Úvod

Pane profesore, co jsou to vlastně čísla?
Abych pravdu řekl, Kropáčku, tak to úplně přesně nevím.
Opravdu nevíte?
Ne, vlastně si ani nejsem jistý, jestli vůbec existují. Ale jinak toho o nich vím poměrně hodně.

Pokud se zabýváme čísly, zpravidla se nevyhneme tomu, že někdy potřebujeme i něco spočítat. Vlastně jsme se učili nejdřív něco spočítat a až následně se občas najde nějaký šťoura, který se ptá, co to vlastně čísla jsou. V tomto textu se budeme věnovat spíš praktickému počítání, přičemž teoretické předpoklady budou někde v pozadí, ze kterého je v případě potřeby vytáhneme.

Počítání nám velmi usnadňuje technika. Zpravidla si bereme na pomoc kalkulačku i při sčítání poměrně malých čísel, i když s trochou snahy bychom snad ještě uměli ručně i dělit. Nejefektivnějším pomocníkem může být samozřejmě počítač, se kterým kromě rippování CD, kódování filmů a úpravy digitálních fotek je možné pomocí vhodných prostředků i něco spočítat.

Pro potřeby tohoto kursu budeme využívat hlavně dva softwarové prostředky - jsou to matematické programy Matlab a Sage. První z nich komerční a je orientován hlavně numericky, druhý je volně dostupný a zvládá i symbolické výpočty. Takže jejich vhodnou kombinací dokážeme pokrýt prakticky všechny výpočty, které budeme potřebovat. Je ovšem třeba brát v úvahu i vývoj tohoto software, proto některé některé zde uváděné informace mohou být časem zastaralé nebo neplatné. Nechť mi to laskavý čtenář promine. O dalších výpočetních prostředcích se zmíníme jen okrajově.

KAPITOLA 1

První pokusy

1. Praktické výpočty

Kropáčku, co se s tím tak páráte, vy to ještě nemáte?
Mám, pane profesore, už jsem to vypočítal.
A kolik vám to vyšlo?
Čtyřicet dva[1].
Čtyřicet dva? To je divné. Co jste to vlastně počítal?
To se právě snažím zjistit.

Učitelé matematiky na všech úrovních se s tím setkávají poměrně často. Zadají příklad a student nebo žák okamžitě začne počítat za použití nějakých vzorečků. Netvrdím, že výpočet musí být vždy špatně, ale zamyslet se nad problémem ještě předtím, než se na něj vrhnu se vzorečky a s kalkulačkou, je zpravidla prospěšné. Často nám to může pomoci odhalit chyby, kterých se při výpočtu dopustíme. Například když nám vyjde objem nějakého tělesa jak komplexní číslo nebo teplota materiálu $-358.16°C$.

Při používání techniky je ovšem třeba mít na paměti ještě další věc, a tou jsou její omezené možnosti. Při převážné většině výpočtů se k hranicím, které výpočetní technika má, nepřiblížíme, ale je dobré o nich vědět. Tyto hranice jsou dány zejména omezeným prostorem, který je pro reprezentaci čísel vymezen. Nejlépe si to ukážeme na následujícím příkladu.

Příklad 1. Čísla jsou v počítači uložena ve dvojkové soustavě, takže i jednoduchá desetinná čísla, např. 0,1, jsou při standardním způsobu ukládání uložena nepřesně, jelikož ve dvojkové soustavě je číslo 0,1 zapsáno pomocí nekonečné řady:

$$0,1 = 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + 2^{-16} + 2^{-17} + \cdots$$

a tuto řadu musíme někde useknout. V Matlabu se dá snadno zjistit, velikost chyby, která se v tomto případě do výpočtu může vloudit:

[1] Viz též Douglas Adams: Stopařův průvodce po Galaxii

```
>> c=0;
>> for k=1:100, c=c+0.1; end
>> 10-c
ans =
     1.9540e-14
>>
```

Nebo elegantněji

```
>> c=0.1*ones(1,100);10-sum(c)
ans =
     1.9540e-14
>>
```

Chyba je sice nepatrná, ale je tam. Často je možné tuto chybu zmenšit správným postupem při výpočtu. V uvedeném příkladu stačí celkový součet rozdělit na několik částečných součtů. To můžeme udělat tak, že stoprvkový vektor `c` přeskládáme do čtvercové matice řádu 10 (pokud ještě nevíte, co je to matice, tak si ji můžete představit jako tabulku, v našem případě o deseti řádcích a deseti sloupečcích), pak sečteme nejprve jednotlivé sloupce a následně získané mezivýsledky:

```
>> c=0.1*ones(1,100);A=reshape(c,10,10);
>> 10-sum(sum(A))
ans =
     1.7764e-15
>>
```

Vidíme, že chyba je zhruba desetkrát menší než při přímém součtu. Tuto skutečnost můžeme zdůvodnit následující úvahou. Předpokládejme, že číslo 0,1 je v počítači uloženo s chybou ε. Pokud sčítáme sto stejných hodnot, na konci výpočtu připočítáváme číslo 0,1 k číslu přibližně stokrát většímu, Protože zaokrouhlení výsledku se bere podle většího čísla, hodnota 0,1 se přičte s chybou zhruba 100ε. Je to podobné, jako bychom počítali s přesností na tři platné číslice a K číslu 111 přičítali 1,11. Celková chyba bude řádově v tisících ε, záleží na konkrétním čísle a „*chytrosti*“ software. Pokud ovšem sčítáme jen deset hodnot, je chyba řádově stokrát menší, při dalším sčítání mezivýsledků se sice zvětší, ale nedosáhne takové hodnoty jako při přímém sčítání.

Při použití programu Sage si často můžeme snadno zvolit, jestli výpočet proběhne symbolicky nebo numericky. V případě symbolického výpočtu zadáváme konstanty ve formě zlomků, pro numerický výpočet použijeme desetinný zápis:

```
sage: M=matrix(1,[1/10 for i in range(100)])
sage: 10-sum(sum(M))
0
```

```
sage: M=matrix(1,[0.1 for i in range(100)])
sage: 10-sum(sum(M))
1.95399252334028e-14
sage: M=matrix(10,[0.1 for i in range(100)])
sage: 10-sum(sum(M))
1.77635683940025e-15
```

Také v Matlabu je možné zobrazovat hodnoty ve formě racionálních čísel, umožní to příkaz `format rat`. V tomto případě se ale jedná pouze o aproximace čísel uložených v počítači standardním způsobem. Takže pokud třeba potřebujeme aproximovat číslo π zlomkem, můžeme zadat

```
>> format rat
>> pi
ans =
     355/113
>>
```

Matlab i Sage umí samozřejmě pracovat s komplexními čísly. V Matlabu slouží jako imaginární jednotka symbol `i`, na zjištění goniometrického tvaru komplexního čísla můžeme použít funkce `abs` a `angle`:

```
>> z=1+i
>> abs(z)
ans =
    1.4142
>> angle(z)
ans =
    0.7854
>>
```

Poslední uvedená hodnota je numericky $\pi/4$.

Pro zadání komplexních čísel v Sage použijeme imaginární jednotku označenou `I`, na zjištění goniometrického tvaru funkce `argument` a `abs`. Přitom funkci `argument` je potřeba použít na numerickou aproximaci komplexního čísla, a použití je následovné:

```
sage: abs(1+I)
sqrt(2)
sage: numerical_approx(1+I).argument()
```

```
0.785398163397448
sage: (1+I).numerical_approx().argument()
0.785398163397448
```

Patrně si pozorný čtenář už povšiml, že funkce `numerical_approx` je možné použít dvěma způsoby: buď obvykle, kdy jak argument použijeme výraz, jehož aproximaci chceme vypočítat, nebo až za výrazem s tečkou jako oddělovačem. Tento dvojí způsob aplikace funkce je v Sage celkem obvyklý. Funkce `numerical_approx` má navíc ještě další nepovinné parametry. Pro nás jsou zajímavé hlavně ty, které mohou specifikovat, s jak velkou přesností se má výraz zobrazit. Přirozené číslo jako další parametr určuje, kolik bitů se má pro aproximaci použít, nebo je možné specifikovat pomocí slova `digits`, kolik číslic se má vypsat. A pokud se někomu nechce vypisovat celý název funkce, může použit zkrácené názvy, které jsou `n` nebo `N`:

```
sage: numerical_approx(pi)
3.14159265358979
sage: n(pi)
3.14159265358979
sage: pi.N()
3.14159265358979
sage: pi.N(100)
3.1415926535897932384626433833
sage: N(pi,100)
3.1415926535897932384626433833
sage: n(pi,digits=20)
3.1415926535897932385
sage: pi.n(digits=20)
3.1415926535897932385
```

Příklad 2. Podíváme se, jak si softwarové prostředky poradí s řešením rovnice $x^3 + x^2 - 2x - 1 = 0$ z příkladu 1.2. Jedná se o hledání kořenů polynomu třetího stupně, neboli kubického polynomu. V Matlabu se polynomy zadávají pomocí koeficientů od nejvyšší mocniny, všechny koeficienty tak tvoří vektor, který je v Matlabu definován pomocí hranatých závorek. Kořeny polynomu pak zjistíme pomocí funkce `roots`:

```
>> p=[1 1 -2 -1];
>> roots(p)
ans =
   -1.8019
    1.2470
```

```
   -0.4450
>>
```

V Sage slouží pro řešení rovnic funkce `solve`, ale příkaz

```
sage: x=var('x')
sage: solve(x^3+x^2-2*x-1,x)
```

dá výsledek

$$
\begin{aligned}
x &= -\frac{1}{2}\left(\frac{7}{18}i\sqrt{3}+\frac{7}{54}\right)^{\left(\frac{1}{3}\right)}\left(i\sqrt{3}+1\right)-\frac{-7i\sqrt{3}+7}{18\left(\frac{7}{18}i\sqrt{3}+\frac{7}{54}\right)^{\left(\frac{1}{3}\right)}}-\frac{1}{3},\\
x &= -\frac{1}{2}\left(-i\sqrt{3}+1\right)\left(\frac{7}{18}i\sqrt{3}+\frac{7}{54}\right)^{\left(\frac{1}{3}\right)}-\frac{7i\sqrt{3}+7}{18\left(\frac{7}{18}i\sqrt{3}+\frac{7}{54}\right)^{\left(\frac{1}{3}\right)}}-\frac{1}{3},\\
x &= \left(\frac{7}{18}i\sqrt{3}+\frac{7}{54}\right)^{\left(\frac{1}{3}\right)}+\frac{7}{9\left(\frac{7}{18}i\sqrt{3}+\frac{7}{54}\right)^{\left(\frac{1}{3}\right)}}-\frac{1}{3}
\end{aligned}
$$

Je zřejmé, že z tohoto výsledku se nedá při nejlepší vůli usoudit, že imaginární část je ve skutečnosti nulová. K získání numerické hodnoty řešení můžeme použít funkci `numerical_approx()`:

```
sage: x=var('x')
sage: S=solve(x^3+x^2-2*x-1.0,x)
sage: x0=S[0].right()
sage: x0.numerical_approx()
-0.445041867912629
```

takže všechna řešení dostaneme např. takto:

```
sage: x=var('x')
sage: S=solve(x^3+x^2-2*x-1.0,x)
sage: x0=S[0].right().numerical_approx()
sage: x1=S[1].right().numerical_approx()
sage: x2=S[2].right().numerical_approx()
sage: x0;x1;x2
-0.445041867912629
-1.80193773580484
1.24697960371747 + 5.55111512312578e-17*i
```

Zdálo by se, že na hledání kořenů polynomů bude Matlab lepším prostředkem než Sage, ale stačí zkusit najít kořeny polynomu

$$p(x) = (x-1)^4 = x^4 - 4x^3 + 6x^2 - 4x + 1,$$

nastavit větší přesnost zobrazování výsledků a budeme rychle vyvedeni z omylu:

```
>> format long
>> p=[1 -4 6 -4 1];
>> roots(p)
ans =
  1.000217162531685
  0.999999969335974 + 0.000217131858872i
  0.999999969335974 - 0.000217131858872i
  0.999782898796371
>>
```

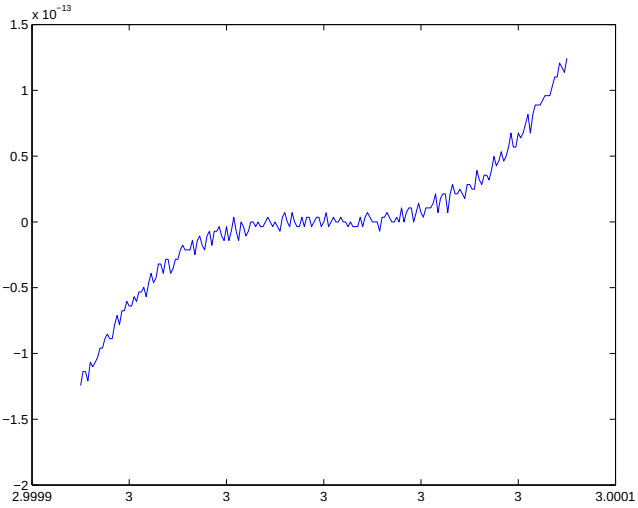
Jak se dá čekat, Sage si s tímto problémem poradí podstatně lépe.

```
sage: solve(x^4-4*x^3+6*x^2-4*x+1==0,x)
[x == 1]
>
```

V posledním příkladu v této kapitole si ukážeme typický problém při numerických výpočtech, kterým je ztráta přesnosti při počítání s různě velkými čísly.

Příklad 3. Mějme polynom $P(x) = (x-3)^3 = x^3 - 9x^2 + 27x - 27$. Předpokládejme, že potřebujeme hodnoty polynomu v malém okolí bodu 3, délka okolí bude 0,00001. Výsledek si zobrazíme graficky

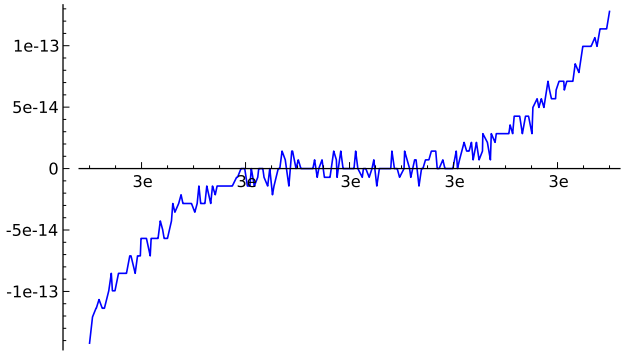
```
>> P=[1 -9 27 -27];
>> x=linspace(2.99995,3.00005,201);
>> y=polyval(P,x);
>> plot(x,y)
```



Výsledkem na daném intervalu by měla být čísla řádově 10^{-15}, protože při výpočtu používáme čísla řádově v desítkách, je chyba zhruba desetkrát větší. V tomto případě i Sage počítá numericky, protože příkaz

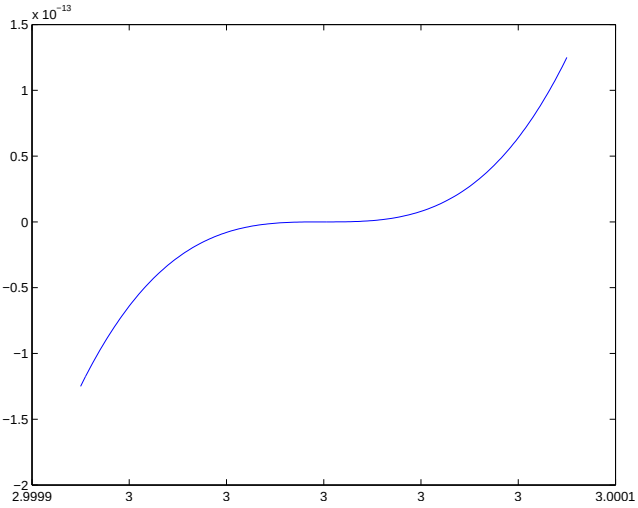
```
sage: g=plot(x^3-9*x^2+27*x-27,2.99995,3.00005);g
```

vyprodukuje následující obrázek:



Chybu ovšem můžeme výrazně snížit, pokud použijeme první vztah pro výpočet polynomu:

```
>> x=linspace(2.99995,3.00005,201);
>> y=(x-3).^3;
>> plot(x,y)
```



Závěrem z této kapitoly nechť je vědomí, že ani velmi kvalitní výpočetní prostředek (zatím?) nenahradí člověka a vždycky je dobré se alespoň trochu zamyslet nad tím, jaké nám to vlastně počítač dává výsledky, co s nimi chceme dále provádět a jestli by to nešlo spočítat trochu přesněji.

2. Kombinatorické výpočty

Základním výpočtem v kombinatorice je výpočet faktoriálu, který udává počet permutací a vyskytuje se v dalších kombinatorických vztazích. V Matlabu se pro tento účel hodí funkce `prod`, která vrací jako výsledek součin prvků vektoru, který je jejím argumentem. Takže 10! zjistíme příkazem

```
>> prod(1:10)
ans =
     3628800
>>
```

Tento postup funguje dobře i pro 0!:

```
>> prod(1:0)
ans =
     1
>>
```

Tímto standardním způsobem lze v Matlabu na 32 bitovém procesoru získat maximálně faktoriál 170:

```
>> prod(1:170)
ans =
```

```
  7.2574e+306
>> prod(1:171)
ans =
   Inf
>>
```

Pokud bychom potřebovali pracovat s faktoriály větších čísel, museli bychom použít speciální knihovny.

V tomto ohledu je Sage daleko před Matlabem. Můžeme zde použít pro výpočet funkci `factorial`:

```
sage: factorial(10)
3628800
sage: factorial(100)
933262154439441526816992388562667004
907159682643816214685929638952175999
932299156089414639761565182862536979
208272237582511852109168640000000000
00000000000000
```

Sage si bez nesnází poradí s faktoriálem 10 000, ale pokud byste chtěli vědět, jak velký je výsledek, použijte místo počítání číslic raději numerickou aproximaci:

```
sage: factorial(10000).numerical_approx()
2.84625968091705e35659
```

Počet číslic výsledku by nám prozradil i dekadický logaritmus:

```
sage: log(factorial(10000),10).numerical_approx()
35659.4542745208
```

Když hovoříme o faktoriálech, nesmíme opominout *gamma* funkci Γ. Ta je definována pomocí integrálu a pro naše účely stačí vědět, že pro přirozená čísla platí $\Gamma(n+1) = n!$. V Matlabu je označena jako `gamma`, v Sage taktéž `gamma`. Zajímavé je, že v Sage dává hodnoty této funkce i funkce `factorial`, nesmíme ovšem zapomenout na posun argumentu o 1. Můžete si vyzkoušet příkazy

```
sage: gamma(1.5)
sage: factorial(1.5)
sage: factorial(0.5)
```

Pokud se týká výpočtu kombinačních čísel, podívejme se nejdříve, jak bychom postupovali při ručním počítání, třeba v Matlabu. Výpočet

podle vztahu

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

je ovšem neefektivní. Lepší postup je určit m jako menší z čísel k a $n-k$ a pak

$$\binom{n}{k} = \frac{n(n-1)\cdots(n-m+1)}{m!}\,.$$

Matlabovská funkce pro výpočet kombinačního čísla pak může vypadat třeba takto:

```
function kc=komb_c(n,k)
m=min([k,n-k]);
kc=prod(n-m+1:n)/prod(1:m);
```

Pomocí ní můžeme spočítat hodnotu např. $\binom{300}{100}$, ale pro $\binom{300}{150}$ už dostaneme výsledek mimo číselný rozsah Matlabu:

```
>> komb_c(300,150)
ans =
   Inf
>>
```

V tomto případě nám může pomoci *beta* funkce označovaná B, která podobně jako gamma funkce je definována pomocí integrálu, a platí pro ni vztah

$$B(x,y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}.$$

Snadno nahlédneme. že

$$\begin{aligned}\binom{n}{k} &= \frac{n!}{k!(n-k)!} = \frac{\Gamma(n+1)}{\Gamma(k+1)\Gamma(n-k+1)} = \\ &= \frac{\Gamma(n+1)\Gamma(n+2)}{\Gamma(k+1)\Gamma(n-k+1)\Gamma(n+2)} = \frac{\Gamma(n+1)}{\Gamma(n+2)}\frac{1}{\frac{\Gamma(k+1)\Gamma(n-k+1)}{\Gamma(n+2)}} = \\ &= \frac{1}{(n+1)B(k+1,n-k+1)}\end{aligned}$$

S použitím tohoto vztahu dokáže Matlab spočítat daleko větší kombinační čísla:

```
>> n=1000;k=500;
>> 1/((n+1)*beta(k+1,n-k+1))
ans =
  2.7029e+299
>>
```

Matlabovská standardní funkce pro výpočet kombinačních čísel je `nchoosek`. Kdy se podíváme na její zdrojak, zjistíme, že je výpočet je prováděn trochu chytřeji, než ve výše uvedené námi vytvořené funkci:

```
   if k > n/2, k = n-k; end
   nums = (n-k+1):n;
   dens = 1:k;
   nums = nums./dens;
   c = round(prod(nums));
```

Při výpočtu samozřejmě vznikají numerické chyby při jednotlivých děleních, jsou ale dostatečně malé, aby se zaokrouhlením odstranily.

Pro výpočet kombinačních čísel se v Sage používá funkce `binomial`. Zvládá spočítat hodně velká kombinační čísla, např.

```
sage: binomial(10000,100);
652084692454725756954159729272157186837813354254 16
743372210247172869206520770178988927510291340552 99
084785303061594709811828237198239270547927119529 61
274155627059484294047536322719590466575951328549 90
606768967505457396473467998111950929802400
```

a když si asi minutu počkáte, můžete zjistit i kolik je třeba $\binom{1\,000\,000}{500\,000}$.

Kromě toho, abychom věděli, kolik kombinací kombinací k-tého stupně z n prvků existuje, se nám taky může hodit všechny tyto kombinace určit. V Matlabu to umí uvedená funkce `nchoosek`, když jako první parametr zadáme vektor čísel, z nichž se mají kombinace vybírat. Podobně funguje i funkce `combnk`, z výstupu je ovšem vidět, že každá pracuje jinak:

```
>> nchoosek(1:5,3)
ans =
     1     2     3
     1     2     4
     1     2     5
     1     3     4
     1     3     5
     1     4     5
     2     3     4
     2     3     5
     2     4     5
     3     4     5
>> combnk(1:5,3)
ans =
```

```
     3     4     5
     2     4     5
     2     3     5
     2     3     4
     1     4     5
     1     3     5
     1     3     4
     1     2     5
     1     2     4
     1     2     3
>>
```

Sage pro tento účel používá funkci `combinations`:

```
sage: mset = [1,2,3,4,5,6]
sage: combinations(mset,2)
[[1, 2], [1, 3], [1, 4], [1, 5], [1, 6], [2, 3], [2, 4],
[2, 5], [2, 6], [3, 4], [3, 5], [3, 6], [4, 5], [4, 6],
[5, 6]]
```

Je také možné si nechat vypsat všechny permutace, v Matlabu použijeme funkci `perms`, v Sage funkci `permutations`.

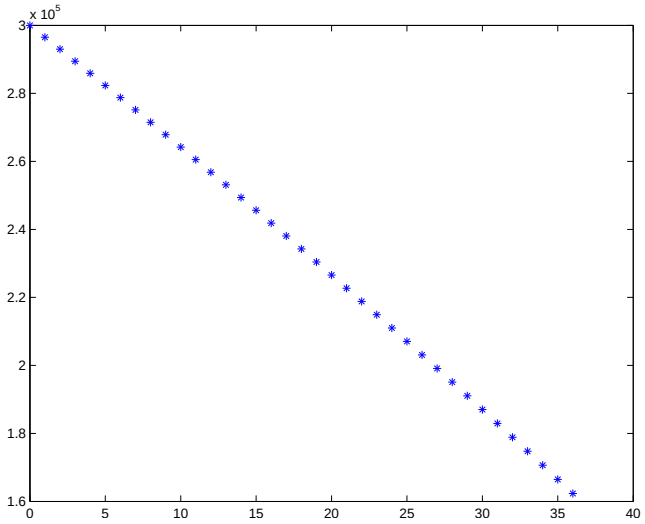
Co se týče dalších kombinatorických pojmů jako např. variace či kombinace s opakováním, pevně věřím, že určení jejich počtu pomocí některého v tomto textu používaného software by již čtenáři nečinilo žádný problém.

3. Diferenční rovnice

Ukažme si na několika příkladech, jak je možné pracovat s diferenčními rovnicemi, aniž bychom se snažili určit explicitní řešení.

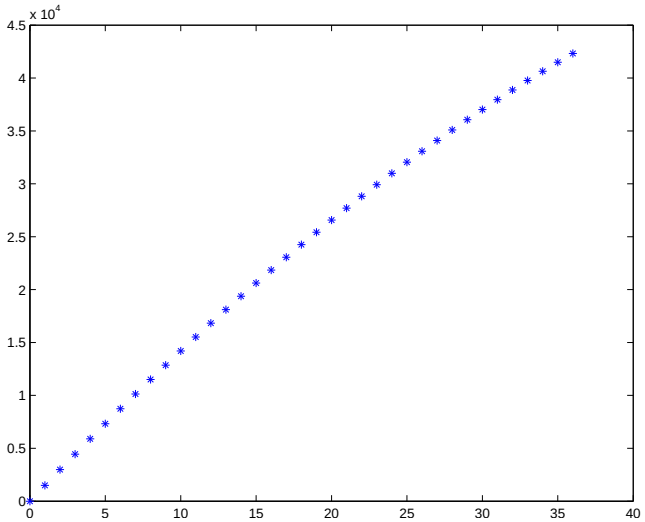
Příklad 4. Vezměme si splácení auta z příkladu 1.20. Výchozí částka je 300 000 korun, úroková míra $m = 6\% = 0,06$, úroky se počítají měsíčně. Je-li měsíční splátka S a zůstatek po k měsících označíme d_k, platí $d_0 = 300000$, $d_k = d_{k-1} + \frac{m}{12} d_{k-1} - S$. Pokud chceme v Matlabu spočítat a zobrazit, jak rychle bude dluh klesat během tří let splácení při platbě 5 000 měsíčně, můžeme postupovat třeba takto:

```
>> mes=36;
>> d_k=300000;m=0.06;S=5000;D=d_k;
>> for k=1:mes, d_k=d_k*(1+m/12)-S;...
D=[D,d_k];end
>> plot(0:mes,D,'*');
```



Může nás ale taky zajímat, jak poroste rozdíl mezi tím, kolik jsme zaplatili a tím, kolik nám ubylo z dluhu:

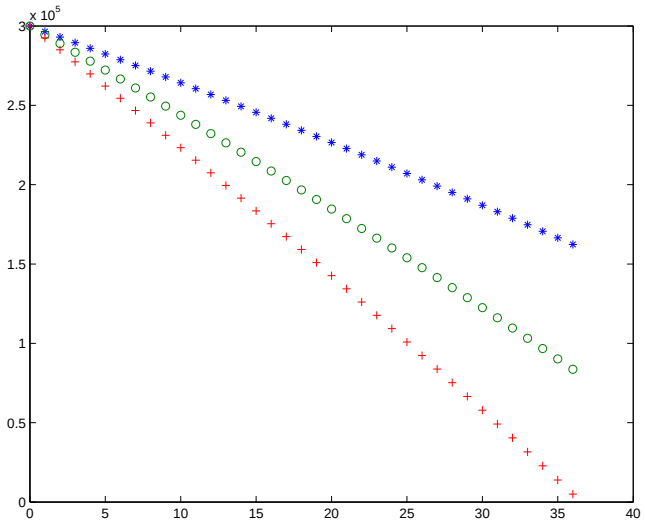
```
>> mes=36;
>> d_k=300000;m=0.06;S=5000;R=0;
>> for k=1:mes, d_k=d_k*(1+m/12)-S;...
R=[R,k*S-(300000-d_k)];end
>> plot(0:mes,R,'*');
```



Případně si můžeme zobrazit vývoj dluhu pro různě velké splátky:

```
>> mes=36;
>> d_k=300000;m=0.06;S=5000;D1=d_k;
>> for k=1:mes, d_k=d_k*(1+m/12)-S; D1=[D1,d_k];end
>> d_k=300000;m=0.06;S=7000;D2=d_k;
```

```
>> for k=1:mes, d_k=d_k*(1+m/12)-S; D2=[D2,d_k];end
>> d_k=300000;m=0.06;S=9000;D3=d_k;
>> for k=1:mes, d_k=d_k*(1+m/12)-S; D3=[D3,d_k];end
>> plot(0:mes,D1,'*',0:mes,D2,'o',0:mes,D3,'+');
```



Další možností by mohlo být zobrazit vývoj dluhu při různých úrokových mírách, ale to už by jistě čtenář zvládl sám.

Příklad 5. Uvažujme super banku, která nabízí při roční výpovědní lhůtě úrok 100 %. Takže při ročním úročení dostaneme za každou vloženou korunu koruny dvě. Co kdyby se ale úročilo pololetně:

```
>> n=2;d_k=1;m=1;
>> for k=1:n,d_k=(1+m/n)*d_k;end
>> d_k
d_k =
    2.2500
```

A co třeba čtvrtletně:

```
>> n=4;d_k=1;m=1;
>> for k=1:n,d_k=(1+m/n)*d_k;end
>> d_k
d_k =
    2.4414
```

Při měsíčním úročení dostaneme:

```
>> n=12;d_k=1;m=1;
>> for k=1:n,d_k=(1+m/n)*d_k;end
>> d_k
```

```
d_k =
    2.6130
```

V případě, že by banka úročila týdně, či dokonce denně, vyšlo by `d_k = 2.6926`, resp. `d_k = 2.7146`. K čemu ty hodnoty asi konvergují?

4. Pravděpodobnost

Tak mi řekněte, jaká je pravděpodobnost jevu A.
Je to jedna polovina, pane profesore.
Jak jste na to, proboha, přišel, Kropáčku?
No, buď to vyjde, nebo ne.

Pro nejrůznější pravděpodobnostní příklady se nám může hodit generování náhodných výsledků, které můžeme použít při simulacích náhodných jevů. Kromě funkcí zmíněných v kombinatorické části budeme používat i generátor náhodných čísel, i když ve skutečnosti se zpravidla jedná o čísla pseudonáhodná. V Matlabu je základním generátorem funkce `rand`, která dává čísla mezi 0 a 1, každé z těchto čísel by mělo mít stejnou pravděpodobnost, že bude zvoleno, v Sage je to `random`. Existuji ovšem funkce pro generování náhodných čísel podle zvolených kritérií, nejčastěji to bývají celá čísla v určeném rozsahu. V obou balících je pro tento účel funkce `randint`, takže stejného výsledku se můžeme dobrat více způsoby. V Sage navíc existuje funkce `random_matrix`, která je použitelná pro generování náhodných čísel z daného číselného okruhu, takže s jeho pomocí můžeme generovat matice reálné, racionální, celočíselné i komplexní.

Příklad 6. Vytvořme matici o dvou řádcích a čtyřech sloupcích s náhodnými celočíselnými prvky mezi 1 a 8. Nejprve možná řešení v Matlabu:

```
>> % pomocí zaokrouhlení
>> A=round(7*rand(2,4))+1
A =
     1     3     5     4
     6     4     8     8
>> % pomocí zaokrouhlení dolů
>> A=floor(8*rand(2,4))+1
A =
     2     3     4     8
     1     2     3     8
>> % pomocí zaokrouhlení nahoru
>> A=ceil(8*rand(2,4))
```

```
A =
     5     6     3     1
     4     6     4     8
>> % přímo
>> A=randint(2,4,[1,8])
A =
     8     1     8     2
     2     5     6     3
```

A řešení v Sage:

```
sage: A=matrix(2,[randint(1,8) for i in range(8)]);A
[6 5 7 6]
[5 8 1 6]
sage: A=random_matrix(ZZ,2,4,x=1,y=9);A
[2 7 1 8]
[4 5 7 8]
```

V posledně uvedeném příkazu se horní mez nedosahuje, proto je potřeba jako její hodnotu vzít 9.

Generování náhodných posloupností nebo matic je velmi užitečné kromě simulování některých náhodných jevů také při metodách souhrnně označovaných jako Monte Carlo. Princip spočívá v tom, že si vygenerujete množství simulovaných dat a z nich vybereme ty, které odpovídají nějakému jevu. Tímto způsobem se dají přibližně počítat nejen pravděpodobnosti jevů, ale také třeba plošné obsahy apod. Demonstrujeme si to na dvou příkladech.

Příklad 7. Určíme přibližnou hodnotu čísla π (viz též učebnice). K tomu využijeme následující skutečnost: pravděpodobnost toho, že dvě náhodně zvolená nezávislá reálná čísla z intervalu [0,1] mají součet druhých mocnin menší nebo roven jedné, je rovna ploše čtvrtiny jednotkového kruhu, což je $\pi/4$. V Matlabu tedy prvky náhodné matice o dvou řádcích umocníme po jednotlivých prvcích na druhou, zjistíme, kolik z nich je menších nebo rovno jedné, vydělíme celkovým množstvím dat, vynásobíme 4 a odečteme od π:

```
>> % 100 hodnot
>> n=100;A=rand(2,n);A2=A.^2;
>> pi-4*sum(sum(A2)<=1)/n
ans =
   -0.3784
>> % 1000 hodnot
>> n=1000;A=rand(2,n);A2=A.^2;
```

```
>> pi-4*sum(sum(A2)<=1)/n
ans =
    0.0256
>> % 1 000 000 hodnot
>> n=1000000;A=rand(2,n);A2=A.^2;
>> pi-4*sum(sum(A2)<=1)/n
ans =
    0.0016
```

Vidíme, že s přesností to není žádná sláva. Pro miliardu hodnot už Matlab odmítá matici vytvořit. Bylo by samozřejmě možné celý výpočet provádět v cyklu, čímž bychom mohli aproximace čísla π takto spočítat ještě přesněji. Případný zájemce by jistě zvládl si příslušný prográmek vyrobit sám.

Příklad 8. V dalším příkladu trochu předběhneme integrační počet. Zkusíme vypočítat plochu pod funkcí sinus na intervalu $[0, \pi]$. Budeme postupovat podobně jako v předchozím příkladě, vygenerujeme náhodné dvojice čísel na obdélníku $[0, \pi] \times [0, 1]$ a určíme poměrnou část těch, které leží pod grafem. Ta by měla být přibližně rovna poměru plochy pod grafem funkce k ploše celého obdélníka. Výsledek můžeme porovnat s přesnou hodnotou, která, jak se dozvíme v blízké budoucnosti, je rovna 2.

```
>> % 100 hodnot
>> n=100;x=pi*rand(1,n);y=rand(1,n);
>> 2-pi*sum(y<=sin(x))/n
ans =
   -0.1049
>> % 1000 hodnot
>> n=1000;x=pi*rand(1,n);y=rand(1,n);
>> 2-pi*sum(y<=sin(x))/n
ans =
   -0.0389
>> % 1 000 000 hodnot
>> n=1000000;x=pi*rand(1,n);y=rand(1,n);
>> 2-pi*sum(y<=sin(x))/n
ans =
  -9.1492e-04
```

Na závěr této části bych jen rád podotkl, že pro Matlab existuje speciální knihovna (toolbox) pro statistiku, který obsahuje velké kvantum nejrůznějších pravděpodobnostních a statistických funkcí. Pro ty,

kteří se budou potřebovat těmito záležitostmi zabývat do větší hloubky, uvádím odkaz na dokumentaci k tomuto toolboxu, která je skutečně velmi obsáhlá:
`http://www.mathworks.com/help/pdf_doc/stats/stats.pdf`

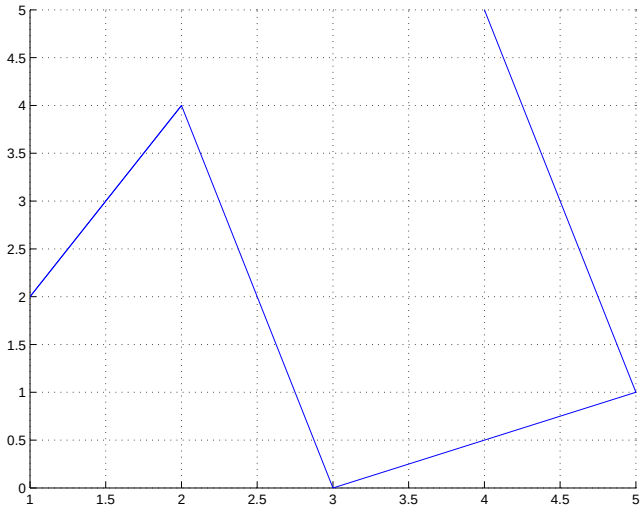
5. Geometrie v rovině

V této kapitole si ukážeme základní způsoby, jak je možné graficky znázorňovat některé rovinné geometrické objekty. Základní objektem je úsečka. Pro její zobrazení se v Matlabu i v Sage používá funkce `line`, syntaxe v je jednotlivých softwarech je, jak se dá očekávat, odlišná. V Matlabu při nejjednodušším použití funkce `line` zadáváme dva parametry, první z nich je vektor x-ových souřadnic bodů, které se mají spojit úsečkami, druhým parametrem je vektor y-ových souřadnic těchto bodů. Jednoduchou úsečku tedy získáme příkazem

```
>> line([1,2],[2,4])
```

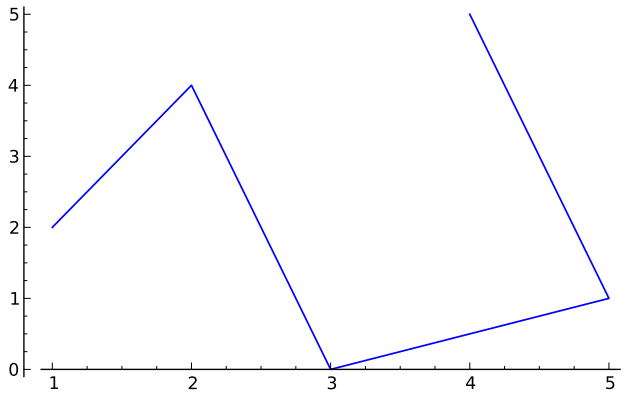
Pokud použijeme větší než dvouprvkové vektory, získáme lomenou čáru. Pro lepší přehled si také můžeme zobrazit mřížku.

```
>> line([1,2,3,5,4],[2,4,0,1,5])
>> grid on
```



V Sage je jako parametr funkce line seznam bodů, které se mají spojit úsečkami. Takže podobný obrázek, jako je předchozí, získáme v Sage příkazem

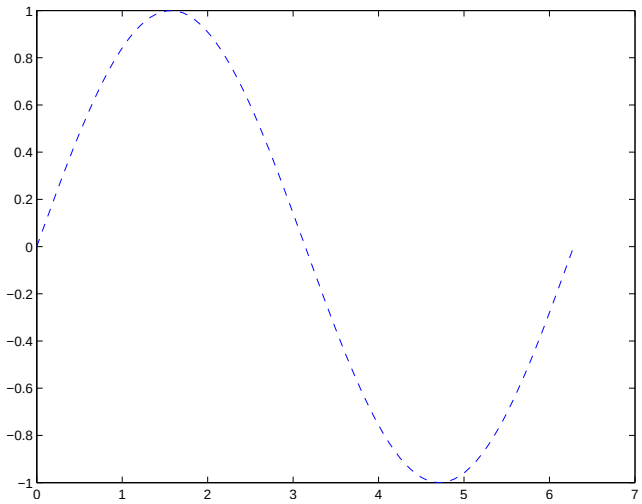
```
sage: line([(1,2), (2,4), (3,0), (5,1), (4,5)])
```



Pokud se nám zdá, že druhý obrázek je oproti prvním poněkud zdeformován, je to způsobeno tím, že měřítko není v obou osách stejné. Obdélníčky mřížky u prvního obrázku by ve skutečnosti měly být čtverečky. Obrázky Matlabu se přizpůsobují oknu ve kterém se zobrazují, v Sage je možné poměr os nastavit pomocí `aspect_ratio`, jak si ukážeme za chvíli.

V Matlabu pro zobrazování v rovině existuje univerzální funkce `plot`, která v základním tvaru má dva parametry podobně jako funkce `line`. Kromě nich lze použít další nepovinné parametry, které ovlivňují barvu obrázku, styl čáry a podobně. Například čárkovanou sinusovku získáme použitím příkazů

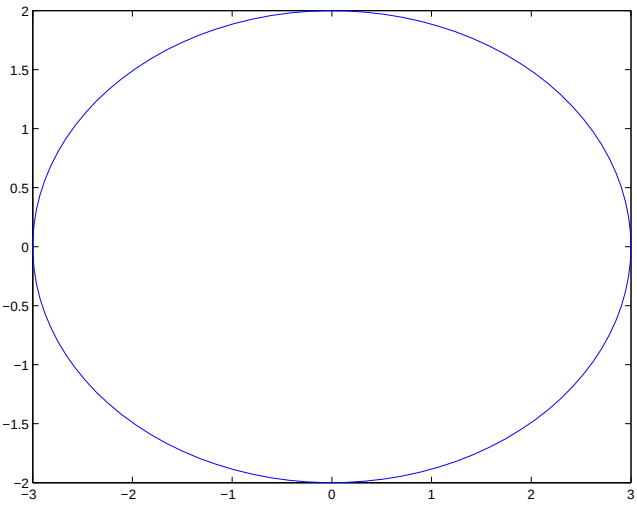
```
>> x=linspace(0,2*pi);y=sin(x);
>> plot(x,y,'--')
```



Nevýhodou tohoto postupu je, že i jednoduché geometrické objekty si musíme nejdříve spočítat, přesněji řečeno, musíme spočítat body, které

je tvoří a to na dostatečně husté síti. Třeba elipsu můžeme nakreslit takto:

```
>> t=linspace(0,2*pi);x=3*cos(t);y=2*sin(t);
>> plot(x,y)
```

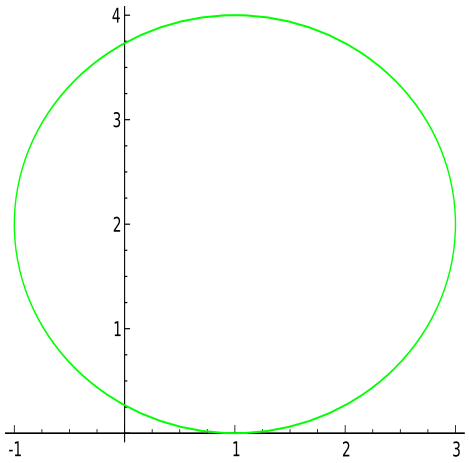


V Sage existuje pro zobrazování objektů v rovně funkcí několik. Např. pro kružnice je tu zvláštní funkce `circle`, která má dva povinné parametry – střed a poloměr, nicméně obrázek získaný příkazem

```
sage: circle((1,2),2)
```

by na první pohled připomínal elipsu. Proto uděláme malinko složitější konstrukci, současně nastavíme barvu kružnice na zelenou.

```
sage: c=circle((1,2),2, rgbcolor=(0,1,0))
sage: c.show(aspect_ratio=1)
```

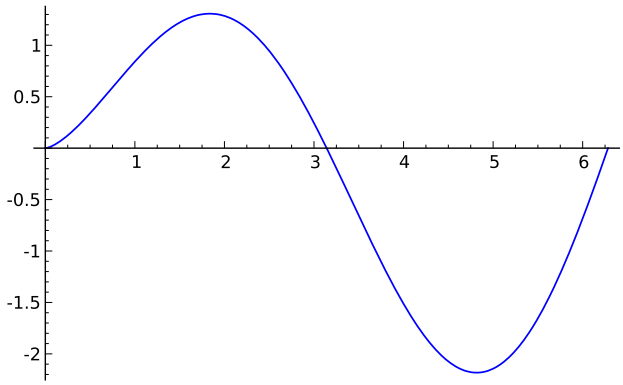


Také v Sage existuje funkce `plot`. Její nejjednodušší použití vypadá takto:

```
sage: plot(cos, (-5,5))
```

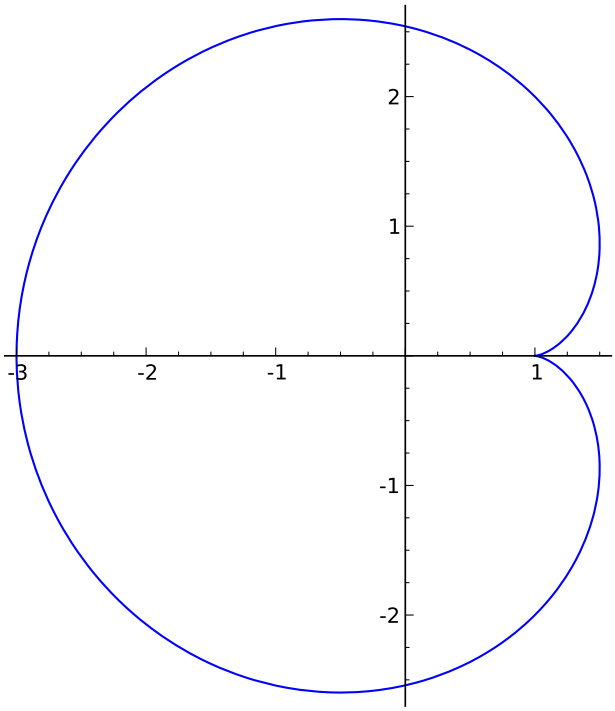
Předpokládám, že si každý dokáže představit obrázek, který vznikne, takže ho nemusím uvádět. Pro složitější funkce můžeme použít třeba něco jako

```
sage: x=var('x')
sage: plot(sqrt(x)*sin(x), (x,0,2*pi))
```



Kromě `plot` je v Sage také funkce `parametric_plot`, která pracuje podobně. Pomocí následujících příkazů získáme křivku zvanou epicykloida:

```
sage: t=var('t')
sage: x=2*cos(t)-cos(2*t)
sage: y=2*sin(t)-sin(2*t)
sage: parametric_plot((x,y),(t,0,2*pi))
```



Čtenář si nyní může sám rozmyslet, jak by bylo možné nakreslit například čtverec, pro různě formulovaná zadání.

Na závěr této části si předvedeme, jak lze pracovat s afinními transformacemi v rovině. Zkusme otočit a posunout výše nakreslenou elipsu. Nejprve určíme body na elipse podobně jako předtím.

```
>> t=linspace(0,2*pi);x=3*cos(t);y=2*sin(t);
```

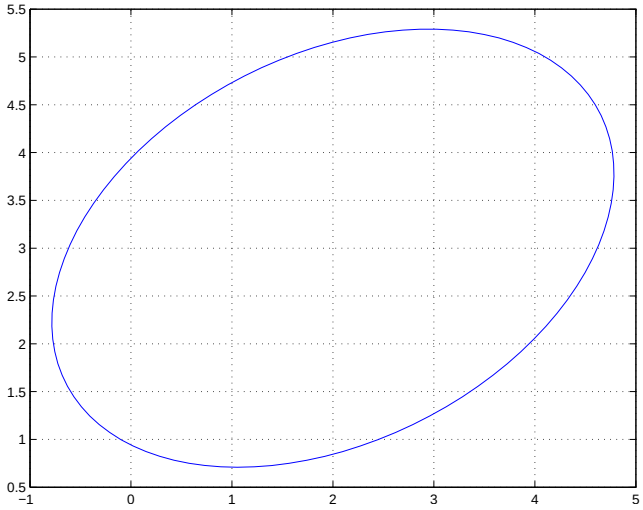
Pak spočítáme matici otočení o daný úhel, v našem případě to bude 30°, tedy $\pi/6$. Maticí vynásobíme souřadnice bodů na elipse. Pokud souřadnice umístíme do matice tak, že každý sloupec bude tvořit souřadnice jednoho bodu, pro transformaci rotace celé elipsy nám stačí jedno maticové násobení. Výsledek pak můžeme zpátky rozdělit na x-ové a y-ové souřadnice.

```
>> phi=pi/6;
>> R=[cos(phi), -sin(phi);sin(phi), cos(phi)];
>> xyR=R*[x;y];
>> xR=xyR(1,:);
>> yR=xyR(2,:);
```

Nakonec všechny body posuneme o potřebnou délku podle souřadnic středu elipsy, v uvedeném příkladě to je [2,3]. Při tom můžeme využít

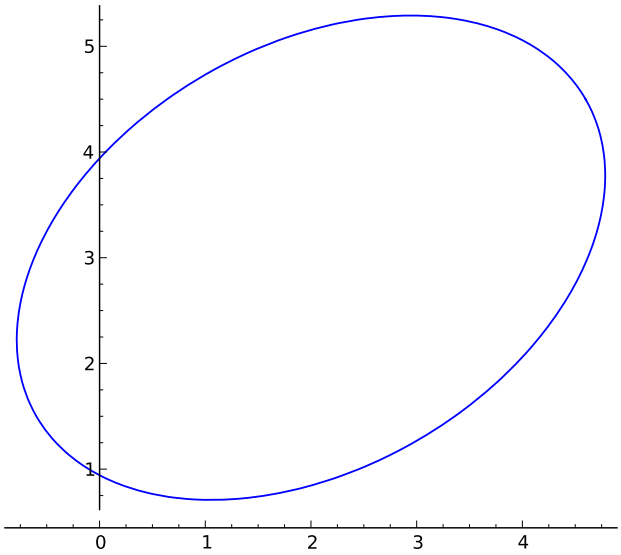
konvenci Matlabu, že při přičítání skaláru k vektoru nebo k matici se přičte příslušná hodnota ke každé složce.

```
>> xP=xR+2;
>> yP=yR+3;
>> plot(xP,yP)
>> grid on
```



A stejná konstrukce v Sage s tím, že maticové násobení vyjádříme explicitně.

```
sage: t=var('t')
sage: phi=pi/6
sage: x=3*cos(phi)*cos(t)-2*sin(phi)*sin(t)
sage: y=3*sin(phi)*cos(t)+2*cos(phi)*sin(t)
sage: parametric_plot((x+2,y+3),(t,0,2*pi))
```



Soudím, že další afinní transformace by již čtenář zvládl bez větších problémů.

KAPITOLA 2

Základy lineární algebry

Kropáčku, vypadáte dnes nějak unaveně. Čím to je?
Zkoušel jsem zjistit něco o maticích, jejich použití a historii, jak jste včera chtěl, pane profesore.
To mě těší. A jak jste postupoval?
Zadal jsem do googlu heslo Matrix *a pak se probíral výsledky.*
A co jste zjistil?
Je to opravdu skvělé téma. Zvládl jsem za noc všechny tři díly.

Z hlediska výpočetní techniky je lineární algebra jedním z nejlépe softwarově obhospodařovaným matematickým odvětvím. Existuje obrovská škála volně šiřitelných i komerčních programů a knihoven, které je možné použít pro řešení nejrůznějších typů obecných i specializovaných problémů, na něž můžeme v lineární algebře narazit. Je to dáno i tím, že nelezení řešení soustavy lineárních rovnic, což je typická úloha lineární algebry, patří mezi nejčastější inženýrské i ekonomické úlohy.

Matlab patří v tomto směru ke špičce mezi numericky orientovaným software. Je to dáno jeho základní orientací od začátku jeho vývoje. Název Matlab je původně zkratka od *Matrix Laboratory* a v jeho dřevních dobách byly matice jediným datovým typem, přičemž číslo se považovalo za matici o jednom řádku a jednom sloupci. I třeba problém hledání kořenů polynomu se převádí na problém nalezení vlastních čísel matice, což, jak jsme viděli, ne vždy dává plně uspokojivé výsledky.

K práci s maticemi a vektory jsme si již trochu přičichli v předchozí kapitole, nyní se pokusíme dané téma obsáhnout podrobněji. Trochu změníme způsob práce, nejprve se budeme věnovat Matlabu a až následně Sage.

1. Matice a vektory v Matlabu

U následujících příkladů nebudu většinou uvádět jejich výsledek, jelikož předpokládám, že si je čtenář bude zkoušet sám a i v případě, že tak neučiní, dokáže si výsledek snadno představit.

Matice v Matlabu můžeme zadat výčtem jejich prvků v hranatých závorkách, přičemž jednotlivé prvky na řádku oddělujeme mezerou nebo čárkou, jednotlivé řádky matice oddělujeme středníkem. Další funkcí středníku je, že pokud jej vložíme za příkaz, výsledek příkazu, např. přiřazení, se nezobrazí.

```
>> A=[1 -1 2 -3; 3 0 4 5; 3.2, 5, -6 12]
>> u=[1 2 3 4]
>> v=[-1;-2;-3]
```

Prázdnou matici vytvoříme příkazem

```
>> Emp=[]
```

Matice lze vytvářet pomoci už definovaných proměnných, je potřeba kontrolovat, aby souhlasily typy jednotlivých proměnných.

```
>> x=pi/4
>> y=[x, 2*x, 3*x]
>> B=[A; u]
>> C=[A, v]
```

Pro vytvoření matic větších rozměrů je možné použít některých funkci MATLABu. Příkaz

```
>> Z=zeros(2,5)
```

vytvoří nulovou matici příslušného typu, příkazem

```
>> O=ones(3,4)
```

vyrobíme matici že samých jedniček. Příkaz

```
>> I=eye(5,8)
```

nám dá *jednotkovou* matici, přičemž jedničky probíhají hlavní diagonálu. Všechny výše uvedené příkazy je možné žádat i s jedním parametrem, v tom případě je výsledkem čtvercová matice příslušného řadu. Funguje taky příkaz

```
>> O=ones(0,5)
```

Vektor, jehož prvky tvoří aritmetickou posloupnost, je možné vytvořit následujícím způsobem:

```
>> x=1:10
```

vytvoří vektor postupně obsahující čísla 1 až 10. Pro jiný rozdíl mezi jednotlivými prvky než 1 lze použít např. příkaz

```
>> y=0:2:12
```

pro y se sudými čísly od 0 do 12 nebo

```
>> z=0:0.01:2
```

pro z s čísly od 0 do 2 s krokem 0.01. Je možno použít i záporný krok:

```
>> x1=10:-1:1
```

Kromě toho taky můžeme použít příkaz `linspace`

```
>> x=linspace(a,b,n)
```

kde `a`, `b` jsou první a poslední prvek vektoru `x` a `n` je počet prvků. Třetí parametr je nepovinný a pokud není uveden, bere se roven 100. Podobně je taky možné definovat vektor s desítkovou logaritmickou škálou:

```
>> x=logspace(a,b,n)
```

který je ekvivalentní příkazu

```
>> x=10.^linspace(a,b,n)
```

Zde je ovšem implicitně n=50.

Rozměr matice zjistíme příkazem `size`. Na jednotlivé prvky matice je možné se odkázat pomoci kulatých závorek - tj. `A(3,2)` je prvek matice `A` ve 3. řádku a 2. sloupci. Toto vyjádření ovšem lze použít obecněji, kdy první parametr je vektor obsahující indexy vybraných řádku a druhý parametr je vektor sloupcových indexu. Pak

```
>> A([1 3],[5 2])
```

je vybrána submatice z `A` tvořena 1. a 3. řádkem a 5. a 2. sloupcem. Je možno taky provádět přiřazení do takto vybraných prvků při zachování sprvnych rozměrů:

```
>> A([1 3],[5 2])=eye(2)
```

Pokud chceme vybrat celý řádek nebo sloupec použijeme symbol ':', např. `B(3,:)` je 3. řádek matice `B`. Příkazem `B(3,:)=[]` vypustíme z matice B tento řádek. Zajímavé je, že nefunguje přiřazení

```
>> o=[];
>> B(3,:)=o
```

Matic stejných rozměrů lze sčítat a odčítat $(+, -)$, matice vhodných rozměrů se dají násobit (*). Je také možné násobit konstantou nebo

přičíst konstantu k matici, pak se přičte ke všem prvkům. Dělení jsou v MATLABu dvě – pravé a levé: '\', '/', přičemž výraz

```
>> X=B\C
```

dá řešení maticové rovnice `B*X=C` a výraz

```
>> X=B/C
```

je řešením maticové rovnice `X*C=B`.

Symbolem " ' " dostaneme hermitovskou transpozici matice, tj. matici transponovanou a komplexně sdruženou. Operátor ^ slouží k umocňování čtvercových matic. Uvedené operátory mají také tzv. tečkové varianty, kdy se operace provádějí na jednotlivé prvky matice, např.

```
>> A=B.*C
```

vynásobí odpovídající prvky matic `B` a `C`, příkazem

```
>> A.^2
```

se umocní všechny prvky matice `A`. Operátor " .' " se používá pro transpozici matic. Na matice lze aplikovat taky všechny běžné funkce (`sin`, `cos`,...), které se aplikují člen po členu.

Pokud máme matici `A` typu $m \times n$, pak pokus o určení hodnoty `A(k,l)`, kde `k`$> m$ nebo `l`$> n$, skončí chybou. Ovšem příkaz `A(k,l)=1` přiřazení provede, přičemž chybějící členy jsou doplněny nulami.

```
>> A=eye(2,3);
>> A(4,5)=1
A =
     1     0     0     0     0
     0     1     0     0     0
     0     0     0     0     0
     0     0     0     0     1
```

Příkazem `reshape` je možné přeskládat matici do jiného typu. Zadáním

```
>> A=randint(3,4,10)
A =
     6     5     2     8
     8     1     8     2
     9     1     2     9
>> B=reshape(A,6,2)
B =
     6     2
```

```
     8     8
     9     2
     5     8
     1     2
     1     9
```

vytvoříme z prvků matice `A` matici `B` o 6 řádcích a 2 sloupcích, přičemž prvky se při přeskládání berou po sloupcích. Pokud bychom chtěli matici `A` přeskládat po řádcích, museli bychom nejdřív matici `A` transponovat a pak opět transponovat výsledek, dostaneme tak ovšem matici o dvou řádcích a šesti sloupcích:

```
>> B=reshape(A',6,2)'
B =
     6     5     2     8     8     1
     8     2     9     1     2     9
```

Překlopit matici, tj. obrátit pořadí řádku nebo sloupců je možné provést příkazem `flipud` resp. `fliplr`. Také je možné matici „*otočit*“ o 90 stupňů příkazem `rot90`, přičemž další nepovinný parametr udává, kolikrát se má rotace provést.

Všechny operátory pro práci s maticemi uvedené dříve mají své funkci ekvivalenty:

funkce	význam	operátor
`plus`	plus	+
`uplus`	unární plus	+
`minus`	minus	−
`uminus`	unární minus	−
`mtimes`	maticové násobení	*
`times`	násobení po prvcích	.*
`mpower`	maticová mocnina	^
`power`	mocnina po prvcích	.^
`mldivide`	levé maticové dělení	\
`mrdivide`	právě maticové dělení	/
`ldivide`	levé dělení po prvcích	.\
`rdivide`	právě dělení po prvcích	./

Příkaz `a:b` případně `a:d:b` lze nahradit příkazem `colon(a,b)` nebo `colon(a,d,b)`.

Dvojí úlohu má funkce `diag`. Jeho aplikaci na vektor získáme diagonální matici s argumentem na hlavní diagonále. Pokud ji použijeme na matici, funkce `diag` vybere z matice hlavní diagonálu a umístí ji do vektoru. Pokud chceme pracovat s jinou diagonálou než s hlavní,

můžeme použít jako druhý parametr funkce `diag` číslo diagonály, přičemž kladná čísla se použijí nad hlavní diagonálou a záporná pod ní. Příklad:

```
\begin{Matlab}
>> A=randint(5,5,10)
A =
     3     3     2     0     1
     1     8     7     0     5
     2     5     7     5     4
     6     5     3     7     0
     4     9     5     9     3
>> x=diag(A)
x =
     3
     8
     7
     7
     3
>> B=diag(x,2)
B =
     0     0     3     0     0     0     0
     0     0     0     8     0     0     0
     0     0     0     0     7     0     0
     0     0     0     0     0     7     0
     0     0     0     0     0     0     3
     0     0     0     0     0     0     0
     0     0     0     0     0     0     0
>> C=diag(pi,-3)
C =
         0         0         0         0
         0         0         0         0
         0         0         0         0
    3.1416         0         0         0
>> d=diag(diag(A))
d =
     3     0     0     0     0
     0     8     0     0     0
     0     0     7     0     0
     0     0     0     7     0
     0     0     0     0     3
```

Pomoci funkci `tril` a `triu` vyrobíme z daně matice dolní nebo horní trojúhelníkovou matici, přičemž je možné podobně jako u `diag` použít další nepovinný parametr.

Příkaz `max` najde maximální prvek ve vektoru, přičemž pokud na výstupu uvedeme i druhý výstupní parametr, uloží se do něj index tohoto prvku. Při použití příkazu `max` na matici se hledají maximální prvky v jednotlivých sloupcích. Podobně funguje příkaz `min`:

```
>> x=rand(1,5)
x =
    0.2785    0.5469    0.9575    0.9649    0.1576
>> M=max(x)
M =
    0.9649
>> [m,k]=min(x)
m =
    0.1576
k =
     5
>> A=rand(4)
A =
    0.9706    0.1419    0.9595    0.9340
    0.9572    0.4218    0.6557    0.6787
    0.4854    0.9157    0.0357    0.7577
    0.8003    0.7922    0.8491    0.7431
>> [m,k]=min(A)
m =
    0.4854    0.1419    0.0357    0.6787
k =
     3     1     3     2
```

U komplexních matic se maximum či minimum hledá mezi absolutními hodnotami prvků.

K setřídění vektoru (vzestupně) se používá příkaz `sort`, u komplexních hodnot se opět provádí třídění podle absolutních hodnot. Do druhého nepovinného výstupního parametrů je umístěna třídicí permutace indexu. Tj. pokud zadáme `[y,ind]=sort(x)` pak platí `y=x(ind)`. Při použití příkazu `sort` na matice se třídí jednotlivé sloupce.

Příkazy `sum` a `prod` sečtou nebo vynásobí všechny prvky vektoru. Aplikovaný na matici provedou příslušnou operaci po jednotlivých sloupcích. Např. faktoriál čísla n zjistíme snadno pomoci příkazu `f=prod(1:n)`, přičemž příkaz funguje správné i pro `n` rovno 0, protože součin přes prázdnou matici je roven 1.

Pokud bychom chtěli zjistit součet všech prvků matice, musíme příkaz `sum` použít dvakrát:

```
>> sum(sum(A))
```

Příkazem `size` zjistíme rozměry matice, přičemž je možné jej použít ve formě `s=size(a)` nebo `[m,n]=size(a)`. Příkaz `length` dává délku vektoru, přičemž pokud jej aplikujeme na matici, dostaneme maximální z rozměrů, tj. `length(A)` dává stejný výsledek jako `max(size(A))`.

V MATLABu je implementováno velké množství funkci a algoritmu lineární algebry. Podrobný výpis lze získat pomoci nápovědy `help matfun`. Zde vyjmenujeme jen některé základní:

`det`	determinant matice
`rank`	hodnost matice
`null`	nulový prostor (jádro) matice
`norm`	maticová nebo vektorová norma
`trace`	stopa matice (součet diagonálních prvků)
`inv`	inverzní matice
`pinv`	pseudoinverzni matice
`lu`	LU rozklad
`qr`	QR rozklad
`svd`	singulární rozklad
`eig`	vlastní hodnoty a vektory matice
`poly`	charakteristický polynom matice, resp. vytvoření polynomu s danými kořeny

Syntaxi a popis jednotlivých funkci zjistíme nejlépe pomoci nápovědy. Blíže se budeme věnovat jen funkci `eig` pro učení vlastních čísel a vlastních vektorů matice.

Tato funkce má dva základní způsoby použití. Pokud je výstupní proměnná pouze jedna, vrátí funkce vektor vlastních hodnot dané matice

```
>> A=randint(3,3,[1,10])
A =
     9    10     3
    10     7     6
     2     1    10
>> D=eig(A)
D =
   19.4625
   -1.9198
    8.4573
```

V případě, že použijeme dvě výstupní proměnné, získáme i matici, jejíž sloupce jsou tvořeny vlastními vektory původní matice. Vlastní čísla už netvoří vektor ale hlavní diagonálu v diagonální matici:

```
>> [V,D]=eig(A)
V =
   -0.7062   -0.6823   -0.5253
   -0.6728    0.7291   -0.2182
   -0.2204    0.0533    0.8225
D =
   19.4625         0         0
         0   -1.9198         0
         0         0    8.4573
```

Pro takto uspořádané vlastní vektory a vlastní čísla by mělo platit `A*V=V*D`:

```
>> A*V
ans =
  -13.7453    1.3099   -4.4425
  -13.0943   -1.3997   -1.8457
   -4.2890   -0.1024    6.9559
>> V*D
ans =
  -13.7453    1.3099   -4.4425
  -13.0943   -1.3997   -1.8457
   -4.2890   -0.1024    6.9559
```

Ve skutečnosti je mezi výsledky rozdíl řádově 10^{-14}.

2. Matice a vektory v Sage

V Sage je pro vytváření vektorů určna funkce `vector`. Její parametrem je seznam prvků.

```
sage: v=vector([1,2,3]);v
(1, 2, 3)
```

Takto se vytvoří vektor řádkový, v případě, že potřebujeme vektor sloupcový, musíme jej transponovat příkazem `transpose`, nebo ho vytvořit coby sloupcovou matici, což je popsáno o něco níže. Standardně se jedná o vektor celočíselný nebo racionální podle typu výrazů, kterými je vytvořen. Je možné ale při definici vnutit typ jiný pomocí prvního argumentu před samotným výčtem prvků.

```
sage: v=vector([1,2,3]);v
(1, 2, 3)
sage: w=vector(RR,[1,2,3]);w
(1.00000000000000, 2.00000000000000, 3.00000000000000)
```

K jednotlivým prvkům vektoru se přistupuje pomocí hranatých závorek, prvky jsou indexovány od nuly, nikoliv od jedničky.

```
sage: w[1]
2.00000000000000
```

Při vytváření vektorů je potřeba dát pozor na to, abychom nepoužili matlabovskou konstrukci

```
sage: u=[2,3,-1];u
[2, 3, -1]
```

Výsledek vypadá stejně jako u vektoru, ale nejedná se o vektor, nýbrž seznam prvků, jak prozradí příkaz `type`:

```
sage: type(v)
<type 'sage.modules.vector_integer_dense.
Vector_integer_dense'>
sage: type(u)
<type 'list'>
```

K vytváření matic používáme funkci `Matrix`, funguje i s počátečním písmenem malým. Nejjednodušší je vytvoření čtvercové matice, k čemuž stačí jeden parametr:

```
sage: A=Matrix(3);A
[0 0 0]
[0 0 0]
[0 0 0]
```

Opět je možné vnutit vytvářené matici jiný typ:

```
sage: B=Matrix(RR,2);B
[0.000000000000000 0.000000000000000]
[0.000000000000000 0.000000000000000]
```

Pomocí dalších parametrů můžeme definovat obdélníkové matice a její prvky, nebo lze vytvořit matici pomocí seznamu jejich prvků:

```
sage: B=Matrix(2,3);B
[0 0 0]
[0 0 0]
```

```
sage: B=Matrix(2,3,[k/6 for k in range(6)]);B
[  0 1/6 1/3]
[1/2 2/3 5/6]
sage: B=Matrix(2,[k/6 for k in range(6)]);B
[  0 1/6 1/3]
[1/2 2/3 5/6]
sage: C=Matrix([[1,2,3],[4,5,6],[7,8,9]]);C
[1 2 3]
[4 5 6]
[7 8 9]
```

Povšimněme si, že druhý a třetí příkaz dávají stejný výsledek, pokud bychom ve druhém příkazu použili jiný počet sloupců než 3, došlo by k chybě. K jednotlivým prvkům přistupujeme opět pomocí hranatých závorek, přitom je možný dvojí způsob syntaxe. Nesmíme zapomenout na indexování od nuly:

```
sage: B[1,1]
2/3
sage: B[1][2]
5/6
```

Rozměry matice `A` zjistíme pomocí `A.nrows()`, resp. `A.ncols()`. Jednotkovou matici získamé pomocí příkazu `identity_matrix`, jejíž parametr udává rozměr matice (pouze čtvercové).

Zdálo by se, že vektory můžeme ekvivalentně vytvořit jako matice o jednom řádku. Což je sice možné, ale operace pak fungují trochu jinak. V Sage je totiž možné násobit matici zprava řádkovým vektorem. Výsledkem je opět řádkový vektor, který má stejné prvky, které bychom dostali při obvyklém násobení sloupcovým vektorem:

```
sage: A=Matrix([[1,2,3],[3,2,1],[1,1,1]])
sage: v=vector([1,2,3])
sage: A*transpose(v)
[14]
[10]
[ 6]
sage: A*v
(14, 10, 6)
```

Na první pohled to je sice trochu nezvyklé, ale po nějaké době používání nám to už zvláštní nepřijde. S řádkovou maticí ovšem tato operace nefunguje:

```
sage: A=Matrix([[1,2,3],[3,2,1],[1,1,1]])
sage: B=Matrix([1,2,3])
sage: A*B
------------------------------------------------
TypeError atd.
```

Pro řešení systému lineárních rovnic je možné použít levé dělení podobně jako v Matlabu, pravé dělení nefunguje. Alternativní metodou je použití funkce `solve_right` nebo `solve_left`:

```
sage: A=Matrix([[1,2,3],[3,2,1],[1,-1,1]])
sage: b=vector([1,2,3])
sage: x=A\b;x
(19/16, -9/8, 11/16)
sage: x=A.solve_right(b);x
(19/16, -9/8, 11/16)
```

Funkce `det` a `rank` fungují tak, jak bychom čekali. Charakteristický polynom matice lze určit takto:

```
sage: A=Matrix([[1,2,3],[3,2,1],[1,-1,1]])
sage: characteristic_polynomial(A)
x^3 - 4*x^2 - 3*x + 16
```

Inverzní matici a stopu matice můžeme vyjádřit pouze následujícím způsobem:

```
sage: A=Matrix([[1,2,3],[3,2,1],[1,-1,1]])
sage: A.inverse()
[-3/16  5/16   1/4]
[  1/8   1/8  -1/2]
[ 5/16 -3/16   1/4]
sage: A.trace()
4
```

Podobně můžeme určit vlastní čísla a vlastní vektory matice:

```
sage: A=Matrix([[1,2],[3,2]])
sage: A.eigenvalues()
[4, -1]
sage: A.eigenvectors_right()
[(4, [
(1, 3/2)
], 1), (-1, [
```

```
(1, -1)
], 1)]
```

Možná, že poslední výsledek vypadá na první pohled poněkud nesrozumitelně, ale není to tak složité. Jedná se o seznam trojic, kde na první místě je vlastní číslo, pak následuje vlastní vektor a třetí v pořadí je násobnost. (Myslím, že přehlednější zformátování výsledku by neškodilo.)

V Sage taky existuje funkce `kernel`, u níž bychom podle jména očekávali, že s její pomocí lze zjistit jádro matice jakožto lineárního zobrazení, přesněji řečeno jeho bázi. To sice možné je, ale tato funkce pracuje po řádcích:

```
sage: A=Matrix([[1,2,3],[3,2,1],[1,1,1]])
sage: A.kernel()
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[ 1  1 -4]
```

Vidíme, že první a druhý řádek spolu skutečně dávají čtyřnásobek třetího. Takže pro nalezení jádra lineárního zobrazení daného maticí bychom ji museli nejdříve transponovat.

Závěrem této části můžete hádat či zkusit, co dostaneme jako výsledek příkazů `A.rows()` a `A.columns()`.

3. Příklady

Příklad 9. Zkusme si nejprve demonstrovat v Matlabu úpravu matice na schodovitý tvar. Vezměme rozšířenou matici soustavy z příkladu 2.1:

```
>> A=[0.5,0.125,0.2 270;0.25,0.75,0.2,270;...
0.25,0.125,0.6,270]
A =
    0.5000    0.1250    0.2000  270.0000
    0.2500    0.7500    0.2000  270.0000
    0.2500    0.1250    0.6000  270.0000
```

Vynásobení řádku konstantou odpovídá násobení zleva diagonální maticí, kde na příslušném řádku je potřebná konstanta.

```
>> D=diag([2,4,4])
D =
     2     0     0
     0     4     0
     0     0     4
```

```
>> A1=D*A
A1 =
   1.0e+03 *
    0.0010    0.0003    0.0004    0.5400
    0.0010    0.0030    0.0008    1.0800
    0.0010    0.0005    0.0024    1.0800
```

Všimněte si konstanty `1.0e+03`, která je vytknuta před matici. Pokud bychom radši viděli zobrazení bez této konstanty, použijeme

```
>> format short g
>> A1
A1 =
            1         0.25          0.4          540
            1            3          0.8         1080
            1          0.5          2.4         1080
```

První řádek vynásobíme -1 a přičteme ho ke druhému a třetímu. Tato úprava odpovídá násobení zleva maticí, která vychází z jednotkové a má navíc v prvním sloupci -1 ve druhém a třetím řádku

```
>> T1=eye(3);T1(2,1)=-1;T1(3,1)=-1
T1 =
     1     0     0
    -1     1     0
    -1     0     1
>> A2=T1*A1
A2 =
            1         0.25          0.4          540
            0         2.75          0.4          540
            0         0.25            2          540
```

Druhý a třetí řádek opět vynásobíme 4

```
>> A3=diag([1,4,4])*A2
A3 =
            1         0.25          0.4          540
            0           11          1.6         2160
            0            1            8         2160
```

přehodíme 2. a 3. řádek

```
>> T3=eye(3);T3=T3([1,3,2],:)
T3 =
     1     0     0
     0     0     1
```

```
     0     1     0
>> A4=T3*A3
A4 =
            1         0.25          0.4          540
            0            1            8         2160
            0           11          1.6         2160
```

a nakonec druhý řádek vynásobíme -11 a přičteme ho ke třetímu

```
>> T4=eye(3);T4(3,2)=-11;A5=T4*A4
A5 =
            1         0.25          0.4          540
            0            1            8         2160
            0            0       -86.4       -21600
```

Nakonec bychom ještě mohli vydělit pokrátit řádek, kvůli jednomu řádku je asi zbytečné vyrábět diagonální matici:

```
>> A5(3,:)=A5(3,:)/A5(3,3)
A5 =
            1         0.25          0.4          540
            0            1            8         2160
            0            0            1          250
```

Pokud by se nyní někomu nechtělo ručně počítat jednotlivé neznáme, může se matici dál upravovat až na jednotkovou (přesněji její první tři sloupce), v tom případě mu v posledním sloupci vyjde řešení.

Pokud bychom tuto soustavu řešili v Sage a zadávali matici podobně jako v Matlabu, dostali bychom desetinná čísla s patnácti číslicemi za desetinou tečkou, takže by obsahovala spoustu zbytečných nul. Proto lepe bude zadat matici jako racionální:

```
sage: A=matrix(QQ,[[0.5,0.125,0.2,270],
[0.25,0.75,0.2,270],[0.25,0.125,0.6,270]]);A
[1/2 1/8 1/5 270]
[1/4 3/4 1/5 270]
[1/4 1/8 3/5 270]
```

Sage ovšem umí provést uvedené úpravy sám, dokonce až do úplného konce

```
sage: A.echelon_form()
[  1   0   0 400]
[  0   1   0 160]
[  0   0   1 250]
```

takže hned vidíme řešení soustavy. Matlab tohle umí taky, příslušná funkce se jmenuje `rref`:

```
>> rref(A)
ans =
     1     0     0   400
     0     1     0   160
     0     0     1   250
```

Ukazovat příklady na výpočet determinantu, hodnosti nebo inverzní matice pomocí Matlabu a Sage je patrně zbytečné. Názvy příslušných funkcí najdeme výše a jejich použití je snadné, bez nějakých záludností. Pevně věřím, že čtenář by dokonce našel několik způsobu (pšt nebo šest) jak vypočítat inverzní matici.

Dobrat se řešení soustavy lineárních rovnic je také možné několika způsoby:

```
sage: A=matrix(QQ,[[0.5,0.125,0.2],[0.25,0.75,0.2],
[0.25,0.125,0.6]]);A
[1/2 1/8 1/5]
[1/4 3/4 1/5]
[1/4 1/8 3/5]
sage: b=vector([270,270,270]);b
(270, 270, 270)
sage: A.inverse()*b
(400, 160, 250)
sage: A\b
(400, 160, 250)
sage: A^(-1)*b
(400, 160, 250)
sage: A.solve_right(b)
(400, 160, 250)
```

V Matlabu lze použít podobně všechny způsoby kromě posledního, jediným drobným rozdílem je `inv(A)` místo `A.inverse()`. Oproti Sage ale v Matlabu existuje funkce `linsolve`, o níž je možné se více dozvědět v nápovědě.

Příklad 10. Určit lineární závislost či nezávislost je poměrně jednoduché. Stačí příslušné vektory poskládat do matice a pak určit její hodnost. V Sage můžeme postupovat například takto:

```
sage: u1=vector([1,2,3])
sage: u2=vector([4,5,6])
sage: u3=vector([7,8,9])
sage: A=matrix([u1,u2,u3]);A
[1 2 3]
[4 5 6]
[7 8 9]
sage: rank(A)
2
```

Řešení v Matlabu je opět velmi podobné, takže ho nebudu uvádět. Problém nalezení jednoho z lineárně závislých vektorů jakožto lineární kombinace ostatních je problém hledání řešení soustavy lineárních rovnic, přičemž toto řešení nemusí být určeno jednoznačně. Zmíníme se o tom o něco níže. Nejdříve se ještě podívejme, jak nalézt bázi nějakého podprostoru, což je problém, který s lineární závislostí či nezávislostí úzce souvisí.

Při hledání báze, můžeme vektory poskládat do matice a tuto pak upravovat na schodovitý tvar, čímž vynulujeme lineárně závislé vektory, nenulové řádky pak tvoří bázi. S úspěchem lze tedy použít funkce `rref` nebo `echelon_form`. V Sage navíc existuje funkce `basis`, která je pro tento účel určena:

```
sage: V = VectorSpace(QQ,3)
sage: S = V.subspace([[1,2,0],[2,2,-1],[-1,0,1]])
sage: basis(S)
[
(1, 0, -1),
(0, 1, 1/2)
]
```

Porovnáme-li výsledek s postupem uvedeným předtím,

```
sage: u1=vector([1,2,0])
sage: u2=vector([2,2,-1])
sage: u3=vector([-1,0,1])
sage: A=matrix([u1,u2,u3])
sage: A.echelon_form()
[ 1  0 -1]
[ 0  2  1]
[ 0  0  0]
```

vidíme, že jediný rozdíl je v násobení druhého bázového vektoru tak, aby měl první koeficient roven jedné.

V Matlabu lze použít funkci `orth`, která hledá ortonormální bázi prostoru generovaného sloupci matice.

```
>> A=reshape(1:9,3,3)
A =
     1     4     7
     2     5     8
     3     6     9
>> Q=orth(A)
Q =
   -0.4797    0.7767
   -0.5724    0.0757
   -0.6651   -0.6253
>> Q'*Q
ans =
    1.0000   -0.0000
   -0.0000    1.0000
```

Tato funkce však evidentně nepracuje Grammovou–Schmidtovou ortogonalizací, to by musel být první sloupce matice `Q` násobek prvního sloupce matice `A`.

Když jsme se dostali k pojmu ortonormální, měli bychom se vrátit ke skalárnímu součinu. V Sage je pro skalární součin určena operace '*':

```
sage: u=vector([1,2,3])
sage: v=vector([-1,2,1])
sage: s1=u*v;s1
6
```

Pokud bychom chtěli použít maticové násobení, je to možné, ale výsledek má jiný typ:

```
sage: s2=u*transpose(v);s2
(6)
sage: type(s1)
<type 'sage.rings.integer.Integer'>
sage: type(s2)
<type 'sage.modules.vector_integer_dense.
Vector_integer_dense'>
```

V Matlabu se pro skalární součin používá funkce `dot`, ale maticové násobení dává stejný výsledek:

```
>> u=[1,2,3];v=[-1,2,1];
>> dot(u,v)
ans =
     6
>> u*v'
ans =
     6
```

Příklad 11. Zaměřme se na postup při hledání ortogonálního doplňku nějakého podprostoru, který je určen svou bází, Také pro ortogonální doplněk stačí nalézt jeho bázi. Hledáme tedy nezávislá řešení homogenní soustavy rovnic `A*x=0`, kde matice `A` je tvořena řádky báze našeho podprostoru. Takže hledaný ortogonální doplněk je vlastně jádrem lineárního zobrazení daného maticí `A`. Teď už tedy problém vyřešíme snadno.
Příklad řešení v Sage:

```
sage: u=vector([1,2,-1,4])
sage: v=vector([-1,2,0,3])
sage: A=matrix([u,v])
sage: B=A.transpose().kernel().basis()
sage: B
[
(1, 2, 1, -1),
(0, 3, -2, -2)
]
```

A pak ještě můžeme ověřit, že odpovídající vektory jsou opravdu kolmé:

```
sage: C=matrix([B[0],B[1]]);C
[ 1  2  1 -1]
[ 0  3 -2 -2]
sage: A*transpose(C)
[0 0]
[0 0]
```

A řešení stejné úlohy v Matlabu:

```
>> u=[1,2,-1,4];
>> v=[-1,2,0,3];
>> A=[u;v];
```

```
>> B=null(A)
B =
    0.3707   -0.2597
   -0.0686   -0.8403
    0.9106   -0.0456
    0.1693    0.4737
>> B'*B
ans =
    1.0000    0.0000
    0.0000    1.0000
>> A*B
ans =
   1.0e-15 *
         0    0.2220
         0         0
```

Vidíme, že nalezená báze je ortonormální a ve výsledku je malá numerická chyba. V podobných případech je zpravidla na uživateli, aby posoudil, jestli výsledek má být ve skutečnosti nulový a případně provedl nějaké dodatečné úpravy.

Příklad 12. Na závěr této části si ukážeme, jak si poradit se systémem, který má více řešení. (Případ, kdy systém nemá řešení poznáme lehce, když porovnáme hodnost matice soustavy a hodnost matice rozšířené soustavy.) Můžeme samozřejmě matici upravit na schodovitý tvar a pak hledat řešení ručně:

```
sage: A=matrix([[1,2,3,1],[4,5,6,1],[7,8,9,1]]);A
[1 2 3 1]
[4 5 6 1]
[7 8 9 1]
sage: A.echelon_form()
[1 2 3 1]
[0 3 6 3]
[0 0 0 0]
```

Z druhého řádku dostávám $y + 2z = 1$, můžeme najít jedno řešení dosazením hodnoty za y nebo z, případně obecné řešení v parametrickém tvaru, pokud dosadíme třeba $y = t$. Toto řešení umí Sage najít i sám, bohužel se mu musí zadat všechny tři rovnice explicitně:

```
sage: x,y,z=var('x,y,z')
sage: solve([x+2*y+3*z==1,4*x+5*y+6*z==1,7*x+8*y+9*z==1],
```

```
x,y,z)
[[x == r1 - 1, y == -2*r1 + 1, z == r1]]
```

Najít jedno (tzv. partikulární) řešení můžeme v Sage nejméně dvěma způsoby:

```
sage: A=matrix([[1,2,3],[4,5,6],[7,8,9]]);A
[1 2 3]
[4 5 6]
[7 8 9]
sage: b=vector([1,1,1]);b
(1, 1, 1)
sage: A.solve_right(b)
(-1, 1, 0)
sage: A\b
(-1, 1, 0)
```

Pokud ještě určíme bázi jádra matice `A`,

```
sage: kernel(A.transpose()).basis()
[
(1, -2, 1)
]
```

tak snadno určíme libovolné řešení v parametrickém tvaru, pakliže víme, že každé takové řešení dostaneme z partikulárního přičtením lineární kombinace báze jádra matice `A`.

V Matlabu se dá určit partikulární řešení také nejméně dvěma způsoby:

```
>> A=[1,2,3;4,5,6;7,8,9],b=[1;1;1],
A =
     1     2     3
     4     5     6
     7     8     9
b =
     1
     1
     1
>> A\b
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 1.541976e-18.
ans =
         -2.5
            4
```

```
         -1.5
>> pinv(A)*b
ans =
         -0.5
   6.9389e-18
          0.5
```

Všimněme si jednak varování u prvního výpočtu, jednak malé druhé složky řešení u druhého výsledku, které by měla být ve skutečnosti nulová a do třetice toho, že výsledky nejsou stejné. První výsledek se počítá pomocí Gaussovy eliminace (viz `help mldivide`), druhý pomocí tzv. pseudoinverzní matice, čímž dostaneme řešení s nejmenší normou (velikostí).

K určení báze jádra matice slouží funkce `null`, o které už jsme se zmiňovali. Standardně bázi normuje na velikost jednotlivých vektorů rovnu jedné, pokud chceme racionální složky bázových vektorů, můžeme použít další parametr `'r'`:

```
>> A=[1,2,3;4,5,6;7,8,9];
>> null(A)
ans =
     -0.40825
       0.8165
     -0.40825
>> null(A,'r')
ans =
     1
    -2
     1
```

Obecné řešení tedy opět získáme jako ve tvaru $u_0 + t[1, -2, 1]$, kde u_0 je partikulární řešení. Současně jsme ověřili známou věc, že totiž obecné řešení lze zapsat vícerým způsobem. Také si čtenář může rozmyslet, pro jaké hodnoty t dostaneme z jednoho z uvedených partikulárních řešení to druhé.