Úvod do SCILABu

Pavla Pecherková

20. února 2017

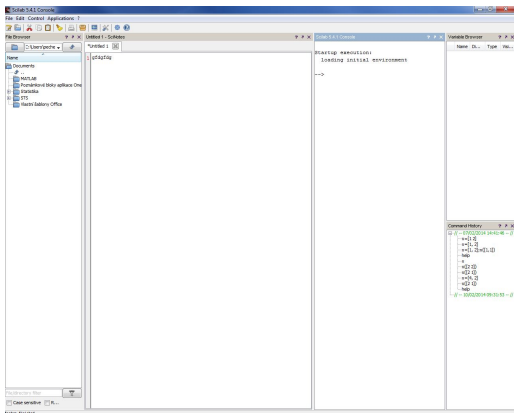
Obsah

1 Aplikace SCILAB

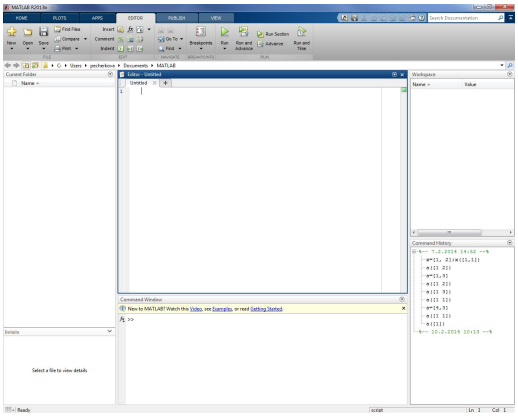
Scilab je volně šiřitelný program pro numerické výpočty podobný systému MATLAB[1]. Program byl vytvořen francouzskými vědeckými institucemi INRIA a ENPC. Jeho licence umožňuje bezplatné používání [CC - wikipedie]. Program je volně ke stažení na následující adrese

`http://www.scilab.org/download/5.4.1`.

Po nainstalování a spuštění programu se objeví následující okno (v porovnání s Matlabem R2013b)



SCILAB



MATLAB

Při prvním spuštění není editor (SciNotes) součástí nabídky. Jak ho pustit, připojit atd. si ukážeme v 1.2.

V následujícím textu bude popis práce s programem Scilab. Pokud bude chtít někdo pracovat v programu Matlab, nebude to problém, jen je potřeba si uvědomit, že bude muset znát lehce odlišný zápis. Stručný návod pro program Matlab je zde `http://www.fd.cvut.cz/personal/nagyivan/PrpStat/Prp/MatIntro.pdf`.

1.1 Popis oken

Po spuštění programu Scilab se tedy objevuje toto okno (bez SciNotes - dále editor)

V levé části aplikace je okno File Browser, kde se zobrazují podadresáře a soubory právě aktivního adresáře (jedná se o adresář, který je vybrán a zobrazen v bílém okénku v horním pruhu ikon).

Ve střední části aplikace se nachází okno Console, tedy tzv. příkazový řádek. V konzoli lze napsat jednoduchý výpočet (například 6+8) a zobrazují se zde výsledky programů.

V pravé části aplikace nahoře je okno Variable Browser zobrazující aktuální proměnné a jejich hodnoty včetně typu. Pod ním je okno Command History, kde lze vidět historii příkazů, jak se zadávaly v konzoli.

Na následujícím obrázku 2 si ukážeme, jak vypadá první zápis (doporučuji zopakovat):

1. Nejdříve byla zavedena proměnná `a=6;` - středník na konci řádku znemožní vypsání výsledku v prostředí konzole, ale její zavedení lze zkontrolovat v okně Variable Brouser.

2. Na dalším řádku je zápis `b=a+2` - proměnnou `a` již `máme` zavedenou, proto tvrdíme, že `b=6+2`. Na konci řádku není středník a i v konzoli je vypsán výsledek, že `b=8.`

3. Na dalším řádku se pokoušíme zjistit hodnotu π, ale výsledek je, že variable `pi` není definována (výsledek: `!--error 4`).

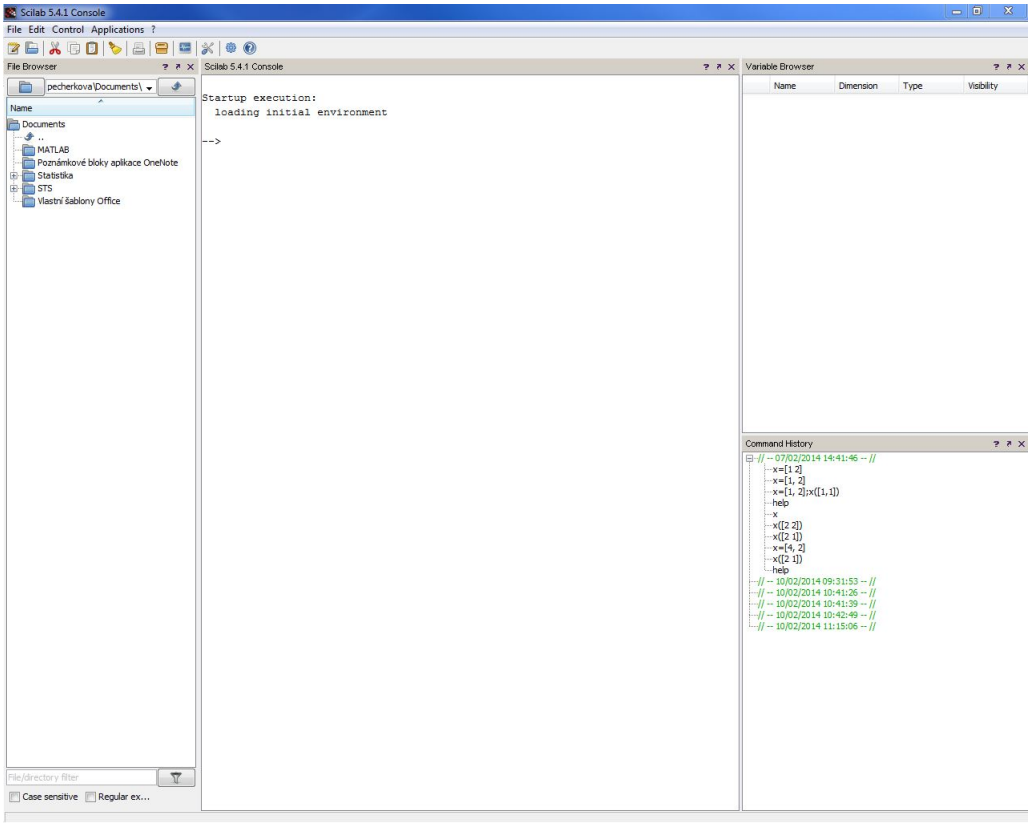
4. Na dalším řádku je zavolána pomoc `help`, abychom zjistili, jaký je správný zápis.

5. Zjišťujeme, že správný zápis je tedy `%pi` jak je vidět na dalším řádku a zobrazí se výsledek `3.1415...` Protože jsme ovšem nepřiřadili `%pi` žádné proměnné v okně Variable Brouser se nezobrazuje

6. Na posledním řádku přiřadíme tuto hodnotu tedy proměnné `pi` a tato proměnná se již ve Variable Browser objevuje.

[1]MATLAB je profesionální interaktivní systém určený pro technické výpočty. Je vyroben a dále vyvíjen firmou The MathWorks, Inc. a je chráněn americkými patenty. Škola jej má legálně zakoupen, ale funguje pouze na síti ČVUT. Ke stažení je na adrese

https://download.cvut.cz



Obrázek 1: Úvodní okno bez editoru

Okna lze aktivovat nebo skrývat z menu Desktop nebo přímo myší - pro přetažení lze okno uchopit za modrou lištu v místě s písmeny a táhnout. Objeví se rámeček, který postupně přeskakuje. V okamžiku, kdy jsme s umístěním spokojeni, okno pustíme a to se usadí. Okno lze rovněž z integrovaného prostředí vytáhnout kliknutím na šipku otočenou šikmo nahoru v pravé části ikonové lišty. Po vytažení se šipka otočí šikmo dolů a okno se opět zasune do aplikace (dokuje).

1.2 SciNotes - editor

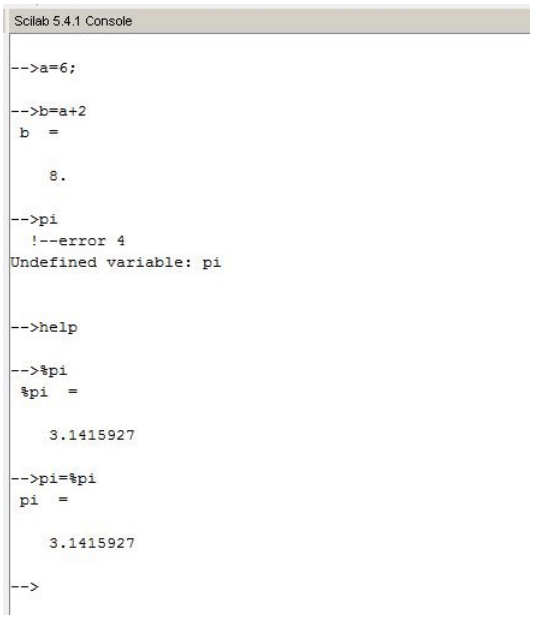
SciNotes editor je nástroj k psaní kódu (programu).Tento editor je propojen přímo s programem Scilab a tak umožňuje psát plnohodnotné programy bez nutnosti instalace dalších podprogramů. Pouze v omezených případech nám stačí k práci konzole. K uložení vytvořeného kódu a budoucím úpravám slouží právě SciNotes - editor. Kdy je vhodné použít konzoli a kdy editor je rozepsáno v kapitole 1.3.

Při prvním spuštění (a při každém dalším, kdy je nejprve vypnut SciNotes a až poté Scilab) se lze k editoru nejrychleji dostat přes tlačítko vlevo nahoře (viz zakroužkovaná oblast na obrázku 3 (a)). Po spuštění se objeví nové okno, viz 3 (b), kde se dá už přímo psát kód. Tento program se jeví téměř jako samostatný, proto má, na rozdíl od ostatních oken, i své vlastní nastavení. Jsou zde dvě možnosti, jak mít editor umístěn. Buď bude samostatně jak je automaticky po startu, tzv. undock nebo bude součástí hlavního okna, tzn. bude jedním z oken, které jsou popsány v kapitole 1.1.

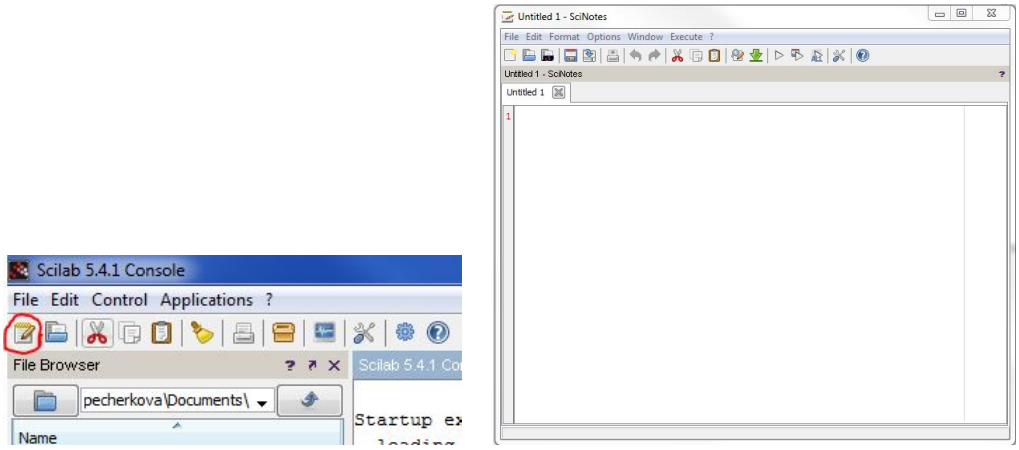
Umístění do hlavního okna se udělá následujícím postupem. Otevřený editor “chytneme” za modrý pruh s názvem souboru (*.sce) a cesty k němu. Na obrázku 4 je označen červeným obdélníkem. Po uchopení (a držení) lze umístit editor kamkoliv na hlavní panel. Budoucí umístění se zobrazí pouze jako obrys a až po uvolnění se tam umístí skutečně. Po umístění lze upravit výšku resp. šířku okna (jakéhokoliv). Úprava se provádí pomocí šipek mezi jednotlivými okny. Při znovuotevření Scilabu se bude umíštění shodovat s umístěním, které bylo při vypnutí.

1.3 Práce v aplikaci Scilab

Jak již bylo zmíněno, ve Scilabu existuje dvojí způsob práce.



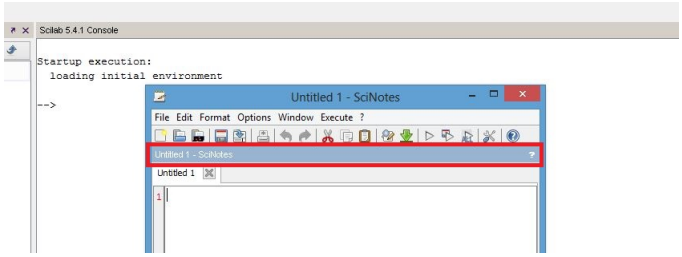
Obrázek 2: Úvodní okno - první práce



(a) spouštění editoru (b) editor

Obrázek 3: SciNotes - zapnutí

- **Interaktivní** - pracuje se pouze v okně Console, kde se zadávají (jednořádkové) příkazy a po odklepnutí pomocí Enter se ihned obdrží odpověď (pokud není za příkazem středník nebo pokud příkaz nějaký výsledek dává).
  - Pokud není zadaný příkaz (výraz) přiřazen proměnné, jeho hodnota je přiřazena obecné proměnné ans. Pod tímto jménem lze spočtenou hodnotu následně zavolat. Pozor, dalším výpočtem bez proměnné bude přepsána.
  - Naposledy zadaný příkaz (a pak i starší) lze vyvolat klávesou ↑ a dále jej editovat. Pokud znáte jak hledaný řádek začíná, je výhodnější napsat začátek řádku a až poté zmáčknout klávesu ↑. Například řádek `obsah=vyska*prumer^2*%pi` lze z historie zavolat tak, že bude zadáno `obs` a poté stisknuta klávesa ↑.
- **Dávkový** - jednotlivé příkazy jsou zapisovány jako program do editoru. Potom jsou příkazy spuštěny najednou. Program lze spustit klávesou F5, ikonkou Execute (▷), která je umístěna na liště, nebo z okna souboru.
  - Výsledky lze získat několika způsoby: (i) tiskem nebo grafem z programu, (ii) dotazem na proměnnou



Obrázek 4: SciNotes - připojení k hlavnímu panelu

po ukončení běhu, (iii) sledováním výsledku v okně Console nebo (iv) v okně Variable Browser, kde lze prohlížet i složité proměnné.

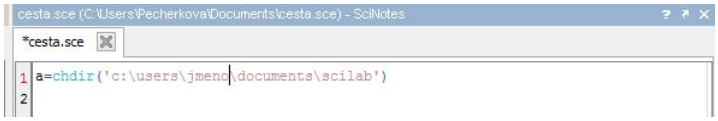
– Pokud program neprojde, ohlásí v Console chybu. Ve výpisu chyby je napsáno místo a příčina chyby. Toto hlášení je velmi užitečné, ale často zavádějící. Funguje zejména u nenadefinování proměnných, u složitějších chyb to často nehlásí správně.

1.4 Pracovní adresář

Scilab pracuje s tzv. pracovním adresářem. Do tohoto adresáře se ukládají programy, které vytvořím. Aktuální adresář může být kterýkoliv který je k tomuto účelu vytvořen.

Matlab pracuje s tzv. pracovním adresářem. Do tohoto adresáře ukládá výsledky a v tomto adresáři také nejdříve hledá volané soubory. Obsah pracovního adresáře se ukazuje v okně Current folder. Nastavit cestu do požadovaného adresáře lze několika způsoby:

- Nejjednodušší způsob je nastavit cestu přímo v souboru *.scilab* nebo *scilab.ini*.
- Druhá možnost je nalistovat cestu přímo v okně File Browser. Test správnosti nastavení lze pomocí funkce `pwd`.
- Třetí možnost je nastavení přímo v příkazovém řádku v Console, pomocí DOSových příkazů `cd` a `dir`. Tedy například `cd('c:\users\scilab')`.
- Poslední možnost je výhodná zejména na soukromých počítačích, pro školní účely není zcela vhodná. Do aktivního adresáře po spuštění se uloží soubor, kde bude pouze nastavení cesty na požadované místo. Například se vytvoří soubor `cesta.sce` kde bude uložena požadovaná cesta. Příklad takového souboru je na obrázku 5. V příkazovém řádku se jen zavolá `('cesta.sce')`.



Obrázek 5: Soubor s cestou

1.5 Podprogramy

V programech jazyku Scilab je možné využívat podprogramy standardní (obsažené v knihovnách Scilabu) nebo vlastní.

Scilab obsahuje velké množství vlastních funkcí. Jejich přehled lze dostat tak, že se v Console zadá příkaz help. Objeví se klikací seznam oblastí funkcí s krátkým popisem. V něm lze dále hledat. Na konkrétní funkci je možné se dotázat příkazem help jméno (například help plot). Lze také využít nápovědu Scilabu, která může být vyvolána nejlépe kliknutím myší na ikonu help - otazník na kterémkoliv okně. Tady se nachází kromě návodu i řada příkladů k použití dané funkce.

Vlastní podprogramy se píší jako samostatný programový soubor se jménem souboru a koncovkou .sci nebo .sce. Podle typu se také volají

- **Skript (procedura)** - skupina příkazů samostatně uložená. Jedná se o skupinu po sobě jdoucích kroků, kde se používají jiné funkce a výstupem je výsledek, graf atd. Zpravidla se nevolá jinou funkcí nebo procedurou i když to není vyloučené. Tyto procedury se ukládají jako soubory s příponou .sce.
- **Funkce** - skupina příkazů samostatně uložená včetně hlavičky s formálními vstupními a výstupními parametry. Volá se jménem se skutečnými parametry, které se předávají do formálních podle pořadí.
- **Jednořádková funkce** - skupina příkazů nebo funkcí, které se píší jako string do jednoho řádku za funkci. Je výhodné ji použít pro nějaké jednoduché (pomocné) výpočty, které se dělají několikrát na různých místech kódu. Například, pokud bude potřeba sečíst dvě hodnoty. Zápis takové funkce se provádí pomocí funkce `deff('[vystup]=nazev_funkce(vstup1,vstup2)','vystup=vstup1/vstup2')`. První část funkce je název a za čárkou (,) je nadále v apostrofech zadáno co se má dělat se vstupnými proměnnými. V tomto případě bude `vstup1` podělen proměnnou `vstup2`.

Příklad zápisu funkce:

```
function [m1,m2,v1,v2,co] = momenty(x,y)
  //mx ... stredni hodnota vektoru x
  //my ... stredni hodnota vektoru y
  //vx ... rozptyl vektoru x
  //vy ... rozptyl vektoru y
  //co ... covariance mezi vektory x,y
  mx=mean(x);
my=mean(y);
vx=varince(x);
vy=variance(y);
co=cvr(x,y); //funguje pouze se stat. balickem
endfunction
```

Příklad použití funkce:

```
x = [1 2 2 3 2 3 1 1];
y = [3 5 2 6 1 2 6 3];
[meanX, meanY, varX, varY, covXY] = momenty(x,y)
```

1.6 Datové soubory

Data se ve Scilabu nachází ve dvou formách - data vstupní, která chceme zpracovat, a data vypočtená v programu. Oba druhy dat lze natáhnout do paměti, nebo naopak uložit na disk pomocí příkazu load nebo save. Nejběžnější je následující syntaxe

příkaz	**význam**
load("jmeno.sod")	volá se soubor jméno.sod ... načtou se všechny proměnné
load("jmeno.sod","prom1","prom2",...,"prom_posledni")	volá se soubor jméno.sod ... načtou se jen vyjmenované proměnné
save("jmeno.sod")	ukládá se soubor jméno.sod ... uloží se všechny v danou chvíli známé proměnné
save("jmeno.sod","prom1","prom2",...,"prom_posledni")	ukládá se soubor jméno.sod ... uloží se jen vyjmenované proměnné

```
vektor_lichy=1:2:50;   // vektor lichých čísel
save('data.sod','vektor_lichy')  // uloží vektor do souboru data.sod
clear   //vymaže všechny proměnné
load('data.sod')  //načte všechny proměnné ze souboru data.sod
```

Pokud z vybranného souboru načteme data, proměnné se zapíší do vnitřní paměti počítače, takže program je již bude znát. Známé proměnné můžete zkontrolovat v okně Variable Browser.

2 Počáteční nastavení - Startup

V případě, že si vytvoříme vlastní funkce nebo funkce získáme z jiných zdrojů než je oficiální Scilab, je důležité je správně načíst. Pokud tyto funkce budeme používat častěji (nebo hrozí, že zapomeneme, kde je máme), je zde možnost načítat tyto funkce automaticky při každém spuštění Scilabu. Jedná se vlastně o automatické načtení souboru s funkcemi (s příponou *.sci*) do vnitřní paměti Scilabu. Tímto "zavoláním" se funkce dostanou do vnitřní paměti Scilabu, který je bude znát.

Jsou zde dvě možnosti, jak tyto funkce načíst: (i) pomocí funkce `getd()` nebo (ii) pomocí knihovny `genlib()`.

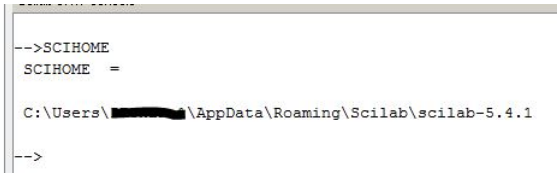
1. `getd(''cesta'')` - jednodušší způsob, ale výpočetně pomalejší. Napíšeme příkaz `getd()` s danou cestou a poté už bude matlab znát všechny funkce, které jsou v sobourech na dané cestě,

2. `genlib(''funlib'',''cesta'')` - funkce v dané cestě přeloží a uloží do knihovny s názvem *funlib*. Název je možno měnit, takže každá knihovna by měla mít svůj unikátní název. Tento způsob je složitější v tom, že se knihovna překládá, ale na druhou stránku to umožní rychlejší přístup pro Scilab a výpočtně se jedná o rychlejší variantu.

2.1 Spouštění Scilabu

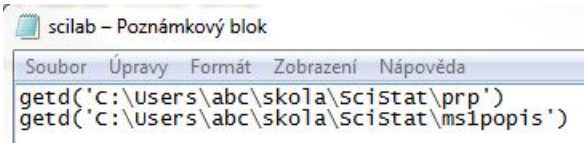
1. pracuji na svém soukromém počítači.

   V tomto případě je nejvhodnější vytvořit soubor *scilab.ini* kam lze napsat cestu ke všem adresářům s funkcemi. Tento soubor se uloží do domovského adresáře Scilabu. Jaký je domovský adresář lze zjistit tak, že na příkazovém řádku se zavolá `SCIHOME`, viz. obrázek č. 6.



Obrázek 6: SCIHOME

   Do tohoto adresáře se napíší všechny cesty k adresářům, kde jsou uloženy požadované funkce. Volání se provede pomocí funkce `getd()`. Například při přidání cesty k adresáři prp to bude: `getd(getd('C:\Users\abc\skola\SciStat\prp`
   Pokud takto vložíme do souboru víc funkcí, jak je ukázáno na obrázku 7. Pokud budete mít správně nastavené cesty, tak po spuštění budete moct používat dané funkce.



Obrázek 7: Scilab.ini

2. pracuji na školním počítači (počítači s omezeným přístupem)

   V takovém případě není možné uložit adresy přímo, ale je možno vytvořit si vlastní soubor, například *spust_ cestu.sce*, kde budou uloženy všechny cesty k adresářům (jako v předcházejícím odstavci - pracuji na svém soukromém počítači), také pomocí funkce `getd`.

   Na rozdíl od předchozí varianty, zde po spuštění programu SCILAB nebudou funkce načteny do vnitřní paměti automaticky, ale bude nutno nejdříve spustit proceduru *spust_ cestu.sce*. Až poté, budou funkce načteny, tedy za předpokladu, že uložené cesty jsou správné. Je vhodné opět otestovat, zda vše proběhlo správně a tedy, zda funguje vybraná funkce.

2.2 Vytvoření/úprava funkce

Další problém (opomenutí) může nastat v případě, že změníte funkci, na které právě pracujete, nebo již dříve načtenou do vnitřní paměti. Ani po uložení nedojde ke změně automaticky ve vnitřní paměti. Po každé takové změně je potřeba aktualizovat adresář, ve kterém se daná funkce nachází funkcí `getd()`. V tomto případě je zbytečné cestu zadávat do zvláštního souboru, ale pouze na příkazovém řádku v konzoli se zadá cesta `getd('C:\user\abc\skoda\scistat\moje')`, resp. adresář, kde jste pracovali. Po takové aktualizaci dojde i ke změně ve vnitřní paměti a Scilab bude pracovat s novými úpravami.

V předchozím textu bylo ukázáno, jak se provede počáteční nastavení pomocí funkce `getd()`. Pokud bychom chtěli pracovat s knihovnami, postup je stejný, jen místo `getd()` dáme `genlib()`. Stále platí, že pokud změníme v paměti načtenou funkci, musíme pro její aktualizaci znovu načíst novou verzi do paměti, jinak pracuje s verzí starší.

3 Úvod do programování v jazyce SCILAB

3.1 Úvodní poznámky

1. Scilab nerozlišuje, zda píšete skalár, vektor nebo matici. Vše je matice, skalár je vlastně matice rozměru 1×1 a vektor je matice rozměru $1 \times n$ (řádkový vektor) nebo $n \times 1$ (sloupcový vektor).

2. Středník za zadanou proměnnou (v konzoli i SciNotes) `a=5;` způsobí, že se výsledek nevypíše v konzoli (proměnná je samozřejmě ve vnitřní paměti známa a jde s ní nadále pracovat). Naopak po zadání proměnné s čárkou nebo bez ničeho, tj. `a=5`, nebo `a=5` se hodnota proměnné vypíše na obrazovku.

3. help „objekt“ zobrazí nápovědu k objektu, který je hledán. `help` v příkazovém řádku Ikona **?** zavolá interaktivní stránku HELP.

4. Komentář je text začínající `//`. Nehraje žádnou roli ve výpočtech programu. Přesto důrazně doporučujeme používat při popisu programu (i po krátkém čase nemusí být jasné, co bylo cílem které části kódu).

5. Příkazy je možné zapsat do textového souboru s koncovkou .sce(batch-file) nebo .sci spustit je najednou zavoláním jména souboru v hlavním programu.

3.2 Proměnné a operace

Typy proměnných

Následně jsou zmíněny základní typy proměnných, se kterými budeme pracovat. Jsou však jen malým zlomkem z existující množiny proměnných.

- **string** slouží pro zápis textu: a=’ahoj’. Stringy lze spojovat do vektorů:
  je-li `a='dobry '`; `b=' den'` a `c=a+b`, pak `c='dobry den'`. Tento postup vytvoří jeden string. Pokud chceme zachovat stringy samostatně, a to ve formě matice nebo vektoru, tak lze použít `matice_string=['a' 'b'; 'c' 'd']`.

- **logické proměnné** jejichž hodnotami je „true“ (kódováno jako T) a „false“ (kódováno jako F). Logická proměnná je výsledkem dotazu, kde výsledkem je opravdu pouze pravda a nepravda (volá se pomocí `%T` resp. `%F`). Pro další práci se někdy booleanský výstup nehodí, proto se převádí pomocí funkce `bool2s()` na hodnoty 0 pro nepravdu (false) a 1 pro pravdu (true). Příkladem může být následující kód:

výsledkem je pravda	výsledkem je nepravda
a=1;	a=1;
`b=a==1;`	`b=a==2;`
`pravda=bool2s(b)`	`nepravda=bool2s(b)`

- **double proměnné** slouží pro přirozené nebo reálné proměnné. Nejčastěji používané jsou v následujícím seznamu:

  1. Skalár = číslo
     (a) `a=5;`

(b) `b=%pi;` // %pi vypíše hodnotu π, tzn. b=3,1415...

2. vektor - sloupcový nebo řádkový
   (a) Řádkový vektor:
      i. `radkovy_vektor1=[5 9 41];` //sloupce odděluje mezera.
      ii. `radkovy_vektor2=[5,9,41];` //sloupce odděluje i čárka. Z důvodu přehlednosti je doporučuje způsob (i) pomocí mezer. Kód programu je čitelnější, jak bude ukázáno níže u matic.
   (b) Sloupcový vektor:
      i. `sloupcovy_vektor1=[4; 8; 7]` // středník odděluje jednotlivé řádky.
      ii. `sloupcovy_vektor2=[4 8 7]'` // vektor lze zapsat jako řádkový a transponovat. K transpozici zde slouží apostrof '.

3. matice - při tvoření nejdříve zadáváme počet řádků, poté počet sloupců.
   Nejprve se zadají čísla v řádku č. 1, poté se udělá středník a zadají se čísla na řádek 2.
   (a) `matice1=[1 2 3; 4 5 6]` // matice 2x3 prvky. Prvky v řádcích jsou odděleny mezerami.
   (b) `matice2=[1,2,3;4,5,6]` // matice rozměru 2x3. Prvky v řádcích jsou odděleny pomocí čárek.
   Při porovnání zápisu matice1 a matice2 je vidět důvod, proč je doporučován způsob (a), tedy zápis s mezerami. Zápis je přehlednější a již na první pohled je zřejmé, jak matice vypadá.
   (c) speciální typy matic: r-počet řádků, c-počet sloupců.
      i. Jednotková matice `eye(r,c)`
      `eye(4,5)` //vytvoří jednotkovou matici, kde budou 4 řádky a 5 sloupců. Na diagonále je hodnota 1. Pozor, i u čtvercové matice se musí zadávat řádek i sloupec, například eye(4,4)
      ii. Nulová matice `zeros(r,c)`
      `zeros(4,5)` //vytvoří matici rozměru 4x5, kde bude na všech prvcích hodnota 0.
      iii. Jedničková matice `ones(r,c)`
      `ones(4,4)` //vytvoří čtvercovou matici rozměru 4x4, kde bude na všech prvcích hodnota 1.
   (d) vybrání části matice - při výběru se muží vždy definovat odkud se vybírá a pak do kulatých závorek se nadefinuje oblast, tedy jméno_matice(řádek,sloupec).
      i. vybrání prvku
      `prvek=matice1(1,2)` //vybere prvek z matice1 a to na řádku č. 1 a ve sloupci č. 2.
      ii. vybrání vektoru
      `vektor=matice1(:,2)` //sloupcový vektor
      `vektor=matice1(1,:)` //řádkový vektor
      iii. vybrání submatice
      `submatice=matice1(1:2,2:3)` //submatice z matice1, do submatice se uloží prvky, které jsou v matice1 na 1. a 2. řádku a ve 2. a 3. sloupci.
      iv. vybrání přesně definovaných prvků - pokud víme na kterém místě chceme vybrat prvky, resp. které prvky chceme vybrat

Zadání vektoru/matice pomocí dvojtečky

Pomocí dvojtečky lze vytvořit vektor posloupností. Tato funkce je velmi důležitá u samotného programování. Často se využívá i při tvoření vektorů či matic. Postup si ukážeme na následujícím příkladu:

Chceme vytvořit vektor $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{bmatrix}$. Je zde několik variant, jak tento vektor vytvořit. Pokud vynecháme klasický zápis (viz výše - řádkový vektor), který je časově náročný, je zde ještě možnost použití dvojtečky, tedy zápis:

`vektor=1:1:10`

Tímto zápisem se vlastně vytvoří vektor, který začíná číslem 1, končí číslem 10 a jednička uprostřed (mezi dvojtečkami) znamená, že každý následující prvek bude o +1 větší. Pokud chceme vytvořit vektor nějaké posloupnosti, je lepší než vypisování hodnot použít tento zápis podle následující schématu:

`xyz=začátek:krok(inkrement):konec`

Další příklady, které se doporučují vyzkoušet přímo ve Scilabu:

- `1:2:14`

- `2:6:26`
- `30:-1:15`
- `16:-4:1`

Operace s proměnnými

S proměnnými nemusíme pracovat jako s celkem, ale například potřebujeme změnit vybraný prvek nebo vybrat jen část matice. Takových možností je spousta a až při práci zjistíme, kterou činnost potřebujeme nejvíce. Následuje seznam nejčastěji používaných operací

1. **Výběr z matice:**

   Například máme matici $M = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$. Z nadefinované matice vybereme její různé části:

   (a) výběr prvku matice - *proměnná=název_ matice(r,c)* - tzn. do nové proměnné uložíme hodnotu z vybrané matice, která je na r-tém řádku a c-tém sloupci.
   `prvek=M(2,3)` ... výsledkem je hodnota 7.

   (b) výběr řádkového vektoru - *proměnná=název_ matice(r,čísla_ sloupců)* - tzn. do nové proměnné uložíme hodnoty z vybrané matice, které jsou na r-tém řádku a v sloupcích zadaných v proměnné *čísla_ sloupců*. To lze několika způsoby, nejčastěji se používá opět dvojtečka:

      i. *název_ matice(r,:)* ... vybere celý řádkový vektor, z řádku r
      `radkovy_vektor1=M(2,:)`

      ii. *název_ matice(r,1:n)* ... vybere z řádku r vektor od 1 prvku do prvku n (místo n lze zadat číslo)
      `radkovy_vektor2=M(3,1:3)`

      iii. *název_ matice(r,m:2:n)* ... vybere z řádku r vektor od m prvku do prvku n (místo m a n lze zadat číslo), ale každé druhé číslo! (je tam zadán krok (inkrement))
      `radkovy_vektor3=M(3,2:2:4)`

      iv. *název_ matice(r,m:$)* ... vybere z řádku r vektor od prvku m do konce vektoru.
      `radkovy_vektor4=M(1,3:$)`

      v. *název_ matice(r,[č1 č2 ... čn])* ... vybere z řádku r prvky *č1, č2... čn* a uloží je do vektoru
      `radkovy_vektor4=M(2,[1 2 4])`

   (c) výběr sloupcového vektoru - *proměnná=název_ matice(čísla_ řádků,c)* - tzn. do nové proměnné uložíme hodnoty z vybrané matice, které jsou v c-tém sloupci a na řádcích zadaných v proměnné *čísla_ řádků*. To lze několika způsoby, nejčastěji se používá opět dvojtečka. Postupuje se obdobně jako u výběru řádkového vektoru, jen s tím rozdílem, že nejdříve navolíme pomocí dvojteček nebo přímým zadání místa a až na druhém místě za čárkou (,) navolíme číslo sloupce.
   `sloupcovy_vektor4=M(:,2)`

   (d) výběr submatice - *proměnná=název_ matice(čísla_ řádků,čísla_ sloupců)* - nakombinuje se výběr řádkového a sloupcového vektoru
   `submatice=M(1:3,3:4)`

2. **Početní operace**

   (a) sčítání `+` nebo odčítání `-`
   - skalár - `a+b` nebo `a-b`
   - matice (vektor) - `matice1+matice2` nebo `matice1-matice2`
     Zde je základní podmínka, že `matice1` a `matice2` musí mít stejné rozměry.

   (b) násobení `*` (nebo `.*`)
   - skalár - `a*b`
   - matice (vektor)
     – násobení matic `matice1*matice2` - pokud se použije pouze `*`, násobí se mezi s sebou celé matice. Zde je základní podmínka rozměru u násobení matic.

- násobení prvků matic `matice1.*matice2` - pokud se použije `.*` násobí se mezi s sebou pouze prvky, tzn. na 1. prvku nové matice bude 1. prvek matice1 * 1. prvek matice2.
  Matice musí mít stejné rozměry.

(c) dělení `/`,`\` nebo `./`,`.\`

- skalár - `a/b` (a děleno b) nebo `a\b` (b děleno a)
- matice (vektor)
  - dělení matic - dělí se celé matice
    - `matice1/matice2` - matice1 * inverze matice2
    - `matice1\matice2` - inverze matice1 * matice2

    Zde je základní podmínka rozměru u násobení matic.
  - dělení prvků matic - dělí se pouze jednotlivé prvky mezi sebou
    - `matice1./matice2` - prvky matice1 děleno prvky matice2
    - `matice1.\matice2` - prvky matice2 děleno prvky matice1

    Matice musí mít stejné rozměry.

(d) mocnina `^`

- skalár - `a^b` ... a^b
- matice (vektor)
  - umocňování matice - matice musí být čtvercová
    `a^b` - umocní matici a na b-tou mocninu
  - umocňování prvků matice - matice nemusí být čtvercová
    `a^b` - umocní prvky matici a na b-tou mocninu

(e) odmocnina `sqrt()` nebo `^(1/b)`

- skalár - `sqrt(a)` nebo `a^(1/b)` ... druhá odmocnina a nebo b-tá odmocnina a

3. **Logické operace**

rovná se	`==`
nerovná se	`~=`
je větší	`>`
je menší	`<`
je větší nebo rovno	`>=`
je menší nebo rovno	`<=`
a (and)	`&`
nebo	`|`

4. **Práce s proměnnými**

V některých případech neznáme rozměr či typ proměnné, například pokud jsme proměnnou získali z jiných zdrojů nebo byla vytvořena někde v programu. Na zjištění této informace se používají následující funkce:

(a) zjištění velikosti (rozměru) proměnné

- `[num_radku,num_sloupcu]=size()` ... do funkce se zadá název proměnné u které chceme zjistit rozměr. První hodnota je počet řádků, druhá počet sloupců.

(b) zjištění počtu prvků v proměnné

- `length()` ... do funkce se zadá název proměnné a výstupem bude počet všech prvků. U matice $m \times n$ to bude $m \cdot n$ prvků.

(c) zjištění typu proměnné

- `type()` ... do funkce se zadá název proměnné a výstupem bude číslo, které je pro různé proměnné unikátní. Které proměnné odpovídá které číslo je uvedeno v Help (zavolá se `help type`).

(d) vymazání proměnné

- `clear` ... vymaže všechny proměnné

- `clear promenna1` ... vymaže proměnnou *promenna1*

5. **Generování dat**

   Generování náhodných čísel je nezbytnou součástí různých simulací a výpočtů. Ve Scilabu je implementován generátor pro dvě základní rozdělení, a to pro rovnoměrné rozdělní $R(0,1)$ resp. $U(0,1)$ a pro normované normální rozdělení $\mathrm{N}(0,1)$.

   (a) `rand(r,c)` resp. `rand(r,c,''uniform'')`
   Při použití tohoto generátoru, který generuje hodnoty z rovnoměrného rozdělení, se vygenerují náhodné hodnoty, které se se stejnou pravděpobností vyskytují v intervalu $(0,1)$.

   (b) `rand(r,c,''normal'')`
   Při použití tohoto generátoru, který generuje hodnoty z normálního rozdělení s nulovou střední hodnotou a rozptylem 1. Tzn., že nejvíce hodnot bude právě v okolí nodnoty 0.

   Pokud je potřeba generovat hodnoty z jiného rozdělení, je možné je naprogramovat nebo použít externí knihovny.

4 Programování ve SCILABu

Programy, které lze ve Scilabu vytvořit lze rozdělit do dvou kategorií: (i) procedura a (ii) funkce. Základní rozdíl je v tom, že pro funkci potřebujeme znát vstupní proměnné, pro proceduru nepotřebujeme. Procedura se ukládá s příponou *.sce*, funkce s příponou *.sci*. Proceduru i funkci budeme psát VŽDY ve SciNotes (případně v jiném textovém editoru) a ne na příkazový řádek v Console.

4.1 Procedura

Proceduru, bychom mohli psát přímo v příkazovém řádku. Ale pokud budeme chtít výpočet opakovat několikrát, případně uložit si, je výhodnější udělat to jako proceduru než souhrn příkazů. Typickým příkladem může být jakýkoliv výpočet s následným vykreslením grafu (vykreslováním grafů se budeme zabývat později. Důležitou funkci má soubor (procedura) *.scilab* nebo *scilab.ini*. Tyto soubory jsou načítány hned při startu a automaticky se spustí všechny příkazy a postupy zde uvedené. Ideální místo například pro nastavení startovacího adresáře, kde budeme pracovat. Více o těchto souborech je v kapitole 2.

Základní požadavky na zápis procedury:

1. jméno souboru musí končit příponou *.sce*,
2. jméno souboru se nedoporučuje začínat číslicí, znakem ($, & atd) nebo s diakritikou,
3. ve jménu souboru se nedoporučuje používat mezera (jsou případy, kdy to může vést k chybě, která se špatně hledá),
4. vždy začínáme kód komentářem, tzn. napiši `//` a popiši co daná procedura dělá a co je výsledkem. Pokud mám proceduru rozdělenou do několika uzavřených sekcí, je dobré odlišit i ty jednotlivé sekce od sebe právě komentářem, kdy stručně popiši co se v dané sekci děje,
5. komentář píši i za každý důležitý řádek, kde například popíši, co je proměnná nebo co daný řádek dělá.

Jak již bylo řečeno výše, procedura je souhrn příkazů, které lze ovšem volat opakovaně. Při zavádění jednotlivých proměnných se ovšem musí hlídat, aby nedošlo k přepisu již jiných zavedených proměnných, které budeme potřebovat dále. V některých případech, je toto možné a žádoucí (například daná proměnná již opravdu nebude potřeba, nebo se daná proměnná změní podle aktuální hodnoty). Pokud s programováním začínáme, je výhodné si každou proměnnou nazývat názvem unikátním. Zjednodušeně řečeno, nikdy nevíte, kdy budete chtít proměnnou znovu zavést.

Pro zavádění by mělo platit stále pravidlo pochopitelnosti názvu. Těžko se pochopí, co znamenal název `sat`, kdežto `strana_a_trojuhelniku` je jasnější. Samožejmě je důležité najít vyvážený poměr mezi délkou a srozumitelností, ale to je již individální požadavek.

Na následujícím příkladu si ukážeme, jak může vypadat procedura.

4.1.1 Příklad

Zadání: Naprogramujte hod šesti-strannou kostkou, kde po zadání počtu hodů se vygenerují jednotlivé hody

Postup:

1. Nejdříve si promyslíme, jaké proměnné budeme potřebovat. Určitě zde bude proměnná určující počet hodů (`n_hodu`), a hody (`skutecne_hody`). Pokud zjistíme, že potřebujeme další proměnné v průběhu vytváření programu, je výhodné je opět napsat na začátek programu / sekce, ke které se vztahují.
2. Hod kostkou pochází z rovnoměrného rozlišení a generátorem takových hodnot je funkce `rand()`.
3. `Rand()` vygeneruje hodnoty mezi 0 a 1 a my potřebujeme hodnoty 1, 2, 3, 4, 5 a 6. Z tohoto důvodu výsledek vynásobíme a zaokrouhlíme na celá čísla.

Scilab:

```
//simulace hodu kostkou
//simulace hodu kostkou, kde vstupem bude pocet hodu a vystupem uskutecnene hody
clear, close()
n_hodu=5; //pocet hodu
skutecne_hody = []; //v tomto pripade neni nutne zavadet, ale neuskodi to
skutecne_hody=floor(rand(1,n_hodu)*6)+1
//vektor skutecne_hody obsahuje jiz //jednotlive hody
//rand(1,n_hodu) ... vygeneruje n_hodu hodnot mezi 0 a 1
//rand(1,n_hodu)*6 ... hodnoty budou v intervalu (0,6)
//floor(rand(1,n_hodu)*6)+1 ... zaokrouhleni smerem dolu, tzn. hodnoty budou nabyvat
//hodnot 0, 1, 2, 3, 4, 5. Protoze potrebujeme hodnoty od 1 do 6, pričteme 1
```

4.2 Funkce

Funkce musí mít následující tvar

```
function [vystupy]=nazev_funkce(vstupy)
        <obsah funkce>
endfunction
```

Zápis funkce:

1. zápis funkce musí začínat nápisem function,
2. jméno funkce se nemusí shodovat s názvem souboru, ve kterém je funkce obsažena, ale tímto jménem se pak funkce volá.
3. jméno souboru, ve kterém jsou obsaženy funkce, končí příponou *.sci*,
4. v jednom souboru může být víc funkcí, ale musí obsahovat pokaždé na konci funkce deklaraci endfunction,
5. funkce mohou být uvedeny i do procedury (v souboru *.sce*) a to před vlastní program. Pro jiné procedury nebudou ovšem viditelné.

Pokud vytvoříme funkci, je nutné jí načíst do paměti. Je více možností, ale nejjednodušší je pomocí funkce `getd(cesta)`, kde místo cesta se uvede absolutní nebo relativní cesta k adresáři, ve kterém je daná funkce umístěna. Další možností je vytváření tzv. knihoven (librery), kde jsou sdružené funkce. I knihovna je nutno ovšem načíst. Pokud změníte jakoukoliv funkci, je nutné ji znovu načíst do paměti. Pokud tak neučiníte, bude Scilab dále pracovat s původní verzí funkce. Bližší informace jsou uvedeny v kapitole 2.

4.2.1 Příklad

Zadání:

1. Naprogramujte hod šesti-strannou kostkou, kde po zadání počtu hodů se vygenerují jednotlivé hody.
2. Naprogramujte hod dvěma šesti-strannými kostkami, kde výsledkem bude součet ok při každém hodu.

Postup:

1. Nejprve si vytvoříme funkci, která nám vygeneruje jednotlivé hody jednou kostkou (viz předchozí kapitola).
2. Vytvoříme si funkci, kde bude součet ok při hodech dvěma kostkami.
3. Zavoláme vytvořené funkce s potřebnými vstupními informacemi.

Scilab:

```
//simulace hodu kostkou s funkci
//simulace hodu kostkou, kde vstupem bude pocet hodu a vystupem uskutecnene hody

//slozitejsi funkce, lepe ji ulozit do souboru s priponou .sci
function [vystup]=kostka(vstup)
    vystup=floor(rand(1,vstup)*6)+1 //vektor vystup obsahuje jednotlive hody
endfunction

//jednoradkova, jednoducha funkce
deff('[vystup]=kostka2(vstup)','vystup=kostka(vstup)+kostka(vstup)')
// ve funkci se sectou dva hody dvěma kostkami a to n-krát (vstup-krát)

n_hodu=50;   //pocet hodu, ktere chceme provest jednou kostkou
n_hodu_2=10; //pocet hodu, ktere chceme provest pri hodu dvema kostkami

hody=kostka(n_hodu)            //volame funkci kostka
soucet=kostka2(n_hodu_2kostky) //volame funkci kostka2
```

4.3 Spouštění procedury

Pokud chci spustit běh své vytvořené procedury je zde víc možností: (i) použití tlačítka **F5**, (ii) použití ikony šipky přímo u Scinotes nebo (iii) příkazem `exec()`.

1. Nejjednodušším způsobem je tedy aktivace pomocí klávesy **F5**. Pokud použijeme tento způsob, program se před spuštěním uloží a spustí se přednastavená funkce `exec('cesta\nazev_prodedury.sce',-1)`. Tímto způsobem se procedura spustí/proběhne, ale žádné hodnoty se nevypíší (zákaz vypisování jakýchkoliv hodnot způsobuje hodnota -1 na konci, viz níže).
2. Stejný způsob je také u použití šipky - viz obrázek 8. Tento způsob je vlastně podobný jako použití funkce **F5**, ale s tím rozdílem, že se procedura neuloží. Pokud chceme proceduru uložit a spustit, použijeme klávesu vpravo od šipky (šipka se čtverečkem). Poslední typ spouštění na liště je pomocí šipky s nápisem all, tedy se uloží vše co bylo změněno a pak se spustí všechny procedury. U žádné z těchto procedur se nevypíše ani kousek kódu na obrazovku.

Obrázek 8: Spouštění procedury

3. Pokud napíšeme do příkazového řádku příkaz `exec('cesta\nazev_prodedury.sce',-1)`, spustí se procedura a nevypíše nic na obrazovku. Co se bude vypisovat (například výsledky) lze zadat právě spouštěním v příkazovém řádku. Samotné volání se liší pouze číslem v druhém parametru u funkce `exec`. Lze používat následující hodnoty
   (a) hodnota -1 - přednastavená hodnota pro obvyklé pouštění, z kódu se nevypíše vůbec nic,
   (b) hodnota 0 - z kódu se vypíše vše co není ukončeno středníkem (středníkem se neukončuje jen to, co chceme vypsat, proto je toto nastavení často používané),
   (c) hodnota 1 - vypíše se celý kód, tedy i části, které jsou ukončeny středníkem,

(d) hodnota 2 - na definovaném místě se vypíše příznak –> To je výhodné, pokud chceme zjistit, zda nějakou částí kódu proběhl program či nikoliv. Příznak do kódu vnoříme funkcí `prompt()`,

(e) hodnota 3 - vypíše se celý kód včetně příznaků.

5 Řídící struktury

Programy, které jsme si dosud ukazovali, byly jen sekvence příkazů, které by šly psát postupně i na příkazovém řádku. Pro programování by tento postup nestačil, proto se seznámíme s řídícími strukturami. Tyto struktury umožňují ovlivňovat běh programu, ale přitom nedávají žádný výsledek. Jen umožňují program větvit, cyklit či jinak měnit běh programu.

Příkladem, kdy je potřeba použít právě řídící struktury je program na výpočet rozptylu za podmínky, že rozptyl budeme počítat pouze z kladných čísel (včetně 0). Záporná čísla k další práci nepotřebujeme, ale hodí se nám zjistit, kolik záporných čísel v zadané posloupnosti čísel bylo. Vývojový diagram této úlohy ze ukázán na obrázku xxx. Řešení této úlohy je zde 5.3.2.

5.1 Podmíněný příkaz (if - else)

Klíčovým prvekm podmíněného příkazu je podmínka. Na základě této podmínky probíhá rozhodování, zda provést či neprovést příkazy, které jsou součástí programu. Podmíněný příkaz začíná příkazem **if** a končí příkazem **end**. Syntaxe příkazu **if** je

```
if podmínka then
   <příkaz>
end
```

Pokud se daná podmínka splní, bude program pokračovat příkazem, který je uvnitř podmínky. V opačném případě se vnitřní příkaz přeskočí a program bude dále pokračovat až za koncem podmínky.

5.1.1 Příklad

Zadání: Rozlište, zda číslo je záporné nebo kladné (včetně 0) - (část úlohy z úvodu řídících struktur).

Postup:

1. Vytvoříme / načteme hodnotu.
2. Vytvoříme proměnnou `zaporna_cisla=0`, která bude indikátor, zda je číslo záporné či kladné.
3. Vytvoříme podmínku - jestliže je číslo menší než 0, pak do proměnné `zaporna_cisla` uložíme hodnotu 1 (pravda).

Scilab:

```
zaporna_cisla=0;   // zaporne cislo je nastaveno na hodnotu 0, tzn. cislo je kladne
if hodnota<0 then  // jestlize je hodnota mensi jak 0
   zaporna_cisla=1 // zaporna_cisla se prepne do stavu, pravda - hodnota byla < 0
end                // ukonceni podminky if
```

Pouze podmínka **if** není vždy dostačující. Kromě rozhodnutí když podmínka platí pokračuj následujícími příkazy, lze využít i informaci, jestliže podmínka neplatí udělej toto. K tomuto slouží rozšíření podmínky **if** o **else**. Syntaxe takového příkazu je

```
if podmínka then
   <příkaz1>
else <příkaz2>
end
```

Pokud se daná podmínka splní, bude program pokračovat *příkazem1*, který je uvnitř podmínky. V opačném případě program bude pokračovat *příkazem2*. Nelze použít oba příkazy najednou, pokud se tedy splní *příkaz1*, resp. *příkaz2*, v programu se bude pokračovat až za celou podmínkou, tedy až za příkazem *end*.

5.1.2 Příklad

Zadání: Rozlište, zda číslo je záporné nebo kladné (včetně 0). Záporné číslo označte pouze indikátorem, kladné číslo uložte jako číslo do vektoru *vektor_kladnych_cisel*.

Postup:

1. Vytvoříme / načteme hodnotu.
2. Vytvoříme proměnnou `zaporna_cisla=0`, která bude indikátor, zda je číslo záporné či kladné.
3. Vytvoříme proměnnou `vektor_kladnych_cisel=[]`. `[]` znamená, že je to prázná množina (matice o rozměrech 0×0).
4. Vytvoříme podmínku - jesltiže je číslo menší než 0, pak do proměnné `zaporna_cisla` uložíme hodnotu 1 (pravda), jinak `vektor_kladnych_cisel=hodnota`

Scilab:

```
zaporna_cisla=0;   // zaporne cislo je nastaveno na hodnotu 0, tzn. cislo je kladne
vektor_kladnych_cisel=[]; // prazdna mnozina
if hodnota<0 then  // jestlize je hodnota mensi jak 0
   zaporna_cisla=1 // zaporna_cisla se prepne do stavu, pravda - hodnota byla < 0
else vektor_kladnych_cisel=hodnota; //jinak priradi do vektoru hodnotu
end                // ukonceni podminky if
```

Pokud není dostatečné rozdělení jen na dvě oblasti, lze použít **elseif**. Počet větví podmíněného příkazu není nijak omezen, přesto se **elseif** používá pouze do určitého (rozumného počtu větví). Pokud je víc větví, doporučuje se použít přepínač **select-case**, viz kapitola 5.2.

```
if podmínka then
   <příkaz1>
elseif <příkaz2> then
else <příkaz3>
end
```

5.1.3 Příklad

Zadání: Vytvořte program, který po zadání přirozeného čísla určí, ke kterému dni toto číslo patří. Předpokládáme klasické řazení Pondělí = 1, Úterý=2,...,Neděle=7.

Postup:

1. Vytvoříme/načteme hodnotu.
2. Zjistíme zbytek po dělení čísla číslem 7 (abychom získali pouze čísla od 0 do 6). Neděle bude, pokud zbytek po dělení bude 0.
3. Vytvoříme podmínku za použití **if**, **else**, **elseif**. Bude platit, že pokud hodnota==1, výsledkem bude pondělí, jinak pokud hodnota==2, výsledkem bude úterý atd.

Scilab:

```
hodnota=15; // vstupni hodnota
zbytek=hodnota-fix(hodnota/7)*7 // vypocet hodnoty zbytku po deleni 7
if zbytek==1 then     // pokud zbytek == 1, vypise se pondeli
   disp("pondeli")
elseif zbytek==2 then // pokud zbytek == 2, vypise se utery
   disp("utery")
elseif zbytek==3 then // pokud zbytek == 3, vypise se streda
   disp("streda")
elseif zbytek==4 then // pokud zbytek == 4, vypise se ctvrtek
   disp("ctvrtek")
elseif zbytek==5 then // pokud zbytek == 5, vypise se patek
   disp("patek")
elseif zbytek==6 then // pokud zbytek == 6, vypise se sobota
   disp("sobota")
else                  // jinak se vypise nedele
   disp("nedele")
end
```

5.2 Přepínač (select-case)

Pokud se podíváte na předchozí příklad, neustále se opakuje stejný postup. Určitému číslu se přiřadí určitý výsledek. V takovém případě, lze pro zjednodušení zápisu použít místo podmíněného příkazu přepínač **select**. Syntaxe takového příkazu je

```
select proměnná
   case podmínka then
      <příkaz1>
   case podmínka then
      <příkaz2>
   else
      <příkaz3>
end
```

Tento zápis je přehlednější a mnohem lépe se tam doplňují další podmínky. Používá se ovšem jen v případě, že větvení bývá jednoduché a je větší množství větví

5.2.1 Příklad

Zadání: Vytvořte program, který po zadání přirozeného čísla určí, ke kterému dni toto číslo patří. Předpokládáme klasické řazení Pondělí = 1, Úterý=2,...,Neděle=7. Stejné zadání jako u příklad u5.1.3.

Postup:

1. Vytvoříme/načteme hodnotu.
2. Zjistíme zbytek po dělení čísla číslem 7 (abychom získali pouze čísla od 0 do 6). Neděle bude, pokud zbytek po dělení bude 0.
3. Vytvoříme podmínku za použití **if**, **else**, **elseif**. Bude platit, že pokud hodnota==1, výsledkem bude pondělí, jinak pokud hodnota==2, výsledkem bude úterý atd.

Scilab:

```
hodnota=15; // vstupni hodnota
zbytek=hodnota-fix(hodnota/7)*7 // vypocet hodnoty zbytku po deleni 7
select zbytek
case 1 then     // pokud zbytek == 1, vypise se pondeli
   disp("pondeli")
case 2 then // pokud zbytek == 2, vypise se utery
   disp("utery")
case 3 then // pokud zbytek == 3, vypise se streda
   disp("streda")
case 4 then // pokud zbytek == 4, vypise se ctvrtek
   disp("ctvrtek")
case 5 then // pokud zbytek == 5, vypise se patek
   disp("patek")
case 6 then // pokud zbytek == 6, vypise se sobota
   disp("sobota")
else                  // jinak se vypise nedele
   disp("nedele")
end
```

5.3 Cyklus for

Cyklus **for** je řídící strukturou, která nám umožní opakovat určitou činnost po přesně definovaný interval (n-krát). Vhodné je ho používat například v případech, kdy potřebujeme generovat data, procházet matice/vektory, ověřovat a upravovat data, atd. Zjednodušeně řečeno, cyklus nám umožňuje opakovat část kódu kolikrát chceme/potřebujeme. Vždy však potřebujeme znát počet opakování, pokud ho neznáme, musíme použít řídící strukturu **while** 5.4.

Syntaxe pro příkaz **for** je

```
for i=začátek:krok:konec //nebo i=vektor
      <příkaz>
end
```

Ze syntaxe pro cyklus **for** je vidět, že příkaz se provede tolikrát, kolik je prvků v intervalu `začátek:krok:konec` resp. kolik je prvků v řádkovém vektoru. Proměnná `i`, která je zde definována je pouze pomocná a mění se pokaždé, když cyklus dokončí příkaz a jde znovu, tzn. v prvním cyklu je hodnota `i=začátek`, v druhém kroku `i=začátek+krok`. Takto se pokračuje až do posledního cyklu, kdy `i=začátek+n*krok`$\leq$`konec`. To znamená, že poslední cyklus může nabývat maximální hodnoty zadaného konce. Proč se automaticky poslední `i` nerovná hodnotě `konec` bude ukázáno v následující úloze, kde bude vysvětlen i postup, proč pracujeme s `i=?:?:?`.

5.3.1 Příklad

Zadání: Vyberte všechny liché prvky z vektoru `x=[4 8 5 1 6 7 8 4 7 9]`.

Postup: Pokud máme takto zadanou úlohu, je nám jasné, že nepotřebujeme vybrat všechna lichá čísla, ale čísla na lichém místě ve vektoru, v našem případě tedy je cílem získat vektor `lichy=[4 5 6 8 7]`. , potřebujeme tedy prvek `x(1)`, `x(3)`, `x(5)`, `x(7)`, `x(9)`. Z tohoto zápisu je již lépe viditelné, které prvky vektoru potřebujeme vybrat[2].

1. načteme/vytvoříme si vektor `x=[4 8 5 1 6 7 8 4 7 9]`,
2. vytvoříme prázdnou matici /vektor `lichy=[]`,
3. vytvoříme cyklus **for** - protože víme /příkazem `size()` zjistíme, kolik je prvků v matici, můžeme použít právě cyklus **for**. Víme, že nás nezajímá každý prvek matice, ale pouze prvky 1, 3, 5, 7, 9. Z tohoto důvodu lze jednoduše začít zápis funkce **for** takto:
   (a) `for i=1:2:9` - tento způsob je sice správný, ale pokud se cokoliv změní ve vektoru `x`, cyklus již nebude fungovat správně,

[2]Při malých hodnotách by nebylo potřeba používat funkci **for**, jsou jednodušší způsoby, ale na ukázku je tento příklad ideální.

(b) `for i=1:2:size(x,''c'')` - tento způsob je stejný, jako `for i=1:2:10`, ale pokud se změní rozměr vektoru `c`, změní se automaticky i hodnota funkce `size()`. Proto při programování používáme tento zápis. U tohoto zápisu je také vidět dříve zmiňované, že konec intervalu nemusí nutně znamenat číslo posledního cyklu. V tuto chvíli je `konec=10`, ale poslední cyklus se provede pro `i=9`.

4. Do cyklu vložíme zápis, aby se do proměnné `lichý` přidal i-tý prvek vektoru `x`.

Scilab:

```
x=[4 8 5 1 6 7 8 4 7 9] //zadany vektor lichy=[];

for i=1:2:size(x,"c")   //udela se pouze kazdy druhy cyklus a to po lichych prvcich
   lichy=[lichy x(i)];  //do vektoru lichy pridej i-ty prvek vektoru x
end //konec for
```

5.3.2 Příklad

Zadání: Výpočtěte rozptyl za podmínky, že rozptyl budeme počítat pouze z kladných čísel (včetně 0). Záporná čísla k další práci nepotřebujeme, ale chceme zjistit, kolik záporných čísel v zadané posloupnosti čísel bylo.

Postup:

1. Vytvoříme / načteme hodnotu.
2. Vytvoříme proměnnou `zaporna_cisla=0`, která bude indikátor, zda je číslo záporné či kladné.
3. Vytvoříme podmínku - jestliže je číslo menší než 0, pak do proměnné `zaporna_cisla` uložíme hodnotu 1 (pravda).

Scilab:

```
//vypocet rozptylu pouze z kladnych cisel
clear
hodnota=fix(rand(1,20,"normal")*10); //generator hodnot
zaporna_cisla=0;          //pocatecni hodnota/pocet zaporneho_cisla
vektor_kladnych_cisel=[]; //prazdna mnozina

for i=1:1:size(hodnota,     //uděláme tolik cyklu, koik je sloupcu ve vektoru hodnota
   if hodnota(i)<0 then             //jestlize je hodnota mensi jak 0
      zaporna_cisla=zaporna_cisla+1 //zvysi se zaporna_cisla o 1
   else //jinak priradi do promenne vektor_kladnych_cisel i-ty prvek z hodnota
      vektor_kladnych_cisel=[vektor_kladnych_cisel hodnota(i)];
 end    //ukonceni podminky if
end       //ukonceni cyklu for

rozptyl=variance(vektor_kladnych_cisel) //vypocet rozptylu pouze z
vektoru_kladnych_cisel
```

5.4 Cyklus while

Cyklus **while** je řídící strukturou podobnou jako **for**, ale na rozdíl od **for** není definován přesně konec cyklu. Cyklus se opakuje dokud je splněna podmínka. Syntaxe pro příkaz **while** je

```
While podmínka do
   <příkaz>
end
```

Nevýhoda totoho cyklu je ta, že špatné nastavení podmínky může vést k zacyklení celého programu. To vede k nekonečnému počítání. Například, pokud bychom chtěli, aby program fungoval do té doby, dokud bude platit `while i>10`. Začneme na hodnotě 15 a budeme tvrdit, že v každém cyklu se hodnota `i` zvýší o hodnotu +1. Tedy zápis bude

```
i=15;
while i>10 do
   disp('funguje to');
   i=i+1;
end
```

Pokud tento program spustíme, hodnota `i` se bude zvyšovat a podmínka v cyklu bude platit stále. Z tohoto důvodu je potřeba dávat si pozor na zápis podmínky u cyklu while.

5.4.1 Příklad

Zadání: Vytvořte program, který bude do vektoru *vyber_ cisel* vybere všechna kladná čísla (včetně 0) než nastoupí první záporné číslo.

Postup:

1. Vytvoříme/načteme vektor.
2. Nastavíme počítač cyklů `i=1`.
3. Protože nevíme, které číslo je první záporné použijeme funkci **while**. Tato funkce bude mít hned dvě podmínky, první pro stav, kdy hlídáme, zda je číslo kladné či nikoliv a druhá podmínka je podmínka, pokud budou všechny čísla kladné, tak skonči na konci vektoru.
4. Do **while** dáme příkaz, ať se uloží vybraná hodnota do vektoru `vyber_cisel`, ale zároveň ještě po každém proběhnutém cyklu zvýšíme počítač cyklů `i` o hodnotu 1.

Scilab:

```
vektor=fix(rand(1,20,"normal")*10); //vytvorime vektor i=1
while i<=10 & vektor(i)>=0 do       //delej dokud i≤10 a zaroven hodnota vektoru je ≥0
   vyber_cisel(1,i)=vektor(1,i);    //na i-ty prvek vektoru vyber_cisel dam vektor(i)
   i=i+1;      //pokud tam nedame toto, bude while porovnavat stale prvni hodnotu
end            //konec while
```

5.5 Rady k programování

Pokud začneme programovat složitější úlohy zjistíme, že některé „triky" nám umožní napsat kód elegantněji a lépe.

5.5.1 Prázdná množina

Pokud potřebujeme vytvořit kód, kdy se vybranná proměnná bude rozšiřovat o prvky, vektory nebo matice je vhodné na začátek použít práznou množinu. Ve Scilabovské syntaxi se používá tento zápis `promenna=[]`. Právě `[]` znamená, že se jedná o prázdnou množinu.

Vytvoříme jednoduchý příklad, kdy budu mít hodnoty $A = \{4\ -4\ 6\ 8\}$ a budu chtít vytvořit jejich třetí mocninu.

```
treti_mocnina=[];          //vytvořím prázdnou množinu
for i=[4 -6 8 10] do       //dělej pro následující hodnoty
   treti_mocnina=[treti_mocnina i^3];  //treti mocnina se rozšíří o proměnnou i^3.
end            //konec for
```

Pokud bychom nepoužili prázdnou množinu, museli bychom rozdělit program na první prvek a ostatní prvky pomocí `if-else`.

5.5.2 Počáteční hodnota proměnné

Pokud potřebujeme mít vlastní sčítač, tedy počítáme, kdy daná situace nastala/nenastala, používá se přičítání v proměnné o číslo 1. Opět, stejně jako u prázdné množiny, musíme nastavit počáteční hodnotu na námi předdefinovanou hodnotu (nejčastěji 0), tedy `promenna=0`. K této hodnotě se bude přičítat hodnota podle zadání.

Rozšíříme předchozí příklad o to, abychom mohli počítat, kolik prvků bylo ve vektoru

```
treti_mocnina=[];          //vytvořím prázdnou množinu
pocet_prvku=0;             //počáteční počet prvků ve vektoru je 0
for i=[4 -6 8 10] do       //dělej pro následující hodnoty
   treti_mocnina=[treti_mocnina i^3];  //treti mocnina se rozšíří o proměnnou i^3.
   pocet_prvku=pocet_prvku+1;  //každým proběhnutým cyklem se hodnota zvětší o +1
end            //konec for
```

Pokud bychom na začátku nenadefinovali proměnnnou `pocet_prvku`, na řádku 4 by se objevila chyba, protože na prave straně je právě tato proměnná. Pokud bychom tento způsob nechtěli použít, museli bychom pro první cyklus vytvořit speciální podmínku pomocí `if-else`.

6 Vykreslování grafů

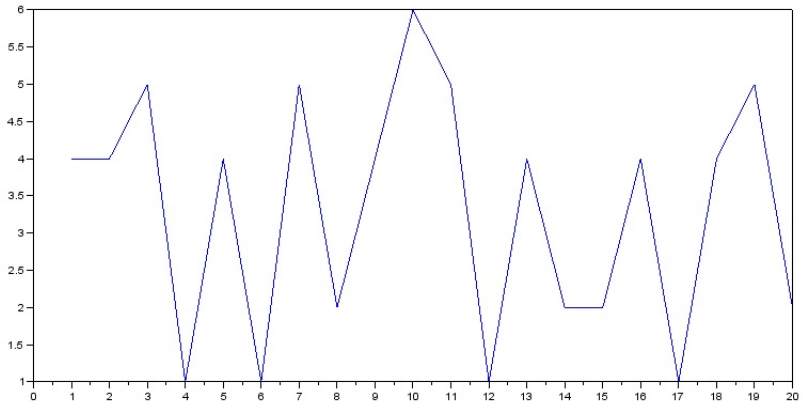
Pro vyhodnocování výsledků se v mnoha případech hodí grafické vyhodnocení. V následující kapitole si ukážeme, jaké jsou základní možnosti vykreslení grafu. Pro bližší a detailnější práci s grafy je potřeba použít nápovědu ve Scilabu.

6.1 Spojnicového grafu

6.1.1 Vykreslení

Spojnicový graf (dále pouze graf) je graf, který se používá nejčastěji. Pod pojmem spojnicový budeme předpokládat graf, který je vykreslen buď spojnicí nebo i body. Pro vykreslení grafu se používá funkce `plot()`. Pokud vezmeme již výše zmíněný příklad s hodem kostkou, můžeme vykreslit například 20 hodu kostkou

```
n_hodu=20;           //pocet hodu
hody=kostka(n_hodu) //vygenerovane hody
plot(hody            //vykresleni grafu
```

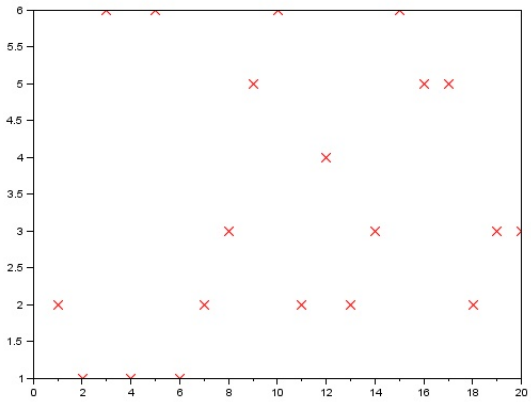


Obrázek 9: Hod kostkou - vykresleni

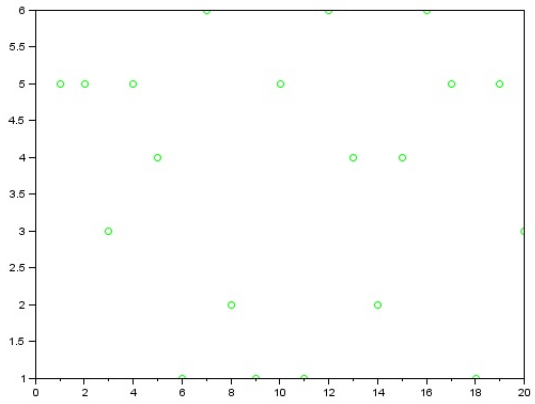
Graf číslo 9 ukazuje nejjednodušší vykreslení výsledků. Pokud budeme chtít změnit barvu či zrušit spojnice a místo toho body, tyto informace napíšeme přímo k vykreslení grafu. Například pro hody použijeme červené křižky nebo zelená kolečka, viz obrázek 10.

Samozřejmě, že lze i libovolně měnit barva u spojnicového grafu. Pokud máme zápis `'rx'` znamená to, že chceme vykreslit červeně (r) a body budou křížky (x). Podobně tedy pracujeme i dále. Pokud budeme chtít červený spojnicový graf, napíšeme funkci `plot(hody,'r')`.

Další typy čar, bodů i barev jsou uvedeny v help Scilabu - `help LineSpec`.



`plot(hody,'rx')`



`plot(hody,'go')`

Obrázek 10: Hod kostkou - vykreslení2

6.1.2 Vykreslení více grafů

Pokud chceme vykreslit více grafů, je důležité si nejdříve uvědomit, zda je chceme vykreslit (i) do sebe (vše do jednoho grafu) nebo (ii) zvlášť (každý graf má své okno).

Vykreslní do jednoho grafu Pokud chceme vykreslit obrázky do jednoho grafu, stačí zavolat víckrát funkci plot, například

Vykreslení do několika grafů Pokud chceme vykreslit obrázky do několika krafů, vykreslíme nejdříve jeden, pak napíšeme funkci `scf` přesněji `scf(číslo)`, která nám otevře další okno (figure) a poté vykreslíme graf druhý. Tento postup můžeme opakovat několikrát. Samožejmě lze vykreslit například dva grafy do sebe a třetí zvlášť, tedy nejdříve vykreslíme první dva grafy, zadáme `scf` a pak vykreslíme poslední graf.

Příklad Vykreslete hody kostkou: 1. do jednoho grafu vykreslete 20 hodů tak, aby hody byly červený křížek a další hody byly modrá čára, 2. vykreslete pouze deset hodů pomocí čárkované čáry.

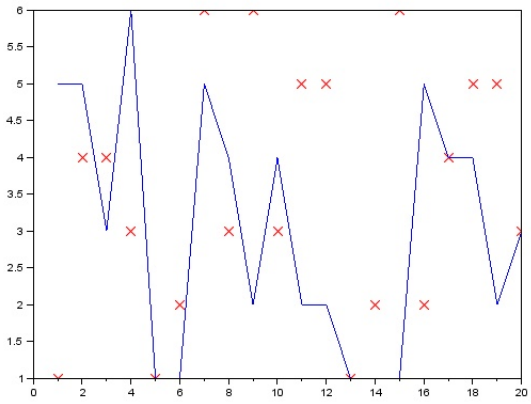
```
//vykresleni grafu
n_hodu=20;            //pocet hodu
hody=kostka(n_hodu)   //vygenerovane hody
hody2=kostka(n_hodu) //vygenerovane hody2
hody3=kostka(10)

// grafy
plot(hody,'rx')  //vykresleni promenne hody
plot(hody2)      //vykresleni promenne hody2
scf              //otevreni noveho okna
plot(hody3,'--') //vykresleni promenne hody2
```

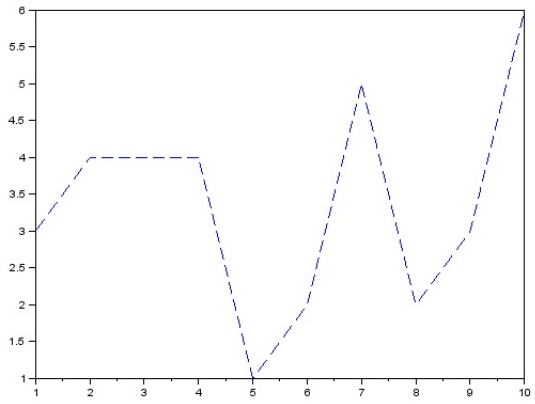
6.1.3 Popis grafu

Samostatný graf nikomu nic neřekne, proto je důležité ho popsat. V této části se budeme věnovat vytváření (í) nadpisu, (ii) legendy a (iii) popisu os.

1. **Titulek** - pokud chceme graf popsat, za vykresleni grafu dáme příkaz `title('titulek grafu')`.
2. **Legenda** - při zápisu více grafů do jednoho obrázku je vhodné pojmenovat jednotlivé grafy. Nejjednodušší je použít legendu. Legenda vzhnikne, pokud za vykreslení grafů dáme příkaz `legend('jméno1','jméno2',...)`.



Vykresleni 2 grafu do 1 obrázku



Vykreslení dalšího grafu

Obrázek 11: Hod kostkou - vykreslení3

Ke každému grafu dáme do apostrofů název, který chceme použít. V pravém horním rohu se objeví legenda s typem čar (bodů) a názvem.

3. **Popis os** - při popisu os se používá různý zápis pro popis x-ové a y-ové osy. Pro popis osy x použijeme `xlabel("co je na ose x")` a pro popis osy y, tedy obdobně `ylabel("co je na ose y")`.

Každý tento popis lze samozřejmě měnit a to jak do orientace, tak do velikosti, typu písma, barvy písma atd. Takové nastavení se dělá rovnou v příslušné funkci. Budeme-li chtít změnit nadpis, aby byl větší a zároveň červený, k tomu popsané osy a legenda, napíšeme

```
plot(hody,'rx') //vykresleni promenne hody
plot(hody2)     //vykresleni promenne hody2
title('Porovnání hodů','fontsize',5,'color','red') //nadpis
legend('jedna','dva') //legenda
xlabel("číslo hodu")  //popis osy x
ylabel("počet ok")    //popis osy y
```

Výsledný grat tedy může vypadat jak je vidět na obrázku 12.

Popis os vypadá mnohem líp, pokud se použíje zápis v LaTex stylu. Je jen potřeba, aby začátek a konec byl označen jako v LaTexu, tzn. například a^2+b_2.

6.1.4 Umístění grafu

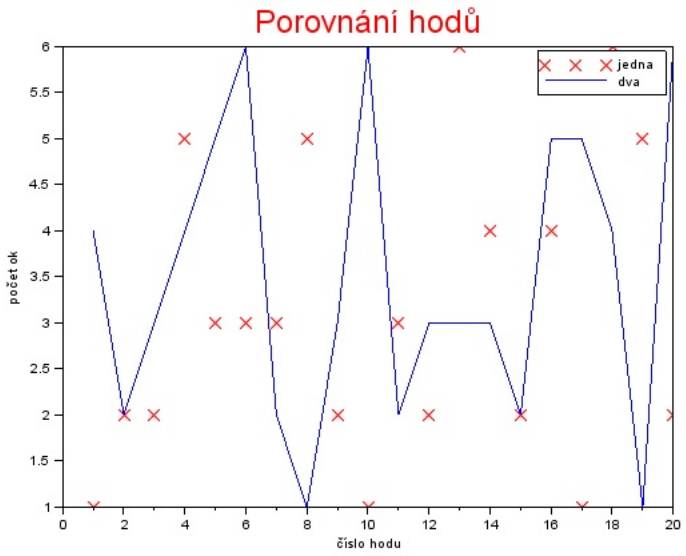
Pokud chceme nastavit meze pro vykreslení os, jejich velikost atd. je potřeba použít následující postup. Po vykreslení grafu přiřadíme přoměnné funkci `gdf()`. Tato funkce umožní různé nastavení os. Pokud budeme chtít měnit osy, budem postupovat takto:

1. `b=gdf();`
2. `b.position=[x y]` ... kde dvojice $\{x, y\}$ určuje umístění levého horního rohu

6.1.5 Nastavení os

Pokud chceme nastavit meze pro vykreslení os, jejich velikost atd. je potřeba použít následující postup. Po vykreslení grafu přiřadíme přoměnné funkci `gca()`. Tato funkce umožní různé nastavení os. Pokud budeme chtít měnit osy, budem postupovat takto:

1. `a=gca();`
2. `a.yyy` ... místo yyy se zada vlastnost, kterou chceme zmenit, viz help - axes properties



Obrázek 12: Hod kostkou - popis grafu

Některá možná nastavení os

- `a.data_bounds=[min(x) max(x) min(y) max(y) min(z) max(z)]; //omezení vykreslení hranic grafu. Pokud nemáme osu z, neudáme žádné hodnoty,`
- `a.x_location = "xxx"; //misto xxx se da bottom, top, middle nebo origin .. umožní to umístit osy jinam,`
- `a.axes_visible="off"; //nebudou zobrazeny osy vůbec. Přednastavená je hodnota on,`
- `a.margins=[0.125 0.125 0.125 0.125] //velikost okrajů grafu. Rozměr mezi hodnotou 0 a 1,`
- `a.grid=[1,1]; //vytvoří mřížku v grafu`

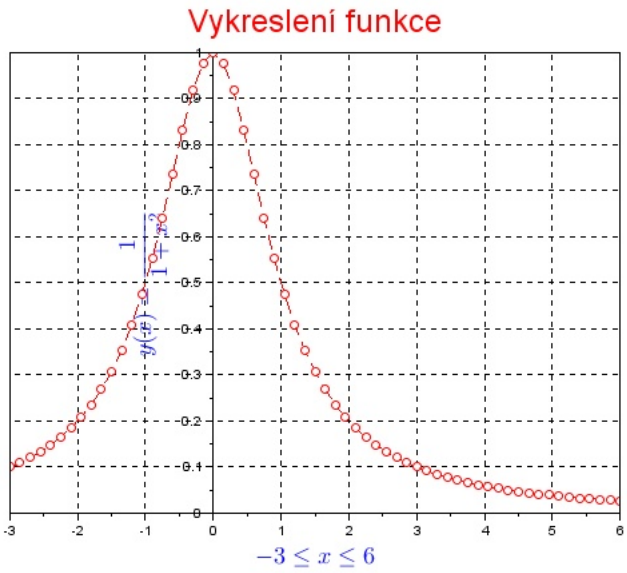
Další možnosti lze získat nápovědě Scilabu (axes properties).

Příklad ukázky grafu, kde jsou použity některé vlastnosti grafu, výsledek je vidět na obrázku 13. Při posunu osy x nebo y se může stát, jako v tomto případě, že popis osy může být nečitelný, protože je v ose a grafu.

```
clear
x=linspace(-3,6,61); //rozdeli na 61 dilku interval -3 a 6
y=1 ./(1+x.^2); //y zavisi na x

plot(x,y,'ro--');
xlabel("$-3\le x\le 6$","fontsize",4,"color","blue"); //popis osy x pomoci LaTex
ylabel("$y(x)=\frac{1}{1+x^2}$","fontsize",4,"color","blue"); //popis osy y pomoci
LaTex
title("Vykreslení funkce",'fontsize',5,'color','red') //titulek

a = gca();   //nastavení grafu
a.grid=[1,1] //mrizka v grafu
a.x_location = "origin"; //vykresleni osy X v 0 + popis popisu osy
a.y_location = "origin"; //vykresleni osy X v 0 + popis popisu osy
```



Obrázek 13: Vykreslení funkce $y = \frac{1}{1+x^2}$

6.2 Vykreslení histogramu

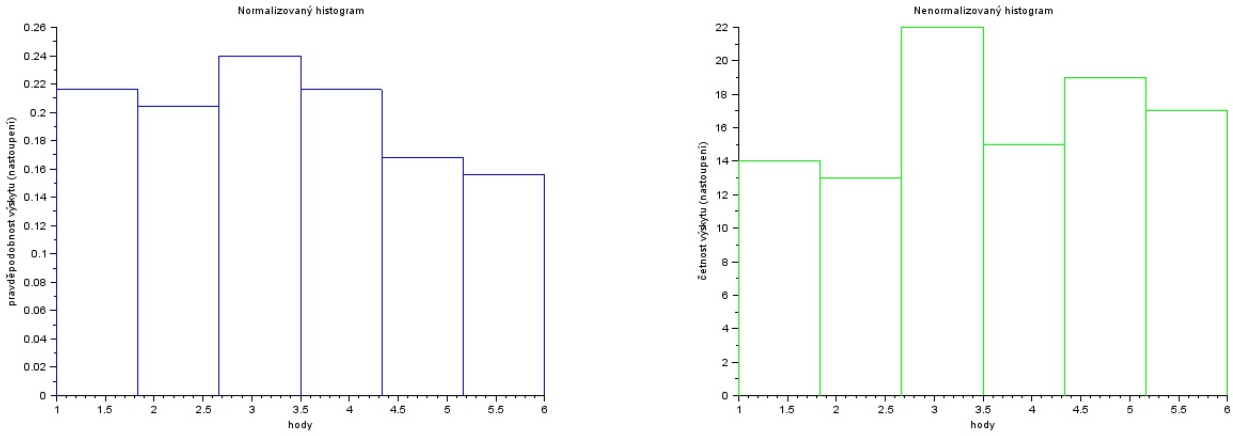
Histogram je graf četností. Často se hodí pro zjištění, zda námi zadané předpoklady jsou pravdivé. Příkladem může být právě hod kostkou, kdy histogramem můžeme potvrdit či vyvrátit zákon velkých čísel. Histogram lze vykreslit normalizovaný a nenormalizovaný. Normalizovaný znamená, že na ose y bude pravděpodobnost výskytu (nastoupení), nenormalizovaný znamená, že na ose y je skutečná četnost výskytu (nastoupení). Histogram pro 100 hodů kostkou lze vykreslit tímto způsobem:

```
hody4=kostka(100)

//normalizovaný histogram
histplot(6,hody4,style=2)
       //vykresleni histogramu, 6 mnozin, data z hody4, graf bude modry
b=gca(); //nastaveni os
b.grid=[-1 -1]; //vypnuti mrizky
xlabel("hody")
ylabel("pravděpodobnost výskytu (nastoupení)")
title('Normalizovaný histogram')

scf(2)
//nenormalizovany histogram
histplot(6,hody4,normalization=%f, style=3) //vykresleni histogramu, graf nebude
normalizovan, zelený
b=gca(); //nastaveni os
b.grid=[-1 -1]; //vypnuti mrizky
xlabel("hody")
ylabel("četnost výskytu (nastoupení)")
title('Nenormalizovaný histogram')
```

Vykreslené grafy budou vypadat takto



Obrázek 14: Histogram

6.3 Vykreslení 3D grafu

Pro některá zadání není vykreslení pomocí 2D grafu dostatečně přehledné. Proto se přechází k vykreslení pomocí 3D grafu. Na následujícím jednoduchém příkladě je ukázán jednoduchý způsob vytvoření 3D grafu. Ve většině případů se dodržuje následující postup:

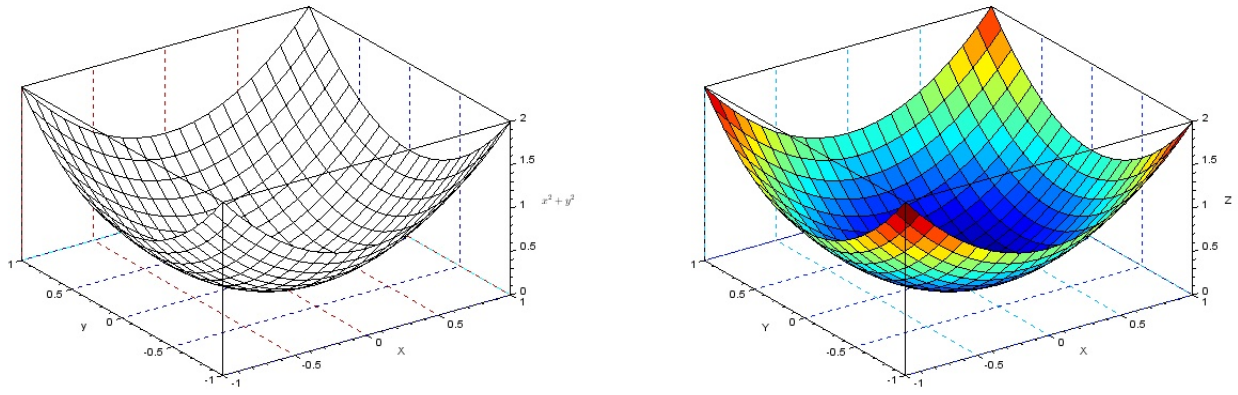
1. vytvoříme/získáme data x,
2. vytvoříme/získáme data y,
3. vytvoříme síť bodů x a y - `meshgrid()`,
4. ke každému společnému bodu na síti se přidá bod z,
5. vykreslí se graf.

```
close
x=-1:0.1:1; //data x
y=-1:0.1:1; //data y
[X,Y]=meshgrid(x,y); //vytvoreni site bodu
Z=X.^2+Y.^2; //data z

mesh(X,Y,Z) //vykresleni 3D grafu
xlabel("X") //popis osy x
ylabel("y") //popis osy y
zlabel("$x^2+y^2$") //popis osy z

scf
xset("colormap",jetcolormap(64)); //zadani barev pro barevny graf
surf(X,Y,Z); //vykresledni 3D barevneho grafu
```

Vykreslené grafy budou vypadat takto



Obrázek 15: Graf 3D