Kodoptimering 1

Problem:

Beräkna $u = u(x,t)$ s.a. $u = u_0$ för $t = 0$, och

$$\dot{u} - \nabla \cdot a\,\nabla u = f \quad \text{i } \Omega,$$

och

$$-a\,\partial_n u = \gamma\,(u - g_G) + g_N \quad \text{på } \Gamma,$$

för $t > 0$, där $u_0 = u_0(x)$, $a = a(x,t)$, $f = f(x,t)$, $\gamma = \gamma(x,t)$, $g_D = g_D(x,t)$ och $g_N = g_N(x,t)$ är givna data i området Ω resp. på randen Γ.

Kodoptimering 2

Diskret problem:

Efter lämplig *diskretisering* av tid och rum erhålls ett motsvarande ekvationssystem för $U_n(x) \approx u(x,\, t_n)$:

$$\underbrace{(M + k_n\,A + k_n\,K)}_{C_n}\,U_n = \underbrace{M\,U_{n-1} + k_n\,F_n + k_n\,G_n}_{b_n},$$

där k_n är tidssteget, M är *massmatrisen*, A är *diffusionsmatrisen*, K är *rand-diffusionsmatrisen*, F_n är *lastvektorn*, och G_n en *"randlastvektor"*.

Notera att vektorn F_n här är beräknad utgående från *tidsmedelvärdet* $\frac{1}{k_n}\int_{I_n}\int_\Omega v\,f$ snarare än $\int_{I_n}\int_\Omega v\,f$ som tidigare. Därav k_n-faktorn! Analogt för G_n.

Kodoptimering 3

Resulterande kod: (med p, t, e givna)

```
u = initialData(p);     % begynnelsedata
time = 0;
finaltime = 1;
dt = .1;                % tidssteg
while time < finaltime
 time = time + dt;
 [M, A, K, F, G] = MyAssembler(p,t,e,time);
 C = (M + dt*A + dt*K);
 b = M*u + dt*F + dt*G;
 u = C \ b;             % ger U_n
end
```


Kodoptimering 4

Tidsoberoende data

Vi noterar att massmatrisen M, med element $\int_\Omega \phi_i\,\phi_j$, borde beräknas *innan* man går in i tidsloopen, eftersom den är *tidsoberoende*, av kostnadsskäl (varför räkna om samma sak många gånger?). Analogt, om data a är tidsoberoende, dvs $a = a(x)$, så kan motsvarande matris A med fördel beräknas *utanför tidsloopen*, vilket sparar ytterligare räknearbete. Detsamma gäller förstås även övriga data γ, f och g och motsvarande matris/vektorer.

Kodoptimering 5

“Separabla” rums-tids-data:

Även för *tidsberoende* data av typen $f(x,t) = f_1(t)\, f_2(x)$ kan man med fördel beräkna lastvektorn motsvarande faktorn $f_2(x)$ utanför tidsloopen, varvid beräkningen av den slutliga tidsberoende lastvektorn, inne i tidsloopen, reduceras till att evaluera tidsfunktionen $f_1(t)$, och multiplicera dess värde med den prefabriserade lastvektorn motsvarande f_2.

Kodoptimering 6

Exempel:

Betrakta t.ex. ekvationen $\dot{u} - \nabla \cdot \nabla u = 0$ med randvillkoret $-\partial_n u = \gamma(x)\,(u - \sin(t))$, (som med lämpligt valda enheter kan tänkas modellera temperaturvariationerna i vårt “hus” med dygns- eller årsvis periodiskt varierande yttertemperatur $g_D = g_1(t)\, g_2(x)$ med $g_1(t) = \sin(t)$ och $g_2(x) = 1$, dvs vi kan här beräkna samtliga ingående matriser och vektorer, så när som på tidsfaktorn $\sin(t)$, innan vi går in i tidsloopen. Den resulterande koden blir då ..

Kodoptimering 7

Resulterande kod:

```
u = initialData(p);     % begynnelsedata
time = 0;
[M, A, K, F, Gx] = MyAssembler(p,t,e,time);
finaltime = 1;
dt = .1;                % tidssteg
while time < finaltime
 time = time + dt;
 C = (M + dt*A + dt*K);
 G = sin(time)*Gx;
 b = M*u + dt*F + dt*G;
 u = C \ b;             % ger U_n
end
```


Kodoptimering 8

Effektiviserad ekvationslösning:

Tidsoberoende data ger alltså utrymme för att assemblera “en gång för alla”.

Är det även möjligt att lösa det ekvationssystem $C\,x = b$ som resulterar i varje tidssteg, “en gång för alla”, på något sätt?

Kodoptimering 9

LU-faktorisering

Gauss eliminationsmetod har två steg; först reduceras koefficientmatrisen C till "uppåt trangulär" form U:

$$\underbrace{\begin{bmatrix} 1 & 2 & -1 \\ 2 & 3 & -3 \\ -2 & 0 & 4 \end{bmatrix}}_{C} \rightarrow \begin{bmatrix} 1 & 2 & -1 \\ 0 & -1 & -1 \\ 0 & 4 & 2 \end{bmatrix} \rightarrow \underbrace{\begin{bmatrix} 1 & 2 & -1 \\ 0 & -1 & -1 \\ 0 & 0 & -2 \end{bmatrix}}_{U}$$

Kan ses som att C multipliceras med

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 4 & 1 \end{bmatrix}}_{M_2} \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix}}_{M_1}$$

Kodoptimering 10

Noterar att matriserna M_1 och M_2 är "nedåt triangulära", och att detsamma gäller deras produkt M:

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 4 & 1 \end{bmatrix}}_{M_2} \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix}}_{M_1} = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -6 & 4 & 1 \end{bmatrix}}_{M}$$

Eftersom M är triangulär, och inte har någon nolla på diagonalen, är matrisen inverterbar med invers $L = M^{-1}$, och även inversen måste vara nedåt triangulär med ettor på diagonalen.

Kodoptimering 11

Vid Gausseliminationen uppdateras samtidigt högerledet på samma sätt, dvs genom multiplikation med $M = M_2\, M_1$:

$$\underbrace{\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}}_{b} \rightarrow \underbrace{\begin{bmatrix} 1 \\ 0 \\ 5 \end{bmatrix}}_{M_1\, b} \rightarrow \underbrace{\begin{bmatrix} 1 \\ 0 \\ 5 \end{bmatrix}}_{M_2\, M_1\, b = M\, b}$$

Kodoptimering 12

Dvs vid Gausselimination *faktoriseras* koefficientmatrisen C som $L\,U$, varvid ekvationssystemet $L\,U\,x = b$ löses i tvåsteg; först beräknas $U\,x$, dvs lösningen $y = M\,b = L^{-1}\,b$ till $L\,y = b$, och sedan beräknas lösningen $x = U^{-1}\,y$ till det reducerade systemet $U\,x = y$. Båda dessa lösningssteg är triviala eftersom koefficientmatriserna L och U är triangulära! Kostnaden ligger i faktoriseringen av C som $L\,U$.

Kodoptimering 13

En slutsats man kan dra av detta är att om man har behov av att lösa ekvationssystemet $C\,x = b$ med flera olika högerled b, med samma C, så bör man återanvända den LU-faktorisering man defacto beräknar första gången, varvid lösandet av $C\,x = L\,U\,x = b$ för de kommande systemen reduceras till lösning av två system, nämligen $L\,y = b$ och sedan $U\,x = y$ med triangulära matriser.

Denna situation föreligger t.ex. vid succesiv tidsstegning med tidsoberoende data i matriserna M, A och K, och konstant tidssteg $k_n = k$.

Kodoptimering 14

Matlabs funktion för LU-faktorisering (en variant av den som här beskrivits) heter lu (se help lu).

Ett naturligt alternativ till LU-faktorisering av C, om man står inför att behöva lösa $C\,x = b$ med flera olika högerled b, skulle kunna tänkas vara att helt enkelt beräkna *inversen* C^{-1} till C, "en gång för alla", och sedan med matris-vektormultiplikation beräkna $C^{-1}\,b$ för de olika högerledsvektorerna b. Något överraskande är dock metoden med LU-faktorisering i regel mera kostnadseffektiv!

Den resulterande koden blir alltså, för exemplet ovan,..

Kodoptimering 15

Resulterande kod:

```
u = initialData(p);     % begynnelsedata
time = 0;
[M, A, K, F, Gx] = MyAssembler(p,t,e,time);
finaltime = 1;
dt = .1;                % tidssteg
C = M + dt*A + dt*K;
[L, U] = lu(C);
while time < finaltime
 time = time + dt;
 G = sin(time)*Gx;
 b = M*u + dt*F + dt*G;
 y = L \ b;
 u = U \ y;             % ger U_n = u
end
```


Kodoptimering 16

Glesa matriser

Vi har noterat att finita elementmatriserna M, A, K osv är *glesa*, dvs att de flesta matriselementen är noll. Det är förstås dumt både att *lagra* alla dessa nollor, och att "*räkna om*" dom vid Gausselimination. T.ex. får ju diffusionsmatrisen för det endimensionella problemet

$$-u'' = f \quad 0 < x < 1, u(0) = 0,\ u(1) = 0,$$

utseendet

$$6\begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 \end{bmatrix}$$

Kodoptimering 17

Matlab har därför en speciell lagringsform för sådana "glesa" matriser, liksom speciella rutiner för att räkna med sådana. Se help sparse!
En gles matris *lagras* m.h.a. en "index lista", med rad/kolonn nummer, och en motsvarande lista på motsvarande nollskilda matriselement. T.ex. lagras matrisen ovan som

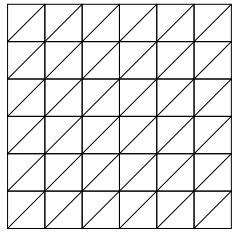
(1,1) 1
(1,2) -1
(2,1) -1
(2,2) 2
(2,3) -1
(3,2) -1
.. osv.

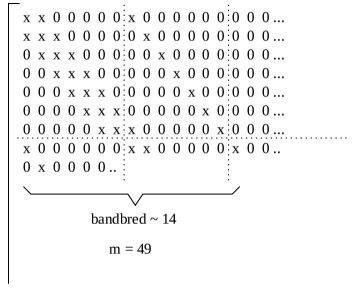
Kodoptimering 18

För de "tridiagonala" fem-matriser som är typiska för en-dimensionella problem krävs sedan endast c:a $10\,m$ räkneoperationer för Gausseliminationen, dvs för att lösa ekvationssystemet ifråga, där m är antalet obekanta.

Kodoptimering 19

Motsvarande två -dimensionella problem får i regel en betydligt större s.k. *bandbredd*, dvs "radindexbredd" på det "band" kring diagonalen i matrisen inom vilket alla nollskilda element kan huseras.





Kodoptimering 20

Eftersom Gausseliminationen berör hela detta band (då den tyvärr, men med nödvändighet, "förstör" många till synes "färdiga" nollor på sin väg) blir antalet räkneoperationer här betydligt större, typiskt av storleksordning $d^2\,m$ där d är bandbredden och m, som förut, antalet obekanta. För tvådimensionella problem är bandbredden d typiskt av storleksordning $2\,\sqrt{m}$, dvs antalet räkneoperationer blir av storleksordning $4\,m^2$.

För 3-dimensionella problem blir bandbredden ännu större, typiskt av storleksordning $2\,m^{2/3}$, och antalet räkneoperationer därför av storleksordning $4\,m^{7/3}$.

Kodoptimering 21

Exempel: I 3 rumsdimensioner och ett kubiskt område med sidlängd 1 (m) och med element av storleksordning 1/10 (m) blir antal obekanta 10e3 och antal räkneoperationer för att lösa motsvarande ekvationssystem med Gauss elimination blir av storleksordning 10e7, i varje tidssteg ! För att följa ett transient (tidsberoende) förlopp med t.ex. 10 tidssteg krävs alltså 10e8 räkneoperationer. Med en dator som kan klara 10e4 räkneoperationer per sekund tar detta 10e4 sekunder, dvs c:a 3 timmar.

Med modern s.k. *multigrid teknik* kan antalet räkneoperationer för att lösa resulterande ekvationssystem i princip reduceras till att bli (direkt) proportionellt mot antalet obekanta m, dvs 10e3 i exemplet. 10 tidsteg skulle då ta endast 1 sekund.

Kodoptimering 22

For loopar:
Implementeringen av (for)loopar i Matlab är känd för att vara förhållandevis ineffektiv. Om möjligt skall man därför försöka undvika sådana genom att använda Matlabs inbyggda matristilldelningssyntax när så är möjligt, vilket kan ge en både kompaktare och betydligt snabbare kod. Vissa loopar är dock oundgängliga, som t.ex. tidsloop och assembleringloopar.

Däremot är det naturligt att använda Matlabs eleganta matrishanteringssyntax t.ex. vid beräkning av de lokala mass- och styvhets/diffusionsmatriserna, och motsvarande lokala bidrag till lastvektorerna, liksom vid “utplaceringen” av dessa bidrag i motsvarande “globala” matriser och vektorer.

Kodoptimering 23

Exempel: T.ex. kan koden för beräkning och assemblering av den lokala styvhetsmatrisen

$$\begin{bmatrix} \int \nabla\phi_1 \cdot \nabla\phi_1 \, dx & \int \nabla\phi_1 \cdot \nabla\phi_2 \, dx & \int \nabla\phi_1 \cdot \nabla\phi_3 \, dx \\ \int \nabla\phi_2 \cdot \nabla\phi_1 \, dx & \int \nabla\phi_2 \cdot \nabla\phi_2 \, dx & \int \nabla\phi_2 \cdot \nabla\phi_3 \, dx \\ \int \nabla\phi_3 \cdot \nabla\phi_1 \, dx & \int \nabla\phi_3 \cdot \nabla\phi_2 \, dx & \int \nabla\phi_3 \cdot \nabla\phi_3 \, dx \end{bmatrix}$$

på element el skrivs på följande kompakta form:

Kodoptimering 24

```
elNodes = t(1:3, el);
dx = area(p(:, elNodes));
Dphi = phiGradients(p(:, elNodes));
A(elNodes, elNodes) = A(elNodes, elNodes) +
Dphi'*Dphi*dx;
```

där `area` returnerar arean av elementet ifråga, och `phiGradients` returnerar (en 2x3 matris med) gradienterna av de tre lokala basfunktionerna på elementet.