Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Algoritmy numerické matematiky a zpracování dat

Studijní opora

doc. Ing. Jan Cvejn, Ph.D.

Algoritmy numerické matematiky a zpracování dat

Téma 1: Úvod do Jazyka C++

Studijní cíl

Seznámit studenty se vybranými rozšířeními jazyka C++ vůči jazyku C, užitečnými zejména pro zpřehlednění a zefektivnění zápisu kódu.

Doba nutná k nastudování

2 hodiny

Klíčová slova

Jazyk C++, objektově orientované programování, konstruktor, destruktor, kompozice, dědičnost

1 Hlavní rysy jazyka C++

Jazyk C++ vychází z jazyka C, který byl navržen pro tvorbu operačních systémů a je rovněž vhodný k psaní kódu, který pracuje přímo s hardware počítače. C++ tyto vlastnosti zachovává, je však rovněž vhodným profesionálním nástrojem pro tvorbu rozsáhlých programů a knihoven, a je takto stále využíván.

Původně byl jazyk C++ vytvořen jako rozšíření jazyka C o konstrukce, které vychází z moderních trendů v programování, které se objevily v 90. letech 20. století. Tyto nové rysy byly původně implementovány pomocí preprocesoru jazyka C, později se však ukázalo jako nezbytné postavit C++ na samostatných základech. Současná podoba C++ je předepsána normou ISO, která však není definitivní a je dosud stále upravována, i když změny již nejsou zásadní.

Jazyk C++ byl původně rozšířením jazyka C v tom smyslu, že kód v C lze přeložit i jako kód C++. V současné době to již zcela neplatí, ale program v jazyku C lze do C++ snadno převést. Překladače C++ jsou zpravidla schopné současně kompilovat i zdrojové soubory v jazyce C. Překladač rozliší jazyk dle přípony zdrojového souboru - .cpp pro programy v C++, zatímco .c pro programy v C.

Rozsáhlejší programy jsou opět rozděleny na více souborů .cpp, do kterých se vkládají hlavičkové soubory s deklaracemi datových typů, proměnných a hlavičkami funkcí. Nově hlavičkové soubory rovněž obsahují deklarace tříd a šablony.

Rozšíření C++ oproti C je více typů, především:

- neobjektová rozšíření
- objektová rozšíření
- genericita
- zpracování výjimek.

Pravděpodobně nejdůležitějším rozšířením je objektově orientované programování, které přináší i změnu celkové koncepce tvorby programu. Objektově orientované programování však v C++ není povinné a je možné psát program podobným stylem jako v jazyku C.

Genericita je unikátní rys C++, který umožňuje vytvářet kód pro předem neznámé datové typy – lze vytvářet tzv. šablony, ze kterých je vygenerován příslušný kód na místě, kde je datový typ specifikován.

Zpracování výjimek umožňuje ošetřit nestandardní situace v programu jednodušším a přehlednějším způsobem, než pomocí návratových hodnot funkcí.

2 Neobjektová rozšíření C++

Jazyk C++ přináší řadu vylepšení oproti C, která mají kód zjednodušit a zpřehlednit. Vybraná neobjektová rozšíření jsou popsána níže.

Proměnné lze definovat v bloku kdekoliv, nemusí být jen na začátku,
např.

```
for(int i=0; i<10; i++) {…}
```

Pro logické operace je přidán typ bool. Logické proměnné lze přiřadit hodnoty true nebo false, případě hodnotu int (kde nula odpovídá false, jiná hodnota true).

Struktury jsou automaticky datovým typem. V C je nutné před jméno struktury psát klíčové slovo struct, případně použít typedef.

```
struct Vect
{
	float x,y;
};

Vect v;
```

Klíčové slovo const pro označení proměnných, které jsou konstantní. Konstanty typu int lze využít všude, kde se v C používají celočíselné konstanty deklarované pomocí #define

```
const int N=10;

float pole[N];
```

Deklarace const se rovněž používá v hlavičkách funkcí pro označení parametrů, které se v těle funkce nemohou měnit, což je užitečné zejm. u argumentů předaných referencí (viz dále).

Parametry funkcí je možné deklarovat jako tzv. referenční proměnné, které při volání nekopírují hodnotu, ale pointer. Použití referenčních proměnných odpovídá předání argumentů přes pointer, ale v těle funkce se s nimi pracuje stejně jako s běžně předanými parametry.

```
float normV(Vect &v) { return abs(v.x)+abs(v.y);}
```

V C je totéž řešeno pomocí pointeru:

```
float normV(Vect *v) { return abs(v->x)+abs(v->y);}
```

Přetěžování funkcí a operátorů. Lze vytvořit více funkcí stejného jména, které se liší typem nebo počtem parametrů. Překladač přiřadí správnou funkci dle typu parametrů ve volání, např.

```
int sum(int x1, int x2) {return x1+x2;}

float sum(float x1, float x2) {return x1+x2;}
```

Přetížit je možné i význam většiny operátorů, které jsou standardně definované pro základní typy, i pro typy uživatelsky definované, např.

```
Vect operator+(const Vect &v1, const Vect &v2)
{
      Vect v(v1.x+v2.x, v1.y+v2.y);
      return v;
}

Vect v1,v2,v3;
…
v3=v1+v2;
```

Inline funkce. Na místě volání překladač dosadí přímo kód funkce místo standardního volání podprogramu, podobně jako u makra. To je vhodné pro krátké funkce, které mají být vykonány co nejrychleji. Inline funkce se mohou definovat v hlavičkových souborech.

```
inline float pow2(float x) {return x*x;}
```

Na rozdíl od C funkce nemusí vracet žádnou hodnotu. Pak mají návratový typ void.

```
int x=0;
void incX() { x+=1; }
```

Klíčové slovo nullptr. Nulový pointer má v C++ hodnotu nullptr, na rozdíl od C, kde se k těmto účelům využívala konstanta NULL definovaná v <stdlib.h>.

Jazyk C++ má operátory new a delete pro dynamickou alokaci a dealokaci paměti. Tím se liší od jazyka C, kde je pro tyto účely nutné využít knihovní funkce jako malloc() a free().

```
Vect *pvect=new Vect;
…
delete pvect;
```

Pro alokaci a dealokaci pole se používá new[]/delete[]:

```
float *pdata=new float[100];
…
delete[] pvect;
```

Vedle komentářů /* … */ ve stylu C lze používat jednořádkové komentáře, které následují za dvojicí znaků // a jsou ukončeny koncem řádku:

```
int c=a+b;  //toto je součet
```

3 Objektová rozšíření C++

Na rozdíl od jazyka C je C++ objektově orientovaný jazyk, kde je možné s datovými strukturami sdružit funkce, které s nimi manipulují - tzv. metody.

Třída je vytvořena pomocí klíčového slova class nebo struct, kde class má standardně nastavenou ochranu přístupu k členům z vnějšku, zatímco u struct jsou členy volně přístupné. K datovým členům a metodám se přistupuje pomocí . , případně ->, obdobně jako u struktur v C. Ochranu přístupu je ale možné upravit pomocí klíčových slov public, resp. private, psanými s dvojtečkou před seznamem členů v deklaraci.

Každá třída může mít definované 2 speciální metody – konstruktor Třída() a destruktor ~Třída(), které pokud jsou definované, provedou se po vytvoření a před zánikem instance (proměnné dané třídy). Konstruktor může být přetížený – pro inicializaci zadanými parametry. Destruktor je vhodné definovat např., jestliže třída obsahuje dynamicky alokovaná data.

```
struct Vect
{
      float x,y;

      Vect() {x=y=0;}     //konstruktor bez parametrů
      Vect(float px,float py) {set(px,py);}

      void set(float px,float py) {x=px;y=py;}
      float getX() {return x;}
      float getY() {return y;}
};

Vect v;    //vytvoření instance třídy
Vect v2(2,3);  //vyvolá se přetížený konstruktor

v2.set(1,2);    //volání metody pro instanci v2
```

```
Vect v3(v2);   //inicializace copy konstruktorem
```

Pro každou třídu je automaticky vytvořen tzv. copy konstruktor ve tvaru

```
Třída(const Třída&);
```

který umožňuje inicializovat nový objekt jinou existující instancí. Kopie objektu je vytvořena zápisem (oba způsoby jsou ekvivalentní)

```
Třída objekt(objekt2);

Třída objekt=objekt2;
```

Copy konstruktor standardně provede bitovou kopii datových členů, ale jeho chování je možné předefinovat. Třída může obsahovat jako členy objekty jiných tříd. Konstruktor v tomto případě nejprve volá konstruktory jednotlivých členů bez parametrů.

Je rovněž možné vlastnosti třídy rozšířit přidáním dalších datových členů a metod – tzv. odvozování (dědění).

```
struct Vect3: Vect
{
      float z;
      Vect3() {z=0;}
};
```

Konstruktor odvozené třídy nejprve vyvolá konstruktor základní třídy. Destruktory jsou volány v opačném pořadí.

Metody definované v deklaraci třídy jsou překladačem zpracované jako inline funkce, takže mohou být i v hlavičkovém souboru. Složitější metody je nutné definovat mimo tělo třídy v souboru .cpp. V deklaraci třídy se pak uvede jen hlavička metody.

Vlastní metoda má stejnou syntax jako obyčejná funkce, jen v názvu se před jménem použije kvalifikátor třídy v tvaru Třída::

```
struct Vect3: Vect
{
      float z;
      Vect3() {z=0;}
      Vect3(float px,float py,float pz);   //jen hlavička
};

Vect3::Vect3(float px,float py,float pz) //definice metody
{
      set(px,py);
      z=pz;
}
```

4 Standardní vstup a výstup v C++

Standardní knihovna jazyka C++ obsahuje i nové zjednodušené prostředky pro vstup a výstup. Jsou definovány tzv. datové proudy pro zápis na obrazovku, do souborů, popř. do

řetězců, a analogické třídy pro čtení dat. Původní funkce jako printf()/fprintf() a scanf()/fscantf() v <stdio.h> je samozřejmě možné také využít.

Pro zápis na obrazovku je definován datový proud cout třídy ostream, která má přetížený operátor << pro standardní datové typy. Operátor << je přetížený tak, že je možné zřetězit více těchto operací v rámci jediného příkazu.

Jsou rovněž definovány speciální objekty, tzv. manipulátory, které v řetězci proudových operací upravují formátování. Nejdůležitější výstupní manipulátory jsou:

- endl – zapíše konec řádku
- setw(n) – nastaví šířku pole pro výstup reálných čísel
- setprecision(n) - nastaví počet platných cifer pro výstup reálných čísel
- dec, hex, oct – nastaví celočíselný základ.

Příklad:

```
#include <iostream>
using namespace std;

float x=1;
int d=0x2f;

cout << ”x=” << x << endl;
cout << ”d=” << hex << d << endl;
```

Pro zápis/čtení ze souboru je možné vytvořit datový proud třídy oftream, resp. ifstream, kde se v konstruktoru zadává jméno souboru. Destruktor datového proudu soubor automaticky zavře.

```
#include <fstream>
using namespace std;

ofstream fs(”data.txt”);

fs << ”x=” << setprecision(5) << x << endl;
```

Příklady

1. Vytvořte v jazyku C++ třídu pro vektor reálných čísel pevné délky N, kde N je konstanta definovaná na začátku programu. Vytvořte:

    a) konstruktor bez parametru, který vytvoří nulový vektor

    b) konstruktor s 1 parametrem - hodnotou, kterou se budou prvky vektoru inicializovat

    c) metodu pro inicializaci náhodnými hodnotami v intervalu <0,1> pomocí knihovní funkce rand() v <cstdlib>

d) metodu pro výpis složek vektoru do datového proudu typu ostream, který je předán jako argument referencí (je možné např. předat cout)

e) přetížené operátory += a -= pro přičtení/odečtení vektoru a *=, /= pro násobení a dělení reálným číslem.

f) přetížené operátory + a – pro sčítání a odečítání vektorů (vrací vektor jako výsledek)

2. Upravte předchozí program tak, aby se délka vektoru zadávala v konstruktoru a pole pro data bylo dynamicky alokované – v konstruktoru použijte operátor new[], v destruktoru operátor delete[]. U operátorů +=, -=, + a – ošetřete situaci, kdy oba vektory nemají stejnou délku – použijte makro assert() v <assert.h> pro případné zastavení programu.

Použitá literatura

LOUIS, D., MEJZLÍK, P. VIRIUS, M. 1999. *Jazyky C a C++ podle normy ANSI / ISO.* Grada Publishing.

PRESS, H., TEUKOLSKY, S.A., VETTERLING, W. T., FLANNERY, B. P. 2007. *Numerical Recipes. The Art of Scientific Programming. Third Edition.* New York: Cambridge University Press, 2007.

Rejtřík

cout, 6
datový proud, 6
deklarace
  const, 3
destruktor, 4
genericita, 2
Inline funkce, 3
Jazyk C++, 1
komentáře
  jednořádkové, 4
konstruktor, 4
  copy, 5
kvalifikátor třídy, 5
manipulátor, 6
odvozování, 5
ochrana přístupu, 4
operátor <<, 6
operátory
  delete, 4
  new, 4
ostream, 6
parametry funkcí, 3
proměnné
  logické, 2
  referenční, 3
přetěžování
  funkcí, 3
  operátorů, 3
rozšíření C++
  neobjektová, 2
  objektová, 4
struktury, 2
třída, 4

Algoritmy numerické matematiky a zpracování dat

Téma 2: Přesnost a náročnost numerických výpočtů

Studijní cíl

Seznámit studenty s reprezentací reálných čísel v paměti počítače a základními matematickými nástroji pro posouzení přesnosti numerických výpočtů a efektivity algoritmů.

Doba nutná k nastudování

2 hodiny

Klíčová slova

Plovoucí řádová čárka, IEEE 754, normalizovaný formát, strojová přesnost, relativní chyba, chyba aproximace, chyba zaokrouhlení

1 Reprezentace reálných čísel v počítači

Na rozdíl od celých čísel, je reprezentace reálných čísel v počítači poměrně komplikovaná.

Normou IEEE 754 jsou definované standardní formáty, které jsou podporované už na úrovni procesorů, a využívají tzv. plovoucí řádové čárky. Nejčastějšími formáty jsou float (32 bitů) a double (64 bitů). V jazyku C, resp. C++ je navíc k dispozici i formát long double, který dříve většinou využíval 80-bitové reprezentace, ale u kompilátoru Microsoft je v současné době identický s double.

Reálné číslo typu float je buďto ve tvaru (tzv. normalizovaný formát)

$$X = (-1)^S \, 2^{E-\mathrm{bias}} \, (1.F) \tag{1}$$

kde S je znaménkový bit, E je číslo v rozsahu 1 až 254, $\mathrm{bias}=127$ a F je 23-bitový řetězec reprezentující číslo menší než 1 (tj. $(1.F)$ je 24-bitové číslo začínající 1), nebo ve tvaru

$$X = (-1)^S \, 2^{-126} \, (0.F) \tag{2}$$

(tzv. nenormalizovaný formát), která se použije jedině pokud je číslo v absolutní hodnotě menší než 2^{-126}. Číslo $(E-\mathrm{bias})$ je exponent a $(1.F)$, případně $(0.F)$ je tzv. mantisa.

Nula odpovídá $F = 0$ a $E = 0$. Navíc jsou definovány i hodnoty $\pm\mathrm{Inf}$ (nekonečno) odpovídající $F = 0$ a $E = 255$, $S = \pm 1$, a NaN (neplatné číslo), což odpovídá $F \neq 0$ a $E = 255$.

Dekadický rozsah čísel typu float je zhruba $|X| \leq 3,4.10^{38}$ a $|X| \geq 1,8.10^{-38}$ pro $X \neq 0$.

V případě čísel typu double, která se v současné době převážně používají, je E v rozsahu 1 až 2046 v případě normalizovaného formátu, bias je 1023 a F má 52 bitů. Dekadický rozsah je pak zhruba $|X| \leq 1,8.10^{308}$ a $|X| \geq 2,23.10^{-308}$ pro $X \neq 0$.

Minimální absolutní hodnota reprezentovaných čísel ale neodpovídá skutečné přesnosti matematických výpočtů, protože při operacích jako + nebo – je nutné nejprve číslo s nižší hodnotou E převést na hodnotu vyššího čísla, čímž dojde k ztrátě cifer.

Maximální chyba reprezentace čísla u normalizovaného formátu Δ odpovídá poslednímu bitu mantisy, tj. 2^{-52} v případě double, ale je násobena hodnotou exponentu $(E - \mathrm{bias})$. Pro různé hodnoty exponentu se tedy chyba reprezentace liší. Vhodnější je proto pracovat s relativní chybou (v případě typu double)

$$\varepsilon_m = \frac{2^{-52}(E - \mathrm{bias})}{(E - \mathrm{bias})} = 2^{-52} \approx 2,2.10^{-16}. \tag{3}$$

Toto číslo se označuje jako strojová přesnost. V případě typu float je $\varepsilon_m \approx 1,2.10^{-7}$.

Chyba výsledků matematických operací způsobená konečnou přesností reprezentace čísel v počítači se nazývá zaokrouhlovací chyba. V případě reprezentace v plovoucí řádové čárce je přirozené studovat především vývoj relativní zaokrouhlovací chyby.

2 Chyby matematických operací

Při výpočtech postupně dochází ke kumulaci chyb. Vzhledem k povaze reprezentace reálných čísel v počítači je přirozenější uvažovat relativní chyby. Jestliže označíme $\delta_A = \varepsilon_A / A$ a $\delta_B = \varepsilon_B / B$, platí v tomto případě:

$$|\delta_{A+B}| \leq \frac{|A||\delta_A| + |B||\delta_A|}{|A+B|},\ |\delta_{A-B}| \leq \frac{|A||\delta_A| + |B||\delta_A|}{|A-B|}\ . \tag{4}$$

Pro operace násobení a dělení je možné získat vztahy

$$|\delta_{AB}| \leq |\delta_A| + |\delta_B|,\ |\delta_{A/B}| \leq |\delta_A| + |\delta_B|. \tag{5}$$

Jelikož každá operace násobení a dělení zhruba sčítá relativní chyby, celková relativní chyba u těchto operací roste pomalu. Podobné je to i případě sčítání dvou kladných čísel. Celková

maximální relativní chyba výpočtu je úměrná počtu operací. Skutečná chyba je obvykle mnohem menší.

Mnohem horší situace ale nastává v případě rozdílu dvou blízkých kladných hodnot (nebo sčítání kladného a záporného čísla s blízkou absolutní hodnotou), kdy relativní chyba výsledku může být o mnoho řádů vyšší než δ_A a δ_B.

Z tohoto důvodu je vhodné se odečítání blízkých čísel vyhnout, je-li to možné. V některých případech je to možné provést úpravou matematických vztahů.

Problematické je rovněž sčítání a odečítání hodnot s výrazně odlišnou hodnotou exponentu, kde dochází ke ztrátě cifer kvůli převedení na stejný řád. Např. výsledkem operace $A+B$ v počítači je A, jestliže $|B|/|A|<\varepsilon_m$.

Vhodnou organizací výpočtu je někdy možné se takovýmto operacím vyhnout, nebo alespoň zredukovat jejich počet. Např. v případě součtu více členů je výhodné začít od nejmenších členů v absolutní hodnotě a pokračovat směrem k nejvyšším.

3 Chyba aproximace

Vedle zaokrouhlovací chyby se v souvislosti s numerickými výpočty rovněž hovoří o tzv. chybě metody nebo aproximace, která vzniká nahrazením přesného vztahu aproximací – např. náhradou derivace podílem diferencí nebo použitím pouze konečného počtu členů nekonečného rozvoje. Chyba metody nezávisí na reprezentaci reálných čísel v paměti počítače.

Pro vyjádření chyby aproximace se v matematice využívá zápis $O\left(f\left(N\right)\right)$. Jestliže chyba nějaké metody způsobená aproximací je $\varepsilon=O\left(f\left(h\right)\right)$, kde h je parametr ovlivňující přesnost v tom smyslu, že pro $h=0$ je chyba nulová, pak platí

$$\frac{\varepsilon}{f\left(h\right)}\rightarrow K \quad \text{pro } h\rightarrow 0 \tag{6}$$

kde K je nějaké číslo.

Příklad: V případě aproximace funkce lineárním přírůstkem platí

$$f\left(x+h\right)=f\left(x\right)+f'\left(x\right)h+O\left(h^2\right). \tag{7}$$

To znamená, že chyba se blíží Kh^2 pro $h\rightarrow 0$ a klesá tedy rychleji než druhý (lineární) člen.

4 Náročnost matematických výpočtů

Pro posouzení náročnosti výpočtu se obvykle uvažuje počet aritmetických operací v závislosti na parametru – např. N, který odpovídá rozměru úlohy. U většiny počítačů je součin (nebo podíl) dvou čísel v plovoucí řádové čárce časově náročnější operací než součet (nebo rozdíl). Často je ale součin následován součtem, takže součet a přičtení je možné považovat za jednu operaci.

Přesný počet vykonaných matematických operací často není nutné znát, ale spíše je důležitý jeho asymptotický odhad pro velká N. Jestliže např. $p(N)$ označuje skutečný počet operací, pak zápis $p(N)=O(f(N))$ znamená, že pro $N \to \infty$ platí

$$p(N) \le K\, f(N) \tag{8}$$

pro nějaké $K > 0$.

Příklad: Jestliže výpočet vyžaduje $p(n)=1,5n^3+2n^2-6n$ operací, je náročnost výpočtu řádu $O(n^3)$.

Příklady

1. Najděte přibližně číslo ε_m programem tak, že počáteční hodnotu $\varepsilon_m = 1$ násobíte číslem $\alpha < 1$ dokud neplatí $1+\varepsilon_m = 1$.

2. U řešení kvadratické rovnice

$$ax^2+bx+c=0 \tag{9}$$

$$x_{12} = \frac{-b \pm \sqrt{D}}{2a}, \quad D = b^2-4ac \tag{10}$$

dochází k problému s odečítáním blízkých hodnot, jestliže $D > 0$ a jeden z kořenů je blízký 0. Upravte výpočet tak, aby bylo možné se problematickému odečítání vyhnout. Využijte faktu, že kvadratický trojčlen je možné rozložit jako $a(x-x_1)(x-x_2)$ a platí $c/a = x_1 x_2$.

3. Odvoďte vztah pro relativní chybu rozdílu $|\delta_{A-B}|$.

3. Odvoďte vztah pro relativní chybu součinu $|\delta_{AB}|$.

Použitá literatura

HAMMING, R. W. 1973. *Numerical Methods for Scientists and Engineers. 2nd ed*. New York: McGraw-Hill.

PRESS, H., TEUKOLSKY, S.A., VETTERLING, W. T., FLANNERY, B. P. 2007. *Numerical Recipes. The Art of Scientific Programming. Third Edition.* New York: Cambridge University Press, 2007.

RALSTON, A. 1987. *Základy numerické matematiky*. Praha: Academia.

YANG, W. Y., CAO, W., CHUNG T-S., MORFIA, J. 2005. *Applied Numerical Methods Using Matlab®*. John Wiley & Sons Inc.,

Rejstřík

bias, 1
bit
  znaménkový, 1
exponent, 1
formát
  double, 1
  float, 1
  long double, 1
  nenormalizovaný, 1
  normalizovaný, 1
chyba
  aproximace, 3
  maximální, 2
  relativní, 2
  zaokrouhlovací, 2
mantisa, 1
náročnost výpočtů, 4
nekonečno, 1
neplatné číslo, 1
odhad
  asymtotický, 4
plovoucí řádové čárka, 1
reprezentace reálných čísel, 1
rozsah čísel, 2
strojová přesnost, 2

Algoritmy numerické matematiky a zpracování dat

Téma 3: Podpora výpočtů v plovoucí řádové čárce v C/C++

Studijní cíl

Seznámit studenty s podporou pro numerické výpočty v jazycích C/C++.

Doba nutná k nastudování

2 hodiny

Klíčová slova

Jazyk C++, numerické typy, zaokrouhlování, numerické výjimky, matematická knihovna, komplexní aritmetika

1 Vlastnosti numerických typů

Charakteristické vlastnosti numerických typů pro daný systém jsou popsány konstantami v hlavičkovém souboru <cfloat> v C++, resp. <float.h> v jazyku C, zvlášť pro typy float, double a long double.

- `FLT_RADIX`

  - číselný základ pro reprezentaci floating-point čísel
- `FLT_DIG, DBL_DIG, LDBL_DIG`

  - počet desetinných cifer reprezentace
- `FLT_MIN_EXP, DBL_MIN_EXP, LDBL_MIN_EXP`
- `FLT_MAX_EXP, DBL_MAX_EXP, LDBL_MAX_EXP`

  - minimální, resp. maximální hodnota exponentu (o základu 2), která generuje normalizované floating-point číslo
- `FLT_MIN_10_EXP, DBL_MIN_10_EXP, LDBL_MIN_10_EXP`
- `FLT_MAX_10_EXP, DBL_MAX_10_EXP, LDBL_MAX_10_EXP`

  - minimální, resp. maximální hodnota exponentu o základu 10, která generuje normalizované floating-point číslo
- `FLT_MIN, DBL_MIN, LDBL_MIN`

- `FLT_MAX, DBL_MAX, LDBL_MAX`

  - minimální kladné, resp. maximální konečné reprezentovatelné číslo
- `FLT_EPSILON, DBL _EPSILON, LDBL_EPSILON`

  - strojová přesnost reprezentace ε_m
- `FLT_ROUNDS`

  - používaný systém zaokrouhlování, který lze upravit pomocí fsetround() (viz níže).

V hlavičkovém souboru <cfloat>, resp. <float.h> jsou dále funkce a makra pro řízení práce s numerickými typy, zejm.

```
int fesetround(int rdir)
```

pro nastavení způsobu zaokrouhlování:

- `FE_DOWNWARD` – směrem dolů
- `FE_TONEAREST` – směrem k nejbližšímu číslu
- `FE_TOWARDZERO` – směrem k 0
- `FE_UPWARD` – směrem nahoru.

Nastavení ovlivňuje:

- konverze čísel na řetězce
- výsledky aritmetických operací mimo konstantní výrazy
- knihovní funkce zaokrouhlování typu rint(), nearbyint().

Nastavení neovlivňuje:

- standardní konverze na celá čísla jako (int)(3.14159), které se vždy zaokrouhlují směrem k nule
- výsledky zaokrouhlovacích funkcí trunc(), ceil(), floor() a round()
- zaokrouhlování v konstantních výrazech, které je vždy směrem k nejbližší hodnotě.

Během výpočtu je možné testovat, jestli nenastala numerická výjimka pomocí

```
int fetestexcept(int excepts)
```

kde excepts je hodnota získaná kombinací získanou příznaků pomocí | :

- `FE_DIVBYZERO` – dělení nulou
- `FE_INEXACT` – výsledek není přesný
- `FE_INVALID` – hodnota argumentu, pro který funkce není definovaná

- `FE_OVERFLOW` – hodnota je příliš velká, než aby mohla být reprezentovaná
- `FE_UNDERFLOW` – hodnota je příliš malá, než aby mohla být reprezentovaná
- `FE_ALL_EXCEPT` – všechny výjimky.

V programu je možné výjimku vyvolat pomocí funkce

```
int feraiseexcept(int excepts).
```

Příznakové slovo je možné smazat pomocí

```
int feclearexcept(int excepts).
```

Numerické výjimky mohou vyvolat i přerušení programu, což může být při ladění programu užitečné, ale závisí to na prostředí.

Floating-point reprezentace čísel umožňuje zaznamenat i hodnoty, které jsou mimo rozsah nebo jsou jinak neplatné. Program s těmito hodnotami může i částečně pracovat. Jsou definovány konstanty reprezentující speciální hodnoty:

- `INFINITY` – číslo Inf, lze přidat znaménko
- `NAN` – číslo NaN, lze přidat znaménko
- `HUGE_VAL` – číslo vrácené v případě, že rozsah reprezentace neumožňuje výsledek zaznamenat, u Microsoft C/C++ stejné jako INFINITY.

2 Knihovna matematických funkcí C/C++

Deklarace matematických funkcí v C++ jsou v hlavičkovém souboru <cmath> , na rozdíl od knihovny C, kde jsou deklarace v <math.h>.

Obě knihovny obsahují podobné funkce, ale funkce v C++ verzi jsou přetížené pro různé numerické typy – např. fabs(x) vrací hodnotu float, double nebo long double dle typu argumentu. V C je knihovna standardně jen double, ale každá funkce může mít více verzí pro různé numerické typy – např.

- double **cos**(double)
- float **cosf**(float)
- long double **cosl**(long double).

V tab. 1-8 následuje přehled funkcí standardní matematické knihovny, rozdělený podle kategorií, pro numerický typ double:

- trigonometrické a hyperbolické funkce (Tab. 1)

- exponenciální a logaritmické funkce (Tab. 2)
- mocniny a odmocniny (Tab. 3)
- minimum, maximum, apod. (Tab. 4)
- speciální matematické funkce (Tab. 5)
- zaokrouhlovací funkce (Tab. 6)
- manipulace s floating-point reprezentací (Tab. 7)
- klasifikační funkce nebo makra (Tab. 8).

Tab. 1 - Trigonometrické a hyperbolické **funkce**

NÁZEV FUNKCE	VÝZNAM
double **cos**(double x)	$\cos x$
double **sin**(double x)	$\sin x$
double **tan**(double x)	$\tan x$
double **acos**(double x)	$\arccos x$
double **asin**(double x)	$\arcsin x$
double **atan**(double x)	$\arctan x$
double **atan2**(double y, double x)	$\arctan\left(y/x\right)$ v rozsahu $\left(-\pi/\pi\right]$
double **cosh**(double x)	$\cosh x$
double **sinh**(double x)	$\sinh x$
double **tanh**(double x)	$\tanh x$
double **acosh**(double x)	$\text{arccosh}\, x$
double **asinh**(double x)	$\text{arcsinh}\, x$
double **atanh**(double x)	$\text{arctanh}\, x$

Tab. 2 - Exponenciální a logaritmické funkce

NÁZEV FUNKCE	VÝZNAM
double **exp**(double x)	e^x
double **log**(double x)	$\ln x$

double **log10**(double x)	$\log_{10} x$
double **exp2**(double x)	2^x
double **expm1**(double x)	$e^x - 1$, pro malá x přesnější než exp(x)-1
double **ilogb**(double x)	celočíselný logaritmus o základu 2
double **log2**(double x)	$\log_2 x$
double **log1p**(double x)	$\ln(x+1)$, přesnější pro malá x než log(x+1)

Tab. 3 - Mocniny a odmocniny

NÁZEV FUNKCE	VÝZNAM
double **pow**(double x,double y)	x^y
double **sqrt**(double)	$\sqrt{x}$
double **cbrt**(double)	$\sqrt[3]{x}$
double **hypot**(double x,double y)	$\sqrt{x^2 + y^2}$

Tab. 4 - Minimum, maximum, apod.

NÁZEV FUNKCE	VÝZNAM
double **fdim**(double x, double y)	$x - y$, jestliže $x > y$, jinak 0
double **fmax**(double x, double y)	$\max\{x, y\}$
double **fmin**(double x, double y)	$\min\{x, y\}$
double **fabs**(double x)	$\lvert x \rvert$
double **abs**(double x)	$\lvert x \rvert$ jen v C++, přetížené i pro celočíselné typy
double **fma**(double x, double y, double z)	$xy + z$ s minimalizací ztráty přesnosti v mezivýsledku.

Tab. 5 - Speciální matematické funkce

NÁZEV FUNKCE	VÝZNAM
double **erf**(double)	chybová funkce $\operatorname{erf}(z) = \left(2/\sqrt{\pi}\right)\int_0^z e^{-t^2}dt$
double **tgamma**(double)	funkce $\Gamma(x)$
double **lgamma**(double)	$\ln\Gamma(x)$

Tab. 6 - Zaokrouhlovací funkce

NÁZEV FUNKCE	VÝZNAM
double **ceil**(double)	zaokrouhlení nahoru
double **floor**(double)	zaokrouhlení dolů
double **trunc**(double)	zaokrouhlení k nule (odstranění desetinné části)
double **round**(double)	zaokrouhlení k nejbližšímu celému číslu, mezní případy (např. 1.5) směrem od nuly
long long int **llround**(double)	zaokrouhlení k nejbližšímu celému číslu s konverzí na long long int
double **rint**(double)	zaokrouhlení, dle režimu zaokrouhlování nastaveného funkcí fsetround(), generuje výjimku FE_INEXACT
long int **lrint**(double)	jako rint(), s konverzí na long int
double **nearbyint**(double)	jako rint(), ale negeneruje výjimku
double **modf**(double x, double *intpa	rozloží číslo na celočíselnou část a zbytek, obě části mají stejné znaménko jako x
double **fmod**(double n, double d)	zbytek po dělení n/d, tj. n- q*d, kde q=(int)(n/d)
double **remainder**(double n, double d)	vrací zbytek po dělení n/d, tj. n- q*d, kde q je podíl n/d je zaokrouhlený k nejbližšímu celému číslu
double **remquo**(double n, double d, int*q)	jako remainder(), ale navíc v *q vrací podíl

Tab. 7 - Manipulace s floating-point reprezentací

NÁZEV FUNKCE	VÝZNAM
double **frexp**(double x, int*exp)	rozloží číslo na mantisu v rozsahu $[0.5,1)$ a exponent
double **ldexp**(double x, int exp)	sestaví číslo z mantisy a exponentu
double **copysign**(double x, double y)	mění znaménko x dle znaménka y
double **nextafter**(double x, double x)	vrátí následující reprezentovatelné číslo od x ve směru y

Tab. 8 - Klasifikační funkce nebo makra

NÁZEV FUNKCE	VÝZNAM
int **fpclassify**(double x)	vrací typ floating-point čísla: FP_INFINITE – číslo je +/- Inf FP_NAN - číslo je +/- NaN FP_ZERO – číslo je 0 FP_SUBNORMAL – č. nemá normalizovaný formát FP_NORMAL – č. má normalizovaný formát
bool **isfinite**(double x)	test, jestli je číslo konečné
bool **isnan**(double x)	test, jestli je číslo +/-Inf
bool **isnormal**(double x)	test, jestli je číslo normální
bool **signbit**(double x)	vrací znaménko

3 Práce s komplexními čísly v C/C++

Datový typ pro komplexní čísla v jazycích C/C++ přímo není obsažen, ale jsou k dispozici příslušné knihovní funkce. V C jsou deklarované v hlavičkovém souboru <complex.h> a jejich názvy jsou rozlišené dle typu (např. cabs() pro double, cabsf() pro float).

V C++ jsou k dispozici nástroje standardní knihovny pro práci s komplexními čísly, které jsou založené na genericitě. Zápisem

```
#include <complex>

typedef  std::complex<Typ> Cplx;
```

se vytvoří třída s názvem Cplx pro specifikovaný numerický typ Typ, který může být float, double nebo long double. Tato třída má přetížené operátory =,+,-,* a /, ==, !=, které umožňují provádět stejné aritmetické operace jako se základními numerickými typy. Jako druhý argument operátorů lze použít i libovolný numerický výraz – konvertuje se na komplexní číslo s nulovou imaginární složkou.

Pozn.: porovnání <, > definováno není. Jsou ale přetíženy operátory << a >> pro čtení a zápis do datových proudů.

Navíc mají komplexní čísla některé speciální metody jsou pro ně přetížené matematické funkce jako exp(), pow(), sqrt(), log() a trigonometrické a hyperbolické funkce.

Metody komplexních čísel jsou následující:

- **real**()   - reálná složka
- **imag**() - imaginární složka
- **abs**()   - absolutní hodnota
- **arg**()   - fázový posun
- **norm**() - vrací druhou mocninu absolutní hodnoty
- **conj**() - vrací číslo komplexně sdružené
- **polar**() - vytvoří komplex. číslo z absolutní hodnoty a fázového posunu.

Příklad:

```
#include <complex>
#include <iostream>

using namespace std;

typedef complex<double> Cplx;

void main()
{
   Cplx a(1,2);
   a+=1;
   cout << a << ” ” << a.abs() << endl;
}
```

I když komplexní aritmetika je v C++ k dispozici, je vhodné upozornit na některé problémy, které mohou obecně při práci s komplexními čísly nastat.

Např. v případě podílu

$$\frac{a+ib}{c+id}=\frac{ac+bd+i\left(bc-ad\right)}{c^2+d^2} \tag{1}$$

je z důvodu zachování využitelného rozsahu hodnot reálné a imaginární složky vhodné provést úpravu

$$\frac{a+ib}{c+id}=\frac{a+b\left(d/c\right)+i\left(b-a\left(d/c\right)\right)}{c+d\left(d/c\right)} \text{ pro } |c|\ge|d| \tag{2}$$

$$\frac{a+ib}{c+id}=\frac{a\left(c/d\right)+b+i\left(b\left(c/d\right)-a\right)}{c\left(c/d\right)+d} \text{ pro } |c|<|d|. \tag{3}$$

Příklady

1. Vypište parametry floating-point reprezentace v hlavičkovém souboru <cfloat> pro typy float, double a long double, zejm.:

- počet desetinných cifer reprezentace
- minimální a maximální hodnotu exponentu o základu 10
- minimální kladné a maximální konečné reprezentovatelné číslo
- strojovou přesnost reprezentace ε_m.

2. Vypište číslo $\sqrt{2}$ na 3 desetinná místa. Použijte funkci fesetround() v <cfloat> pro nastavení zaokrouhlení nahoru, dolů, k nebližšímu číslu a k 0, a porovnejte výsledky.

3. S využitím komplexní aritmetiky v C++ vytvořte program, který vypíše všechna řešení rovnice $x^n=1$ v komplexním oboru – jsou to hodnoty takové, že $|x|=1$ a $\angle x=2k\pi/n$, kde $k=0,\ldots,n-1$.

4. Upravte vztah pro výpočet absolutní hodnoty komplexního čísla

$$|a+ib|=\sqrt{a^2+b^2} \tag{4}$$

tak, aby nedošlo k zmenšení využitelného rozsahu hodnot reálné a imaginární složky. Ověřte, jestli tuto modifikaci využívá metoda abs() u reprezentace komplexních čísel v C++.

Použitá literatura

PRESS, H., TEUKOLSKY, S.A., VETTERLING, W. T., FLANNERY, B. P. 2007. *Numerical Recipes. The Art of Scientific Programming. Third Edition.* New York: Cambridge University Press, 2007.

Microsoft C runtime library (CRT) reference. [online] https://learn.microsoft.com/en-us/cpp/c-runtime-library.

C++ Standard Library reference (STL). [online] https://learn.microsoft.com/en-us/cpp/standard-library.

Rejstřík

funkce, 2, 3, 4, 5, 6, 7, 8
- exponenciální a logaritmické, 4
- floating-point reprezentace, 7
- klasifikační, 7
- minimum a maximum, 5
- mocniny a odmocniny, 5
- speciální, 6
- trigonometrické, 4
- zaokrouhlovací, 6

hodnota exponentu
- maximální, 1
- minimální, 1

komplexní čísla, 7
- aritmetika, 8
- metody, 8

konstanty
- floating-point, 3

matematická knihovna, 3
numerická výjimka, 2
příznakové slovo, 3
systém zaokrouhlování, 2
vlastnosti numerických typů, 1

Algoritmy numerické matematiky a zpracování dat

Téma 4: Datové struktury

Studijní cíl

Seznámit studenty se základními datovými strukturami a jejich vlastnostmi.

Doba nutná k nastudování

2 hodiny

Klíčová slova

Datový typ, datové struktury, záznam, pole, zřetězený seznam, zásobník, fronta

1 Rozdělení datových typů a algoritmů pro práci s nimi

Programovací jazyky mají sadu základních datových typů a je v nich možné vytvářet datové struktury. Datová struktura se skládá z:

- základních datových typů
- jiných struktur.

Datové struktury lze rozdělit na:

- statické - jsou definovány v programovacím jazyce. V průběhu výpočtu se mění jen obsažená data, nikoliv struktura samotná
- dynamické – mění hodnotu i strukturu, jsou implementovány pomocí základních datových struktur.

Jiné možné rozdělení datových struktur je:

- homogenní – skládají se pouze z dat stejného základního typu
- heterogenní - skládají se z dat různých typů.

Algoritmy, které s daty pracují, lze rozdělit na

- obecné – pracují pouze s elementárními typy nebo využívají pomocné datové struktury
- vážící se konkrétní datovou strukturu.

Datový typ je uspořádaný, jestliže mezi prvky typu je definována binární relace (relační operátor) „$<$” s těmito vlastnostmi:

- pro libovolné prvky $X \neq Y$ platí $X < Y$ nebo $Y < X$
- jestliže $X < Y$ a $Y < Z$, platí $X < Z$ (transitivnost)
- neplatí $X < Y$ a současně $Y < X$ (antisymetričnost)
- neplatí $X < X$ (irreflexivnost).

Je-li definován operátor „$<$”, zbývající relační operátory mezi daty typu
jsou definovány doplněním o operátor ekvivalence („$=$”) .

Nestrukturovaný uspořádaný typ se nazývá skalární typ (skalár).

V jazyku C/C++ jsou uspořádané všechny základní typy – kromě číselných typů je v C++ uspořádaný např. i typ bool, protože se automaticky konvertuje na hodnoty 0 nebo 1. Uspořádané ale mohou být i strukturované typy, jestliže je pro ně definován operátor „$<$”.

Statické datové struktury jsou dvojího typu:

- záznamy
- pole.

2 Záznam

Záznam je heterogenní statická struktura, kde každé složce přísluší pevný offset vůči základní adrese, který je určen už při kompilaci.

V jazycích C/C++ je záznam deklarován pomocí klíčového slova struct. V C++ je však struct současně i třídou, která může mít své metody.

Skutečná délka záznamu bývá zarovnána na násobek délky slova počítače, aby bylo možné záznamy efektivně ukládat do pole. Z výkonnostních důvodů mohou být i členy struktury zarovnány na násobek délky slova (u kompilátorů lze zapnout/vypnout).

Variantní záznamy mají část, která má různou interpretaci v závislosti na některých datových členech. Délka záznamu z hlediska uložení v paměti je ale konstantní. V C/C++ je toto řešeno pomocí unionů.

Skutečnou délku struktury lze v C/C++ vždy zjistit pomocí operátoru sizeof.

3 Pole

Pole je homogenní statická lineární struktura pevné délky s náhodným přístupem. V praxi se využívá i dynamické pole, které může měnit svou velikost, což je ale neefektivní operace.

Typ prvku, který pole tvoří, se nazývá bázový typ. Selektor je příkaz pro výběr prvku pole, který je obvykle ve tvaru pole[i], kde i je výraz, který vrací celočíselné hodnoty, tzv. index.

V C/C++ pole vždy začínají indexem 0.

Bázový typ může být skalárním nebo statickým strukturovaným typem.

Mezi poli A, B stejného typu délky N_A a N_B, s počátečním indexem 0, lze následujícím způsobem definovat uspořádání:

- $A < B$, jestliže existuje index $k < \min\{N_A, N_B\}$ takový, že $A[k] < B[k]$ a současně $A[i] = B[i]$ pro všechna $0 \le i < k$.

Z hlediska výkonnosti je vhodné, aby uložení pole začínalo na adrese, která je dělitelná délkou slova. U některých počítačů v minulosti to bylo nezbytné, protože slovo bylo nejmenší adresovatelnou částí paměti. U procesorů Intel to nutné není, ale je to vhodné z hlediska výkonnosti.

Jestliže prvky mají délku nedělitelnou délkou slova, kompilátor buďto nastaví sizeof na délku dělitelnou (z důvodu zvýšení výkonnosti), nebo nechá přesně, ale přístup je pak pomalejší. To lze obvykle nastavit volbou kompilace.

Adresa prvku pole na základě indexu je vypočítána tzv. mapovací funkcí ve tvaru

$$\mathrm{adr}(i) = \mathrm{base} + \sum_{k=1}^{n} (i - D)B \tag{1}$$

kde base je počátek pole, D je dolní mez pole a B je délka bázového typu v bytech. Pro urychlení operací je proto výhodnější implementovat pole vždy tak, aby začínaly od 0, jak je to v C/C++.

4 Vícerozměrná pole

U vícerozměrných polí je výpočet mapovací funkce složitější, obecně

$$\begin{aligned} &\mathrm{adr}(i_1, i_2, ..., i_n) = \mathrm{base} + \sum_{k=1}^{n} (i_k - D_k)B_k \\ &B_1 = B \\ &B_k = (H_{k-1} - D_{k-1} + 1)B_{k-1} \end{aligned} \tag{2}$$

kde D_k a H_k jsou horní a dolní meze $(n-k+1)$-té dimenze pole. Konstanty v mapovacích funkcích jsou určeny už v době překladu, ale přesto využívání vícerozměrných polí není výhodné.

Může být proto efektivnější dvou a vícerozměrná pole realizovat pomocí jednorozměrného pole dat a tzv. přístupových vektorů, což jsou pole pointerů na počátky stránek v poli dat.

Zejm. sekvenční operace je pak možné realizovat efektivněji přístupem k datům přes pointer bez výpočtu mapovací funkce. Tato reprezentace je navíc obecnější, protože umožňuje, aby datové stránky nebo přístupové vektory neměly stejnou délku.

Na druhé straně, realizace je složitější a paměťově náročnější, zejm. u mnohorozměrných polí. V tomto případě by hierarchicky nejvyšší přístupový vektor měl odpovídat indexu s největším rozsahem.

5 Zřetězené seznamy

Zřetězený seznam je dynamická datová struktura, kde každý prvek má odkaz na sousední prvek. Tato vazba, která je v C/C++ nejčastěji realizována pomocí pointerů, umožňuje procházení seznamu. Hlavním přínosem je efektivní vkládání/odebírání prvků z libovolné pozice bez nutnosti přesunu ostatních uložených dat.

Zřetězené seznamy mohou být jednosměrné nebo obousměrné, případě vícecesté, které umožňují více způsobů procházení.

Obousměrný zřetězený seznam umožňuje procházení oběma směry, umožňuje vložit prvek před a za aktuální prvek a vyjmout aktuální prvek. Je třeba uchovávat pointer na první a poslední prvek.

Pokud je to z hlediska využití možné, je však efektivnější pracovat s jednosměrným zřetězeným seznamem, který ale umožňuje přímo vkládat prvky pouze za aktuální prvek a neumožňuje aktuální prvek odstranit. Tento nedostatek je možné obejít při sekvenčním procházení seznamem – např. při vyhledávání, kdy při průchodu se uchovává pointer na předchozí prvek. Data je nejjednodušší přidávat o odebírat ze začátku seznamu, na který je uchováván pointer.

V C++ je nejpřirozenější realizace pomocí třídy, která dle typu seznamu obsahuje ukazatel na první prvek nebo první a poslední prvek, a počet členů. Každý prvek seznamu je strukturou, která kromě vlastních dat obsahuje i příslušné pointery na sousední prvky. Možná realizace jednosměrného zřetězeného seznamu čísel double v C++ je následující:

```
struct Data   //datový typ
{
      Data *pNext;  //odkaz na následující prvek v seznamu
      double x;

      Data(int px) { x=px; pNext=nullptr; }
};

struct DataList  //třída seznamu
{
      Data *pHead;

      DataList() { pHead=nullptr; }
```

```
        ~DataList() { while(pHead) popHead();}

        void pushHead(Data *pData)  //přidání na začátek seznamu
        {
              pData->pNext=pHead;
              pHead=pData;
        }

        Data *popHead()    //vyjmutí prvního prvku
        {
              Data *p=pHead;
              if(p) pHead=p->pNext;
              return p;
        }
  };
```

Třída DataList má kromě konstruktoru a destruktoru 2 metody: pushHead() pro přidání prvku na začátek seznamu, a popHead() pro vyjmutí prvního prvku ze seznamu. Důležitým rysem této implementace je, že třída DataList neprovádí žádnou správu paměti pro datové členy. O tu se musí postarat program, který třídu DataList využívá. Pokud se do seznamu vloží pointer na lokální proměnnou, která v průběhu provádění programu zanikne, seznam se poškodí. Zřetězené seznamy ale většinou využívají dynamicky alokovanou paměť. V tomto případě, jestliže pointery na prvky seznamu nejsou jinde v programu uloženy, je třeba destruktor upravit tak, aby provedl dealokaci paměti u zbývajících prvků:

```
~DataList()
{
      while(Data *p=popHead()) delete p;
}
```

Příklad využití seznamu je níže.

```
DataList list;

Data *p1=new Data(1);
list.pushHead(p1);

Data *p2=new Data(2);
list.pushHead(p2);

for(Data *p=list.pHead;p;p=p->next) //průchod seznamem
      cout << p->x << endl;
```

Dynamicky alokovaná paměť však nemusí být v některých systémech k dispozici. Úložiště pro data seznamu pak může být realizováno pomocí pole, kde prázdná místa je možné evidovat rovněž zřetězeným seznamem.

6 Zásobník a fronta

Zásobník je struktura, která umožňuje přidání na konec a odebrání prvku z konce (operace typu LIFO– Last In First Out). Zásobník je obvykle implementován pomocí dostatečně velkého pole a pointeru za poslední obsazený prvek.

Fronta umožňuje operaci FIFO (First In First Out) – data se na jeden konec přidávají, z druhého konce se odebírají.

Fronta může být snadno realizována pomocí jednosměrného zřetězeného seznamu, ale je třeba navíc uchovávat pointer na poslední prvek, za který se přidávají nová data. Data se vždy odebírají z počátku seznamu.

Příklady

1. Vytvořte třídu, která má 2 datové členy: int n a double x. Předefinujte operátory < a == pro porovnání objektů této třídy (oba vrací bool). Má platit a<b v případě, kdy buďto a.n< b.n, nebo a.n==b.n a a.x<b.x .

2. Do deklarace třídy zřetězeného seznamu v kap. 5 přidejte datový člen uchovávající počet prvků a metodu pro přidání prvku za aktuální pozici reprezentovanou pointerem.

3. Rozšiřte realizaci zřetězeného seznamu v kap. 5 tak, aby umožňovala obousměrný průchod a odebrání prvku na aktuální pozici (vytvořte obousměrný seznam).

4. Vytvořte frontu bez dynamické alokace paměti uvnitř pole pevné délky.

Použitá literatura

LOUIS, D., MEJZLÍK, P. VIRIUS, M. 1999. *Jazyky C a C++ podle normy ANSI / ISO*. Grada Publishing.

SEDGEWICK, R. 2003. *Algoritmy v C. Části 1-4*. Praha: SoftPress.

WIRTH, N. 1987. *Algoritmy a štruktúry údajov*. Bratislava: Alfa.

Rejstřík

algoritmy
  obecné, 1
  pro konkrétní dat. strukturu, 1
bázový typ, 3
datové struktury
  dynamické, 1
  heterogenní, 1
  homogenní, 1
  statické, 1
datový typ
  uspořádaný, 2
destruktor, 5
dynamicky alokovaná paměť, 5
FIFO, 6
fronta, 6
index, 3
konstruktor, 5
LIFO, 6
mapovací funkce, 3
pointer, 4
pole, 2
relace <, 2
selektor, 3
seznam

jednosměrný, 4
obousměrný, 4
vícecestý, 4
zřetězený, 4
sizeof, 3
skalár, 2
třída, 4
uspořádání, 3
zásobník, 6
záznam, 2

Algoritmy numerické matematiky a zpracování dat

Téma 5: Vyhledávání dat

Studijní cíl

Seznámit studenty s algoritmy vyhledávání dat a základy problematiky organizace dat z hlediska efektivity vyhledávání.

Doba nutná k nastudování

2 hodiny

Klíčová slova

Vyhledávání dat, sekvenční vyhledávání, binární vyhledávání, vyhledávací stromy

1 Sekvenční vyhledávání

K základním algoritmům práce s daty patří vyhledávání. Zpravidla záznamy obsahují datový člen, podle kterého se vyhledává – tzv. klíč. Záznamy se stejnou hodnotou klíče se z hlediska vyhledávání považují za rovnocenné. Tím je mezi daty definována relace „=“ (ekvivalence).

Pro zadanou hodnotu *X* máme v datech najít první prvek, libovolný prvek, případně všechny prvky s touto hodnotou klíče.

Nejjednodušší postup, který nepředpokládá žádnou speciální organizaci dat, je sekvenční vyhledávání. Některé datové struktury jiný způsob vyhledávání ani neumožňují – např. zřetězené seznamy.

Datové struktury, které umožňují pouze sekvenční vyhledávání, jsou však obvykle flexibilní z hlediska přidávání a odebírání prvků – např. v případě pole se nový prvek může přidat na první uvolněné místo.

Při sekvenčním vyhledávání se prochází všemi datovými prvky od začátku do konce, případně se hledání ukončí, jestliže je prvek nalezen. Pokud prvek zadané hodnoty klíče v poli není, je nutné projít všechna data.

U základní realizace sekvenčního prohledávání pole je v každém cyklu nutné testovat 2 podmínky – jestli je na dané pozici prvek s hodnotu *X* a jestli není dosaženo konce pole. Ukazuje se, že druhou podmínku je možné odstranit tzv. technikou zarážky, čímž se prohledávání urychlí. Stačí za konec pole vložit prvek s hodnotou *X*. Hledání se pak vždy

ukončí nalezením prvku s hodnotou X , takže překročení hranice pole v cyklu není třeba testovat. Jestliže nalezený prvek je na pozici zarážky, znamená to, že prvek X v poli není.

V případě, že data jsou seřazená, je možné rozpoznat před dosažením konce pole, že data v poli nejsou. Hledání je pak možné ukončit dříve – v průměru je pak hledání rychlejší o 50%.

V případě, že je známo, že některé hodnoty jsou hledány častěji než jiné, je možné sekvenční hledání zefektivnit tím, že se často hledaná data umístí na začátek struktury, takže jejich vyhledání vyžaduje menší počet kroků.

2 Binární vyhledávání

Jestliže je datový typ uspořádaný, vyhledávání může být mnohem efektivnější. Binární vyhledávání je jednoduché a velmi efektivní, ale vyžaduje přístup k datům struktury pomocí indexu, a data musí být seřazená, tzn. platí

$$A[0] \le A[1] \le \ldots \le A[N-1] \tag{1}$$

kde N je počet prvků pole.

Algoritmus binárního vyhledávání je typicky rekurzivní, ale je možné ho snadno realizovat bez rekurze. Jestliže se prvek X má vyhledat v rozsahu indexů $i_1,\ldots,i_2$, kde $i_2 > i_1$, vypočítá se nejprve $i_s = [\![(i_1+i_2)/2]\!]$, kde $[\![.]\!]$ označuje zokrouhlení směrem k nule (v C/C++ běžná konverze na int). Pokud $X \le A[i_s]$, pokračuje se analogicky v intervalu $[i_1, i_s]$, jinak v intervalu $[i_{s+1}, i_2]$. Jestliže $i_1 = i_2$, hledání končí – jestliže $A[i_1] = X$, je nalezen první prvek v poli danou hodnotou, jinak prvek X v poli není. Počáteční nastavení parametrů je $i_1 = 0$, $i_2 = N-1$. V případě, že pole je seřazené sestupně, jen se v porovnání zamění relační operátory $\le$ a $\ge$.

Vzhledem k tomu, že v každém kroku se interval hledání půlí, algoritmus provede

$$[\![\log_2 N + 1]\!] \approx \log_2 N \tag{2}$$

kroků v nejhorším případě. Pro vysoká N je urychlení oproti sekvenčnímu vyhledávání dramatické - např. pro $N = 10^6$ stačí pouze cca 20 porovnání.

V některých případech je známo, že hledaný prvek se nachází v malém okolí aktuálního prvku. Není pak nutné binárně prohledávat celé pole, ale stačí jen omezit podinterval, ve kterém se data nachází. Jestliže je např. $X \ge A[i_0]$, je možné zkoušet $j = 1,2,4,\ldots$, dokud neplatí $i_0 + j \ge N$ nebo $X < A[i_0 + j]$. Pak se provede binární hledání v intervalu

$$\left[i_0 + \frac{j}{2}, \min\left\{ i_0 + j, N-1 \right\} \right]. \quad (3)$$

Seřazené pole je však obtížné udržovat při odebírání prvků a přidávání nových záznamů. Nejlepší je v tomto případě nové prvky přidávat neseřazené na konec pole, kde se hledání provádí sekvenčně. Při odebírání prvků je nutné v setříděné části pole ponechat prázdná místa. Po určitém počtu přidaných/odebraných záznamů je nutné prázdná místa zaplnit prvky z konce pole a pole znovu seřadit.

3 Vyhledávací stromy

Vyhledávací stromy jsou dynamické struktury, které umožňují rychlé vyhledávání a současně i efektivní vkládání/odebírání prvků. Základním typem je binární vyhledávací strom, kde každý prvek (vrchol) V má 2 odkazy na následující vrcholy $L(V)$ a $R(V)$, přičemž platí:

$$L(V) \leq V \text{ a } V \leq R(V). \quad (4)$$

Odkazy $L(V)$ a $R(V)$ mohou být prázdné. Vrchol stromu V_0, který nemá žádného předchůdce, se nazývá kořen. Vrcholy, které nemají následníky, se nazývají listy. Nejdelší cesta od kořene k listu se nazývá výška stromu h.

Algoritmus vyhledání vrcholu s hodnotou X je následující:

1. Nastav $V \leftarrow V_0$.
2. Jestliže $X = V$, ukonči a vrať V.
3. Jestliže $X < V$:

   Jestliže $L(V)$ není prázdný, $V \leftarrow L(V)$ a pokračuj od bodu 2. Jinak ukonči – vrchol nenalezen.

   Jestliže $X > V$:

   Jestliže $R(V)$ není prázdný, $V \leftarrow R(V)$ a pokračuj od bodu 2. Jinak ukonči – vrchol nenalezen.

Jestliže vrchol s hodnotou X v stromu není, je na poslední pozici $L(V)$ nebo $R(V)$ nalezenou algoritmem výše možné nový vrchol X přímo vložit. Jestliže vrchol V s hodnotou X už v stromu je a má 2 následníky, je možné nový vrchol přidat do jeho pravé větve a zařadit původní vrchol $L(V)$, resp. $P(V)$ za něj, takže vrcholy se stejnou hodnotou zůstanou pohromadě. Tím ale dochází k degeneraci stromu. Vhodnější je proto i duplicitní hodnoty vkládat vždy na volné pozice (do listu).

Vyjmutí vrcholu, který má max. 1 následníka, je zřejmé. V případě, že má vrchol 2 následníky, je na uvolněné místo nutné vložit prvek s maximální hodnotou z levého podstromu nebo

minimální hodnotou z pravého podstromu. Pro nalezení maximálního, resp. minimálního prvku stačí stromem projít vždy větví vpravo, resp. vlevo, až k listu – může mít nejvýše 1 následníka.

Jestliže má každý vnitřní vrchol stromu právě 2 následníky, má strom celkem

$$N = 1 + 2 + 4 + \ldots + 2^h = 2^{h+1} - 1 \tag{5}$$

vrcholů, což znamená, že výška stromu je v tomto případě

$$h = \log_2(N+1) - 1 \approx \log_2 N\,. \tag{6}$$

Vyhledání pak vyžaduje pouze $\log_2 N$ kroků. Na druhé straně, není vyloučený ani případ, kdy u každého vrcholu je obsazena vždy jen jedna větev, takže strom je zdegenerovaný do seznamu. Vyhledání pak vyžaduje až N kroků.

Pro efektivní vyhledávání je tedy třeba, aby strom byl vyvážený, což je splněno, jestliže výška podstromů u každého vnitřního vrcholu se liší maximálně o 1.

Při náhodném přidávání a odebírání prvků se strom obvykle příliš neliší od vyváženého stromu. Pro zajištění efektivity v nejhorším případě je ale nutné rozšířit algoritmy vkládání a odebírání prvků o prvky vyvažování. V případě tzv. AVL stromů je takto zaručena výška menší než cca $1.44\log_2 N$.

Výpis vrcholů podle velikosti, což odpovídá seřazení prvků, je možné realizovat jednoduchým rekurzivním algoritmem, kdy se nejprve provede stejná operace v levém podstromu daného vrcholu. Zde je důležité, aby strom nebyl zdegenerovaný, protože při velkém množství dat by mohlo dojít k přetečení zásobníku.

Příklady

1. Vytvořte strukturu (třídu) uchovávající záznamy ve tvaru (n, x), kde n je typu int a x je typu double, v poli dostatečně velké délky N. Vytvořte metodu pro přidání záznamu na první neobsazenou pozici a sekvenční vyhledání a vypsání všech záznamů s danou hodnotou n. Strukturu vyplňte náhodně generovanými záznamy, kde n má omezený rozsah, např. 1-100 a x je v intervalu $[0,1]$. Využijte knihovní funkce rand() v <cstdlib>.

2. Upravte strukturu v příkladu výše tak, aby byla data uchovávána v pořadí dle velikosti n a využijte binární vyhledávání pro:

    a) vyhledání prvního prvku s hodnotou n

    b) vyhledání všech záznamů s hodnotou n.

3. Pro uložení záznamů (n, x) v příkladech výše použijte binární vyhledávací strom, kde data jsou porovnávána na základě hodnoty n. Vytvořte metody pro:
    a) vyhledání některého vrcholu s danou hodnotou n
    b) výpis všech záznamů s danou hodnotou n
    c) výpis všech vrcholů vzestupně dle velikosti n.

Použitá literatura

PRESS, H., TEUKOLSKY, S.A., VETTERLING, W. T., FLANNERY, B. P. 2007. *Numerical Recipes. The Art of Scientific Programming. Third Edition.* New York: Cambridge University Press, 2007.

SEDGEWICK, R. 2003. *Algoritmy v C. Části 1-4*. Praha: SoftPress.

WIRTH, N. 1987. *Algoritmy a štruktúry údajov*. Bratislava: Alfa.

RYCHLÍK, J. 1993. *Programovací techniky*. České Budějovice: Kopp.

Rejstřík

ekvivalence, 1
index, 2
klíč, 1
kořen, 3
list, 3
relace <=, 2
stromy
    AVL, 4
    vyhledávací, 3
vnitřní vrchol, 4
vyhledání vrcholu, 3
vyhledávání
    binární, 2
    sekvenční, 1
výška stromu, 4
vyvažování, 4

Algoritmy numerické matematiky a zpracování dat

Téma 6: Algoritmy řazení polí

Studijní cíl

Seznámit studenty se základními algoritmy řazení polí.

Doba nutná k nastudování

2 hodiny

Klíčová slova

Řazení polí, stabilita řazení, adaptivní řazení, Bubblesort, Shellsort, Quicksort

1 Rozdělení algoritmů řazení

Algoritmy řazení se rozlišují na vnitřní a vnější, kde vnitřní řazení se týká polí a vnější souborů na externím úložišti. S ohledem na zaměření předmětu zde budou diskutovány pouze algoritmy vnitřního řazení.

Pro seřazené pole platí

$$A[0] \le A[1] \le \ldots \le A[N-1] \quad (1)$$

kde N je počet prvků pole. Sestupné řazení je možné získat stejným algoritmem, jen se použité relační operátory „$\le$“ a „$<$ zamění za „$\ge$“ a „$>$“.

Algoritmus řazení se označuje jako přirozený, jestliže je rychlejší pro částečně uspořádanou posloupnost.

Algoritmus řazení je adaptivní, jestliže posloupnost kroků, které vykoná, se liší v závislosti na vstupních datech. Efektivnější verze algoritmů řazení jsou obvykle adaptivní, ale neadaptivní algoritmy jsou zajímavé z hlediska možné hardwarové implementace (třídicí sítě).

Algoritmus řazení je stabilní, jestliže u záznamů se stejnou hodnotou zachovává jejich pořadí.

U rychlosti řazení se posuzuje:

- počet porovnání $C(N)$
- počet přesunů $M(N)$.

Pro posouzení efektivity se rozlišuje především:

- nejhorší případ
- střední případ.

Operace porovnání a přesunu dat se mohou výrazně lišit z hlediska náročnosti – u rozsáhlých záznamů, kde se porovnává podle jednoduchého klíče, může být přesun pomalejší. V takových případech je výhodné k datům přistupovat přes přístupový vektor (pole pointerů na data), který je možné seřadit efektivněji. Někdy je naopak náročnější porovnání – např. v případě pole dynamicky alokovaných řetězců.

2 Bublinkové řazení (Bubblesort)

Porovnává se vždy aktuální a předchozí prvek pole a jestliže nejsou ve správném pořadí, prohodí se. Po prvním průchodu se na konci pole objeví prvek s maximální hodnotou. Postup se opakuje v rozsahu $1,\ldots,j$, kde $j = N-2, N-1, \ldots 1$.

Počet porovnání je v každém případě

$$C\left(N\right) = (N-1) + (N-2) + \ldots + 1 = \frac{N(N-1)}{2} \quad . \tag{2}$$

Pro počet přesunů platí (1 výměna jsou 3 přesuny)

$$M\left(N\right) \le 3\frac{N(N-1)}{2} . \tag{3}$$

Střední počet přesunů je

$$\bar{M}\left(N\right) = 1.5\frac{N(N-1)}{2} . \tag{4}$$

Index j je možné nastavit tak, aby předcházel pozici první výměny v předchozím kroku. Jestliže žádná výměna neproběhla, je možné zpracování ukončit.

Nejjednodušší a nejméně efektivní algoritmus, $O\left(N^2\right)$ porovnání i přesunů, je přirozený a stabilní.

3 Řazení výběrem maxima/minima

Průchodem polem se najde prvek s maximální hodnotou v rozsahu $0,...,N-1$. Ten se pak vymění s posledním prvkem pole. Postup se opakuje v rozsahu horního indexu $N-2, N-3,...,2$. Obdobně je možné vybírat minima a ukládat je na pozice $0,1,...,N-2$.

V tomto případě je opět

$$C(N) = \frac{N(N-1)}{2}. \tag{5}$$

Počet přesunů dat není větší než $3(N-1)$, ale při průchodu polem se rovněž přepisuje uchovávaný index maximálního prvku. Maximální počet těchto přesunů je

$$(N-1)+(N-2)+..+1 = \frac{N(N-1)}{2} \tag{6}$$

takže platí také $M = O(N^2)$. Ve středním případě je ale M podstatně menší, protože index maximálního prvku se přepisuje jen někdy. Průměrný počet přesunů je $N(\ln N - k)$, kde k je konstanta.

Algoritmus je stabilní, z hlediska počtu porovnání se ale nechová přirozeně – pro nalezení minima/maxima je vždy nutné porovnat všechny prvky v rozsahu. Je vhodný v případech, kdy přesun dat je operací časově náročnější než porovnání.

4 Řazení přímým vkládáním a Shellsort

Prvek na pozici $i = 1,2,...,N-1$ se zařazuje do seřazeného úseku v rozsahu $0,...,i-1$.

Zařazení prvku vyžaduje nejvýše i porovnání, takže platí

$$C(N) \le 1+2+...+(N-1) = \frac{N(N-1)}{2} = \frac{N^2-N}{2}. \tag{7}$$

Pro počty přesunů platí

$$M(N) \le 3+4+...+(N-1)+2 = \frac{N(N-1)}{2} + 2(N-1) = \frac{N^2+3N-4}{2}. \tag{8}$$

Jelikož je úsek $[0, i-1]$ seřazený, pozice vkládaného prvku je většinou nalezena po menším počtu kroků. Střední počet porovnání je

$$\bar{C}(N) = \frac{N^2 + N - 2}{4}. \tag{9}$$

Je rovněž možné využít binární vyhledávání. Celkový počet porovnání je pak zhruba $N(\log_2 N + k)$, kde k je konstanta.

Algoritmus je přirozený a stabilní, je vhodný zejména v situacích, kdy je potřeba zatřídit malý počet prvků do seřazeného pole.

Významným vylepšením metody přímého vkládání je Shellsort, který se snaží pole nejprve částečně seřadit provedením výměn dat na velké vzdálenosti. Přímým vkládáním se řadí podposloupnosti dat s krokem h, který se postupně zmenšuje, až v poslední fázi, kdy $h = 1$, je pole už je ale z velké části seřazené, takže prvky se přesouvají jen o malý počet pozic a je třeba malý počet porovnání.

5 Quicksort

Quicksort je rekurzivní algoritmus řazení využívající faktu, že nejefektivnější jsou výměny na velké vzdálenosti.

Princip algoritmu je následující – seřazení v úseku $[a,b]$ pole:

1. Vybere se libovolný index $k \in [a,b]$ a uloží se $X = A[k]$ (tzv. pivot).
2. Pole se rozdělí na 2 části tak, že:
   - část I obsahuje jen prvky $< X$
   - část II jen prvky $\geq X$.
3. Pro části I a II se rekurzivně postupuje stejným způsobem, jestliže obsahují více než 1 prvek.

Krok 2 se realizuje tak, že se indexy i_1 a i_2 se nastaví na a a b a i_1 se zvyšuje, dokud platí $A[i_1] < X$. Pak se snižuje i_2, dokud platí $A[i_1] \leq X$ a $i_2 \geq 0$. Prvky $A[i_1]$ a $A[i_2]$ se následně prohodí a postup se opakuje, dokud platí $i_2 > i_1$. Rekurzivně se pak pokračuje v intervalech $[a, i_2]$ a $[i_1, b]$.

V případě, že jako pivot zvolíme vždy medián daného úseku pole (tzn. že polovina prvků je větších než X), platí

$$C(N) = N \log_2 N \tag{10}$$

protože celkový počet porovnání ve všech úsecích pole pro stejnou úroveň rekurze je N, a celková hloubka rekurze je $\log_2 N$.

Pro počet přesunů ve středním případě platí

$$\bar{M}(N) = \frac{N}{2}\log_2 N\,. \qquad (11)$$

Tyto výkonnostní ukazatele jsou nejlepší mezi známými algoritmy založenými na porovnání prvků. Hledat medián v každém kroku by ale bylo neefektivní. Spíše se proto volí např. prostřední prvek úseku nebo medián jen ze tří zvolených prvků.

V nejhorším případě je sice počet operací $O(N^2)$, ale tato situace nastane jen tehdy, když zvolíme jako pivot vždy nejmenší nebo největší prvek z dané oblasti. Jestliže pivot volíme náhodně, je pravděpodobnost, že právě tento případ nastane, zanedbatelně malá.

Není ale vhodné volit jako pivot vždy první nebo poslední prvek, protože v praxi je poměrně běžné, že pole je téměř setříděné. V podobných případech by rovněž mohlo dojít k narůstání zásobníku na hodnoty úměrné N. Tomu je ale zabráněno rekurzivním voláním nejprve v kratším úseku pole.

Quicksort je nestabilní a nepřirozený algoritmus. Pro seřazená pole je dokonce méně efektivní. Je vhodný především pro rozsáhlá pole.

Algoritmus je k dispozici v standardní knihovně jazyka C/C++ jako univerzální procedura, v hlavičkovém souboru <stdlib.h>/<cstdlib>:

```
void qsort(void *base,size_t number,size_t width,
        int (*compare)(const void *elem1, const void *elem2);
```

kde význam parametrů je následující:

- base – pointer na první prvek pole (je nutné ho konvertovat na void *)
- number – počet prvků pole
- width - velikost prvku pole v bytech
- compare – pointer na uživatelem vytvořenou funkci, která vrátí výsledek porovnání dvou prvků pole, předaných pointery elem1 a elem2.

Dodaná funkce compare musí vracet hodnotu:

- <0 jestliže *elem1 je menší než *elem2
- =0 jestliže *elem1 je ekvivalentní *elem2
- >0 jestliže *elem1 je větší než *elem2.

Pointery elem1 a elem2 je ve funkci compare nutné konvertovat na příslušný datový typ.

Příklady

1. Vytvořte pole záznamů ve tvaru (n,x) délky N, kde n je typu int v rozsahu např. 1-1000, x je typu double v intervalu $[0,1]$, a $N \approx 10^4 - 10^5$. Strukturu vyplňte náhodně generovanými záznamy s pomocí knihovní funkce rand() v <cstdlib>. Seřaďte pole dle hodnoty n a pro stejná n dle hodnoty x:

   a) algoritmem výběru maxima

   b) bublinkovým tříděním

   c) vkládáním.

   Seřazené pole vypište do souboru.

2. Seřaďte pole z příkladu 1 algoritmem Quicksort:

   a) využijte knihovní implementací algoritmu Quicksort v C/C++

   b) napište vlastní funkci pro seřazení pole algoritmem Quicksort s využitím rekurze.

Použitá literatura

PRESS, H., TEUKOLSKY, S.A., VETTERLING, W. T., FLANNERY, B. P. 2007. *Numerical Recipes. The Art of Scientific Programming. Third Edition.* New York: Cambridge University Press, 2007.

SEDGEWICK, R. 2003. *Algoritmy v C. Části 1-4*. Praha: SoftPress.

WIRTH, N. 1987. *Algoritmy a štruktúry údajov*. Bratislava: Alfa.

Rejstřík

algoritmy řazení
  adaptivní, 1
  přirozené, 1
  stabilní, 1
  vnější, 1
  vnitřní, 1
počet porovnání, 1
počet přesunů, 1
Quicksort, 4
řazení
  bublinkové, 2
  přímým vkládáním, 3
  výběrem maxima/minima, 3
Shellsort, 4
třídicí sítě, 1

Algoritmy numerické matematiky a zpracování dat

Téma 7: Maticové operace

Studijní cíl

Seznámit studenty se základními algoritmy práce s maticemi a jejich realizací v C++.

Doba nutná k nastudování

2 hodiny

Klíčová slova

Matice, reprezentace matic, Gaussova eliminační metoda, inverzní matice, determinant

1 Operace s maticemi

Matice rozměru (m,n) je dvourozměrné pole čísel ve tvaru

$$\mathbf{A} = (a_{ij}) = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \dots & \dots & \dots \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \tag{1}$$

kde $m > 0$ je počet řádků a $n > 0$ počet sloupců. Matice, kde jeden z rozměrů je roven 1, se nazývají vektory.

Pro matice stejných rozměrů je definován součet/rozdíl po prvcích:

$$\mathbf{C} = \mathbf{A} \pm \mathbf{B} \Leftrightarrow c_{ij} = a_{ij} \pm b_{ij} . \tag{2}$$

Čtvercová matice má stejný počet řádků i sloupců, tj. $m = n$. Ostatní matice jsou obdélníkové.

Jednotková čtvercová matice řádu n je matice $\mathbf{I}_n$ taková, že

$$(\mathbf{I}_n)_{ij} = \begin{cases} 0 & \text{pro } i \neq j \\ 1 & \text{pro } i = j \end{cases} \tag{3}$$

tedy

$$\mathbf{I}_n = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (4)$$

Sloupce a řádky jednotkové matice jsou jednotkové vektory ortogonálního souřadného systému v prostoru dimenze n.

Pro každou matici je definováno násobení konstantou – násobí se všechny prvky.

Dále, pro matice $\mathbf{A}$ o rozměrech (m_A, n_A) a $\mathbf{B}$ o rozměrech (m_B, n_B), kde platí $n_A = m_B$, tj. že $\mathbf{A}$ má stejný počet sloupců jako matice $\mathbf{B}$ řádků, tedy, je definován součin $\mathbf{C} = \mathbf{AB}$ zápisem

$$c_{ij} = \sum_{k=1}^{n_A = m_B} a_{ik} a_{kj} \text{ , } i = 1, \dots, m_A \text{, } j = 1, \dots, n_B \text{ .} \quad (5)$$

Rozměry výsledné matice jsou (m_A, n_B). Součin matic není komutativní – obecně neplatí $\mathbf{AB} = \mathbf{BA}$.

Pro matice stejných rozměrů je definován i součin po složkách $\mathbf{C} = \mathbf{A} \odot \mathbf{B}$.

Jestliže $\mathbf{A}$ má rozměry (m, n), má matice transponovaná $\mathbf{A}^T = (\tilde{a}_{ij})$ rozměry (n, m) a platí pro ni $\tilde{a}_{ji} = a_{ij}$, $i = 1, \dots, m$, $j = 1, \dots, n$.

2 Práce s maticemi v C/C++

V C++ může být matice reprezentována více způsoby, zejména:

1) Dvourozměrným statickým polem, např.

```
const int MAX=10;
typedef double[MAX][MAX] Matice;

Matice A;
```

Selektor pro přístup k prvku na pozici (i, j) je A[i][j].

Matice má ale obvykle menší rozměry než MAX – tento parametr spíše nastavuje maximální možné rozměry pro třídu matic, se kterými se v programu pracuje. Tato reprezentace je jednoduchá, ale velmi plýtvá s pamětí.

2) Vektorem ukazatelů na dynamicky alokovaná pole řádků:

```
const int MAX=100;
typedef double *[MAX] Matice;
```

```
void createM(Matice M,int m,int n)
{
      for (int i=0;i<MAX;i++)
            M[i]= i<m ? new double[n] : nullptr;
}

Matice A;
createM(A,m,n);
```

Přístup k prvku na pozici (i,j) je opět zápisem A[i][j]. Tato reprezentace je paměťově výrazně méně náročná a umožňuje rychlejší přístup k položkám. Výhodou je také možnost vyměňovat řádky, což je operace, která se provádí např. při řešení soustav rovnic, pouhou výměnou pointerů. Je ale třeba se postarat o uvolnění alokované paměti:

```
void destroyM(Matice M)
{
      for (int i=0;i<MAX;i++)  delete[] M[i];
}
```

Ve funkci destroyM() není nutné zadávat rozměry matice jako parametry, protože nevyužité položky ve vektoru pointerů na řádky jsou v createM() nastaveny na nullptr. Pro tyto položky operátor delete[] nic neprovede.

3) Jednorozměrným dynamicky alokovaným polem polem a vlastní mapovací funkcí. V C++ je výhodné je využít reprezentace třídou s dynamicky alokovanou pamětí. Pro přístup k datům je výhodné přetížit operátor () tak, aby vracel referenci na prvek na pozici (i,j).

3 Matice v systémech lineárních rovnic

Soustavu lineárních rovnic

$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ &\ldots \\ &a_{n1}x_1 + a_{12}x_2 + \ldots + a_{nn}x_n = b_n \end{aligned} \tag{6}$$

je možné přepsat do tvaru

$$\mathbf{Ax} = \mathbf{b}\,. \tag{7}$$

kde $\mathbf{A}$ je čtvercová matice (n,n) a $\mathbf{x}$, $\mathbf{b}$ jsou sloupcové vektory.

Matice $\overline{\mathbf{A}} = (\mathbf{A},\mathbf{b})$ se nazývá rozšířená matice soustavy, má $n+1$ sloupců.

Řešení soustavy se nezmění výměnou řádků matice $\overline{\mathbf{A}}$, vynásobení řádku konstantou a přičtení libovolného násobku jednoho řádku k jinému (tzv. ekvivalentní úpravy).

Tohoto faktu se využívá při řešení Gaussovou eliminační metodou. Matice $\bar{\mathbf{A}}$ se nejprve převede na horní trojúhelníkovou matici, která má pod hlavní diagonálou nuly, následujícím postupem:

- Přičtením $-a_{i1}/a_{11}$ - násobku prvního řádku k řádkům $i = 2,...,n$ se vynulují prvky a_{i1} pro $i = 2,...,n$.

- V dalších sloupcích se postupuje analogicky, přičítá se $-a_{ik}/a_{kk}$ - násobku k-tého řádku k řádkům $i = k+1,...,n$, kde $k = 1,2,...,n-1$:

  $a_{ik} = 0$, $i = k+1,...,n$

  $a_{ij} \leftarrow a_{ij} - \frac{a_{ik}}{a_{kk}} a_{kj}$, pro $i = k+1,...,n$, $j = k+1,...,n+1$.

Takto vznikne (za předpokladu, že vždy platí $a_{kk} \neq 0$) horní trojúhelníková matice.

Aby prvek a_{kk} nikdy nebyl nulový nebo v absolutní hodnotě velmi malý vzhledem ostatním hodnotám v matici, což by způsobilo sčítání/odčítání čísel výrazně odličných velikostí při řádkových operacích, je vhodné před vynulováním k-tého sloupce najít v tomto sloupci prvek s největší absolutní hodnotou a vyměnit ho s k-tým řádkem, aby se tento prvek dostal na pozici a_{kk} (tzv. pivotování).

Jestliže takový nenulový prvek neexistuje, soustava není jednoznačně řešitelná (buďto nemá řešení, nebo jich má nekonečně mnoho).

Je možné i hledat jako pivot prvek s největší absolutní hodnotou v celé podmatici $\left(a_{ij}\right)$, $i = k,...,n$, $j = k,...,n$, ale přehození sloupců je nutné evidovat, protože odpovídá přehození složek vektoru $\mathbf{x}$.

Po provedení kroků popsaných výše je soustava převedena do tvaru

$$\begin{aligned} c_{11}x_1 + c_{12}x_2 + ... + c_{1n}x_n &= d_1 \\ &... \\ c_{n-1,n-1}x_{n-1} + c_{n-1,n}x_n &= d_{n-1} \\ c_{nn}x_n &= d_n. \end{aligned} \quad (8)$$

Nyní je už možné snadno vyjádřit řešení algoritmem zpětného dosazování:

$$
\begin{aligned}
&x_n = d_n / c_{nn} \\
&x_{n-1} = \left(d_n - c_{n-1,n} x_n\right) / c_{n-1,n-1} \\
&\ldots \\
&x_1 = \left(d_1 - c_{1n} x_n - c_{1,n-1} x_{n-1} - \ldots - c_{11} x_1\right) / c_{11} \ .
\end{aligned}
\tag{9}
$$

První fáze metody vyžaduje zhruba $n^3/3$ operací (součin a rozdíl), zatímco druhá fáze pouze $n^2/2$.

Metodu je možné modifikovat tak, že se vždy přičítá $-a_{ik}/a_{kk}$- násobku k-tého řádku ke všem řádkům kromě k (tzv. Jordanova eliminační metoda). Takto vznikne na pozici $\mathbf{A}$ diagonální matice. Po vydělení k-tého řádku a_{kk}, $k = 1,\ldots,n$, je na pozici vektoru $\mathbf{b}$ přímo řešení. Tento postup je ale pro velká n méně efektivní.

4 Získání inverzní matice a determinantu

Inverzní matice je taková, že pro soustavy rovnic $\mathbf{Ax} = \mathbf{b}$ řešení platí $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. Známe-li $\mathbf{A}^{-1}$, je možné soustavu vyřešit pro libovolnou pravou stranu $\mathbf{b}$ jen vynásobením $\mathbf{A}^{-1}$.

Matici $\mathbf{A}^{-1}$ je možné získat tak, že soustavu vyřešíme pro $\mathbf{b} = \mathbf{e}_i$, $i = 1,\ldots,n$, kde $\mathbf{e}_i$ je *i*-tý jednotkový vektor (*i*-tý sloupec jednotkové matice). Jelikož $\mathbf{A}^{-1}\mathbf{e}_i$ je *i*-tý sloupec $\mathbf{A}^{-1}$, z těchto řešení je možné sestavit $\mathbf{A}^{-1}$. Je rovněž možné využít Jordanovy eliminace pro matici $\left(\mathbf{A},\mathbf{I}_n\right)$. Po dokončení algoritmu je výsledná matice ve tvaru $\left(\mathbf{I}_n,\mathbf{A}^{-1}\right)$.

Vzhledem k tomu, že ekvivalentní úpravy nemají vliv na determinant, pro výpočet determinantu stačí matici $\mathbf{A}$ převést do trojúhelníkového tvaru Gaussovou eliminační metodou. Potom

$$
\det \mathbf{A} = c_{11}\, c_{22} \ldots c_{nn} \ . \tag{10}
$$

Příklady

1. Vytvořte třídu Matice pro reprezentaci matice založenou na dynamicky alokovaném poli s vlastní mapovací funkcí. Paměť alokujte v konstruktoru pomocí new double[], rozměry matice jsou předány jako parametry.

2. Rozšiřte třídu Matice v příkladu 1 o metody a přetížené operátory:

    a) +=, -= pro 1 argument typu double

    b) = pro 1 argument typu matice

    c) metody pro vytvoření nulové a jednotkové matice

d) metody pro výpočet součtu a násobení 2 matic kompatibilních rozměrů předaných jako argumenty referencí

e) metodu pro spojení 2 matic se stejným počtem řádků, předaných jako argumenty referencí, do jediné matice.

3. Pro třídu Matice z příkladu 1 vytvořte metodu, která Gaussovou eliminační metodou matici převede do tvaru s nulovými prvky pod hlavní diagonálou.

4. V návaznosti na řešení kroku 2:

   a) vytvořte metodu pro výpočet determinantu čtvercové matice

   b) vytvořte metodu, která vypočítá řešení soustavy $\mathbf{AX}=\mathbf{B}$, kde matice $\mathbf{A}$ a $\mathbf{B}$ jsou předány jako argumenty referencí a mají stejný počet řádků, matice $\mathbf{A}$ je čtvercová. Výsledkem je matice $\mathbf{X}$ stejných rozměrů jako $\mathbf{B}$.

Použitá literatura

LOUIS, D., MEJZLÍK, P. VIRIUS, M. 1999. *Jazyky C a C++ podle normy ANSI / ISO.* Grada Publishing.

PRESS, H., TEUKOLSKY, S.A., VETTERLING, W. T., FLANNERY, B. P. 2007. *Numerical Recipes. The Art of Scientific Programming. Third Edition.* New York: Cambridge University Press, 2007.

YANG, W. Y., CAO, W., CHUNG T-S., MORFIA, J. 2005. *Applied Numerical Methods Using Matlab®*. John Wiley & Sons Inc.,

Rejstřík

determinant, 6
eliminační metoda
  Gaussova, 4
  Jordanova, 5
matice, 1
  čtvercová, 1
  inverzní, 5
  jednotková, 1
  součet/rozdíl, 1
  součin, 2
  trojúhelníková, 4
matice soustavy
  rozšířená, 4
pivot, 5
reprezentace matice, 2
  polem, 2
  vektorem ukazatelů, 2
vektor, 1

Algoritmy numerické matematiky a zpracování dat

Téma 8: Práce s polynomy

Studijní cíl

Seznámit studenty se základními algoritmy práce s polynomy.

Doba nutná k nastudování

2 hodiny

Klíčová slova

Polynom, součin polynomů, dělení polynomů, de Vièteovy vztahy

1 Reprezentace polynomů

Polynom

$$A(x) = a_n x^n + \ldots + a_1 x + a_0 \tag{1}$$

je přirozeně reprezentován polem koeficientů a celočíselným řádem n. Řád polynomu $A(x)$ bude označován také jako n_A.

Dynamické systémy jsou popsané obrazovými přenosy, což jsou racionální funkce, jejichž čitatelé a jmenovatelé jsou polynomy. Přenos složených systémů je pak možné získat pomocí operací s polynomy.

Např. v případě regulačního obvodu s PID-regulátorem

$$F_{ew}(s) = \frac{1}{1 + \frac{r_2 s^2 + r_1 s + r_0}{s} \frac{B(s)}{A(s)}} = \frac{sA(s)}{sA(s) + \left(r_2 s^2 + r_1 s + r_0\right) B(s)}. \tag{2}$$

kde $B(s)$ a $A(s)$ jsou polynomy.

Budou zde uvažovány pouze polynomy s reálnými koeficienty. V C/C++ jsou pole indexovaná vždy od 0, takže umístění koeficientu v poli odpovídá matematickému zápisu.

V C++ je nejpřirozenější způsob reprezentace pomocí třídy, např.

```
const int MAX=50;

struct Poly
{
	double a[MAX];
	int n;

	Poly() {n=0;}

	Poly(double *pa, int pn)
	{
		n=pn;
		for(int i=0;i<n;i++) a[n-1-i]=pa[i];
	}
};
```

Druhý konstruktor umožňuje inicializovat polynom pomocí pole koeficientů, zapsaného v pořadí odpovídající matematickému zápisu, tj. a_n nejvíce vlevo:

```
double ak[]={1,2,3,0};
Poly p(ak,4);
```

vytvoří polynom $p(x) = x^3 + 2x^2 + 3x$.

Základní operace s polynomy jsou:

- součet a rozdíl 2 dvou polynomů
- násobení a dělení polynomu reálným číslem
- násobení a dělení 2 polynomů
- výpočet hodnoty pro dané x
- získání polynomu z kořenů.

K složitějším operacím, které budou diskutovány samostatně, patří:

- interpolace zadanými body
- výpočet kořenů.

2 Součet a rozdíl dvou polynomů

Pro součet nebo rozdíl $C = A \pm B$ platí

$$\begin{aligned} c_k &= a_k \pm b_k,\ i = 0,\ldots,n_C \\ n_C &= \max\{n_A, n_B\} \end{aligned} \tag{3}$$

kde $a_k = 0$ pro $k > n_A$ a $b_k = 0$ pro $k > n_B$.

Příklad implementace v C++ s využitím podmíněného výrazu:

```
struct Poly
{
…
      void add(const Poly &pa, const Poly &pb)
      {
            n=max(pa.n,pb.n);
            for(int i=0;i<n;i++)
                  a[i]=(i<pa.n? pa[i]:0)+(i<pb.n? pb[i]:0);
      }
};

Poly A,B,C;
…
C.add(A,B);
```

3 Součin polynomů

Pro součin $C = A*B$ platí $n_C = n_A + n_B$ a

$$c_k = \sum_{j=0}^{k} a_j b_{k-j} = \sum_{j=0}^{k} b_j a_{k-j} \tag{4}$$

kde je $a_j = 0$ pro $j > n_A$ a $j < 0$, $b_{k-j} = 0$ pro $k - j > n_B$ a $k - j < 0$.

Násobení polynomu x^k odpovídá posunu o k pozic a vynulování a_j pro $j = 0,...,k-1$. Součin je proto možné jednoduše implementovat jako součet polynomů $b_k x^k A$, $k = 0,...,n_B$:

1. for $j = 0...n_A + n_B$: $c_j \leftarrow 0$.
2. for $k = 0...n_B$:

   for $j = 0...n_A$: $c_{k+j} \leftarrow c_{k+j} + b_k a_j$.

4 Podíl polynomů

Pro podíl dvou polynomů A/B je možné použít podobný algoritmus jako pro dělení dvou čísel. V případě, že $n_A \geq n_B$, výsledkem jsou 2 polynomy – podíl D a zbytek R, pro které $n_R < n_D$ a platí:

$$A = B*D + R. \tag{5}$$

Algoritmus dělení je výrazně zjednodušený, pokud B je ve tvaru $B(x) = x - x_0$. Pak platí rekurzivní vztah:

$$\begin{aligned} d_{k-1} &= a_k + d_k x_0, \quad k = n-1,...,1 \\ d_{n-1} &= a_n \end{aligned} \tag{6}$$

Zbytek je pouze číslo $r = a_0 + d_0 x_0$.

5 Výpočet hodnoty pro dané *x*

Počítat hodnoty mnohočlenu přímo z definičního vztahu

$$A(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_n x + a_0 \tag{7}$$

je nevhodné. Základní možností je od nejnižších mocnin přičítat koeficienty násobené mocninou x, která se v každém kroku zvyšuje o řád:

$$\begin{aligned} A_k &= A_{k-1} + a_k p_k, \quad k = 1,\ldots,n \\ p_k &= p_{k-1} x \end{aligned} \tag{8}$$

kde $A_0 = 0$, $p_0 = 1$ a $A(x) = A_n$. Existuje však efektivnější způsob. Polynom je možné zapsat následovně:

$$A(x) = a_n x^n + \ldots + a_1 x + a_0 = a_0 + x\left(a_1 + x\left(a_2 + \ldots.\right)\right). \tag{9}$$

Tomuto zápisu odpovídá rekurzivní výpočet

$$A_{k-1} = a_{k-1} + x A_k, \quad k = n,\ldots,1 \tag{10}$$

kde $A_n = a_n$ a $A(x) = A_0$, v kterém je menší počet operací oproti předchozímu.

Uvažujme

$$A(x) = (x - x_0) D(x) + r \tag{11}$$

kde $D(x)$ je polynom řádu $n-1$. Hodnotu $A(x_0)$ je proto také možné efektivně vypočítat jako zbytek dělení $A(x)$ a $(x - x_0)$.

6 Získání polynomu z kořenů

Jsou-li $x_1,\ldots,x_n$ kořeny, je odpovídající mnohočlen ve tvaru

$$p(x) = (x - x_1)\ldots(x - x_n). \tag{12}$$

Pro získání koeficientů $p(x)$ je možné opakovaně aplikovat algoritmus pro násobení polynomů. V případě komplexních kořenů jsou dva faktory komplexně sdružené

$$\left(x-i\left(a+ib\right)\right)\left(x+i\left(a+ib\right)\right)=x^2-2ax+\left(a^2+b^2\right) \tag{13}$$

takže je možné násobit celým kvadratickým trojčlenem.

Platí rovněž de Vièteovy vztahy:

$$\left(x-x_1\right)\ldots\left(x-x_k\right)=x^k+a_{k-1}x^{k-1}+\ldots+a_1x+a_0 \tag{14}$$

kde

$$\begin{aligned}
-a_{k-1} &= x_1+x_2+\ldots+x_k \\
a_{k-2} &= x_1x_2+x_1x_3+\ldots+x_{k-1}x_k = \sum_{\substack{i,j=1\\ i<j}}^{k} x_ix_j \\
-a_{k-3} &= x_1x_2x_3+x_1x_2x_4+\ldots+x_{k-2}x_{k-1}x_k = \sum_{\substack{i,j,k=1\\ i<j<k}}^{k} x_ix_jx_k \\
&\ldots \\
\left(-1\right)^k a_0 &= x_1x_2\ldots x_k\,.
\end{aligned} \tag{15}$$

Příklad: Pro polynom s kořeny $x_1=1$, $x_2=-1$, $x_3=-2$ dostáváme

$$\begin{aligned}
&\left(x-1\right)\left(x+1\right)\left(x+2\right)=x^3-\left(-2\right)x^2+(-1-2+2)x-\left(2\right)= \\
&=x^3+2x^2-x-2.
\end{aligned} \tag{16}$$

Příklady

1. Vytvořte třídu pro reprezentaci polynomu v C++ a vytvořte metody pro:

   a) inicializaci polem koeficientů
   b) operátory *= a /= pro násobení a dělení polynomu reálným číslem
   c) metody pro součet a rozdíl 2 dvou polynomů, předaných jako argumenty referencí
   d) metodu pro součin dvou polynomů, předaných jako argumenty referencí.

2. Pro třídu z příkladu 1 vytvořte metodu pro výpočet hodnoty polynomu pro dané *x*.

3. Odvoďte vztah pro dělení polynomu faktorem $x-x_0$.

4. Pro třídu z příkladu 1 vytvořte metodu pro sestrojení polynomu na základě předaného pole reálných, případně i komplexních kořenů.

Použitá literatura

BRONSHTEIN, I. N., SEMENDYAYEV, K.A., MUSIOL ,G. MUEHLIG H. 2007. *Handbook of Mathematics. 5th Edition*. Berlin Heidelberg: Springer-Verlag.

HAMMING, R. W. 1973. *Numerical Methods for Scientists and Engineers. 2nd ed*. New York: McGraw-Hill.

PRESS, H., TEUKOLSKY, S.A., VETTERLING, W. T., FLANNERY, B. P. 2007. *Numerical Recipes. The Art of Scientific Programming. Third Edition.* New York: Cambridge University Press, 2007.

RALSTON, A. 1987. *Základy numerické matematiky*. Praha: Academia.

Rejstřík

de Vièteovy vztahy, 5
kořeny, 5
kvadratický trojčlen, 5
podíl polynomů, 3
polynom, 1
řád polynomu, 1
součet a rozdíl polynomů, 2
součin polynomů, 3
výpočet hodnoty, 4
zbytek, 3, 4
získání polynomu z kořenů, 5

Algoritmy numerické matematiky a zpracování dat

Téma 9: Polynomiální interpolace

Studijní cíl

Seznámit studenty se základy problematiky polynomiální interpolace.

Doba nutná k nastudování

2 hodiny

Klíčová slova

Interpolace, Lagrangeův interpolační polynom, Newtonův interpolační polynom, Čebyševova aproximace

1 Problém interpolace

Problém sestrojení funkce, která prochází zadanou posloupností bodů se nazývá interpolace. Uvažujme zadané body (x_i, y_i), kde $i = 0, \ldots, N$ a polynom

$$A(x) = a_n x^n + \ldots + a_1 x + a_0 \tag{1}$$

kde $n = N$. Pro určení koeficientů $A(x)$ platí soustava rovnic

$$\begin{aligned} a_N x_0^N + \ldots + a_1 x_0 + a_0 &= y_0 \\ &\ldots \\ a_N x_N^N + \ldots + a_1 x_N + a_0 &= y_N \end{aligned} \tag{2}$$

kterou je možné přepsat do tvaru

$$\begin{bmatrix} 1 & x_0 & \ldots & x_0^N \\ 1 & x_1 & \ldots & x_1^N \\ & & \ldots & \\ 1 & x_N & \ldots & x_N^N \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \ldots \\ a_N \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \ldots \\ y_N \end{bmatrix}. \tag{3}$$

Jestliže hodnoty x_i jsou vzájemně různé, má soustava výše teoreticky vždy řešení, ale často je řešitelná obtížně – je špatně podmíněná.

V případě rovnoměrně rozložených hodnot x_i se ale většinou nepracuje s velkým počtem bodů. Polynom řádu N, který prochází $N+1$ zadanými body může mít N lokálních extrémů. To naznačuje tendenci interpolačních polynomů oscilovat. Ukazuje se, že oscilace se zhoršuje s rostoucím řádem N, takže interpolační polynomy vyšších řádů než 5-6 většinou nedávají lepší průběhy než polynomy nižších řádů.

2 Lagrangeův interpolační polynom

Koeficienty a_k je ale možné získat i bez řešení soustavy rovnic. Nejznámější metoda je tzv. Lagrangeův interpolační polynom.

$$L_N(x) = \sum_{k=0}^{N} y_k V_k(x) \tag{4}$$

kde

$$\begin{aligned} V_k(x) &= \frac{(x-x_0)\dots(x-x_{k-1})(x-x_{k+1})\dots(x-x_N)}{(x_k-x_0)\dots(x_k-x_{k-1})(x_k-x_{k+1})\dots(x_k-x_N)} = \\ &= \prod_{\substack{j=0\\ j\neq k}}^{N}(x-x_j) \Bigg/ \prod_{\substack{j=0\\ j\neq k}}^{N}(x_k-x_j) . \end{aligned} \tag{5}$$

Každý člen $V_k(x)$ je polynom řádu $N+1$ a je zvolen tak, že $V_k(x_k)=1$, pro $j \neq k$ je $V_k(x_j)=0$.

Příklad: Polynom má procházet body $(0,0)$, $(1,1)$ a $(2,-1)$. Určete Lagrangeův interpolační polynom. Členy $V_k(x)$ jsou ve tvaru

$$V_0(x) = \frac{(x-1)(x-2)}{(0-1)(0-2)} = \frac{1}{2}(x-1)(x-2) \tag{6}$$

$$V_1(x) = \frac{(x-0)(x-2)}{(1-0)(1-2)} = -x(x-2) \tag{7}$$

$$V_2(x) = \frac{(x-0)(x-1)}{(2-0)(2-1)} = \frac{1}{2}x(x-1) . \tag{8}$$

Tedy

$$L(x) = -x(x-2) - \frac{1}{2}x(x-1) = -1.5x^2 + 2.5x. \tag{9}$$

3 Newtonův interpolační polynom

Vedle Lagrangeova interpolačního polynomu existuje ještě tzv. Newtonův interpolační polynom, který je ve tvaru

$$\begin{aligned} W_N(x) = c_0 + c_1(x-x_0) + c_2(x-x_0)(x-x_1) + \ldots + \\ + c_N(x-x_0)(x-x_1)\ldots(x-x_{N-1}). \end{aligned} \tag{10}$$

Platí

$$W_{k+1}(x) = W_k(x) + c_{k+1}(x-x_0)(x-x_1)\ldots(x-x_k) \tag{11}$$

takže Newtonův interpolační polynom je možné snadno rozšiřovat o nové body. Nově přidaný člen je vždy nulový pro všechny předchozí body. Na druhé straně, určení koeficientů $c_0, \ldots, c_N$ je ale komplikovanější než u Lagrangeova interpolačního polynomu. Platí $c_k = D^k y_0$, kde $D^k y_j$ je definováno rekurzivně:

$$D^{k+1} y_j = \frac{D^k y_{j+1} - D^k y_j}{x_{k+j+1} - x_j}, \; D^0 y_j = y_j. \tag{12}$$

Odtud

$$D^1 y_0 = \frac{y_1 - y_0}{x_1 - x_0}, \; D^1 y_1 = \frac{y_2 - y_1}{x_2 - x_1}, \tag{13}$$

$$D^2 y_0 = \frac{D^1 y_1 - D^1 y_0}{x_2 - x_0}, \ldots . \tag{14}$$

Příklad: Určeme Newtonův interpolační polynom pro body $(0,0)$, $(1,1)$ a $(2,-1)$. Platí

$$D^0 y_0 = 0 \tag{15}$$

$$D^1 y_0 = \frac{1}{1}, \; D^1 y_1 = \frac{-2}{1} \tag{16}$$

$$D^2 y_0 = \frac{-2-1}{2} = -1.5 \tag{17}$$

tedy

$$W_N(x) = 0 + (x-0) - 1.5(x-0)(x-1) = x - 1.5x(x-1) = 2.5x - 1.5x^2. \quad (18)$$

Lagrangeův a Newtonův interpolační polynom jsou identické, jen jsou sestrojeny odlišným způsobem. Interpolační vzorce umožňují vypočítat hodnotu pro zadané x, ale neurčují přímo koeficienty polynomu.

4 Čebyševova aproximace

Polynom je možné použít pro aproximaci funkce, s níž se má shodovat v daném počtu bodů. Polynomiální interpolace, který byl diskutována výše, uvažovala body (x_i, y_i) zadané uživatelem. Polynomy vyšších řádů mají ale v případě rovnoměrně rozmístěných bodů tendenci oscilovat.

Ukazuje se však, že toto chování je do značné míry způsobeno tím, že kromě y_i jsou zadané i x_i. Pokud x_i vhodně zvolíme, je možné s pomocí polynomů dosáhnout podstatně lepších vlastností.

Čebyševovy polynomy jsou definované vztahem

$$T_n(x) = \cos(n \arccos x) \quad (19)$$

kde $x \in [-1,1]$. Pro obecný interval $t \in [a,b]$ je možné provést transformaci pomocí

$$t = \frac{1}{2}\left((b-a)x + (b+a)\right). \quad (20)$$

Jelikož $\cos(n\theta)$ je polynom argumentu $\cos\theta$, pro $\theta = \arccos x$ je $T_n(x)$ polynomem x.

Interval $x \in [-1,1]$ odpovídá $\theta \in [0,\pi]$. Jelikož

$$\cos((n+1)\theta) + \cos((n-1)\theta) = 2\cos\theta \cos n\theta \quad (21)$$

platí

$$T_{n+1}(x) - 2xT_n(x) + T_{n-1}(x) = 0. \quad (22)$$

Výpis Čebyševových polynomů do řádu $n = 5$ je v Tab. 1. V tzv. uzlových bodech

$$x_k = \cos\left(\frac{(2k+1)\pi}{2N}\right) = \cos\left(\frac{(k+0.5)\pi}{N}\right), \quad k = 0,1,\ldots,N-1 \quad (23)$$

platí $T_N(x_k) = 0$.

Tab. 1 - Čebyševovy polynomy.

n	$T_n(x)$
0	1
1	x
2	$2x^2-1$
3	$4x^3-8x^2+1$
4	$8x^4-8x^2+1$
5	$16x^5-20x^3+5x$

Jestliže se uvažuje aproximace funkce ve tvaru

$$\tilde{f}(x)=\sum_{j=0}^{N-1} c_j T_j(x) \tag{24}$$

kde má platit $\tilde{f}(x)=f(x)$ pro $x=x_k$, pro koeficienty c_j je možné získat vztahy

$$\begin{aligned} c_0 &= \frac{1}{N}\sum_{k=0}^{N-1} f(x_k) \\ c_j &= \frac{2}{N}\sum_{k=0}^{N-1} f(x_k) T_j(x_k), \quad j=1,..,N-1. \end{aligned} \tag{25}$$

Při aproximaci funkce $f(x)$ rozvojem (24) je možné uvažovat, že chyba aproximace přibližně odpovídá dalšímu členu v rozvoji, tj. $c_N T_N(x)$. Vzhledem k tomu, že $|T_N(x)|$ má stejná maxima v intervalu $[-1,1]$, je chyba rozptýlena rovnoměrně. Proto získaný polynom obvykle nemá tendenci se mezi uzlovými body výrazně odchylovat a je možné ho využít jako vhodnou náhradu funkce v daném intervalu.

Příklady

1. Rozšiřte třídu pro práci s polynomy o metodu pro interpolaci polynomu polem zadaných bodů ve tvaru $[x_i, y_i]$, $i=0,...,N-1$. Hodnoty polynomu se zvoleným krokem vypište do textového souboru a data zobrazte v podobě grafu s pomocí vhodného software. Použijte:

    a) Lagrangeův interpolační polynom

b) Newtonův interpolační polynom

2. Vypište do textového souboru hodnoty Čebyševových polynomů do řádu 6 se zvoleným krokem a data zobrazte v podobě grafu s pomocí vhodného software.

3. Pro funkci $f(x)=e^{-t}(\sin t)^2$ v intervalu $[0,\pi]$ získejte aproximaci pomocí Čebyševových polynomů. Hodnoty interpolačního polynomu a původní funkce se zvoleným krokem vypište do textového souboru a data zobrazte v podobě grafu s pomocí vhodného software. Porovnejte výsledky s klasickou interpolací jako v příkladu 1, tj. s pevným krokem $\pi/(N-1)$, kde N je počet bodů.

Použitá literatura

BRONSHTEIN, I. N., SEMENDYAYEV, K.A., MUSIOL ,G. MUEHLIG H. 2007. *Handbook of Mathematics. 5th Edition*. Berlin Heidelberg: Springer-Verlag.

HAMMING, R. W. 1973. *Numerical Methods for Scientists and Engineers. 2nd ed*. New York: McGraw-Hill.

PRESS, H., TEUKOLSKY, S.A., VETTERLING, W. T., FLANNERY, B. P. 2007. *Numerical Recipes. The Art of Scientific Programming. Third Edition.* New York: Cambridge University Press, 2007.

YANG, W. Y., CAO, W., CHUNG T-S., MORFIA, J. 2005. *Applied Numerical Methods Using Matlab®*. John Wiley & Sons Inc.,

Rejstřík

Čebyševova aproximace, 4
interpolace, 1
interpolační polynom
  Lagrangeův, 2
  Newtonův, 3
polynomy
  Čebyševovy, 4
uzlové body, 5

Algoritmy numerické matematiky a zpracování dat

Téma 10: Numerické řešení nelineárních rovnic

Studijní cíl

Seznámit studenty se základními metodami numerického řešení nelineárních rovnic.

Doba nutná k nastudování

2 hodiny

Klíčová slova

Numerické řešení rovnic, metoda půlení intervalu, metoda regula-falsi, Newtonova-Raphsonova metoda, metoda sečen

1 Vymezení oblasti řešení

Pro numerické řešení jsou rovnice většinou uvažovány ve tvaru

$$f(x) = 0 \tag{1}$$

kde $f(x)$ je funkce alespoň spojitá, v některých případech se předpokládá spojitá diferencovatelnost, případně vyšších řádů. Řešení rovnice výše nazýváme kořenem.

U některých metod je nutné nejprve vymezit intervaly, ve kterém se kořeny nachází. Jestliže je intervalů s kořeny více, pak je v každém vymezeném intervalu $[l_i, h_i]$ je provedeno iterační hledání. U ostatních metod je vždy vhodné výpočet startovat z bodu dostatečně blízko kořene, jinak iterační výpočet může selhat.

Intervaly $[l_i, h_i]$ mohou být vymezeny na základě různých znamének $f(l_i)$ a $f(h_i)$, případně podmínky $f(l_i)f(h_i) < 0$. Ze spojitosti funkce pak vyplývá, že v intervalu $[l_i, h_i]$ musí $f(x)$ procházet nulou, takže $[l_i, h_i]$ obsahuje alespoň jeden kořen.

Situace je však komplikovanější v případě násobných kořenů. Např. jestliže x je kořenem $f(x)$, je současně kořenem funkce $f_2(x) = f^2(x)$, která je ale nezáporná, takže interval $[l, h]$, kde $f(l)f(h) < 0$, nelze vymezit. V takových případech se ale v intervalu $[l, h]$ mění

znaménko $f'(x)$. Za předpokladu, že funkci $f(x)$ je možné analyticky derivovat, je proto jednou z možností hledat navíc rovněž body, kde $f'(x)=0$ a z nich vybrat ty, kde $f(x)=0$.

2 Metoda půlení intervalu

Jestliže lze vymezit je vymezte počáteční interval $[l,h]$ tak, aby $f(l)f(h)<0$, je možné řešení vždy nalézt metodou půlení intervalu:

1. Vypočítej $c=(l+h)/2$
2. Jestliže $f(l)f(c)<0$, $h\leftarrow c$, jinak $l\leftarrow c$.
3. Jestliže $h-l\le\varepsilon$, kde ε je zvolené číslo, ukonči. Jinak pokračuj od bodu 1.

Výhodou této metody je, že zaručeně nalezne jeden kořen ve výchozím intervalu $[l,h]$, i když je zde kořenů více. Vzhledem k tomu, že v každém kroku se interval hledání půlí, na konci výpočtu platí

$$h_n-l_n=\frac{h_0-l_0}{2^n}\approx\varepsilon \tag{2}$$

kde n je počet kroků, takže

$$n\approx\log_2\left(\frac{h-l}{\varepsilon}\right). \tag{3}$$

V každé iteraci je c odhadem kořene. Takže pro řešení na okraji intervalu $[l,h]$ platí pro chybu odhadu $e_k=x-c$

$$|e_{k+1}|/|e_k|=0.5\,. \tag{4}$$

Posloupnost $\{|e_k|\}$ klesá k nule stejně rychle jako geometrická řada, což se označuje jako lineární řád konvergence.

3 Metoda regula-falsi

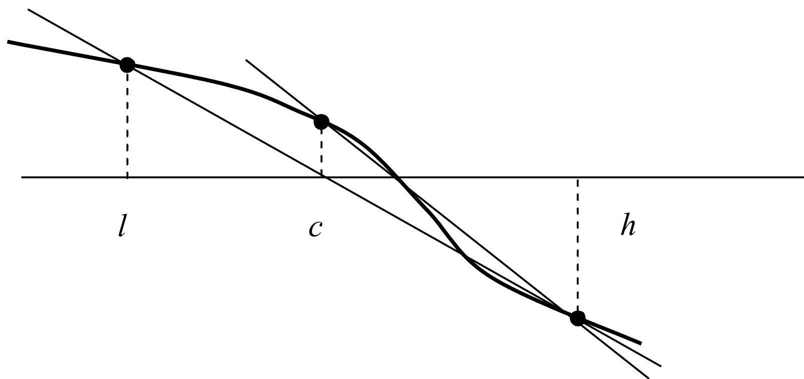
Opět se předpokládá, že je vymezen počáteční interval $[l,h]$ tak, aby $f(l)f(h)<0$. Pokud se funkce $f(x)$ uvažuje v okolí kořene přibližně lineární, je možné dobrý odhad kořene c

v každém kroku získat jako průsečík úsečky proložené body $(l, f(l))$ a $(h, f(h))$ s osou *x* (Obr. 1). Platí

$$f(l) + (c - l)\frac{f(h) - f(l)}{h - l} = 0. \tag{5}$$

Odtud

$$c = l + \frac{f(l)(h - l)}{f(h) - f(l)} = \frac{hf(l) - lf(h)}{f(h) - f(l)}. \tag{6}$$



Obr. 1 - Metoda Regula-falsi

Další postup je stejný jako u půlení intervalu – na základě znaménka $f(l)f(c)$ se pokračuje buď v intervalu $[l, c]$, nebo $[c, h]$.

Metoda je většinou efektivnější než půlení intervalu, ale jelikož nedělí interval symetricky, může postupovat i podstatně pomaleji pro některé typy funkcí.

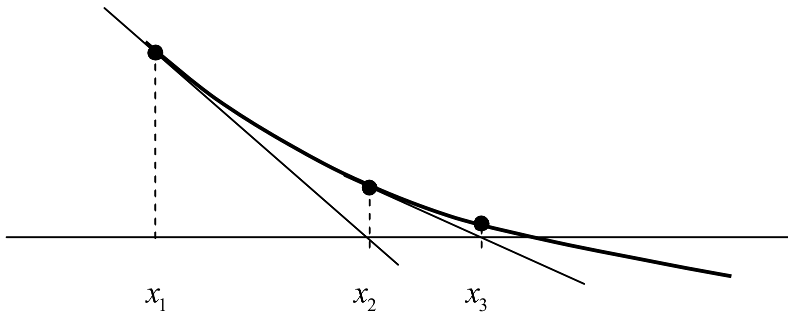
4 Newtonova-Raphsonova metoda

Jestliže nahradíme funkci $f(x)$ v okolí bodu x_k lineárním modelem

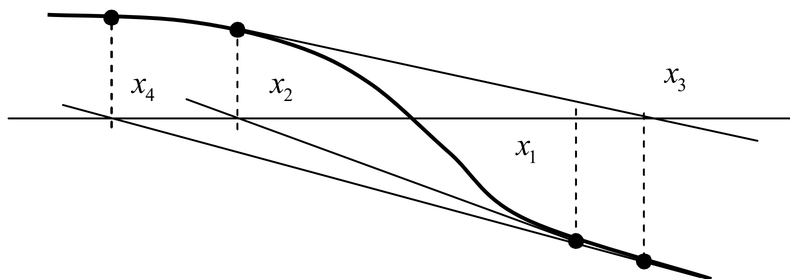
$$f(x) \approx f(x_k) + f'(x_k)(x - x_k) \tag{7}$$

je možné nový bod x_{k+1} určit tak, aby byl model nulový pro $x = x_{k+1}$ (Obr. 2). Odtud dostáváme

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}. \tag{8}$$



Obr. 2 - Newtonova-Raphsonova metoda



Obr. 3 - Problém s konvergencí Newtonovy-Raphsonovy metody

Newtonova-Raphsonova nevyžaduje počáteční vymezení intervalu a je velmi efektivní, jestliže výchozí bod je zvolen dostatečně blízko řešení. V opačném případě však nemusí konvergovat. Metoda často nekonverguje v okolí inflexních bodů (Obr. 3). Lokální minima způsobí chaotické chování nebo nepřiměřeně velký krok mimo.

Základní modifikací je omezení délky kroku: Jestliže $\left|f\left(x_{k+1}\right)\right|>\left|f\left(x_k\right)\right|$, je vhodné krok zkrátit, tedy zvolit

$$x_{k+1}=x_k-\alpha\frac{f\left(x_k\right)}{f'\left(x_k\right)} \tag{9}$$

kde $\alpha\in\left(0,1\right)$, aby platilo $\left|f\left(x_{k+1}\right)\right|<\left|f\left(x_k\right)\right|$. Pokles $\left|f\left(x\right)\right|$ pro dostatečně krátký krok je zaručen, protože

$$f\left(x_k\right)+f'\left(x_k\right)\left(x_{k+1}-x_k\right)=f\left(x_k\right)-\alpha f\left(x_k\right)=\left(1-\alpha\right)f\left(x_k\right). \tag{10}$$

Důvodem efektivity Newtonovy-Raphsonovy metody je kvadratický řád konvergence, což znamená, že

$$\left|e_{k+1}\right| / \left|e_k\right|^2 \to K \text{ pro } k \to \infty \tag{11}$$

kde K je konstanta. V případě, že $f'(x)$ není k dispozici v analytickém tvaru, je možné ji počítat numericky, např. pomocí

$$f'(x_k) \approx \frac{f(x_k + h) - f(x_k - h)}{2h} \tag{12}$$

kde $h > 0$ je malé číslo. Metodu je možné použít i pro hledání komplexních kořenů polynomů, pokud se použije komplexní aritmetika.

5 Metoda sečen a Müllerova metoda

Jinou možností, jak odstranit problém s výpočtem derivací u Newtonovy-Raphsonovy metody, je nahradit $f'(x_k)$ pomocí diferencí z předchozího kroku:

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}. \tag{13}$$

Metoda ale nemusí konvergovat. Zkrácení délky kroku jako u Newtonovy-Raphsonovy metody obecně nezaručuje pokles $\left|f(x)\right|$.

Vylepšení metody sečen je Müllerova metoda, která pracuje s posledními třemi body a interpoluje jimi parabolu, u které se vypočítá bližší kořen. Výhodou Müllerovy metody je, že umožňuje pracovat s komplexními kořeny.

6 Výpočet všech kořenů polynomu

Jedná se o komplexní problém, který zde nemůže být diskutován v plné šíři.

V případě polynomu je počet kořenů znám. Základním přístupem je redukce polynomu po získání kořene, který je možné nalézt některou z obecných metod výše.

Je-li

$$P_k(x) = p_{k,n}x^n + \ldots + p_{k,1}x + p_{k,0} \tag{14}$$

polynom v k-tém kroku a x_k jeho jeden kořen, je dělením faktorem $(x - x_k)$ získán polynom $P_{k-1}(x)$, který je řádu $k-1$. Postup je pak možné opakovat, dokud nejsou získány všechny reálné kořenové činitele.

V případě dvojice komplexních kořenů jsou dva faktory komplexně sdružené

$$\left(x-i\left(a+ib\right)\right)\left(x+i\left(a+ib\right)\right)=x^2-2ax+\left(a^2+b^2\right) \tag{15}$$

takže je opět vhodné dělením odebrat celý kvadratický trojčlen.

Vzhledem k tomu, že u redukce polynomu postupně narůstá vliv numerických chyb, je výhodné tímto způsobem spíše získat pouze odhady pozice kořenů, které se pak zpřesňují Newtonovou-Raphsonovou metodou. U komplexních kořenů je možné použít Bairstowovu metodu, která je založená na stejném principu, ale pracuje přímo s kvadratickým trojčlenem.

Příklady

1. Najděte výchozí interval pro numerické řešení rovnice $2\,\mathrm{arctg}\left(x\right)+x=\pi$ a porovnejte rychlost konvergence k řešení:
   a) metodou půlení intervalu
   b) metodou regula-falsi
   c) Newtonovou metodou (pro počáteční bod $x_0=0$).
2. Odvoďte vztah pro výpočet x_{k+1} Müllerovou metodou pomocí Lagrangeovy interpolace.

Použitá literatura

BRONSHTEIN, I. N., SEMENDYAYEV, K.A., MUSIOL ,G. MUEHLIG H. 2007. *Handbook of Mathematics. 5th Edition*. Berlin Heidelberg: Springer-Verlag.

HAMMING, R. W. 1973. *Numerical Methods for Scientists and Engineers. 2nd ed*. New York: McGraw-Hill.

PRESS, H., TEUKOLSKY, S.A., VETTERLING, W. T., FLANNERY, B. P. 2007. *Numerical Recipes. The Art of Scientific Programming. Third Edition.* New York: Cambridge University Press, 2007.

YANG, W. Y., CAO, W., CHUNG T-S., MORFIA, J. 2005. *Applied Numerical Methods Using Matlab®*. John Wiley & Sons Inc.,

Rejstřík

metoda
  Müllerova, 5
  Newtonova-Raphsonova, 4
  prostá iterační, 2
  půlení intervalu, 2
  regula falsi, 3

sečen, 5
násobné kořeny, 1
vymezení oblasti řešení, 1
výpočet všech kořenů polynomu, 6

Algoritmy numerické matematiky a zpracování dat

Téma 11: Numerická derivace a integrace

Studijní cíl

Seznámit studenty s vybranými metodami numerického derivování a integrování.

Doba nutná k nastudování

2 hodiny

Klíčová slova

Numerická derivace, numerická integrace, kompozitní vzorce, Gaussova integrace, Legendreovy polynomy

1 Numerická derivace

Derivaci spojitě diferencovatelné funce je možné počítat numericky pomocí diferencí. Nejjednodušší varianta je

$$f'(x) \approx \frac{f(x+h)-f(x)}{h} \tag{1}$$

kde *h* je malé číslo. Vztah odpovídá náhradě funkce lineárním přírůstkem, což naznačuje, že pro dostatečně přesný výsledek musí být krok *h* velmi malý. Na druhé straně je však v čitateli rozdíl dvou velmi blízkých čísel, což je operace, která způsobuje velkou relativní chybu. Proto není možné volit *h* libovolně malé.

Chyba způsobená aproximací $f(x+h)$ je ve tvaru $\alpha h^2/2$, kde α odpovídá hodnotě druhé derivace v uvažovaném rozsahu. Jestliže chyba výpočtu $f(x)$ a $f(x+h)$ vlivem zaokrouhlení je $L\varepsilon_m$, kde ε_m je strojová přesnost a L odpovídá rozsahu hodnot $f(x)$, potom je celková chyba

$$\varepsilon \approx \frac{\alpha h}{2} + \frac{2L\varepsilon_m}{h}. \tag{2}$$

Je možné vypočítat i hodnotu h která chybu minimalizuje. Výsledek ukazuje, že optimální krok h je úměrný $h_{opt} = \sqrt{\varepsilon_m}$.

Výhodnější je derivaci počítat pomocí centrálních diferencí, který vychází z Taylorova rozvoje 2. řádu:

$$f'(x) \approx \frac{f(x+h)-f(x-h)}{2h}. \qquad (3)$$

V tomto případě je optimální volba je úměrná $\sqrt[3]{\varepsilon_m}$.

2 Rozdělení metod numerické integrace

Metody numerické integrace v intervalu $[a,b]$ pracují se sítí bodů x_k, kde $x_0 = a$, $x_N = b$ a

$$x_0 < x_1 < \ldots < x_N . \qquad (4)$$

V bodech x_k jsou vypočítány funkční hodnoty. Hodnota integrálu je získána v obecném tvaru

$$\int_a^b f(x)\,dx \approx \sum_{j=0}^{N} w_j f(x_j). \qquad (5)$$

Jestliže výpočet pracuje s hodnotami x_0 a x_N, metody se označují jako uzavřené, jinak jako otevřené.

Metody je možné rozdělit na několik skupin, např.:

- metody pracující s konstantní vzdáleností bodů
- metody určující pozice bodů x_k
- adaptivní metody

Do první skupiny patří klasické metody založené na Newtonových-Cotesových a jiných vzorcích. Do druhé skupiny patří zejm. varianty Gaussovy integrační metody nebo integrace založená na Čebyševově aproximaci. Tím, že pozice bodů je optimálně nastavená, je možné dosáhnout podstatně vyšší přesnosti pro daný počet bodů. Adaptivní algoritmy přizpůsobují velikost kroku dle dosažené přesnosti.

3 Newtonovy-Cotesovy vzorce

Základní Newtonovy-Cotesovy vzorce v současné době nemají přímé využití – pracují jen s malý počtem bodů, takže jsou velmi nepřesné, ale jsou základem kompozitních vzorců.

Označme $h = x_{k+1} - x_k = \text{konst.}$ a $y_k = f(x_k)$.

Pravidlo středního bodu:

$$\int_{x_0}^{x_0+h} f(x)\,dx = hf(x_0 + h/2) + O(h^2). \tag{6}$$

Lichoběžníkové pravidlo:

$$\int_{x_0}^{x_0+h} f(x)\,dx = \frac{h}{2}(y_0 + y_1) + O(h^3). \tag{7}$$

Simpsonova metoda – získaná interpolací polynomu třemi body a integrací:

$$\int_{x_0}^{x_0+2h} f(x)\,dx = \frac{h}{3}(y_0 + 4y_1 + y_2) + O(h^5). \tag{8}$$

4 Kompozitní vzorce

Interval se rozdělí na dvoubodové nebo vícebodové úseky, kde je možné aplikovat základní vzorce. Součtem se získají tzv. kompozitní vzorce:

Lichoběžníková metoda:

$$\int_{x_0}^{x_N} f(x)\,dx = h\left[\frac{y_0}{2} + y_1 + y_2 + \ldots + y_{N-1} + \frac{y_N}{2}\right] + O(h^2). \tag{9}$$

Rozšířená metoda $O(h^3)$:

$$\int_{x_0}^{x_N} f(x)\,dx = h\left[\frac{5}{12}y_0 + \frac{13}{12}y_1 + y_2 + \ldots + y_{N-2} + \frac{13}{12}y_{N-1} + \frac{5}{12}y_N\right] + O(h^3). \tag{10}$$

Simpsonova metoda (pro *N* sudé):

$$\int_{x_0}^{x_N} f(x)\,dx = \frac{h}{3}\left[y_0 + 4y_1 + 2y_2 + 4y_3 + \ldots + 2y_{N-2} + 4y_{N-1} + y_N\right] + O(h^4). \tag{11}$$

Vzorce vyšších řádů mohou být přesnější pouze v případě, že jsou použity pro funkce, které jsou dostatečně hladké. Integrovat je možné i funkce pouze po částech spojité. V tom případě je ale vhodné použít jedině lichoběžníkovou metodu.

5 Iterační metoda integrace

Přesnost kompozitních vzorců kromě h^k závisí i na velikosti $\left|f^{(k)}\right|$ (u základních je to spíše $\left|f^{(k-1)}\right|$). Ta obvykle není známa, takže může být obtížné skutečnou přesnost odhadnout.

Jednou z možností, která tento problém odstranit, je opakovaná integrace s krokem h, který se v každé iteraci půlí. Označme S_N vypočítanou hodnotu v *k*-tém kroku. Výpočet je možné ukončit, jestliže $\left|S_N - S_{2N}\right| < \varepsilon S_{2N}$, kde ε je malé číslo.

Polovinu hodnot y_k v každém kroku ale není nutno znovu počítat, protože už byly vypočítány v předchozím kroku. Využití předchozích hodnot je jednoduché zejména u lichoběžníkové metody, ale je možné i u vzorců vyšších řádů.

6 Gaussova metoda integrace

Metoda je založená na náhradě integrálu součtem

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{j=1}^{N} w_j f(x_j) \tag{12}$$

kde x_j jsou body zvolené v intervalu $[-1,1]$. Pro obecný interval $t \in [a,b]$ se provede substituce

$$x = \frac{2t-(b+a)}{b-a}. \tag{13}$$

Koeficienty w_j se určí tak, aby

$$\int_{-1}^{1} p_k(x)\,dx = \sum_{j=1}^{N} w_j p_k(x_j) \tag{14}$$

platilo pro zvolený systém funkcí $p_k(x)$, $k = 0,\dots,N-1$. Funkce $p_k(x)$ je nejvýhodnější volit jako tzv. ortogonální polynomy, pro které platí

$$\int_{-1}^{1} p_j(x)\,p_k(x)\,dx = 0 \quad \text{pro} \quad j \neq k \tag{15}$$

a body x_j jako jejich kořeny. Jednou z možností jsou tzv. Legendreovy polynomy, jejich kořeny jsou reálné, různé a všechny leží v intervalu $(-1,1)$. V tom případě je možné

dosáhnout podstatně lepší přesnosti pro stejný počet bodů, než v případě kompozitních vzorců.

Legendreovy polynomy jsou definovány rekurzivním vztahem

$$(j+1)P_{j+1}(x)=(2j+1)xP_j(x)-jP_{j-1}(x) \tag{16}$$

kde

$$P_{-1}(x)=0,\ P_1(x)=1\ . \tag{17}$$

V Tab. 1 jsou vypsány Legendreovy polynomy do stupně $n=4$.

Tab. 1 - Legendreovy polynomy.

n	$P_n(x)$
0	1
1	x
2	$(3x^2-1)/2$
3	$(5x^3-3x)/2$
4	$(35x^4-30x^2+3)/8$

Příklady

1. Vypočítejte numericky derivaci funkce $\sin x$ pro 10 hodnot v intervalu $[0,\pi/2]$ pomocí obou uvedených vzorců a vypočítejte v obou případech průměrnou absolutní hodnotu chyby pro $h=10^{-3}$. Získané výsledky porovnejte.
2. Získejte vztah pro optimální velikost kroku u vzorce pro numerickou derivaci.
3. Odvoďte Simpsonův vzorec (8), např. s využitím Lagrangeovy interpolace.
4. Vypočítejte numericky $\int_0^{\pi/2} x\cos x\,dx$ a porovnejte s přesnou hodnotou, pomocí:
   a) kompozitní lichoběžníkové metody pro $N=10$ a $N=20$
   b) kompozitní Simpsonovy metody pro $N=10$.

5. Vypočítejte numericky $\int_0^{2\pi} e^{-t} \left(\sin x\right)^2 dx$ iterační lichoběžníkovou metodou tak, aby relativní přesnost výsledku byla $\varepsilon = 10^{-9}$.
6. Vypište do textového souboru hodnoty Legendreových polynomů do řádu 6 se zvoleným krokem a data zobrazte v podobě grafu s pomocí vhodného software.

Použitá literatura

PRESS, H., TEUKOLSKY, S.A., VETTERLING, W. T., FLANNERY, B. P. 2007. *Numerical Recipes. The Art of Scientific Programming. Third Edition.* New York: Cambridge University Press, 2007.

RALSTON, A. 1987. *Základy numerické matematiky*. Praha: Academia.

YANG, W. Y., CAO, W., CHUNG T-S., MORFIA, J. 2005. *Applied Numerical Methods Using Matlab®*. John Wiley & Sons Inc.,

Rejstřík

metoda integrace
- Gaussova, 4
- iterační, 4
- lichoběžníková, 3
- rozšířená, 3
- Simpsonova, 3

numerická derivace, 1
numerická integrace, 2
optimální krok, 1
vzorce
- kompozitní, 3
- Newtonovy-Cotesovy, 2

Algoritmy numerické matematiky a zpracování dat

Téma 12: Metody řešení soustav obyčejných diferenciálních rovnic

Studijní cíl

Seznámit studenty se základními numerickými metodami řešení obyčejných diferenciálních rovnic a jejich soustav.

Doba nutná k nastudování

2 hodiny

Klíčová slova

Numerické řešení diferenciálních rovnic, lichoběžníková metoda, metoda Runge-Kutta, vícebodové metody

1 Princip numerického řešení diferenciálních rovnic

Je uvažována soustava diferenciálních rovnic ve tvaru

$$\mathbf{x}' = \mathbf{f}\left(\mathbf{x}, t\right), \quad \mathbf{x}\left(0\right) = \mathbf{x}_0 . \tag{1}$$

Zápis výše znanemá, že

$$x_i' = f_i\left(x_i, t\right),\ i = 1, \ldots, n . \tag{2}$$

Samostatné diferenciální rovnice nebo soustavy s derivacemi vyšších řádů je vždy možné převést do tvaru výše, takže je možné se omezit pouze na soustavy diferenciálních rovnic prvního řádu.

Např. rovnici

$$y^{(n)} + a_{n-1} y^{(n-1)} + \ldots + a_0 y = b_0 u \tag{3}$$

je možné rovněž zapsat jako soustavu

$$\begin{aligned} x_1' &= x_2 \\ x_2' &= x_3 \\ &\dots \\ x_n' &= -a_{n-1}x_{n-1}' - \dots - a_0 x_1' + b_0 u \end{aligned} \tag{4}$$

kde $x_1 = y$.

Standardní zápis zahrnuje i situaci, kdy na vstupu systému působí signál $\mathbf{u}(t)$, jehož průběh je v čase známý:

$$\mathbf{x}' = \mathbf{g}(\mathbf{x}, \mathbf{u}(t)). \tag{5}$$

Řešení je možné obecně vyjádřit ve tvaru

$$\mathbf{x}(t) = \mathbf{x}_0 + \int_0^t \mathbf{f}(\mathbf{x}(t), t)\, dt \tag{6}$$

ale tento vztah je spíše využívaný v matematické analýze, nikoliv pro numerické řešení.

Základní numerická metoda je založena na náhradě derivace:

$$\mathbf{x}' = \frac{\mathbf{x}(t+h) - \mathbf{x}(t)}{h} + O(h) \tag{7}$$

kde h je zvolená velikost kroku.

Dosazením pro $t = 0, t_1, t_2, \dots.$ je takto získána tzv. Eulerova metoda řešení, která je popsána vztahem

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h_k\, \mathbf{f}(\mathbf{x}_k, t_k) \tag{8}$$

kde $\mathbf{x}_{k+1} = \mathbf{x}(t+h)$, $\mathbf{x}_k = \mathbf{x}(t)$ a $h_k = t_{k+1} - t_k$. Potom numerické řešení spočívá v generování posloupnosti

$$\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_2, \dots \; . \tag{9}$$

V základním případě se t zvyšuje vždy o konstatní krok h, takže $t_k = kh$. Ale obecně je možné krok během výpočtu měnit a přizpůsobovat tvaru funkce nebo dosažené přesnosti, pokud je možné ji nějakým způsobem odhadnout.

Eulerova metoda je ale velmi nepřesná, spíše se využívá jen pro vysvětlení základního principu řešení.

2 Lichoběžníková metoda

Integrujme obě strany rovnice $\mathbf{x}' = \mathbf{f}(\mathbf{x}, t)$ v intervalu $[t_k, t_{k+1}]$:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \int_{t_k}^{t_{k+1}} \mathbf{f}(\mathbf{x}(t), t)\, dt. \tag{10}$$

Jestliže uvažujeme, že $\mathbf{f}(\mathbf{x}, t)$ je zhruba lineární pro $t \in [t_k, t_{k+1})$, platí

$$\int_{t_k}^{t_{k+1}} \mathbf{f}(\mathbf{x}(t), t)\, dt = \frac{h}{2}\left(\mathbf{f}(\mathbf{x}_k, t_k) + \mathbf{f}(\mathbf{x}_{k+1}, t_{k+1})\right) + O(h^2). \tag{11}$$

Přitom $\mathbf{f}(\mathbf{y}_{k+1}, t_{k+1})$ není známé, protože neznáme $\mathbf{x}_{k+1}$, ale je možné použít odhad získaný Eulerovou metodou:

$$\mathbf{f}(\mathbf{x}_{k+1}, t_{k+1}) \approx \mathbf{f}(\mathbf{x}_k + h\mathbf{f}(\mathbf{x}_k, t_k), t_{k+1}). \tag{12}$$

Výsledný vztah je

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \frac{h}{2}\left(\mathbf{f}(\mathbf{x}_k, t_k) + \mathbf{f}(\mathbf{x}_k + h\mathbf{f}(\mathbf{x}_k, t_k), t_{k+1})\right). \tag{13}$$

3 Metody Runge-Kutta

Vylepšením lichoběžníkové metody jsou získány metody Runge-Kutta, které pracují rovněž s hodnotami vypočtenými uvnitř intervalu $[t_k, t_{k+1}]$.

Metoda Runge-Kutta 3.řádu je ve tvaru

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \frac{h}{6}(\mathbf{f}_0 + 4\mathbf{f}_1 + \mathbf{f}_2) \tag{14}$$

kde

$$\begin{aligned} \mathbf{f}_0 &= \mathbf{f}(\mathbf{x}_k, t_k) \\ \mathbf{f}_1 &= \mathbf{f}\left(\mathbf{x}_k + \frac{h}{2}\mathbf{f}_0, t_k + \frac{h}{2}\right) \\ \mathbf{f}_2 &= \mathbf{f}(\mathbf{x}_k + h(2\mathbf{f}_1 - \mathbf{f}_0), t_k + h). \end{aligned} \tag{15}$$

Výpočet $\mathbf{x}_{k+1}$ odpovídá integraci Simpsononovým pravidlem.

Asi nejpoužívanější metodou vůbec je metoda Runge-Kutta 4. řádu:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \frac{h}{6}\left(\mathbf{f}_0 + 2\mathbf{f}_1 + 2\mathbf{f}_2 + \mathbf{f}_3\right) \tag{16}$$

kde

$$\begin{aligned}
\mathbf{f}_0 &= \mathbf{f}\left(\mathbf{x}_k, t_k\right) \\
\mathbf{f}_1 &= \mathbf{f}\left(\mathbf{x}_k + \frac{h}{2}\mathbf{f}_0, t_k + \frac{h}{2}\right) \\
\mathbf{f}_2 &= \mathbf{f}\left(\mathbf{x}_k + \frac{h}{2}\mathbf{f}_1, t_k + \frac{h}{2}\right) \\
\mathbf{f}_3 &= \mathbf{f}\left(\mathbf{x}_k + h\mathbf{f}_2, t_k + h\right).
\end{aligned} \tag{17}$$

4 Vícebodové metody

Vícebodové metody pracují ve 2 krocích: neprve vypočítají predikci $\mathbf{x}_{k+1}$ na základě extrapolace několika předchozích hodnot $\mathbf{f}$ a integrálu (10). V druhém kroku se pak provede korekce $\mathbf{x}_{k+1}$ stejným postupem, s využitím získané hodnoty $\mathbf{f}\left(\mathbf{x}_{k+1}, t_{k+1}\right)$.

Pro extrapolaci $\mathbf{f}\left(\mathbf{x}_{k+1}, t_{k+1}\right)$ je možné použít např. Lagrangeův interpolační polynom. V případě Adamsovy-Bashforthovy-Moultonovy metody je využit polynom 3. řádu a predikce $\mathbf{x}_{k+1}$ je získána ve tvaru

$$\tilde{\mathbf{x}}_{k+1} = \mathbf{x}_k + \frac{h}{24}\left(-9\mathbf{f}_{k-3} + 37\mathbf{f}_{k-2} - 59\mathbf{f}_{k-1} + 55\mathbf{f}_k\right). \tag{18}$$

Dosazením $\tilde{\mathbf{x}}_{k+1}$ do $\mathbf{f}\left(\mathbf{x}_{k+1}, t_{k+1}\right)$ se pak stejným postupem získá korekce na základě interpolace body $\mathbf{f}_{k-2}, \ldots, \mathbf{f}_{k+1}$:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \frac{h}{24}\left(\mathbf{f}_{k-2} + 5\mathbf{f}_{k-1} + 19\mathbf{f}_k + 9\mathbf{f}_{k+1}\right) + O\left(h^5\right). \tag{19}$$

Navíc je možné v odhadnout výslednou chybu prediktoru a korektoru, která může být kompenzována v dalším kroku.

Metody prediktor-korektor jsou komplikovanější než metody Runge-Kutta, ale jsou zpravidla mnohem přesnější v případě, že funkce je dostatečně hladká.

Vzhledem k tomu, že na počátku nejsou k dispozici předchozí hodnoty $\mathbf{f}_k$, je nutné pro $k < 3$ počítat $\mathbf{x}_{k+1}$ jiným způsobem. Je možné výpočet startovat např. metodou Runge-Kutta.

Příklady

1. Uvažujte rovnici $y'' + y = 0$ pro počáteční podmínky $y'(0) = 0$ a $y(0) = 1$. Přepište rovnici do tvaru $\mathbf{x}' = \mathbf{f}(\mathbf{x}, t)$.

2. Řešte rovnici z příkladu 1 numericky v intervalu $[0, 2\pi]$ a porovnejte s přesným řešením. Použijte

   a) Eulerovu metodu

   b) Lichoběžníkovu metodou

   c) Metodu Runge-Kutta.

Použitá literatura

PRESS, H., TEUKOLSKY, S.A., VETTERLING, W. T., FLANNERY, B. P. 2007. *Numerical Recipes. The Art of Scientific Programming. Third Edition.* New York: Cambridge University Press, 2007.

RALSTON, A. 1987. *Základy numerické matematiky*. Praha: Academia.

YANG, W. Y., CAO, W., CHUNG T-S., MORFIA, J. 2005. *Applied Numerical Methods Using Matlab®*. John Wiley & Sons Inc.,

Rejstřík

metoda řešení
- ABM, 4
- Eulerova, 2
- lichoběžníková, 2
- Runge-Kutta, 3
- vícebodová, 4

metody prediktor-korektor, 4
náhrada derivace, 2
princip numerického řešení diferenciálních rovnic, 1
soustava diferenciálních rovnic, 1

Algoritmy numerické matematiky a zpracování dat

Téma 13: Hledání minima funkce jedné proměnné

Studijní cíl

Seznámit studenty se základy problematiky hledání minima funkce jedné proměnné.

Doba nutná k nastudování

2 hodiny

Klíčová slova

Numerická optimalizace, hledání minima, globální minimum, metoda půlení intervalu, metoda zlatého řezu

1 Typy problémů optimalizace

Řadu technických problémů je možné formulovat jako problém hledání minima nebo maxima funkce určitého počtu parametrů. Zde je uvažován pouze případ hledání minima funkce jedné proměnné $f(x)$, o které se předpokládá, že je spojitá.

I když se problém označuje jako hledání minima, spíše hledáme bod, který odpovídá minimu funkce, tedy bod

$$x^* = \arg\min\{f(x)\}. \tag{1}$$

Problém hledání maxima není třeba uvažovat zvlášť, protože

$$\arg\max\{f(x)\} = \arg\min\{-f(x)\}. \tag{2}$$

V optimalizaci funkce jedné proměnné jsou dva základní typy problémů:

- problém hledání globálního minima v daném intervalu
- problém hledání jednoho lokálního minima.

V zadaném intervalu $[l,h]$ může mít funkce více lokálních minim. V tom případě je možné skutečné (globální) minimum určit výběrem z nalezených lokálních minim. Lokální minima mohou vzniknout i na hranici intervalu $[l,h]$.

V řadě případů je ale známo nebo je možné předpokládat, že funkce $f(x)$ má v zadaném intervalu jen jedno lokální minimum. Pak se hledání globálního minima redukuje na nalezení jediného lokálního minima, což je výrazně jednodušší problém.

V případě, že minimalizovaná funkce je diferencovatelná, je v principu možné problém hledání minima $f(x)$ převést na hledání řešení $f'(x)=0$, kde je navíc nutné vybrat ty kořeny, které odpovídají lokálním minimům, což je možné rozlišit na základě znaménka $f''(x)$. Typ lokálního extrému (minimum/maximum) je často možné rozeznat už ve fázi vymezení intervalů, ve kterých se kořen $f'(x)$ nachází. Např. při řešení půlením intervalu je možné pokračovat pouze v intervalech $[l,h]$, kde $f'(l)<0$ a $f'(h)>0$. Tím je zajištěno, že nalezený extrém je lokálním minimem.

V řadě případů derivace minimalizované funkce však není k dispozici v analytickém tvaru. Numerický výpočet $f'(x)$ nemusí být efektivní, ale především je nepřesný. Požadavek diferencovatelnosti je navíc omezující – často je třeba hledat minimum funkce, o které je pouze známo, že je spojitá, ale nemusí mít spojitou derivaci. Ukazuje se ale, že pro hledání minima není nutno pracovat s derivací $f(x)$ a stačí, aby funkce $f(x)$ byla spojitá.

Algoritmus hledání minima obecně má 2 základní kroky:

1. vymezení intervalů, ve kterém se nachází lokální minimum
2. iterační zkrácení intervalů nejistoty na dostatečně malou délku
3. zpřesnění výsledku interpolací polynomem.

2 Vymezení intervalů, ve kterém se nachází lokální minimum

V případě problému globální optimalizace je např. možné v intervalu $[l,h]$ zvolit $x_0=l$, $x_N=h$ a vygenerovat $N-1$ bodů x_k uvnitř intervalu $[l,h]$ tak, aby

$$l<x_1<x_2<\ldots<x_{N-1}<h\,. \tag{3}$$

Jestliže existuje trojice bodů x_k taková, že platí

$$f(x_{k-1})>f(x_k) \text{ a } f(x_k)<f(x_{k+1}), \tag{4}$$

interval $[x_{k-1},x_{k+1}]$ obsahuje lokální minimum.

V případě, že žádný interval $[x_{k-1},x_{k+1}]$ nebyl nalezen a je známo, že funkce v $[l,h]$ má minimum, je možné interval $[l,h]$ rozšířit nebo postup opakovat s vyšším N.

V případě hledání jednoho lokálního minima je obvykle zadán výchozí bod x_0, ze kterého se minimum hledá. Pak je třeba určit směr, ve kterém funkce v bodě x_0 klesá (kladný, nebo záporný). Stačí např. vypočítat $f(x_1)$ pro $x_1 = x_0 + h$, kde $h > 0$ je zvolené malé číslo. Jestliže neplatí $f(x_1) < f(x_0)$, dosadí se $h \leftarrow -h$. Pokud $f(x_1) \geq f(x_0)$ v obou případech, je nutné krok h zkrátit.

Dále, je třeba nalézt bod x_2 takový, že $f(x_2) > f(x_1)$. Je možné zkusit $x_2 = x_1 + (x_1 - x_0)$. Pokud $f(x_2) > f(x_1)$ neplatí, je možné za x_1 zvolit předchozí x_2 a postup opakovat. Algoritmus níže se ukončí, jestliže jsou nalezeny body x_1 a x_2 takové, že platí

$$f(x_0) > f(x_1) \text{ a } f(x_1) < f(x_2) \tag{5}$$

nebo je $|x_1 - x_0|$ větší než zadaná hodnota $d_{\max}$, což indikuje neúspěšnost.

1. $x_1 \leftarrow x_0 + h$. Jestliže $f(x_1) \geq f(x_0)$:

   $x_1 \leftarrow x_0 - h$.

   Jestliže $f(x_1) \geq f(x_0)$, zvol menší h a opakuj bod 1. Jestliže h je příliš malé, ukonči – neúspěch.

2. $x_2 \leftarrow x_1 + (x_1 - x_0)$. Jestliže $|x_1 - x_0| > d_{\max}$, ukonči – neúspěch.
   Jestliže podmínka (5) je splněna, ukonči – vrať body x_1 a x_2.
   V opačném případě, $x_1 \leftarrow x_2$ a pokračuj od bodu 2.

3 Iterační zkrácení intervalů nejistoty

Ve vymezených intervalech $[x_{k-1}, x_{k+1}]$ se nachází lokální minimum, které je možné najít iteračním způsobem. Základním algoritmus využívá techniku půlení intervalu. Na rozdíl od metody půlení intervalu pro hledání kořene však pracuje s 5 body.

Označme $[x_L, x_H]$ vymezený interval a $x_M = (x_L + x_H)/2$, a předpokládejme, že platí $f(x_L) > f(x_M)$ a $f(x_M) < f(x_H)$ jako výsledek předchozího kroku. Algoritmus půlení intervalu je následující:

1. Vypočítej body $x_{LM} = (x_L + x_M)/2$ a $x_{HM} = (x_M + x_H)/2$.
2. Jestliže $f(x_{LM}) < f(x_M)$ proveď $x_H \leftarrow x_M$ a $x_M \leftarrow x_{LM}$ (minimum leží v $[x_L, x_M]$).

Jestliže $f(x_{HM}) < f(x_M)$, proveď $x_L \leftarrow x_M$ a $x_M \leftarrow x_{HM}$ (minimum leží v $[x_M, x_H]$).

Jinak, proveď $x_L \leftarrow x_{LM}$ a $x_H \leftarrow x_{HM}$ (minimum leží v $[x_{LM}, x_{HM}]$).

3. Jestliže $|x_H - x_L| < \varepsilon$, kde $\varepsilon > 0$ je zvolené, ukonči výpočet – vrať bod x_M.

V každém kroku je interval nejistoty redukován na polovinu. Pro šířku intervalu nejistoty na konci platí $h_k = h_0 / 2^k$, kde k je počet kroků a $h_0 = x_M - x_L$ je výchozí délka. Jelikož $h_k < \varepsilon$, platí

$$k = [\![\log_2 (h_0 / \varepsilon) + 1]\!] \tag{6}$$

kde $k = [\![.]\!]$ označuje zaokrouhlení směrem k nule. Pro zkrácení intervalu nejistoty na polovinu je třeba vypočítat vždy jen 2 body x_{LM} a x_{HM} a jejich funkční hodnoty, protože ostatní hodnoty je možné využít z předchozího kroku. Celkem je tedy třeba vypočítat zhruba $2\log_2(h_0/\varepsilon)$ hodnot.

Ukazuje se, že z hlediska počtu vypočtených funkčních hodnot metoda půlení intervalu není nejefektivnější. Tzv. algoritmus zlatého řezu pracuje pouze se 4 body a v každém kroku vypočítává pouze jednu novou hodnotu. V tomto případě testovací body nejsou stejně vzdálené, ale v průběhu výpočtu si vzdálenosti zachovávají stejný poměr. Platí $h_{k+1} = q\, h_k$, kde

$$q = \left(\sqrt{5} - 1\right)/2 \approx 0.618 \tag{7}$$

je tzv. kvocient zlatého řezu.

4 Zpřesnění výsledku interpolací polynomem

Vzhledem k tomu, že z předchozího kroku jsou známé hodnoty $f(x)$ v několika bodech v blízkém okolí minima, je možné těmito body interpolovat polynom, jehož minimum je možné vypočítat analyticky. Jestliže je takto získán bod s menší hodnotou, nahradí původní odhad.

Např. v případě metody půlení intervalu jsou v posledním kroku k dispozici 3 body x_L, x_H a x_M, kterými je možné proložit polynom 2. řádu

$$\tilde{f}(x) = a_2 x^2 + a_1 x + a_0 . \tag{8}$$

Jestliže $a_2 > 0$, minimum odpovídá $x = -a_1 / (2a_2)$.

Příklady

1. Pro funkci $f(x)=\mathrm{tg}(x)+x^{-1}$:

   a) najděte výchozí interval nejistoty $[x_L, x_H]$, ve kterém leží minimum, uvnitř intervalu $[0, \pi/2]$.

   b) najděte iteračně minimum metodou půlení intervalu tak, aby pro konečný interval nejistoty platilo $x_H - x_L < 10^{-6}$.

2. Pokuste se zpřesnit řešení příkladu 1 interpolací kvadratickým polynomem body x_L, x_H a x_M a výpočtem minima analyticky (ověřte, jestli takto získaný bod má menší hodnotu). Využijte Lagrangeovu nebo Newtonovu interpolační metodu.

Použitá literatura

HANUŠ, B. 1985. Teorie automatického řízení II (skripta). Liberec: Fakulta strojní.

PRESS, H., TEUKOLSKY, S.A., VETTERLING, W. T., FLANNERY, B. P. 2007. *Numerical Recipes. The Art of Scientific Programming. Third Edition.* New York: Cambridge University Press, 2007.

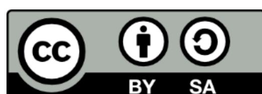
Rejstřík

algoritmus zlatého řezu, 4
hledání minima
  globálního, 1
  lokálního, 1
metoda půlení intervalu, 3
typ lokálního extrému, 2
typy problémů optimalizace, 1
vymezení intervalů, 2
zkrácení intervalů nejistoty, 3
zpřesnění výsledku interpolací, 4

UNIVERZITA
PARDUBICE
FAKULTA
ELEKTROTECHNIKY
A INFORMATIKY

Vytvořeno v rámci projektu **Studijní program Automatizace (SPAUT)**
na Univerzitě Pardubice, reg. č. NPO_UPCE_MSMT-16591/2022.

Toto dílo podléhá licenci Creative Commons BY 4.0. Pro zobrazení licenčních podmínek navštivte https://creativecommons.org/licenses/by-sa/4.0/.



Financováno
Evropskou unií
NextGenerationEU

Národní
plán
obnovy

MINISTERSTVO ŠKOLSTVÍ,
MLÁDEŽE A TĚLOVÝCHOVY