PA081: Programování numerických výpočtů

9. Systémy lineárních rovnic a optimalizované implementace lineární algebry

Aleš Křenek

jaro 2013

Motivace

- hledáme řešení systému

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1N}x_N &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2N}x_N &= b_2 \\
&\dots \\
a_{M1}x_1 + a_{M2}x_2 + \cdots + a_{MN}x_N &= b_M
\end{aligned}
$$

- maticové vyjádření

$$\mathbf{Ax} = \mathbf{b}$$

- součást metod řešení nelineárních rovnic, optimalizací atd.
- diferenciální rovnice
  - simulace dynamických systémů
  - metoda konečných prvků
- realistické osvětlení scény (radiosita)
- a mnoho dalších …

Motivace

Simulace pohybu tělesa

- založena na Newtonově zákoně $F = ma$
- vede na systém diferenciálních rovnic (13 proměnných)

$$\frac{d\mathbf{y}(t)}{dt} = \begin{pmatrix} d\mathbf{x}/dt \\ d\mathbf{q}/dt \\ d\mathbf{P}/dt \\ d\mathbf{L}/dt \end{pmatrix} = \begin{pmatrix} \frac{1}{m}\mathbf{P} \\ \frac{1}{2}\omega_q \mathbf{q} \\ \mathbf{F} \\ \mathbf{T} \end{pmatrix}$$

- pro simulaci potřebujeme opakovaně řešit

$$\mathbf{y}_n = \mathbf{y}_{n-1} + \Delta t \left.\frac{d\mathbf{y}}{dt}\right|_{\mathbf{y}_n}$$

- triková aproximace – zanedbání vyšších členů Taylorova rozvoje $d\mathbf{y}/dt$ jako funkce $\mathbf{y}$
- konečný krok od $\mathbf{y}_{n-1}$ k $\mathbf{y}_n$ znamená řešení systému 13 lineárních rovnic

Motivace

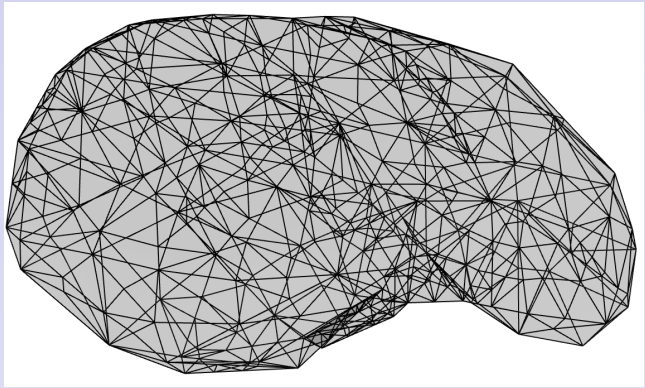
Modely měkkých tkání

- Interakce s měkkými tkáněmi
  - tréning chirurgů – vytvořit simulaci chování tkání
  - potřebujeme vytvořit matematický model tkáně
  - s touto tkání pak můžeme interagovat (např. hapticky)
  - je třeba simulovat chování tkáně s dostatečnou přesností
- použité matematické modely
  - deformovatelná tělesa
  - chování je popsáno parciálními diferenciálními rovnicemi, zpravidla neznáme analytické řešení
  - řešíme numericky pomocí diskretizace metodou konečných prvků (FEM)

Motivace

Metoda konečných prvků

- simulovaná tkáň je geometricky aproximována meshem



- deformace popisuje teorie elasticity
- geometricky lineární model – systém lineárních rovnic
- geometricky nelineární model – systém nelineárních rovnic
  - v každém kroku iterační metody systém lineárních rovnic
- systémy rovnic jsou řídké (ovlivňují se jen přímo spojené uzly)

Motivace

Rekonstrukce signálu z ultrazvuku

- pro vývoj filtrů rekonstruujících data z ultrazvuku je výhodné mít počítačový model přístroje a vyšetřovaného objektu
- snáze pak ladíme metody rekonstrukce, nevnášíme nepřesnost danou nedokonalým modelem vyšetřovaného objektu či nedokonalým přístrojem
- pro výpočet šíření vln v objektu použijeme FEM
  - na vlnovou délku ultrazvuku potřebujeme několik lineárních elementů
  - velikost elementu je pak v desetinách mm
  - vyšetřovaný objekt má velikost v jednotkách až desítkách centimetrech
- systémy o jednotkách až desítkách miliónů rovnic
  - vysoké nároky na výpočetní kapacitu
  - vysoké paměťové nároky

Klasifikace problémů

- $M = N$ – dobře podmíněné systémy
  - naděje na jednoznačné řešení (je-li $\mathbf{A}$ regulární)
- $M < N$ – nedostatečně podmíněné systémy
  - žádné řešení
  - nekonečně mnoho řešení ($N - M$-rozměrný prostor)
- $M > N$ – příliš podmíněné systémy
  - přesné řešení zpravidla neexistuje
  - hledáme „nejlepší kompromis“

Klasifikace problémů

- $M = N$ – dobře podmíněné systémy
  - naděje na jednoznačné řešení (je-li $\mathbf{A}$ regulární)
- $M < N$ – nedostatečně podmíněné systémy
  - žádné řešení
  - nekonečně mnoho řešení ($N - M$-rozměrný prostor)
- $M > N$ – příliš podmíněné systémy
  - přesné řešení zpravidla neexistuje
  - hledáme „nejlepší kompromis“

- husté vs. řídké
  - $O(N^2)$ vs. $O(N)$ nenulových prvků

Klasifikace problémů

- $M = N$ – dobře podmíněné systémy
  - naděje na jednoznačné řešení (je-li $\mathbf{A}$ regulární)
- $M < N$ – nedostatečně podmíněné systémy
  - žádné řešení
  - nekonečně mnoho řešení ($N - M$-rozměrný prostor)
- $M > N$ – příliš podmíněné systémy
  - přesné řešení zpravidla neexistuje
  - hledáme „nejlepší kompromis“

- husté vs. řídké
  - $O(N^2)$ vs. $O(N)$ nenulových prvků

- velikost – dopady kumulace chyby
  - $N \sim 50$ – zpravidla stačí `float`
  - N ~200–500 – `double`
  - větší – vyžadují speciální metody

Přehled metod

- přímé
  - daný algoritmus, vždy stejný počet operací
  - nemusí fungovat dobře pro „skoro singulární“ nebo příliš velké systémy
  - preferované pro „normální“ problémy
  - Gaussova eliminace, LU dekompozice, ...
- iterační
  - postupně vylepšované řešení
  - dosažení kritéria konvergence
  - různá náročnost pro různé vstupy
  - Jacobi, Gauss-Seidel, sdružené směry ...

Přehled metod

- specializované metody pro řídké matice
  - pro různé vzory řídkých matic
  - výrazně efektivnější, jediná možnost řešení velkých systémů
  - přímé i iterační
- metody pro singulární systémy
  - analyticky i numericky („skoro“) singulární
  - nalezení celého prostoru řešení
  - standardní metody zhavarují
  - dekompozice na singulární hodnoty (v příští přednášce)
- řešení příliš podmíněných systémů
  - specializované metody nejmenších čtverců

Dostupné knihovny

- zpravidla sáhneme k hotové knihovně
  - nejsme první, kdo tento problém řeší
  - implementace optimalizované pro různé případy
  - k volbě vhodné metody je třeba rámcově rozumět jejich principům
- Numerical Recipes in C
  - hlavní zdroj pro tuto přednášku
  - jednoduché přehledné implementace
- LAPACK `http://www.netlib.org/lapack/`
  - plnohodnotná volně dostupná knihovna
  - využívá nízkoúrovňové knihovny BLAS, zpravidla implementované strojově závisle
  - obecné a specializované funkce pro různé typické případy
- NAG `http://www.nag.co.uk/`
  - rozsáhlá komerční numerická knihovna
- a další ...

Gaussova eliminace

Základní postup

- standardní „školní“ metoda
  - řešení systému se nezmění, nahradíme-li libovolný řádek **A** a odpovídající prvek **b** lineární kombinací tohoto řádku s libovolným dalším
- vynulujeme $a_{21}, \ldots, a_{M1}$ odečtením $\frac{a_{i1}}{a_{11}}$ násobku prvního řádku
- obdobně pokračujeme druhým sloupcem od a_{32}
- výsledkem je matice s vynulovanými prvky pod diagonálou
- řešení systému získáme zpětnou substitucí
  - poslední rovnice $a_{MM}x_M = b_M$ je triviální
  - do předposlední dosadíme získanou hodnotu x_M atd.

Gaussova eliminace

Numerické problémy

- diagonální prvek a_{kk} je 0 nebo příliš malý
  - v k-té iteraci se jím dělí
  - dojde k přetečení
- při odečítání dochází k přílišné ztrátě platných cifer

$$\begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} \epsilon & 1 \\ 0 & 1 - \frac{1}{\epsilon} \end{bmatrix}$$

  zaokrouhlení může vést až k $a_{22} = -\frac{1}{\epsilon}$, to odpovídá původní matici

$$\begin{bmatrix} \epsilon & 1 \\ 1 & 0 \end{bmatrix}$$

- metoda v této podobě je numericky nestabilní

Gaussova eliminace

Pivoting

- další tvrzení o systému
  - řešení systému se nezmění výměnou libovolné dvojice řádků
  - řešení systému se nezmění výměnou libovolné dvojice sloupců, zaměníme-li zároveň příslušné komponenty vektoru $\mathbf{x}$
- pivot je prvek, který po aplikaci těchto kroků použijeme k dělení při eliminaci k-tého sloupce
- částečný pivoting (parciální)
  - v iteraci k vybíráme z a_{jk} pro $j \geq k$, tj. jen z nulovaného sloupce
- plný pivoting
  - vybíráme z a_{ij} pro $i, j \geq k$, tj. z celé podmatice doprava a dolů
  - vyžaduje udržovat vznikající permutaci proměnných

Gaussova eliminace

Pivoting

- za pivota volíme maximum z kandidátů
  - pro všechny faktory v eliminačních krocích platí $|\frac{a_{ik}}{a_{kk}}| \leq 1$
- parciální i plný pivoting jsou pak numericky stabilní
- přínos plného pivotingu je jen okrajový

Gaussova eliminace

Pivoting

- za pivota volíme maximum z kandidátů
  - pro všechny faktory v eliminačních krocích platí $|\frac{a_{ik}}{a_{kk}}| \leq 1$
- parciální i plný pivoting jsou pak numericky stabilní
- přínos plného pivotingu je jen okrajový
- volba pivota závisí na řádu koeficientů původního systému
  - vynásobení jedné rovnice faktorem 10^{10} ...
  - za pivota lze volit koeficient, který by byl největší, kdyby byl celý původní systém normalizovaný, tj. největší koeficient všech rovnic roven 1
  - vyžaduje dodatečnou údržbu faktorů škálování, může přinést větší robustnost

Gaussova eliminace

Gauss-Jordanova eliminace

- Gaussovu metodu lze použít pro více pravých stran současně
- speciálně pro N bázových vektorů dostaneme výpočet matice $\mathbf{A}^{-1}$
- rozšíření – Gauss-Jordanova eliminace
  - v každé iteraci eliminujeme prvky nad i pod diagonálou $\mathbf{A}$
  - výsledkem je jednotková matice
  - řešení systému je přímo modifikovaný vektor $\mathbf{b}$
  - resp. dostáváme $\mathbf{A}^{-1}$
- srovnatelné se základní implementací
  - reálně provádíme tytéž operace

Gaussova eliminace

Výhody a nevýhody

- jednoduchá, dobře pochopitelná a stabilní metoda
- je třeba znát pravé strany dopředu
  - jsou součástí celého výpočtu
- výpočet $\mathbf{A}^{-1}$ je zatížen poměrně velkou numerickou chybou
  - přímý výpočet řešení systému $\mathbf{Ax} = \mathbf{b}$ je přesnější než výpočet $\mathbf{A}^{-1}$ a následné $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$

Rozklady matic

- rozklad matice na vhodné činitele
  - řešení systému rovnic je pak triviální
- LU
  - čtvercové matice
  - de facto jiné vyjádření Gaussovy eliminace
  - explicitní vyjádření při více pravých stranách
- Choleského
  - čtvercové, symetrické, pozitivně definitní matice
  - stabilnější a rychlejší algoritmus
- QR
  - obecné $m \times n$
  - numerický stabilní, náročnější výpočet
- singulární hodnoty
  - náročnější výpočet, extrémně stabilní
  - dává přímo „řešení“ pro $M \leqslant N$

Iterační zpřesnění

- i přímé metody řešení lineárních rovnic ztrácí přesnost
  - v závislosti na velikosti systému a poměru koeficientů
  - kumulace chyb pocházejících ze sčítání/odčítání
  - v optimistickém případě 2–3 platná místa
  - u „skoro singulárních“ matic podstatně horší
- lze vylepšit iteračním procesem
- $\mathbf{x}$ je přesné řešení $\mathbf{Ax} = \mathbf{b}$, známe ale jen $\mathbf{x} + \delta\mathbf{x}$; položíme

$$\mathbf{A}(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b}$$

odečtením dostaneme

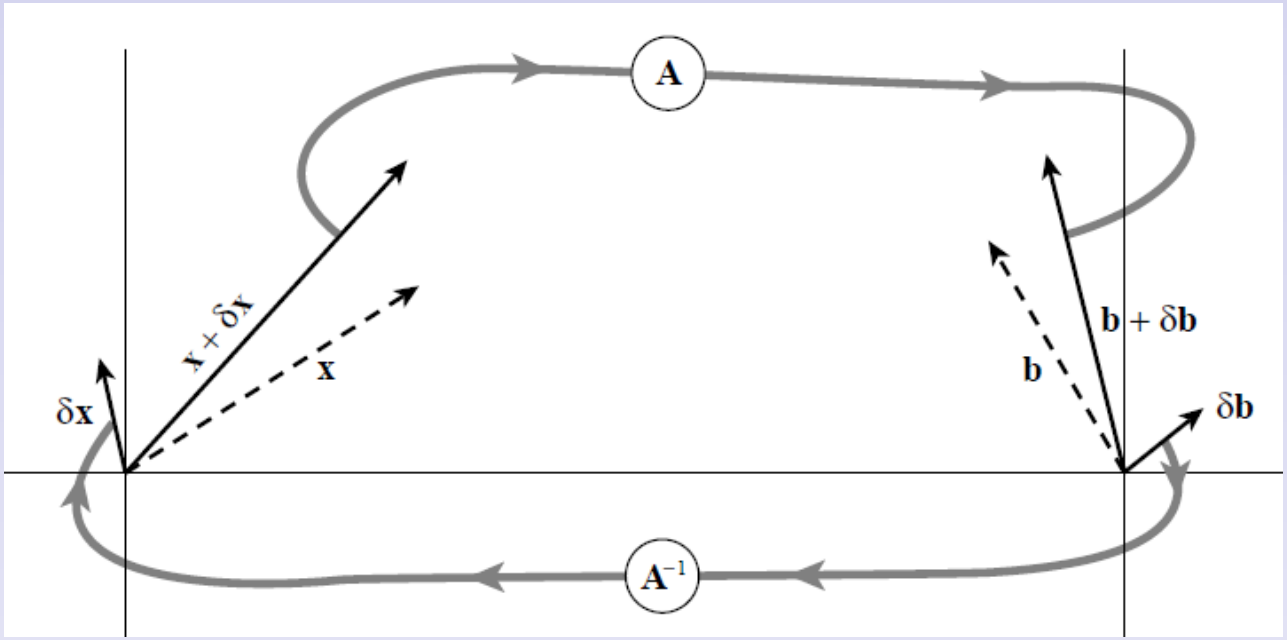
$$\mathbf{A}\delta\mathbf{x} = \delta\mathbf{b} = \mathbf{A}(\mathbf{x} + \delta\mathbf{x}) - \mathbf{b}$$

pravou stranu umíme spočítat, řešíme nový systém rovnic pro $\delta\mathbf{x}$

  - pro výpočet pravé strany je nutná vyšší přesnost odčítání
  - s výhodou využijeme recyklaci LU dekompozice
- výsledným $\delta\mathbf{x}$ korigujeme původní $\mathbf{x}$

Iterační zpřesnění

- schematické znázornění



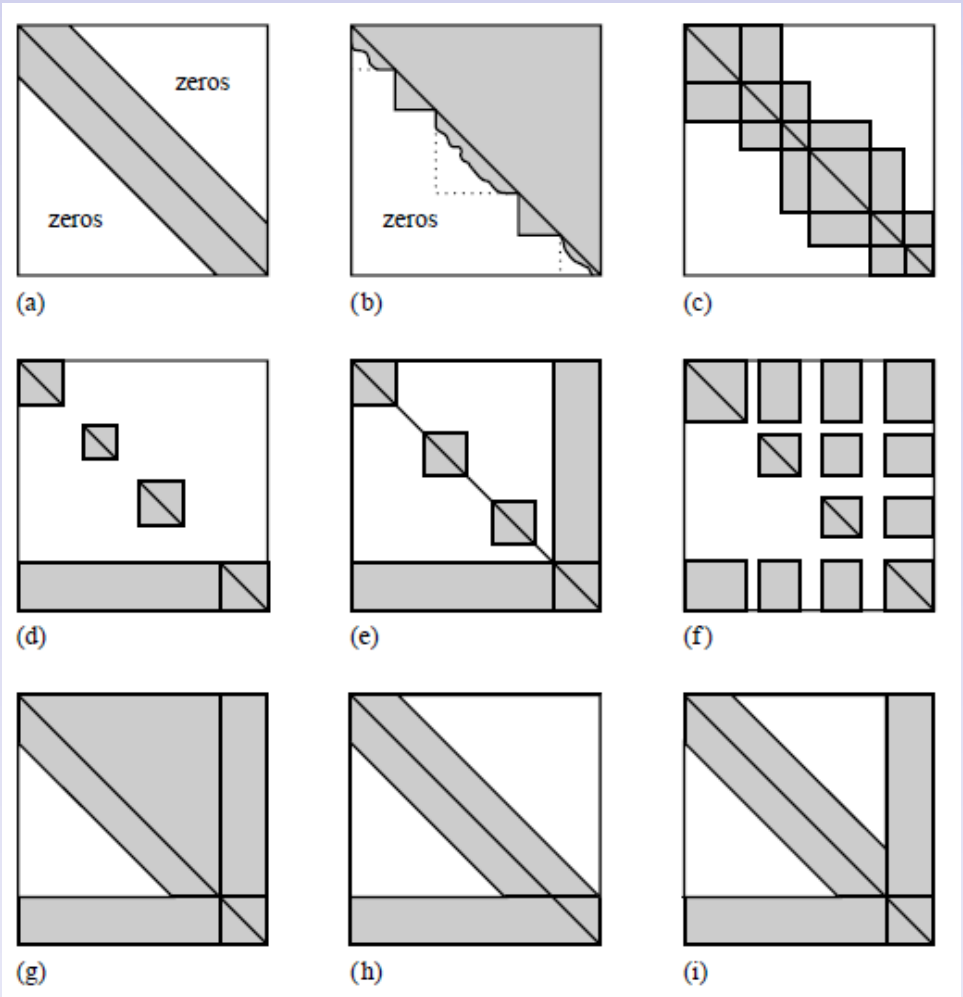
- postup lze opakovat, zpravidla stačí 1–2 krát

Řídké matice

- typicky rozsáhlé problémy (miliony rovnic)
  - ale počet nenulových koeficientů je malý, zpravidla $O(N)$
- v extrémním případě neřešitelné standardními metodami
  - příliš velká paměťová a časová náročnost
  - de-facto nepřekonatelné problémy s přesností výpočtu
- specializované metody
  - využívají nulových koeficientů
  - vyžadují jen $O(N)$ operací i paměti
  - závislé na konkrétním vzoru nenulových prvků
  - např. LU dekompozice tridiagonální matice – jen N prvkový vektor navíc a 2 cykly o $N - 1$ iteracích
- v některých případech aproximační
  - nenulové prvky se během řešení „množí“ nad míru
  - je třeba některé zanedbávat

Řídké matice

Některé speciální vzory



Řídké matice

Uložení v paměti

- problematická není jen výpočetní náročnost, ale zejména nároky na paměť
  - $N = 10^6$ by vyžadovalo ve floatech 4 TB
- existují různá schémata, např.
  - pole hodnot `float val[]` a indexů `int idx[]`
  - prvních N prvků `val[]` jsou všechny diagonální prvky, zpravidla bývají nenulové
  - prvních N prvků `idx[]` jsou indexy ve `val[]`, kde jsou uloženy první nenulové nediagonální prvky jednotlivých řádků matice
  - `idx[`$N+1$`]` ukazuje za poslední prvek `val[]`
  - další prvky `val[]` jsou nenulové nediagonální prvky matice uspořádané po řádcích a sloupcích
  - další prvky `idx[]` jsou indexy sloupců odpovídajících prvků `val[]`

Řídké matice

Uložení v paměti

- původní matice

$$\begin{bmatrix} 3 & 0 & 1 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 \\ 0 & 7 & 5 & 9 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 6 & 5 \end{bmatrix}$$

- je reprezentována

	1	2	3	4	5	6	7	8	9	10	11
`idx[]`	7	8	8	10	11	12	3	2	4	5	4
`val[]`	3	4	5	0	5	n/a	1	7	9	2	6

Řídké matice

Sherman-Morrisonův postup

- řídké matice, které se „trochu liší“ od známého vzoru
  - navíc řádek, sloupec apod.
  - obecně $\mathbf{A}' = \mathbf{A} + (\mathbf{u} \otimes \mathbf{v})$
- lze ukázat

$$(\mathbf{A}')^{-1} = (\mathbf{A} + (\mathbf{u} \otimes \mathbf{v}))^{-1} = \mathbf{A}^{-1} - \frac{(\mathbf{A}^{-1}\mathbf{u}) \otimes (\mathbf{v}\mathbf{A}^{-1})}{1 + \mathbf{v}\mathbf{A}^{-1}\mathbf{u}}$$

- některou z hotových metod vypočteme $\mathbf{A}^{-1}$
- aplikací vzorce získáme $(\mathbf{A}')^{-1}$
  - je-li $\mathbf{A}^{-1}$ řídká v řádu $O(N)$, náročnost celé metody je $O(N)$
- postup lze opakovat pro ruzná $\mathbf{u}, \mathbf{v}$
- existují další zobecnění (Woodbury, viz literatura)

Iterační metody

- obecně méně přesné než přímé metody
- řešení řídkých systémů
  - neexistuje přímá metoda pro konkrétní vzor
  - výpočet iteračního kroju je zpravidla $O(N)$
  - nevyžaduje další paměť
- téměř singulární systémy
  - přímé metody ztrácí přesnost kumulací chyby

Iterační metody

Prostá iterace

- rovnici $\mathbf{Ax} = \mathbf{b}$ převedeme na tvar $\mathbf{x} = \mathbf{A}'\mathbf{x} + \mathbf{b}'$
- opakovaně počítáme iterační krok
- kritérium konvergence
  - zobrazení $\mathbf{x} \mapsto \mathbf{A}'\mathbf{x} + \mathbf{b}'$ musí být kontrakce
  - stejné jako u nelineárních rovnic
- naplnění předpokladů věty o pevném bodě
  - $\lim_{k\to\infty} |(\mathbf{A}')^k| = 0$ pro nějakou maticovou normu
  - Frobeniova norma
    $$|\mathbf{A}| = \sqrt{\sum_{i,j} a_{ij}^2}$$
  - spektrální norma
    $$|\mathbf{A}| = \rho(\mathbf{A}^T\mathbf{A})$$
    kde $\rho()$ je maximální absolutní hotnota vlastních hodnot matice
  - maximální součet sloupce nebo řádku

Iterační metody

Prostá iterace – příklady konkrétních metod

- Jacobi: z rozkladu $\mathbf{A} = \mathbf{D} - \mathbf{L} - \mathbf{U}$ dostaneme $\mathbf{x} = \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{x} + \mathbf{D}^{-1}\mathbf{b}$, tj.

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1, j \neq i}^{n} a_{ij} x_j^{(k)} \right)$$

- Gauss-Seidel: $\mathbf{x} = (\mathbf{D} - \mathbf{L})^{-1}\mathbf{U}\mathbf{x} + (\mathbf{D} - \mathbf{L})^{-1}\mathbf{U}\mathbf{b}$

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right)$$

  lze recyklovat uložení $\mathbf{x}$
- konvergence není zaručena automaticky
  - pro konkrétní $\mathbf{A}$ některé metody konvergují, jiné ne
  - specifická kritéria viz literatura

Iterační metody

Metoda konjugovaných gradientů

- lze využít i metody řešení nelineárních rovnic
  - smysl to má jen ve velmi specifických případech
  - výjimkou je metoda konjugovaných gradientů
- minimalizujeme funkci

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}\mathbf{A}\mathbf{x} - \mathbf{b}\mathbf{x}$$

- v minimu je její gradient nulový, tj.

$$0 = \nabla f = \mathbf{A}\mathbf{x} - \mathbf{b}$$

  tedy minimum f přesně odpovídá řešení $\mathbf{A}\mathbf{x} = \mathbf{b}$
- metoda tedy najde minimum po N iteracích
  - f je kvadratická forma, viz minulá přednáška
- takto jednoduše funguje jen pro pozitivně definitní $\mathbf{A}$, lze rozšířit
- při výpočtu je třeba jen násobit $\mathbf{A}\mathbf{x}$
  - to lze u řídkých matic velmi efektivně

Lineární algebra rychle

- proč má smysl optimalizovat právě algoritmy lineární algebry?
- frekventované problémy
- zdánlivě jednoduché algoritmy (násobení matic) lze výrazně vylepšit
- problematika se stále vyvíjí
- naznačíme směry, kterými se lze vydat
- rafinované algoritmy už jednou někdo vymyslel a implementoval
  - nemá smysl učit se je nazpaměť
  - stačí o nich vědět a rozumět základní myšlence

Vyrovnávací paměti

- hlavní paměť je řádově pomalejší než procesor
  - rychlou paměť v plné velikosti je technicky/finančně nemožné vyrobit
- hierarchie vyrovnávacích pamětí (cache)
- pro Intel Nehalem/Westmere
  - L1 – plná rychlost, 32 kB instrukce, 32 kB data/jádro
  - L2 – latence 10 cyklů, 256 kB/jádro
  - L3 – latence 40 cyklů, 8 MB/procesor (sdílená mezi jádry)

Vyrovnávací paměti

Organizace přístupu

- řádky (cache-line)
  - typicky 64 B (16× `float`, 8× `double`)
- přímo mapovaná cache
  - blok dat z hlavní paměti má dán jeden řádek v cache

    (adresa/velikost řádku) % počet řádků
  - snadno dojde ke kolizím, např. pro 32 kB

```
double pole[100][512][8];
for (i=0; i<100; i++) a += pole[i][0][0];
```

  - řešením je deklarace `pole[100][513][8]`

Vyrovnávací paměti

Organizace přístupu

- plně asociativní
  - blok dat z hlavní paměti může být umístěn v kterémkoli řádku
  - implementačně náročné, ve standardních CPU se nepoužívá
- n-cestná asociativní
  - kompromisní řešení
  - sady po n řádcích
  - blok z hlavní paměti může skončit v kterémkoli řádku sady
  - Intel Nehalem/Westmere: $n = 8$

Vyrovnávací paměti

Praktické zásady

- optimalizovat pro všechny úrovně
- využít celý řádek
  - např. může se vyplatit transponovat matici
- dovolit procesoru/kompilátoru současně přenášet data a počítat
  - dobře čitelný kód bez možných vedlejších efektů
- zabránit předčasným kolizím v jedné sadě řádků
- měřit dosažený výkon (flop/s)
- atd. . . .

Vyrovnávací paměti

Praktické zásady

- optimalizovat pro všechny úrovně
- využít celý řádek
  - např. může se vyplatit transponovat matici
- dovolit procesoru/kompilátoru současně přenášet data a počítat
  - dobře čitelný kód bez možných vedlejších efektů
- zabránit předčasným kolizím v jedné sadě řádků
- měřit dosažený výkon (flop/s)
- atd. . . .
- aplikovat jen pro skutečně kritické sekce kódu
- používat speciální optimalizované knihovny, kde to jde

Blokové algoritmy

- zjednodušený model jen s L1
- násobení vektoru maticí $\mathbf{y} := \mathbf{y} + \mathbf{A}\mathbf{x}$

načti $\mathbf{x}$ do cache
načti $\mathbf{y}$ do cache
`for` $i = 1, \ldots, n$
    načti řádek A_{i*} do cache
    `for` $j = 1, \ldots, n$
        $y_i = y_i + A_{ij} x_j$
zapiš $\mathbf{y}$ do hlavní paměti

- počet přístupů do pomalé paměti $m = 3n + n^2$
- počet aritmetických operací $f = 2n^2$
- efektivita $f/m \approx 2$
- algoritmus je limitovaný rychlostí paměti
  - pomůže recyklace řádků cache – vyplatí se ukládat vektory spojitě, ne jako sloupce matice (v C)
  - jinak se s tím nedá nic moc dělat

Blokové algoritmy

Maticové násobení

- násobení matic $\mathbf{C} = \mathbf{C} + \mathbf{AB}$

```
for i = 1,...,n
    načti řádek A_{i*} do cache
    for j = 1,...,n
        načti C_{ij} do cache
        načti sloupec B_{*j} do cache
        for k = 1,...,n
            C_{ij} = C_{ij} + A_{ik}B_{kj}
        zapiš C_{ij} do hlavní paměti
```

- přístupy do paměti $m = n^2 + n^3 + 2n^2$
- aritmetické operace $f = 2n^3$
- efektivita opět ≈ 2

Blokové algoritmy

Maticové násobení

- matice rozdělíme na N^2 bloků velikosti $b \times b, b = n/N$

```
for i = 1,...,N
    for j = 1,...,N
        načti blok C_ij do cache
        for k = 1,...,N
            načti blok A_ik do cache
            načti blok B_kj do cache
            C_ij = C_ij + A_ik B_kj (maticové násobení)
        zapiš blok C_ij do cache
```

- přístupy do paměti
  - čtení bloků **A**: $N^3 (n/N)^2 = N * n^2$
  - dtto **B**: $N * n^2$
  - čtení a zápis **C**: $2n^2$
- efektivita $f/m = \frac{2n^3}{(2N+2)n^2} \approx b$

Blokové algoritmy

Maticové násobení

- velikost reálné L1 cache je 32 kB,
- do L1 se vejdou 3 matice velikosti 36×36 (v `double`)
- L2 cache je $10\times$ pomalejší
- procesor je dobře využit
  - i při využití vektorových instrukcí
  - proto je cache právě tak velká
- žádoucí aplikovat rekurzivně i pro L2 a L3
- implementováno v optimalizovaných knihovnách
- násobení matic je na současných procesorech jeden z nejefektivnějších algoritmů
  - redukce problému na mat. násobení při řádovém zachování počtu operací téměř vždy vede k výraznému zrychlení

Knihovna BLAS

- *Basic Linear Algebra Subprograms*, `http://www.netlib.org/blas`
- základní operace, např.
  - `DOT`: $\mathbf{xy}$
  - `AXPY`: $\mathbf{y} + \mathbf{Ax}$
  - `NRM2`: $|\mathbf{x}|^2$
  - `TRMV`: $\mathbf{Ax}$
  - `TRSV`: $\mathbf{A}^{-1}\mathbf{x}$
  - `GEMM`: $\beta\mathbf{C} + \alpha\mathbf{AB}$
- verze pro typy `float`,`double`,`complex`
- specializované varianty pro symetrické, trojúhelníkové, a některé řídké matice
- de-facto standardizované rozhraní
- implementace vyladěné pro konkrétní typy CPU
- využívané v knihovnách vyšší úrovně (např. LU dekompozice)

Asymptotická složitost

- časová složitost násobení matic je $O(n^3)$

Asymptotická složitost

- časová složitost násobení matic je $O(n^3)$
- Strassen (1969): $O(n^{2.81})$
- rozdělení matic na 2×2 bloky, potom

$$\mathbf{M}_1 = (\mathbf{A}_{11} + \mathbf{A}_{22})(\mathbf{B}_{11} + \mathbf{B}_{22})$$
$$\mathbf{M}_2 = (\mathbf{A}_{21} + \mathbf{A}_{22})\mathbf{B}_{11}$$
$$\mathbf{M}_3 = \mathbf{A}_{11}(\mathbf{B}_{12} + \mathbf{B}_{22})$$
$$\mathbf{M}_4 = \mathbf{A}_{22}(\mathbf{B}_{21} - \mathbf{B}_{11})$$
$$\mathbf{M}_5 = (\mathbf{A}_{11} + \mathbf{A}_{12})\mathbf{B}_{22}$$
$$\mathbf{M}_6 = (\mathbf{A}_{21} - \mathbf{A}_{11})(\mathbf{B}_{11} + \mathbf{B}_{12})$$
$$\mathbf{M}_7 = (\mathbf{A}_{12} - \mathbf{A}_{22})(\mathbf{B}_{21} + \mathbf{B}_{22})$$

$$\mathbf{C}_{11} = \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7$$
$$\mathbf{C}_{12} = \mathbf{M}_3 + \mathbf{M}_5$$
$$\mathbf{C}_{21} = \mathbf{M}_2 + \mathbf{M}_4$$
$$\mathbf{C}_{22} = \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6$$

- rekurzivní aplikace, počet operací $T(n)$

$$T(n) = 7T(n/2) + 18(n/2)^2 = O(n^{\log_2 7}) = O(n^{2.81})$$

Asymptotická složitost

- časová složitost násobení matic je $O(n^3)$
- Strassen (1969): $O(n^{2.81})$
- rozdělení matic na 2×2 bloky, potom

$$\mathbf{M}_1 = (\mathbf{A}_{11} + \mathbf{A}_{22})(\mathbf{B}_{11} + \mathbf{B}_{22})$$
$$\mathbf{M}_2 = (\mathbf{A}_{21} + \mathbf{A}_{22})\mathbf{B}_{11}$$
$$\mathbf{M}_3 = \mathbf{A}_{11}(\mathbf{B}_{12} + \mathbf{B}_{22})$$
$$\mathbf{M}_4 = \mathbf{A}_{22}(\mathbf{B}_{21} - \mathbf{B}_{11})$$
$$\mathbf{M}_5 = (\mathbf{A}_{11} + \mathbf{A}_{12})\mathbf{B}_{22}$$
$$\mathbf{M}_6 = (\mathbf{A}_{21} - \mathbf{A}_{11})(\mathbf{B}_{11} + \mathbf{B}_{12})$$
$$\mathbf{M}_7 = (\mathbf{A}_{12} - \mathbf{A}_{22})(\mathbf{B}_{21} + \mathbf{B}_{22})$$

$$\mathbf{C}_{11} = \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7$$
$$\mathbf{C}_{12} = \mathbf{M}_3 + \mathbf{M}_5$$
$$\mathbf{C}_{21} = \mathbf{M}_2 + \mathbf{M}_4$$
$$\mathbf{C}_{22} = \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6$$

- rekurzivní aplikace, počet operací $T(n)$

$$T(n) = 7T(n/2) + 18(n/2)^2 = O(n^{\log_2 7}) = O(n^{2.81})$$

- Coppersmith-Winograd (1987): $O(n^{2.376})$

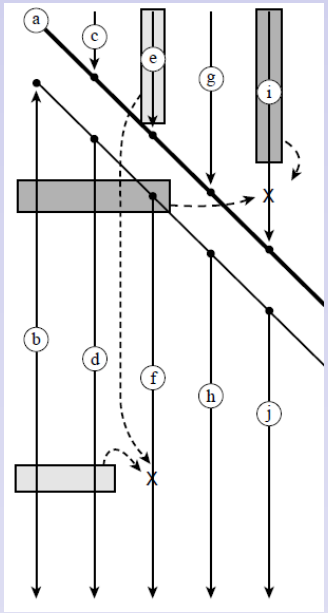
Domácí úkol

- implementujte co nejefektivněji Strassenův algoritmus
- od určité velikosti matice přepněte na BLAS
- vyhodnoťte chování (kdy začne být výhodnější, co to stojí)

LU dekompozice

- rekurzivní formulace algoritmu

$$\left(\begin{array}{c|c|c} \mathbf{A}_{00} & \mathbf{a}_{01} & \mathbf{A}_{02} \\ \hline \mathbf{a}_{10}^T & \alpha_{11} & \mathbf{a}_{12}^T \\ \mathbf{A}_{20} & \mathbf{a}_{21} & \mathbf{A}_{22} \end{array}\right)$$



- skalární a bloková verze

$$\mathbf{a}_{21} = \mathbf{a}_{21} / \alpha_{11}$$
$$\mathbf{a}_{12}^T \quad \text{zůstává}$$
$$\mathbf{A}_{22} = \mathbf{A}_{22} - \mathbf{a}_{21}\mathbf{a}_{12}$$

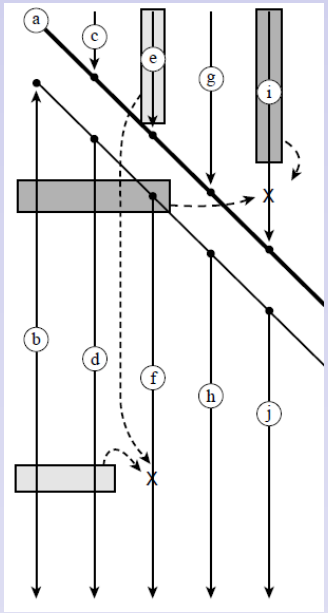
$$\left(\frac{\mathbf{A}_{11}}{\mathbf{A}_{21}}\right) = LU\left(\frac{\mathbf{A}_{11}}{\mathbf{A}_{21}}\right)$$
$$\mathbf{A}_{12} = \mathbf{L}_{11}^{-1}\mathbf{A}_{12}$$
$$\mathbf{A}_{22} = \mathbf{A}_{22} - \mathbf{L}_{21}\mathbf{A}_{12}$$

- podstatná část operací je násobení matic
- takto implementuje knihovna LAPACK

LU dekompozice

- rekurzivní formulace algoritmu

$$\left(\begin{array}{c|cc} \mathbf{A}_{00} & \mathbf{a}_{01} & \mathbf{A}_{02} \\ \hline \mathbf{a}_{10}^T & \alpha_{11} & \mathbf{a}_{12}^T \\ \mathbf{A}_{20} & \mathbf{a}_{21} & \mathbf{A}_{22} \end{array}\right)$$



- skalární a bloková verze

$$\mathbf{a}_{21} = \mathbf{a}_{21} / \alpha_{11}$$
$$\mathbf{a}_{12}^T \quad \text{zůstává}$$
$$\mathbf{A}_{22} = \mathbf{A}_{22} - \mathbf{a}_{21}\mathbf{a}_{12}$$

$$\left(\frac{\mathbf{A}_{11}}{\mathbf{A}_{21}}\right) = LU\left(\frac{\mathbf{A}_{11}}{\mathbf{A}_{21}}\right)$$
$$\mathbf{A}_{12} = \mathbf{L}_{11}^{-1}\mathbf{A}_{12}$$
$$\mathbf{A}_{22} = \mathbf{A}_{22} - \mathbf{L}_{21}\mathbf{A}_{12}$$

- podstatná část operací je násobení matic
- takto implementuje knihovna LAPACK
- problém algoritmu – „dlouhé“ matice $\mathbf{A}_{21}$ a $\mathbf{A}_{12}$
- Quintana et. al (2008): algoritmus s bloky $2p \times p$

Paralelní blokové algoritmy

- paralelismus na úrovni BLAS
  - existují optimalizované implementace
  - implicitní synchronizace na konci každého volání
  - nemusí být dosažitelný optimální výkon
- blokové operace jsou efektivní provedené vcelku
  - všechna data se dostanou naráz do cache
  - na úrovni L1/L2 se vyplatí na jednom jádru CPU
- vzájemně nezávislé blokové operace na více jádrech

Paralelní blokové algoritmy

- Quintana et. al (2008) – prototypová implementace SuperMatrix
  - konkrétní algoritmus proběhne „abstraktně“
  - blokové operace s deklarovanými závislostmi se pouze zařadí do fronty
  - následně se vyhodnotí pořadí zpracování
  - operace se provedou potenciálně paralelně
- čitelná formulace algoritmu bez explicitního paralelismu
- efektivní provedení
  - matice $n > 5000$
  - 16 jader CPU
  - LU dekompozice na více než 50 % teoretického výkonu

Vektorové instrukce

- historie – např. Cray v předsálí budovy D
- současné vektorové systémy (např. NEC SX)
  - řešení velmi specializovaných problémů
  - nízký výkon na skalárních operacích
- vektorová rozšíření v architektuře x86
  - MMX: pouze celá čísla, kolize s float registry
  - SSE: 8 (16) registrů po 128 bitech, postupně přidávané instrukce (rsqrt)
  - AVX: 256 bitové registry, 3-operandové instrukce
- snažší využití plného výkonu procesoru

Vektorové instrukce

- triviální příklad

```
        float a[4],b[4];
        int     i;
        for (i=0; i<4; i++) a[i] += b[i];
...
        movaps    32(%rsp), %xmm0
        addps     (%rsp), %xmm0
        movaps    %xmm0, 32(%rsp)
```

- vektorizaci kódu provede chytřejší kompilátor (icc)
  - musíme hlídat, abychom tomu nebránili
  - recyklace proměnných, vedlejší efekty in-line funkcí, ...
  - vyplatí se kontrola vygenerovaného assembleru

Využití GPU

- téměř vše, co platí o cache, platí také o GPU
  - rychlá paměť omezené velikosti
  - netriviální režie kopírování z/do hlavní paměti
- paralelní kód
  - daleko masivnější (desítky až stovky)
- viz PV197: GPU Programming

Shrnutí

- řešení lineárních rovnic je součást řady problémů
- v operacích LA se tráví velká část výpočetního času
- má smysl hledat optimalizované a numericky stabilní algoritmy
- neexistuje univerzální řešení
- k dispozici řada optimalizovaných implementací