INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Numerické výpočty II

Jiří Zelinka

Tento učební text vznikl za přispění Evropského sociálního fondu a státního rozpočtu ČR prostřednictvím Operačního programu Vzdělávání pro konkurenceschopnost v rámci projektu Univerzitní výuka matematiky v měnícím se světě (CZ.1.07/2.2.00/15.0203).

Obsah

1 Polynomy **4**
1.1 Hornerovo schema a základní úkony s polynomy 5
1.2 Interpolace 9
1.3 Kořeny polynomů 16
1.4 Splajny 25
1.5 Bersteinovy polynomy a Bézierovy křivky 31

2 Derivace **38**
2.1 Limity 38
2.2 Symbolické derivování 41
2.3 Taylorův rozvoj 43
2.4 Numerické derivování 44

3 Integrály **47**
3.1 Symbolické integrování 47
3.2 Numerické integrování 50

4 Řady **54**
4.1 Symbolické součty řad 54
4.2 Fourierovy řady 56

Úvod

Tento text je zaměřen na základy numerické aproximace, zejména polynomiální a splajnovou interpolaci, kromě toho ovšem se budeme zabývat také diferenciálním a integrálním počtem funkcí jedné proměnné. Bližší informace o teoretickém základu, na němž tento text staví může čtšnář najít zejména ve skriptech [1] a [2], dále také v knihách [3] nebo [4].

Kapitola 1

Polynomy

Kropáčku, jak vám vyšly ty kořeny polynomu? Haló, slyšíte?
Promiňte, pane profesore, trochu jsem se zamyslel. Na co jste se ptal?
Ptám se, jaké máte kořeny.
Moravské. Proč?

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 služku 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

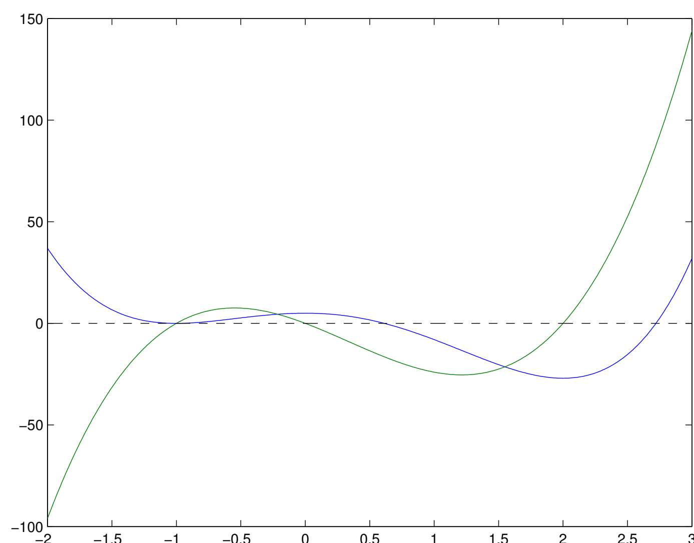
$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

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

Dobrý den, studenti, dnes nás čeká interpolace. Co myslíte, Kropáčku, že to je?
To je něco v parlamentu, ne?
To jste si spletl s interpelací.
Aha, no vlastně. A není to pátrání Interpolu?

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

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

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 operaci, 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 $\omega_{n+1}(x) = (x - x_0) \ldots (x - x_n) = \prod_{i=0}^{n}(x - x_i)$. 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
```

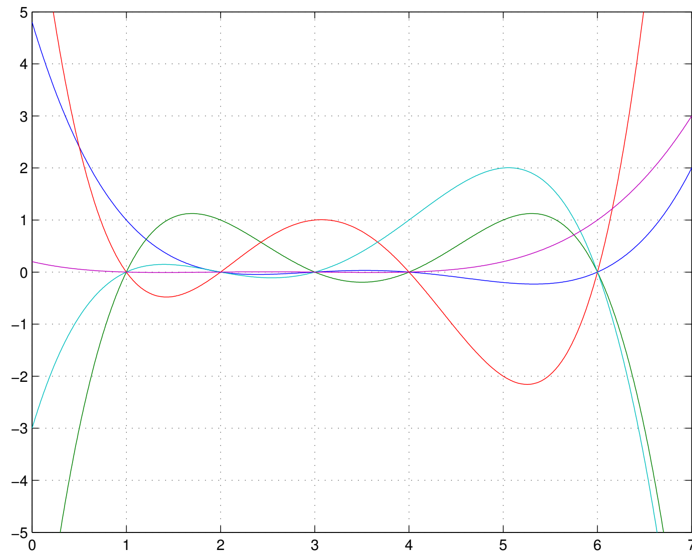
Jako nepovinný druhý výstupní parametr funkce vrací fundamentální polynomy řádcích matice.

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);
```

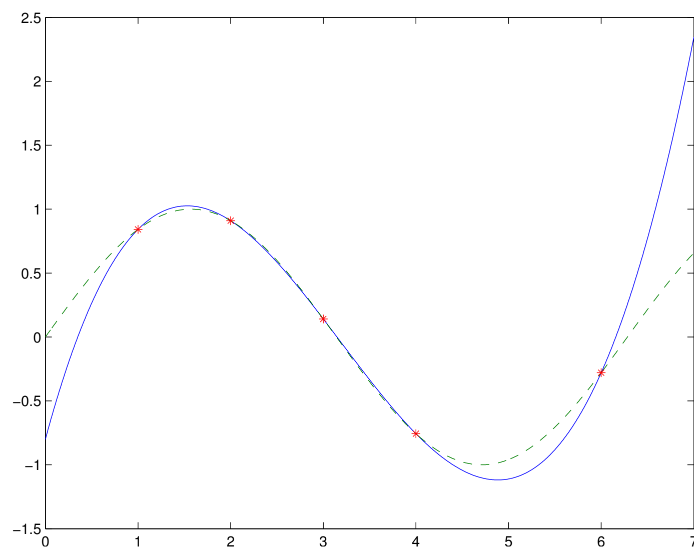
Protože máme spočtené i Lagrangeovy fundamentální polynomy, můžeme si je prohlédnout:

```
>> 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 $P_{i,i+1,\dots,i+k}$ interpolační polynom na uzlech $x_i,\dots,x_{i+k}$, 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 $P_{i,i+1,\dots,i+k}$ sestrojíme z polynomů nižších stupňů následujícím způsobem:
Nechť $P_{i,\dots,i+k-1}$ a $P_{i+1,\dots,i+k}$ jsou interpolační polynomy na příslušných uzlech, 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:

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

Postupným použitím vztahu (1.2) pak dostáváme

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

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

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

kde a_k*,* $k = 0, 1, \dots, n$*,* $a_0 a_n \neq 0$*, jsou koeficienty polynomu* P*, Pak pro všechny kořeny* $x_k, k = 0, 1, \dots, n$*, 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 $signP_1(x_0) = -signP_0'(x_0)$.*
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

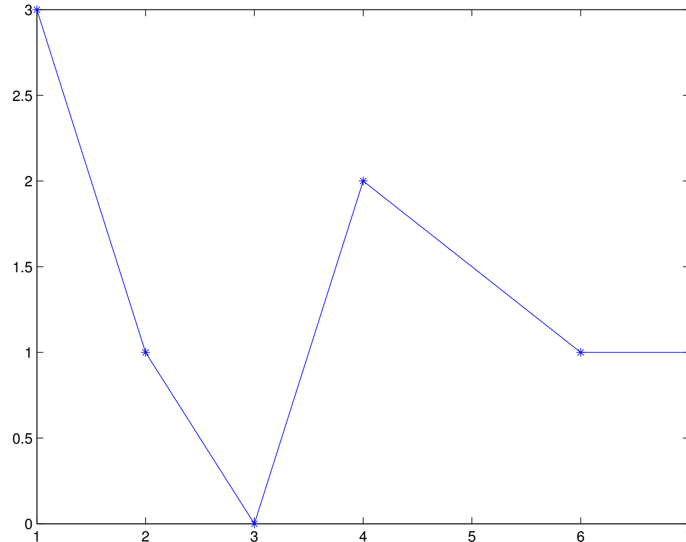
Pane profesore, tady v těch materiálech máte pravopisnou chybu.
Opravdum, Kropáčku? Ukažte.
Tady máte „splyne“ s měkkým i.
Vypadá to, Kropáčku, že kvůli vám budu muset začít používat český přepis slova spline.

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
```

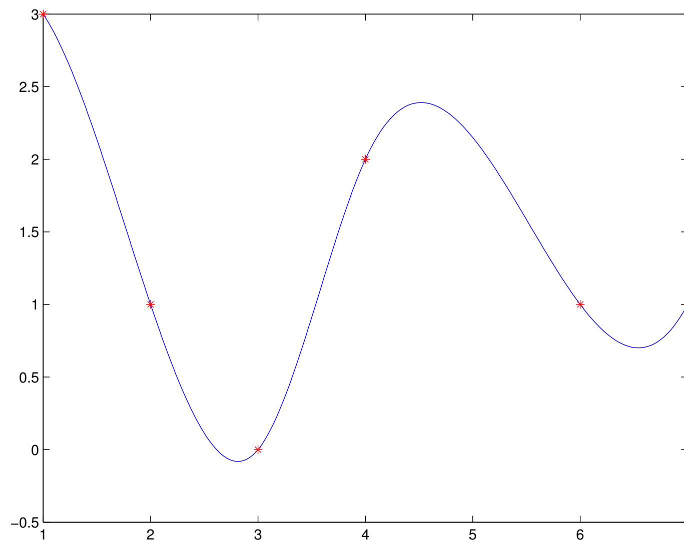
Označení 'pp' ukazuje, že se jedná o po částech polynomiální funkci, Dále následují uzly, tedy hraniční body pro jednotlivé části polynomu. Pak můžeme zjisti koeficienty na jednotlivých intervalech, počet intervalů a stupeň polynomu (ve skutečnosti spíše počet koeficientů na každém intervalu). Význam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

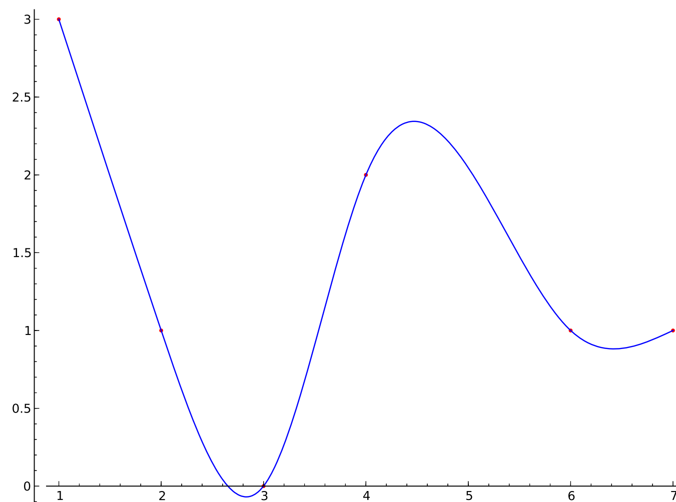
aspoň zjistíme, o jaký datový typ se jedná, ale koeficienty na jednotlivých intervalech nezjistíme. A pokud vím, u této základní funkce se tyto koeficienty přímo zjistit nedají. Pokud by je člověk opravdu potřeboval, musí si stáhnout další knihovny, které obsahují poněkud více sofistikované funkce pro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

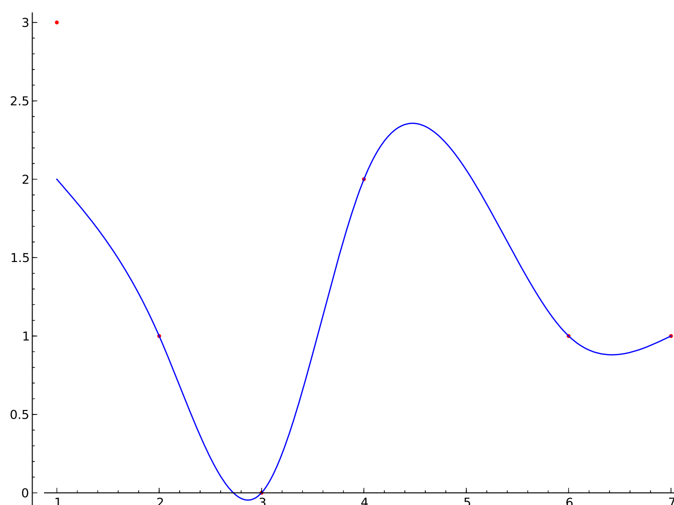
Nechat si splajn vykreslit (případně i s uzly a příslušnými hodnotami) taky není problém:

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



V Sage je velmi jednoduché změnit hodnotu v některém uzlu:

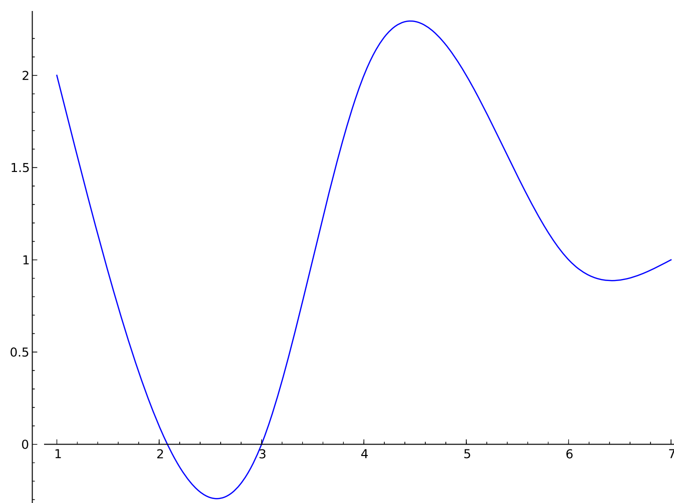
```
sage: S[0]
[1, 3]
sage: S[0]=[1,2]
sage: S
[[1, 2], [2, 1], [3, 0], [4, 2], [6, 1], [7, 1]]
sage: plot(S,(1,7))+list_plot(values,color='red')
```



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

Kropáčku, co víte o Bersteinovi?
West Side Story je fakt skvělý muzikál. Jako film dostal deset oskarů.
Aha, tak příště zkuste do vyhledávače přidat křestní jméno Sergej.

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

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

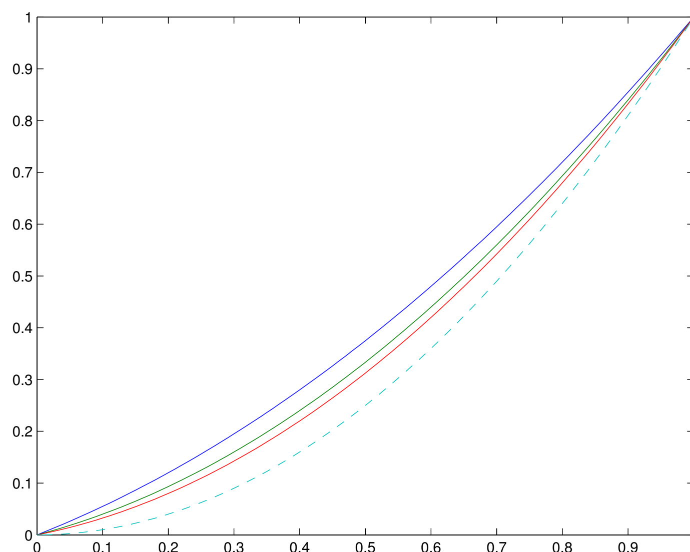
Dále nechť f je funkce definovaná na intervalu $[0, 1]$, položme $f_k = f(\frac{k}{n})$ pro $k = 0, \ldots, n$. Bernsteinův polynom stupně n funkce f definujeme výrazem

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

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 nazna-

čeno, 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

$$Q = Q(t) = P_0(1-t)^3 + 3P_1 t(1-t)^2 + 3P_2 t^2(1-t) + P_3 t^3$$

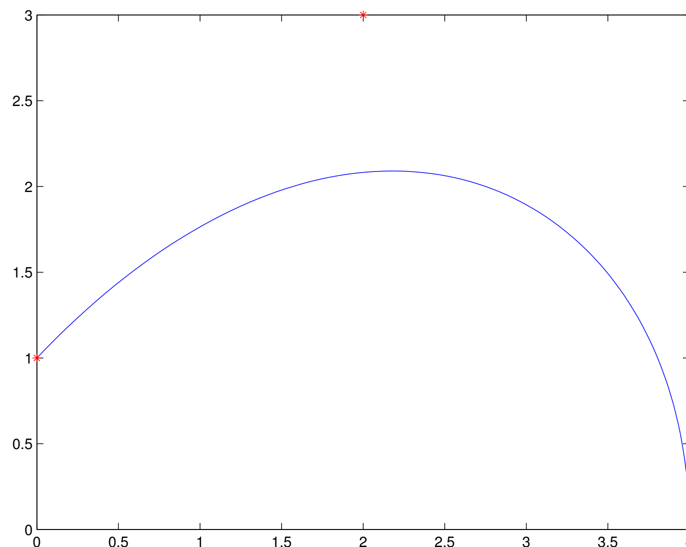
Z vyjádření navíc plyne, že pro $t = 0$ dostaneme výchozí kontrolní bod P_0, pro $t = 1$ koncový kontrolní bod P_3.

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

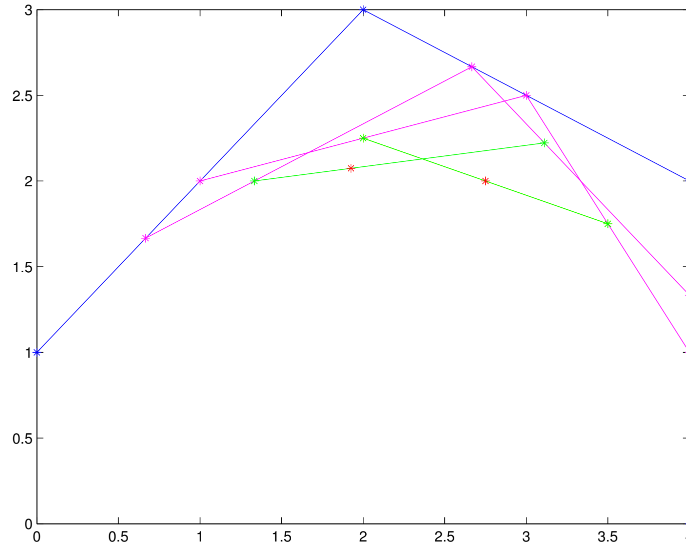
```
>> P0=[0;1]; P1=[2;3];P2=[4;2];P3=[4;0];
>> t=0:0.01:1;
>> Q=P0*((1-t).^3)+3*P1*(t.*(1-t).^2)+...
3*P2*(t.^2.*(1-t))+P3*(t.^3);
+>> plot(Q(1,:),Q(2,:))
```

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

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

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

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

Kapitola 2

Derivace

Pane profesore, chápu to správně, že když si jako funkci vyjádříme dráhu tělesa v v závislosti na čase, tak její derivace podle času je rychlost tělesa?
Ano, Kropáčku, to je běžný matematický model, který ve většině případů dává uspokojivé výsledky.
A jak zjistíme rychlost času, pane profesore?
To je velmi dobrá otázka, Kropáčku. Jak jste k ní dospěl?
No, některé přednášky, tím samozřejmě nemyslím vaše, se vlečou tak, že hodina trvá celé dopoledne, zatímco při posledním testu jsem si sotva stačil opsat zadání a hodina byla pryč.

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)
```

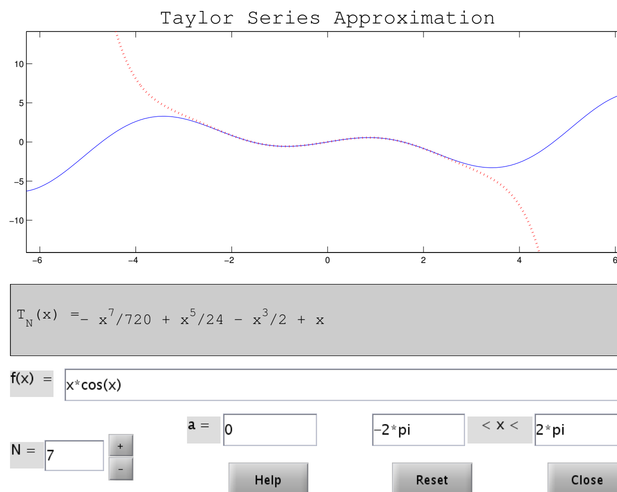
Derivování v Sage je podobně jednoduché, jen nejdříve musíme definovat proměnné. Máme dokonce na výběr, který příkaz pro derivování použijeme a samozřejmostí je počítání vyšších derivací.

```
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

Pane profesore, nevíte prosím, na jakých matematických principech je postavena důchodová reforma?
To netuším, Kropáčku, ale domnívám se, že to nebude nic složitého. Proč se na to ptáte?
No, četl jsem včera na jednom internetovém serveru, že „Zřízení sociální pojišťovny je integrální součástí důchodové reformy.“ *Ale není mi jasné, jaké funkce se vlastně integruje.*

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.

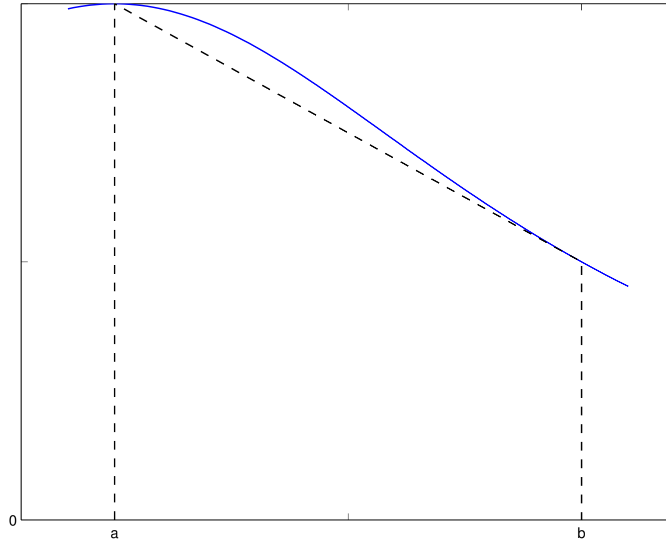
Odvodíme si některé nejpoužívanější formule. Situací s jedním uzlem se zabývat nebudeme, tak je až příliš jednoduchá. Pro dva uzly většinou máme $a = x_0 < x_1 = b$. V tomto případě má interpolační polynom tvar například

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

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

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

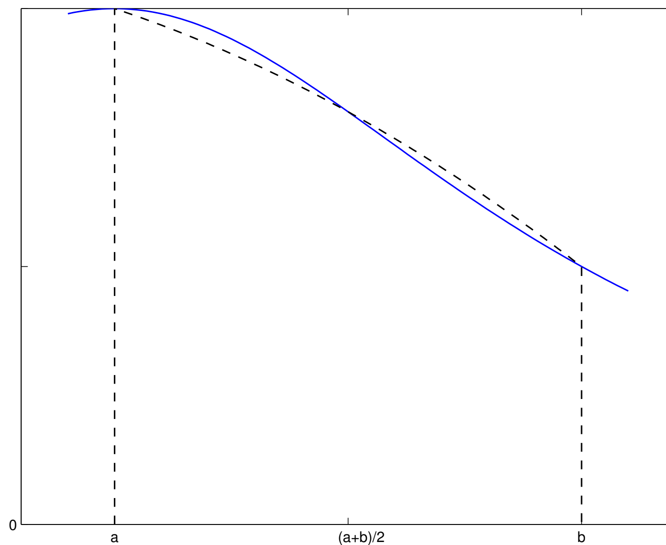
což je v podstatě plocha lichoběžníka (postaveného na bok),jehož strany mají délky rovny funkčním hodnotám funkce f v krajních bodech intervalu a jehož výška je rovna $b - a$. Proto se taky této formuli říká *lichoběžníkové pravidlo*.



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

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

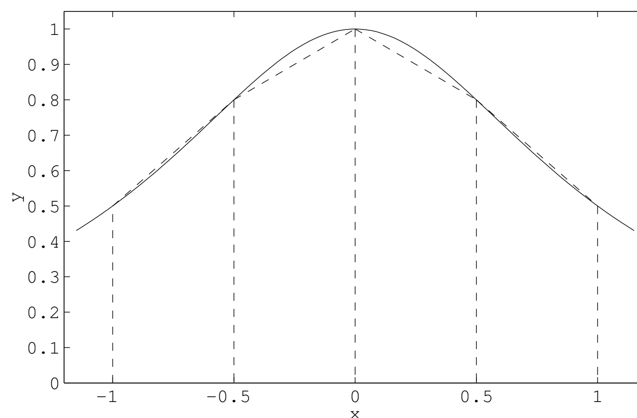
která se nazývá Simpsonovo pravidlo.



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 lichoběžníkovécho 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

Kryšpín Kropáček: Řada

Při sčítání řady dočkal jsem se zrady
ze strany své spolužačky Novákové Lady.
Nevím si s tím rady, skoro šilhám hlady,
nejvíc mě však rozptylují její krásné vnady.

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

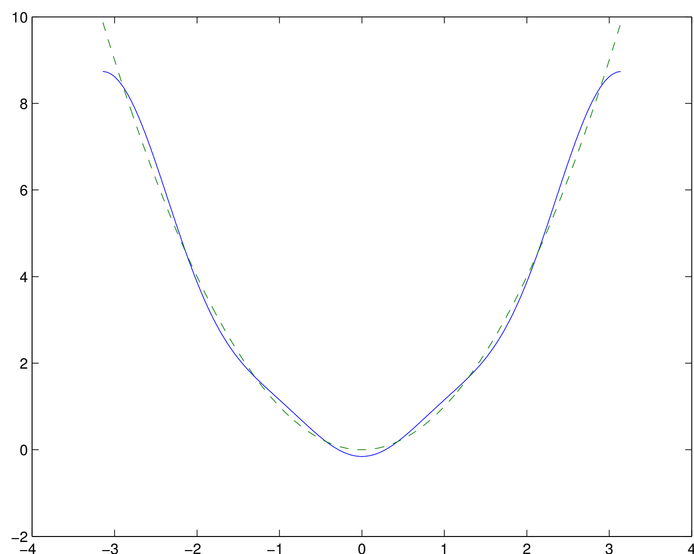
Spočítat koeficienty Fourierovy řady na základě toho, co už známe, jistě nebude pro čtenáře problém. Například pokud bychom chtěli v Matlabu určit prvních pár členů Fourierovy řady pro funkci x^2, můžeme postupovat třeba takto:

```
>> 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 $[-\pi, \pi]$:

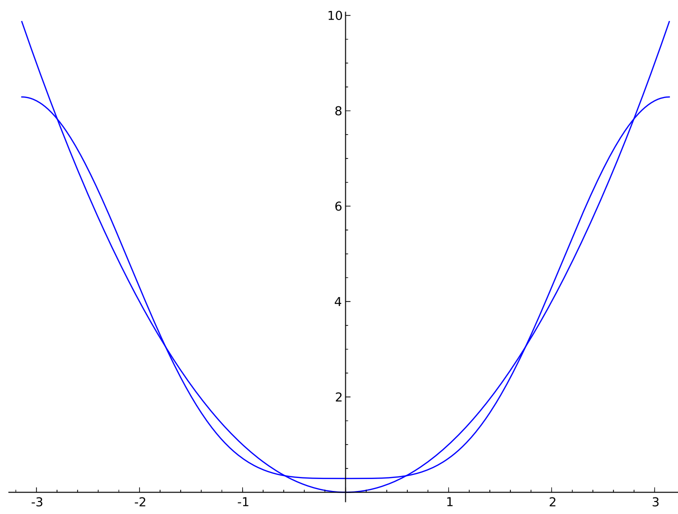
```
sage: x=var('x')
sage: f=Piecewise([[(-pi,pi),x^2]])
```

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

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`