

Notes for Ma221 Lecture 11, Apr 5, 2022

Chapters 6 and 7: Introduction to Iterative Methods for $A*x=b$ and $A*x = \lambda*x$, and the Poisson Equation, our "model problem"

Goals:

Contrast direct and iterative methods

For $Ax=b$ or least squares: Use iterative methods when direct methods are too slow, or use too much memory (from fill-in during factorization), or you don't need as much accuracy.

For eigenproblems or SVD: Use iterative methods for the above reasons, or when only a few eigenvalues/vectors are desired.

A good iterative method exploits knowledge of the underlying physical or mathematical model that gives rise to matrix.

Large diversity of methods and software tools exist, no one is best.

See the class web page for pointer to software.

Model problem to compare methods: Poisson equation in 2D or 3D

Arises in electrostatics, heat flow, quantum mechanics, incompressible fluid mechanics, graph theory, ...

Summary of iterative methods as applied to Model problem, from simple (slow) to sophisticated (fast):

- (1) Simple ones apply to more problems than Model problem
- (2) Simple ones are building blocks for more sophisticated ones

Methods covered:

(1) "Splitting Methods"

"Split" $A = M - K$ with M nonsingular, so $A*x=b$ becomes $M*x = K*x+b$

Then we iterate: Solve $M*x_{i+1} = K*x_i + b$ for x_{i+1} .

To see how it converges, subtract $M*x = K*x+b$, let $e(i) = x(i)-x$, so $M*e(i+1) = K*e(i)$ or $e(i+1) = \text{inv}(M)*K*e(i) = (\text{inv}(M)*K)^{i+1}*e(0)$.

It is easy to implement if solving with M is cheap.

It converges (fast) if $(\text{inv}(M)*K)^i \rightarrow 0$ (fast).

We will consider 3 classical splitting methods:

Jacobi, Gauss-Seidel, and Successive Overrelaxation (SOR)

(2) Krylov Subspace Methods (KSMs):

What can you do if all you have access to is a subroutine that computes $A*x$ for any x ? Or if A is so big that $A*x$ is all you can afford to do?

Then what a KSM does may be described as follows, starting with $x(0)$

1) Compute $x(1) = A*x(0)$, $x(2) = A*x(1)$, ... $x(k) = A*x(k-1)$

(or a similar set of vectors that span the same subspace).

2) Choose the linear combination $\sum_{i=0}^k a(i)*x(i)$ that "best" approximates the answer (for some definition of "best").

3) If the approximation is not good enough, increase k and repeat.

Depending on

how the set of vectors is computed,
properties of A (eg symmetry, positive definiteness, ...) and
the definition of "best approximation"

one gets a large number of different algorithms, all used in practice:
Conjugate Gradients (CG), generalized minimum residuals (GMRES), and many
more that we won't discuss in detail.

Instead of asking for the "best approximation" to $A^*x=b$, one can instead
ask for the best approximation to $A^*x = \lambda x$; this yields the eigenvalue
algorithms discussed in Chapter 7.

(3) Preconditioning. How fast do the above methods for $A^*x=b$ converge?

It depends on a variety of sometimes subtle properties of A , but it should be
no surprise that KSMs on well-conditioned matrices generally converge much
faster than on ill-conditioned matrices. But suppose we can find a
"preconditioner", i.e. a matrix M where (1) M^*A is well conditioned, and
(2) multiplying by M is cheap. Then one applies the KSM to solve $M^*A^*x = M^*b$.
Finding good preconditioners can depend in complicated ways on A .
And it is not always straightforward to use them. For example, CG depends on
 A being s.p.d., whereas M^*A will rarely be symmetric. Still it is possible
to reorganize CG to be able to use preconditioners.

(4) Multigrid: This is one of the most effective preconditioners for problems
with a suitable mathematical structure. If your matrix
arises from solving a physical problem, say Poisson's equation, it is
straightforward to find an approximation with, say, half as many mesh
points in each direction, and use the solution of that smaller, and so
easier, problem (eg $1/4$ the dimension in 2D), as a preconditioner.
But if the dimension n is very large, so is $n/4$. So we just apply the
same idea recursively, using a problem of size $n/16$ to get a starting guess
for the problem of size $n/4$, etc. When multigrid works, it can be optimal,
solving an $n \times n$ linear system in $O(n)$ flops. But not every problem has the
structure that permits this.

(5) Domain decomposition. This can be thought of as a way to hybridize the
above methods, using different methods in different "domains" or submatrices,
where A may be easier to work with than as a whole.

Optimizing performance

Most methods assume the only thing you can afford to do is $y=A^*x$.

Computing A^*x is generally the bottleneck, and usually because of communication:

For example, if A is too large to fit in cache, it needs to be read out of
main (slow) memory every iteration, with only one multiplication and addition
per entry. So recent work has addressed communication-avoiding KSMs, where methods
like CG are redesigned to take $s>1$ steps for each access of A , instead of 1 step.

See links on the class webpage for details.

(We pause the recorded lecture here.)

Our Model Problem: Poisson's Equation

1D with Dirichlet boundary conditions

$$-d^2 v(x)/dx^2 = f(x) \text{ on } (0,1) \text{ with } v(0)=v(1)=0$$

Recalling Lecture 6 (on Gaussian elimination for matrices with structure), discretizing this yields

$$T_N * [v_1; \dots; v_N] = T_N * v = h^2 * [f_1; \dots; f_N] = h^2 * f$$

where T_N is tridiagonal with diagonals = 2 and offdiagonals = -1.

This is often referred to as a "stencil" on a 1D mesh. (draw)

Eigenvalues and eigenvectors of T_N

Lemma: $T_N * z_j = \lambda_j * z_j$ where $\lambda_j = 2 * (1 - \cos(\pi * j / (N+1)))$

$$z_j(k) = \sqrt{2/(N+1)} * \sin(j * k * \pi / (N+1)) \text{ and } \|z_j\|_2 = 1$$

Proof: Trigonometry (homework Question 6.1)

Corollary: The matrix Z where $Z_{jk} = \sqrt{2/(N+1)} * \sin(j * k * \pi / (N+1))$

is orthogonal (related to FFT-Fast Fourier Transform matrix later)

(see Fig 6.1 in text (graph711.eps) for eigenvalues,

Fig 6.2 in text (graph712.eps) for eigenvectors)

Eigenvalues range from smallest $\lambda_j \sim (\pi * j / (N+1))^2$ for small j

to largest $\lambda_N \sim 4$

This tells us the condition number of T_N ,

$$\lambda_N / \lambda_1 \sim (2(N+1)/\pi)^2 = (2/\pi)^2 * h^{-2}$$

2D with Dirichlet boundary conditions

$$-d^2 v(x,y) / dx^2 - d^2 v(x,y) / dy^2 = f(x,y) \text{ on square } (0,1)^2,$$

with $v(x,y)=0$ on boundary of square

Discretizing as before and writing v_{ij} for $v(i*h, j*h)$ we get

$$4 * v_{ij} - v_{i-1,j} - v_{i+1,j} - v_{i,j-1} - v_{i,j+1} = h^2 * f_{ij}$$

This is often referred to as a "stencil" on a 2D mesh. (draw)

Letting V be the $N \times N$ matrix of v_{ij} , we can also write

$$2 * v_{ij} - v_{i-1,j} - v_{i+1,j} = (T_N * V)_{ij}$$

$$2 * v_{ij} - v_{i,j-1} - v_{i,j+1} = (V * T_N)_{ij}$$

or

$$4 * v_{ij} - v_{i-1,j} - v_{i+1,j} - v_{i,j-1} - v_{i,j+1} = (T_N * V + V * T_N)_{ij}$$

and

$$T_N * V + V * T_N = h^2 * F$$

a system of N^2 linear equations for the N^2 entries of V , though

not in the usual " $A * x = b$ " form. (See Q4.6 in the text, on the Sylvester Equation.)

We can ask about the corresponding eigenvalues and eigenvectors

in the analogous form

$$T_N * V + V * T_N = \lambda * V$$

Suppose that $T_N * z_i = \lambda_i * z_i$ and $T_N * z_j = \lambda_j * z_j$ are any two eigenpairs of T_N . Letting $V = z_i * z_j^T$, we get

$$\begin{aligned} T_N * V + V * T_N &= T_N * z_i * z_j^T + z_i * z_j^T * T_N \\ &= (T_N * z_i) * z_j^T + z_i * (z_j^T * T_N) \\ &= (\lambda_i * z_i) * z_j^T + z_i * (z_j^T * \lambda_j) \\ &= (\lambda_i + \lambda_j) * (z_i * z_j^T) \\ &= (\lambda_i + \lambda_j) * V \end{aligned}$$

so that V is an "eigenvector". Since there are N^2 unknowns, we expect N^2 eigenpairs, for all possible sums of pairs $\lambda_i + \lambda_j$.

We want to write V as a single vector in a way that easily extends to higher dimensional Poisson equations:

In the 3x3 case, write V columnwise from left to write as:

$$v = [v_{11}; v_{21}; v_{31}; v_{12}; v_{22}; v_{32}; v_{13}; v_{23}; v_{33}]$$

and $T_{N \times N}$ as

$$\begin{bmatrix} 4 & -1 & 0 & -1 & & & & & \\ -1 & 4 & -1 & & -1 & & & & \\ 0 & -1 & 4 & & & -1 & & & \\ -1 & & & 4 & -1 & 0 & -1 & & \\ -1 & -1 & -1 & 4 & -1 & -1 & & & \\ & -1 & 0 & -1 & 4 & & -1 & & \\ & & -1 & & & 4 & -1 & 0 & \\ & & & -1 & -1 & 4 & -1 & & \\ & & & & -1 & 0 & -1 & 4 & \end{bmatrix}$$

The pattern is that in each row, the

- 4 on the diagonal multiplies v_{ij} ,
- 1s next to the diagonal multiply $v_{i-1,j}$ and $v_{i+1,j}$
- 1s N away from the diagonal multiply $v_{i,j-1}$ and $v_{i,j+1}$

Note that some rows have fewer than 4 -1s, when these are on the boundary of the square. One may confirm that

$$T_{N \times N} * v = h^2 * f$$

Another way to write $T_{N \times N}$ is

$$\begin{bmatrix} T_N + 2 * I_N & -I_N & & \\ & -I_N & T_N + 2 * I_N & -I_N \\ & & \dots & \\ & & & -I_N & T_N + 2 * I_N \end{bmatrix}$$

a pattern we will generalize to arbitrary dimensions, using Kronecker products.

(We pause the recorded lecture here.)

Poisson's equation in d dimensions.

We need to convert arrays most naturally indexed by 2 or more indices

(like V above) into vectors.

Def: Let X be $m \times n$, then $\text{vec}(X)$ is the $m \times n \times 1$ vector gotten by stacking the columns of X atop one another, from left to right.

(In Matlab, $\text{vec}(X) = \text{reshape}(X, m \times n, 1)$)

Def: Let A be $m \times n$ and B be $p \times q$, then A kron B is the $m \times p \times n \times q$ matrix

$$\begin{bmatrix} A_{11} * B & A_{12} * B & \dots & A_{1n} * B \\ A_{21} * B & A_{22} * B & \dots & A_{2n} * B \\ \dots & \dots & \dots & \dots \\ A_{m1} * B & A_{m2} * B & \dots & A_{mn} * B \end{bmatrix}$$

which is called the Kronecker product of A and B. In Matlab, $A \text{ kron } B = \text{kron}(A, B)$.

Lemma: Let A be $m \times m$, B be $n \times n$ and X be $m \times n$, then

1. $\text{vec}(A * X) = (I_n \text{ kron } A) * \text{vec}(X)$
2. $\text{vec}(X * B) = (B^T \text{ kron } I_m) * \text{vec}(X)$
3. The 2D Poisson equation $T_N * V + V * T_N = h^2 * F$ may be written $(I_N \text{ kron } T_N + T_N \text{ kron } I_N) * \text{vec}(V) = h^2 * \text{vec}(F)$

Proof of 1: $I_n \text{ kron } A = \text{diag}(A, A, \dots, A)$ n times so $(I_n \text{ kron } A) * \text{vec}(X)$
 $= \text{diag}(A, A, \dots, A) * [X(:,1); X(:,2); \dots ; X(:,n)]$
 $= [A * X(:,1); A * X(:,2); \dots ; A * X(:,n)]$
 $= \text{vec}(A * X)$

Proof of 2: similar (homework, question 6.4)

Proof of 3: Apply part 1 to $T_N * V$ and part 2 to $V * T_N$, noting that T_N is symmetric.

Note that

$$\begin{aligned} I_N \text{ kron } T_N + T_N \text{ kron } I_N = \\ \begin{bmatrix} T_N & & & \\ & T_N & & \\ & & \dots & \\ & & & T_N \end{bmatrix} + \begin{bmatrix} 2 * I_N & -I_N & & \\ & -I_N & 2 * I_N & -I_N \\ & & & \dots \\ & & & & -I_N & 2 * I_N \end{bmatrix} \end{aligned}$$

In Matlab, this is two lines:

$$\begin{aligned} TN = 2 * \text{eye}(N) - \text{diag}(\text{ones}(N-1,1), -1) - \text{diag}(\text{ones}(N-1,1), +1) \\ TNN = \text{kron}(\text{eye}(N), TN) + \text{kron}(TN, \text{eye}(N)) \end{aligned}$$

Lemma (homework, question 6.4)

- (1) Assume that $A * C$ and $B * D$ are well defined.
Then $(A \text{ kron } B) * (C \text{ kron } D) = (A * C) \text{ kron } (B * D)$
- (2) If A and B are invertible then $(A \text{ kron } B)^{-1} = A^{-1} \text{ kron } B^{-1}$
- (3) $(A \text{ kron } B)^T = A^T \text{ kron } B^T$

Prop: Let $T = Z * \Lambda * Z^T$ be the eigendecomposition of the $N \times N$ symmetric matrix T. Then the eigendecomposition of

$$\begin{aligned}
& I \text{ kron } T + T \text{ kron } I \\
&= (Z \text{ kron } Z) * (I \text{ kron } \text{Lambda} + \text{Lambda} \text{ kron } I) * (Z^{\wedge}T \text{ kron } Z^{\wedge}T)
\end{aligned}$$

where $I \text{ kron } \text{Lambda} + \text{Lambda} \text{ kron } I$ is a diagonal matrix whose $((i-1)*N+j)$ th diagonal entry is $\text{lambda}_i + \text{lambda}_j$, and $Z \text{ kron } Z$ is orthogonal with $((i-1)*N+j)$ th column $z_i \text{ kron } z_j$.

Proof: Multiply out the factorization using the last lemma to get

$$\begin{aligned}
& (Z \text{ kron } Z) * (I \text{ kron } \text{Lambda} + \text{Lambda} \text{ kron } I) * (Z^{\wedge}T \text{ kron } Z^{\wedge}T) \\
&= (Z * I \text{ kron } Z * \text{Lambda} + Z * \text{Lambda} \text{ kron } Z * I) * (Z^{\wedge}T \text{ kron } Z^{\wedge}T) \\
&= (Z * I * Z^{\wedge}T \text{ kron } Z * \text{Lambda} * Z^{\wedge}T + Z * \text{Lambda} * Z^{\wedge}T \text{ kron } Z * I * Z^{\wedge}T) \\
&= (I \text{ kron } T + T \text{ kron } I) \text{ as desired}
\end{aligned}$$

Similarly, Poisson's equation in 3D leads to the matrix

$$\begin{aligned}
T_{N \times N \times N} = & (T_N \text{ kron } I_N \text{ kron } I_N) + \\
& (I_N \text{ kron } T_N \text{ kron } I_N) + \\
& (I_N \text{ kron } I_N \text{ kron } T_N)
\end{aligned}$$

with eigenvalue matrix

$$\begin{aligned}
& (\text{Lambda}_N \text{ kron } I_N \text{ kron } I_N) + \\
& (I_N \text{ kron } \text{Lambda}_N \text{ kron } I_N) + \\
& (I_N \text{ kron } I_N \text{ kron } \text{Lambda}_N)
\end{aligned}$$

i.e. N^3 eigenvalues $\text{lambda}_i + \text{lambda}_j + \text{lambda}_k$ for all triples (i,j,k) , and corresponding eigenvector matrix $Z \text{ kron } Z \text{ kron } Z$.

Poisson's equation in d dimensions is similarly represented.

(We pause the recorded lecture here.)

We now know enough to describe how to solve Poisson's equation with the FFT. This is a direct method, not iterative, but we want to compare it to the other methods.

We describe it first for 2D Poisson, and then how to extend to higher dimensions.

Start with 2D Poisson written as $T_N * V + V * T_N = F$; this is a special case of the Sylvester equation described in HW Question 4.6: We substitute the eigendecomposition

$$T_N = Z * \text{Lambda} * Z^{\wedge}T$$

yielding

$$Z * \text{Lambda} * Z^{\wedge}T * V + V * Z * \text{Lambda} * Z^{\wedge}T = F$$

Premultiply by $Z^{\wedge}T$ and postmultiply by Z , yielding

$$\text{Lambda} * Z^{\wedge}T * V * Z + Z^{\wedge}T * V * Z * \text{Lambda} = Z^{\wedge}T * F * Z$$

Rename $V' = Z^{\wedge}T * V * Z$ and $F' = Z^{\wedge}T * F * Z$, yielding

$$\text{Lambda} * V' + V' * \text{Lambda} = F'$$

Since Lambda is diagonal, this is a diagonal system of equations, with

$$(\text{Lambda} * V' + V' * \text{Lambda})(i,j) = \text{lambda}(i) * V'(i,j) + V'(i,j) * \text{lambda}(j) = F'(i,j)$$

or

$$V'(i,j) = F'(i,j) / (\text{lambda}(i) + \text{lambda}(j))$$

This yields the overall algorithm:

$$(1) \text{ compute } F' = Z^{\wedge}T * F * Z$$

(2) compute $V'(i,j) = F'(i,j) / (\lambda(i) + \lambda(j))$ for all (i,j)

(3) compute $V = Z * V' * Z^T$

The main costs are the matrix multiplications by Z and Z^T in (1) and (3), which would cost $O(N^3)$ if done straightforwardly. But they can be done in $O(N^2 \log N)$ operations (recall N^2 is the number of unknowns) by noting that Z is closely related to the Fast Fourier Transform matrix:

$FFT(i,j) = \exp(\sqrt{-1} * \pi * i * j / N) = \cos(\pi * i * j / N) + \sqrt{-1} * \sin(\pi * i * j / N)$
for which there are $O(N * \log N)$ methods for multiplication by a vector.

The same idea extends to high dimensions by using Kronecker product notation:

$$I \text{ kron } T + T \text{ kron } I = (Z \text{ kron } Z) * (I \text{ kron } \Lambda + \Lambda \text{ kron } I) * (Z^T \text{ kron } Z^T)$$

so

$$\text{inv}(I \text{ kron } T + T \text{ kron } I)$$

$$= \text{inv}((Z \text{ kron } Z) * (I \text{ kron } \Lambda + \Lambda \text{ kron } I) * (Z^T \text{ kron } Z^T))$$

$$= \text{inv}(Z^T \text{ kron } Z^T) * \text{inv}(I \text{ kron } \Lambda + \Lambda \text{ kron } I) * \text{inv}(Z \text{ kron } Z)$$

$$= (Z \text{ kron } Z) * \text{inv}(\text{diagonal matrix } D) * (Z^T \text{ kron } Z^T)$$

where the diagonal entries of D are sums $\lambda_i + \lambda_j$.

For 3D Poisson we would get

$$(Z \text{ kron } Z \text{ kron } Z) * \text{inv}(\text{diagonal matrix } D) * (Z^T \text{ kron } Z^T \text{ kron } Z^T)$$

where the diagonal entries of D are sums $\lambda_i + \lambda_j + \lambda_k$.

(We pause the recorded lecture here.)

Summary of performance of methods for solving the Poisson equation in 2D (and 3D).

We will count #Flops, Memory used, #Parallel Steps, and #Processors.

"Parallel Steps" is the fewest possible, on a machine with as many processors as needed (#Procs), and so it refers only to the available parallelism, not how you would actually implement it on a real machine with a smaller number of processors.

The algorithms are sorted in two related orders: from slowest to fastest for the Poisson Equation, and (roughly) from most general (fewest assumptions about the matrix) to least general.

Some explanations and abbreviations:

Explicit inverse assumes we have precomputed the exact inverse of the matrix (and are not counting the cost of doing this!) and only need to multiply by it.

SOR = Successive overrelaxation

SSOR/Chebyshev = Symmetric SOR with Chebyshev Acceleration

FFT = Fast Fourier Transform

BCR = Block Cyclic Reduction

Lower Bound is a lower bound that assumes one flop per component of the output vector

SpMV means sparse-matrix times dense-vector multiplication.

For 2D Poisson on an $n \times n$ mesh, we let $N = n^2 = \text{\#unknowns}$
 For 3D Poisson on an $n \times n \times n$ mesh, we let $N = n^3 = \text{\#unknowns}$
 If the table entry for 3D Poisson differs from 2D, it is shown in parentheses.
 All table entries are meant in a Big-Oh sense.

Method	Direct or Iterative	#Flops	Mem	#Parallel Steps	#Procs
Dense Cholesky works on any spd matrix	D	N^3	N^2	N	N^2
Explicit Inverse	D	N^2	N^2	$\log N$	N^2
Band Cholesky Works on any spd band matrix with cost $O(N * bw^2)$, 2D: $bw = n = N^{1/2}$ 3D: $bw = n^2 = N^{2/3}$	D	N^2 ($N^{7/3}$)	$N^{3/2}$ ($N^{5/3}$)	N	N ($N^{4/3}$)
Jacobi 3D case: Homework! #Flops = $O(\text{\#Flops}(\text{SpMV}) * \text{\#iterations})$, where $\text{\#Flops}(\text{SpMV}) = O(\text{\#nonzeros in } A)$ and $\text{\#iterations} \sim \text{cond}(A) = n^2$ in 2D and 3D Works on any diagonally dominant matrix, convergence rate depends on $\text{cond}(A)$	I	N^2 ($N^{5/3}$)	N	N ($N^{2/3}$)	N
Gauss-Seidel 3D case: Homework! cost analogous to Jacobi (constants differ) Works on any diagonally dominant or spd matrix	I	N^2 ($N^{5/3}$)	N	N ($N^{2/3}$)	N
Sparse Cholesky Assumes we are using the best ordering known (nested dissection) Works on any spd matrix; complexity arises from cost of dense Cholesky on trailing $n \times n$ submatrix (in 2D) or $n^2 \times n^2$ submatrix (in 3D)	D	$N^{3/2}$ (N^2)	$N * \log N$ ($N^{4/3}$)	$N^{1/2}$ ($N^{2/3}$)	N ($N^{4/3}$)
Conjugate Gradients 3D case: Homework! #Flops = $O(\text{\#Flops}(\text{SpMV}) * \text{\#iterations})$, where $\text{cost}(\text{SpMV}) = O(\text{\#nonzeros in } A)$ and $\text{\#iterations} = O(\sqrt{\text{cond}(A)})$ Works on any spd matrix	I	$N^{3/2}$ ($N^{4/3}$)	N	$N^{1/2} * \log N$ ($N^{1/3}) * \log N$)	N
SOR 3D case: Homework!	I	$N^{3/2}$ ($N^{4/3}$)	N	$N^{1/2}$ ($N^{1/3}$)	N

SOR = Successive Overrelaxation

Cost analysis analogous to Conjugate Gradients.

Works on any spd matrix, convergence rate depends on cond

SSOR/Chebyshev	I	$N^{5/4}$	N	$N^{1/4}$	N
3D case: Homework!		$(N^{7/6})$		$(N^{1/6})$	

#Flops = $O(\text{\#Flops}(\text{SpMV}) * \text{\#iterations})$, where $\text{\#flops}(\text{SpMV}) = O(\text{\#nonzeros in } A)$
and $\text{\#iterations} = (\text{cond}(A))^{1/4}$

Works on any spd matrix, but need to know range of eigenvalues

FFT	D	$N * \log N$	N	$\log N$	N
-----	---	--------------	---	----------	---

Works on model problem

BCR	D	$N * \log N$	N	?	?
-----	---	--------------	---	---	---

BCR = Block Cyclic Reduction
3D case: Homework!

Works on problems slightly more general than model problem

Multigrid	I	N	N	$\log^2 N$	N
-----------	---	---	---	------------	---

Many variants of this algorithm, which work quite generally on problems arising from elliptic PDE and FEM models

Lower bound		N	N	$\log N$	
-------------	--	---	---	----------	--
