

Dnešní cvičení budeme věnovat **tvorbě skriptů a funkcí**, jejich úpravám a rozšiřování. V závěru se podíváme na nástroj profile, kterým jsme se zabývali již na přednášce. K vzorovým příkladům jsem přidal i přednášku ve formátu pdf.

Byť je funkcí hodně, jsou pouze lehce modifikované a jako základ slouží fibonacciho posloupnost. Soubory proto není potřeba studentům distribuovat, neboť jistě všechny změny – budou-li chtít – stihnout spolu s vyučujícím. Pouze jim dejte event. chvilku na začátku, ať si stihnout opsat tělo funkce (zde je stejně časově náročný úvod s vysvětlením jednotlivých řádek – alespoň si je u toho budou moci opsat).

V případě dotazů nebo poznámek mne prosím kontaktujte zde:
miloslav.capek@fel.cvut.cz

Bod (1)

Nejprve si vytvoříme skript `fibonacci.m` (je přiložen) a popíšeme ho. Využíváme zde cyklu `while` (dokud platí nějaká podmínka, pokračuj ...). Jde o generaci fibonacciho řady, velikost posledního člena řady nesmí překročit hodnotu stanovenou v proměnné `limit` (defaultně `limit = 1e3`).

Po spuštění cyklu vidíme, že všechny proměnné jsou vyhodnoceny a uloženy v **základním pracovním prostoru** Matlabu (prostoru pracovního okna, odkud skript voláme).

Bod (2)

Nyní otevřeme připravený skript `fibonacci_pause2.m`, nebo pouze upravíme stávající skript tak, že doplníme `pause(2)` před příkaz `plot(f)`.

Je vidět, že skript čeká po dobu 2 sekund a až potom pokračuje vykreslením grafu. Pokud upravíme příkaz `pause(2)` na `pause` (smažeme časový údaj), čeká Matlab na stisk libovolné klávesy nebo tlačítka a poté pokračuje. Na zastavení skriptu upozorňuje Matlab varováním v levém dolním rohu hlavního okna.

Bod (3)

Vymažeme pracovní prostor Matlabu pomocí příkazů:
`clear, clc;`

a vyzkoušíme příkaz `whos` před startem a po dokončení zpracování skriptu. Změny popíšeme (vznik a uložení proměnných, jejich datové typy atd.).

Bod (4)

Původní skript `fibonacci.m` se pokusíme přepsat na funkci. Tuto práci můžeme nechat chvíli i na studentech, ať si to sami vyzkouší. V podstatě stačí připsat hlavičku k funkci (`function ...`). Pokud chceme, aby např. `limit` byl proměnný, do hlavičky ho přiřadíme mezi vstupní proměnné. Funkci `fibonacciFcn.m` (přítomna v adresáři se cvičením) ponecháme zatím bez výstupních proměnných.

Bod (5)

Opět zkusíme spustit. Nyní však vidíme, že již nestačí pouze stisk klávesy `F5` – funkce skončí chybou. Matlab totiž nezná velikost proměnné `limit`. Spustíme tedy funkci korektně:

```
fibonacciFcn(1e3);
```

Zde můžeme poukázat na první výhodu funkcí – můžeme efektivně kontrolovat velikost proměnné `limit` před startem funkce.

Nyní opět vymažeme pracovní prostor Matlabu:

```
clear, clc;
```

a vyzkoušíme příkaz `whos` před startem a po dokončení zpracování funkce. Nyní vidíme, že v obou případech je základní pracovní prostor funkce prázdný. Je to dáno tím, že funkce má vlastní pracovní prostor, který se vytvoří ihned po jejím zavolání. Tato problematika byla obšírně diskutována na přednášce.

Pro úplnost (opět po vymazání pracovního okna), umístěte příkaz `whos` do těla funkce, krátce před jejím koncem (před `plot`), po spuštění se v základním okně objeví všechny z dřívějšíka známé proměnné. To je důkaz toho, že uvnitř pracovního prostoru funkce existují.

Bod(6) (malá odbočka – funkce `edit`)

Známe-li jméno funkce a dokáže-li tuto funkci Matlab najít, můžeme se na ní podívat pomocí příkazu v základním okně Matlabu:

```
edit jmeno_funkce
```

Toho nyní využijeme. Nejprve se podíváme na náš původní skript `fibonacci.m` (předpokládám, že je již v editoru uzavřen):

```
edit fibonacci.m
```

Poté se můžeme podívat např. na funkci `sin`:

```
edit sin
```

Na příkladu této funkce je vhodné ukázat, jak má vypadat vzorová nápověda pro funkci, vč. H1 řádky (1. řádky nápovědy funkce), čímž si připravujeme půdu pro vyložení komentářů. Druhá poznámka směřuje k poslední řádce nápovědy – v případě `sinu` se jedná o build-in funkci. Tzn. že je neustále přítomna v jádře Matlabu, její využívání je velice rychlé a průběh funkce je vysoce optimalizovaný. Takovou funkci nelze měnit (editovat).

Na závěr se podívejme např. na funkci `pdetool` (GUI rozhraní PDE toolboxu, s kterým jsme pracovali minulé cvičení):

```
edit pdetool
```

Prosím, upozorněte studenty, ať **v této funkci nic nemění!** Tato funkce již není build-in, neboť je to funkce z PDE toolboxu. Další zajímavostí je její rozsah (jedná se o GUI a ty jsou typicky v Matlabu velice rozsáhlé – uvádí se, že na určitý objem funkčního kódu připadá 2-10x tolik kódu obsluhujícího grafické rozhraní). Tímto příkladem již lehce zabíháme do dalšího cvičení, které se bude věnovat handle grafice v Matlabu.

Bod (7)

Vraťme se ale nyní k naší funkci `fibonacciFcn.m`. Řekněme, že bychom rádi ještě ovlivňovali, zda se má zobrazit ve výsledné funkci `grid` a jakou barvu má mít průběh fibonacciho posloupnosti. Tyto požadavky se zjevně neobejdou bez rozšíření vstupních proměnných funkce. Založte proto novou funkci `fibonacciFcn2.m` a pro tyto požadavky ji upravte. Další možností je otevřít tuto funkci z připraveného adresáře.

2. vstupní proměnná je ve formátu `[r g b]` a protože přímo vstupuje jako parametr do funkce `plot`, musí splňovat podmínky Matlabu $\{r,g,b\} \in (0,1)$. 3. proměnná `isGrid` je datového typu `logical` (1B) nebo `double` (8B), můžeme ji tedy zadat několika způsoby:

```
isGrid = true / false
isGrid = 0 (false) / libovolné číslo (true)
```

V tuto chvíli je nutné zadat vždy všechny 3 vstupní parametry, jinak funkce skončí chybou (vyzkoušejte si). Později si funkci upravíme tak, že bude volatelná s jedním, dvěma nebo i třemi parametry.

Bod (8)

Nyní si funkci rozšíříme i na výstupní straně – zavedeme rovnou dva výstupní parametry a zkusíme funkci volat s jedním i dvěma výstupními parametry. Budeme se zajímat o počet členů řady, které byly vypočteny pro daný `limit` a velikost posledního členu této řady. Funkci naleznete v adresáři pod názvem `fibonacciFcn3.m`. Pokud jí nyní spustíme následovně:

```
[clenu posClen] = fibonacciFcn3(1e3,'b',0)
```

vyjde nám `clenu = 16` a velikost `posClen = 987`. Vidíme, že barva grafu může být zadána i pomocí jednoho z tagů, které Matlabu akceptuje ('k' – černá, 'r' – červená, 'y' – žlutá, ...).

Funkci však můžeme, jak bylo uvedeno na přednášce, volat bez výstupních parametrů, nebo pouze pro první, od verze R2009b dokonce i pouze pro druhý, vyzkoušejte:

```
[clenu posClen] = fibonacciFcn3(1e3,'b',0)
clenu = fibonacciFcn3(1e3,'b',0)
[~, posClen] = fibonacciFcn3(1e3,'b',0)
fibonacciFcn3(1e3,'b',0)
```

Bod (9)

Funkci lze volat i s funkcí jako vstupním parametrem – Matlab je schopen „jít odzadu“ a postupně vyhodnocovat funkce jako argumenty. Takto lze vrstvit více funkcí do sebe a to ve všech případech (argumenty funkcí, nastavení grafiky, práce s maticemi atd.).

```
[clenu posClen] = fibonacciFcn3(sqrt(1e6),'r',0)
[clenu posClen] = fibonacciFcn3(abs(-sqrt(1e6)),'y',1)
```

Bod (10) (malá odbočka – komentáře)

Pro další účely se nám bude hodit umět efektivně pracovat **s komentáři ve funkcích Matlabu**. Nejen, že komentáře zpřehledňují jakoukoliv funkci pro ostatní, ale i vlastnímu autorovi pomohou – vrací-li se k obtížné části kódu po delší době – navázat, kde skončil. V Matlabu lze psát jednořádkové a víceřádkové komentáře, komentovat lze i v rozdělených řádcích kódu. Byť nepsané, ale velice důležité je i pravidlo komentovat všechny funkce v Matlabu bezprostředně pod hlavičkou funkce. Zde zopakujeme jméno funkce, vstupy, výstupy, příp. metodu, kterou využíváme, příklady volání, uvedeme autora a rok vzniku, event. verzi a rozdíly oproti verzím předešlým. Všechny tyto zásady byly implementovány na soubor `fibonacciFcn_comments.m`.

Vyzkoušejte ho znovu spustit. Ukažte studentům jednotlivé typy komentářů a zásady v komentování souboru. Ukažte odezvu na příkazy:

```
help fibonacciFcn_comments
lookfor fibonacciFcn_comments
```

Jedná se o v praxi často užívané dotazy. Srovnejte s:

```
help cos
lookfor cos (najde všechny H1 řádky, kde se cos objevuje)
```

Bod (11)

Nyní se pokusíme upravit funkci tak, abychom mohli i na vstupní straně měnit počet parametrů. Pro tyto účely otevřete `fibonacciFcn4.m`. V tomto bodě využijeme funkci `nargin`, která je schopna zjistit, kolik parametrů je na vstupu dané funkce. Hojně již využíváme i komentářů (nápověda funkce z časových důvodů vypuštěna). Vyzkoušejte:

```
fibonacciFcn4
fibonacciFcn4(10e3)
fibonacciFcn4(10e3, 'y')
[clenu clen] = fibonacciFcn4(1e5, 'c', true)
```

Bod (12)

V tomto kroku se budeme věnovat funkcím `varargin` a `varargout`. Umožňují využívat proměnného počtu vstupních a výstupních parametrů přímo v hlavičce funkce. Vše je zobrazeno ve funkci `fibonacciFcn5.m`.

Uživatel může volat funkci s proměnným počtem vstupů a výstupů (příčemž min. 1 vstup – `limit` – je vždy vyžadován), podle počtu proměnných se data rozbalí nebo zabalí do funkcí `varargin` a `varargout`. Vyzkoušejte následující příkazy:

```
[clenu] = fibonacciFcn5(1e5)
[clenu] = fibonacciFcn5(1e5, 'k')
[clenu] = fibonacciFcn5(1e5, [.4 .9 .1], true)
[clenu clen] = fibonacciFcn5(1e7, [.4 .2 .95], true)
[clenu clen isGrid] = fibonacciFcn5(1e7, [.85 .05 .15], true)
[clenu clen isGrid] = fibonacciFcn5(1e3)
```

Věnujte, prosím, dostatečnou pozornost **řádkám 6 až 16** ve funkci `fibonacciFcn5.m`.

Bod (13)

Příkaz `return`. Zkusíme do funkce `fibonacciFcn5.m` vložit příkaz `return` (stačí odkomentovat, je připraveno), pokud je barva grafu modrá. Má-li být graf modrý, je funkce ukončena bez jeho vykreslení díky příkazu `return`. Vyzkoušejte pro zelenou barvu (bude vykresleno):

```
[clenu clen] = fibonacciFcn5(1e3, 'g', true)
```

... a pro modrou barvu (nebude vykresleno):

```
[clenu clen] = fibonacciFcn5(1e3, 'b', true)
```

Bod (14)

Funkce `warning` a `error`. Otevřeme funkci `fibonacciFcn6.m`. Zde jsou navíc přidány příkazy `warning` (vypíše upozornění) a `error` (zahlásí chybu a ukončí program, s chybami lze pracovat podobně jako v Javě – chyby zachytávat, zahazovat, vytvářet atd.).

Nejprve zkusíme zobrazit upozornění `warning`. Upozornění je vypsáno, pokud ve funkci nebude zobrazen `grid`. Zavoláme tedy funkci se zobrazením a bez zobrazení `gridu` a porovnáme výsledky:

```
[clenu clen] = fibonacciFcn6(1e3, 'g', true)
[clenu clen] = fibonacciFcn6(1e3, 'g', false)
```

Nyní se pokusíme vyvolat chybu – chybové hlášení je implementováno tak, že se zobrazí, pokud je zadán pouze jeden vstupní parametr, tj. `limit`. Při více vstupních parametrech není zobrazeno. Srovnajte:

```
[clenu clen] = fibonacciFcn6(1e3, 'y', 327)
[clenu clen] = fibonacciFcn6(1e3)
```

Bod (15)

Nested funkce, vedlejší funkce. Stávající funkci rozdělíme na tři funkce:

`fibonacciMainFcn.m` (hlavní funkce, volá `nested` a vedlejší funkci, hlídá parametry)

`fibonacciSecFcn.m` (vedlejší funkce, funkční část kódu – počítá posloupnost)

`fibonacciMainFcn.m` > `plotGraphics` (zobrazuje graf posloupnosti, event. vykresluje `grid`, nested funkce pro funkci `fibonacciMainFcn.m`)

Všechny funkce jsou předpřipraveny k použití v adresáři cvičení. Na `nested` funkci můžete ukázat, že její pracovní prostor vidí i proměnné z pracovního prostoru hlavní funkce. Vedlejší funkce je v tomto případě logicky i fyzicky oddělena od funkce hlavní. Lze ji volat i zvnějšku:

```
f = fibonacciSecFcn(1e3)
```

Využíváme však volání:

```
[totTerms lostTerm] = fibonacciMainFcn2(1e20, 'g', true)
```

Členění do více funkce zpravidla napomáhá přehlednosti a omezuje duplicitu tím, že určitou část kódu můžeme v hlavní funkci volat opakovaně bez nutnosti přepisování. Využití více funkcí je vhodné o pro rozsáhlé aplikace (zde to pomalu ani jinak nejde), protože se práce lépe rozděluje mezi více pracovníků, lépe se soubory aktualizují a pomocí hlaviček funkcí jsou dána pevná rozhraní, kterými se musí všichni řídit.

Nested funkcí může být více, to platí i pro vedlejší funkce. Mohou se dokonce shodovat jménem, potom funkci přetěžujeme (lze přetížít např. funkci `sin`). Všechny tyto zásady a vlastnosti měly být řešeny na přednášce, zde na ně není mnoho prostoru.

Na závěr postupně od-komentujte `whos` v hlavní funkci a zanořené funkci a srovnajte jejich pracovní prostory.

Chceme-li nyní vylepšit grafický výstup programu, stačí změnit funkci `plotGraphics`. To je ukázáno ve funkci `fibonacciMainFcn2.m`. Vidíme, že stále pro výpočet voláme stejnou funkci `fibonacciSecFcn.m`, ale změnil se styl vykreslování. Bohužel zde není čas popsat jednotlivé kroky vykreslování. Část by však již měla být probrána, část bude řešena na příštím cvičení (handle grafika, nastavování vlastností atd.).

Bod (16)

Zkuste si funkci `fibonacciSecFcn` přesunout do vytvořeného adresáře `[private]` (měl by být zatím prázdný). Opět volejte:

```
fibonacciMainFcn2(1e7, 'g', true)
```

Funkce funguje. Důvod je jednoduchý – když se hlavní funkce `fibonacciMainFcn2` snaží najít funkci `fibonacciSecFcn`, hledá i v adresáři `[private]`, je-li tento dostupný. Funkce v tomto adresáři mohou volat pouze funkce bezprostředně nad ním, pro všechny ostatní funkce je adresář za normálních podmínek neviditelný.

Toto je důvod, proč pro rozsáhlé aplikace, které jsou stále programovány strukturálně, najdeme většinu podružných funkcí v podobném adresáři. Je to otázka bezpečnosti přístupu i přehlednosti.

Bod (17)

Často je vhodné vědět, jak dlouho která konkrétní funkce, nebo určitá její pasáž běžela. Pro tyto účely je v Matlabu definována dvojice funkcí `tic` a `toc`. Můžeme je využívat jak v pracovním okně, tak ve skriptech a funkcích. Zkuste si např. v pracovním okně napsat `tic`, poté provést několik základních operací a poté napsat `toc`. Matlab sám ukáže výsledný čas, který oběhl mezi zadáním `tic` a `toc`. S oběma funkcemi lze pracovat více (`a = tic`, `b = toc`, tedy pomocí přiřazení), ale to je nad rámec tohoto cvičení.

Další možností jak postihnout čas v Matlabu je využití funkcí `clock` a `etime`. Funkce `clock` nedělá nic jiného, než že zobrazí velice přesný čas na konkrétním počítači. Zkuste si:

```
timer1 = clock
```

Čas je ve formátu rok-měsíc-den-hodina-minuta-milisekunda, tedy v US formátu. Pokud chceme někdy později (v další části funkce) vědět, kolik času uplynulo, použijeme příkaz:

```
totTime = etime(clock, timer1);
```

Z toho plyne, že funkce `etime` pouze vypočte rozdíl mezi `clock` (současným časem) a časem `timer1` a vrátí ho do proměnné `totTime` (v sekundách). To má proti `tic-toc` několik výhod, jedna z nich je, že časových vektorů (timerů) můžeme mít neomezeně mnoho. Lépe se s nimi také pracuje. Pokud si chcete tuto část procvičit, vložte několik timerů do funkce `fibonacciMainFcn2`.

Bod (18)

Příkaz `inputname`. Tento příkaz umožňuje zjistit uvnitř funkce, jak se jmenovali vstupní parametry ve chvíli, kdy byla funkce volána. Pro tyto účely otevřete funkci `fibonacciMainFcn3.m`. Je vhodné poznamenat, že pokud byla funkce volána např. následovně:

```
fibonacciMainFcn2(sqrt(1e6), 'b', true)
```

vrátí `inputname(1)` prázdné pole, neboť `sqrt(1e6)` není jméno proměnné. To platí i pro `inputname(2) /'b'/` a `inputname(3) /true/`.

Budeme-li funkci volat se skutečnými proměnnými:

```
lmt = 1e3;  
clr = 'b';  
isGrd = true;  
fibonacciMainFcn3(lmt, clr, isGrd)
```

dostaví se očekávaný výsledek, tj. uvnitř funkce jsme schopni stanovit, jak se vstupní parametry jmenovali vně funkce.

Bod (19)

Posledním bodem cvičení je analýza funkce pomocí nástroje `profile`. Tento nástroj umožňuje kompletní rozbor celé aplikace, logických závislostí a zejm. časové náročnosti jednotlivých kroků. Zejm. velice komplikované funkce s rozsáhlými maticemi a více cykly je vhodné takto analyzovat.

Všechny základní příkazy byly probrány na přednášce, takže zde se pouze podíváme na vlastní funkci `profile`. Nejprve nástroj vypneme, vynulujeme timery a opět spustíme.

```
profile off  
profile clear  
profile on
```

Poté voláme funkci, kterou chceme analyzovat. Je vhodné hledat delší posloupnost, získáme tak relevantnější data:

```
fibonacciMainFcn2(1e20, 'g', true)
```

Nyní zobrazíme výsledky:

profile report

Většina odkazů je interaktivních a zavedou nás buď do podružné funkce, nebo přímo na řádku kódu, který je za daný krok zodpovědná. Zbývá-li čas, můžete nástroj řádně prozkoumat.

Zadání samostatné práce:

- a) Nahrazení cyklu `while` cyklem `for` (změna podmínky z `limit` na počet členů řady).
- b) Tvorba vlastní uživatelské funkce. Min. jedna vedlejší funkce, min. 2 vstupní parametry a 2 výstupní parametry. Test počtu vstupních parametrů, event. i test typu dat a jejich korektnosti. Funkce může zpracovávat libovolný inženýrský problém.

Domácí úkol:

- vše znovu projít, promyslet dotazy a nejasnosti
- vyzkoušet jinak vrstvené nested funkce a vedlejší funkce (je to skutečně důležité)
- vyzkoušet příkazy `evalin` a `assignin`
- zkouška anonymních funkcí a handleů funkcí (vše je zpracováno v přednášce příp. v helpu Matlabu; usnadňuje některé partikulární úlohy při vypracovávání vlastních studentských prací)