Dense matrices

- Households algorithm to bring $A$ to tridiagonal form $T$

- then use QR or bisection or...

to get $\{\lambda_i\}, \{x_i\}$ eigen-system

- well implemented in LAPACK, NAG, IMSL, ...

$\text{MUL}_1, ...$

$\rightarrow$ all eigenvals + vecs!
- use symmetry in $A$ to fist write as

$$
\begin{pmatrix}
A_1 & A_2 & \cdots & A_3 & A_4 & A_5 \\
& A_2 & \cdots & A_3 & \cdots & \cdots \\
& & \ddots & \cdots & \cdots & \cdots \\
& & & A_3 & \cdots & \cdots \\
& & & & A_4 & \cdots \\
& & & & & A_5
\end{pmatrix}
$$

thus use LAPACK etc for $A_i$'s!
Sparse matrices

- Matrix A contains many zeros, hence storage and arithmetic with "0"s should be avoided for memory and computing efficiency (how many? Enough to make a difference?)

- Many storage formats exist (CRS, CCS, ...), see wiki-pedia

- Implementation of matrix-vector multiplication is important,
\[(A \times)_{i} = \sum_{j} A_{ij} X_{j}\]

\[= \sum_{ij \text{ s.t. } A_{ij} \neq 0} A_{ij} X_{j}\]

- hence structure of matrix can be important.

- \(\{X_i\}\) takes as much memory as dense storage of \(A\) \(-\) best to avoid!
Power series method:

\( X_0 \) arbitrary

\( x_i, \ i = 1, \ldots, n \) eigenvectors of \( A \)

\( \lambda_i, \ i = 1, \ldots, n \) eigenvalues of \( A \)

\[ X_0 = \sum a_i x_i \]

\[ AX_0 = \sum_i a_i \lambda_i x_i \]

\[ = \sum_i a_i \lambda_i x_i \]

\[ = \lambda_{\max} \sum_i a_i \frac{\lambda_i}{\lambda_{\max}} x_i \leq 1 \]

where \( \lambda_{\max} = \{ \lambda_{i_0} | \lambda_{i_0} \geq |\lambda_i| \theta_i \} \)

\[ A^n X_0 = (\lambda_{\max})^n \sum_i a_i \left( \frac{\lambda_i}{\lambda_{\max}} \right)^n x_i \]

\[ \rightarrow (\lambda_{\max})^n X_{i_0} \quad \text{as} \quad n \rightarrow \infty \]

\( i_0 = \max \)
\[ A^m X_0 \rightarrow (\lambda_{i_0})^m X_{i_0} \]

To eigenvector corresponding to largest eigenvalue.

- Getting all Eval/vecs:

\[ \{ x_{01}, x_{11}, x_{21}, \ldots, x_{n-1} \} \]

\[ x_i \perp x_j \]

\[ A^n X_0 \rightarrow (\lambda_{\text{max}}) X_{\text{max}} \]

\[ A^m X_1 \rightarrow (\lambda_{\text{max}}) X_{\text{max}} \]

5. Keep \( \{ x_0, \ldots, x_{n-1} \} \perp \), i.e. reorthogonalize

\[ x_1 \perp x_0 \]

\[ x_2 \perp x_{11} x_2 \]

\[ x_3 \perp x_{21} x_{11} x_0 \]

\[ x_j \perp x_0, x_{1j}, \ldots, x_{j-1} \]
\[ \rightarrow A^m x_0 \rightarrow (\lambda_{\text{max}})^m x_{\text{max}} \]

\[ A^m x_1 \rightarrow (\lambda_{\text{ind. logest}})^m x_{\text{ind. logest}} \]

\[ \vdots \]

\[ A^m x_{m-1} \rightarrow (\lambda_{\text{smallest}})^m x_{\text{smallest}} \]

- basis of all iterative methods, but seldom used
Lanczos method

- Construct sym. tridiagonal matrix

\[ T_k = U_k^T A U_k \]

\[ U_k = \{ v_1, v_2, \ldots, v_k \} \]

- \( U_k \) are constructed via a recursion (iteration):

\[ \beta_{i+1} v_{i+1} = A v_i - \alpha_i v_i - \beta_i v_{i-1} \]

\[ \alpha_i = v_i^T A v_i \]

\[ \beta_{i+1} = v_{i+1}^T A v_i \]

(\( v_0 = 0, \ v_1 = \text{arbitrary} \))

- In principle, \( v_j \perp v_i \) for \( i \neq j \)

(and hence \( \beta_{i+1} = || A v_i - \alpha_i v_i - \beta_i v_{i-1} || \))

and
Krylov space \( \{ \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k \} \)

is complete basis with \( k=n \).

\[ T_k = \begin{pmatrix}
\beta_1 & \beta_2 & 0 & \cdots & 0 & 0 \\
\beta_2 & \beta_3 & \beta_4 & \cdots & 0 & 0 \\
\beta_3 & \beta_4 & \beta_5 & \cdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \ddots & \ddots & \ddots \\
0 & \cdots & 0 & \cdots & \beta_{m-1} & \beta_m \\
0 & \cdots & 0 & \cdots & \beta_m & \beta_m
\end{pmatrix} \]

- In practice, \( \mathbf{v}_j \times \mathbf{v}_i \neq \mathbf{v}_i \times \mathbf{v}_j \)
due to numerical rounding.

Solutions:

- "live with it":
  - "ghost" eigenvalues will emerge; since good EVs will replicate, have \( k \gg n \) (usually \( k = 4n \))
  - until \( k \) EVs have
replicated
→ takes longer,

degeneracies very hard to spot

- "reorthogonalize:
  make UU\_j U\_i \_by
  reorthogonalization
  → takes lots of time
  and, in particular,
  memory to store
  \( k = m \) vectors of length
  \( m \)!

- "partial reorthogonalization":
  ARPACK, see below,
  uses "implicit restarting".
- Uses:

  every where, when
  a few select EVals
  one needed
Steroids for diag methods

- Pre-conditioning

- Condition number

\[ \kappa(A) = \frac{\lambda_{\text{max}}(A)}{\lambda_{\text{min}}(A)} \geq 1 \]

So, remember power series algorithm, if 
\[ \kappa \gg 1 \quad \text{or} \quad \|A\| \geq 1 \]
Then convergence is slow fast!

\[-\] Pre-conditioning: let \[ \kappa \] close to 1! away from

- Pick:

\[ A x = AP^{-1}P x = b , \]

Then if

12
\[ k(A) \leq k(AP^{-1}), \]

one has improved

couze (at the price

of an additional matrix

multiplication - \( P \) should

be given simply).

- examples:

  * Jacobi precondition:

    \[ P = \text{diag}(A) = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix} \]

  * Three-main-diagonals:

    \[ P = \begin{pmatrix} a_{11} & a_{12} & 0 & \cdots & 0 \\ a_{21} & a_{22} & a_{23} & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n-1,n-1} & a_{n,n} \\ 0 & 0 & \cdots & 0 & a_{nn} \end{pmatrix} \]

work well if matrix is
diagonally dominant.
Convergence acceleration

Iterative methods work best for $\lambda_{\max}$. Choose $f(A)$ such that

$$\lambda_{\max}(f(A)) \gg \lambda_{\max}(A)$$

(same idea for $\lambda_{\min}$)

- example:
i.e. Chebyshev polynomials

Fig 5.1

A CHALLENGE FOR MODERN EIGENVALUE METHODS

\begin{figure}
\centering
\begin{tikzpicture}
\begin{axis}[
width=\textwidth,
height=\textwidth,
axis lines=middle,
xtick = {-10,-5,...,10},
ytick = {1,5,10,15,20,25},
ylabel={\(p_{20}(x)\)},
xlabel={x},
]
\addplot[domain=-1:1,samples=100,black] {cos(deg(10*x))};
\end{axis}
\end{tikzpicture}
\caption{Chebyshev polynomial \(p_{20}(x)\) with \(x_1 = 1\) and \(x_2 = 10\).}
\end{figure}
shift and mix

use as accelerator

\[ \frac{1}{A - \lambda_0 \|} \]

since

\[ \frac{1}{A - \lambda_0 \|} \cdot \frac{X}{x_0} = \infty \]

so in practice, guess for \( x^* \approx x_0 + \delta x \) gives

\[ \frac{1}{A - \lambda_0 \|} \cdot \frac{x^*}{x_0} \to \text{very large} \]

\( \delta \) adjustable accelerator for area around \( \lambda_0 \).
However, inversion $A^{-1}$ is of course at least as computationally expensive as eigen-system solution!

**Trick:**

Instead of

$$U_{k+1} = A \overline{U_k},$$

use

$$U_{k+1} = \frac{1}{A - \lambda I} U_k$$

So

$$U_k = (A - \lambda I) U_{k+1}$$

known $\equiv$ (lin. sys. equ) unknown

$\iff$ equivalent to lin. sys. soln

$\rightarrow$ use recursion versions!
Many such modern methods our literature as well!
ARPACK (= ARNOldi Package)

(works for unsymmetric matrices)

\[ \mathbf{L} \mathbf{u} = \mathbf{A} \mathbf{u} \]

- Lanczos + Gram-Schmidt reorthogonalization
  of \( \mathbf{k}_0 \ll N \)
  \[ \mathbf{u}_k \in \{ \mathbf{u}_1, \mathbf{u}_2, \ldots \} \]

+ Implicit Restart:

  When reorthogonalization
  of \( \mathbf{k}_0 \ \{ \mathbf{u}_k \} \) shows
  increasing loss of orthogonality,

  then select new \( \mathbf{u}_0 \perp \{ \mathbf{u}_k \} \)
  and start process again
  for remaining \( \mathbf{M} - \mathbf{k}_0 \) \textbf{Ritz vectors}
Jacobi-Davidson

Assume

\((x, \lambda)\) eigenpair s.t. \(Ax = \lambda x\)

\((u, \theta)\) guess s.t. \(Ay - \theta y = r\)  \(\text{(residual)}\)

Then choose \(t\) s.t.

\[x = u + t, \quad u \perp t\]

\[A(u + t) = \lambda(u + t) \quad (\star)\]

part of \((\star)\) \(\perp u:\)

\[(1-uu^T)A(u + t) = (1-uu^T)\lambda(u + t)\]

\((\Delta)\)

\[(1-uu^T)(A - \lambda I) t = (1-uu^T)(-Au + \theta u)\]

\[\text{RHS} = -(1-uu^T)(Au) = -(1-uu^T)(r + \theta u)\]

\[= -r\]
Since \( t \mathbf{1} \mathbf{u} \), we can write

\[
(1 - \mathbf{u} \mathbf{u}^T) t = t,
\]

Then \((\Delta)\) is

\[
(1 - \mathbf{u} \mathbf{u}^T) (\mathbf{A} - \lambda \mathbf{I}) (1 - \mathbf{u} \mathbf{u}^T) t = -r
\]

sym. matrix

But \( \lambda \) unknown, hence \( \lambda \to \Theta \)

and

\[
(1 - \mathbf{u} \mathbf{u}^T) (\mathbf{A} - \Theta \mathbf{I}) (1 - \mathbf{u} \mathbf{u}^T) t = -r
\]

\[
\mathbf{B} \cdot t = -r \quad (\square)
\]

(\( \square \)) \textbf{min. system of eqns}

for \( \mathbf{B} \) with known RHS

\( \to \) solve for \( t \)

\textbf{then improve guess}

\[
\mathbf{u}_{j+1} = \mathbf{u}_j + t
\]

\[
\Theta_{j+1} = \Theta_j + \mathbf{u}_j^T \mathbf{A} t
\]
Advantage:
The LSE need not be solved with extreme accuracy since we improve outside loop any how!

Overview:

\[ \{ u_j \} \quad u_j \perp u_i \]

\[ \{ r_j \} \quad t_j = \overline{B}^{-1} (-r_j) \]

via iterative construction of \( t_j \)’s

\[ \{ u_{j+1} = u_j + t_j \} \]

until \( \| r_j \| \leq \text{target} \)

\( (*) \) we use "ILUPack".

\( ^\wedge \text{inverse-based incomplete LU decomposition} \)
Large Scale Anderson Diagonalization

2 levels of iterations to solve $H\Psi = E_0\Psi$

Outer iteration: Jacobi-Davidson (JD)
Inner iteration: ILUPack$^+$ to solve shift-and-invert problem

- iterative methods use $H^n\Psi_0 \rightarrow E_{max}^n\Psi_{max}$
- Krylov sequence $\Psi_{n+1} = H\Psi_n = H^2\Psi_{n-1} = \ldots$

1999 30 days
2006 3 days

- shift-and-invert approach $H \rightarrow 1/(H - E_0)$ to target $E_0$ region
rewrite as $(H - E_0)\Psi_{n+1} = \Psi_n$, aka sys. lin. equations

6 hrs

- fast iterative SLE solver ILUPack and tricks

40 min

- JD is ok with approximate states $\Psi_{n+1}$

20 min

- tinkering with ILUPack and jdbsym interface

12 min

- use new package JADAMILU

10 min

$N = L^3 = 350^3 = 42,875,000$ in 2006!

Schenk et al., SIAM Reviews 50, 91–112 (2008)

Fig. 2  Plot of the electronic eigenstate at the metal-insulator transition with $\lambda = 0$, $w = 16.5$, and
$N = 350^3$. The box-and-color scheme is as in Figure 1. Note how the state extends nearly
everywhere while at the same time exhibiting certain localized regions of higher $|z_1|^2$ values.
The eigenstate has been constructed using ILUPACK-based Jacobi-Davidson. See section 9
for details.
**PARDISO - a direct solver**

- \[ B \frac{t}{t} = -r \] \[ t = ? \] \{ LSE \}

- Shift-and-invert with PARDISO also possible, but generally needs memory (fill-in)

- Port of Intel MKL (you may have used it already!)

- ARPACK + SLEP + PARDISO is fast if you have the memory!
More than a few eigenpairs?

Use overlapping intervals in eigenvalue space, preferably via

naive / task form parallelization.
Conclusions

use iterative solvers if

- only few $\{ x, A \}$ needed

or

- for capability challenge:
  - A too big to fit in memory
  - no other method works
References:


Using JADAMILU is free for non-commercial applications. (For commercial use, please contact the authors). You can acknowledge, using the references:


The PARDISO Version 5.0.0 has been released in January 2014. It contains full support of multi-threaded Schur-complement computations and full support for parallel selected inversion. In case that you are using the new version 5.0.0 please cite:

