Iterační metody řešení (ne)lineárních rovnic

1. Funkce

Funkce v matematickém významu je zobrazení z jedné množiny do druhé *f: X→Y*. My se budeme zabývat funkcemi, které pracují s reálnými čísly, takže $X \subseteq \mathbb{R}$, $Y \subseteq \mathbb{R}$. Funkce je předpis, který pro každý prvek ze vstupní množiny *X* (definiční obor funkce, doména) jednoznačně přiřazuje hodnotu z výstupní množiny *Y* (obor hodnot funkce).

Protože se budeme zabývat spojitými funkcemi na množině reálných čísel $\mathbb{R}$, nelze toto zobrazení vyjmenovat. Musíme mít funkční předpis, neboli vzorec, který který bude funkci definovat. Obvykle funkce zapisujeme pomocí definičního přiřazení.[1]

$$f(x)=3x^2-1 \tag{1}$$

Zde hodnota *x* představuje hodnotu z definičího oboru (v grafu osa x), zápis *f(x)* označuje hodnotu z oboru hodnot (v grafu je to osa y) a pravá strana za rovnítkem představuje předpis, jak výslednou hodnotu získat.

Výraz na pravé straně přiřazení (1) je polynom, neboli mnohočlen. Funkce, které jsou definovány takto, nazýváme *polynomiální funkce*.

Funkce, které dokážeme zapsat pomocí konečného počtu elementárních operací (+, -, *, /) a složení z konstantní, mocninné, exponenciální, logaritmické, goniometrické, cyklometrické, hyperbolické a hyperbolometrické funkce, nazýváme *elementární funkce*.

Polynomiální funkce je tedy speciální případ elementární funkce. Lineární funkce je zase speciálním případem polynomiální funkce (je tvořena polynomem *ax+b*). Všechny ostatní elementární funkce jsou *nelineární funkce*.[2]

2. Rovnice

Častou úlohou matematické analýzy je řešení rovnic. Obecně je rovnice tvrzení o rovnosti dvou výrazů (respektive funkcí). Nejčastěji se ale tato rovnost zapisuje ve tvaru[3][4]

$$f(x)=0 \tag{2}$$

Řešením takovéto rovnice jsou všechny hodnoty *x* takové, že tato rovnost platí. Těmto hodnotám *x* pak říkáme *kořeny* funkce. Graficky jde o průsečíky grafu funkce *f* s osou x.

Pro některé elementární funkce (lineární, kvadratická, kubická) umíme najít řešení analyticky (tj. umíme odvodit vzorec popisující řešení). Pro drtivou většinu elementárních funkcí je ale analytické řešení příliš složité, proto se uchylujeme k numerickým metodám (iteračním výpočtům se zadanou přesností ε).

3. Hornerovo schéma

Protože při numerickém řešení rovnic budeme často vyšetřovat polynomiální funkce, je dobré umět je efektivně vyčíslit.

Polynomiální funkce n-tého stupně je definována jako

$$P(x)=a_n x^n+a_{n-1}x^{n-1}+\cdots+a_1 x+a_0 \tag{3}$$

Zkráceně to můžeme zapsat jako

$$P(x)=\sum_{i=0}^{n} a_i x^i \tag{4}$$

Hodnotou *n* označujeme řád polynomu a symboly *a* představují koeficienty polynomu.

Při vyčíslování to lze dělat naivně podle předpisu (4), ale to vyžaduje opakované počítání mocnin. Algoritmus pak má kvadratickou složitost.

Chytřejší algoritmus se nazývá Hornerovo schéma a využívá toho, že polynom lze postupným vytýkáním a závorkováním převést do tvaru, ve kterém žádná mocnina nezůstane.

$$P(x)=(\cdots((a_n)x+a_{n-1})x+\cdots+a_1)x+a_0 \tag{5}$$

Tento algoritmus má složitost lineární.

---

1 Někdy se také používá dvojtečka, aby se to odlišilo od zápisu rovnic.

2 Konstantní funkci můžeme považovat za speciální případ lineární funkce.

3 Každou rovnost funkcí *f(x) = g(x)* lze převést do ekvivalentního tvaru *f(x) – g(x) = 0*.

4 Zde již nejde o definiční rovnítko, ale o porovnání.

Úkol č. 1

Napiš funkci `horner`, která vyčíslí hodnotu polynomu v zadaném bodě x. Koeficienty zadávej jako pole.

```
float horner(float koef[], int n,
             float x);
```

Poznámka č. 1: Prvky v poli koeficientů budou mít indexy v opačném pořadí oproti vzorci (5). To ale nevadí, protože takový zápis bude přirozenější. V algoritmu bude proměnná `suma`, kterou budeš v každé iteraci cyklu násobit hodnotou x a přičítat další koeficient. Je nutné to dělat od nejvyšší mocniny – v našem případě tento koeficient bude na indexu 0.
Poznámka č. 2: Pokud bude hodnota n řád polynomu, pak pole bude o jedničku delší.
Poznámka č. 3: Proměnnou suma nezapomeň správně inicializovat.

Otestuj správnost funkce na několika zvolených polynomech (definuj je mimo tuto funkci). Správnost můžeš ověřit například pomocí tabulkového procesoru.

Pro další práci si vytvoř několik polynomiálních funkcí tímto způsobem:

```
float f1(float x) {
  float koef [] = {-5.0, 3.2, 7.5};
  return horner(koef, 2, x);
}
```

Připrav si takto i několik elementárních funkcí obsahujících například goniometrické funkce, logaritmy atd. Důležité je, aby tyto pomocné funkce měly jediný parametr x a vracely funkční hodnotu. Připrav si je tak, aby měly alespoň jeden kořen.

4. Iterační řešení rovnic

Numerické algoritmy pro hledání kořene zadané rovnice f(x) = 0 nemohou kořen hledat hrubou silou, tj. zkoušením všech bodů x, protože pracujeme v oboru reálných čísel a takových bodů je v každém neprázdném intervalu nekonečně mnoho.

Všechny probírané algoritmy spoléhají na několik základních podmínek, za kterých mohou fungovat:

1. Je zadána přesnost ε, jako maximální přípustná vzdálenost od přesného řešení.
2. Je zadán počáteční bod „rozumně“ blízko předpokládaného kořene.
   a) U interpolačních metod je vyžadován interval, v němž se nachází právě jeden kořen – tzv. separace kořene. Pro interval $\langle a, b\rangle$ musí platit, že $f(a)f(b) < 0$.
   b) U extrapolačních metod je vyžadován počáteční bod zadaný tak, aby mezi ním a kořenem byla funkce pouze rostoucí nebo pouze klesající. V opačném případě není zaručena konvergence.
3. Vyšetřovaná funkce f je spojitá (alespoň v intervalu vstupních hodnot, kudy se bude algoritmus blížit k řešení).

Algoritmy dělíme na interpolační a extrapolační. Interpolační metody hledají řešení uvnitř zadaného intervalu a pokud jsou dodrženy výše uvedené vstupní podmínky, vždy konvergují k řešení.

Extrapolační metody nemají zadán interval, ve kterém by řešení hledaly. Hledají ho v okolí zadaného počátečního bodu (řešení ale může být i poměrně daleko). Pokud se ale mezi vyšetřované body a kořen dostane nějaký bod v němž je derivace nulová, není konvergence metody zaručena. Obecně ale bývají rychlejší než metody interpolační.

4.1 Práce s přesností

Podstatou řešení naší úlohy je najít takový bod x, že funkční hodnota v tomto bodě bude nula, tedy $f(x) = 0$. Protože toho nedokážeme dosáhnout v rozumném čase, spokojíme se s takovou hodnotu x, pro které bude funkční hodnota od nuly vzdálená maximálně o zadanou hodnotu přesnosti ε. Budeme tedy vyšetřovat podmínku

$$|f(x)|<\varepsilon \qquad (6)$$

Pokud je vyšetřovaná funkce v okolí kořene velmi plochá, může se stát, že nalezneme bod, který sice bude splňovat tuto podmínku, ale bude od přesného řešení velmi daleko – ve směru osy x. Proto je možné přesnost vyhodnocovat i v tomto směru. Například u interpolačních metod, kde se během výpočtu interval $\langle a, b\rangle$ posunuje stále blíž ke kořeni, lze vyhodnocovat podmínku

$$|b-a|<\varepsilon \qquad (7)$$

5. Interpolační algoritmy

Interpolační algoritmy mají zadaný interval <*a, b*>, v němž musí být separován kořen. Musí tedy platit podmínka, že $f(a)f(b) < 0$ (to znamená, že funkční hodnoty v krajních bodech intervalu musí mít opačná znaménka). Princip této podmínky se využívá v algoritmu pro rozhodování, v jaké části intervalu se bude kořen hledat v dalším kroku.

Obecně má interpolační algoritmus jako vstup interval <*a, b*>, přesnost ε a vyhodnocovanou funkci *f*. Kroky algoritmu:

1) Zvol bod *c* uvnitř intervalu <*a, b*>.
2) Dokud není dosažena přesnost ε opakuj (buď podle (6), nebo podle (7)).
   a) Pokud $f(a)f(c) < 0$
      - nastav $b = c$
   b) jinak
      - nastav $a = c$
   c) Zvol bod *c* uvnitř upraveného intervalu <*a, b*>.

5.1 Bisekce – půlení intervalu

Algoritmus bisekce, neboli půlení intervalu je známý už ze starověku. Jde o nejjednodušší interpolační metodu.

Z výše uvedeného interpolačního algoritmu se stane metoda půlení intervalu tak, že operaci „zvol bod c“ vyřešíme takto

$$c = \frac{a+b}{2} \qquad (8)$$

Úkol č. 2

Vytvoř funkci `bisekce`, která zadanou funkci *f* vyšetří v intervalu <*a, b*> a najde v něm kořen metodou půlení intervalu se zadanou přesností ε, jehož pozici vrátí jako funkční hodnotu. Pokud v intervalu žádný kořen není, vrať hodnotu NAN (not a number, stačí vrátit výraz 0.0/0.0, což je také nedefinovaná hodnota). Tuto hodnotu lze ve volající funkci testovat pomocí funkce `isnan` z knihovny `math.h`.

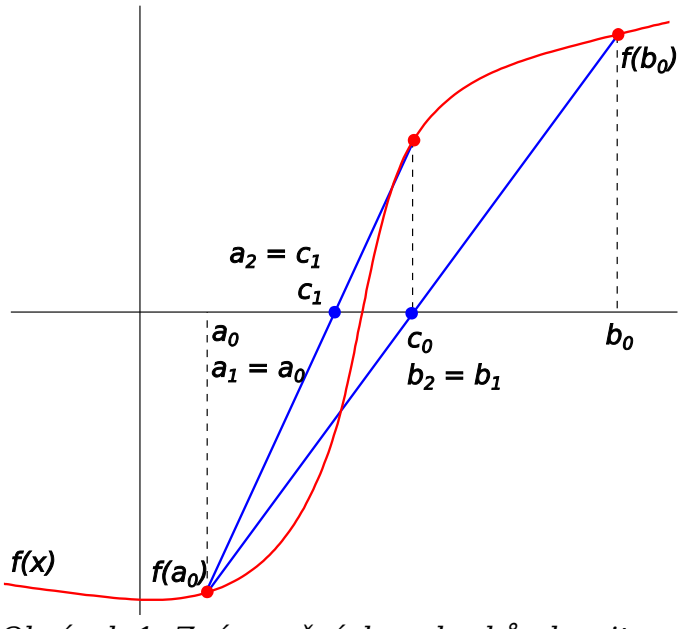
Funkce `bisekce` bude vyšetřovat zadanou funkci f. V jazyce C lze používat funkce jako parametry jiných funkcí. Dělá se to pomocí specifikace typu ukazatel na funkci (zde nový typ `Tfun`). V místě takového parametru pak lze při volání funkce bisekce dosadit jako argument jméno kompatibilní funkce (zde má jeden parametr typu float a vrací float). V této ukázce jsem dosadil jméno funkce `f1` z předchozí ukázky v sekci Úkol č. 1.

```
// deklarace typu Tfun
typedef float (*Tfun)(float);

float bisekce(float a, float b,
              Tfun f, float epsilon) {
  ...
  float fa = f(a); // vyčíslení f
  ...
}

...
// volání funkce bisekce s parametrem
// f1, což je název funkce z předchozí
// ukázky
float koren = bisekce(0.0, 2.0, f1,
                      0.0001);
...
```

5.2 Regula falsi

I tento algoritmus je znám už od starověku. V české literatuře se mu někdy říká metoda tětiv. Vychází z úvahy, že když funkci f v krajních bodech intervalu nahradíme úsečkou (tětivou), bude to dostatečná aproximace tvaru funkce *f*.

Čím blíže bude interval hledanému kořenu, tím bude tato podobnost větší (viz Obrázek 1), proto najdeme kořen rychleji než metodou půlení intervalu, která na tvar funkce nebere žádný ohled.



Obrázek 1: Znázornění dvou kroků algoritmu Regula falsi.

Poloha bodu *c* se dá odvodit jednoduchou úměrou. Tětiva a krajní body intervalu vymezují dva podobné trojúhelníky. Pak musí platit, že poměr stran *–f(a)* ku *f(b)* je stejný jako poměr stran *c–a* ku *b–c*. Z toho snadno odvodíme, že poloha bodu c na ose x je

$$c = \frac{a\,f(b) - b\,f(a)}{f(b) - f(a)} \qquad (9)$$

Úkol č. 3

Vytvoř funkci `regulafalsi`, která zadanou funkci *f* vyšetří v intervalu <*a, b*> a najde v něm kořen metodou Regula falsi se zadanou přesností *ε*. Postupuj obdobně jako v případě funkce `bisekce` z předchozího úkolu.

Odzkoušej ji na alespoň pěti různých funkcích (polynomy, goniometrické, složené).

6. Extrapolační algoritmy

Extrapolační algoritmy se liší od interpolačních tím, že hledají výslednou hodnotu i mimo zadaný počáteční interval.

V praxi jsou tyto metody rychlejší, ale méně spolehlivé – není u nich totiž zaručena konvergence. Metody mohou divergovat zejména tehdy, když se mezi krajními body intervalu a hledaným kořenem nachází lokální minima, maxima nebo stacionární inflexní body.[5] Pokud má funkce více kořenů, může také extrapolační algoritmus v takové situaci najít jiný kořen, než ten nejbližší počátečnímu intervalu.

Dvě metody, které si uvedeme jsou rychlejší než předchozí interpolační. Říkáme o nich, že *konvergují kvadraticky*, na rozdíl od interpolačních, které *konvergují lineárně* (tedy v tomto případě pomaleji).

V praxi se nespolehlivost extrapolačních metod dá řešit tak, že se nejprve vhodným způsobem přiblížíme hledanému kořeni (například několika kroky interpolační metody) a pak velice rychle najdeme řešení extrapolační metodou.

6.1 Metoda sečen

Vstupem tohoto algoritmu jsou opět dva body na ose x, které vymezují počáteční interval, podobně jako u předchozích metod. Jiné u této metody je, že nyní může kořen ležet i mimo tento interval. Ideální je, když mezi počátečním intervalem a hledaným kořenem neleží žádné body s nulovou derivací.

Funkčními hodnotami v těchto bodech protáhneme přímku (proto metoda sečen) a spočteme její průsečík s osou x. Vztah pro hledání tohoto průsečíku je stejný, jako u metody Regula falsi (9). Když totiž bude kořen náhodou ležet uvnitř zadaného intervalu, je situace úplně totožná s metodou Regula falsi (viz Obrázek 1).

Oproti metodě Regula falsi je zde použit jiný postup:

1. Z počátečních bodů *a, b* a funkčních hodnot *f(a), f(b)* vypočti průsečík s osou x – bod *c*.
2. Pokud je |*f(c)*| větší, nebo rovno *ε*, opakuj:
   a) Zahoď nejstarší bod intervalu, tedy bod a. Interval <*b, c*> se nyní stává intervalem <*a, b*> (kopíruj proměnné)
   b) Z nových hodnot *a, b, f(a), f(b)* vypočti novou hodnotu bodu *c*.
3. Po ukončení cyklu je *c* hledaným kořenem.

Kvůli možnému zacyklení se často do tohoto algoritmu přidává počítadlo iterací. Když toto počítadlo dosáhne předem daného počtu (například 1000), algoritmus skončí s nedefinovaným výsledkem.[6]

Úkol č. 4

Vytvoř funkci `secny`, která zadanou funkci *f* vyšetří v intervalu <*a, b*> a najde v něm kořen metodou sečen se zadanou přesností *ε*.

Odzkoušej ji na alespoň pěti různých funkcích (polynomy, goniometrické, složené).

6.2 Newtonova metoda

Tato metoda se česky také nazývá Metoda tečen. Isaac Newton, který tuto metodu vymyslel, je považován za jednoho ze zakladatelů diferenciálního a integrálního počtu. Tato metoda je praktickou ukázkou odvození a využití derivací.

5 Jsou to všechno body v nichž je první derivace funkce nulová, tedy $f'(x)=0$.

6 Nedefinovaná hodnota jde v jazyce C v typech float a double vyjádřit konstantou NAN z matematické knihovny, případně hodnotou výrazu 0.0/0.0.

Newtonova úvaha spočívala v tom, že když vezmeme známou metodu sečen a budeme body počátečního intervalu přibližovat k sobě, bude se výsledná sečna stále více a více přibližovat tečně, kterou bychom vedli z funkční hodnoty uprostřed tohoto intervalu. Pomocí takové limity se skutečně dá definovat derivace funkce v bodě x. Derivace funkce v bodě x, tedy $f'(x)$, má hodnotu směrnice tečny vedené tímto bodem.

Průsečík tečny s osou x odvodíme snadno dosazením do rovnice:

$$ax+b=0 \qquad (10)$$

Koeficient a je směrnicí tečny, dosadíme za něj tedy $f'(x_p)$, kde x_p je vyšetřovaný bod. Koeficient b představuje posunutí ve svislém směru, dosadíme za něj tedy $f(x_p)$. Tečna je ve vodorovném směru posunutá právě o vzdálenost x_p, proto za x dosadíme $(x_n - x_p)$, *kde* x_n je nový hledaný průsečík s osou x.

$$f'(x_p)(x_n-x_p)+f(x_p)=0 \qquad (11)$$

Osamostatněním x_p dostaneme výsledný vzorec pro výpočet dalšího bodu:

$$x_n=x_p-\frac{f(x_p)}{f'(x_p)} \qquad (12)$$

Algoritmus Newtonovy metody je velice podobný metodě sečen, ze které vlastně vzešel. Vstupem algoritmu je teď vyšetřovaná funkce f a nyní pouze jediný bod x poblíž předpokládaného kořene. Dále musíme mít k dispozici funkční předpis f', tedy derivaci původní funkce. Samozřejmě je vstupem algoritmu ještě požadovaná přesnost ε.

Algoritmus Newtonovy metody je nyní oproti předchozím velice jednoduchý:

1. Pokud $|f(x)| \geq \varepsilon$, opakuj
   a) podle vztahu (12) vypočti z hodnot x, $f(x)$ a $f'(x)$ nový bod a vlož jej do proměnné x.
2. Po ukončení cyklu je x hledaným kořenem.

Opět je možné do algoritmu přidat počítadlo iterací a při překročení určitého počtu kroků ukončit výpočet s nedefinovaným výsledkem.

Úkol č. 5

Vytvoř funkci `newton`, která u zadané funkce f najde kořen poblíž bodu a Newtonovou metodou se zadanou přesností ε.

Odzkoušej ji na alespoň pěti různých funkcích, u nichž umíš vypočítat funkční předpis derivace f'. Tato druhá funkce bude dalším vstupem algoritmu.

Například u polynomu

$$f(x)=6x^3+4x^2-7x-2 \qquad (13)$$

využijeme znalosti pravidla o derivování mocniny $(x^a)' = ax^{a-1}$, proto bude jeho derivace

$$f'(x)=18x^2+8x-7 \qquad (14)$$