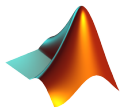


# GNU Octave a Matlab – procedury a funkce

**Programování F1400 + F1400a**

**doc. RNDr. Petr Mikulík, Ph.D.**

podzimní semestr 2020



```
function f = faktorial_cyklus ( n )  
    f = 1;  
    for k=2:n f = f*k; end  
end  
function f = faktorial_prod ( n )  
    f = prod(1:n);  
end  
function f = faktorial_rek ( n )  
    if (n<=1) f = 1;  
    else f = n * faktorial_rek(n-1);  
    end  
end
```

# Často používané kusy kódu

- Hodí se **rozdělit program na části**, které se často nebo opakovaně používají.
- Často opakované činnosti, tedy bloky kódu (v rámci jednoho programu nebo i ve více programech, např. `sqrt(x)`, `exp(x)`, výpis chyb, ...) přesuneme bokem, pak se na vyžádání daná činnost vykoná a vrátí se výsledek  $\Rightarrow$ 
  - zpřehlednění a uspořádání kódu,
  - při ladění či změnách stačí tento kód opravit na jediném místě.
- **Struktura programu:** zdrojový kód se tak rozdělí na část s deklarací **globálních proměnných**, implementací **funkcí** a **procedur**, a na hlavní **tělo programu**.
- **Funkce:** blok kódu, který vrátí nějaký výsledek (např. derivaci nějaké funkce, počet malých písmen v řetězci, počet pulsů načtených z detektoru, inverzní matici, data načtená ze souboru, apod.).
- **Procedura:** blok kódu, který nic nevrací (může třeba vypsát nějakou zprávu na obrazovku, poslat soubor na server apod.).
- Přesné používání terminologie funkce/procedura není striktně vyžadováno.

# Funkce a procedury

Uložte to jako: `privitani.m`, `soucet_soucin.m`, `letosni_rok.m`, `delka_uhel_2d.m`

## Jednoduchá procedura

```
1 % Vypis neco kratkeho na obrazovku
2 function privitani ()
3     fprintf('Trivialni program.\n');
4     fprintf('Vitej zde!\n');
5 end
```

## Procedura se vstupními daty

```
1 % Secti a vynasob dve cisla ci matice
2 function soucet_soucin (a, b)
3     fprintf('Soucet je %g\n', a+b);
4     fprintf('Soucin je %g\n', a*b);
5 end
```

## Funkce s výstupem

```
1 % Funkce vraci letosni rok, napr. 2019
2 function rok = letosni_rok ()
3     a = clock(); % vektor: datum cas
4     rok = a(1);
5 end
```

## Funkce se vstupem i výstupem

```
1 % Delka a uhel od osy x vektoru a
2 function [d, uhel] = delka_uhel_2d (a)
3     d = sqrt( a(1)*a(1) + a(2)*a(2));
4     uhel = atan2(a(2), a(1));
5 end
```

## Použití

```
1 privitani();
2 privitani;
3 fprintf('PF %i\n', letosni_rok()+1);
```

## Použití

```
1 x=1.5; y=4.0; soucet_soucin(x,y);
2 v = [1.0, 2.0];
3 [d, alfa] = delka_uhel_2d(v);
4 fprintf('Delka %g, uhel %g\n',d,alfa);
```

# Rekurentní funkce

- **Rekurentní funkce:** Funkce volající sama sebe.

Pozor na **zacyklení**! Vždy nastavit **okrajovou podmínku**.

- Příklad – **výpočet faktoriálu**, z přímé nebo rekurentní definice

$$n! = \prod_{k=1}^n k$$

Přímý výpočet cyklem for

```
1 % Vypocet faktorialu cyklem
2 function f = faktorial_cyklus ( n )
3     f = 1;
4     for k=2:n
5         f = f*k;
6     end
7 end
```

Přímý výpočet vektorově

```
1 % Vypocet faktorialu vektorove
2 function f = faktorial_prod ( n )
3     f = prod(1:n);
4 end
```

$$n! = n \cdot (n-1)!, \quad 1! = 0! = 1$$

Rekurentní výpočet

```
1 % Rekurentni vypocet faktorialu
2 function f = faktorial_rek ( n )
3     if (n<=1) % n<=1: vysledek zname
4         f = 1;
5     else      % pocitame rekurentne
6         f = n * faktorial_rek(n-1);
7     end
8 end
```

- Ke každému rekurentnímu algoritmu existuje i přímý.
- Rekurentní algoritmy spotřebují hodně paměti.

# Předčasné ukončení funkce

- Klíčovým slovem `return` vyskočíme (kdykoliv, předčasně) z funkce:

```
1 function vypis_odmocninu ( x )
2
3 if (x < 0)
4     fprintf('Pozadovana odmocnina za zaporneho cisla.\n');
5     fprintf('Vyjde komplexni cislo 0+%gi\n', sqrt(-x));
6     return
7
8 fprintf('Odmocnina z nezaporneho cisla je %g\n', sqrt(x));
9 fprintf('Ctvrta odmocnina z nezaporneho cisla je %g\n', sqrt(sqrt(x)));
10
11 end
```

- Pokud bychom místo příkazu `return` použili

```
1 exit(1);
```

tak by došlo k ukončení celého interpretru.

- Příkaz `error('popis chyby ci pouziti funkce')` slouží k předčasnému ukončení funkce a ohlášení chyby na daném řádku:

```
1 error('Spatny pocet parametru nebo neocekavana hodnota na vstupu!');
```

# Počet a ověření parametrů vstupujících do funkce

- V lokální proměnné `nargin` je počet vstupních parametrů funkce (tj. skutečný počet parametrů, se kterými byla funkce volána). Příklad:

```
1  function f = faktorial_cyklus ( n )
2
3  if (nargin ~= 1)
4      error('Funkci volejte s jednim vstupnim parametrem (celym cislem)!');
5  end
6
7  if ~isscalar(n) || ~isnumeric(n)
8      error('Vstupem musi byt cislo, ne vektor ani matice!');
9  end
10
11 % Vypocet faktorialu n!
12 f = prod(1:n);
13 end
```

- Vyzkoušejte volání této funkce s různým počtem a typem parametrů:

```
faktorial_cyklus
faktorial_cyklus()
faktorial_cyklus(5)
faktorial_cyklus([7:2:19]), faktorial_cyklus(ones(7,10))
```

- V **překládaných** programovacích jazycích (C, C++, Fortran, Pascal, ...) kontroluje počet a typ vstupních parametrů překladač, u **interpretovaných** kontrola prováděna není a na problém se přijde až během vykonávání programu.

# Zpracování více parametrů vstupujících do funkce

- V lokální proměnné `nargin` je skutečný počet parametrů, se kterými byla funkce volána. Příklad:

```
1 function testuj_nargin ( a, b, c )
2 if (nargin == 0 || nargin > 3)
3     fprintf('Chybne volani teto funkce.\n');
4     error('Volani teto funkce vyžaduje dva nebo tri parametry!');
5 end
6 % pokud funkci volame pouze se 2 parametry, tak dopln hodnotu promenne c
7 if (nargin < 3)
8     c = 42;
9 end
10
11 % a nyní již jsou definovány všechny tři vstupní parametry
12 a, b, c
```

Pak můžeme volat funkci s různým počtem parametrů a vyhodnocení nespadne:

```
testuj_nargin
testuj_nargin(1.0)
testuj_nargin(2, 3)
testuj_nargin(4, 5, 6)
testuj_nargin(4, 5, 6, 7)
```

- Pozn.: Srovnej toto chování s překládanými programovacími jazyky.

# Globální vs lokální proměnné

- **Lokální proměnné** platí pouze ve funkci, v níž byly použity.
- **Globální proměnné** platí ve všech funkcích, musí se ale deklarovat s klíčovým slovem `global`.
- **Příklad** Výpis chyby v různých jazycích

```
1 function vypis_chybu ( cislo_chyby )
2 global jazyk
3 if (jazyk == 1)
4     fprintf('Doslo k chybe cislo %i\n', cislo_chyby);
5 elseif (jazyk == 2)
6     fprintf('Il y avait une erreur %i\n', cislo_chyby);
7 else
8     fprintf('There was an error %i\n', cislo_chyby);
9 end
10 end
```

## Použití

```
jazyk=1; err=1; vypis_chybu(err)
jazyk=2; err=err+1; vypis_chybu(err)
jazyk=0; err=err+1; vypis_chybu(err)
global jazyk; fprintf('\n')
jazyk=1; err=err+1; vypis_chybu(err)
jazyk=2; err=err+1; vypis_chybu(err)
jazyk=0; err=err+1; vypis_chybu(err)
```

## Výstup na obrazovku

```
There was an error 1
There was an error 2
There was an error 3

Doslo k chybe cislo 4
Il y avait une erreur 5
There was an error 6
```



# Příkazy týkající se zpracování funkcí

- Funkční soubory musí být v prohledávané cestě, tj. buď v aktuálním adresáři nebo v seznamu prohledávaných adresářů – příkaz `path` je vypíše, příkaz `addpath('/cesta/k/adresari')` přidá zadaný adresář do cesty.

- Příkaz `which`

```
which faktorial_rek
which faktorial_rek.m
which cputime
```

vypíše, kde se nachází soubor s definicí dané funkce, příp. zdali se jedná o zabudovanou (built-in) funkci.

- Příkaz

```
type faktorial_rek
type faktorial_rek.m
```

vypíše obsah daného funkčního souboru.

- Pomocí příkazů

```
echo on all
faktorial_rek(5)
echo off
```

se budou postupně vypisovat příkazy z právě vykonávaného souboru.

- Problém s generováním *opravdu náhodných čísel* (resp. náhodného šumu) na deterministickém počítači.
- Pseudonáhodná čísla (s rovnoměrným rozdělením) v intervalu  $[0; 1)$  a `[from; to)`:

```
rand(5,6)
```

```
from=10.0; to=20.0;  
from + (to - from) * rand(5,6)
```
- Pseudonáhodná celá čísla (s rovnoměrným rozdělením) v intervalu  $[1; m]$  a `[from; to]`:

```
randi(m,5,6)
```

```
from=10; to=20;  
??? + ??? * randi(?,5,6)  % rozmyslete a vyzkousejte sami!
```

# Náhodná celá čísla a histogram

- Vygenerujte pseudonáhodná celá čísla v intervalu [from; to] a pomocí histogramu zjistěte jejich rozložení:

```
1  from=6; to=12; n=1000;
2  % vygeneruj n nahodnych celych cisel
3  c = from-1 + randi(to-from+1, 1, n)
4
5  % spocti histogram
6  [nn,xx] = hist(c,length(from:to))
7
8  % co chceme v grafu na ose x: hodnoty
9  xx          % automaticke hodnoty z vypoctu histogramu, anebo:
10 xx=from:to   % ekvidistantni hodnoty v celych cislech
11
12 clf % vymaz (clear) predchozi grafy
13 % tri ruzne grafy v jednom okne:
14 subplot(1,3,1); bar(xx,nn);    ylabel('četnost')
15 subplot(1,3,2); stairs(xx,nn); grid
16 subplot(1,3,3); stem(xx,nn, 'LineWidth', 4, 'MarkerSize', 20, 'Marker', 'o');
17 xlim([from-1, to+1])
```