Iterační metody řešení soustav lineárních rovnic

1. Iterační metoda pro aproximaci funkcí

Iterační metoda je numerická metoda používaná pro výpočty fungující na principu vyčíslování *nekonečné posloupnosti* a postupného *zpřesňování*.

Nekonečná posloupnost (s pamětí délky 1) je dána rekurentním vztahem, jehož zobecněný zápis vypadá takto:

$$Y_{i+1} = F(Y_i) \tag{1}$$

Znamená to, že každý další prvek posloupnosti Y_{i+1} se vypočítá pomocí funkčního vztahu F z předchozího prvku Y_i. Aby to dávalo smysl, musí být dán nějaký počáteční prvek Y_0, musí platit, že žádné dva prvky posloupnosti se nesmí shodovat (jinak by vznikla periodicky se opakující řada hodnot) a musíme umět v nějakém konečném kroku n rozpoznat hledanou hodnotu.

Zobecněný algoritmus popisující řešení rekurentního vztahu (1) se nazývá **algoritmické schéma**.

$$\begin{aligned} &Y = y_0 \\ &\text{while}(\neg B(Y)) \\ &\quad Y = F(Y) \end{aligned} \tag{2}$$

Symbol Y reprezentuje zobecněnou proměnnou, symbol B je predikát, neboli podmínka testující aktuální hodnotu Y a symbol F značí nějakou funkci, která umí z předchozí hodnoty posloupnosti vypočítat hodnotu novou.

Tento typ výpočtů se typicky používá pro *aproximaci funkčních hodnot*. Pro jakoukoli běžnou matematickou funkci lze najít rekurentní vztah popisující nekonečnou řadu, která se limitně blíží k přesné hodnotě této funkce v zadaném bodě. Takové řadě, která se limitně blíží k nějaké hodnotě říkáme **konvergentní řada**.

Protože taková řada je nekonečná, pracujeme s **přesností**, neboli **chybou výpočtu ε**. To je malé kladné číslo, udávající (absolutní nebo relativní) největší přípustnou vzdálenost prvku posloupnosti od přesné hodnoty řešení.

Protože přesné řešení předem neznáme, pracujeme s přesností ε v podmínce B typicky tak, že testujeme absolutní hodnotu rozdílu aktuálního a předchozího prvku posloupnosti.

$$|Y_i - Y_{i-1}| < \varepsilon \tag{3}$$

To si můžeme dovolit, protože pracujeme s konvergentní posloupností, tudíž se od určitého kroku i musí rozdíly mezi následujícími kroky zákonitě zmenšovat. Pokud by to neplatilo, mluvíme o *divergentní řadě* a náš výpočet by se zacyklil.

2. Iterační metoda pro soustavy lineárních rovnic

Tento postup se dá použít i pro řešení lineárních rovnic řádu n. Obecně vypadá i-tá rovnice takto:

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n = b_i \tag{4}$$

To můžeme zkráceně zapsat jako

$$\sum_{j=1}^{n} a_{ij}x_j = b_i \tag{5}$$

Princip je takový, že z každé rovnice soustavy osamostatníme jednu neznámou, čímž získáme n rekurentních vztahů. My je ale všechny můžeme zapsat obecně takto:

$$x_i = \frac{b_i - \sum_{j=1, i\neq j}^{n} a_{ij}x_j}{a_{ii}} \tag{6}$$

Obecný zápis je důležitý, protože popisuje obecný algoritmus.

Nyní můžeme soustavu rovnic řešit tak, že na počátku zvolíme náhodné hodnoty neznámých x_i a budeme počítat n posloupností s předem zadanou přesností ε. Výpočet ukončíme, když v nějakém kroku pro všechny posloupnosti zjistíme, že absolutní hodnota rozdílu předchozího a aktuálního kroku je menší než zadané ε.

2.1 Konvergence

Uvedený postup je radikálně odlišný od eliminačních metod, protože během výpočtu nemusíme nijak měnit koeficienty soustavy. Jediné, co se mění jsou hodnoty samotných neznámých. To je výhodné zvláště u rozsáhlých soustav, kde neustálé změny v matici výpočet velmi zdržují.

Není to ale zadarmo. Iterační metody fungují jen pro konvergentní posloupnosti. U soustav rovnic je dokázáno, že výpočet konverguje, když je matice tzv. **řádkově diagonálně dominantní**. Čili pro každý řádek platí, že absolutní hodnota diagonálního prvku je ostře větší než součet absolutních hodnot ostatních koeficientů (absolutní člen se do toho nepočítá). Matematicky to lze zapsat takto:

$$\forall i \in \langle 1, n \rangle : |a_{ii}| > \sum_{j=0, j \neq i}^{n} |a_{ij}| \qquad (7)$$

Čteme: Pro všechna i od 1 do n platí: absolutní hodnota a_{ii} je ostře větší než součet absolutních hodnot ostatních koeficientů kromě a_{ii}.

Algoritmicky výhodnější je pak všechny koeficienty na řádku posčítat a diagonální prvek poté odečíst, případně sumu porovnávat s jeho dvojnásobkem.

Poznámka č. 1: U iterační metody není potřeba testovat nuly na diagonále, protože taková matice by stejně nevyhověla testu diagonální dominance.

Poznámka č. 2: Pokud matice testu nevyhoví, je možné se pokusit ji do vyhovujícího tvaru dostat pomocí ekvivalentních úprav typu výměna řádků a sloupců. Pokud ani to nepomůže, neznamená to, že je soustava nutně neřešitelná. Pouze není *zaručena* konvergence při výpočtu iterační metodou.

Úkol č. 1

Napiš funkci `jeDDM`, která bude vracet logickou hodnotu `true`, když zadaná matice soustavy je řádkově diagonálně dominantní

```
bool jeDDM(Tmatice *m);
```

Pro práci s logickými hodnotami použij systémovou knihovnu stdbool.h. Pro práci s maticemi, jejich načítání ze souboru a tisk použij modul matice.h z předchozích cvičení.

Funkčnost vyzkoušej v testovací funkci *testDDMatice* (napiš ji). Připrav si soubory s různě velkými soustavami, které tuto vlastnost splňují a nesplňují. Například tato soustava ji nesplňuje kvůli prvnímu řádku matice:

```
4 5
8.5 1.8 2.8 3.9 5.5
1.1 5.9 -.1 2.6 1.8
 -1 4.6 -17 3.2 -24
6.4 2.7 6.1 25  2.4
```

2.2 Úprava matice pro iterační výpočet

Podíváme-li se na vzorec (6), zjistíme, že při výpočtu řady opakovaně děláme zbytečné operace. Například vždy dělíme diagonálním prvkem. Kdybychom to dělat nemuseli, výpočet by určitě byl rychlejší. Naštěstí toho jde docílit jednoduchou úpravou matice soustavy. Než začneme samotný iterační výpočet, vydělíme každý řádek rozšířené matice soustavy jejím diagonálním prvkem. Tím na diagonále zbude hodnota jedna a jedničkou dělit nemusíme.

Další problém způsobuje ono $j \neq i$ v cyklu počítajícím sumu. Protože neznámá od diagonálního koeficientu je na levé straně rovnice, nemůžeme s ní počítat na pravé straně. Abychom v cyklu nemuseli tento krok testovat a přeskakovat jej, můžeme matici soustavy ještě mírně poupravit. Protože diagonální prvky teď mají hodnotu 1, už je nebudeme dále pro výpočet potřebovat. Tak je rovnou změníme na hodnotu 0. Když pak v cyklu počítajícím sumu narazíme na tento prvek, dojde k přičtení hodnoty $0 \cdot x_i$, což je nula a nic to tedy nepokazí[1].

Nyní tedy budeme moci iterační metodou počítat podle zjednodušeného vztahu[2]:

$$x_i = b_i - \sum_{j=1}^{n} a_{ij} x_j \qquad (8)$$

Tip: Test konvergence jde provádět i na upravené matici soustavy. Poté je možno zjednodušit i samotný test, kdy se suma absolutních hodnot koeficientů na řádku porovnává přímo s hodnotou 1.

1 V praxi by se použila řídká matice a nula by se nemusela ukládat vůbec. Pochopitelně by se s ní pak ani nepočítalo.

2 Pozor! Při implementaci v jazyce C půjdou indexy od nuly do n-1. To platí pro všechny zde popisované algoritmy.

Úkol č. 2

Napiš funkci `upravaMatice`, která upraví zadanou matici soustavy podle výše popsaného návodu.

```
void upravaMatice(Tmatice *m);
```

Výsledkem bude upravená matice soustavy s přepočítanými koeficienty a nulami na diagonále.

2.3 Gauss-Seidelova iterační metoda

Tato metoda funguje za podmínky, že vstupní matice soustavy je *diagonálně dominantní*. My navíc budeme pracovat s *upravenou maticí soustavy*.

Vstupem algoritmu je tedy upravená matice soustavy. Pro iterační výpočet pak můžeme s výhodou použít jednodušší vztah (8). Dalším vstupem je přesnost ε.

Výstupem bude jediné pole neznámých x, které na začátku nastavíme na nulové hodnoty. Bylo by sice možné použít náhodná čísla, ale jednak by bylo zbytečně náročné je generovat, ale hlavně by se algoritmus obtížně ladil.

Gauss-Seidelova iterační metoda se vyznačuje použitím jediného pole neznámých x. Prvky tohoto pole pak ve vztahu (8) figurují jak na levé, tak na pravé straně. Při výpočtu nové hodnoty neznámé x_i si nejprve musíme zapamatovat její předchozí hodnotu a bezprostředně poté testovat dosaženou přesnost ε.

Jeden krok algoritmu představuje vypočtení nových hodnot pro všech n neznámých. Na konci tohoto kroku je třeba vyhodnotit, zda už pro všechny neznámé bylo dosaženo zadané přesnosti. Pokud u jediného řádku toto nenastalo, musí výpočet pokračovat dalším krokem.

Finta s logickou proměnnou

Protože si tato metoda nepamatuje všechny hodnoty neznámých z minulého kroku, je třeba přesnost testovat průběžně. Použijeme k tomu jedinou logickou proměnnou, jejíž hodnotu budeme v rámci jednoho kroku průběžně aktualizovat pomocí logické operace AND.

Schéma takové finty je naznačeno v následujícím pseudokódu.

```
// na začátku pesimisticky
bool jePresny = false;

while (!jePresny) {
  // na začátku každé iterace
  // nastavit optimisticky
  jePresny = true;
  for (r: 0 → radku-1) {
    xpred ← x[r];
    x[r] ← nová hodnota podle (8);
    jePresny = jePresny &&
      | xpred - x[r] | < ε;
  }
}
```

Všimněte si, že když se proměnná `jePresny` uvnitř cyklu `for` jednou nastaví na `false`, už v tom stavu zůstane až do jeho konce. Tím se vynutí další iterace cyklu `while`, který si proměnnou `jePresny` na začátku zase nastaví na `true` a nechá cyklus `for` spočítat další sadu nových hodnot x. Teprve když bylo dosaženo zadané přesnosti ε pro všechny neznámé, cyklus while skončí a v poli x budou výsledky.

Úkol č. 3

Napiš funkci `resGS`, která vypočte zadanou soustavu Gauss-Seidelovou metodou.

```
void resGS(Tmatice *m, float eps,
           float x[]);
```

Vstupem funkce bude upravená matice soustavy a přesnost eps, výstupem pole x, délky m→radku. Funkce nebude nic tisknout.

Alternativně lze pro vektor x použít již hotový typ Tmatice, u nějž nastavíme počet sloupců na 1 (funkcí maticeAlokuj). Takovýto způsob je zvláště výhodný, když je typ vnitřně implementován jako dynamický datový typ.

```
void resGS(Tmatice *m, float eps,
           Tmatice *x);
```

Dále napiš funkci `testujGS`, která načte soustavu ze souboru (pomocí funkce z modulu matice.h `maticeCtiZeSouboru`), otestuje ji (`jeDDM`), pokud bude řešení konvergovat, upraví ji (`upravaMatice`), vypočítá řešení pomocí `resGS` a vytiskne výsledky. Jméno vstupního souboru a epsilon zjisti vhodným způsobem od uživatele.

```
void testujGS(void);
```

2.4 Jacobiho metoda

Tato metoda se od Gauss-Seidelovy metody liší jen málo. Místo jednoho pole s neznámými hodnotami x používá pole dvě: x^{pred} a x^{nove} pro uložení hodnot předchozího a aktuálního kroku. Předchozí hodnoty se teď budou vyskytovat na pravé straně rovnice, nové na levé straně. Na konci jedné iterace algoritmu (cyklus while) se pak novými hodnotami přepíší předchozí výsledky a pokračuje se dále.

Ačkoli by teď bylo možné testovat přesnost ε až na konci celé iterace, je to zbytečné. Dá se použít stejný postup s pomocnou logickou proměnnou, jako jsme použili u Gauss-Seidelovy metody.

Ačkoli má tato metoda větší paměťové nároky, protože potřebujeme další pomocné pole navíc, je pro moderní počítače výhodnější – dá se totiž paralelizovat. Cyklus for, který počítá nové hodnoty pro jednotlivé neznámé jde rozdělit do vláken a každá neznámá se může v rámci jedné iterace počítat v jednom vlákně. Pro opravdu rozsáhlé soustavy to může znamenat velké urychlení výpočtu. Při výpočtu Gauss-Seidelovou metodou toto udělat nejde.

Úkol č. 4

Napiš funkci `reseniJacobi`, která vypočte zadanou soustavu Jacobiho metodou.

```
void resJacobi(Tmatice *m, float eps,
               float x[]);
```

Vstupem funkce bude upravená matice soustavy a přesnost eps, výstupem pole x, délky m→radku. Funkce nebude nic tisknout. Druhé pomocné pole si funkce vyrobí sama.

Opět je možné pro vektor x použít typ Tmatice.

Dále napiš funkci `testujJacobi`, která načte soustavu ze souboru (pomocí funkce z modulu matice.h `maticeCtiZeSouboru`), otestuje ji (`jeDDM`), pokud bude řešení konvergovat, upraví ji (`upravaMatice`), vypočítá řešení pomocí `resJacobi` a vytiskne výsledky. Jméno vstupního souboru a epsilon zjisti vhodným způsobem od uživatele.

```
void testujJacobi(void);
```