Numerické výpočty

Jiří Zelinka

Obsah

1 Polynomy **4**
1.1 Hornerovo schema a základní úkony s polynomy 4
1.2 Interpolace 9
1.3 Kořeny polynomů 16
1.4 Splajny 24
1.5 Bersteinovy polynomy a Bézierovy křivky 30

2 Derivace **37**
2.1 Limity 37
2.2 Symbolické derivování 40
2.3 Taylorův rozvoj 41
2.4 Numerické derivování 43

3 Integrály **45**
3.1 Symbolické integrování 45
3.2 Numerické integrování 47

4 Řady **52**
4.1 Symbolické součty řad 52
4.2 Fourierovy řady 54

5 Řešení nelineárních rovnic **57**
5.1 Motivační úloha 57
5.2 Základní metody řešení nelineární rovnice 61
5.3 Rychlost konvergence 73
5.4 Řešení systému nelineárních rovnic 77

6 Funkce více proměnných **79**
6.1 Zobrazovnání grafů 79
6.2 Výpočet parciáních derivací 83
6.3 Integrování ve více proměnných 86

7 Řešení diferenciálních rovnic **89**
7.1 Analytické řešení 89
7.2 Numerické řešení 92

8 Statistické metody **95**
8.1 Intevalové odhady 97
8.2 Testování hypotéz 100
8.3 Lineární regrese 102

Úvod

Tento text je zaměřen na základy numerické aproximace a řešení nelineárních rovnic, Bližší informace o teoretickém základu, na němž tento text staví může čtenář najít zejména ve skriptech [1] a [2], dále také v knihách [3] nebo [4].

Kapitola 1

Polynomy

Nebudeme se zda zabývat příliš teorií a základními pojmy jako např. stupeň polynomu apod. Literatura v tomto směru je velmi obsáhlá a čtenáři jistě nebude činit pražádné potíže si nějakou vyhledat. Nás budou zajímat především výpočty.

První věc, na kterou bych rád upozornil, je nejednotnost zápisu polynomů v různé literatuře. Zpravidla se setkáme se zápisem

$$P(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$$

ale občas narazíme na opačné pořadí indexů, tedy na polynom ve tvaru

$$P(x) = a_0 x^n + a_1 x^{n-1} + \cdots + a_{n-1}x + a_n.$$

Pro konkrétní polynom, kdy koeficienty nejsou vyjádřeny obecně ale čísly, je to samozřejmě jedno, ale pokud chceme pro polynom použít nějaké tvrzení, je potřeba si dávat pozor, v jakém pořadí ta či ona věta uvádí pořadí koeficientů. V této příručce se budu držet prvního způsobu, tedy indexy u koeficientů budou stejné jako příslušná mocnina x. Pokud bude potřeba zvýraznit stupeň polynomu, budeme používat značení P_n.

1.1 Hornerovo schema a základní úkony s polynomy

Na polynom budeme nahlížet jako na funkci – vrazíte do ní x a vypadne funkční hodnota podle daného vztahu. Základní věc, kterou tedy s polynomem potřebujeme dělat, je výpočet funkční hodnoty. Nejrychlejší metodou

k tomu je tzv. Hornerovo schema. Nebudu uvádět jeho podrobné odvození, které by pro čtenáře bylo snadným cvičením. Hornerovo schema je založeno na dělení polynomu P polynomem prvního stupně $P_1(x) = x - c$, tedy

$$P(x) = (x - c) \cdot Q(x) + A, \tag{1.1}$$

kde Q je podíl (polynom stupně o jedna menší než P) a A je zbytek po dělení, který musí mít stupeň menší než dělitel, tedy je to konstanta. Z uvedeného zápisu je jasné, že $P(c) = A$, tedy tento zbytek po dělení je současně hodnota polynomu P v bodě c. Z (1.1) se porovnáním koeficientů u stejných mocnin dají odvodit vztahy pro koeficienty polynomu Q, Předpokládáme-li polynom Q ve tvaru

$$Q(x) = b_{n-1}x^{n-1} + \cdots + b_1 x + b_0$$

dostáváme postupně

$$\begin{aligned}
b_{n-1} &= a_n \\
b_{n-2} &= a_{n-1} + c \cdot b_{n-1} \\
&\vdots \\
b_{k-1} &= a_k + c \cdot b_k \\
&\vdots \\
b_0 &= a_1 + c \cdot b_1 \\
A &= a_0 + c \cdot b_0.
\end{aligned}$$

Hornerovo schema zpravidla v literatuře nalezname coby následujíccí tabulku:

	a_n	a_{n-1}	a_{n-2}	$\cdots$	a_2	a_1	a_0
c	b_{n-1}	b_{n-2}	b_{n-3}	$\cdots$	b_1	b_0	A

Pravidlo pro zapamatování výpočetního algoritmu je: „*První číslo opíšeme ($b_{n-1} = a_n$), každé další dostaneme tak, že k číslu nad čarou připočteme c násobek předchozího čísla ($b_{k-1} = a_k + c \cdot b_k$)*“.

Hornerovo schema se při ručních výpočtech nejčastěji používá k testování, zda dané číslo c je kořenem polynomu, v tom případě vyjde $A = 0$. Ale vidíme, že kromě hodnoty polynomu v bodě c dostáváme i podíl – polynom Q.

Někoho možná napadne, co by se stalo, kdybychom Hornerovo schema použili se stejnou hodnotou c znovu, tentokrát na polynom Q, pak znovu na výsledný podíl a tak dál. Pro tento účel si označíme polynom Q jako Q_1 a

hodnotu A jakožto A_0, v dalším kroku dostaneme podíl Q_2 a hodnotu A_1, a tak dál, takže obecně máme

$$Q_k(x) = (x - c) \cdot Q_{k+1}(x) + A_k.$$

Hornerovo schema pak (symbolicky zkráceno) vypadá takto:

$$\begin{array}{c|ccccc} & & & P & & \\ \hline c & & & Q_1 & & A_0 \\ c & & Q_2 & & A_1 & \\ c & Q_3 & & A_2 & & \\ \vdots & & \iddots & & & \\ c & A_n & & & & \end{array}$$

Pro polynom P pak dostáváme

$$\begin{aligned} P(x) &= (x-c)Q_1(x) + A_0 = \\ &= (x-c)((x-c)Q_2(x) + A_1) + A_0 = \\ &= (x-c)^2 Q_2(x) + A_1(x-c) + A_0 = \\ &= (x-c)^2((x-c)Q_3(x) + A_2) + A_1(x-c) + A_0 = \\ &= (x-c)^3 Q_3(x) + A_2(x-c)^2 + A_1(x-c) + A_0 = \cdots = \\ &= A_n(x-c)^n + A_{n-1}(x-c)^{n-1} + \cdots + A_1(x-c) + A_0 \end{aligned}$$

Hodnoty $A_n, \ldots, A_0$ jsou tedy koeficienty polynomu P posunutého do bodu c. Toto vyjádření se dá také velmi dobře použít pro výpočet derivací polynomu P v bodě c, ale to bychom předbíhali.

V Matlabu definujeme polynom pomocí vektoru jeho koeficientů od nejvyšší mocniny, takže příkazem

```
>> P=[1 3 -2 0 5]
```

definujeme polynom $P(x) = x^4 + 3x^3 - 2x^2 + 5$, jak se dá snadno ověřit převodem polynomu na symbolický objekt:

```
>> poly2sym(P)
ans =
x^4 + 3*x^3 - 2*x^2 + 5
```

Pro výpočet hodnoty polynomu v bodě použijeme funkci `polyval` a můžeme ji aplikovat nejen na jednu hodnotu ale na vektor či matici hodnot, přičemž hodnoty polynomu se počítají pro každou složku zvlášť. Takže graf polynomu na intervalu můžeme získat třeba pomocí příkazů

```
>> P=[1 3 -2 0 5];
>> x=-3:0.01:3;
>> y=polyval(P,x);
>> plot(x,y)
```

Operace s polynomy

Mezi základní úkony, které lze s polynomy provádět, patří sčítání, násobení skalárem, vzájemné násobení a dělení se zbytek. První dvě operace patrně nepotřebují další komentáře, jen je potřeba vědět, že pro sčítání polynomů v Matlabu musí mít příslušné vektory koeficientů stejnou délku, takže je nutné je doplnit v případě potřeby nulami. Pro násobení polynomů se používá funkce `conv`, kde jako vstupní parametry zadáme polynomy, které chceme násobit.

Dělení polynomu se zbytkem se provádí podobně, jako je tomu u celých čísel. Nejznámějším použitím dělení polynomu se zbytkem je Eukleidův algoritmus pro hledání největšího společného dělitele dvou polynomů. Algerotmus je všeobecně známý, proto jej zde nebudeme uvádět. Pro dělení se zbytkem v Matlabu je možné použít funkci `deconv`, která má dva vstupní a dva výstupní parametry. Na výstupu dostáváme podíl a zbytek po dělení. Například při dělení polynomu $x^4 + 3x^3 - 4x + 1$ polynomem $x^2 + 1$ dostáváme

```
>> P=[1 3 0 -4 1];
>> Q=[1 0 1];
>> [D,R]=deconv(P,Q)
D =
     1     3    -1
R =
     0     0     0    -7     2
```

Zbytek po dělení `R` má formálně stejný stupeň jako dělenec `P`, aby platila rovnost `P=D*Q+R`. Tuhle skutečnost je potřeba brát v potaz při některých výpočtech v Matlabu, například právě u zmíněného Eukleidova algoritmu.

1.1.1 Derivace polynomu

V této části trochu předběhneme učivo matematického kursu, protože derivace polynomu se nám bude hodit v dalším výkladu a navíc pro polynomy se počítá poměrně jednoduše. Obecně lze říci, že derivace funkce udává, jak rychle se funkce mění. Tedy pokud funkce hodně rychle roste, má derivace velké hodnoty, pokud se funkce na nějakém intervalu nemění (je konstantní), je tam její derivace nulová a tak podobně. Derivace je kladná tam, kde funkce roste, záporná, když funkce klesá, v bodech, kde má funkce lokální minimum nebo maximum, je derivace nulová. Pro polynom

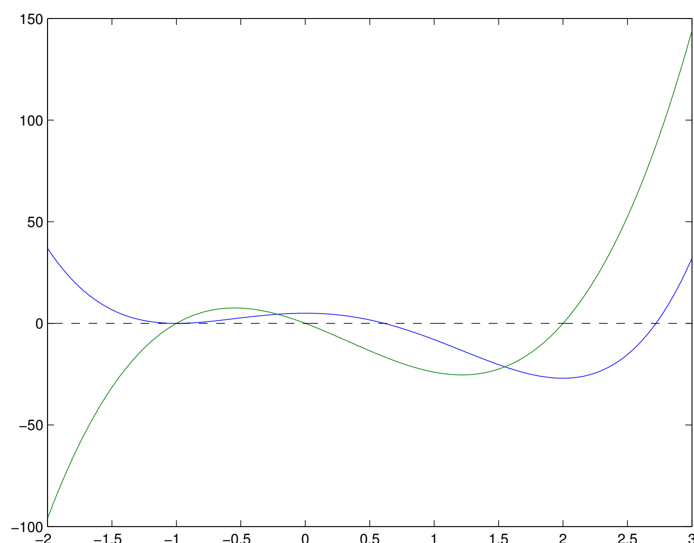
$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

se jeho derivace vypočítá podle vztahu

$$P'(x) = n \cdot a_n x^{n-1} + (n-1) \cdot a_{n-1} x^{n-2} + \cdots + 2 \cdot a_2 x + a_1$$

V Matlabu se derivace vypočítá příkazem `polyder`, takže pokud si chceme zobrazit polynom spolu s jeho derivací, můžeme to udělat například takto:

```
>> P=[3 -4 -12 0 5];
>> DP=polyder(P)
DP =
    12   -12   -24     0
>> x=-2:0.01:3;
>> y=polyval(P,x);
>> dy=polyval(DP,x);
>> z=zeros(size(x));
>> plot(x,y,x,dy,x,z,'k--')
```



Modře je zobrazen polynom, zeleně jeho derivace.

1.2 Interpolace

Problém interpolace patří do teorie aproximace. Zpravidla se snažíme aproximovat nějakou funkci, z níž známe jenom hodnoty v diskrétních bodech, někdy je dokonce můžeme mít nepřesné.

Lagrangeova interpolace
Předpokládejme, že jsou dány body x_i, $i = 0, 1, \ldots, n$, $x_i \neq x_k$ pro $i \neq k$ a hodnoty funkce f v těchto bodech: $f(x_i) = f_i$, $i = 0, 1, \ldots, n$. Hledáme polynom P_n stupně nejvýše n takový, že

$$P_n(x_i) = f_i, \qquad i = 0, 1, \ldots, n.$$

Body x_i, $i = 0, 1, \ldots, n$, $x_i \neq x_k$ pro $i \neq k$, budeme nazývat *uzly*, polynom P_n *interpolační polynom.* Platí následující tvrzení, které se dá poměrně snadno dokázat:

Pro $(n + 1)$ *daných dvojic čísel*

$$(x_i, f_i), \qquad i = 0, 1, \ldots, n, \quad x_i \neq x_k \text{ pro } i \neq k,$$

existuje nejvýše jeden polynom P_n *stupně menšího nebo rovného* n *takový, že*

$$P_n(x_i) = f_i, \qquad i = 0, 1, \ldots, n.$$

Pro důkaz existence interpolačního polynomu použijeme Lagrangeovu konstrukci, podle níž se interpolační polynom nazývá Lagrangeův:

Položme

$$l_i(x) = \frac{(x - x_0)\dots(x - x_{i-1})(x - x_{i+1})\dots(x - x_n)}{(x_i - x_0)\dots(x_i - x_{i-1})(x_i - x_{i+1})\dots(x_i - x_n)} = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{(x - x_j)}{(x_i - x_j)}.$$

Je zřejmé, že l_i je polynom stupně n a

$$l_i(x_j) = \begin{cases} 0 & \text{pro } i \neq j \\ 1 & \text{pro } i = j. \end{cases}$$

Definujme nyní Lagrangeův interpolační polynom P_n vztahem:

$$P_n(x) = l_0(x)f_0 + l_1(x)f_1 + \ldots + l_n(x)f_n = \sum_{i=0}^{n} l_i(x)f_i.$$

Snadno se ověří, že tento polynom splňuje dané interpolační podmínky a je polynomem stupně nejvýše n.

Polynomy l_i se nazývají Lagrangeovy fundamentální polynomy a tvoří bázi prostoru polynomů stupně nejvýše n.

Podívejme se, jaké je výpočetní složitost Konstrukce Lagrangeova interpolačního polynomu. Při konstrukci jednoho fundamentálního polynomu začínáme polynomem prvního stupně $x - x_0$, který postupně násobíme členy $x-x_j$, stupeň polynomu se postupně zvětšuje a pro jeho násobení potřebujeme řádově tolik operací, jaký je jeho stupeň. Celkový počet operací pro konstrukci l_i tedy dostaneme jako součet konečné aritmetické řady, což je řádově n^2 operací. Abychom sestrojili všechny fundamentální polynomy potřebujeme tedy řádově n^3 operací.

Při ruční konstrukci interpolačního polynomu ovšem zjistíme, že spoustu operací provádím opakovaně, takže při vhodném postupu by bylo možné konstrukci urychlit. A skutečně, optimální konstrukce Lagrangeova interpolačního polynomu je založena na funkci $\omega_{n+1}(x) = (x - x_0)\dots(x - x_n) = \prod_{i=0}^{n}(x - x_i)$. Pro určení funkce ω_{n+1} potřebujeme řádově také n^2 operací, ale tuto funkci konstruujeme jen jedenkrát. Čitatel fundamentálního polynomu l_i získáme dělením $\omega_{n+1}(x) : (x - x_i)$, k čemuž se dá využít Hornerovo schema, které je co se týče počtu operací lineární.

Dále se dá odvodit, že jmenovatel fundamentálního polynomu l_i je roven derivaci funkce ω_{n+1} v bodě x_i. Pro derivování polynomu je potřeba také řádově n operací a pro výpočet hodnoty derivace v bodě jakbysmet. Jednodušševji se dá ovšem jmenovatel spočítat na základě faktu, že je roven hodnotě čitatele v uzlu x_i. Určení hodnoty polynomu v bodě ale také dokážeme určit v linerání závislsti na stupni polynomu. Celkově tedy pro jeden fundamentální polynom potřebujeme jen lineární množství operací, pro všechny fundamentální polynomy je to pak řádově n^2 operací.

Matlabovská funkce pro konstrukci Lagrangeova interpolačního polynomu pak může vypadat následovně:

```
function [lip,lfp] = laintpol2(uzly,ff)
% function [lip,lfp] = laintpol(uzly,ff)
% Lagrangeuv interpolacni polynom
% uzly,ff - radkove vektory
% lip - Lagr. interpol. polynom
% lfp - matice fundamentalnich polynomu

om=poly(uzly);   % funkce omega
omd=polyder(om); % omega'
n=length(uzly)-1;
lip=zeros(1,n+1);
lfp=[];
for ii=0:n
    i1=ii+1;
    xi=uzly(i1);
    citatel=deconv(om,[1,-xi]);
    jmenovatel=polyval(omd,xi);
    li=1/jmenovatel*citatel; % fundamentalni polynom
    lfp=[lfp;li];
end % konec cyklu
lip=ff*lfp;
end % konec funkce
```

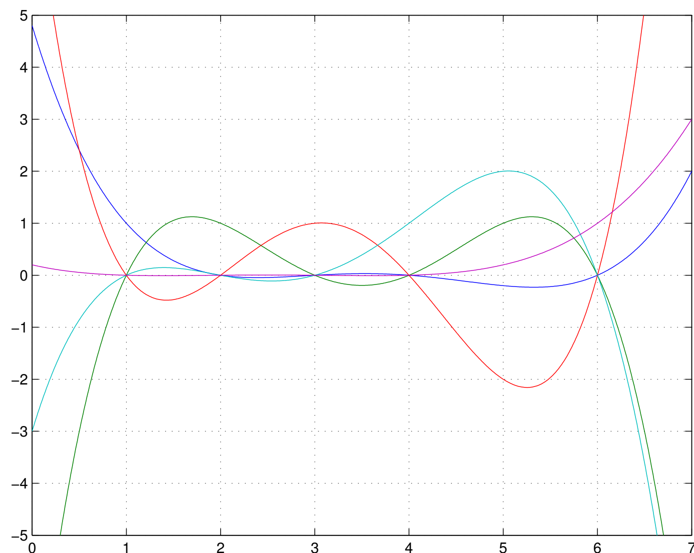
Jako nepovinný druhý výstupní parametr funkce vrací fundamentální polynomy řádcích matice.

Pro vyzkoušení programu zkusíme interpolovat hodnoty funkce sin. Nejprve definujeme uzly a určíme funkční hodnoty:

```
>> uzly=[1 2 3 4 6];
>> ff=sin(uzly);
>> [lip,lfp] = laintpol2(uzly,ff);
```

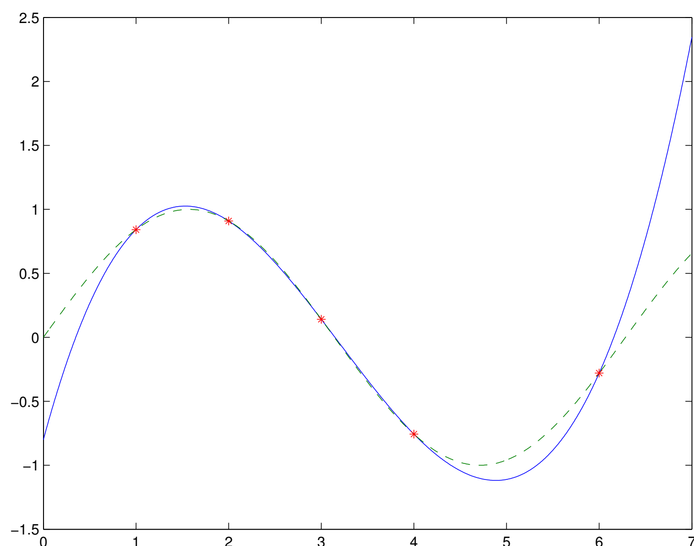
Protože máme spočtené i Lagrangeovy fundamentální polynomy, můžeme si je prohlédnout:

```
>> l1=lfp(1,:);
>> l2=lfp(1,:);
>> l2=lfp(2,:);
>> l3=lfp(3,:);
>> l4=lfp(4,:);
>> l5=lfp(5,:);
>> xx=0:0.01:7;
>> L1=polyval(l1,xx);
>> L2=polyval(l2,xx);
>> L3=polyval(l3,xx);
>> L4=polyval(l4,xx);
>> L5=polyval(l5,xx);
>> plot(xx,[L1;L2;L3;L4;L5])
>> grid on
>> axis([0,7,-5,5])
```



Nakonec si necháme vykreslit i interpolační polynom spolu s funkcí sin pro porovnání:

```
>> Px=polyval(lip,xx);
>> fx=sin(xx);
>> plot(xx,Px,xx,fx,'--',uzly,ff,'*')
```



Vidíme, že polynom funkci aproximuje poměrně dobře, chyba je větší v oblasti, kde jsou uzly dále od sebe. Chyba se samozřejmě zvětšuje vně intervalu obsahující uzly.

Hlavní nevýhodou Lagrangeovy konstrukce je ten fakt, že v případě, když potřebujeme přidat uzel, musíme přepočítat všechny fundamentální polynomy. Proto je v některých případech výhodnější Newtonova konstrukce, kterou si teď odvodíme. Je několik způsobů, jak to udělat, my použijeme tzv. iterovanou interpolaci.

Iterovaná a Newtonova interpolace

Označme $P_{i,i+1,\ldots,i+k}$ interpolační polynom na uzlech $x_i,\ldots,x_{i+k}$, tedy polynom stupně k. Pro $k = 0$ dostáváme polynom nultého stupně P_i, tedy se jedná o konstantní polynom, který je všude roven hodnotě f_i. Polynom $P_{i,i+1,\ldots,i+k}$ sestrojíme z polynomů nižších stupňů následujícím způsobem:
Nechť $P_{i,\ldots,i+k-1}$ a $P_{i+1,\ldots,i+k}$ jsou interpolační polynomy na příslušných uz-

lech, oba stupně $k-1$. Pak polynom $P_{i,i+1,\dots,i+k}$ získáme ze vztahu

$$P_{i,\dots,i+k}(x) = \frac{1}{x_{i+k}-x_i}\begin{vmatrix} P_{i+1,\dots,i+k}(x) & P_{i,\dots,i+k-1}(x) \\ x-x_{i+k} & x-x_i \end{vmatrix} =$$

$$= \frac{1}{x_{i+k}-x_i}[P_{i+1,\dots,i+k}(x)\cdot(x-x_i) - P_{i,\dots,i+k-1}(x)\cdot(x-x_{i+k})].$$

Dosazením jednotlivých uzlů do této formule snadno zjistíme, že výsledný polynom skutečně splňuje interpolační podmínky.

Uvedený vztah trochu upravíme:

$$\begin{aligned} P_{i,\dots,i+k}(x) &= \frac{1}{x_{i+k}-x_i}[P_{i,\dots,i+k-1}(x)\cdot((x_{i+k}-x_i)-(x-x_i)) + \\ &+ P_{i+1,\dots,i+k}(x)\cdot(x-x_i)] \\ &= P_{i,\dots,i+k-1}(x) + \frac{P_{i+1,\dots,i+k}(x)-P_{i,\dots,i+k-1}(x)}{x_{i+k}-x_i}(x-x_i) \end{aligned}$$

Rozdíl polynomů v čitateli zlomku je samozřejmě polynom stupně $k-1$, označme jej například R. Protože jak polynom $P_{i,\dots,i+k-1}$ tak i $P_{i+1,\dots,i+k}$ mají stejné funkční hodnoty v uzlech $x_{i+1},\dots,x_{i+k-1}$, jsou tyto uzly kořeny polynomu R. Protože těchto uzlů je $k-1$ lze polynom R zapsat ve tvaru

$$R(x) = a\cdot(x-x_{i+1})\cdots(x-x_{i+k-1}),$$

kde a je koeficient u nejvyšší mocniny x, tedy u x^{k-1}. Tento koeficient je ale roven rozdílu koeficientů u nejvyšší mocniny polynomů $P_{i+1,\dots,i+k}$ a $P_{i,\dots,i+k-1}$. Celkem dostáváme

$$P_{i,\dots,i+k}(x) = P_{i,\dots,i+k-1}(x) + \frac{a}{x_{i+k}-x_i}(x-x_i)\cdots(x-x_{i+k-1}). \tag{1.2}$$

Interpolační polynom na uzlech $x_i,\dots,x_{i+k}$ tedy dostaneme z interpolačního polynomu na uzlech $x_i,\dots,x_{i+k-1}$ přidáním dalšího členu, který je roven součinu kořenových činitelů $\prod_{j=0}^{k-1}(x-x_{i+j})$ vynásobenému koeficientem u nejvyšší mocniny.

Označme tento koeficient jako $f[x_i,\dots,x_{i+k}]$. Je jasné, že $f[x_i]=f_i$ a dále ze vztahu (1.2) dostáváme

$$f[x_i,\dots,x_{i+k}] = \frac{f[x_{i+1},\dots,x_{i+k}]-f[x_i,\dots,x_{i+k-1}]}{x_{i+k}-x_i}.$$

Způsob výpočtu těchto koeficientů jim dává také název - označujeme je jako poměrné diference. Jejich výpočet pro všechny uzly lze provést pomocí následujícího schematu:

$$\begin{array}{c|lllll}
x_0 & f[x_0]=f_0 & & & & \\
 & & > f[x_0,x_1] & & & \\
x_1 & f[x_1]=f_1 & & > f[x_0,x_1,x_2] & & \\
 & & > f[x_1,x_2] & & \ddots & \\
x_2 & f[x_2]=f_2 & & & & f[x_0,\ldots,x_n] \\
\vdots & \quad\vdots & & & \iddots & \\
\vdots & \quad\vdots & & > f[x_{n-2},x_{n-1},x_n] & & \\
 & & > f[x_{n-1},x_n] & & & \\
x_n & f[x_n]=f_n & & & &
\end{array}$$

Postupným použitím vztahu (1.2) pak dostáváme

$$\begin{array}{rcl}
P_{0,\ldots,n}(x) & = & f[x_0]+f[x_0,x_1](x-x_0)+ \\
 & + & f[x_0,x_1,x_2](x-x_0)(x-x_1)+ \\
 & + & \ldots+f[x_0,\ldots,x_n](x-x_0)\cdots(x-x_{n-1}).
\end{array} \tag{1.3}$$

Tato konstrukce se nazývá Newtonův interpolační polynom. Výsledek samozřejmě musí být totožný s Lagrangeovým interpolačním polynomem. Hlavní výhodou Newtonovy konstrukce ale je snadné přidávání dalších uzlů – stačí vypočítat další řádek v tabulce pro výpočet poměrných diferencí a k předchozímu interpolačnímu polynomu přidat součin příslušných kořenových činitelů násobený výslednou poměrnou diferencí.

Pro interpolaci v Matlabu je možné použít také jeho vlastní funkci `polyfit`, která je ale trochu obecnější a umožňuje najít také aproximaci dat polynomem nižšího stupně pomocí metody nejmenších čtverců.

Intrpolovat lze samozřejmě take v Sage. Abychom dostali jako výsledek racionální polynom, použijeme pro příklad jiná data než výše uvedená:

```
sage: P = PolynomialRing(QQ, 'x')
sage: P.lagrange_polynomial([(0,1),(2,2),(3,-2),(-4,9)])
-23/84*x^3 - 11/84*x^2 + 13/7*x + 1
```

Dají se zde určit i poměrné diference pro Newtonův interpolační polynom:

```
sage: P.divided_difference([(0,1),(2,2),(3,-2),(-4,9)])
[1, 1/2, -3/2, -23/84]
```

Předpokádám, že čtenáři nečiní žádné potíže si ověřit, že z vypočítaných nodnot pomocí Newtonovy konstrukce získáme stejný polynom jako v prvním případě.

1.3 Kořeny polynomů

Najít kořen daného polynomu je jednou ze základních matematických úloh. K ověřování, zda dané číslo je nebo není kořenem polynomu zpravidla používáme Hornerovo schema, existují ale další nástroje, které nám mohou hledání kořenů usnadnit.

Například pokud máme polynom s celočíselnými koeficienty a zajímají nás racionální kořeny, tedy kořeny ve tvaru $\frac{p}{q}$, kde p je celé číslo, q je přirozené číslo a p a q jsou nesoudělná, tak se dá lehce zjistit, že koeficient u nejvyšší mocniny polynomu (tedy a_n) musí být dělitelný číslem q, zatímco absolutní člen (t.j. a_0) musí být dělitelný p.

Další pomůckou může být stanovení oblasti, kde všechny kořeny leží. K tomu nám může pomoci například tvrzení, které lze najít i s důkazem v [1].

Nechť

$$\begin{aligned} A &= \max\left(|a_0|, \ldots, |a_{n-1}|\right), \\ B &= \max\left(|a_1|, \ldots, |a_n|\right), \end{aligned}$$

kde a_k, $k = 0, 1, \ldots, n$, $a_0 a_n \neq 0$, *jsou koeficienty polynomu* P, *Pak pro všechny kořeny* $x_k, k = 0, 1, \ldots, n$, *polynomu* P *platí*

$$\frac{1}{1 + \dfrac{B}{|a_0|}} \leq |x_k| \leq 1 + \frac{A}{|a_n|}. \tag{1.4}$$

Všechny kořeny tedy v komplexním oboru leží v mezikruží, jehož hranice jsou dány uvedenou nerovností. Uvedené hranice, zejména pak horní hranice, jsou poměrně hodně hrubě odhadnuty. Například pro polynom, jehož kořeny jsou čísla od jedné do šesti, dostaneme následující hranice:

```
>> P=poly(1:6)
P =
  Columns 1 through 6
      1    -21    175   -735    1624     -1764
  Column 7
      720
>> n=length(P);
>> A=max(abs(P(2:n)))
A =
        1764
>> B=max(abs(P(1:n-1)))
B =
        1764
>> H=1+A/P(1) % horni hranice
H =
        1765
>> D=1/(1+B/P(1)) % dolni hranice
D =
   5.6657e-04
```

Existují další kriteria, která v některých případech mohou poskytnout lepší odhad velikosti absolutní hodnoty kořenů, uveďme aspoň některá:
Pro všechny kořeny $x_k, k = 0, 1, \ldots, n$, *polynomu* P *platí*

$$
\begin{aligned}
|x_k| &\leq \max\left\{1, \sum_{j=0}^{n-1}\left|\frac{a_j}{a_n}\right|\right\} \\
|x_k| &\leq 2\max\left\{\left|\frac{a_{n-1}}{a_n}\right|, \sqrt{\left|\frac{a_{n-2}}{a_n}\right|}, \ldots, \sqrt[n]{\left|\frac{a_0}{a_n}\right|}\right\} \\
|x_k| &\leq \max\left\{\left|\frac{a_0}{a_n}\right|, 1+\left|\frac{a_1}{a_n}\right|, \ldots, 1+\left|\frac{a_{n-1}}{a_n}\right|\right\}.
\end{aligned}
$$

Dalším zjednodušením při hledání kořenů polynomu může být odstranění násobných kořenů. Obecně platí, že každý polynom lze zapsat ve tvaru

$$P(x) = a_n(x - x_1)^{n_1} \cdots (x - x_k)^{n_k},$$

kde $x_1,\ldots,x_k$ jsou vzájemně různá komplexní čísla, $n_1,\ldots,n_k$ jsou jejich násobnosti a $n_1+\ldots+n_k = n = st(P)$. Platí také, že pokud je kořen x_i násobnosti $n_i > 1$, pak tento kořen je i kořenem derivace polynomu P s násobností $n_i - 1$. Odtud plyne, že snížit násobnost všech kořenů na 1 lze tak, že určíme největší společný dělitel D polynomu P a jeho derivace P'. Kořeny tohoto největšího společného dělitele jsou právě ty kořeny původního polynomu, které mají násobnost větší než jedna a jejich násobnost v děliteli D je o jedna menší než v polynomu P. Vydělením polynomu P dělitelem D získáme tedy polynom, který má stejné kořeny jako P, ale všechny jsou násobnosti 1.

Z numerického hlediska je potřeba poznamenat, že tento postup v praxi funguje dobře pro polynomy se celočíselnými koeficienty, které se pohybují v „rozumných“ mezích. Ale i v jednoduchých případech je potřeba postupovat obezřetně:

```
>> format long
>> P=poly([1 1 1 1])
P =
     1    -4     6    -4     1
>> DP=polyder(P)
DP =
     4   -12    12    -4
>> roots(P)
ans =
  1.000217162530750
  0.999999969335793 + 0.000217131857745i
  0.999999969335793 - 0.000217131857745i
  0.999782898797672
>> roots(DP)
ans =
  1.000004930958320 + 0.000008540746793i
  1.000004930958320 - 0.000008540746793i
  0.999990138083364
```

Vidíme, že pokud bychom posuzovali shodnost kořenů u uvedeného polynomu a jeho derivace, přičemž ani nepožadujeme příliš velkou přesnost, tak nejenže ke shodě nedochází, ale kořeny ani nejsou násobné. Nicméně pokud bychom hledali největší společný dělitel, postup povede ke správnému výsledku:

```
>> [Q,R]=deconv(P,DP)
```

```
Q =
   0.250000000000000  -0.250000000000000
R =
     0     0     0     0     0
>> D=Q % D je nejv.spol.delitel
D =
   0.250000000000000  -0.250000000000000
>> roots(D)
ans =
     1
```

Vidíme, že výsledek je přesný, a to i navzdory numerickým nepřesnostem během výpočtu. Ukažme si ještě jeden příklad, kde budou dva násobné kořeny blízko sebe:

```
>> P=poly([0.9 0.9 1.1 1.1 1.1])
P =
 1.0000 -5.1000  10.3800  -10.5380  5.3361 -1.0781
>> DP=polyder(P)
DP =
 5.0000 -20.4000  31.1400 -21.0760  5.3361
>> [Q1,R1]=deconv(P,DP)
Q1 =
  0.2000 -0.2040
R1 =
    0    0   -0.0096    0.0298   -0.0306    0.0105
>> R1(1:2)=[] % odstranime pocatecni nuly
R1 =
   -0.0096    0.0298   -0.0306    0.0105
>> [Q2,R2]=deconv(DP,R1)
Q2 =
 -520.8333  510.4167
R2 =
   1.0e-11 *
         0         0   -0.1766    0.2515   -0.0769
```

V tomto místě je nutné usoudit, že zbytek po dělení je ve skutečnosti nulový, tudíž nějvětší společný dělitel je předchozí zbytek, tedy polynom $R1$.

```
>> D=R1
D =
   -0.0096    0.0298   -0.0306    0.0105
>> P0=deconv(P,D)
P0 =
 -104.1667  208.3333 -103.1250
>> roots(P0)
ans =
    1.1000
    0.9000
>> format long
>> roots(P0)
ans =
   1.100000000001784
   0.899999999998279
```

Vidíme, že výsledek je opět poměrně přesný. Kořeny původního polynomu Matlab spočítá s daleko větší chybou:

```
>> roots(P)
ans =
  1.100021372535598 + 0.000037011183255i
  1.100021372535598 - 0.000037011183255i
  1.099957254925198
  0.900000434848828
  0.899999565154781
```

1.3.1 Separace reálných kořenů

Ukážeme si algoritmus, s jehož pomocí je možné přesně určit počet reálných polynomů v daném intervalu, pokud předpokládáme, že všechny reálné kořeny jsou jednoduché. Tento algoritmus je založen na konstrukci tzv. Sturmovy posloupnosti, která se definuje následovně:

Posloupnost reálných polynomů

$$P = P_0, P_1, \ldots, P_m$$

se nazývá Sturmovou posloupností *příslušnou polynomu P, jestliže*

1. *Všechny reálné kořeny polynomu P_0 jsou jednoduché.*
2. *Je-li x_0 reálný kořen polynomu P_0, pak $sign P_1(x_0) = -sign P_0'(x_0)$.*
3. *Jestliže x_0 je reálný kořen polynomu P_i, platí*

$$P_{i+1}(x_0)P_{i-1}(x_0) < 0,$$

   pro $i = 1, 2, \ldots, m-1$.

4. *Poslední polynom P_m nemá reálné kořeny.*

Sturmovu posloupnost lze zkonstruovat poměrně snadno, Pokud předpokládáme, že výchozí polynom $P = P_0$ nemá reálné kořeny, stačí položit $P_1 = -P_0'$, abychom zaručili splnění druhé vlastnosti. Třetí vlastnost z definice dostaneme, pokud polynom P_{i+1} vezmeme jako záporně vzatý zbytek po dělení $P_{i-1} : P_i$,tedy

$$P_{i-1} = P_i \cdot Q - P_{i+1}$$

Pro kořen x_0 polynomu P_i tedy máme $P_{i-1}(x_0) = -P_{i+1}(x_0)$, přičemž daný výraz nemůže být nulový, jinak bychom indukcí dostali, že x_0 je kořenem P_0 i P_1, což je spor s tím, že kořeny P_0 jsou jednoduché.

Konstrukce založená na dělení se zbytkem zaručuje, že stupně polynomů v posloupnosti postupně klesají. Poslední polynom P_m je zpravidla konstantní, ale je možné konstrukci ukončit i dříve, pokud víme, že polynom nemá reálné kořeny, např. pro polynom $x^2 + 1$.

Počet kořenů v daném intervalu pak lze zjistit pomocí Sturmovy věty:

Počet reálných kořenů polynomu P v intervalu $[a, b]$ je roven $W(b) - W(a)$, kde $W(x)$ je počet znaménkových změn ve Sturmově posloupnosti $P_0(x), \ldots, P_m(x)$ v bodě x (z níž jsou vyškrtnuty nuly).

Předpokládám, že pojem *počet znaménkových změn* je natolik intuitivně jasný, že nepotřebuje definici. Důkaz Sturmovy věty je založen na faktu, že k navýšení či snížení počtu znaménkových změn může dojít jen při přechodu přes kořen některého z polynomů ve Sturmově posloupnosti. Ze druhé vlastnosti v definici Sturmovy posloupnosti skutečně plyne, že pokud x roste, pak při přechodu přes kořen $P = P_0$ jsou dvě možnosti: buď přecházíme ze záporných hodnot polynomu do kladných, v tom případě polynom P_0 roste, jeho derivace je tedy v okolí kořene kladná, tudíž P_1 je tam záporný a přibude

jedna znaménková změna. Nebo P_0 v kořenu klesá, takže P_1 je kladný a také přibude jedna znaménková změna.

	$x-h$	x	$x+h$
P_0	$-$	0	$+$
P_1	$-$	$-$	$-$
$W(x)$	0	0	1

	$x-h$	x	$x+h$
P_0	$+$	0	$-$
P_1	$+$	$+$	$+$
$W(x)$	0	0	1

Při přechodu přes kořen polynomu P_i pro $i > 0$ jsou celkem čtyři možnosti, při žádné z nich však podle třetí vlastnosti z definice nedochází ke změně počtu znaménkových změn:

	$x-h$	x	$x+h$
P_{i-1}	$-$	$-$	$-$
P_i	$-$	0	$+$
P_{i+1}	$+$	$+$	$+$
$W(x)$	1	1	1

	$x-h$	x	$x+h$
P_{i-1}	$+$	$+$	$+$
P_i	$-$	0	$+$
P_{i+1}	$-$	$-$	$-$
$W(x)$	1	1	1

	$x-h$	x	$x+h$
P_{i-1}	$-$	$-$	$-$
P_i	$+$	0	$-$
P_{i+1}	$+$	$+$	$+$
$W(x)$	1	1	1

	$x-h$	x	$x+h$
P_{i-1}	$+$	$+$	$+$
P_i	$+$	0	$-$
P_{i+1}	$-$	$-$	$-$
$W(x)$	1	1	1

K navýšení počtu znaménkových změn tak dochází jenom při přechodu přes kořen polynomu P_0, odkud bezprostředně plyne tvrzení věty.

Příklad 1. Zjistíme počet kladných a záporných kořenů polynomu x^4-4x^2+8x-2. Nejprve sestrojím Sturmovu posloupnost:

```
>> P0=[1 -4 0 8 -2];
>> P1=-polyder(P0)
P1 =
    -4    12     0    -8
```

Protože nás zajímají jen znaménka a jejich změny, je možné polynom vydělit nebo vynásobit kladným číslem. To se hodí hlavně při ručním počítání, když se chceme vyhnout složitějším zlomkům. Pak dál pokračujeme v konstrukci Sturmovy posloupnosti: provedeme dělení se zbytkem, odstraníme nulové koeficienty a polynom opět můžeme vydělit kladným číslem:

```
>> P1=P1/4
P1 =
```

```
    -1     3     0    -2
>> [Q,R]=deconv(P0,P1)
Q =
    -1     1
R =
     0     0    -3     6     0
>> P2=-R/3;
>> P2(1:2)=[]
P2 =
     1    -2     0
```

Celou činnost opakujeme, dokud nedostaneme konstantní polynom:

```
>> [Q,R]=deconv(P1,P2)
Q =
    -1     1
R =
     0     0     2    -2
>> P3=-R/2;
>> P3(1:2)=[]
P3 =
    -1     1
>> [Q,R]=deconv(P2,P3)
Q =
    -1     1
R =
     0     0    -1
>> P4=1;
```

Podle předchozího textu je horní hranice pro absolutní hodnotu všech kořenů rovna 9, takže všechny kořeny leží v intervalu $[-9, 9]$. Při ručním počítání je jednodušší určit znaménko limity hodnoty v $\pm\infty$ podle parity nejvyšší mocniny a znaménka jejího koeficientu. Počet znaménkových změn v Matlabu určíme pomocí příkazů:

```
>> x=-9;
>> [polyval(P0,x),polyval(P1,x),polyval(P2,x),...
polyval(P3,x),polyval(P4,x)]
```

```
ans =
     9403       970        99        10         1
>> x=0;
>> [polyval(P0,x),polyval(P1,x),polyval(P2,x),...
polyval(P3,x),polyval(P4,x)]
ans =
    -2    -2     0     1     1
>> x=9;
>> [polyval(P0,x),polyval(P1,x),polyval(P2,x),...
polyval(P3,x),polyval(P4,x)]
ans =
     3715      -488        63        -8         1
```

Při ručním počítání stačí určovat znaménka a výsledky můžeme přehledně umístit do tabulky:

x	$P0(x)$	$P1(x)$	$P2(x)$	$P3(x)$	$P4(x)$	$W(x)$
$-\infty$	+	+	+	+	+	0
0	−	−	0	+	+	1
∞	+	−	+	.	+	4

Celkem tedy máme 4 kořeny, z toho je jeden záporný a tři kladné.

1.4 Splajny

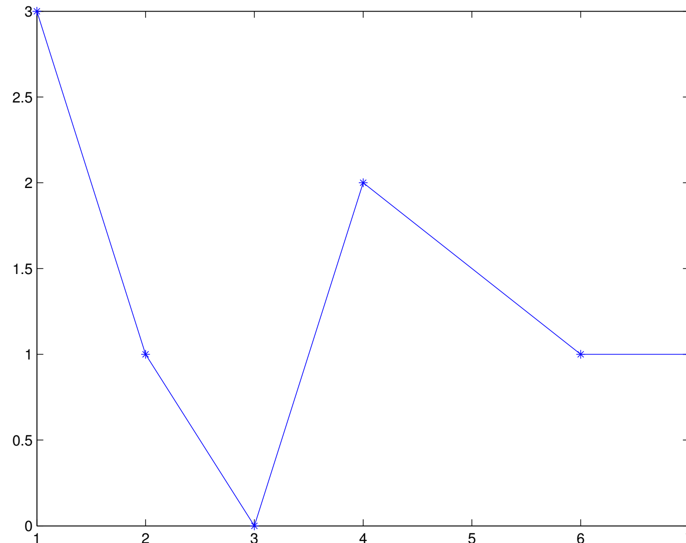
Teď zase trochu předběhneme. budeme totiž opět potřebovat derivace, o kterých budeme podrobněji mluvit později, ovšem výklad o splajnech dobře navazuje na interpolaci, takže jsem se rozhodl jej zařadit už sem. Snad to nebude příliš vadit, protože tato část bude spíše jen doplňková a informativní.

V češtině už se pro funkce, o nichž teď budeme mluvit, zaběhl počeštěný výraz *splajn*. Ve starší literatuře je možné narazit na anglický termín *spline*, výslovnost je ovšem stejná jako u počeštěného výrazu.

Splajny patří mezi interpolační techniky, protože většinou procházejí přesně zadanými hodnotami. Existují ale i tzv. vyhlazovací splajny, o těch tady v tuto chvíli taktně pomlčíme.

Splajn se zpravidla definuje jako po částech polynomiální funkce, která je spojitá do určitého řádu. Nejjednodušším příkladem splajnu je spojitá po částech lineární funkce, která prochází zadanými body v rovině. Přesněji

řečeno předpokládejme opět, že jsou dány body x_i, $i = 0, 1, \ldots, n$, $x_i \neq x_k$ pro $i \neq k$ a hodnoty funkce f v těchto bodech: $f(x_i) = f_i$, $i = 0, 1, \ldots, n$. Lineární spojitý splajn je funkce s, která je spojitá, lineární na každém intervalu $[x_i, x_{i+1}]$, $i = 0, \ldots, n-1$ a platí $s(x_i) = f_i$, $i = 0, \ldots, n$. Na následujícím obrázku můžeme vidět příklad takové funkce:



Nejčastěji se setkáme s kubickými splajny. Jedná se o funkce, které jsou po částech polynomy třetího stupně a jsou spojité do řádu dva, jsou tedy spojité a mají spojité i první a druhé derivace, Uvedeme radši opět přesnou definici: Nechť jsou dány body x_i, $i = 0, 1, \ldots, n$, $x_i \neq x_k$ pro $i \neq k$ (zvané uzly) a hodnoty funkce f v těchto bodech: $f(x_i) = f_i$, $i = 0, 1, \ldots, n$. Kubický splajn je funkce s, která je spojitá včetně první a druhé derivace a je kubickým polynomem na každém intervalu $[x_i, x_{i+1}]$, $i = 0, \ldots, n-1$ a platí $s(x_i) = f_i$, $i = 0, \ldots, n$.
Podmínky spojitosti musíme uplatnit v uzlech, jelikož polynomy mají spojité derivace všech řádů, takže uvnitř intervalů není se spojitostí problém.
Máme celkem n intervalů, na každém potřebujeme sestrojit polynom 3. stupně, který má čtyři neznámé koeficienty, dohromady tedy máme $4n$ neznámých koeficientů. Ovšem pro každý z n polynomů máme zadána dvě funkční hodnoty (jednu v každém krajním bodě intervalu $[x_i, x_{i+1}]$), tím se nám počet neznámých koeficientů redukuje na polovinu a současně zajistíme spojitost kubického splajnu. Ve vnitřních bodech $(x_1, \ldots, x_{n-1})$ dále máme podmínky na spojitost první a druhé derivace, čímž se nám počet neznámých koeficientů zredukuje ne 2. Další podmínky už ale nemáme, takže kubický splan

není funkčními hodnotami jednoznačně zadán a je potřeba dvě podmínky přidat,aby bylo možné jej sestrojit.

Nejčastěji se zadávají hodnoty první derivace v krajních bodech x_0 a s_n, pak hovoříme o úplném kubickém splajnu, nebo hodnoty druhé derivace v těchto bodech. Pokud mají být tyto hodnoty nulové, splajn se nazývá přirozený kubický splan.

Nebudeme tady uvádět přesnou konstrukci kubických splajnů, zájemce najde podrobné informace třeba v [2]. Ukážeme si, jak zkonstruovat splajn za pomocí software. V Matlabu se pro konstrukci splajnu dá použít základní funkce `spline`. V nejjednodušší verzi počítá koeficienty polynomů, jako výsledek pak vrací složitější strukturu, která obsahuje víc informací. Pro data z předchozího obrázku dostaneme:

```
>> x=[1 2 3 4 6 7];
>> f=[3 1 0 2 1 1];
>> plot(x,f,'-*')
>> print -depsc splajn1.eps
>> s=spline(x,f)
s =
      form: 'pp'
    breaks: [1 2 3 4 6 7]
     coefs: [5x4 double]
    pieces: 5
     order: 4
       dim: 1
```

Označení 'pp' ukazuje, že se jedná o po částech polynomiální funkci, Dále následují uzly, tedy hraniční body pro jednotlivé části polynomu. Pak můžeme zjisti koeficienty na jednotlivých intervalech, počet intervalů a stupeň polynomu (ve skutečnosti spíše počet koeficientů na každém intervalu). Význam poslední položky by měl jasný.

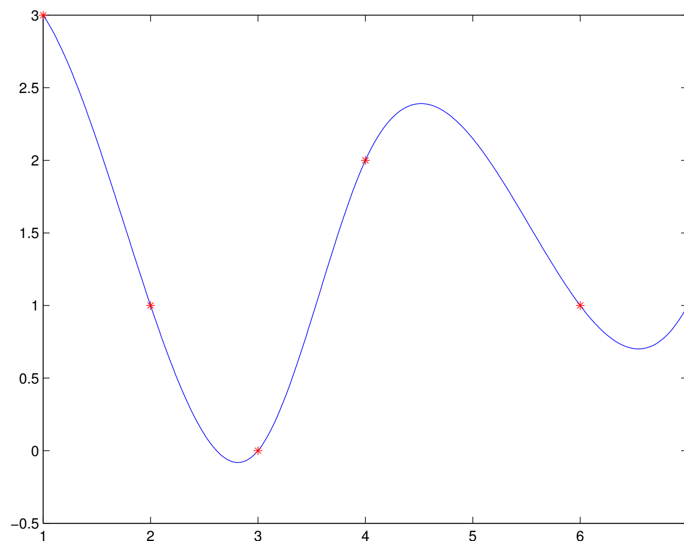
Pokud bychom tedy chtěli znát koeficienty splajnu na jednotlivých intervalech stačí zadat

```
>> s.coefs
ans =
    0.6978   -1.5935   -1.1044    3.0000
    0.6978    0.5000   -2.1978    1.0000
   -1.4891    2.5935    0.8956         0
```

```
    0.4081   -1.8738    1.6153    2.0000
    0.4081    0.5748   -0.9829    1.0000
```

Nás budou ovšem spíš zajímat hodnoty splanu, abychom si jej mohli případně nakreslit. K tomu lze použít funkci `ppval`:

```
>> xx=1:0.01:7;
>> ss=ppval(s,xx);
>> plot(xx,ss,x,f,'r*')
```



V případě, že nás nezajímají koeficienty na jednotlivých intervalech, ale jen výsledné funkční hodnoty, stačí zadat

```
>> ss=spline(x,f,xx);
```

Zvídavého čtenáře teď možná napadne, že jsme při konstrukci zadávali jen funkční hodnoty a žádné okrajové podmínky. Právem se tedy může ptát, jaké okrajové podmínky byly použity. Tuto otázku jistě uspokojivě zodpoví dokumentace Matlabu.

V Sage se splajn definuje pro body v rovině, jimiž má procházet, takže pro stejná data jako výše můžeme použít následující postup:

```
sage: values = [[1,3],[2,1],[3,0],[4,2],[6,1],[7,1]]
sage: S = spline(values)
sage: S
[[1, 3], [2, 1], [3, 0], [4, 2], [6, 1], [7, 1]]
```

Vidíme, že samotný obsah proměnné `S` nám moc informací nedá. Příkazem

```
sage: type(S)
sage.gsl.interpolation.Spline
```

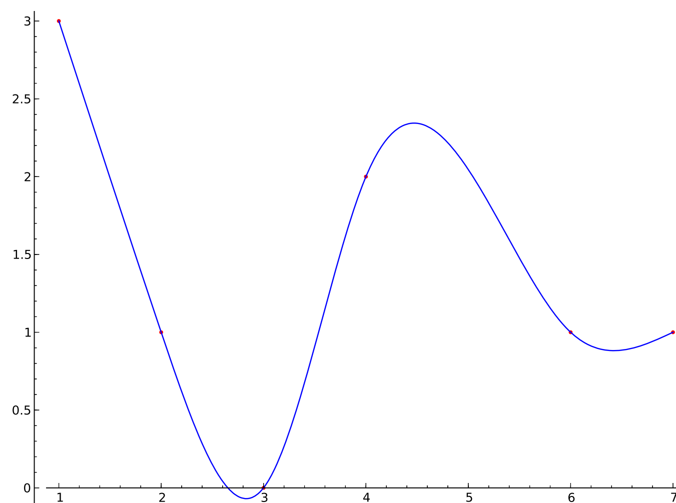
aspoň zjistíme, o jaký datový typ se jedná, ale koeficienty na jednotlivých intervalech nezjistíme. A pokud vím, u této základní funkce se tyto koeficienty přímo zjistit nedají. Pokud by je člověk opravdu potřeboval, musí si stáhnout další knihovny, které obsahují poněkud více sofistikované funkce pro práci se splajny.

Pro naše účely ovšem základní funkce zcela postačuje, například hodnotu polynomu v bodě zjistíme snadno:

```
sage: S(1)
3.0
sage: S(1.5)
1.9917763157894737
sage: S(sqrt(2))
2.1640477491461865
```

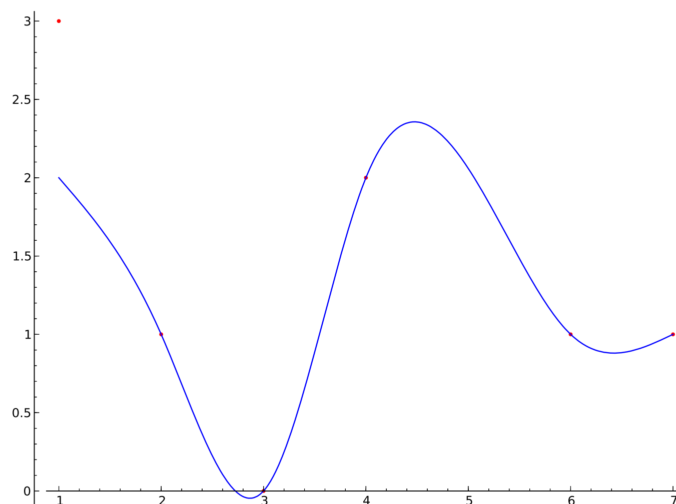
Nechat si splajn vykreslit (případně i s uzly a příslušnými hodnotami) taky není problém:

```
plot(S,(1,7))+list_plot(values,color='red')
```



V Sage je velmi jednoduché změnit hodnotu v některém uzlu:

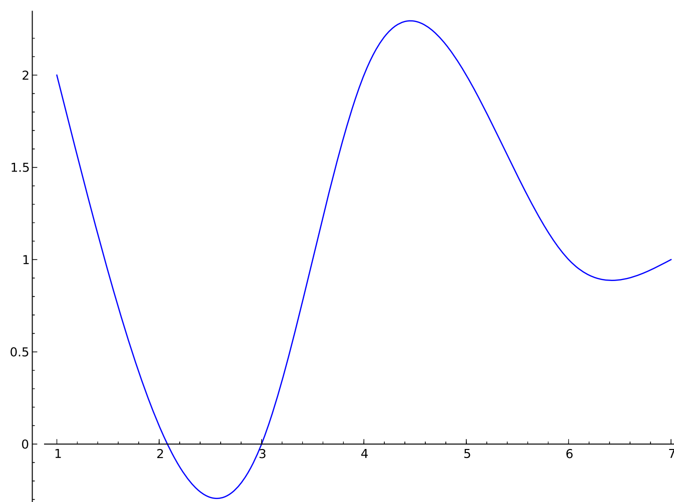
```
sage: S[0]
[1, 3]
sage: S[0]=[1,2]
sage: S
[[1, 2], [2, 1], [3, 0], [4, 2], [6, 1], [7, 1]]
sage: plot(S,(1,7))+list_plot(values,color='red')
```



První hodnota samozřejmě neleží na splajnu, jelikož jsem ji nepředefinovali. Splajn ale prochází danými body. Jednoduše se dá taky bod ubrat nebo naopak přidat:

```
sage: del S[1]
sage: S
[[1, 2], [3, 0], [4, 2], [6, 1], [7, 1]]
sage: S.append([5,2])
sage: S
[[1, 2], [3, 0], [4, 2], [6, 1], [7, 1], [5, 2]]
```

Když si necháme splajn zobrazit, vidíme, že nazáleží na pořadí uzlů.



1.5 Bersteinovy polynomy a Bézierovy křivky

Sergej Bernstein byl ruský matematik, který se během svého dlouhého života dosáhl významných výsledků v několika matematických odvětvích. Jeho jméno by se mělo vyslovovat rusky, tedy „*bernštejn*“ a někdy se takto i v češtině píše. Často se ovšem setkáme s výslovností německou („*bernštajn*“) nebo s anglickou („*bernstain*“).

Nás bude zajímat jeho příspěvek k teorií aproximací. Bernsteinovy polynomy totiž mají tu důležitou vlastnost, že s libovolnou přesností aproximují spojitou funkci na intervalu $[0, 1]$ Jednoduchou substitucí pak jimi můžeme

aproximovat spojitou funkci na libovolném uzavřeném intervalu. Jejich konstrukce je navíc velmi jednoduchá a tedy i snadno naprogramovatelná.

Nechť tedy n je pevně dané přirozené číslo a k je přirozená číslo nebo nula, $k \leq n$. Bersteinův bázový polynom $b_{n,k}$ definujeme vztahem

$$b_{n,k}(x) = \binom{n}{k} x^k (1-x)^{n-k}.$$

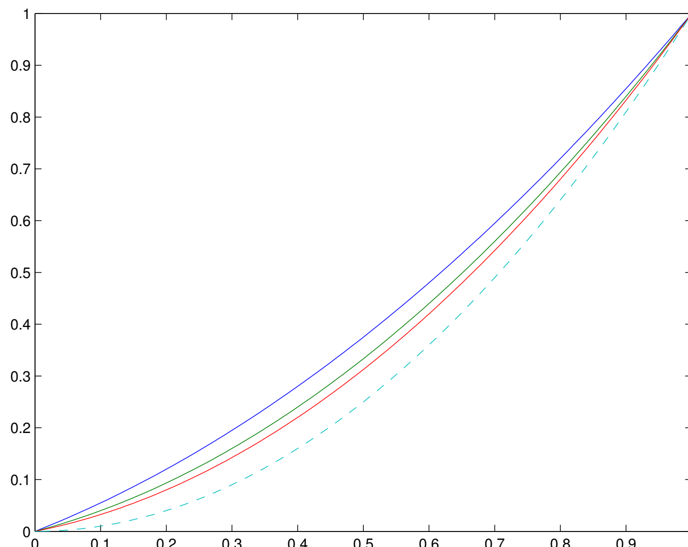
Dále nechť f je funkce definovaná na intervalu $[0,1]$, položme $f_k = f(\frac{k}{n})$ pro $k = 0, \ldots, n$. Bernsteinův polynom stupně n funkce f definujeme výrazem

$$B_{n,f}(x) = \sum_{k=0}^{n} f_k \cdot b_{n,k}(x). \tag{1.5}$$

Snadno se ověří, že

$$\sum_{k=0}^{n} f_k \cdot b_{n,k}(x) = 1,$$

odkud pak bezprostředně plyne, že pro funkci f konstantní na $[0,1]$ platí $f(x) = B_{n,f}(x)$ pro každé $x \in [0,1]$. Totéž platí i v případě, že f je polynomem prvního stupně, jak se dá též poměrně snadno spočítat. Pro polynomy vyšších stupňů to už ale neplatí, jak dokládá následující obrázek, kde čárkovaně je zobrazena funkce x^2, křivky, které se k ní blíží jsou postupně Bernsteinovy polynomy druhého, třetího a čtvrtého stupně.



Tento obrázek byl získán pomocí matlabovského programu pro výpočet Bersteinova polynomu, přesněji řečeno jeho koeficientů. Jako vedlejší produkt dostáváme i Bernsteinovy bázové polynomy uspořádané v matici. Při jejich konstrukci se využívá toho, že daný bázový polynom má kořeny 0 a 1 násobností k a $n - k$.

```
function [BP,B] = bern_pol(fk)
% function BP = bern_pol(fk)
% Bernsteinuv polynom pro hodnoty fk
% B - matice Bern. bazovych polynomu

n=length(fk)-1;
B=[]; % matice bazovych Bernsteinovych polynomu
for k=0:n
    bk=nchoosek(n,k)*poly( [zeros(1,k),ones(1,n-k)]);
    bk=bk*(-1)^(n-k);
    B=[B;bk];
end
BP=fk*B;
end
```

Uvedený obrázek pak dostaneme například pomocí následujícího postupu:

```
>> f=inline('x.^2');
>> n=2;
>> k=0:n;
>> xk=k/n;
>> f2=f(xk);
>> B2=bern_pol(f2);
>> n=3;
>> k=0:n;
>> xk=k/n;
>> f3=f(xk);
>> B3=bern_pol(f3);
>> n=4;
>> k=0:n;
>> xk=k/n;
>> f4=f(xk);
```

```
>> B4=bern_pol(f4);
>> xx=0:0.01:1;
>> Bx2=polyval(B2,xx);
>> Bx3=polyval(B3,xx);
>> Bx4=polyval(B4,xx);
>> plot(xx,[Bx2;Bx3;Bx4],xx,xx.^2,'--')
```

Nejdůležitější vlastností Bernsteinových polynomů, jak už bylo naznačeno, je, že pro funkcí f spojitou na $[0.1]$ konverguje posloupnost $B_{n,f}$ stejnoměrně k f

Kromě teoretického významu se Bernsteinovy polynomy používají i prakticky, totiž ke konstrukci Bězierových křivek. „Legenda“ praví, že tyto křivky objevili nezávisle na sobě dva pracovníci konstrukčních kancelářích dvou renomovaných francouzských automobilek. První objevitel ovšem přišel o prvenství tím, že kvůli konkurenčnímu boji jeho objev podléhal utajení.

V dnešní době jsou Bézierovy křivky běžnou součástí téměř každého kreslícího software, často bez uvedení, o jaký typ křivek se jedná. Nejčastěji můžeme narazit na kubické Bézierovy křivky, jejichž konstrukci si zde ukážeme. Zobecnění pro vyšší stupně je zcela přímé a čtenář by je jistě bez problémů zvládl.

Bézierovy křivky jsou zadány parametricky, konstrukce je navíc v podstatě totožná v rovině i v trojrozměrném prostoru.

Mějme tedy 4 body v rovině, označme je P_0,P_1,P_2,P_3, $P_i = [x_i, y_i]$, tyto body se nazývají kontrolní nebo řídící. Bézierova křivka je množina všech bodů $Q = [x, y)$ pro něž platí

$$\begin{aligned} x = x(t) &= x_0 \cdot b_{3,0}(t) + x_1 \cdot b_{3,1}(t) + x_2 \cdot b_{3,2}(t) + x_3 \cdot b_{3,3}(t) \\ &= x_0(1-t)^3 + 3x_1 t(1-t)^2 + 3x_2 t^2(1-t) + x_3 t^3 \\ y = y(t) &= y_0 \cdot b_{3,0}(t) + y_1 \cdot b_{3,1}(t) + y_2 \cdot b_{3,2}(t) + y_3 \cdot b_{3,3}(t) \\ &= y_0(1-t)^3 + 3y_1 t(1-t)^2 + 3y_2 t^2(1-t) + y_3 t^3, \end{aligned}$$

kde proměnná t probíhá interval $[0, 1]$ Pokud bychom uvažovali body v prostoru, t.j. $P_i = [x_i, y_i, z_i]$, přibyla by nám rovnice pro třetí souřadnici bodu Q:

$$\begin{aligned} z = z(t) &= z_0 \cdot b_{3,0}(t) + z_1 \cdot b_{3,1}(t) + z_2 \cdot b_{3,2}(t) + z_3 \cdot b_{3,3}(t) \\ &= z_0(1-t)^3 + 3z_1 t(1-t)^2 + 3z_2 t^2(1-t) + z_3 t^3 \end{aligned}$$

Souřadnice bodu Q na Bézierově křivce jsou tedy hodnoty Bersteinova polynomu pro příslušné souřadnice kontrolních bodů. Stručně se body na Bézierově křivce dají vyjádřit jako

$$Q = Q(t) = P_0(1-t)^3 + 3P_1t(1-t)^2 + 3P_2t^2(1-t) + P_3t^3$$

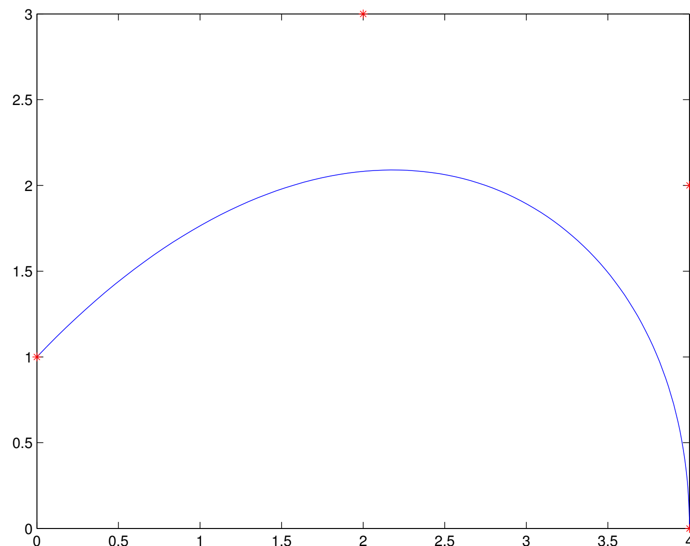
Z vyjádření navíc plyne, že pro $t = 0$ dostaneme výchozí kontrolní bod P_0, pro $t = 1$ koncový kontrolní bod P_3.

Sestrojit Bézierovu křivku v Matlabu je velmi jednoduché, není ani potřeba vytvářet pro tento účel zvláštní funkci:

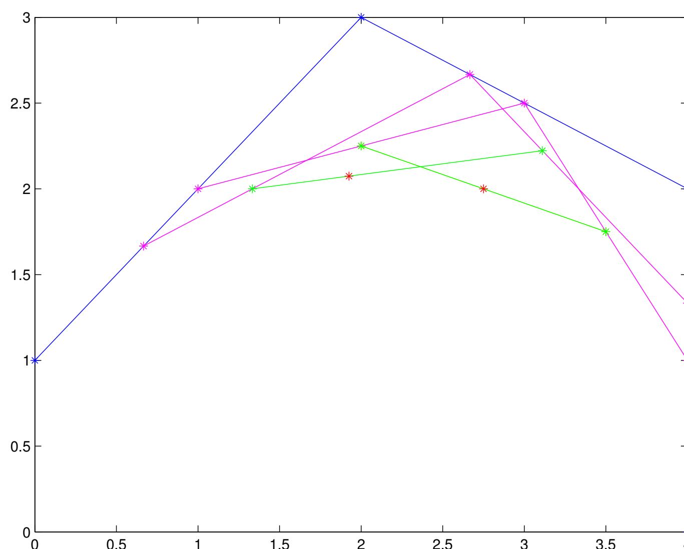
```
>> P0=[0;1]; P1=[2;3];P2=[4;2];P3=[4;0];
>> t=0:0.01:1;
>> Q=P0*((1-t).^3)+3*P1*(t.*(1-t).^2)+...
3*P2*(t.^2.*(1-t))+P3*(t.^3);
+>> plot(Q(1,:),Q(2,:))
```

Pokud chceme zobrazit i kontrolní body, nejjednodušší je poskládat je do matice. kterou zobrazíme po řádcích:

```
>> hold on
>> PP=[P0,P1,P2,P3];
>> plot(PP(1,:),PP(2,:),'r*')
```



Pro konstrukci Bézierových křivek se dá použít také algoritmus de Casteljau pojmenovaný po jednou z objevitelů Bézierových křivek. Spočívá v rozdělení spojnic mezi kontrolními body v daném poměru. Tím získáme tři body, jejichž spojnice opět rozdělíme ve stejném poměru, Stejně tak rozdělíme i spojnici takto získaných dvou bodů a tím získáme bod, který leží na Bézierově křivce. Následující obrázek ukazuje tento proces pro dělící poměr 1:1 a 1:2, to znamená, že všechny spojnice dělíme na poloviny, respektive na třetinu a dvě třetiny.



To, že uvedený postup skutečně dává body na Bézierově křivce, snadno dokážeme následujícím výpočtem. Dělící poměr pro rozdělení spojnice dvou bodů je určen hodnotou $\alpha \in (0, 1)$, např. bod C na spojnici bodů A a B se dá vyjádřit jako $C = \alpha \cdot A + (1 - \alpha) \cdot B$, přičemž výsledný bod je blízko bodu A pro α blízké jedné. Délky údeček AC a CB jsou pak v poměru $(1 - t) : t$.

Položme tedy

$$\begin{aligned} R_0 &= \alpha \cdot P_0 + (1 - \alpha) \cdot P_1, \\ R_1 &= \alpha \cdot P_1 + (1 - \alpha) \cdot P_2, \\ R_2 &= \alpha \cdot P_2 + (1 - \alpha) \cdot P_3 \end{aligned}$$

čímž získáme první trojici bodů. Pokračujme dále:

$$\begin{aligned} S_0 &= \alpha \cdot R_0 + (1 - \alpha) \cdot R_1, \\ S_1 &= \alpha \cdot R_1 + (1 - \alpha) \cdot R_2 \end{aligned}$$

a konečně

$$T = \alpha \cdot S_0 + (1-\alpha) \cdot S_1.$$

Zpětným dosazováním dostáváme

$$\begin{aligned}
T &= \alpha \cdot S_0 + (1-\alpha) \cdot S_1 \\
&= \alpha \cdot (\alpha \cdot R_0 + (1-\alpha) \cdot R_1) + (1-\alpha) \cdot (\alpha \cdot R_1 + (1-\alpha) \cdot R_2) \\
&= \alpha^2 \cdot R_0 + 2\alpha(1-\alpha) \cdot R_1 + (1-\alpha)^2 \cdot R_2 \\
&= \alpha^2 \cdot (\alpha \cdot P_0 + (1-\alpha) \cdot P_1) + 2\alpha(1-\alpha) \cdot (\alpha \cdot P_1 + (1-\alpha) \cdot P_2) + \\
&\quad +(1-\alpha)^2 \cdot (\alpha \cdot P_2 + (1-\alpha) \cdot P_3) \\
&= \alpha^3 \cdot P_0 + 3\alpha^2(1-\alpha) \cdot P_1 + 3\alpha(1-\alpha)^2 \cdot P_2 + (1-\alpha)^3 \cdot P_3.
\end{aligned}$$

Vidíme, že pro $t = 1 - \alpha$ jsem dostali přesně vyjádření bodu na Bézierově křivce.

Kapitola 2

Derivace

2.1 Limity

Ještě před samotnými derivacemi se aspoň krátce zmíníme o limitách. V Sage to není problém, stačí zadat například

```
sage: n=var('n')
sage: limit((1+1/n)^n,n=oo)
e
sage: limit((1-1/n)^n,n=oo)
e^(-1)
sage: limit((1+10/n)^n,n=oo)
e^10
sage: x=var('x')
sage: limit((1+x/n)^n,n=oo)
e^x
sage: limit(sin(x)/x,x=0)
1
```

V Matlabu symbolické limity nenajdeme, nicméně některé limity pro $n \to \infty$ se dají spočítat nebo odhadnout tak, že daný výraz se pokoušíme spočítat pro hodně velká čísla. Má to ale svoje úskalí. Pokud bychom se pomocí výše uvedené limity pokoušeli určit Eulerovo číslo s přesností na 6 desetinných míst, dostáváme:

```
>> format long
>> n=1;
```

```
>> e0=(1+1/n)^n
e0 =
     2
>> n=10;
>> e1=(1+1/n)^n
e1 =
   2.593742460100002
>> while abs(e1-e0)>0.000001, e0=e1; n=10*n;...
e1=(1+1/n)^n, end
e1 =
   2.704813829421528
e1 =
   2.716923932235594
e1 =
   2.718145926824926
e1 =
   2.718268237192297
e1 =
   2.718280469095753
e1 =
   2.718281694132082
e1 =
   2.718281798347358
```

Pokud bychom ale chtěli tímto způsobem určit Eulerovo číslo s přesností na 15 desetinných čísel, což je zhruba přesnost, s jakou jsou běžně v počítači uložena čísla řádově v jednotkách, vyjde

```
>> while abs(e1-e0)>0.000000000000001, e0=e1; n=10*n;...
e1=(1+1/n)^n, end
e1 =
   2.704813829421528
e1 =
   2.716923932235594
e1 =
   2.718145926824926
e1 =
   2.718268237192297
```

```
e1 =
   2.718280469095753
e1 =
   2.718281694132082
e1 =
   2.718281798347358
e1 =
   2.718282052011560
e1 =
   2.718282053234788
e1 =
   2.718282053357110
e1 =
   2.718523496037238
e1 =
   2.716110034086901
e1 =
   2.716110034087023
e1 =
   3.035035206549262
e1 =
     1
e1 =
     1
```

Pro velká n je totiž hodnota $1/n$ menší než přesnost, s jakou je uložena jednička, takže při sčítání $1 + 1/n$ dostáváme výsledek 1. Pokud bychom chtěli přesto s pomocí Matlabu Eulerovo číslo určit jako limitu nějakého nekonečného procesu, můžeme využít vztah

$$e = \sum_{n=0}^{\infty} \frac{1}{1!}.$$

Součet nekonečné řady je limita částečných součtů, které můžeme snadno vyjádřit:

```
>> n=0;a=1;s=1;
>> while a>0.000000000000001, n=n+1;a=a/n;s=s+a; end
>> s
```

```
s =
   2.718281828459046
>> exp(1)
ans =
   2.718281828459046
>> n
n =
    18
```

Ve výpisu vidíme, že dosažená hodnota je v rámci přesnosti stejná jako hodnota, se kterou Matlab pracuje. Na určení stačilo sečíst 18 členů řady, protože 1/18! je řádově 10^{-16}, tedy další členy by se ve výsledku již neprojevily.

2.2 Symbolické derivování

Symbolické derivování je poměrně snadné v Matlabu i v Sage. V Matlabu si nejdříve definujeme symbolický objekt a pak s ním můžeme provádět symbolické operace, tedy i derivování:

```
>> f1=sym('sin(x)*cos(x)')
f1 =
cos(x)*sin(x)
>> diff(f1)
ans =
cos(x)^2 - sin(x)^2
>> f2=sym('log(t^2)');
>> diff(f2)
ans =
2/t
```

Vidíme, že pokud zadaná funkce obsahuje jen jednu proměnnou, derivuje se automaticky podle ní a výsledek je současně upraven. Pokud máme funkci více proměnných, je potřeba si zvolit, podle které chceme derivovat, pokud to není zrovna x.

```
>> f3=sym('cos(x^2)*sin(y)');
>> diff(f3)
ans =
```

```
-2*x*sin(x^2)*sin(y)
>> diff(f3,'y')
ans =
cos(x^2)*cos(y)
```

Je taky možné počítat vyšší derivace:

```
>> diff(f2,3)
ans =
4/t^3
>> diff(f3,'y',2)
ans =
-cos(x^2)*sin(y)
```

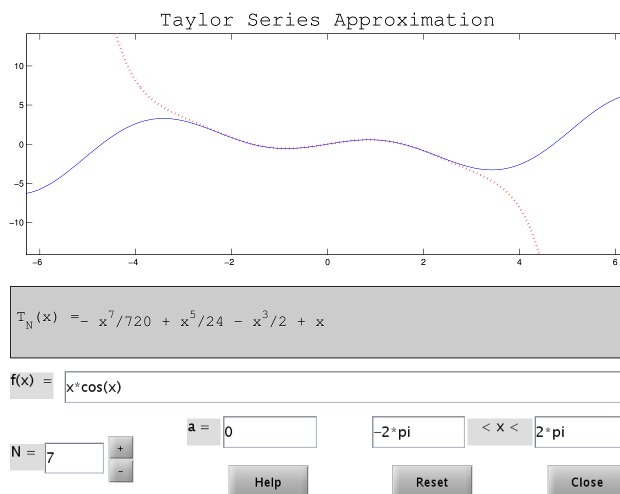
Derivování v Sage je podobně jednoduché, jen nejdříve musíme definovat proměnné. Máme dokonce na výběr, který příkaz pro derivování použijeme a samozřejmostí je počítání vyšších derivací.

```
sage: x=var('x')
sage: t=var('t')
sage: diff(exp(x)*cos(x),x)
-e^x*sin(x) + e^x*cos(x)
sage: derivative(exp(x)*cos(x),x)
-e^x*sin(x) + e^x*cos(x)
sage: diff(log(t)*exp(x),t)
e^x/t
sage: diff(log(t)*exp(x),x)
e^x*log(t)
sage: diff(log(t)*exp(x),t,3)
2*e^x/t^3
```

2.3 Taylorův rozvoj

Určit Taylorův rozvoj funkce, když dokážeme počítat derivace, je už snadné. Spočítali bychom derivace funkce v daném bodě, určili tak příslušné koeficienty, věřím, že by s s tímto problémem čtenář bez problémů poradil. V

Matlabu ale existuje hezký prográmek `taylortool`, ve kterém si můžete zadat, po jakou funkci chcete Taylorův rozvoj spočítat, ve kterém bodě a do jakého stupně a jako výsledek uvidíte něco jako na dalším obrázku.



Jak by se dalo čekat. určit Taylorův rozvoj funkce v Sage je jednoduché. Stačí definovat proměnnou, funkci a pak použít funkci `taylor`:

```
sage: x=var('x')
sage: f1=sqrt(x+1)
sage: f1.taylor(x,0,6)
-21/1024*x^6 + 7/256*x^5 - 5/128*x^4 + 1/16*x^3 -
1/8*x^2 + 1/2*x + 1
sage: f2=sqrt(x)
sage: f2.taylor(x,1,6)
-21/1024*(x - 1)^6 + 7/256*(x - 1)^5 - 5/128*(x - 1)^4 +
1/16*(x - 1)^3 - 1/8*(x - 1)^2 + 1/2*x + 1/2
```

Jde to i bez definování funkce:

```
sage: taylor(sin(x)*cos(x),x,pi,6)
-pi - 2/15*(pi - x)^5 + 2/3*(pi - x)^3 + x
```

Možná by stálo za to trochu prozkoumat, podle čeho se určuje pořadí členů na výstupu.

2.4 Numerické derivování

Může se stát, že známe hodnoty funkce jen v diskrétních uzlech, nicméně potřebujeme spočítat alespoň přibližně hodnotu její derivace v nějakém bodě. Zpravidla se jedná o některý z uzlů, ale nemusí to tak být vždy.

V takové situaci máme dvě možnosti: sestrojíme interpolační polynom a ten zderivujeme, nebo si vypomůžeme Taylorovým rozvojem. Oba přístupy dávají stejné výsledky, u Taylorova rozvoje dostaneme navíc odhad chyby, výsledky dosažené derivací interpolačního polynomu zase dostaneme bez velkého přemýšlení.

Začneme tedy s interpolačním polynomem. Zde se hodí Lagrangeův tvar

$$P_n(x) = \sum_{i=0}^{n} f_i l_i(x),$$

kde l_i jsou Lagrangeovy fundamentální polynomy. Derivováním této formule dostaneme

$$f'(x) \approx P_n'(x) = \sum_{i=0}^{n} f_i l_i'(x).$$

Podrobnosti o přesnosti a dalších vlastnostech této formule se čtenář může dočíst v [1], my se zde blíže podíváme jen na některé speciální případy. Odvodíme formuli pro přibližný výpočet derivace v prostředním ze tří ekvidistantních uzlů. Kvůli symetrii formulí si je označíme trochu jinak než obvykle: x_{-1}, x_0, x_1, $x_i = x_0 + ih$, $i = \pm 1$.

Nejprve sestrojíme Lagrangeův interpolační polynom:

x_i	x_{-1}	x_0	x_1
f_i	f_{-1}	f_0	f_1

$$\begin{aligned} P_2(x) &= f_{-1}\frac{(x-x_0)(x-x_1)}{(x_{-1}-x_0)(x_{-1}-x_1)}f_{-1} + f_0\frac{(x-x_{-1})(x-x_1)}{(x_0-x_{-1})(x_0-x_1)} + \\ &+ f_1\frac{(x-x_{-1})(x-x_0)}{(x_1-x_{-1})(x_1-x_0)}, \end{aligned}$$

Výraz zderivujeme

$$P_2'(x) = f_{-1}\frac{2x-x_0-x_1}{2h^2} - f_0\frac{2x-x_{-1}-x_1}{h^2} + f_1\frac{2x-x_{-1}-x_0}{2h^2},$$

pro $x_i = x_0 + ih$, $i = \pm 1$. Pokud dosadíme $x = x_0$ dostaneme

$$f'(x_0) \approx P_2'(x_0) = \frac{1}{2h}(f_1 - f_{-1}). \tag{2.1}$$

Všimněme si ještě geometrického významu této formule. Výraz $(f_1 - f_{-1})/2h$ je směrnice sečny, která je určena body (x_{-1}, f_{-1}) a (x_1, f_1). Podobně můžeme odvodit i formule pro výpočet derivace v dalších uzlových bodech x_{-1}, x_1:

$$f'(x_{-1}) \approx \frac{1}{2h}(-3f_{-1} + 4f_0 - f_1) \tag{2.2}$$

$$f'(x_1) \approx \frac{1}{2h}(f_{-1} - 4f_0 + 3f_1) \tag{2.3}$$

Tyto formule se nazývají *tříbodové*.

Podívejme se ještě, jak se dají stejné formule odvodit z Taylorova rozvoje. Pro uzly $x_{\pm 1} = x_0 \pm h$ dostáváme

$$\begin{aligned} f(x_1) &= f(x_0) + f'(x_0)h + \frac{f''(x_0)}{2}h^2 + \frac{f'''(x_0)}{6}h^3 + \frac{f^{(4)}(x_0)}{24}h^4 + \cdots \\ f(x_{-1}) &= f(x_0) - f'(x_0)h + \frac{f''(x_0)}{2}h^2 - \frac{f'''(x_0)}{6}h^3 + \frac{f^{(4)}(x_0)}{24}h^4 - \cdots \end{aligned}$$

Odečtením obou rovností a vydělením $2h$ dostáváme

$$f'(x_0) = \frac{1}{2h}(f_1 - f_{-1}) - \frac{f'''(x_0)}{3}h^2 - \cdots, \tag{2.4}$$

takže máme stejnou formuli jako v předchozím případě, navíc vidíme, že chyba je řádově h^2, takže pokud například hodnotu h zmenšíme na polovinu, chyba klesne přibližně na čtvrtinu.

Pokud bychom pro přibližné vyjádření derivace v bodě x_0 použili jen jeden Taylorův rozvoj, dostaneme

$$f'(x_0) = \frac{1}{h}(f_1 - f_0) - \frac{f''(x_0)}{2}h - \cdots,$$

tedy výraz s chybou o řád horší než vztah předchozí.

Uvedené Taylorovy rozvoje ale můžeme i sečíst, v tom případě dostaneme vztah pro přibližný výpočet druhé derivace:

$$f''(x_0) = \frac{1}{h^2}(f_{-1} - 2f_0 + f_1) - \frac{f^{(4)}(x_0)}{12}h^2 - \cdots. \tag{2.5}$$

Vidíme, že chyba uvedeného vztahu je řádově opět rovna h^2.

Kapitola 3

Integrály

3.1 Symbolické integrování

Co se týče počítání symbolického počítání neurčitých i určitých integrálů, zvládají to oba námi používané softwarové prostředky bez velkých problémů. Třeba v Matlabu pro výpočet neurčitého nebo určitého integrálu používáme funkci `int`, rozdíl je jenom v zadaných parametrech:

```
>> f=sym('sin(x)*cos(2*x)');
>> int(f)
ans =
cos(x) - (2*cos(x)^3)/3
>> int(f,0,pi)
ans =
-2/3
```

Je možné i integrovat podle jedné z více proměnných:

```
>> g=sym('exp(-t)*sin(2*x*t)');
>> int(g,'t')
ans =
-(sin(2*t*x) + 2*x*cos(2*t*x))/(exp(t)*(4*x^2 + 1))
>> int(g,'x',0,pi/2)
ans =
-(cos(pi*t) - 1)/(2*t*exp(t))
```

Dají se dokonce vyjádřit integrály z funkcí, jejichž primitivní funkce sice existují, ale nedají se vyjádřit v rozumném konečném tvaru:

```
>> G=sym('exp(-x^2)');
>> int(G)
ans =
(pi^(1/2)*erf(x))/2
>> I=int(G,0,1)
I =
(pi^(1/2)*erf(1))/2
>> eval(I)
ans =
    0.7468
>> int(G,0,inf)
ans =
pi^(1/2)/2
```

Výrazem `erf` v Matlabu získáme primitivní funkci k e^{-x^2}, jak zjistíme z nápovědy, a Matlab umí spočítat hodnoty této funkce:

```
>> help erf
erf Error function.
  Y = erf(X) is the error function for each element of X.
  X must be real. The error function is defined as:

   erf(x) = 2/sqrt(pi) * integral from 0 to x of
   exp(-t^2) dt.

  See also erfc, erfcx, erfinv, erfcinv.

  Overloaded methods:
     sym/erf

  Reference page in Help browser
     doc erf
>> erf(1)
ans =
    0.8427
```

V Sage je to velmi podobné:

```
sage: x=var('x')
sage: integral(sin(x)*x,x)
-x*cos(x) + sin(x)
sage: integral(sin(x)*x,x,0,pi)
pi
sage: integral(sin(x)*x,x,0,pi/2)
1
sage: t=var('t')
sage: integral(sin(x),t)
t*sin(x)
sage: integral(1/sqrt(x),x,0,1)
2
sage: integral(1/x^2,x,0,1)

Traceback (click to the left of this block for traceback)
...
ValueError: Integral is divergent.
```

I primitivní funkce k e^{-x^2} se vyjadřuje podobně:

```
sage: integral(exp(-x^2),x)
1/2*sqrt(pi)*erf(x)
sage: integral(exp(-x^2),x,0,oo)
1/2*sqrt(pi)
sage: I0=integral(exp(-x^2),x,0,1);I0
1/2*sqrt(pi)*erf(1)
sage: I0.N()
0.746824132812427
```

3.2 Numerické integrování

Při numerickém integrování se podobně jako při derivování nesnažíme určit primitivní funkci z funkčních hodnot v zadaných diskrétních uzlech, ale snažíme se na základě těchto dat určit přibližně určitý integrál přes nějaký interval.

Základem získaných formulí jsou opět interpolační polynomy. Při integrování Lagrangeova tvaru dostáváme:

$$\int_a^b f(x)dx \approx \int_a^b P_n(x)dx = \sum_{i=0}^n f_i \int_a^b l_i(x)dx = \sum_{i=0}^n f_i A_i$$

pro $A_i = \int_a^b l_i(x)dx$. Podobný typ výrazů dostáváme také při odvozování Riemannova integrálu, kde aproximaci určitého integrálu dostáváme jako součet funkčních hodnot násobených koeficienty zahrnujícími délky subintervalů při dělení původního intervalu na menší dílky. Těmto výrazům, tedy

$$\int_a^b f(x)dx \approx \sum_{i=0}^n f_i A_i$$

říkáme kvadraturní formule. Kromě odhadu chyby nás u kvadraturních formulí zajímá také její stupeň přesnosti, který udává, pro polynomy jakých stupňů je tato formule přesná. Přesněji, pokud je stupeň přesnosti formule roven n pak je tato formule je přesná pro polynomy do stupně n včetně. Dá se snadno ukázat (viz [1]), že pro $n+1$ uzlů je stupeň přesnosti roven maximálně $2n+1$. Na druhou stranu je zřejmé, že pokud kvadraturní formuli získáme postupem ukázaným výše, tedy integrací interpolačního polynomu, musí být stupeň přesnosti roven alespoň n.

Konstrukce kvadraturních formulí je možné ještě poněkud zobecnit tím, že do integrálu přidáme ještě tzv. vahovou funkci. Do ní můžeme zahrnout například singularity nebo společnou část pro nějakou třídu funkcí. V tom případě kvadraturní formule nahrazují integrál

$$\sum_{i=0}^n f_i A_i \approx \int_a^b w(x(\cdot f(x)dx.$$

Koeficienty kvadraturní formule pak získáme ze vztahu

$$A_i = \int_a^b w(x) \cdot l_i(x)dx.$$

Výhodou takového postupu je, že vahová funkce není zahrnuta ve funkčních hodnotách funkce f. V tomto textu se ale budeme zabývat pouze jednoduchou situací s vahovou funkcí rovnou jedné.

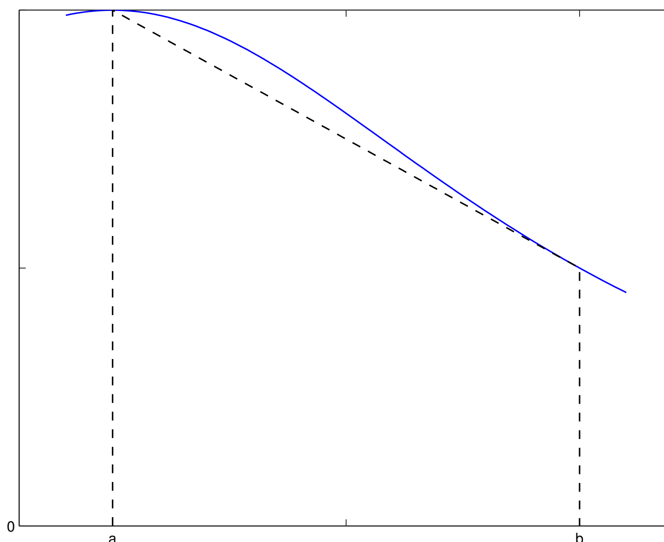
Odvodíme si některé nejpoužívanější formule. Situací s jedním uzlem se zabývat nebudeme, tak je až příliš jednoduchá. Pro dva uzly většinou máme $a = x_0 < x_1 = b$. V tomto případě má interpolační polynom tvar například

$$P_1(x) = f(a) + \frac{f(b) - f(a)}{b - a}(x - a).$$

Jeho integrováním přes interval $[a, b]$ vyjde kvadraturní formule

$$\int_a^b f(x)dx \approx \frac{f(a) + f(b)}{2}(b - a),$$

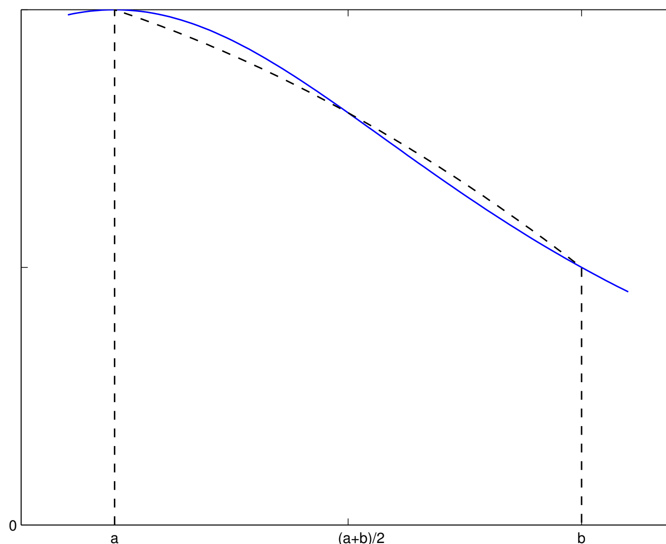
což je v podstatě plocha lichoběžníka (postaveného na bok),jehož strany mají délky rovny funkčním hodnotám funkce f v krajních bodech intervalu a jehož výška je rovna $b - a$. Proto se taky této formuli říká *lichoběžníkové pravidlo*.



V případě, že pudeme funkci aproximovat polynomem druhého stupně, tedy parabolou, dostaneme pro uzly $x_0 = a$, $x_1 = \frac{a+b}{2}$, $x_2 = b$ formuli

$$\int_a^b f(x)dx \approx \frac{b - a}{6}\left(f(a) + 4f(\frac{a + b}{2}) + f(b)\right),$$

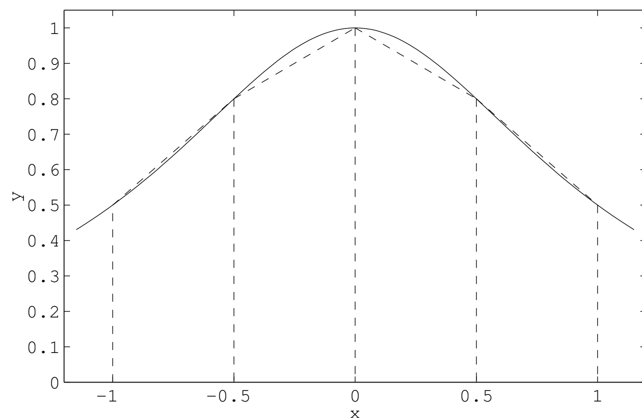
která se nazývá Simpsonovo pravidlo.



Z těchto dvou formulí se odvozují patrně nejpoužívanější kvadraturní formule. Jejich myšlenka je jednoduchá: pokud máme více uzlů, pak nám rozdělují interval $[a, b]$ na více subintervalů, na každém tomto subintervalu použijeme lichoběžníkové pravidlo nebo Simpsonovo pravidlo a výsledek pak sečteme. Tím dostaneme složené lichoběžníkové nebo složené Simponovo pravidlo. U složeného lichoběžníkového pravidla v podstatě počítáme integrál z aproximace funkce lomenou čárou, respektive lineárním splajnem. U Složeného Simpsonova pravidla zase vyžadujeme, aby celkový počet uzlů byl lichý, na což by jistě čtenář přišel sám. pro ekvidistantní uzly $x_0, \ldots, x_n$, kde vzdálenost sousedních dvou je rovna h dostáváme složené lichoběžníkové pravidlo ve tvaru

$$\int_a^b f(x)dx \approx \frac{h}{2}\left(f_0 + 2f_1 + 2f_2 + \cdots + 2f_{n-1} + f_n\right),$$

kde $f_i = f(x_i)$.



Složené Simponovo pravidlo pak je

$$\int_a^b f(x)dx \approx \frac{h}{3}\left(f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \cdots + 2f_{n-2} + 4f_{n-1} + f_n\right).$$

Kapitola 4

Řady

Řad už jsme částečné dotkli, když jsme se zmínili o Taylorově rozvoji v souvislosti s derivacemi. Teď si zkusíme ukázat, jak se dají počítat některé součty řad a nesmíme zapomenout taktéž na řady Fourierovy.

4.1 Symbolické součty řad

Jak se dá čekat, v Matlabu je se symbolickým sčítáním řad potíž. Můžeme zde maximálně odhadnout číselný součet řady, jestliže připočítáváme k součtu členy, které jsou už z numerického hlediska zanedbatelné, jak jsme to ukázali při výpočtu Eulerova čísla. Proto se v dalším výkladu soustředíme na Sage.

Nejprve si zkusíme sečíst jednoduchou geometrickou řadu:

```
var('x','k','n')
sum(x^k,k,0,oo)
Traceback (click to the left of this block for traceback)
...
Is  abs(x)-1  positive, negative, or zero?
```

Získali jsme pouze chybové hlášení. Chybí totiž bližší informace o proměnné x a víme, že uvedená řada má součet jen v případě, že $|x| < 1$. Proto tento požadavek zadáme do Sage:

```
sage: assume(abs(x)<1)
sage: sum(x^k,k,0,oo)
-1/(x - 1)
```

Vidíme, že tohle funguje a můžeme zkoušet další součty:

```
sage: sum(k*x^k,k,1,oo)
x/(x^2 - 2*x + 1)
sage: sum(1/k^4, k, 1, oo)
1/90*pi^4
sage: sum(x^k/factorial(k), k, 0, oo)
e^x
```

Zatím to vypadá všechno bezproblémově. V některých situacích si ovšem ani Sage neporadí. Ukážeme si to na poměrně jednoduchém příkladu. Nejprve si spočítáme Taylorův rozvoj funkce $\sqrt{1-x}$.

```
sage: f=sqrt(1-x)
sage: f.taylor(x,0,6)
-21/1024*x^6 - 7/256*x^5 - 5/128*x^4 - 1/16*x^3
- 1/8*x^2 - 1/2*x + 1
```

Čísla, která se zde objevují, jsou binomické koeficienty, které jsou pro reálný argument a definované vztahem

$$\binom{a}{k} = \frac{a(a-1)\cdots(a-k+1)}{k!}$$

V našem případě jsou to koeficienty pro $a = 1/2$ (až na znaménko), jak snadno ověříme:

```
[binomial(1/2,k) for k in range(7)]
[1, 1/2, -1/8, 1/16, -5/128, 7/256, -21/1024]
```

Zkusíme tedy řadu zpětně sečíst. Problémem bude trochu první člen, ale ten by se měl projevit jen posunem výsledku o konstantu:

```
sum(x^k*(-1)^k*binomial(1/2,k),k,0,oo)
Traceback (click to the left of this block for traceback)
...
TypeError: Either m or x-m must be an integer
```

Vidíme, že na tomto Sage ztroskotal.

4.2 Fourierovy řady

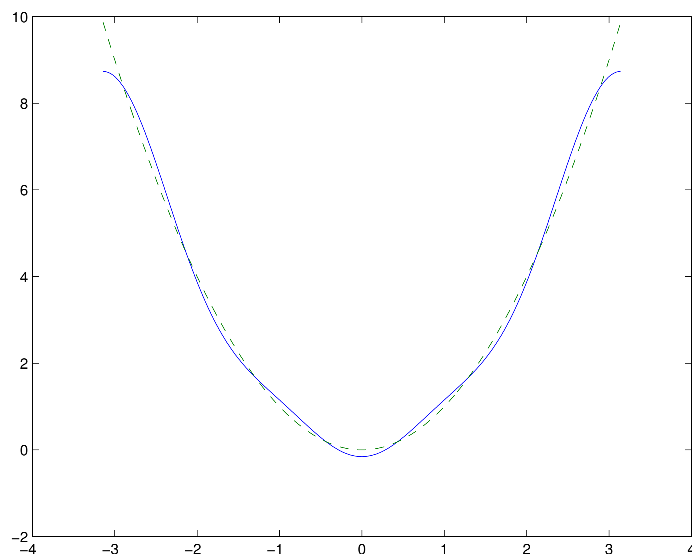
Spočítat koeficienty Fourierovy řady na základě toho, co už známe, jistě nebude pro čtenáře problém. Například pokud bychom chtěli v Matlabu určit prvních pár členů Fourierovy řady pro funkci x^2, můžeme postupovat třeba takto:

```
>> f=sym('x^2');
>> a0=int(f,-pi,pi)/(2*pi)
a0 =
pi^2/3
>> a1=int('cos(x)'*f,-pi,pi)/pi
a1 =
-4
>> a2=int('cos(2*x)'*f,-pi,pi)/pi
a2 =
1
>> a3=int('cos(3*x)'*f,-pi,pi)/pi
a3 =
-4/9
```

Koeficienty u sinových členů jsou nulové, neboť x^2 je sudá funkce. Sečteme tedy první čtyři členy řady a výsledek porovnáme s původní funkcí:

```
>> x=-pi:0.01:pi;
>> S=eval(a0+a1*cos(x)+a2*cos(2*x)+a3*cos(3*x));
>> plot(x,S,x,x.^2,'--')
```

Pro výpočet hodnot součtů v bodech daných vektorem `x` musíme použít funkci `eval`, protože koeficienty `a0`,. . . ,`a3` jsou symbolické objekty.



Pokud chceme určit Fourierovu řadu v Sage, je potřeba si funkci definovat jen na příslušném intervalu, tedy např. na $[-\pi, \pi]$:

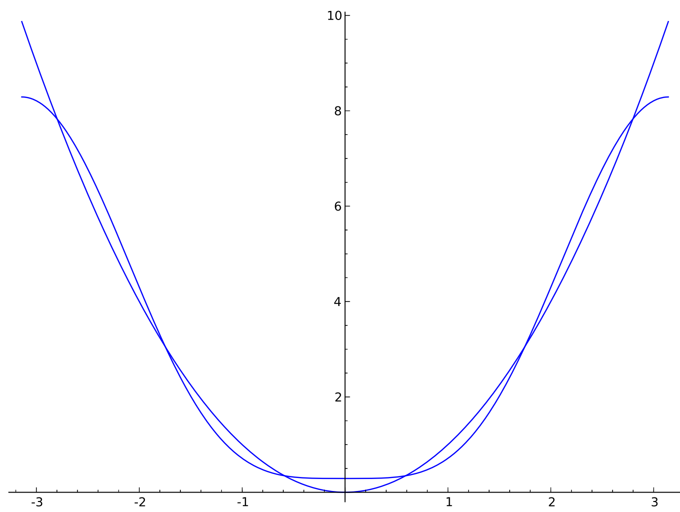
```
sage: x=var('x')
sage: f=Piecewise([[(-pi,pi),x^2]])
```

Stejné koeficienty jako v Matlabu určíme pomocí příkazů

```
sage: f.fourier_series_cosine_coefficient(0,pi)
2/3*pi^2
sage: f.fourier_series_cosine_coefficient(1,pi)
-4
sage: f.fourier_series_cosine_coefficient(2,pi)
1
sage: f.fourier_series_cosine_coefficient(3,pi)
-4/9
```

První vstupní parametr říká, který koeficient určujeme, druhý je polovina intervalu, na němž řadu počítáme. Částečný součet řady ale můžeme získat i přímo a porovnat ho s původní funkcí:

```
sage: g=f.fourier_series_partial_sum(3,pi)
sage: pl=plot([g,x^2],-pi,pi)
```

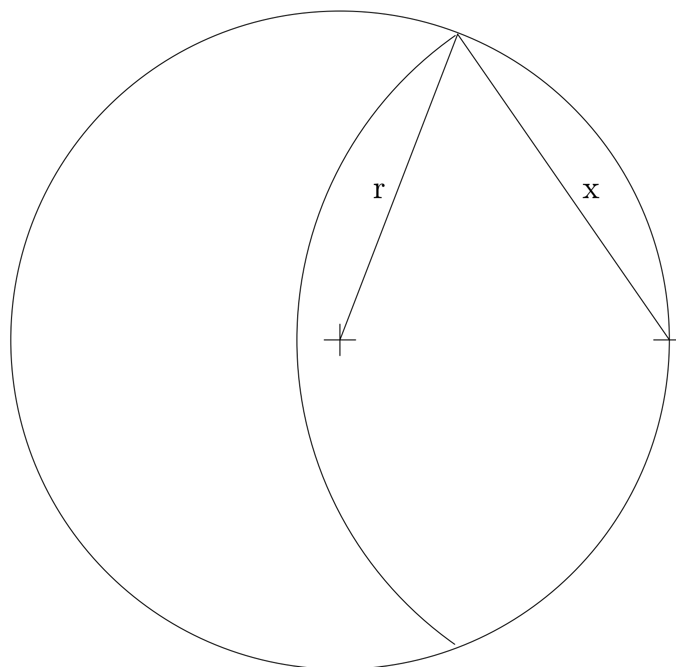

10
8
6
4
2
-3
-2
-1
1
2
3

Kapitola 5

Řešení nelineárních rovnic

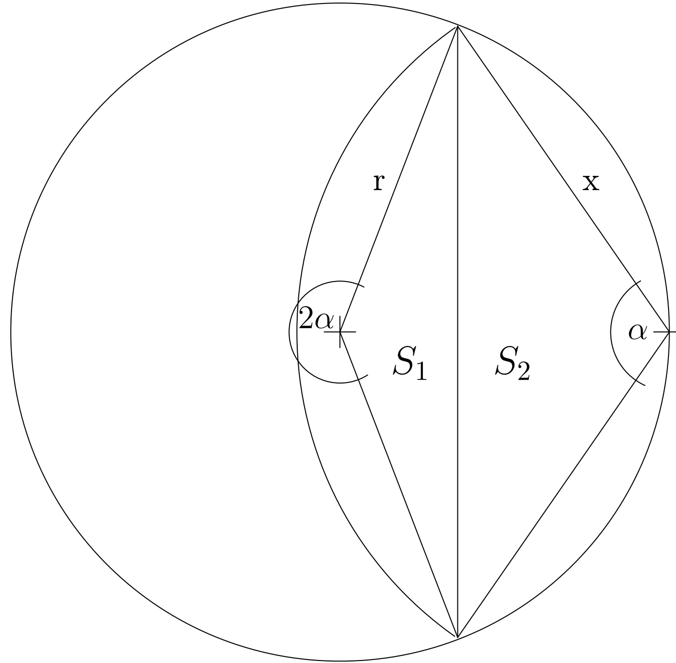
5.1 Motivační úloha

Začneme geometrickou úlohou s jednoduchým zadáním. Mějme kruh v rovině o zadaném poloměru r. Máme z hranice kruhu udělat kruhový oblouk o neznámém poloměru x tak, aby část ohraničená oběma křivkami měla plochu, která je rovna polovině plochy původního kruhu. Situaci si můžeme znázornit na obrázku:



Tato úloha je matematickým vyjádřením úlohy ze zemědělství. Sedlák má kruhový pozemek, na kterém se pase koza. Protože sedlák chce, aby jí tráva na pozemku vystačila na dva dny, uváže ji ke kůlu na okraji pozemku tak dlouhým provazem, aby za první den spásla polovinu trávy. Druhý den ji nechá k dispozici celý pozemek, kde může spást zbylou trávu.

Pro řešení úlohy si do obrázku doplníme několik údajů.



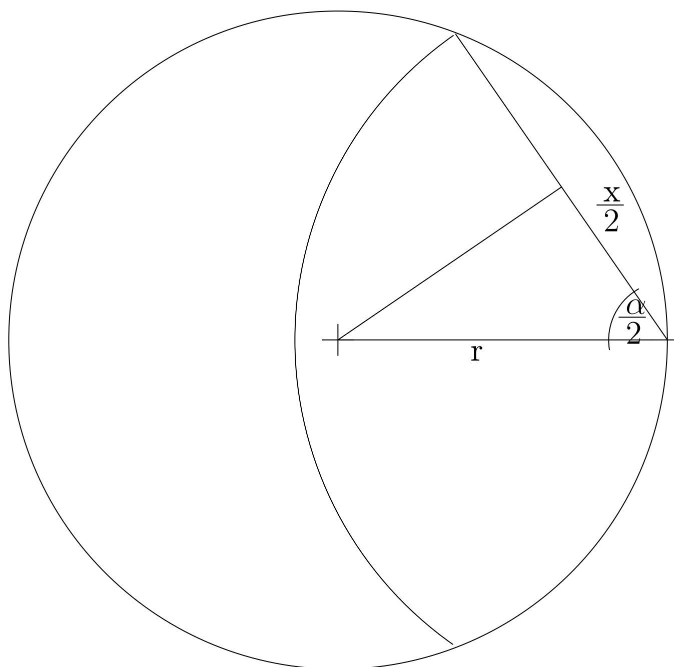
Spojíme střed kružnice i oblouku s průsečíky na kružnici. Průvodiče vycházející od kraje kružnice svírají úhel α, průvodiče vycházející ze středu kružnice tedy svírají úhel 2α. Oblast, která nás zajímá, se skládá ze dvou kruhových úsečí, první z nich má obsah S_1 a je dána poloměrem x a úhlem α, druhá má obsah S_2, poloměr r a úhel $2\pi - 2\alpha$. Označíme-li obsah celého kruhu S, dostáváme

$$S_1 + S_2 = \frac{1}{2}S.$$

Obsah kruhové úseče určíme tak, že od obsahu kruhové výseče odečte obsah trojúhelníka o stranách rovných poloměru výseče,jež svírají úhel, jež určuje výseč. Tedy

$$S_1 = \frac{1}{2}x^2(\alpha - \sin\alpha), \qquad S_2 = \frac{1}{2}r^2((2\pi - 2\alpha) - \sin(2\pi - 2\alpha))$$

Ještě potřebujeme vztah mezi r a x. K tomu můžeme použít pravoúhlý trojúhelník, který dostaneme, když spojíme střed kružnice se středem tětivy délky x podle následujícího obrázku:



V uvedeném obrázku platí

$$\frac{x/2}{r} = \cos\frac{\alpha}{2},$$

tedy

$$x = 2r\cos\frac{\alpha}{2}.$$

Můžeme tedy namísto x hledat úhel α, z předchozího vztahu pak x snadno vypočítáme.

Nyní už máme všechny potřebné vztahy pohromadě a použitím goniometrických vzorečků (toto cvičení jistě čtenář snadno zvládne) získáme rovnici

$$\alpha = \tan\alpha - \frac{\pi}{2\cos\alpha}.$$

Přesné řešení této rovnice ovšem získat neumíme. Nezbývá, než použít nějakou numerickou metodu, kterýmžto se budou věnovat následující řádky.

5.2 Základní metody řešení nelineární rovnice

Budeme se zabývat hledáním řešení rovnice

$$f(x) = 0, \qquad x \in I = [a, b] \tag{5.1}$$

pro spojitou funkci f. To že řešení na uvedeném intervalu existuje nám zaručí například podmínka $f(a) \cdot f(b) \leq 0$, tedy v krajních bodech intervalu I má funkce f opačná znaménka. Řešení ovšem může být více. Označme řešení rovnice jako ξ, nazýváme jej také kořenem funkce f.

Všechny metody a tvrzení, které zde budou vedeny, může čtenář nalézt v detailním znění ve skriptech [1].

5.2.1 Půlení intervalu

Nejjednodušší metodou pro nalezení řešení je metoda půlení intervalu, zvané též metoda bisekce. Její myšlenka je velmi jednoduchá: jestliže máme splněnu podmínku $f(a) \cdot f(b) \leq 0$, rozpůlíme interval $[a, b]$ a jako nový interval vezmeme ten, pro který platí, že v jeho krajních bodech má funkce f opět opačná znaménka, tedy tento poloviční interval opět obsahuje řešení rovnice. Pokračujeme s půlením tak dlouho, dokud nedostaneme interval dostatečné malý (stále obsahující řešení), tedy máme přibližné řešení s požadovanou přesností. Jako přibližné řešení $\bar{\xi}$ stanovíme střed konečného intervalu.

U metody půlení intervalu je možné dokonce předem určit, kolik iterací budeme potřebovat. Jestliže například požadujeme, aby chyba přibližného řešení byla menší než δ, dostáváme pro počet kroků k nerovnost

$$\frac{b-a}{2^{k+1}} \leq \delta$$

odkud logaritmování při základu 2 dostáváme

$$k \geq \log_2 \frac{b-a}{2\delta}.$$

Metoda půlení intervalu má jednu základní výhodu, že totiž za uvedených předpokladů vždy konverguje. Její nevýhodou je, že konverguje poměrně pomalu. A jelikož jsme v podstatě vyčerpali všechny důležité informace o ní, budeme se věnovat metodám dalším.

5.2.2 Prostá iterační metoda

U této metody se zaměříme na rovnici v jiném tvaru:

$$x = g(x),$$

hledáme tedy hodnotu ξ, kterou funkce g zobrazí samu na sebe. Proto se bod ξ nazývá pevný bod funkce g.

Pro funkci g samozřejmě požadujeme spojitost na intervalu $I = [a, b]$. Pro existenci pevného bodu na tomto intervalu pak postačuje, aby se interval I zobrazil sám do sebe, tedy $\forall x \in I : g(x) \in I$. Tato podmínka ovšem nezaručí jednoznačnost pevného bodu. Pro ni potřebujeme, aby funkce g byla kontrakcí na intervalu I, to jest musí existovat konstanta L, $0 \leq L < 1$, že pro každé $x, y \in I$ platí

$$|g(x) - g(y)| \leq L \cdot |x - y|.$$

V případě platnosti této podmínky nám jednoznačnost pevného bodu zaručí Banachova věta o pevném bodě, kterou mohl čtenář náhodou zaslechnout na přednáškách věnovaných metrickým prostorům. Konstanta L, která v podmínce vystupuje, se nazývá Lipschitzova konstanta, a funkce, které tuto podmínku splňují (i bez omezení $L < 1$), se nazývají lipschitzovsky spojité.

Pakliže je funkce g kontrakcí na intervalu I, platí navíc, že pokud vezmeme libovolnou počáteční aproximaci $x_0 \in I$ a sestrojíme posloupnost rekurentně předpisem

$$x_1 = g(x_0), \ldots, x_{k+1} = g(x_k), \ldots,$$

bude tato posloupnost konvergovat k pevnému bodu ξ funkce g.

Tato vlastnost je zásadní pro hledání přibližného řešení rovnice. Ověření Lipschitzovské spojitosti s konstantou $L < 1$ by ovšem bylo poněkud komplikované. Naštěstí pro funkce, které mají na intervalu I spojitou derivaci je tato vlastnost zaručena, pokud pro každé $x \in I$ platí $|g'(x)| \leq L < 1$, což vyplývá bezprostředně z Lagrangeovy věty o střední hodnotě.

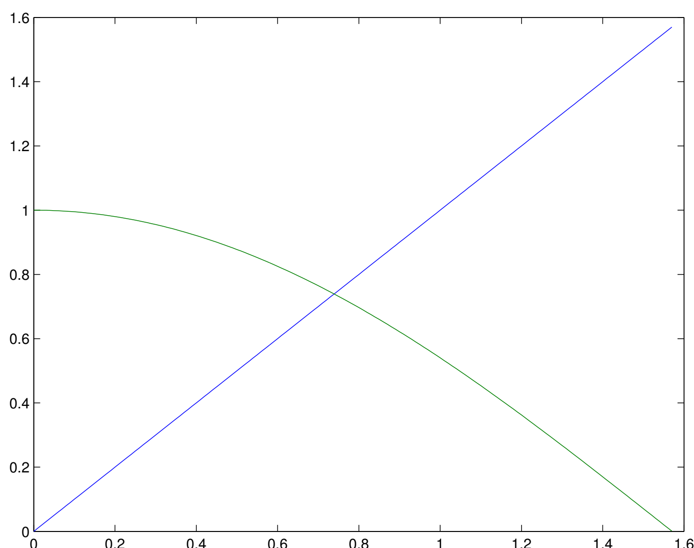
Protože výpočet posloupnosti je dán jednoduchým předpisem $x_{k+1} = g(x_k)$, nazývá se metoda založená na Banachově větě o pevném bodě *prostou iterační metodou*.

Ukažme si na jednoduchém příkladu postup ověření uvedených podmínek a nalezení přibližného řešení rovnice:

Příklad 2. Mejme rovnici

$$x = \cos x.$$

Pokud si problém představíme graficky, hledáme v podstatě průsečík grafu funkce $g(x) = \cos x$ s přímkou $y = x$ (viz obrázek):



Nejprve je potřeba najít interval, který funkce g zobrazí do sebe. Snadno zjistíme, že takovým intervalem je například interval $I = [0, \pi/2]$. Na tomto intervalu je navíc funkce $\cos x$ klesající, takže stačí ověřit, že se do intervalu I zobrazí jeho krajní body.

Dále musíme ověřit, zda na daném intervalu je maximu derivace funkce g v absolutní hodnotě ostře menší než jedna. Pokud tomu tak bude, stačí zvolit jako konstantu L ono maximum absolutní hodnoty derivace. Zde ovšem narazíme na problém – maximum je v pravém krajním bodě intervalu a je rovno jedné. Musíme tedy interval o něco zkrátit a to z obou stran, aby se i po zkrácení zobrazovala do sebe.

Vhodnou volbou (nikoliv jedinou možnou) je $I = [\cos 1.5, 1.5)$, pak $L = \sin 1.5 \doteq 0.9975$. Pokud pak budeme hledat přibližné řešení například v Matlabu, stačí zvolit libovolnou počáteční aproximaci a pak počítat jednoduchým způsobem další aproximace, dokud nedostaneme dostatečné přesné přibližné řešení. Takto jednoduché výpočty můžeme zadávat i ručně, napsat si jednoduchý program by čtenář jistě také zvládl.

```
>> presnost=0.0001;
>> x=1;
```

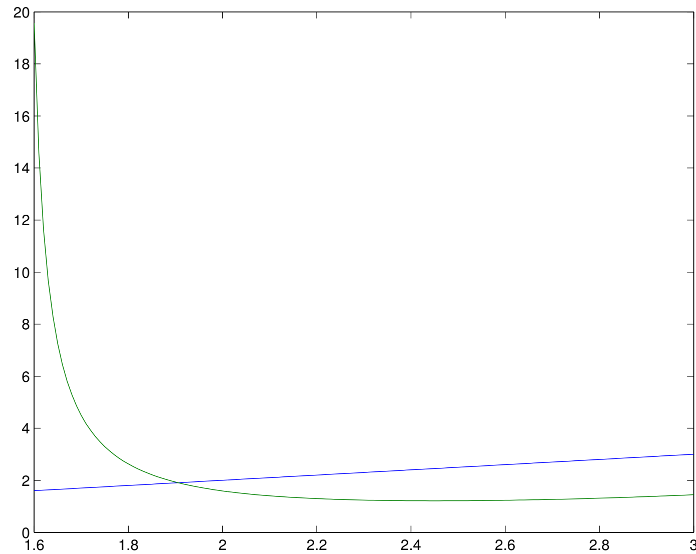
```
>> while abs(x-cos(x))>presnost, x=cos(x), end
x =
0.5403
x =
0.8576
x =
0.6543
x =
0.7935
x =
0.7014
.
.
.
x =
0.7392
x =
0.7390
x =
0.7391
>>
```

Na dosažení požadované přesnosti je potřeba něco přes dvacet iterací.

Pokud bychom chtěli použít prostou iterační metodu na řešení úvodního příkladu pro rovnici

$$x = \tan x - \frac{\pi}{2\cos x},$$

tedy pro funkci $g(x) = \tan x - \frac{\pi}{2\cos x}$, můžeme si vypomoci obrázkem, abychom mohli lépe odhadnout interval, kde se pevný bod nachází. Při tom využijeme toho, že musí ležet někde v intervalu $[\pi/2, \pi]$, který je třeba o něco zkrátit, jelikož v levém krajním bodě jde funkce g do nekonečna.



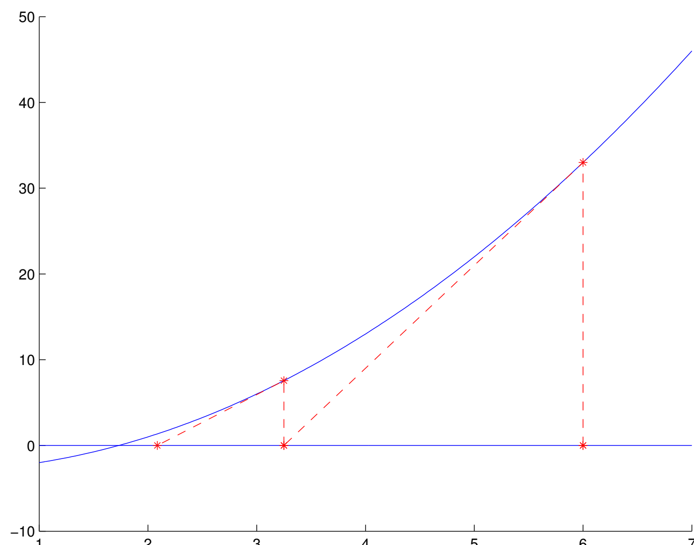
Zjistíme, že pevný bod leží v intervalu $[1.8, 2]$. Na tomto intervalu ale použít prostou iterační metodu není možné, neboť absolutní hodnota derivace funkce g je zde větší než jedna.

5.2.3 Newtonova metoda

Vrátíme se nyní zpět k rovnici

$$f(x) = 0.$$

Na řešení této rovnice můžeme nahlížet jakožto na průsečík grafu funkce f s osou x. Myšlenka Newtonovy metody je prostá. Zvolíme počáteční iteraci x_0 a další iterací je průsečík tečny ke grafu s osou x. V podstatě hledáme přibližné řešení na lineární aproximaci funkce f. Tento postup opakujeme tak dlouho, dokud nemáme přibližné řešení s dostatečnou přesností. Metodu si můžeme dobře ilustrovat graficky:



Je zřejmé, že v tomto případě kromě spojitosti funkce f potřebujeme také spojitost její derivace. Vyjádříme-li si z rovnice tečny k f v bodě $[x_k.f(x_k)]$ její průsečík z osou x, dostáváme iterační vztah

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

Stejný vztah bychom obdrželi použitím Taylorova rozvoje, v němž bychom zanedbali členy od druhé derivace včetně.

Máme vlastně opět prostou iterační metodu, tentokrát pro funkci g ve speciálním tvaru

$$g(x) = x - \frac{f(x)}{f'(x)},$$

můžeme tedy opět zkoumat podmínky, za nichž je tato funkce kontrakcí. Obecné vyjádření je ovšem složité, platí ale tvrzení, že pokud je derivace funkce f nenulová na intervalu obsahujícím řešení rovnice, pak existuje okolí tohoto řešení, na němž je funkce g kontrakcí, tedy pro libovolnou počáteční aproximaci z tohoto okolí Newtonova metoda konverguje.

Toto kriterium vypadá v praxi jen špatně použitelné, pokud se týče hledání počáteční aproximace. Naštěstí existují jiné podmínky, snadněji ověřitelné, umožňující její volbu, jsou to například Fourierovy podmínky:

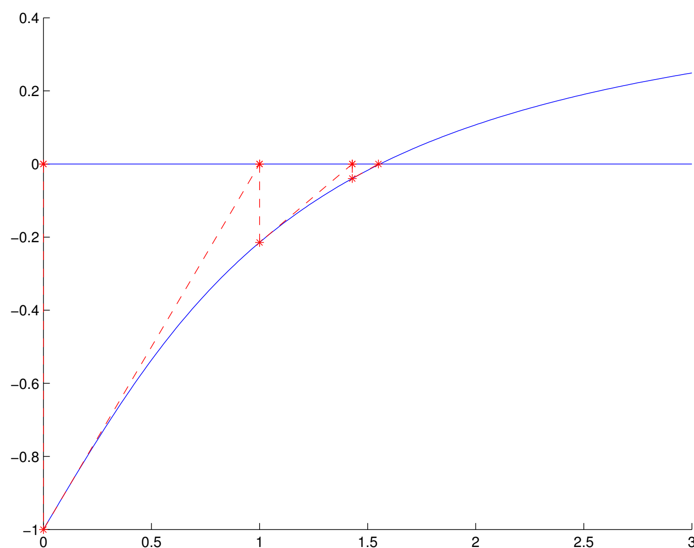
1. Funkce f má na intervalu I spojité druhé derivace a v krajních bodech I opačná znaménka.

2. Derivace funkce f je nenulová na celém intervalu, tedy ani nemění znaménko.

3. Druhá derivace funkce f nemění na I znaménko.

Při splnění těchto podmínek Newtonova metoda konverguje na I, a to dokonce monotonně, pakliže jako počáteční iteraci zvolíme ten krajní bod intervalu, v němž má funkční hodnota stejné znaménko jako je znaménko druhé derivace.

Uvedené podmínky vlastně říkají, že funkce f je na intervalu I ryze monotonní (rostoucí nebo klesající) a to buď konvexní nebo konkávní. Celkem jsou tedy čtyři případy, následující obrázek ukazuje situaci, kdy je f rostoucí a konkávní. V tom případě volíme jako počáteční aproximaci ten krajní bod, v němž je funkční hodnota záporná.



Další tři možnosti by si čtenář jistě dokázal snadno představit.

Příklad 3. Zkusme požít Newtonovu metodu na úvodní příklad, tedy na rovnici

$$x = \tan x - \frac{\pi}{2\cos x}.$$

Nejprve všechny členy převedeme na jednu stranu, abychom dostali funkce f:

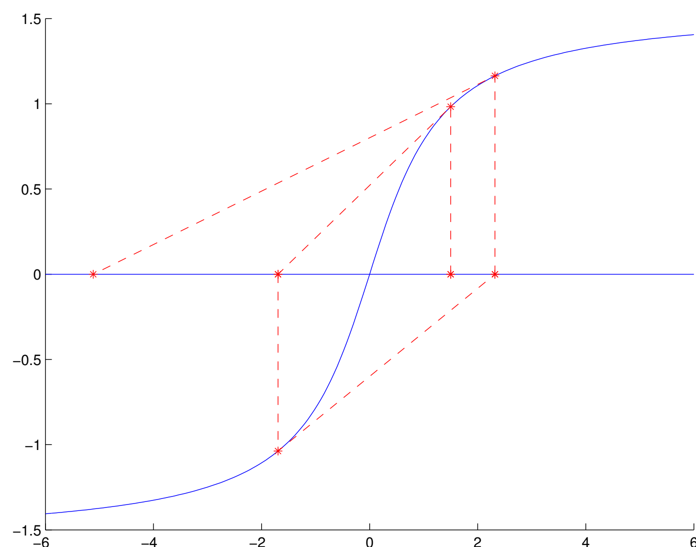
$$x - \tan x + \frac{\pi}{2\cos x} = 0$$

Pak zapojíme Matlab, přičemž použijeme toho, že z už známe interval, kde řešení leží.

```
>> f=sym('x-tan(x)+pi/(2*cos(x))')
f =
x - tan(x) + pi/(2*cos(x))
>> df=diff(f)
df =
(pi*sin(x))/(2*cos(x)^2) - tan(x)^2
>> x=1.8;
>> x=x-eval(f)/eval(df)
x =
1.8735
>> x=x-eval(f)/eval(df)
x =
1.9028
>> x=x-eval(f)/eval(df)
x =
1.9057
>> x=x-eval(f)/eval(df)
x =
1.9057
>>
```

Vidíme, že aproximaci řešení na čtyři desetinná místa jsem nalezli velmi rychle. Mohli bychom samozřejmě nastavit větší přesnost a počítat dál, ale i tak bychom dosáhli během několika dalších iterací nejlepší možné aproximace v rámci počítačové přesnosti.

Vypadá to tedy, že Newtonova metoda je velmi rychlá, i při jejím použití ovšem nesmíme zapomínat na jistou míru bedlivost. zejména co se týče volby počáteční aproximace. To si můžeme dobře demonstrovat na jednoduchém příkladě – totiž na funkci $f(x) = \arctan x$. Zvolíme-li jako počáteční aproximaci hodnotu 1.5, iterační posloupnost nekonverguje, jak naznačuje obrázek. Volbou $x_0 = 1$ by vše bylo v pořádku.



5.2.4 Metoda sečen a regula falsi

Další metody, o nichž si něco řekneme, jsou podobné. Jejich hlavní myšlenka spočívá v náhradě derivace v bodě x_k. To je nutné v případech, že derivaci nelze vyjádřit, například když hodnoty funkce f nejsou dány nějakým vzorcem, ale jsou to třeba výsledky fyzikálního měření. Derivaci pak nahradíme poměrnou diferencí, potřebujeme ovšem funkční hodnoty ve dvou bodech:

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$$

Použijeme-li tuto náhradu v iteračním vztahu pro Newtonovu metodu, dostáváme

$$x_{k+1} = x_k - f(x_k) \cdot \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}.$$

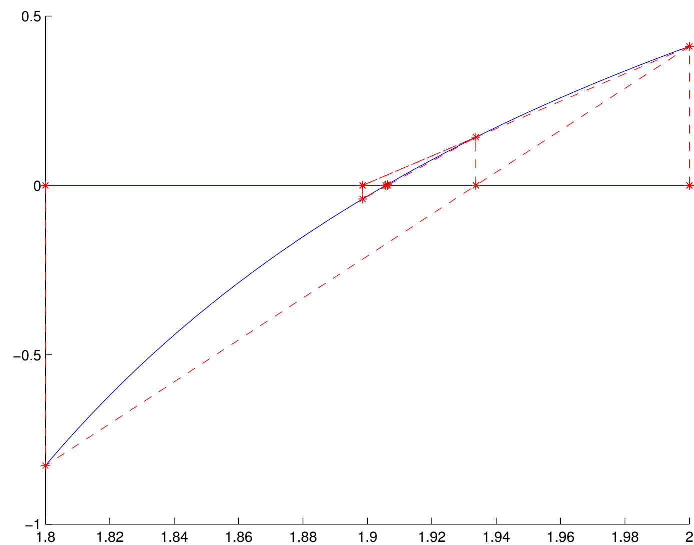
Metoda daná tímto iteračním vztahem se nazývá *metoda sečen*. Tento název je odvozen z toho, že tečná z Newtonovy metody je nahrazena sečnou protínající graf funkce f ve dvou bodech.

Z iteračního vztahu metody sečen také plyne, že na jejím začátku potřebujeme dvě iterace – x_0 a x_1. Také v každém kroku používáme dvě funkční hodnoty, stačí ale počítat jen jednu a tu druhou si pamatovat od minule. Vyzkoušejme si opět metodu sečen v Matlabu na úvodním příkladu:

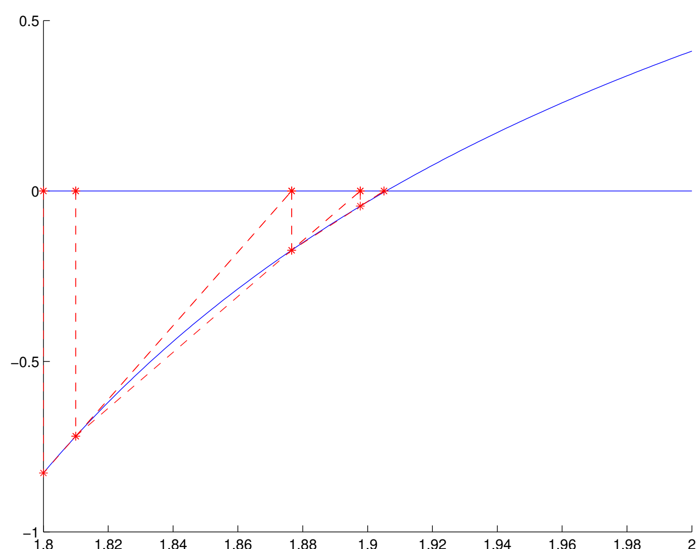
```
>> f=inline('x-tan(x)+pi/(2*cos(x))');
>> x0=1.8;
>> x1=2;
>> f0=f(x0)
f0 =
-0.8274
>> f1=f(x1)
f1 =
0.4104
>> x2=x1-f1*(x1-x0)/(f1-f0)
x2 =
1.9337
>> f2=f(x2)
f2 =
0.1422
>> x3=x2-f2*(x2-x1)/(f2-f1)
x3 =
1.8985
>> f3=f(x3)
f3 =
-0.0401
>> x4=x3-f3*(x3-x2)/(f3-f2)
x4 =
1.9063
>> f4=f(x4)
f4 =
0.0031
>> x5=x4-f4*(x4-x3)/(f4-f3)
x5 =
1.9057
>> f5=f(x5)
f5 =
6.1196e-05
>> x6=x5-f5*(x5-x4)/(f5-f4)
x6 =
1.9057
>> f6=f(x6)
```

```
f6 =
-9.5082e-08
>>
```

Počítat tímto způsobem je jistě poněkud nepraktické, zde to slouží jen jako ukázka. Čtenář by jistě zvládl celý problém řešit elegantněji, například pomocí cyklu. Grafické znázornění metody sečen ukazuje následující obrázek:



Není ovšem potřeba volit počáteční aproximace tak, aby funkční hodnoty měly opačná znaménka. Naopak, pokud budou blízko sebe, bude příslušná sečna bližší tečně z Newtonovy metody:

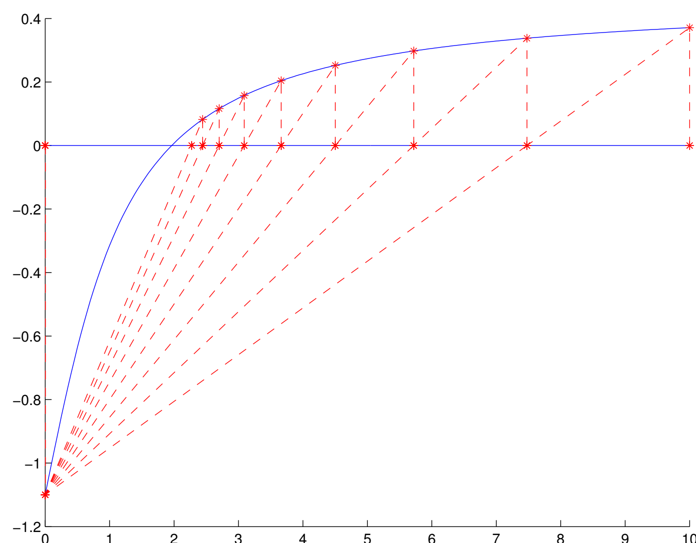


U metody *regula falsi* naopak počáteční aproximace z opačnými funkčními hodnotami požadujeme a tento požadavek pak platí pro všechny další iterace. Jedná se o podobnou myšlenku jako u metody půlení intervalu, iterace ale počítáme podobně jako u metody sečen:

$$x_{k+1} = x_k - f(x_k) \cdot \frac{x_k - x_s}{f(x_k) - f(x_s)},$$

kde s je největší index menší než k takový, že $f(x_s) \cdot f(x_k) \leq 0$.

Pro tuto metodu je typické, že pokud je funkce f na intervalu I konvexní nebo konkávní, jeden z bodů počátečních aproximací je tzv. *fixní*, tedy index s je pořád stejný, jak vidíme i z následujícího obrázku:



5.3 Rychlost konvergence

Z uvedených příkladů vidíme, že rychlost konvergence se liší u různých metod. Proto si definujme *řád konvergence* iterační metody, který udává rychlost její konvergence:
Nechť posloupnost (x_k) je iterační posloupnost získaná nějakou metodou, $x_k \to \xi$. Označme e_k chybu v k-tém kroku, t.j. $e_k = x_k - \xi$. Řekneme, že iterační metoda je řádu $p \geq 1$, jestliže platí

$$\lim_{k\to\infty} \frac{|e_{k+1}|}{|e_k|^p} = C \neq 0.$$

Řád konvergence vlastně říká, jak musíme umocnit chybu v k-tém kroku, abychom limitně dostali chybu v kroku $k+1$ (až na násobení konstatnou). Je jasné, že čím vyšší je řád metody, tím je konvergence rychlejší. Například pro $p = 1$ (říkáme také, že metoda je lineárního řádu) je přechod mezi jednotlivými chybami jen přibližně násobení konstantou (ta tedy musí být menší než 1), kdežto pro metodu řádu 2 (kvadratického řádu) platí, že pokud je chyba v některém kroku v desetinách, v dalším bude v setinách, v následujícím už v desetitisícinách atd.

Pro metody, které jsem si ukazovali je řád znám. Bisekce je lineární, Newtonova metoda kvadratická, metoda sečen má řád roven zlatému řezu $\frac{1+\sqrt{5}}{2}$, metoda regula falsi je opět jen lineární. Pro prostou iterační metodu s funkcí

g platí, že pokud g má spojité derivace do řádu p včetně, ξ je pevný bod funkce a $g'(\xi) = \ldots = g^{(p-1)}(\xi) = 0$, $g^{(p)}(\xi) \neq 0$, pak metoda je řádu p.

Je známa metoda, pomocí níž se dá rychlost konvergence urychlit. Jedná se o Aitkenovu δ^2 metodu, která je založena na tom, že pokud by iterační posloupnost konvergovala přesně lineárně, je možné pomocí jednoduchého výpočtu ze tří členů určit limitu.

Mějme tedy posloupnost (x_k). Z ní sestrojíme posloupnost $(\hat{x}_k)$ s použitím vztahu

$$\hat{x}_k = x_k - \frac{(x_{k+1} - x_k)^2}{x_{k+2} - 2x_{k+1} + x_k}.$$

Platí, že pokud původní posloupnost konverguje k ξ, konverguje k této hodnotě i nová posloupnost. Dále pokud je řád konvergence původní posloupnosti lineární, je nová posloupnost řádu alespoň kvadratického, je li řád původní posloupnosti $p > 1$, je řád nové posloupnosti $2p-1$. Tento postup tedy urychlí i Newtonovu metodu.

5.3.1 Steffensenova metoda

Aitkenovu metodu nelze použít tak, jak je uvedena, jelikož bychom potřebovali znát celou původní posloupnost, dokázali bychom tedy odhadnout i její limitu, nebylo by tedy potřeba počítat ji znovu, byť rychleji. Nicméně kombinací Aitkenovy metody s prostou iterační metodou získáme Steffensenovu metodu, která zpravidla dává rychle konvergující posloupnost. Myšlenka Steffensenovy metody je prostá: určíme tři iterace prostou iterační metodou (počáteční iteraci a dvě další) a pak použijeme Aitkenovu metodu na urychlení výpočtu. Pak určíme další tři iterace prostou iterační metodou a zase provedeme urychlení, což opakujeme, dokud nedosáhneme požadované přesnosti. Celý postup můžeme zapsat třeba takto (pro počáteční iteraci x_0):

$$y_0 = g(x_0), \quad z_0 = g(y_0), \quad x_1 = x_0 - \frac{(y_0 - x_0)^2}{z_0 - 2y_0 + x_0}$$

a obecný krok je pak

$$y_k = g(x_k), \quad z_k = g(y_k), \quad x_{k+1} = x_k - \frac{(y_k - x_k)^2}{z_k - 2y_k + x_k}.$$

Steffensenova metoda konverguje pro počáteční iteraci z nějakého okolí pevného bodu ξ funkce g, jestliže $g'(\xi) \neq 1$.

Příklad 4. Vyzkoušíme Steffensenovu metodu na řešení úvodního příkladu pro funkci $g(x) = \tan x - \frac{\pi}{2\cos x}$. Jako počáteční iteraci zvolíme $x_0 = 1.8$ a výpočet provedeme v Matlabu:

```
>> g=inline('tan(x)-pi/(2*cos(x))');
>> x=1.8;
>> y=g(x),z=g(y)
y =
2.6274
z =
1.2392
>> x=x-(x-y)^2/(z-2*y+x)
x =
2.1090
>> y=g(x),z=g(y)
y =
1.3894
z =
-3.2542
>> x=x-(x-y)^2/(z-2*y+x)
x =
2.2410
>> y=g(x),z=g(y)
y =
1.2672
z =
-2.0623
>> x=x-(x-y)^2/(z-2*y+x)
x =
2.6435
>> y=g(x),z=g(y)
y =
1.2442
z =
-1.9440
>> x=x-(x-y)^2/(z-2*y+x)
x =
3.7379
```

```
>>
```

Vidíme, že nevhodně zvolená počáteční iterace ke konvergenci nevede. Zkusíme počáteční iteraci poněkud zpřesnit:

```
>> x=1.85;
>> y=g(x),z=g(y)
y =
2.2117
z =
1.2865
>> x=x-(x-y)^2/(z-2*y+x)
x =
1.9517
>> y=g(x),z=g(y)
y =
1.7283
z =
3.7177
>> x=x-(x-y)^2/(z-2*y+x)
x =
1.9291
>> y=g(x),z=g(y)
y =
1.8087
z =
2.5417
>> x=x-(x-y)^2/(z-2*y+x)
x =
1.9121
>> y=g(x),z=g(y)
y =
1.8775
z =
2.0449
>> x=x-(x-y)^2/(z-2*y+x)
x =
1.9062
```

```
>> y=g(x),z=g(y)
y =
1.9034
z =
1.9159
>> x=x-(x-y)^2/(z-2*y+x)
x =
1.9057
>> y=g(x),z=g(y)
y =
1.9057
z =
1.9058
>> x=x-(x-y)^2/(z-2*y+x)
x =
1.9057
>>
```

5.4 Řešení systému nelineárních rovnic

Mějme systém nelineárních rovnic

$$\begin{aligned} f_1(x_1,\ldots,x_n) &= 0 \\ &\vdots \\ f_n(x_1,\ldots,x_n) &= 0 \end{aligned}$$

Můžeme jej stručně zapsat ve tvaru

$$F(\mathbf{x}) = \mathbf{0},$$

kde F i $\mathbf{x}$ bereme jako vektory. Nejpožívanější metodou je Newtonova metoda, kterou dostaneme tak, že vezmeme Taylorův rozvoj funkce více proměnných do prvního řádu.

Označme J_F matici derivací vektoru funkcí F, tedy

$$J_F(\mathbf{x}) = \left(\frac{\partial f_i(\mathbf{x})}{x_j} \right).$$

Pak Newtonovu metodu můžeme vyjádřit podobným způsobem jako je to pro funkci jedné proměnné:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - J_F^{-1}(\mathbf{x_k}) \cdot F(\mathbf{x_k}).$$

Protože ale práci s funkcemi více proměnných jsme zatím v Matlabu ani v Sage nezkoušeli, odložíme prozatím praktické vyzkoušení Newtonovy metody.

Kapitola 6

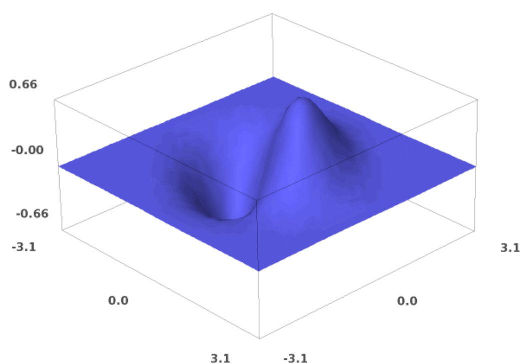
Funkce více proměnných

6.1 Zobrazovnání grafů

První věcí, kterou si ukážeme, či spíše zopakujeme, pro funkce více proměnných je nakreslení jejich grafu. To samozřejmě dokážeme jen pro funkce dvou proměnných. V Sage můžeme postupovat třeba takto:

```
sage: x,y = var('x,y')
sage: plot3d(sin(x+y)*exp(-(x^2+y^2)/2),(-pi,pi),(-pi,pi))
```

Tím dostaneme následující obrázek:

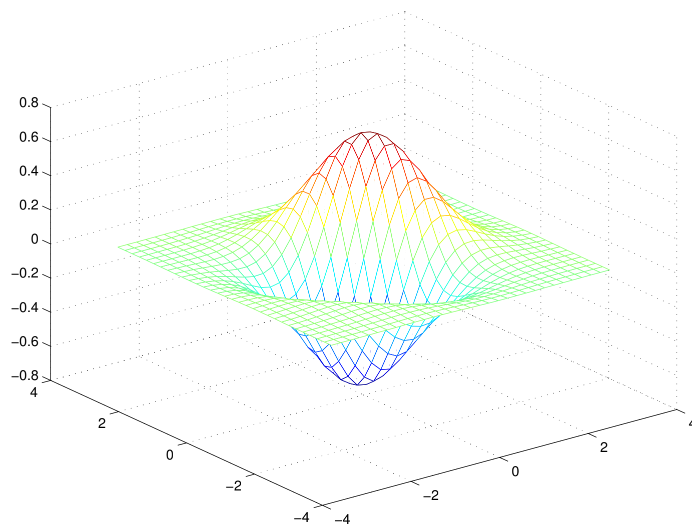


Získat podobný obrázek v Matlabu dá ovšem o něco víc práce. Nejprve musíme definovat síť bodů, v nichž se má funkce vyčíslit. K tomu se hodí matlabovská funkce `meshgrid`. Ukážeme její použití i výsledek, který dává:

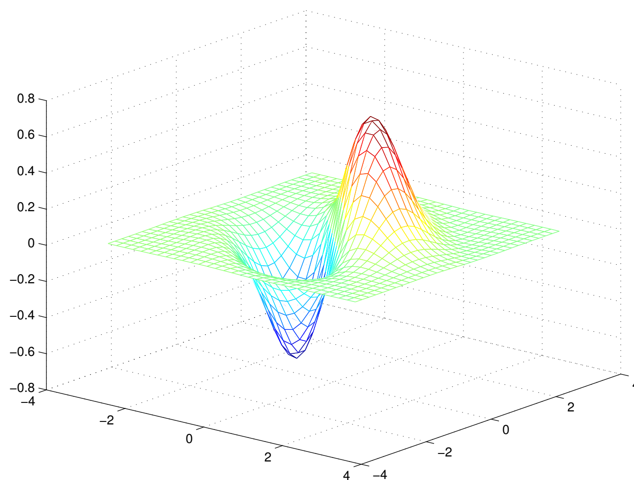
```
>> x=linspace(-pi,pi,31);
>> y=linspace(-pi,pi,31);
>> [X,Y]=meshgrid(x,y);
>> X(1:5,1:5)
ans =
-3.1416   -2.9322   -2.7227   -2.5133   -2.3038
-3.1416   -2.9322   -2.7227   -2.5133   -2.3038
-3.1416   -2.9322   -2.7227   -2.5133   -2.3038
-3.1416   -2.9322   -2.7227   -2.5133   -2.3038
-3.1416   -2.9322   -2.7227   -2.5133   -2.3038
>> Y(1:5,1:5)
ans =
-3.1416   -3.1416   -3.1416   -3.1416   -3.1416
-2.9322   -2.9322   -2.9322   -2.9322   -2.9322
-2.7227   -2.7227   -2.7227   -2.7227   -2.7227
-2.5133   -2.5133   -2.5133   -2.5133   -2.5133
-2.3038   -2.3038   -2.3038   -2.3038   -2.3038
>> size(X)
ans =
31    31
>>
```

Vidíme, že ve výsledku se opakují vstupní vektory, v jedné výstupní matici po sloupcích, ve druhé po řádcích. Tyto výstupní matice se pak dají použít pro výpočet funkčních hodnot a výsledek si pak zobrazíme:

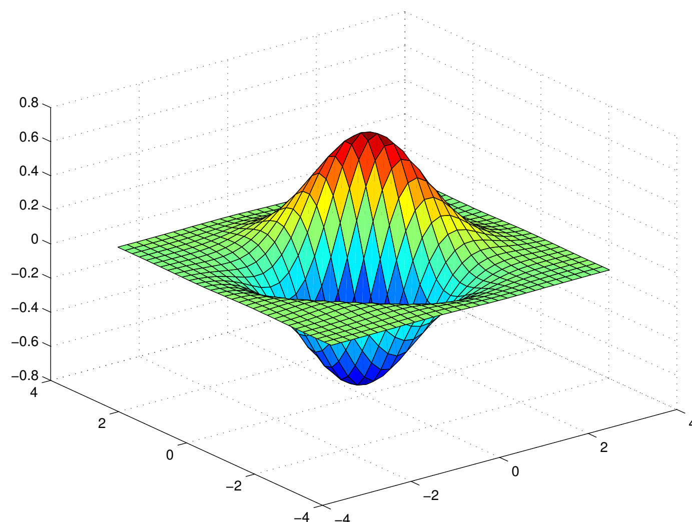
```
>> f=sin(X+Y).*exp(-(X.^2+Y.^2)/2);
>> mesh(x,y,f)
>>
```



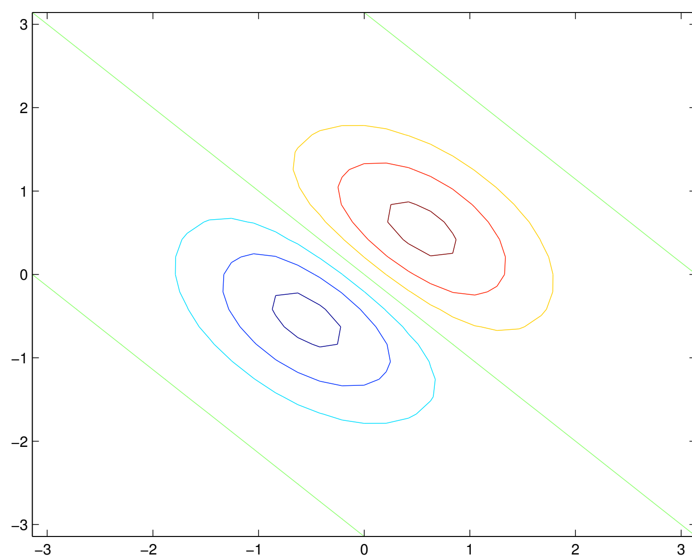
Graf funkce je zobrazen z jiné strany, pomocí ovládacích prostředků Matlabu si jej ovšem můžeme natočit do stejné polohy jako je to na předchozím obrázku.



Kromě funkce `mesh` můžeme také pro zobrazení použít funkci `surf`, která graf funkce f zobrazuje pomocí plošek:



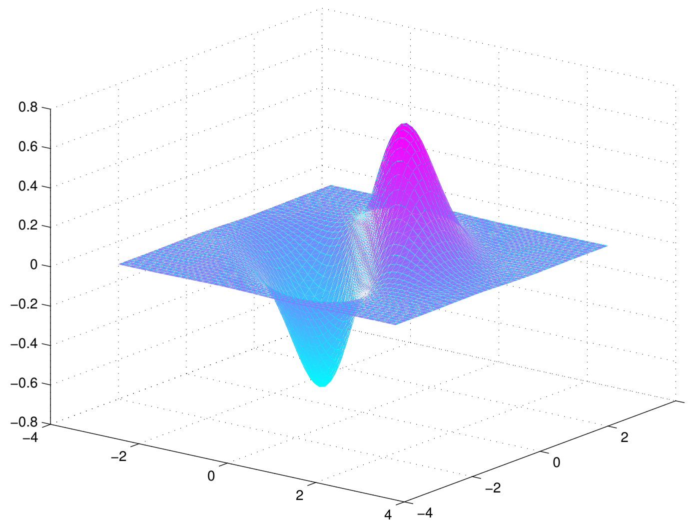
Další možností je zobrazit vrstevnice grafu, k tomu slouží funkce `contour`:



Můžeme samozřejmě graf funkce f zobrazit na jemnější síti, abychom dostali podobný výsledek jako u Sage. Je také možné použít jiné barevné zobrazení, k tomu slouží funkce `colormap`, a nastavit přechody mezi jednotlivými ploškami. Další podrobnosti je možné získat v dokumentaci:

```
>> x=linspace(-pi,pi,51);
>> y=linspace(-pi,pi,51);
```

```
>> [X,Y]=meshgrid(x,y);
>> f=sin(X+Y).*exp(-(X.^2+Y.^2)/2);
>> surf(x,y,f)
>> colormap(cool)
>> shading interp
>>
```



6.2 Výpočet parciáních derivací

Výpočet parciálních derivací je v obou našich používaných softwarech velmi jednoduchý. V Sage použijeme následující postup:

```
sage: x,y=var('x,y')
sage: f=sin(x+y)*exp(x-y)
sage: diff(f,x)
e^(x - y)*sin(x + y) + e^(x - y)*cos(x + y)
sage: diff(f,y)
-e^(x - y)*sin(x + y) + e^(x - y)*cos(x + y)
sage:
```

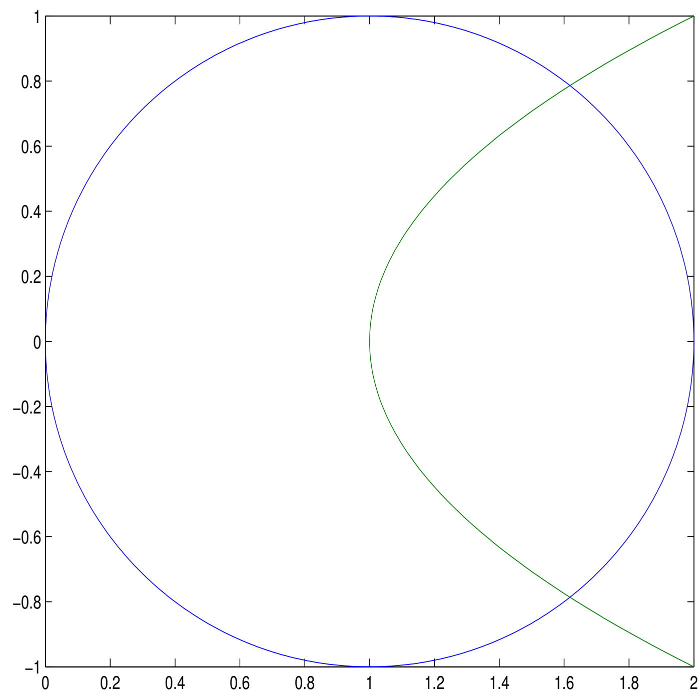
V Matlabu je postup obdobný:

```
>> f=sym('sin(x+y)*exp(x-y)')
f =
exp(x - y)*sin(x + y)
>> diff(f,'x')
ans =
exp(x - y)*cos(x + y) + exp(x - y)*sin(x + y)
>> diff(f,'y')
ans =
exp(x - y)*cos(x + y) - exp(x - y)*sin(x + y)
>>
```

Příklad 5. Nyní si můžeme v Matlabu ukázat, jak bychom řešili systém dvou nelineárních rovnic pomocí Newtonovy metody. Uvažujme rovnice

$$\begin{aligned} f_1(x,y) &= x^2 - 2x + y^2 = 0 \\ f_2(x,y) &= x - y^2 - 1 = 0 \end{aligned}$$

Hledáme vlastně průsečík kružnice a paraboly.



Najít v Matlabu přibližné řešení za pomocí Newtonovy metody můžeme třeba takto:

```
>> f1=sym('x^2-2*x+y^2');
>> f2=sym('x-y^2-1');
>> f1x=diff(f1,'x')
f1x =
2*x - 2
>> f1y=diff(f1,'y')
f1y =
2*y
>> f2x=diff(f2,'x')
f2x =
1
>> f2y=diff(f2,'y')
f2y =
-2*y
>> J=[f1x,f1y;f2x,f2y]
J =
[ 2*x - 2,  2*y]
[       1, -2*y]
>> F=[f1;f2]
F =
x^2 - 2*x + y^2
- y^2 + x - 1
>> x=2;y=1; %pocatecni aproximace
>> xy=[x;y]-inv(eval(J))*eval(F)
xy =
1.6667
0.8333
>> x=xy(1);y=xy(2);
>> xy=[x;y]-inv(eval(J))*eval(F)
xy =
1.6190
0.7881
>> x=xy(1);y=xy(2);
>> xy=[x;y]-inv(eval(J))*eval(F)
xy =
```

```
1.6180
0.7862
>> x=xy(1);y=xy(2);
>> xy=[x;y]-inv(eval(J))*eval(F)
xy =
1.6180
0.7862
>>
```

Newtonova metoda i v tomto případě konverguje velmi rychle. Zde by samozřejmě bylo možné najít analytické vyjádření řešení, neboť dosazením druhé rovnice do první bychom dostali kvadratickou rovnici. Čtenář tak může učinit pro jeho porovnání s výše získaným přibližným řešením.

6.3 Integrování ve více proměnných

Určit primitivní funkci podle jedné či druhé proměnné s použitím Sage nebo Matlabu je jednoduché:

```
sage: x,y=var('x,y')
sage: f=sin(x+y)*cos(x-y)^2
sage: f1=integral(f,x);f1
-1/12*cos(-3*x + y) + 1/4*cos(-x + 3*y) - 1/2*cos(x + y)
sage: f2=integral(f1,y);f2
-1/12*sin(-3*x + y) + 1/12*sin(-x + 3*y) - 1/2*sin(x + y)
sage:
```

```
>> f=sym('sin(x+y)*cos(x-y)^2');
>> f1=int(f,'y')
f1 =
cos(y - 3*x)/4 - cos(3*y - x)/12 - cos(x + y)/2
>> f2=int(f1,'x')
f2 =
sin(3*x - y)/12 - sin(x - 3*y)/12 - sin(x + y)/2
>>
```

Pak také není problémem spočítat určitý integrál přes obdélníkovou oblast, například pro uvedenou funkci přes $[0,\pi]\times[0,\pi/2]$, přitom by nemělo záležet na pořadí integrování:

```
sage: I1=integral(f,x,0,pi);I1
-1/2*cos(3*y) + 7/6*cos(y)
sage: Ix=integral(f,x,0,pi);Ix
-1/2*cos(3*y) + 7/6*cos(y)
sage: II=integral(Ix,y,0,pi/2);II
4/3
sage:
```

```
>> Iy=int(f,'y',0,pi/2)
Iy =
(4*cos(x))/3 + sin(x)/3 - cos(x)^3 + cos(x)^2*sin(x)
>> II=int(Iy,'x',0,pi)
II =
4/3
>>
```

Jestliže ale potřebujeme spočítat integrál přes jinou oblast, nezbývá, než si meze oblasti vyjádřit ručně, neboť zdá se, že integrovat přes obecné oblasti zatím námi používaný software neumí.

Takže pokus bychom potřebovali určit integrál z funkce $f(x,y)=(x+y)^2$ přes kruh se středem v počátku a poloměrem 2, musíme si pro dané y určit, v jakých mezích se pohybuje x. To je celkem snadné meze jsou $[-\sqrt{4-y^2},\sqrt{4-y^2}]$. Protože ale v mezích je obsaženo ometení pro y, musíme ho zadat i do výpočtu:

```
sage: f=(x+y)^2
sage: assume(abs(y)<=2)
sage: I1=integral(f,x,-sqrt(4-y^2),sqrt(4-y^2));I1
4/3*sqrt(-y^2 + 4)*(y^2 + 2)
sage: I2=integral(I1,y,-2,2);I2
8*pi
sage:
```

U jednoduchých oblastí, jako je například kruh či elipsa, si můžeme nechat hranice spočítat:

```
sage: S=solve(x^2+y^2-4,x);S
[x == -sqrt(-y^2 + 4), x == sqrt(-y^2 + 4)]
sage: I1=integral(f,x,S[0].right(),S[1].right());I1
4/3*sqrt(-y^2 + 4)*(y^2 + 2)
sage: I2=integral(I1,y,-2,2);I2
8*pi
sage:
```

V Matlabu se obejdeme bez předpokladů, nutno ovšem dodat, že výpočet posledního integrálů zde trval daleko delší dobu, než tomu bylo v Sage:

```
>> f=sym('(x+y)^2');
>> ylims=solve('x^2+y^2-4','y')
ylims =
(2 - x)^(1/2)*(x + 2)^(1/2)
-(2 - x)^(1/2)*(x + 2)^(1/2)
>> f=sym('(x+y)^2');
>> ylims=solve('x^2+y^2-4','x')
ylims =
(2 - y)^(1/2)*(y + 2)^(1/2)
-(2 - y)^(1/2)*(y + 2)^(1/2)
>> I1=int(f,'x',ylims(2),ylims(1))
I1 =
(4*(y^2 + 2)*(2 - y)^(1/2)*(y + 2)^(1/2))/3
>> I2=int(I1,'y',-2,2)
I2 =
8*pi
>>
```

Kapitola 7

Řešení diferenciálních rovnic

Budeme se zabývat zejména hledáním řešení rovnice ve tvaru

$$y' = f(x, y)$$

pro nějakou funkci f, s případnou počáteční podmínkou $y(x_0) = y_0$ pak hovoříme o počáteční úloze. Zobecněním této úlohy jsou rovnice vyšších řádů, z nichž velkou důležitost mají okrajové úlohy – tedy rovnice druhého řádu se zadanými podmínkami v krajních bodech nějakého intervalu $[a, b]$:

$$y'' = f(x, y, y'), \quad y(a) = y_1, \quad y(b) = y_2$$

7.1 Analytické řešení

Matlab i Sage obsahují nástroje pro nalezení analytického řešení diferenciální rovnice. V Matlabu se používá funkce `dsolve`, které v nejjednodušší podobě stačí jeden vstupní parametr – textový řetězec s rovnicí, kde derivace je označena jako `D`: Základní typy rovnic zvládá bez problémů:

```
>> dsolve('Dy = y')
ans =
C4*exp(t)
>> dsolve('Dy = -y')
ans =
C2/exp(t)
>> dsolve('Dy = t*y')
```

```
ans =
C6*exp(t^2/2)
>> dsolve('Dy = y/t')
ans =
C8*t
>>
```

Je možné použít i rovnici s parametrem a pokud Matlab rovnici nedokáže vyřešit, slušně nám to oznámí:

```
>> dsolve('Dy = a*y*sin(t)')
ans =
C10/exp(a*cos(t))
>> dsolve('Dy = cos(t*y)')
Warning: Explicit solution could not be found.
> In dsolve at 161
ans =
[ empty sym ]
>>
```

Lze také samozřejmě označit jinak výstupní nezávislou proměnnou nebo funkci, kterou hledáme, či zadat počáteční podmínku:

```
>> dsolve('Dy = y^2','x')
ans =
0
-1/(C38 + x)
>> dsolve('Dx = -s*x^2','s')
ans =
0
-1/(- s^2/2 + C54)
>> dsolve('Dy = -y^2','y(1)=1','x')
ans =
1/x
>>
```

Problémem nejsou ani okrajové úlohy pro rovnice druhého řádu:

```
>> dsolve('D2y = -y+cos(x)','y(0)=0','y(pi/2)=1','x')
ans =
cos(3*x)/8 - cos(x)/8 + sin(x)*(x/2 + sin(2*x)/4)
- sin(x)*(pi/4 - 1)
>>
```

Řešení diferenciálních rovnic v Sage je také snadné. Nejprve definujeme `x` jako nezávislou proměnnou, `y` jakožto funkci proměnné `x` a pro nalezení řešení rovnice použijeme funkcei `desolve`. Zde se implicitně předpokládá, že pokud pravá strana rovnice není uvedena, je nulová. Stejnou rovnici tedy mohu zadat dvěma způsoby:

```
sage: x=var('x')
sage: y=function('y',x)
sage: desolve(diff(y,x) ==x*y, y)
c*e^(1/2*x^2)
sage: desolve(diff(y,x) -x*y, y)
c*e^(1/2*x^2)
sage:
```

V některých případech je potřeba z výsledku funkci `y` vyjádřit a to i při zadání počáteční podmínky:

```
sage: desolve(diff(y,x) == y^2, y)
-1/y(x) == c + x
sage: desolve(diff(y,x) == y^2, y,[1,-1])
-1/y(x) == x
sage:
```

Pro rovnice vyšších řádu se uvádí řád derivace jako třetí parametr funcke `diff`. Zkusíme-li vyřešit v Sage stejnou okrajovou úlohu jako v Matlabu, dostaneme o něco jednodušší vyjádření výsledku:

```
sage: desolve(diff(y,x,2) == -y+cos(x), y,[0,0,pi/2,1])
-1/4*(pi - 4)*sin(x) + 1/2*x*sin(x)
sage:
```

Čtenář si jako cvičení může zkusit ověřit, zda oba výsledky jsou totožné.

V případě, že Sage nějakou rovnici nedokáže vyřešit (nebo uděláme chybu v zadání), objeví se chybové hlášení, ze kterého je vidět, že Sage pro řešení diferenciálních rovnic používá software zvaný Maxima. Více informací o něm mohou zájemci zjistit na internetových stránkách `http://maxima.sourceforge.net/`.

7.2 Numerické řešení

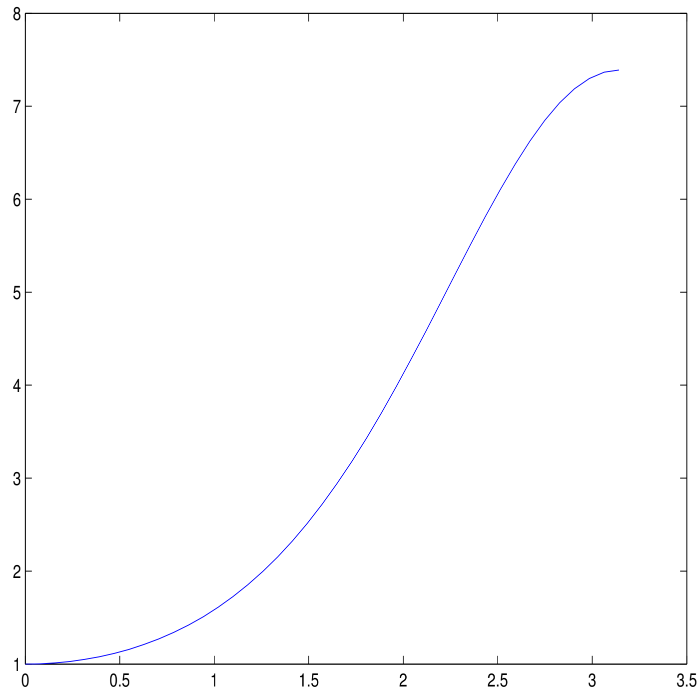
Pro nalezení numerického řešení budeme používat Matlab. Numerické řešení hledáme na nějaké síti bodů $(x_i)_{i=0}^n$, takže nelze hledat obecné řešení, ale pouze přibližné partikulární řešení nějaké počáteční úlohy.

V současné době patrně nejpoužívanější metody jsou metody Runge–Kutta, které patří mezi jednokrokové metody, to znamená, že výpočet přibližných hodnot hledané funkce y v bodě x_{k+1} probíhá na základě hodnoty v předchozím bodě x_k. Kromě toho známe také vícekrokové metody (pro výpočet v x_{k+1} se použije více předchozích hodnot), ale ty jsou v základním matlabovském balíku využívány jen velmi málo (funkce `ode113`).

Nejpoužívanějšími metodami Runge–Kutta jsou metody čtvrtého řádu, v Matlabu je jedna z nich implementována ve funkci `ode45`. Nápověda k této funkci je poměrně obsáhlá a taky v ní můžeme zjisti další funkce použitelné pro numerické řešení diferenciálních rovnic. Zaměříme se jen na tuto funkci a postačí nám základní použití.

V nejjednodušší verzi musíme zadat funkci f z pravé strany diferenciální rovnice, interval, na němž chceme přibližné řešení spočítat a také počáteční podmínku. Výsledkem jsou dva vektory – síť bodů, a příslušné funkční hodnoty. Výsledek si zobrazíme:

```
>> f=inline('y*sin(t)');
>> I=[0,pi];
>> cond=[1];
>> [t,y]=ode45(f,I,cond);
>> plot(t,y)
>>
```



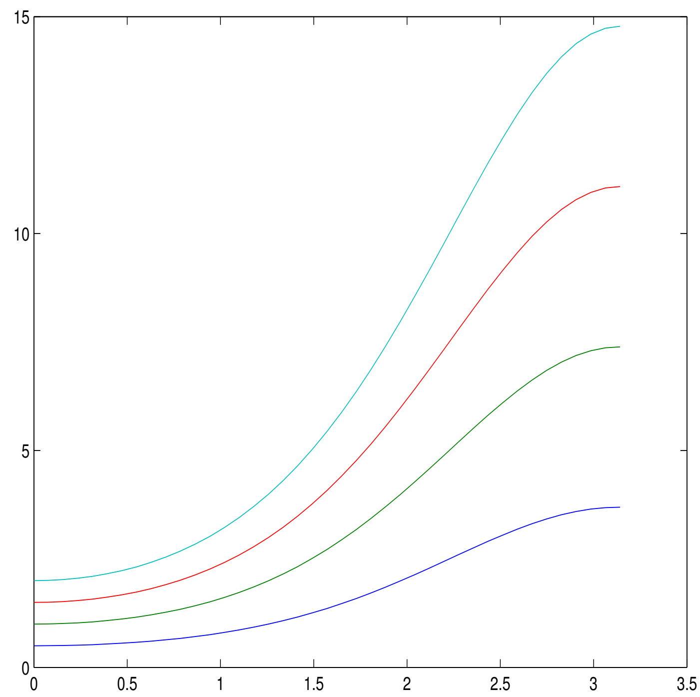
Protože nás zajímá, s jakou chybou je řešení určeno, zkusíme je porovnat s hodnotami řešení přesného:

```
>> dsolve('Dy = y*sin(x)','y(0)=1','x')
ans =
exp(1)/exp(cos(x))
>> ff=inline('exp(1)./exp(cos(x))')
ff =
Inline function:
ff(x) = exp(1)./exp(cos(x))
>> yy=ff(t);
>> max(abs(y-yy))
ans =
2.1830e-05
>>
```

Chyba je tedy poměrně velmi malá, na grafickém znázornění by se přesné řešení od přibližného nedalo rozlišit.

Co se týče počátečních podmínek, berou se automaticky v krajním bodě zadaného intervalu, ale je možné jich zadat více najednou:

```
>> cond=[0.5 1 1.5 2];
>> [t,y]=ode45(f,I,cond);
>> plot(t,y)
>>
```



Kapitola 8

Statistické metody

V této kapitole se nebudeme věnovat teorii pravděpodobnosti a teoretické statistice, zkusíme se zaměřit na praktické statistické výpočty.

Co se týče pravděpodobnosti a statistiky, je Matlab skvěle vybaven, zejména pokud máte k dispozici statistický toolbox. Zde jsou implementovány všechny běžné metody a algoritmy, ale také spousta metod a algoritmů speciálních. Protože popis celého statistického toolboxu je nad rámec tohoto textu, omezíme se toliko na základní použití.

Pokud chceme vyzkoušet některé dostupné programy a nemáme k dispozici vhodná data, není problém si je vygenerovat. nabízí se generátor náhodných čísel, který umí vytvořit data s požadovaným rozdělením. Tak například pro data s normálním rozdělením lze použít funkci `normrnd`, s exonenciálním `exprnd`, s rovnoměrným `unifrnd`, s beta rozdělením `betarnd` atd.

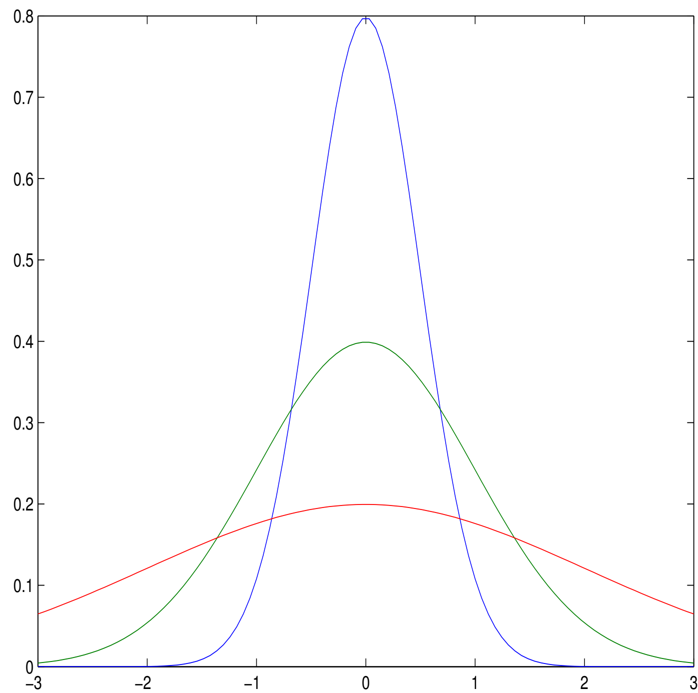
Podobně pro výpočet hustoty náhodné veličiny jsou k použití funkce končící „`pdf`“ (z anglického *probability density function*), tedy `normpdf`, `unifpdf`, `exppdf` apod., pro distribuční funkce můžeme požít funkce `normcdf`, `unifcdf`, `expcdf` atd. (z anglického *cumulative distribution function*).

Pro kvantily se používají funkce `norinv`, `unifinv`, `expinv` apod. jako hodnoty inverzní distribuční funkce. Co se týče parametrů uvedených funkcí, zaleží samozřejmě na tom, jakými parametry je určeno konkrétní rozdělení, takže nejlepší bude poradit se v každém konkrétním případě s nápovědou.

Pokud bychom se třeba chtěli podívat, jak se mění hustota normální rozdělení v závislosti na směrodatné odchylce, stačí zadat:

```
>> x=linspace(-3,3);
```

```
>> s1=0.5;s2=1;s3=2;
>> y1=normpdf(x,0,s1);
>> y2=normpdf(x,0,s2);
>> y3=normpdf(x,0,s3);
>> plot(x,[y1;y2;y3])
```



Když už jsem se zmínili o směrodatné odchylce, pro její výpočet se používá příkaz `std`, pro rozptyl příkaz `var`. A nesmíme zapomenout na střední hodnotu a medián – `mean` a `median`.

Tyto čtyři funkce nejsou součástí statistického toolboxu ale jsou již v základní matlabovské výbavě. Jedná se samozřejmě o odhady příslušných charakteristik náhodných veličin, nikoliv o přesné hodnoty.

Zkusme porovnat u vygenerovaných dat teoretické hodnoty s vypočítanými. Pokud bychom v podobném duchu udělali rozsáhlé testování, mohou nám výsledky leccos napovědět o kvalitě generátoru náhodných čísel, který Matlab používá.

Vygenerujeme 100 normálně rozdělených hodnot se střední hodnotou 1 a rozptylem 0.25. Nesmíme ovšem zapomenout, že jako parametr zadáváme směrodatnou odchylku:

```
>> x=normrnd(1,0.5,1,100);
>> mean(x)
ans =
1.0615
>> median(x)
ans =
1.0477
>> var(x)
ans =
0.3378
>> std(x)
ans =
0.5812
>>
```

Vidíme, že u rozptylu je relativně největší chyba, z jednoho vzorku ovšem nemůžeme vyvozovat žádné závěry.

8.1 Intevalové odhady

Jednou ze základních statistických úloh je určení intervalu, v němž s danou pravděpodobností leží nějaký parametr náhodného rozdělení. Nejčastěji předpokládáme, že máme data s normálním rozdělením nebo s rozdělením s normálního odvozeným.

Podíváme se, jak se mění interval spolehlivosti pro odhad střední hodnoty normálních dat, pokud roste jejich počet. Nejprve předpokládejme, že známe hodnotu rozptylu, pak se používají kvantily standardizovaného normálního rozdělení:

```
>> n=100;
>> mu=1;
>> sigma=0.5;
>> X=normrnd(mu,sigma,1,n);
>> M=mean(X)
M =
0.9813
>> alpha=0.05;
```

```
>> D=M-norminv(1-alpha/2,0,1)*sigma/sqrt(n);
>> H=M+norminv(1-alpha/2,0,1)*sigma/sqrt(n);
>> I=[D,H]
I =
0.8833    1.0793
>> n=500;
>> X=normrnd(mu,sigma,1,n);
>> M=mean(X)
M =
0.9878
>> D=M-norminv(1-alpha/2,0,1)*sigma/sqrt(n);
>> H=M+norminv(1-alpha/2,0,1)*sigma/sqrt(n);
>> I=[D,H]
I =
0.9439    1.0316
>> n=1000;
>> X=normrnd(mu,sigma,1,n);
>> M=mean(X)
M =
1.0266
>> D=M-norminv(1-alpha/2,0,1)*sigma/sqrt(n);
>> H=M+norminv(1-alpha/2,0,1)*sigma/sqrt(n);
>> I=[D,H]
I =
0.9956    1.0576
>>
```

V případě, že rozptyl není známý, použijeme kvantily Studentova rozdělení:

```
>> n=100;
>> alpha=0.05;
>> mu=1;
>> sigma=0.5;
>> X=normrnd(mu,sigma,1,n);
>> M=mean(X)
M =
0.9590
>> S=std(X)
```

```
S =
0.5047
>> D=M-tinv(1-alpha/2,n-1)*S/sqrt(n);
>> H=M+tinv(1-alpha/2,n-1)*S/sqrt(n);
>> I=[D,H]
I =
0.8589    1.0592
>> n=500;
>> X=normrnd(mu,sigma,1,n);
>> M=mean(X)
M =
1.0160
>> S=std(X)
S =
0.4909
>> D=M-tinv(1-alpha/2,n-1)*S/sqrt(n);
>> H=M+tinv(1-alpha/2,n-1)*S/sqrt(n);
>> I=[D,H]
I =
0.9729    1.0592
>> n=1000;
>> X=normrnd(mu,sigma,1,n);
>> M=mean(X)
M =
1.0172
>> S=std(X)
S =
0.4930
>> D=M-tinv(1-alpha/2,n-1)*S/sqrt(n);
>> H=M+tinv(1-alpha/2,n-1)*S/sqrt(n);
>> I=[D,H]
I =
0.9866    1.0478
>>
```

Je jasné, že inerval se zkracuje (délka klesá s o odmocninou n), ale rozdíl mezi oběma případy není patrný.

8.2 Testování hypotéz

Pro testování hypotéz je v Matlabu možné využít již hotových programů. Základním z nich je `ttest`, který v nejjednodušší verzi testuje, zda rozptyl náhodného výběru je nulový. Hladina významnosti je implicitně 5 procent:

```
>> n=100;
>> mu=1;
>> sigma=0.5;
>> X=normrnd(mu,sigma,1,n);
>> ttest(X)
ans =
1
```

Výsledek 1 znamená zamítnutí hypotézy. Pokud bychom chtěli vědět také pravděpodobnost (tzv. p-hodnota), přidáme výstupní parametr:

```
>> [H,P]=ttest(X)
H =
1
P =
1.1433e-32
>>
```

Vidíme, že střední hodnota není nulová téměř jistě. Provedeme tady drobnou úpravu:

```
>> M=mean(X)
M =
0.9926
>> [H,P]=ttest(X,M)
H =
0
P =
1
>>
```

Ještě si otestujeme, jak dopadne skutečná střední hodnota použitá při generování náhodného výběru:

```
>> [H,P]=ttest(X,1)
H =
0
P =
0.8946
>>
```

Funkce `ttest` se dá také použít na testování rovnosti středních hodnot dvou náhodných výběru stejného rozsahu:

```
>> Y=normrnd(mu,sigma,1,n);
>> [H,P]=ttest(X,Y)
H =
0
P =
0.5040
>> Y=normrnd(mu-0.1,sigma,1,n);
>> [H,P]=ttest(X,Y)
H =
0
P =
0.2128
>>
```

K testování rovnosti středních hodnot u dvou výběrů se dá ooužít také funkce `ttest2`, zdá se ale, že pracuje poněkud jinak než funkce `ttest`:

```
>> [H,P]=ttest(X,Y)
H =
0
P =
0.5040
>> [H,P]=ttest2(X,Y)
H =
0
P =
0.5276
>>
```

8.3 Lineární regrese

Studenti si lineární regresi občas pletou s konstrukcí regresní přímky, což je ale jen speciální případ lineární regrese. Na tuto problematiku jsme už narazili v prvním semestru, když jsem hovořili o pseudoinverzní matici.

V lineární regresi předpokládáme, že pozorovaná data, která zpracováváme, jsou lineárními kombinacemi funkcí předem daných hodnot, přičemž v pozorování je nějaký náhodná chyba. Pozorování je tedy náhodná veličina $Y(x) = \beta_1 f_1(x) + \cdots + \beta_k f_k(x) + \varepsilon$. Měříme hodnoty Y v zadaných bodech $x_1, \ldots, x_n$, $n \geq k$, tím dostáváme celkem soustavu n rovnic pro k neznámých. Matice soustavy se ve statistice nazývá *matice plánu* a zpravidla se označuje $\mathbf{X}$. Při řešení se snažíme minimalizovat reziduum (rozdíl mezi levou a pravou stranou), k čemuž se výborně hodí pseudoinverzní matice.

Jestliže navíc předpokládáme normalitu chyby *veps* se známým rozptylem, dají se spočítat i intervalové odhady pro parametry β_j.

Příklad 6. Mějme třídící algoritmus a chceme otestovat jeho kvalitu. Dobré třídící algoritmy pro n dat potřebují $O(n \log n)$ času, horší algoritmy potřebují $O(n^2)$ času. Algoritmus jsme otestovali pro náhodné výběry různých rozsahů a následná tabulka udává průměrné časy pro jednotlivé rozsahy v milisekundách:

n	100	200	300	400	500
t	0.42.883	127.39	264.08	435.69	665.0

n	600	700	800	900	1000
t	920.24	1220.4	1565.8	1941.4	2348.6

Vyrobíme matici plánu, přičemž předpokládáme, že výsledná funkce může být polynom druhého stupně doplněná funkcemi $\log n$ a $n \log n$.

```
>> n
n =
Columns 1 through 5
100         200         300         400         500
Columns 6 through 10
600         700         800         900         1000
>> t
t =
Columns 1 through 5
```

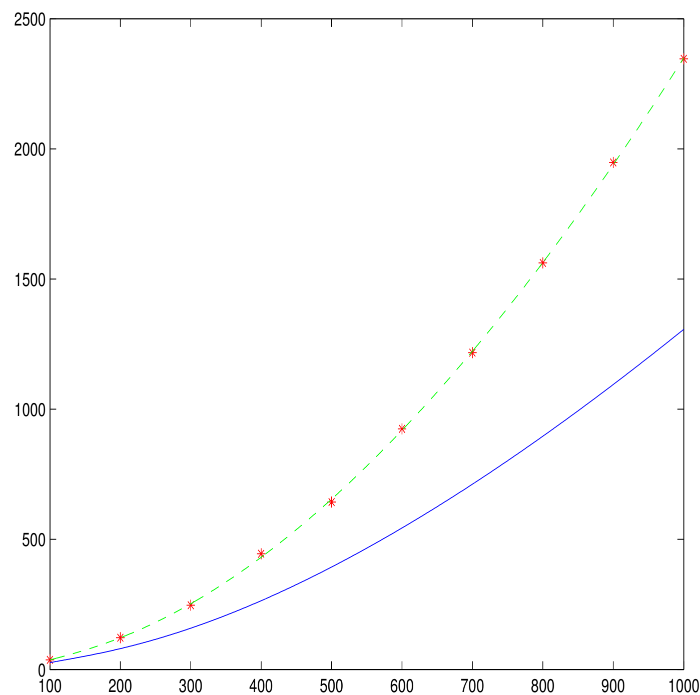
```
42.883    127.39    264.08    435.69    665.08
Columns 6 through 10
920.24    1220.4    1565.8    1941.4    2348.6
>> X1=ones(10,1);
>> X2=n';
>> X3=(n.^2)';
>> X4=(log(n))';
>> X5=(n.*log(n))';
>> X=[X1,X2,X3,X4,X5]
X =
1       100       10000     4.6052     460.52
1       200       40000     5.2983     1059.7
1       300       90000     5.7038     1711.1
1       400     1.6e+05     5.9915     2396.6
1       500     2.5e+05     6.2146     3107.3
1       600     3.6e+05     6.3969     3838.2
1       700     4.9e+05     6.5511     4585.8
1       800     6.4e+05     6.6846     5347.7
1       900     8.1e+05     6.8024     6122.2
1      1000       1e+06     6.9078     6907.8
>>
```

Nakonec provedeme výpočet odhadu parametrů s použitím pseudoinverzní matice:

```
>> Y=t';
>> beta=pinv(X)*Y
beta =
-278.98
-5.6652
0.0013371
99.146
0.90803
>>
```

Zdá se že koeficient u druhé mocniny je zanedbatelný oproti ostatním koeficientům. Podíváme se ještě, jak funkce, kterou jsme získali, prochází aproximuje data. Přitom ověříme rozdíl mezi tím když uvedený koeficient zanedbáme a když nikoliv:

```
>> nn=100:1000;
>> f1=beta(1)+beta(2)*nn+beta(4)*log(nn)+...
beta(5)*nn.*log(nn);
>> f2=beta(1)+beta(2)*nn+beta(3)*nn.^2+beta(4)*log(nn)+...
beta(5)*nn.*log(nn);
>> plot(n,t,'*r',nn,f1,'b',nn,f1,'g--')
>>
```



Vidíme, že koeficient zanedbat nemůžeme, protože je sice řádově daleko menší než ostatní koeficienty, ale vzhledem k tomu, že se jím násobí velká čísla (druhé mocniny n), nelze jej vynechat.

Literatura

[1] Horová, I.; Zelinka, J.: *Numerické metody*. Brno: Masarykova univerzita, druhé vydání, 2008, ISBN 978-80-210-3317-7.

[2] Kobza, J.: *Splajny*. Univerzita Palackého, 1993, ISBN 9788070672655.
URL `http://books.google.cz/books?id=-OnXYgEACAAJ`

[3] Ralston, A.: *Základy numerické matematiky*. Praha: Academia, druhé vydání, 1978.

[4] Schumaker, L.: *Spline Functions: Basic Theory*. Cambridge University Press, ISBN 9781139463430.
URL `http://books.google.cz/books?id=2uZLawUhXfgC`