Welcome to Ma221! (Apr 5)

Divide and Conquer Alg for Sym $Ax = \lambda x$

Main Ingredient: given $A = Q \Lambda Q^T$

update cheaply for $A + \alpha v v^T$

$$A - \alpha v v^T = Q(\Lambda + \alpha (Q^T v)(v^T Q)) Q^T$$

$$= Q(\underbrace{\Lambda + \alpha v v^T}_{\text{update diagonal + rank-1}}) Q^T$$

Use characteristic polynomial

$$\det(\Lambda - \alpha v v^T - \lambda I)$$

$$= \prod_i (\lambda - \lambda_i)\left(1 + \alpha \sum_{i=1}^{n} \frac{v_i^2}{\lambda_i - \lambda}\right)$$

$$= \prod_i (\lambda - \lambda_i)\, f(\lambda)$$

solve $f(\lambda) = 0$

$f'(\lambda) = \alpha \cdot$ sum of squares

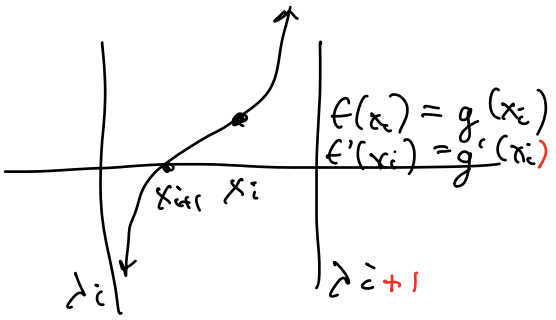
$\Rightarrow$ monotonic

poles at $\lambda = \lambda_i$

see Figs 5.2 and 5.3 in book
for why plain Newton won't work

Variant of Newton: approximate
$f(\lambda)$ by $g(\lambda)$ that has
poles at λ_i and λ_{i+1}
and matches f and f' at
last iteration

$$g(\lambda) = c_1 + \frac{c_2}{\lambda_i - \lambda} + \frac{c_3}{\lambda_{i+1} - \lambda}$$



Need evecs:

Lemma: if λ eval of $\Lambda + \alpha v v^T$
then its evec is $(\Lambda - \lambda I)^{-1} v$

proof: $(\Lambda + \alpha v v^T)(\Lambda - \lambda I)^{-1} v$

$= (\Lambda - \lambda I + \lambda I + \alpha v v^T)(\Lambda - \lambda I)^{-1} v$

$$= v + \lambda(\Lambda - \lambda I)^{-1} v + v(\underbrace{\alpha v^T (\Lambda - \lambda I)^{-1} v}_{=-1 \text{ since } f(\lambda)=0})$$

cancels $-v$

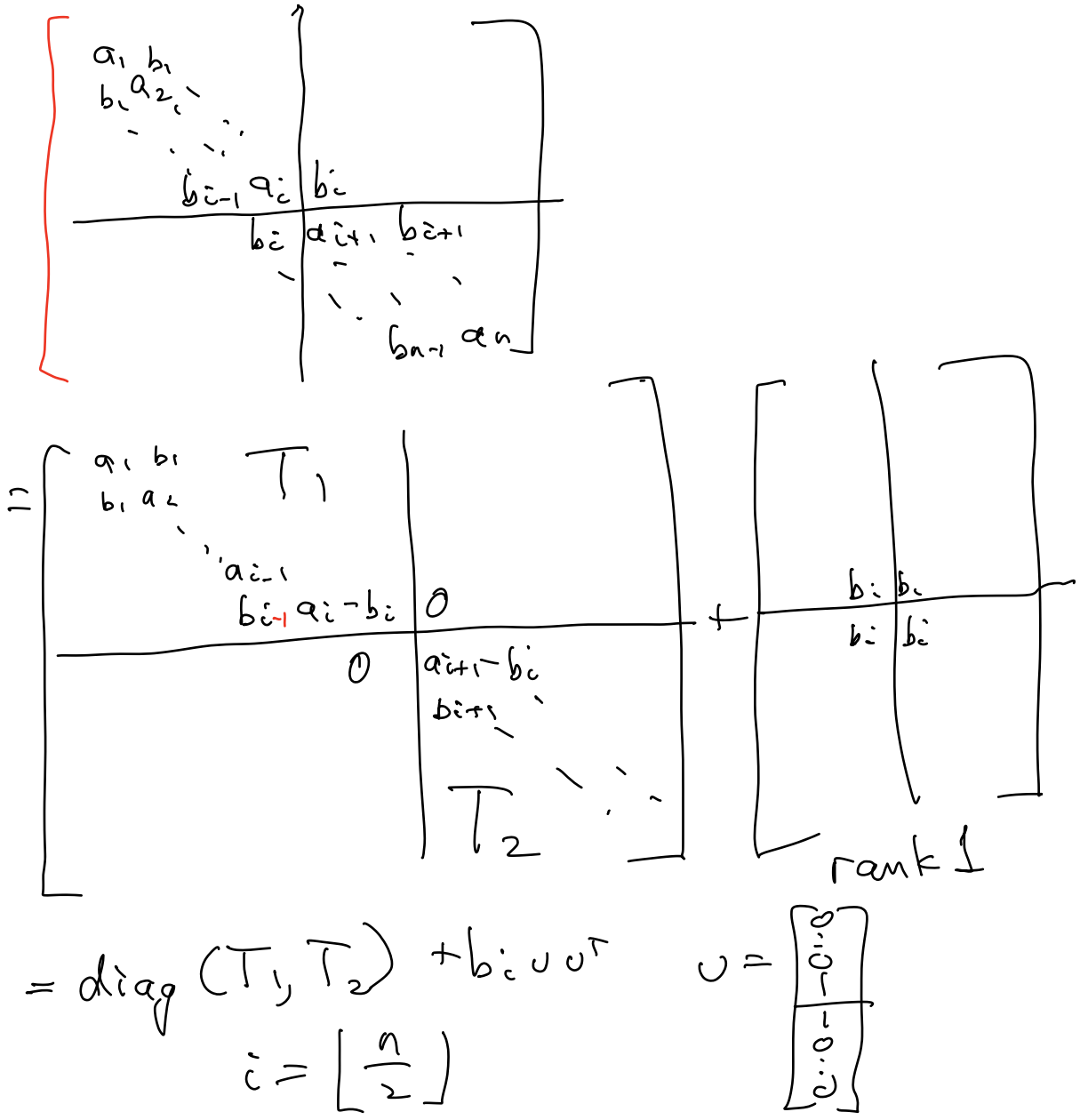
$$= \lambda(\Lambda - \lambda I)^{-1} v$$

Unfortunately, not numerically stable
because of cancellation in $(\lambda_i - \lambda)^{-1}$
clever fix in text (Gu, Eisenstat)

For general eigenproblem
Reduce A to tridiagonal T
Write alg just presented as

$$[Q', \Lambda'] = \text{Eig\_update}[Q, \Lambda, \alpha, v]$$

use as building block in Divide & Conquer
How to divide tridiagonal T into
2 problems of $\frac{1}{2}$ size?



$$= \mathrm{diag}(T_1, T_2) + b_i u u^T \qquad u = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

$$i = \left\lfloor \frac{n}{2} \right\rfloor$$

function $[Q, \Lambda] = DC\_eig(T)$

... return $T = Q \Lambda Q^T$

... n = dimension of T

if n small enough

use QR

else

$i = \left[\frac{n}{2}\right]$

... write $T = diag(T_1, T_2) + b_i v v^T$

... no work, just notation

$[Q_1, \Lambda_1] = DC\_eig(T_1)$

$[Q_2, \Lambda_2] = DC\_eig(T_2)$

... $\begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix} \begin{bmatrix} T_1 & \\ & T_2 \end{bmatrix} \begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix}^T = \begin{bmatrix} \Lambda_1 & \\ & \Lambda_2 \end{bmatrix}$

$[Q, \Lambda] = Eig\_update(diag(Q_1, Q_2), diag(\Lambda_1, \Lambda_2), b_i, v)$

endif

return

cost is $O(n^g)$ $2 < g < 3$

Algorithm for evals only

(just some of them)

use bisection, Sylvester's Thm

inertia (A) = (#eval <0, #evals $=0$, #evals >0)

= inertia $(X A X^T)$

X nonsingular

count #evals in any interal $[x, y]$

by inertia of $A - xI$, $A - yI$

How to do cheaply: choose X so

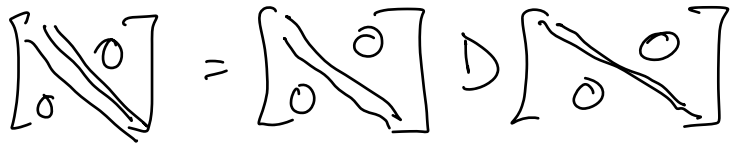
$X(A - xI)X^T$ = diagonal D

Use Gaussian elimination

without pivoting

$A \rightarrow$ tridiagonal T

$T - xI = LDL^T$



cost = $O(n)$

what about numerical stability?

function c = Negcount(T, x)

... c = # evals of T < x

= # negative entries of D where

$T - xI = LDL^T$

... diag(T) = $a_1, a_2, \ldots a_n$

... off(diag(T)) = $b_1, b_2, \ldots b_{n-1}$

... only compute diagonal of D, $d_1, d_2, \ldots$

$c = 0$

$b_0 = 0$, $d_0 = 1$

for $i = 1$ to n

$d_i = (a_i - x) - b_{i-1}^2 / d_{i-1}$

... obey parentheses!

if $d_{i-1} = 0$
$\Rightarrow d_i = -\infty$
$\Rightarrow d_{i+1} = a_{i+1} - x$

if $d_i < 0$, $c = c + 1$

end

We don't need $l_i = i^{th}$ offdiagonal of L

because

$(T - xI)(i, i+1) = b_i$

$= (LDL^T)(i, i+1) = d_i \cdot l_i$

Usual inner loop of GE:

$d_i = a_i - x - l_{i-1}^2 \cdot d_{i-1}$

replaced by

$d_i = a_i - x - b_{i-1}^2 / d_{i-1}$

Thm: Assuming we don't divide by 0
Negcount(T, x) is exactly correct
for $T+E$ where $E = E^T$,
tridiagonal, $E(i,i) = 0$
$|E(i,i+1)| \le 2.5 \cdot \varepsilon \cdot |T(i,i+1)|$

$$d_i = \left((a_i - x) - \frac{b_{i-1}^2 / d_{i-1}}{(1+\delta_2)\cdot(1+\delta_3)} \right) (1+\delta_4)$$
$$\frac{(1+\delta_1)}{}$$

$$d_i \cdot (1+\delta_1)^{-1}(1+\delta_4)^{-1} = (a_i - x) - b_{i-1}^2 (1+\delta) / d_{i-1}$$
several

$$d_i \cdot F_i = (a_i - x) - b_{i-1}^2 G_i / d_{i-1}$$
some $(1+\delta)$ factors
$$= (a_i - x) - b_{i-1}^2 G_i F_{i-1} / d_{i-1} F_{i-1}$$

$$d_i' = (a_i - x) - b_{i-1}'^2 / d_{i-1}'$$

exact recurrence for $T+E$
where E changes b_i to b_i'

values of d_i' have same signs
as $d_i \Rightarrow$ count correct for $T+E$

Why OK to divide by $d_{i-1} = 0$?
see above

Chap 6: Iterative Methods for $Ax=b$ (and $Ax=\lambda x$ Chap 7)

Model Problem: Poisson Equation (introduced in Lecture 10)

Goals: Contrast direct and iterative Methods!

- for $Ax=b$ or least squares: Use iterative methods when direct methods too slow or use too much memory, or you need less accuracy
- For $Ax=\lambda x$ or SVD: use iterative methods for same reasons, or when only need a few evals, evecs

Choosing best iterative methods depends on mathematical structure of $A \Rightarrow$ large diversity of algs and software (lots of links on class webpage)

Illustrate using Model Problem (Poisson)

Poisson arises in electrostatics, heat flow, quantum mechanics, fluid mechanics, graph theory

Present methods, from simple to more complicated, simple ones are building blocks for complicated ones

Methods covered:

(1) "Splitting Methods" for $Ax=b$

"Split" $A = M - K$, M nonsingular

so $Ax = b$ becomes $Mx = Kx + b$

Iterate: $Mx_{i+1} = Kx_i + b$, given x_0

Convergence: subtract $Mx_{i+1} = Kx_i + b$ and $Mx = Kx + b$, let $e_i = x_i - x$, get

$$Me_{i+1} = Ke_i \Rightarrow e_{i+1} = M^{-1}Ke_i$$

$$= (M^{-1}K)^{i+1} e_0$$

How fast does $(M^{-1}K)^i \to 0$?

Consider three methods:

Jacobi, Gauss-Seidel,

Successive Overrelaxation (SOR)

eg Jacobi: $M = \text{diag}(A)$

(2) Krylov Subspace Methods (KSMs)

What can you do if all you have is a subroutine that multiplies $A \cdot x$ for any input x? Or if A so large, all you can afford is $A \cdot x$?

Overview of how KSMs work

Given x_0

1) $x_1 = Ax_0$, $x_2 = Ax_1$, ..., $x_k = Ax_{k-1}$

(or a similar set, spanning same subspace)

2) Choose a linear combination of these vectors that gives the "best" approximate solution (for some def of "best")

3) If approx not good enough, increase k, repeat

Depending on
- How x_i are computed
- Properties of A (eg symmetric, positive def, ..)
- the def of "best"

you get a large set of algorithms, all used in practice, we will cover

Generalized MinimumResidual (GMRES) (A genera)

Conjugate Gradients (A s.p.d.)

lots of links to code on web page

Same idea for $Ax = \lambda x$, find best approx evec in subspace (Lanczos, ... see Chap 7)

(3) Preconditioning: How fast do these algs converge?

Complicated, but in general depends on condition # $\kappa(A)$: faster if $\kappa(A)$ smaller

Suppose we can find a matrix M^{-1}

(1) multiplying by M cheap

(2) MA better conditioned than A

$\Rightarrow$ Apply all methods to $MAx = Mb$

$$AM(M^{-1}x) = b$$

Finding good preconditioners depends on structure of A

eg for conjugate gradients

$M \cdot A$ not s.p.d. but still works

(4) Multigrid: Most Effective Preconditioner, apply to Poisson

Idea: If A arises from approximating some physical problem, then "straight forward" to find a "coarser" = smaller approximation to same problem

Eg: Poisson on 2D mesh $n \times n$

$\Rightarrow n^2$ unknowns

Approximate by Poisson on
$\frac{n}{2} \times \frac{n}{2}$ mesh, use solution of
smaller problem as preconditioner
But $\left(\frac{n}{2}\right)^2 = \frac{n^2}{4}$ still large

Use same idea recursively!
Approx by Poisson on $\frac{n}{4} \times \frac{n}{4}$ mesh, etc

$\Rightarrow$ Solves Poisson in O(#unknowns) time

(5) Domain Decomposition!
How to hybridize all above
methods, using different methods
in different domains ~ submatrices,
depending on which fastest

Optimizing Communication:
Most above methods depend
on sparse-matrix-vector-multiply
(Ax) - bottleneck if A large,
doesn't fit in memory.

lots of recent work on improving this,
eg. multiplying $x_1 = Ax_0$, $x_2 = Ax_1$, ... $x_k = Ax_{k-1}$
$= A^k x_0$
but only reading A once from
slow memory (assumes A sparse
enough!)