KAPITOLA 11

Numerické algoritmy

V této kapitole:
- Vyhledávání nulových bodů funkcí
- Iterativní výpočet hodnot funkce
- Interpolace funkcí Lagrangeovou metodou
- Derivování funkcí
- Integrování funkcí Simpsonovou metodou
- Řešení soustav lineárních rovnic Gaussovou metodou
- Závěrečné poznámky

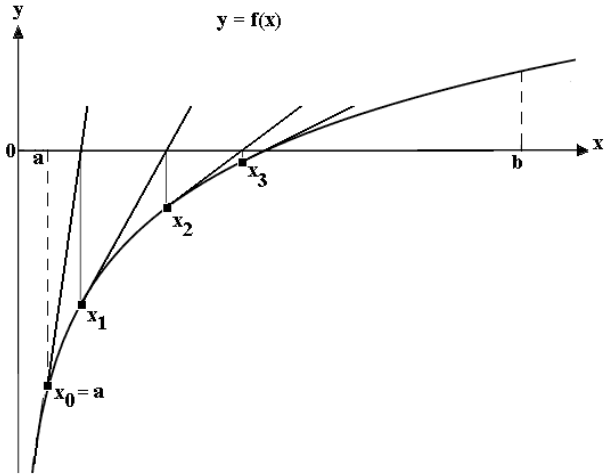
Desítky let se počítače používaly primárně a hlavně k urychlení výpočtů (mnoho lidí dodnes „počítač" považuje za výkonnější „kalkulačku"). Oblast těchto aplikací je stále aktuální. Měli bychom si však uvědomit, že opakované vymýšlení stejných řešení má jen minimální praktický význam. V uplynulých letech se objevila celá řada hotových programů, které dokáží řešit typické matematické problémy (např. hledat řešení soustav rovnic, interpolovat a aproximovat, integrovat a derivovat, provádět symbolické úpravy atd.). Těm, kdo potřebují používat pokročilé matematické funkce, lze tedy doporučit nákup vhodného nástroje, k nimž patří mj. Matlab (*http://www.mathworks.com*), Mathcad (*http://www.mathsoft.com*) nebo Mathematica (*http://www.wolfram.com*). Tyto softwarové balíky, které občas mívají i bezplatné verze, se neomezují jen na automatizaci výpočtů, ale navíc spolupracují s jinými programovacími jazyky a poskytují grafickou prezentaci výsledků.

V této kapitole představíme několik užitečných metod z oblasti numerických algoritmů, které lze potenciálně uplatnit v rámci větších programátorských projektů. Nebudeme příliš zabíhat do matematických důkazů uvedených poznatků, ale pokusíme se ukázat, jakým způsobem je možné numerický algoritmus převést na spustitelný kód jazyka C++. Podoba algoritmů popsaných v této kapitole vychází hlavně z těchto prací: skripta **[Kla87]** dostupného v Polsku a klasického díla **[Knu10]**. Čtenářům by však neměly činit potíže ani jiné příručky, které pojednávají o tématu numerických algoritmů, protože jich v posledních letech vyšlo poměrně hodně. Všechny programy přetištěné v této kapitole je nutné brát jako „výukové" implementace, které poskytují představu o využití jazyka C++ při řešení výpočetních úloh. Méně nároční uživatelé je však mohou převzít jako hotové „kuchařské recepty".

Vyhledávání nulových bodů funkcí

Matematici dosti často stojí před úkolem vyhledat nulové body funkce. Existuje mnoho numerických metod, které umožňují tento úkol vyřešit pomocí počítače. V této kapitole se omezíme na jednu z jednodušších – tzv. *Newtonovu metodu*. Princip této metody zjednodušeně spočívá v systematickém přibližování k nulového bodu pomocí tečen ke křivce, jak je patrné na obrázku 11.1.

Newtonova metoda klade na zkoumanou funkci $y = f(x)$ řadu omezení. V analyzovaném intervalu $[a, b]$ se například nachází právě jeden kořen, funkce na okrajích intervalu nabývá hodnot lišících se znaménkem a její první a druhá derivace v tomto intervalu nemění své znaménko.



Obrázek 11.1: Newtonův algoritmus vyhledávání nulových bodů

Z hlediska programátora lze *Newtonův* algoritmus vyjádřit jako iterativní opakování následujících operací (*i* označuje fázi iterace):

- $z_i = z_{i-1} - \frac{f(z_{i-1})}{f'(z_{i-1})}$
- stop, jestliže $|f(z_i)| < \varepsilon$

Symbol ε označuje určitou kladnou konstantu (např. 0,00001), která zajišťuje, že se algoritmus zastaví. Úplně na začátku samozřejmě inicializujeme proměnnou z_0 jistou počáteční hodnotou. Navíc potřebujeme znát explicitní rovnice *f* a *f'* (funkci a její první derivaci)[1].

Navrhněme rekurzivní[2] verzi algoritmu, která jako parametry přijímá mj. ukazatele na funkce reprezentující *f* a *f'*. Podívejme se na příklad, jak lze pomocí *Newtonovy* metody vypočítat nulové body funkce $f(x) = 3x^2 - 2$. Procedura `nula` přesně odráží schéma, které jsme uvedli na začátku:

newton.cpp

```
const double epsilon=0.0001;
double f(double x)      // funkce f(x)=3x^2-2
{
  return 3*x*x-2;
}
double fp(double x)     // derivace f'(x)=(3x^2-2)'=6x
{
  return 6*x;
}
```

1 Musíme je do kódu programu v C++ zapsat „natvrdo".

2 Všimněme si, že se jedná o příklad tzv. koncové rekurze, kterou lze přirozeným způsobem převést na algoritmicky rovnocennou iterativní verzi.

```
double nula(double x0, double(*f)(double), double(*fp)(double) )
{
  if(fabs( f(x0) ) <epsilon)
    return x0;
  else
    return nula(x0-f(x0)/fp(x0),f,fp);
}
int main()
{
  cout << "Nulový bod funkce 3x*x-2 se rovná "<<nula(1,f,fp)<<endl;
    // výsledek 0,816497
}
```

Díky ukazatelům na funkci je procedura `nula` univerzálnější, ale samozřejmě nic nebrání tomu, abychom tyto funkce používali přímo.

Iterativní výpočet hodnot funkce

S efektivním postupem výpočtu hodnoty mnohočlenů se seznámíme v kapitole 13, kde popíšeme tzv. *Hornerovo schéma*. Nyní se budeme zabývat algoritmem na iterativní výpočet hodnoty funkce, který se sice v praxi nepoužívá příliš často, ale občas může být užitečný.

Předpokládejme, že nás zajímá jistá funkce $y = f(x)$. Převeďme ji do tzv. implicitní formy:

$$F(x, y) = 0.$$

Označme parciální derivaci, která se počítá podle proměnné y, jako $F'_y(x, y)$, kde $F'_y(x, y) \neq 0$ v každé iteraci.

S jistým zjednodušením můžeme pomocí Newtonovy metody vypočítat hodnotu pro určité x iterativním způsobem:

- $y_{n+1} = y_n - \frac{F(x,y_n)}{\frac{\partial F}{\partial y}F(x,y)_{y_n}}$
- stop, jestliže $|y_{n+1} - y_n| < \varepsilon$

Počáteční hodnota y_0 by měla být co nejbližší hledané hodnotě y.

Problém výpočtu hodnoty funkce $f(x)$ jsme v popsané metodě převedli na hledání nulového bodu funkce $F(x, y)$.

Výhody *Newtonovy* metody se projevují zvláště u některých funkcí, jejichž kvocient se může (ale nemusí) značně zjednodušit.

Příklad: Pro $y = 1/x$ máme $F(x, y) = x - 1/y$ a $F'_y(x, y) = 1/y^2$. Z podmínky $F(x, y) = 0$ dostáváme iterativní vzorec $y_{n+1} = 2y_n - x(y_n)^2$.

Výše uvedené vzorce lze vyjádřit následujícím programem v jazyce C++:

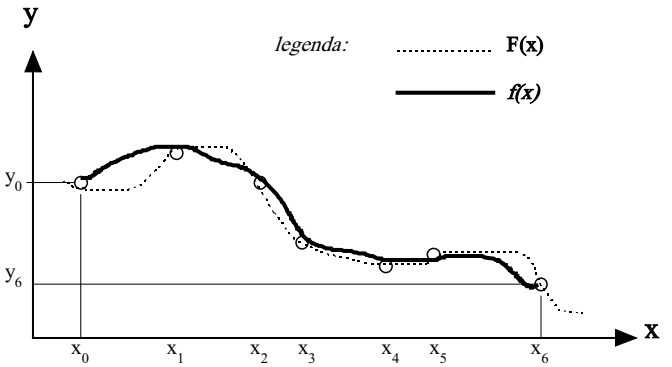
wartf.cpp

```
const double epsilon=0.00000001;

double hodn(double x, double yn)
{
  double yn1=2*yn-x*yn*yn;
  if(fabs(yn-yn1)<epsilon)  // fabs = absolutní hodnota
    return yn1;
```

```
  else
    return hodn(x,yn1);
}
int main()
{
  cout << "Funkce y=1/x má pro x=7 hodnotu "<<hodn(7,0.1);
    // výsledek: 14,2857
}
```

Interpolace funkcí Lagrangeovou metodou

V předchozích částech této kapitoly jsme často využívali explicitní vzorce funkce a její derivace. Jak ovšem postupovat, disponujeme-li částí grafu funkce (tzn. známe její hodnoty pro konečnou množinu argumentů), případě by výpočet funkce podle vzorce byl vzhledem k jeho složitosti velmi časově náročný? V obou případech si můžeme pomoci metodami tzv. *interpolace funkce*. Přitom se k funkci přibližujeme pomocí jednodušší funkce (např. mnohočlenu určeného stupně) tak, aby interpolační funkce procházela známými body grafu původní funkce (viz obrázek 11.2).



Obrázek 11.2: Interpolace funkce f(x) pomocí mnohočlenu F(x)

V příkladu znázorněném na obrázku máme k dispozici 7 dvojic $(x_0, y_0)\ldots (x_6, y_6)$ a na jejich základě dokážeme vypočítat mnohočlen $F(x)$, díky němuž lze hodnoty $f(x)$ určovat mnohem snáze (ačkoli občas mohou být výsledky daleko od pravdy).

Interpolační mnohočlen lze konstruovat pomocí výpočetně náročného *Vandermondova* determinantu, který umožňuje najít koeficienty hledaného mnohočlenu[3]. Pokud nás ovšem zajímá pouze hodnota funkce v určitém bodě z, existuje prostší a efektivnější *Lagrangeova* metoda:

$$F(z) = (z - x_0)(z - x_1)\ldots(z - x_n)\sum_{j=0}^{n}\frac{y_j}{(z - x_j)\prod_{i=0, i\neq j}^{n}(x_j - x_i)}$$

Navzdory dosti odstrašující formě lze výše uvedený vzorec snadno převést na kód jazyka C++, který obsahuje dva vnořené cykly `for`.

3 Viz např. *https://cs.wikipedia.org/wiki/Vandermondova_matice*.

U následujícího výpisu však hrozí riziko dělení nulou (pokud se hodnota *z* rovná některému z uzlů). Na procvičení můžete uvedenou funkci vylepšit a rozšířit ji o kontrolu vypočítaných hodnot a příslušnou signalizaci chyb.

interpol.cpp

```
const int n=3; // stupeň interpolačního mnohočlenu

// pole hodnot funkce (y[i]=f(x[i]))

double x[n+1]={3.0,    5.0,    6.0,    7.0};
double y[n+1]={1.732,  2.236,  2.449,  2.646};
// (ve skutečnosti funkce na výpočet kořene z hodnoty 'x'
double interpol(double z, double x[n], double y[n])
{ // vrácení hodnoty funkce v bodě 'z'
  double wnz=0, om=1, w;
  for(int i=0; i<=n; i++)
  {
    om=om*(z-x[i]);
    w=1.0;
    for(int j=0; j<=n; j++)
      if(i!=j) w=w*(x[i]-x[j]);
        wnz=wnz+y[i]/(w*(z-x[i]));
  }
  return wnz=wnz*om;
}
int main()
{
  double z=4.5;
  cout << "Funkce sqrt(x) má v bodě " << z << " hodnotu "
       << interpol(z,x,y) <<endl;
}
```

Derivování funkcí

V předchozích částech této kapitoly jsme často využívali vzorce funkce a její derivace, které byly přímo zapsány do kódu C++. Výpočet derivace však občas bývá náročný a pracný. Hodí se proto metody, které tento problém dokáží vyřešit a obejdou se přitom bez explicitního vzoru funkce.

Mezi oblíbené metody numerického derivování patří tzv. *Stirlingův* vzorec. Popis jeho odvození přesahuje rámec této publikace. Předvedeme proto pouze praktické výsledky a do matematických důkazů se nebudeme pouštět.

Stirlingův vzorec umožňuje jednoduše vypočítat derivace *f'* a *f''* v bodě x_0 pro jistou funkci $f(x)$, jejíž hodnoty známe v tabulkové formě:

$$\ldots(x_0-2h, f(x_0-2h)), (x_0-h, f(x_0-h)), (x_0, f(x_0)), (x_0+h, f(x_0+h)), (x_0+2h, ,0f(x_0+2h))\ldots$$

Parametr *h* je jistý stálý krok v oboru hodnot *x*.

Stirlingova metoda využívá tzv. *pole centrálních rozdílů*, jehož konstrukce je znázorněna na obrázku 11.3. Rozdíly δ se počítají shodným způsobem v celém poli, např.:

$$\delta f(x_0 - 3/2\ h) = f(x_0 - h) - f(x_0 - 2h) \text{ atd.}$$

Přijmeme-li zjednodušený předpoklad, že budeme vždy počítat derivace pro centrální bod $x = x_0$, nabývají Stirlingovy vzorce následující podoby:

$$f'(x) = \frac{1}{h}\left(\frac{\delta f\left(x-\frac{1}{2}h\right)+\delta f\left(x+\frac{1}{2}h\right)}{2} - \frac{1}{6}\frac{\delta^3 f\left(x-\frac{1}{2}h\right)+\delta^3 f\left(x+\frac{1}{2}h\right)}{2} + \frac{1}{30}\frac{\delta^5 f\left(x-\frac{1}{2}h\right)+\delta^5 f\left(x+\frac{1}{2}h\right)}{2} + \cdots\right)$$

$$f''(x) = \frac{1}{h^2}\left(\delta^2 f(x) + \frac{1}{12}\delta^4 f(x) - \frac{1}{90}\delta^6 f(x) + \cdots\right)$$

x	$f(x)$	$\delta f(x)$	$\delta^2 f(x)$	$\delta^3 f(x)$	$\delta^4 f(x)$
$x_0 - 2h$	$f(x_0 - 2h)$				
	–	$\delta f = (x_0 - 3/2\ h)$			
$x_0 - h$	$f(x_0 - h)$		$\delta^2 f(x_0 - h)$		
		$\delta f = (x_0 - 1/2\ h)$		$\delta^3 f = (x_0 - 1/2\ h)$	
x_0	$f(x_0)$		$\delta^2 f(x_0)$		$\delta^4 f(x_0)$
		$\delta f = (x_0 + 1/2\ h)$		$\delta^3 f = (x_0 + 1/2\ h)$	
$x_0 + h$	$f(x_0 + h)$		$\delta^2 f(x_0 + h)$		
		$\delta f = (x_0 + 3/2\ h)$			
$x_0 + 2h$	$f(x0 + 2h)$				

Obrázek 11.3: Pole centrálních rozdílů v Stirlingově metodě

Kontrolních bodů funkce může být samozřejmě mnohem více než 5. Zde se zaměříme na velmi jednoduchý příklad s pěti hodnotami funkce, který vede k poli centrálních rozdílů nízkého řádu.

Vzorový program v C++, který počítá derivace určité funkce $f(x)$, může vypadat takto:

pochodna.cpp

```
const int n=5; // řád počítaných centrálních rozdílů činí n-1

double t[n][n+1]=
{
  {0.8, 4.80},   // dvojice (x[i], y[i]) pro y=5x*x+2*x
  {0.9, 5.85},   // (zapsané jsou dva první sloupce, a nikoli řádky)
  {1,   7.00},
  {1.1, 8.25},
  {1.2, 9.60}
};

struct DERIVACE{double f1,f2;};

DERIVACE stirling(double t[n][n+1])
// funkce vrací hodnoty f'(z) a f''(z), kde z je centrálním
// prvkem: zde t[2][0]; pole 't' je nutné předtím centrálně
```

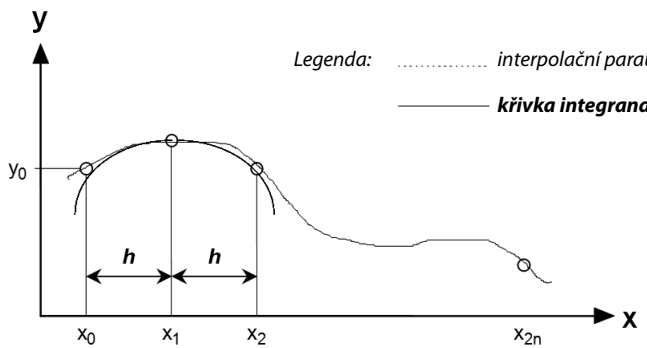
```
// inicializovat, jeho správnost se nekontroluje
{
  DERIVACE res;
  double h=(t[4][0]-t[0][0])/(double)(n-1);  // krok argumentu 'x'
  for(int j=2;j<=n;j++)
    for(int i=0;i<=n-j;i++)
      t[i][j]=t[i+1][j-1]-t[i][j-1];
  res.f1=((t[1][2]+t[2][2])/2.0-(t[0][4]+t[1][4])/12.0)/h;
  res.f2=(t[1][3]-t[0][5]/12.0)/(h*h);
  return res;
}

int main()
{
  DERIVACE res=stirling(t);
  cout << "f'=" << res.f1 << ", f''=" << res.f2 <<endl;
}
```

Když se už zabýváme numerickým derivováním, měli bychom zdůraznit, že je poměrně nepřesné. Čím je hodnota parametru `h` menší, tím větší vliv na výsledek mají chyby zaokrouhlení. Zvětšování parametru `h` však odporuje ideji *Stirlingovy* metody (která má přece napodobovat *skutečné* derivování!). *Stirlingova* metoda se nehodí k derivování na okrajích intervalů proměnlivosti argumentu funkce. Pokud vás toto téma zajímá, můžete najít další informace v příslušné literatuře. Téma je totiž širší, než se může zdát.

Integrování funkcí Simpsonovou metodou

Některé funkce se těžko integrují, protože výpočet jejich integrálu je dosti symbolicky náročný. Abychom dostali požadovaný výsledek, musíme občas provést hodně obtížných transformací (např. substituce, rozklady na posloupnosti atp.).

Můžeme si však pomoci interpolačními metodami (které komplikovanou funkci nahradí přibližnou a výpočetně jednodušší formou). Princip numerického integrování je znázorněn na obrázku 11.4.



Obrázek 11.4: Přibližné integrování funkce