2 = @(x,y) x.^2 + y.^2 + k;`
  - `>> H1(1,1)`
  - `>> H2(1,1)`
- Informace o anonymní funkci – *functions(function handle)*
  - `>> info = functions(H1)`
- **Použití anonymních funkcí**
  - rychlé vytvoření a použití funkce bez nutnosti ukládat ji jako soubor
  - "předeklarace" pořadí, formy nebo počtu vstupů existujících funkcí

- např. úloha minimalizace – *fminsearch*
  - `>> edit matmodel`
  - `>> matmodel(1,1,0,0.5)`
    - pro hledání minima z hlediska bodu  $x$  nelze takto předat, funkce musí mít jediný vstupní parametr  $x$
- `>> m = @(x) matmodel(1,1,0,x) ... a,b,c je zafixováno`
- `>> m(0.5)`
- `>> fplot(m, [-1,1]); grid`
- `>> fminsearch(m,0)`
- úloha pro curve fitting by byla opačná
  - zafixujeme zadané  $x$  a  $y$  a hledáme parametry
  - definujeme integrální kritérium
- změna z více skalárních parametrů na vektorový vstup a naopak
  - máme funkci  $f1$ , která má dva vstupy a voláme ji  $f1(1,2)$
  - vytvoříme anonymní funkci  $f2$  s jedním (vektorovým) vstupem
    - $f2 = @(c) f1(c(1),c(2))$
  - voláme ji  $f2([1 \ 2])$

## Cesta v MATLABu

- **MATLAB vidí aktuální složku (Current Folder)**
  - cesta k aktuální složce příkazem *pwd*
    - `>> pwd`
  - lze změnit graficky nebo příkazem `cd('složka')`
  - lze nastavit tzv. *spouštěcí složku MATLABu*, neboli co bude *Current Folder* po spuštění programu MATLAB (*Initial Working Folder*). Dva způsoby:
    - *Preferences > General > Initial Working Folder*
    - ve spouštěcí ikoně MATLABu v položce *Spustit v*
  - ALE – časté přecházení mezi složkami je nepohodlné
- **Path** = seznam cest, které MATLAB prohledává při hledání funkcí
  - `>> path ...` vypíše seznam cest
  -  `Set Path` grafické okno s výpisem cest
- Co dělá MATLAB při zadání textu do příkazové řádky
  - 1. Je to proměnná?
  - 2. Je to skript či funkce v aktuální složce
  - 3. Je to skript či funkce ve složkách, které jsou v PATH?
    - cesty prohledává shora dolů
- **Přidání složky do cest**
  - Při spuštění se vždy přidá do cest adresář zobrazitelný a nastavitelný pomocí funkce *userpath*:
    - `>> userpath`
  - Funkce *addpath*
    - `addpath('složka')`
      - `>> addpath('H:\MATLAB2')`
        - přidání aktuální složky do cest:
          - `>> addpath(pwd)`
      - pamatuje se jen po dobu spuštění MATLABu
      - lze volat na začátku hlavního skriptu nebo vytvořit inicializační skript
        - přidání cest pro daný výpočet, pro daný projekt, ...
      - lze zadat do souboru *startup.m*

- uživatelem vytvořený soubor
- je-li ve spouštěcí složce MATLABu (případně ve složce uložené v cestách, např. v *userpath*), tak se automaticky spustí při startu MATLABu
  - ⇒ použití *addpath* v *startup.m* je nezávislé na verzi MATLABu
- přidání složky včetně podsložek využitím funkce *genpath('složka')*
  - *addpath(genpath('složka'))*
- GUI
  - tlačítko  > tl. Add Folder
    - totéž co zavolání příkazu *addpath*
    - pamatuje se jen po dobu spuštění MATLABu
  - tlačítko  > tl. Add Folder > Save
    - cesta se uloží k ostatním cestám v souboru *pathdef.m*
  - nevýhody:
    - soubor je umístěn v instalační složce MATLABu ⇒ musím mít právo zápisu (nebo jej lze umístit do spouštěcí složky)
    - ! v souboru *pathdef.m* jsou cesty do instalační složky dané verze MATLABu ⇒ obsah je svázán s danou verzí ⇒ s novou verzí MATLABu je nový *pathdef.m* !
- Existuje daná funkce a kde je?
  - *which jmeno\_funkce* ... vypíše první složku, kde funkci najde
    - `>> which moje_funkce`
    - `>> which plot`
  - *which jmeno\_funkce -all* ... všechny složky, kde funkci najde
    - `>> which plot -all`
  - *exist jmeno* ... vrátí číslo, kde 0 = neexistuje, 1 = ... viz. doc exist
    - `>> exist moje_funkce`

⇒ **Pozor:**

Funkce se stejným jménem se mohou překrýt – použije první nalezenou

## Projekty

- MATLAB Projects / Simulink Projects
  - pro rozsáhlé úlohy a aplikace
    - práce v týmu
    - soubory ve vícero adresářích
  - nastavení a správa cest napříč týmy
  - grafická analýza závislostí a kontrola vyžadovaných souborů
  - možnost vytvářet odkazy na často používané soubory
  - sledování a správa změn v souborech
    - vestavěná podpora verzovacích systémů Git a Subversion (SVN)
- projekt sdružuje více souborů pod společnou správu
  - projekty jsou společné pro MATLAB i Simulink
- založení projektu
  - *Home > New > Project*
  - lze založit
    - prázdný projekt, do kterého se postupně přidají soubory
    - projekt založený na složce – zahrne všechny soubory z vybrané složky do projektu

- projekt založený na verzovacím systému Git/SVN
  - spustí se průvodce založením projektu
- v projektu lze nastavit:
  - *Project Path*
    - cesty ke složkám, které se automaticky přidají do cest MATLAB při otevírání projektu
  - *Startup Shutdown* soubory
    - zvolené skripty se automaticky spustí při otevírání / zavírání projektu
    - dále lze automaticky načíst MAT soubory, otevřít modely, ...
- přidání a odebrání souborů
  - soubory lze přidávat do projektu / odebírat z projektu pomocí kontextového menu
- soubory v projektu lze označovat štítky – *classification*
- analýza závislostí
  - tlačítko *Dependency Analyzer*
- uživatelské odkazy
  - záložka *Project Shortcuts*
  - odkazy na modely, skripty a funkce pro jejich rychlé otevření
- integrace s verzovacími systémy
  - podpora systémů Git a SVN
  - tlačítko *Use Source Control* > průvodce integrací s verzovacím systémem
- referencované projekty
  - velké projekty lze rozdělit do více částí a ty mezi sebou referencovat

### Úloha:

- vytvořit projekt ze složky mujProjekt: přejít do složky > New > Project > From Folder
- projít průvodce nastavením projektu
- projít nástroje a možnosti projektu

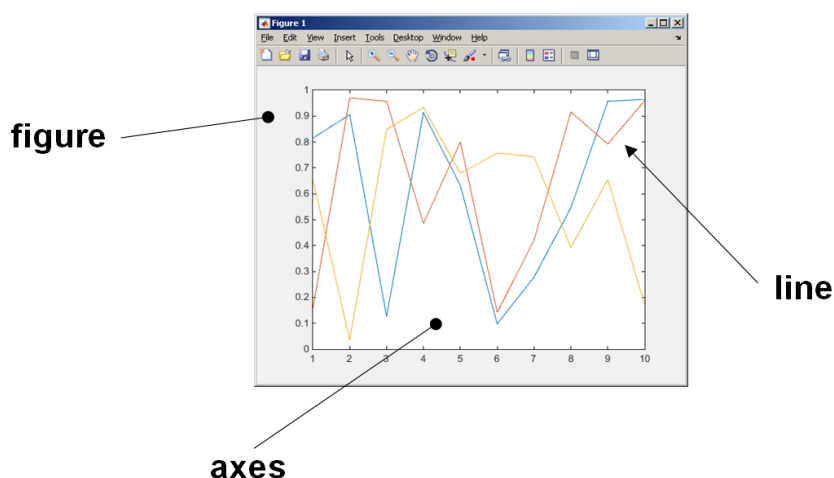
### **Pokročilá práce s grafikou**

- Základy práce s grafikou v MATLABu (viz. školení MATLAB I)
  - grafy se kreslí do oken *Figure*
    - interaktivně
      - označit data v okně *Workspace*
      - vybrat vhodný graf z galerie grafů v záložce *PLOTS*
    - pomocí příkazů
  - základní funkce pro vykreslení grafu: *plot*
    - *plot(x,y)*
    - *plot(x,y,'--rx')* ... s přiřazením barvy, typu čáry a značky
  - základní popisky grafu
    - *xlabel, ylabel, title, legend*
  - mřížka
    - *grid on, grid off*
  - více grafů v jednom souřadném systému
    - *hold on, hold off*
  - více souřadných systémů v jednom okně figure
    - *subplot*
  - interaktivní úprava vlastností grafu
    - menu a panel nástrojů v okně figure

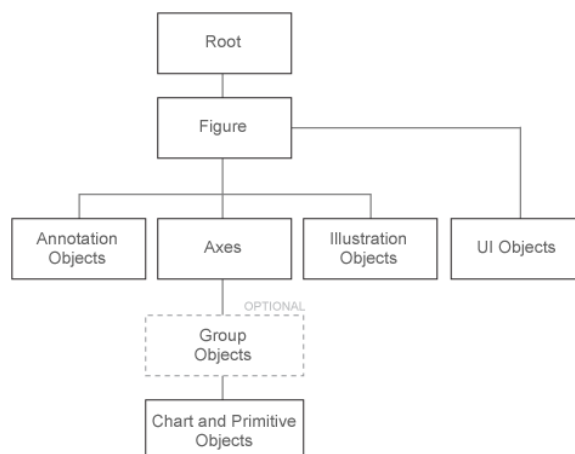
- nástroje *Property Inspector*



- další funkce pro 2-D grafy
  - *scatter, stairs, stem, area, bar, pie, polar, compass, ...*
  - viz. galerie grafů
- základní funkce pro 3-D grafy
  - *plot3, mesh, surf, contour*
- Pro pokročilejší práci s grafikou je potřeba začít pracovat s grafy jako objekty
- **Grafické objekty v MATLABu**
  - každý grafický prvek v MATLABu je objekt



- hierarchie grafických objektů



- `>> load bode_data`
- `>> surf(w,ksi,magdb)`
  - při zadání příkazu pro vykreslení grafu (*Plot Object*) se automaticky vytvoří i axes a figure
- Grafické objekty mají vlastnosti a jejich hodnoty určují vzhled objektu
  - vlastnosti lze editovat pomocí nástroje *Property Inspector*



⇒ zobrazení všech vlastností vybraného grafického prvku

- **vlastnosti lze editovat pomocí příkazů**
- Na každý objekt odkazuje jeho handle
  - handle můžeme uložit do proměnné, pomocí které pak na grafický objekt odkazujeme
- **Získání handle**
  - při tvorbě grafického prvku
    - `h = plot(x,y) ...` handle v proměnné `h` bude odkazovat na čáru grafu
  - dotazem
    - `gcf, gca ...` handle na aktuální figure, axes
    - `findobj ...` hledání grafických objektů podle vlastností
- **Od verze R2014b je handle objekt**
  - **přístup k vlastnostem lze i tečkovou notací**
  - **doplňování jmen vlastností klávesou `Tab`**
- Získání hodnoty vlastnosti
  - `Hodnota = get(handle, 'Vlastnost');`
  - `Hodnota = handle.Vlastnost;` ... od R2014b
- Změna hodnoty vlastnosti
  - `set(handle, 'Vlastnost', Hodnota);`
  - `handle.Vlastnost = Hodnota;` ... od R2014b
- Další možnosti zadání příkazů `set` a `get`:
  - `get(handle) ...` bez zadání vlastnosti vypíše všechny vlastnosti s hodnotami
    - `>> handle ⇔ Show all properties ...` od R2014b
  - `set(handle, 'Vlastnost')`
    - vypíše výčet možností zadávaných hodnot (např.: `'on', 'off'`)
    - pouze pro vlastnosti, kde hodnoty tvoří uzavřenou množinu možností
  - `set(handle) ...` jako předchozí, ale vypíše možnosti pro všechny vlastnosti

#### Pozn.:

Dvojice *vlastnost-hodnota* je možné často předávat již při vytváření grafického objektu ve formě dodatečných parametrů – `plot(x,y,'LineWidth',2)`. V praxi většinou kombinujeme obě možnosti, tedy některé vlastnosti nastavujeme při vytváření grafického objektu, jiné pak dodatečně změnou vlastností (`get`, `set`, `.`).

#### Pozn.:

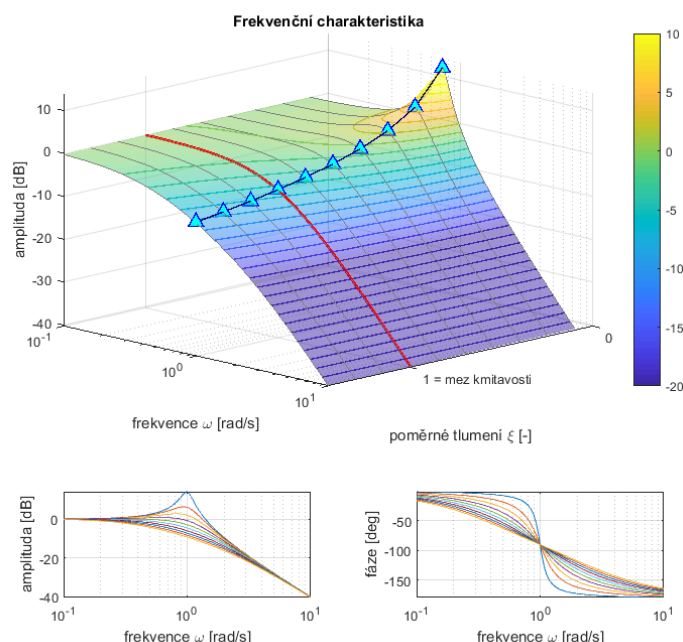
Funkcí `set` lze jedním voláním nastavit více vlastností a to i pro více objektů naráz (*doc set*).

- **Root = obrazovka**
  - handle
    - odkaz na obrazovku hodnotou 0
    - funkce `groot ...` od R2014b
  - získání vlastností prvku root
    - `>> get(0)`
    - `>> set(0)`
  - `ScreenSize`
    - velikost obrazovky ... [1 1 1280 1024]
- **Figure**
  - vytvoření nového prázdného grafického okna
    - `>> hf = figure`
    - handle (od R2014b položka `handle.Number`) grafických oken figure je celé číslo zobrazené v záhlaví okna

- opakované zavolání příkazu figure s hodnotou handle udělá dané okno aktuální
  - `>> figure(hf)`
- zjištění handle na aktuální figure
  - `>> gcf`
- vlastnosti okna figure
  - `>> get(hf)` ... pozice, velikost, nastavení barev, titulek, ...
- **Axes**
  - vytvoření prázdného objektu typu axes
    - `>> ha = axes`
  - zjištění handle aktuální axes
    - `gca`
  - opakované zavolání příkazu axes s hodnotou handle nastaví daný axes jako aktuální
    - `axes(handle)`
  - vlastnosti axes
    - `>> get(ha)` ... pozice, nastavení os, mřížky, pohled, barvy, fonty
- **Graf**
  - vykreslení grafu a předání jeho handle
    - `>> hs = surf(w,ksi,magdb)`
  - vlastnosti grafu
    - `>> get(hs)` ... barvy, tloušťky a styl čar, značky, stínování, ...
- Po vykreslení grafu často následují příkazy pro nastavení vlastností figure, axes a grafů pro dosažení požadovaného výsledku
- Použití příkazů je výhodné zejména v případě, že chceme daný graf vykreslovat opakovaně s různými daty

⇒ Př.: graf.m (otevřít)

Graf frekvenčních charakteristik



- **Popisky grafu**
  - základní popis os
    - `>> xlabel('frekvence \omega [rad/s]')`
    - `>> ylabel('poměrné tlumení \xi [-]')`
    - `>> zlabel('amplituda [dB]')`

- `>> title('Frekvenční charakteristika')`
- přidává do figure textové objekty
  - vlastnost *Interpreter*: *none*, *tex*, *latex* ... pro řeckou abecedu, ...
- **Osy**
  - změna osy X na logaritmickou
    - `>> set(ha, 'XScale', 'log')`
  - změna směru osy Y
    - `>> set(ha, 'Ydir', 'reverse')`
  - pozice značek - aby se neměnil jejich počet a byly pevně fixovány
    - `>> set(ha, 'YTick', [0 1])`
    - zadáním se změní *YTickMode* z *'auto'* na *'manual'* => nebude se již automaticky měnit (přizpůsobovat velikosti okna, ...)
  - popis
    - `>> set(ha, 'YTickLabel', {'0', '1 = mez kmitavosti'})`
    - zadáním se změní *YTickLabelMode* z *'auto'* na *'manual'* => nebude se již automaticky měnit
    - alternativně lze změnit pouze vybrané
      - `ha.YTickLabel(2) = {'text'};`
      - nebo `texty = get(...) > upravit texty > set(...)`
  - číselné nastavení rozsahu jednotlivých os
    - `>> set(ha, 'YLim', [0 max(ksi)])`
      - totéž lze příkazem: `ylim([ymin ymax])`
    - zadáním se změní *YLimMode* z *'auto'* na *'manual'* => nebude se již automaticky měnit (přizpůsobovat velikosti okna, ...)
  - příkaz *axis* ...
    - mění nastavení rozsahu os pohromadě
      - `axis([xmin xmax ymin ymax zmin zmax])`
    - přednastavené volby pro různé úpravy zobrazení os
      - *tight*, *fill*, *ij*, *xy*, *equal*, *image*, *square*, *vis3d*, *normal*, *off*, *on*

### Pozn.:

2-D graf se dvěma osami Y na levé a pravé straně, každá s jinými popisky a měřítkem: *plotyy*

- **Mřížky**
  - *grid on*, *grid off* ... zapnutí/vypnutí mřížky pro všechny osy
  - vlastnosti *XGrid*, *YGrid*, ... zobrazení mřížky pro jednotlivé osy zvlášť
  - styl čar zobrazené mřížky
    - `>> set(ha, 'GridLineStyle', '-')`
  - osa X má zobrazen minor grid, který se zapnul automaticky při přepnutí do logaritmického měřítka. Minor grid lze zapínat pro jednotlivé osy samostatně (*XMinorGrid*, ...) a lze měnit styl čar (*MinorGridLineStyle*)
- **Pohled**
  - zjištění souřadnic aktuálního pohledu
    - `>> [az, el] = view`
  - nastavení pohledu
    - `>> view([0 0])`
    - `>> view([45 0])`
    - `>> view([45 20])`
- **Barvy**
  - barevná stupnice
    - `>> colorbar`



- barevná mapa: *colormap hsv, colormap hot, ...* výchozí je *colormap jet*
- změna přiřazení barevné mapy k hodnotám z-souřadnic
  - `>> set(ha, 'CLim', [-20 10])`
    - totéž lze příkazem: *caxis([cmin cmax])*
- interpolované barvy
  - `>> set(hs, 'FaceColor', 'interp')`
    - možnosti: *flat, interp, texturemap, none*
    - totéž lze příkazem: *shading interp*
- změna zobrazení sítě grafu
  - `>> set(hs, 'EdgeColor', [0.5 0.5 0.5])` ... barva čar
  - `>> set(hs, 'MeshStyle', 'row')` ... lze též *'column', 'both'*
- průhlednost
  - `>> set(hs, 'FaceAlpha', 0.5)`
- nasvícení a odlesky
  - *light, camlight*
  - vlastnosti *FaceLighting, EdgeLighting*
    - možnosti: *flat, gouraud, none*
    - totéž lze příkazem: *lighting gouraud (flat, none)*
- **Přidání dalších grafů**
  - zamezení odstranění stávajícího grafu při vykreslení nového
    - `>> hold on`
  - graf 3-D kontur
    - `>> [~,hc] = contour3(w,ksi,magdb,30,'LineWidth',1);`
  - zvýraznění čar pro  $\xi = 0$  (pozice 7) a  $\omega = \omega_n$  (pozice 50)
    - `>> hp1 = plot3(w,ones(size(w)),magdb(7,:), 'r','LineWidth',2);`
    - `>> hp2 = plot3(wn*ones(size(ksi)),ksi,magdb(:,50),...`
    - `'Color',[0 0 0.5],...`
    - `'LineStyle','-',...`
    - `'LineWidth',1,...`
    - `'Marker','^',...`
    - `'MarkerEdgeColor',[0 0 1],...`
    - `'MarkerFaceColor',[0 1 1],...`
    - `'MarkerSize',10);`

### Pozn.:

Hledání pozice příkazem `idx = find(w == wn)` není pro datový typ *double* vhodné, lepší je použít `[~,idx] = min(abs(w - wn))` nebo `idx = find(abs(w - wn) < tol)`.

- **Více axesů v jednom figure**
  - příkaz *subplot* ... umožní jen rovnoměrný rastr  $n \times m$  axesů
  - přesnější zadání polohy a rozměrů: objekt *axes* + vlastnost *Position*
  - úprava polohy stávajícího axesu
    - `>> set(ha, 'Position', [0.1 0.4 0.75 0.5])`
    - rozměry  $[x0\ y0\ \Delta x\ \Delta y]$  od levého dolního rohu okna figure
    - jednotky normalizované  $\langle 0;1 \rangle$ , tj. poměrné k velikosti okna figure  $\Rightarrow$  při změně velikosti okna figure se mění i velikost axesu
      - dáno vlastností axesu *Units = normalized*
      - možné změnit na *centimeters, pixels, ...*
  - 2 nové objekty typu axes s grafy
    - `>> ha2 = axes('Position', [0.1 0.1 0.35 0.15]);`
    - `>> semilogx(w,magdb)`
    - `>> grid`

- `>> axis tight`
- `>> xlabel('frekvence \omega [rad/s]')`
- `>> ylabel('amplituda [dB]')`
  
- `>> ha3 = axes('Position',[0.6 0.1 0.35 0.15]);`
- `>> semilogx(w,phase)`
- `>> grid`
- `>> axis tight`
- `>> xlabel('frekvence \omega [rad/s]')`
- `>> ylabel('fáze [deg]')`
- svázání zoom dvou axesů
  - `>> linkaxes([ha2 ha3], 'x')` ... svázání zobrazení rozsahu osy X
  - projeví se při následném použití lupy

### Pozn.:

Svázání libovolných vlastností dvou objektů funkcí *linkprop*.

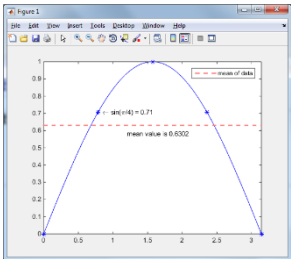
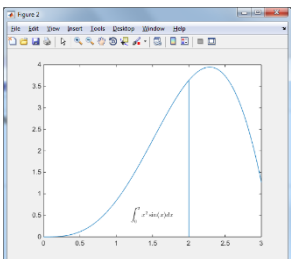
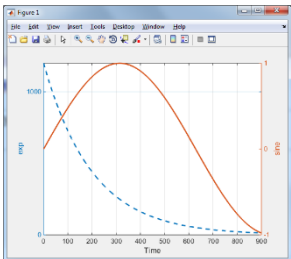
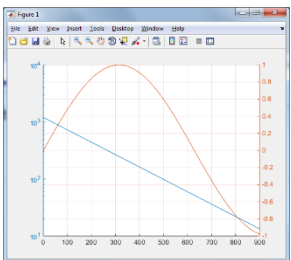
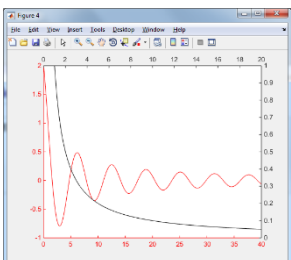
- **Poloha celého okna figure**
  - `>>set(hf,'Position',[50 100 700 700]);`
    - rozměry [x0 y0 Δx Δy] od levého dolního rohu okna obrazovky
    - jednotky v pixelech ... lze změnit vlastností *Units*
- **Nastavení defaultních vlastností objektů**
  - pro uživatelsky nastavitelné parametry
  - u předka daného objektu pomocí textového řetězce složeného ze tří částí:
    - 'default'
    - ObjectType (např. 'Line')
    - PropertyName (např. 'LineWidth')
    - `>> set(groot,'defaultLineLineWidth',2)`
- **Smazání objektu**
  - `delete(handle)` ... smaže objekt
- **Kopírování objektu**
  - `copyobj(co,kam)` ... kopíruje objekt
    - přiřazení existujícího handle jiné proměnné (ha2 = ha) pouze zkopíruje handle, nikoliv objekt
- **Vlastnosti, které mají všechny objekty: akce**
  - *CreateFcn* ... při vytváření objektu
  - *DeleteFcn* ... při zavírání objektu
  - *ButtonDownFcn* ... při klepnutí myší
    - můžu přiřadit funkci, která se při akci vykoná

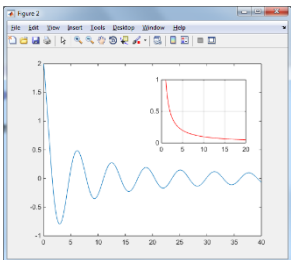
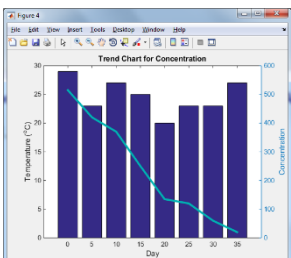
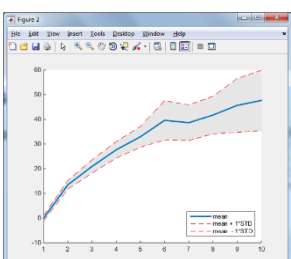
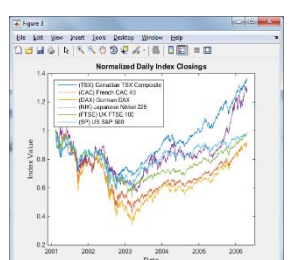
### Pozn.:

Objekty UI Control, UI Menu, UI Context Menu

- vlastnost *'Callback'* = funkce ~ funkce při použití prvku
  
- **Viditelnost handle**
  - $\Rightarrow$  pokud nevidím objekt, může být skrytý
  - například objekty s textem (*xlabel*, *title*, ...)
  - funkce *findobj* skryté objekty nenajde
  - zobrazení: `set(0,'ShowHiddenHandles','on')`
- **Rodiče a potomci**
  - vlastnost *Parent* ... handle na rodiče daného objektu

- vlastnost *Children* ... handle na objekty, které se v daném objektu nacházejí, např. handle na okna figure v rootu, handle na axesy ve figure, atd ...

<p>Ukázky dalších (zajímavých) 2-D grafů a jak jich docílit</p> <p>⇒ <a href="#">Př.: grafy2D.mlx</a> (otevřít)</p>		
Graf se zvýrazněnými body, čarou a textem		<p><i>plot</i> <i>hold on</i></p> <p><i>line</i> <i>text</i></p>
Zobrazení rovnice v grafu		<p><i>text</i> <i>\$\$LaTeX\$\$</i></p>
Graf se dvěma osami Y		<p><i>plotyy(x,y1,x,y2)</i></p> <p>nebo</p> <p><i>yyaxis left</i> <i>plot(x,y1)</i> <i>yyaxis right</i> <i>plot(x,y2)</i></p>
Graf se dvěma osami Y – odlišný typ grafů		<p><i>plotyy(x,y1,x,y2,'semilogy','plot')</i></p> <p>(nebo <i>yyaxis</i>)</p>
Graf s více osami X a Y		<p>Použít více objektů <i>Axes</i> přes sebe</p>

Malý graf ve velkém		Použít více objektů Axes přes sebe
Kombinace sloupcového a čárového grafu – různé osy Y		<code>plotyy(x1,y1,x2,y2,'bar','plot')</code> (nebo <code>yyaxis</code> )
Barevné zvýraznění intervalu		<code>patch</code> <code>plot</code>  <code>hold on</code>
Časová osa		<code>datetime</code> <code>duration</code> , <code>calendarDuration</code>

### Pozn.:

MATLAB Plot Gallery:

<https://www.mathworks.com/products/matlab/plot-gallery.html>

### Pozn.:

Grafické prvky se nepřekreslují stále. Např. pokud měníme grafické prvky v cyklu `for`, jsou překresleny až po dokončení cyklu. Překreslení lze vynutit funkcí `drawnow`. Překreslování však zpomaluje průběh programu – je třeba používat rozumně.

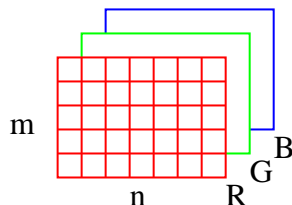
## Rastrová grafika v MATLABu

- Načítání a ukládání obrázků
  - je možné načítat různé typy obrázků – zde např. `.jpg`, `.gif`
  - `imread('jmeno_souboru')`
    - `>> A = imread('Dum.jpg');`

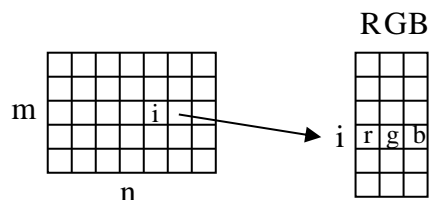
- obrázek = číselné pole
- `imwrite` ... zápis
- `image, imshow` ... zobrazení
  - `>> image(A)`

- **Formáty obrázků**

- RGB



- 3 matice  $m \times n$  pro složky R, G a B
  - `uint8 = 0 .. 255`  $\Rightarrow$  True Color – 24-bit barvy
  - normalizovaný  $\sim$  `double = 0 .. 1`
- Indexový




- 2 matice
  - matice  $m \times n$  hodnot indexů  $i$
  - matice `colormap` = přiřazení RGB hodnot indexům  $i$  i-tým řádkem colormapy
    - `>> [B,map] = imread('Dum.gif');`
    - `>> image(B)`
  - zobrazení obrázku s výchozí barevnou mapou přiřazenou k oknu figure
    - `>> colormap(map)`
- Intenzitní (šedotónový)
  - jediná matice  $m \times n$
  - datový typ
    - `uint8 = 0 .. 255`
    - normalizovaný  $\sim$  `double = 0 .. 1`
    - (lze pracovat v `uint16`)
  - `>> I = rgb2gray(A);`
  - `>> imshow(I)`
- Černo-bílý (BW)
  - jediná matice  $m \times n$ 
    - datový typ `logical = 0 a 1`
  - `>> M = I > mean(I(:))`
  - `>> imshow(M)`

## Grafické uživatelské rozhraní (GUI)

- GUI vytváříme v oknech figure/uitable
- **Při tvorbě GUI můžeme využít:**
  - Grafické ovládací prvky a indikátory
  - Menu a panely nástrojů

- Předdefinovaná dialogová okna
- **Tvorba GUI**
  - Graficky – nástroj *App Designer* (nebo starší nástroj *GUIDE*)
  - Programově

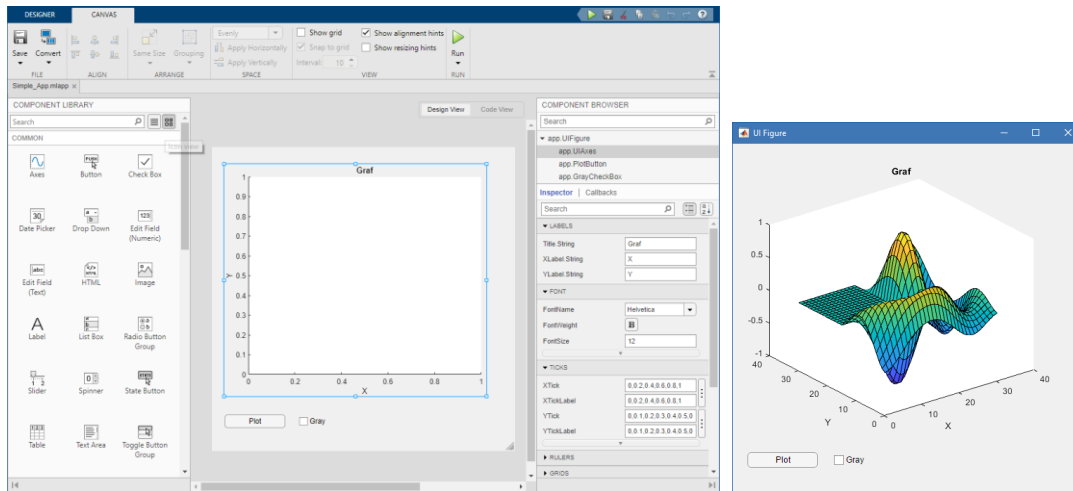
### Tvorba GUI pomocí grafického nástroje App Designer

- Grafické aplikace jsou v nástroji App Designer označovány jménem App
- App vytváříme v oknech *uifigure*
  - *uifigure* je speciální grafické okno pro aplikace vytvářené nástrojem App Designer
- Založení nové App
  - New  > App
- **Nástroj App Designer obsahuje dva náhledy**
  - *Design View* – grafický návrh App
    - hlavní okno s plátnem pro tvorbu grafického rozhraní
    - okno *Component Library*
      - ikony pro přidávání grafických ovládacích prvků
    - panel nástrojů *Canvas*
      - nástroje pro zarovnávání objektů, zapnutí mřížky, ...
  - *Code View* – programování aplikace
    - hlavní okno s programovým kódem
      - postaveno na OOP ... App je třída
      - callbacky ... ve formě metod (*methods*)
      - sdílené proměnné ... ve formě *properties*
      - sdílené funkce ... další obecné metody
    - panel nástrojů *Editor* pro editaci programového kódu
    - okno *Code Browser*
      - seznam callbacků, funkcí a proměnných
    - okno *App Layout*
      - grafický náhled aplikace
  - Společná část
    - okno *Component Browser*
      - seznam vložených grafických prvků
      - *Inspector*
        - vlastnosti vybraného grafického prvku
      - *Callbacks*
        - callbacky vybraného grafického prvku

⇒ Př.: Simple\_App.mlapp

- Vložené prvky:

Prvek	Text	Jméno
Axes	-	UIAxes
Button	Plot	PlotButton
Check Box	Gray	GrayCheckBox



- Funkce spouštěná stisknutím tlačítka ~ callback


```
surf(app.UIAxes, membrane(randi(11))) % vykreslení funkce membrane

if app.GrayCheckBox.Value == 1
    colormap(app.UIAxes, 'gray')
else
    colormap(app.UIAxes, 'parula')
end
```

- Kroky při tvorbě aplikace:
  - **1. Vložení grafických ovládacích prvků**
    - v náhledu *Design View*
    - přetažení grafických prvků z okna *Component Library* do pracovní plochy ... prvek se vloží
    - nastavení rozměrů prvku tažením za čtverečky v rozích vybraného prvku, změna polohy přetažením prvku
    - zarovnání prvků – ručně pomocí vodítek, tlačítka v záložce *Canvas*
  - **2. Nastavení hodnot vlastností grafických prvků**
    - v okně *Component Browser*
    - výběr prvku > změna vlastností v části *Inspector*
  - **3. Tvorba menu**
    - prvek *Menu Bar*
  - **4. Programování funkčnosti aplikace**
    - v náhledu *Code View*
    - je možné psát pouze do míst určených pro zápis uživatelského kódu
    - vytvoření callbacku
      - a) výběr prvku > v okně *Component Browser* v části *Callbacks* nechat vytvořit nový nebo přiřadit existující
      - b) pravý klik na prvek v okně *Component Browser* > *Callbacks* > nechat vytvořit nový nebo přiřadit existující
    - zápis kódu do callbacku
      - čtení a změna hodnoty vlastností prvků tečnovou notací
        - `hodnota = app.jmenoprvek.Vlastnost;`
        - `app.jmenoprvek.Vlastnost = hodnota;`
      - prvním argumentem grafických funkcí musí být odkaz na zvolený *UIAxes*

- `surf(app.jmenoaxesu, data)`
  - sdílené proměnné pomocí přidání *Property* s daným jménem
    - čtení nebo změna hodnoty pomocí `app.jmenopromenne`
- **5. Uložení GUI**
  - uloží se do souboru `.mlapp`
- **6. Spuštění GUI**
  - tlačítkem Run v toolstripu nástroje App Designer
  - zadáním jména aplikace do příkazové řádky MATLABu

### **Sdílení vytvořených aplikací s dalšími uživateli MATLABu**

- **Sdílení vytvořených souborů**
  - při větším počtu souborů může být nepřehledné
  - nemusí být zřejmé, kterým souborem se má aplikace spustit
- **MATLAB Apps**
  - aplikaci vytvořenou v MATLABu lze se všemi potřebnými soubory zabalit do jediného souboru, tzv. MATLAB App
  - zabalení lze provést pomocí grafického rozhraní v záložce APPS
    - Package App 
    - specifikace hlavního souboru
      - automaticky přidá všechny volané funkce
    - uživatel může přidat doplňkové soubory (nápověda, datové soubory), ikonu, popisné informace
    - $\Rightarrow$  `soubor.mlappinstall`
  - instalace
    - = rozbalení aplikace do složky dané v preferencích MATLABu
    - poklepáním na soubor `mlappinstall` v okně Current Folder
    - aplikace se přidá do záložky APPS v toolstripu
  - výhoda
    - snadné sdílení aplikací mezi uživateli MATLABu
    - spouštění aplikace ikonou – spustí hlavní soubor