CS475 Computational Linear Algebra

Contents

[Module 01_Introduction – 05.04]

Key Themes: Direct vs Iterative Methods

Direct: an algorithm in which a finite sequence of arithmetic operations yields the solution (at least in theory, in exact arithmetic)

- e.g. Gaussian elimination (LU factorization)

Iterative: an algorithm in which the same operation are repeated to gradually improve an approximate solution

- e.g. Jacobi, Gauss-Seidel, Conjugate Gradient

KT: Matrix Structure

Dense matrices: most or all entries are **non-zero**. Store N x N array, manipulate "normally"
Sparse matrices: most entries are **zero**. Non-zero locations may exhibit *patterns*.

Can we exploit sparse patterns/structure to save memory and/or flops?

KT: Factorization

How can we express a given matrix as a product of other matrices?

e.g. A = BCD

- What kind of factorizations exist?
- What are their properties?
- How can we exploit their properties in algorithm design?

KT: orthogonality

Notions of orthogonality will arise repeatedly in designing our algorithms.
Recall:

- real vectors u,v are orthogonal if $u^Tv = 0$
- a real matrix is orthogonal if $Q^T = Q^{-1}$ (i.e. $Q^TQ = I$)
-

Topics: Linear Systems

Solve Ax = b where A is size n x n, b is n x 1 (and full-rank)
Basic idea should be pretty familiar
Doing it efficiently and accurately on large problems can take substantial effort

Topics: Least Squares Problems

Solve more general Ax = b where problem may have:

- "too many" equations/constraints (over-determined)
- "too few" equations/constraints (under-determined)

May be familiar from data-fitting or regression
e.g. fit a line (linear function) to a set of many data points

Topics: Eigenvalue Problems

Solve for the set of *eigenvalues* λ and *eigenvectors* v of a given matrix A, such that:

$Av = \lambda v$

Equivalent to a factorization of A into

$A = Q\Lambda Q^{-1}$

Where Λ is diagonal, and columns of Q are eigenvectors
e.g. Google's PageRank relies on finding eigenvectors

Topics: Singular Value Decomposition

Solve for a factorization of a general m x n matrix A such that

$$A = U\Sigma V^T$$

Where:

- U and V are orthogonal matrices
- Σ is diagonal with positive entries (called the *singular values* of A)

e.g. can be used to build low-rank *approximations* of matrices or data

Applications

We'll likely explore/discuss a few practical applications:

- Temperature modeling via finite difference heat equation
- Image de-noising
- Spectral clustering
- Fluid animation

[Module 02_Introductory – 05.04]

Range

The **range** of a matrix A defined by

range(A) = {y : $y = Ax$ for some x}

The set/space of all vectors that can be generated by (left-)multiplying some vector with the matrix *A*

Interpreting matrix-vector multiplication

Can view matrix-vector multiplication as taking a linear combination of A's columns, where x gives the "weights"
e.g. Ax = b implies:

[b] = [a1 | a2 | … | an][x1 x2 … x_n]^T = x1[a1] + x2[a2] + … + x_n[a_n]

$$\begin{bmatrix} b \end{bmatrix} = \begin{bmatrix} a_1 \end{bmatrix} \begin{bmatrix} a_2 \end{bmatrix} \begin{bmatrix} \dots \end{bmatrix} \begin{bmatrix} a_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} = x_1 \begin{bmatrix} a_1 \end{bmatrix} + x_2 \begin{bmatrix} a_2 \end{bmatrix} + \cdots + x_n \begin{bmatrix} a_n \end{bmatrix}$$

Range = Column Space

So all vectors in the range of A can be expressed as linear combinations of *columns* of A
Therefore range is also called the **column space**

Row Space

Analogous to column space:
The space of vectors that can be written as linear combinations of the *rows* of a matrix

Nullspace

The **nullspace** (or "kernel") of matrix A is

null(A) = {x: Ax = **0**}

i.e. the set of vectors that produce the zero-vector **0** when (left-)multiplied by A

Dimension of a Vector Space

If a vector space *V* has a basis with *n* elements, we say that the **dimension** of *V*, denoted dim(*V*), is *n*
e.g. one basis for R^3 is {(1,0,0), (0,1,0), (0,0,1)}, so dim(R^3) = 3
recall: not (necessarily) the same as vector length
e.g. the vectors {(1,0,0), (0,1,0)} only span a two-dimensional space

Rank and Nullity of a Matrix

Column rank: the dimension of the column space (range)
Row rank: the dimension of the row space
It can be proven that column rank = row rank, so we just say **rank**

Dimension of the nullspace is the **nullity** of A

In general:

rank(A) + nullity(A) = n (i.e. # of columns)

Full Rank

A m x n matrix *A* is of **full rank** if:

Rank(A) = min(m,n)

In particular, if m ≥ n (i.e. A is tall) and full rank, then it has *n* linearly independent column vectors
Equivalently:
A defines a one-to-one map
$Av_1 \neq Av_2$ for any $v_1 \neq v_2$

Matrix Inverse

A square matrix A is of full rank is called **invertible** or **nonsingular**
The **inverse** of A is denoted A^{-1}, and is the unique matrix satisfying

$A^{-1}A = A^{-1}A = I$

Where *I* is the identity

Invertible matrices

For *real square A* the following statements are equivalent:

- A has an inverse A^{-1}
- Rank(A) = m
- Range(A) = R^m
- Null(A) = {0}
- A has no zero eigenvalues
- A has no zero singular values
- det(A) ≠ 0

Matrix Inverse – Some Identities

$(AB)^{-1} = B^{-1}A^{-1}$

$(A^{-1})^T = (A^T)^{-1} = A^{-T}$

$B^{-1} = A^{-1} - B^{-1}(B - A)\,A^{-1}$

(How can we verify the last one?)

The Sherman-Morrison formula

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1 + v^TA^{-1}u}$$

For $A \in R^{n \times n}$ and vectors *u*, *v*
Allows us to "update" A^{-1} without starting from scratch

Sherman-Morrison-Woodbury formula

Generalizing gives the Woodbury identity (Sherman-Morrison-Woodbury formula):

$(A + UV^T)^{-1} = A^{-1} - A^{-1}U(I + V^TA^{-1}U)V^TA^{-1}$

Where $U, V \in R^{n \times k}$

So: a rank k modification to matrix A yields a rank k correction to A^{-1}

[Module 03_LUFactorization – 05.04/05.06]

Solving Linear Systems of Equations

Many practical problems rely on solving systems of linear equations of the form

Ax = b

Where A is a matrix, *b* is a right-hand side (column) vector, and *x* is a (column) vector of unknowns

Methods for *nonlinear* problems also often need to solve linear systems as building blocks.

Example: Animating Fluids

Computing one time step typically requires solving a linear system involving at least **one million unknowns** (e.g. for a sim grid of 100x100x100 cells)

i.e. the matrix A has dimensions of at least 1000000x1000000

Must be done at least once per frame; animations are usually played back at 24-30 fps

- e.g. for 10s of video, must solve ~300 problems of solve 1000000^2 each.

So: need methods to solve linear systems **efficiently** and **accurately.**

Direct Methods

We'll start by looking at *direct* methods for *Ax* = *b*.

Builds on familiar ideas from Gaussian elimination (LU factorization).

We still strive to avoid constructing A^{-1} itself, since this can be

- less efficient (in terms of operation counts)
- less accurate (incurs more round-off error)
- more costly in terms of storage (depending on matrix sparsity)

Gaussian Elimination: ***Ax = b***

We interpret Gaussian elimination as follows:

1. **Factor** matrix A into L = LU, where L and U are *triangular*. (Factorization)
2. **Solve** *Lz* = *b* for intermediate vector *z*. (Forward/Backward Solve [substitution])
3. **Solve** *Ux* = *z* for *x*. ""

Visualizing the Factorization Process

We are zeroing out the lower triangle, one column at a time.

$$\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix} \rightarrow \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} \rightarrow \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \rightarrow \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix}$$

Resulting matrix is *U*.

The multiplicative factors at each step give us *L*.

Algorithm – LU Factorization (Decomposition)

```
For k = 1 ... n                          //iterate over all rows
    For i = k + 1, …, n                  //iterate over each row i beneath row k
        Mult = a_ik / a_kk               //determine row i's multiplicative factor
        a_ik = mult                      //store this factor (instead of a zero)
        For j = k + 1, …, n              //iterate over all columns in the row
            a_ij = a_ij – mult * a_kj    //subtract the scaled row data
        End
    End
End
```

Example

We will try to solve $\begin{bmatrix}1 & 1 & 1\\ 1 & -2 & 2\\ 1 & 2 & -1\end{bmatrix}\begin{bmatrix}x_0\\ x_1\\ x_2\end{bmatrix} = \begin{bmatrix}0\\ 4\\ 2\end{bmatrix}$ for the vector $\vec{x}$.

First let's factor *A* into *LU*.

Factorization and Triangular Solves

Given the factorization $A = LU$, we can use this to solve $Ax = b$.
This is the same as solving $LUx = b$. rewrite it as $L(Ux) = b$.
If we define $z = Ux$ we can rewrite this as two separate solves:

First: solve $Lz = b$ for z.
Then: Solve $Ux = z$ for x.

Why are two matrix solves better than one?

Triangular Solves – Advantage?

Our two solves are:

$Lz = b$ for z. "forward solve"
$Ux = z$ for x. "backward solve"

L and *U* are both **triangular**: all entries above, or below, the diagonal are zero, respectively.
This makes them easier (i.e. more efficient) to solve.

Forward Solve

The "forward solve", $Lz = b$ gives us the same RHS as standard Gaussian elimination yields after row-reduction.
e.g. $Lz = b$ is:

$$\begin{bmatrix}1 & 0 & 0\\ 1 & 1 & 0\\ 1 & -1/3 & 1\end{bmatrix}\begin{bmatrix}z_0\\ z_1\\ z_2\end{bmatrix} = \begin{bmatrix}0\\ 4\\ 2\end{bmatrix}$$

Solving it gives $z = \begin{bmatrix}0\\ 4\\ 10/3\end{bmatrix}$.

Backward Solve

The backward solve $Ux = z$ is just the back-substitution step in standard Gaussian elimination.
e.g. Back-substitution on:

$$\begin{bmatrix}1 & 1 & 1\\ 0 & -3 & 1\\ 0 & 0 & -5/3\end{bmatrix}\begin{bmatrix}x_0\\ x_1\\ x_2\end{bmatrix} = \begin{bmatrix}0\\ 4\\ 10/3\end{bmatrix}$$

Solution is $x = \begin{bmatrix}4\\ -2\\ -2\end{bmatrix}$.

Triangular Solves (Forward and Backward)

```
For i = 1, ..., n
    z_i := b_i
    For j = 1, ..., i − 1
        z_i := z_i − l_ij * z_j
    EndFor
EndFor
```

```
For i = n, ..., 1
    x_i := z_i
    For j = i + 1, ..., n
        x_i := x_i − u_ij * x_j
    EndFor
    x_i := x_i / u_ii
EndFor
```

Costs of Gaussian Elimination

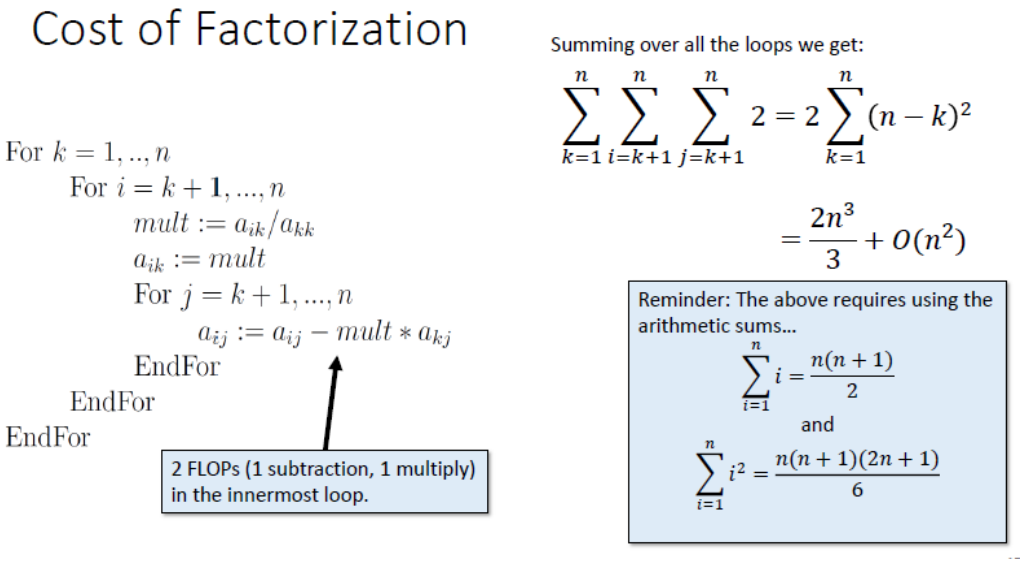
We want to know the (asymptotic) cost to solve a system of size *n*.
We will measure cost in total *FLOPs*: *floating point operations*.
Approximate as the number of: *adds* + *subtracts* + *multiplies* + *divides*

A true count will depend on the actual hardware

- E.g. fused-multiply-add (FMA) may be a single operation.

(Careful: FLOPS is also floating-point-operations-per-second)

Cost of Factorization



So LU factorization costs approximately $\frac{2n^3}{3} + O(n^2) + O(n^3)$ FLOPs.

What about our forward/backward solves?
Counting gives $n^2 + O(n)$ FLOPs for *each* triangular solve.

Total is $O(n^3)$. For large *n*, the cost of factorization is dominant.

Given factorization, solving an (additional) RHS is fairly cheap: $O(n^2)$

Exercise: Counting FLOPs

Show that the cost of forward substitution is: $n^2 + O(n)$ FLOPs.
<see 03_examples>

Finding $\mathbf{A^{-1}}$

How would we construct A^{-1}, if we really wanted to:
Note that $A = A^{-1} = I$

1. Factor A = LU
2. Use forward/back solves with 3 different RHS for columns of A^{-1}:
   i.e. $LUv_1 = [1,0,0]$, $LUv_2 = [0, 1, 0]$, $LUv_3 = [0, 0, 1]$

Recall: most numerical algorithms avoid actually computing A^{-1}

Summary

- Looked at a number of useful linear algebra concepts
- Considered basic LU factorization and its cost

Next time:

Can speed up solution exploiting:

- Matrix structure (banded matrices)
- Matrix numerical properties (symmetric positive definiteness)

[Module 04_SpecialMatrices – 05.06]

Solving "Special" linear systems

Last time:

Solved systems $Ax = b$ using basic Gaussian elimination.

Interpret GE as a three step process:

1. LU factorization
    a. Find *triangular* L, U s.t. $A = LU$
2. Forward solve
    a. Solve for z s.t. $Lz = b$
3. Backward solve
    a. Solve for x s.t. $Ux = z$

Today: Special Systems

Can we exploit matrix properties for efficiency in specific (common) cases?

Systems of interest:

1. Symmetric systems ($A = A^T$)
2. Positive definite systems (A satisfies $x^TAx > 0, \forall x \neq 0$)
3. Symmetric positive definite (both 1 and 2)
4. Banded systems
5. Tri-diagonal
6. General sparse

[1] Symmetric Systems

Consider systems satisfying $A = A^T$

e.g., $\begin{bmatrix} 1 & 2 & 5 \\ 2 & 4 & 4 \\ 5 & 4 & 7 \end{bmatrix}, \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 5 & 0 \\ 2 & 0 & 4 & 2 \\ 5 & 4 & 7 & 0 \\ 0 & 2 & 0 & 7 \end{bmatrix}$

(clearly such systems must also be square)

LDM^T Factorization

Consider a new factorization $A = LDM^T$ (variation on usual LU)

Theorem: if A possesses a LU factorization, then there exist unique lower triangular matrices L and M, and a diagonal matrix D such that:

$A = LDM^T$

Note: if the leading principal submatrices of A are nonsingular, it has a unique LU factorization (no pivoting required).

Can show by induction on n.

Recall: the k^{th} leading principal submatrix of A is the smaller matrix obtained by deleting the n-k rows and column of A

Proof: <04_example, p1>

LDT^T Factorization

Flop count is essentially unchanged: flops(LU) ≈ flops(LDM^T)

Valid for both symmetric and non-symmetric matrices.

But symmetry provides an advantage:

- By skipping redundant computation, can save ~half the cost of LU.

Theorem: if A is also symmetric, then $A = LDL^T$ (i.e. M = L)

Proof: <04_example, p2>

[2] Positive Definiteness

Definition: *A* is positive definite if $x^TAx > 0$ for all vectors $x \neq 0$
(roughly generalizes positive scalars *a*: $xax > 0$ for all values $x \neq 0$)
Positive definiteness also implies invertibility.

A useful theorem

Theorem: if $A \in R^{n \times n}$ is positive definite (PD), and $X \in R^{n \times k}$ with rank k (≤ n), then $B = X^TAX$ is also PD (ie. $z^TBz > 0, \forall z \neq 0$)
Proof:
let $z \in R^{k \times 1}$. Then $z^TBz = z^TX^TAXz = (Xz)^TA(Xz)$
let $x = Xz$. Is $x = 0$?
Not unless z = 0. Since we said that *X* is full rank *k* (full rank).
Since by definition of PD, $x^TAx > 0$, then $(Xz)^TA(Xz) > 0$ -> $z^TBz > 0$ -> B is positive definite

Corollaries

1. If A is PD then all its principal submatrices are PD. In particular, diagonal entries are positive. Why?
   - Can choose the matrix *X* to "pick out" arbitrary principal submatrices.
     Recall: a principal submatrix is a smaller matrix produced by deleting a set of rows and corresponding columns.
2. If A is PD, then diagonal D of $A = LDM^T$ has strictly positive entries.

<proof: 04_examples, p3>

[3] Symmetric Positive Definite Systems

Both symmetric ($A = A^T$) and PD ($x^TBx > 0, \forall x \neq 0$).
Abbreviate as SPD.

Theorem: if A is SPD, then there exists a unique lower triangular matrix G such that:

$A = GG^T$

This is the **Cholesky factorization** and G is the **Cholesky factor**.
<proof: 04_examples, p3>
Proof: existence of Cholesky factorization

Symmetry –> $A = LDL^T$
Positive definite -> $D = diag(d1, d2, \ldots, d_n)$, $d_i > 0$
Let $D^{1/2} = diag(sqrt(d1), sqrt(d2), \ldots, sqrt(d_n))$
Let $G = LD^{1/2}$. G is lower triangular.
Then $GG^T = LD^{1/2}(D^{1/2}L^T) = LDL^T = A$

Cholesky Factorization Algorithm

Note: this exploits symmetry by working only with sub-diagonal entries.

```
For k = 1:n
        a_kk = sqrt(a_kk)                    Iterate from top to bottom along diag.
                                             Factor the diagonal element. (sqrt(α))
        For j = k+1:n
                a_jk = a_jk / a_kk           Update current column entries
        End                                  below the diagonal. (v := v/sqrt(α))
        For j=k+1:n
                For i=j:n
                        a_ij = a_ij - a_ik a_jk   Update lower right block (below the
                End                               diagonal only). B := B - vv^T/α
        End
End
```

Cost of Cholesky

Focusing on the innermost loop, we have:

$$\sum_{k=1}^{n} \sum_{j=k+1}^{n} \sum_{i=j}^{n} 2$$

Through the usual process, can ultimately show that:

Flops(Cholesky) = $\frac{n^3}{3} + O(n^2)$

Cholesky flop count (in detail)

<math, s13>

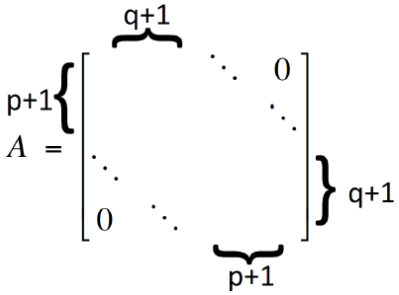
Banded Systems – Examples

Only "bands" adjacent to the diagonal are non-zero:

e.g. $\begin{bmatrix} 3 & -1 & & & \\ 2 & 3 & -1 & & \\ 1 & 2 & 3 & -1 & \\ & 2 & 2 & 3 & -1 \\ & & 3 & 2 & 3 \end{bmatrix}, \begin{bmatrix} 3 & -1 & 2 & & & \\ & 3 & -1 & 1 & & \\ & & 5 & -2 & -3 & \\ & & & 3 & -1 & 2 \\ & & & & 3 & 2 \\ & & & & & 3 \end{bmatrix}$

Band Systems

In general, we have a form like:



Definition: A has:

- Upper bandwidth q if $a_{ij} = 0$ for $j > i + q$
- Lower bandwidth p if $a_{ij} = 0$ for $i >$ j + p

Basically # of bands above (q) or below (p) the main diagonal.

e.g. what are p and q here?

$\begin{bmatrix} 3 & -1 & & & \\ 2 & 3 & -1 & & \\ 1 & 2 & 3 & -1 & \\ & 2 & 2 & 3 & -1 \\ & & 3 & 2 & 3 \end{bmatrix}$, $\begin{bmatrix} 3 & -1 & 2 & & & \\ & 3 & -1 & 1 & & \\ & & 5 & -2 & -3 & \\ & & & 3 & -1 & 2 \\ & & & & 3 & 2 \\ & & & & & 3 \end{bmatrix}$

q=1, p=2 q=2, p=0

Banded Systems

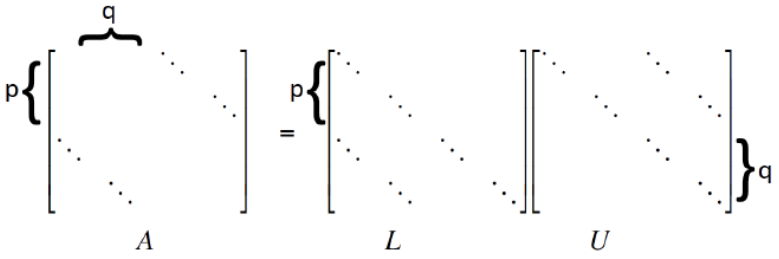
(see graphic above)

Consider how you might store such matrices.

Factoring Banded Matrices

If A is banded, so are these factorizations: *LU*, GG^T, and LDM^T

Theorem: let A = *LU*. If A has upper bandwidth q and lower bandwidth p, then U has upper bandwidth q, and L has lower bandwidth p.



Named Variations of Band Matrices

Matrix Type	p (lower bandwidth)	q (upper bandwidth)
Diagonal	0	0
Upper triangular	0	n-1
Lower triangular	m-1	0
Tridiagonal	1	1
Upper bidiagonal	0	1
Lower bidiagonal	1	0
Upper Hessenberg	1	n-1
Lower Hessenberg	m-1	1

Algorithm – Band LU Factorization

```
For k=1:n-1                                          Iterate over all rows.
    For i=k+1:min(k+p,n)                             }
        a_ik = a_ik/a_kk                             } Determine multiplicative factors
    End                                              }
    For i=k+1:min(k+p,n)                             }
        For j=k+1:min(k+q,n)                         } Subtract scaled row data, only in the
            a_ij = a_ij − a_ik a_kj                  } non-zero bands.
        End                                          }
    End                                              }
End
```

Band LU – Cost

If n >> p and n >> q, flops(band GE) ≈ 2npq

Compare to $\frac{2n^3}{3}$ for naïve LU – band LU can be must faster!

Exercises:

1. Verify cost of band LU
2. Work out *efficient* algorithms for band forward/backward solves

Summary

Depending on matrix properties/structure, LU factorization can often be dramatically sped up.

We considered several example types: symmetric, SPD, banded, etc.

[Module 05 – General Sparse Matrices and Poisson problems – 05.11]

Recap

Last time:

- Considered variants of LU factorization for symmetric positive definite, and band(ed) matrices

Points to follow up on:

1. Intuition for positive definiteness
2. Motivation for the form of Cholesky factorization

Positive Definiteness – Intuition

Recall: A is positive definite if $x^TAx > 0 \; \forall x \neq 0$

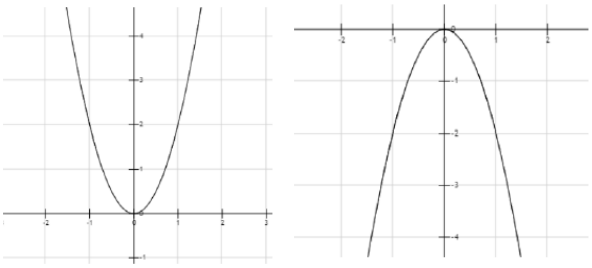
Intuition?

We can write a general quadratic in the form:

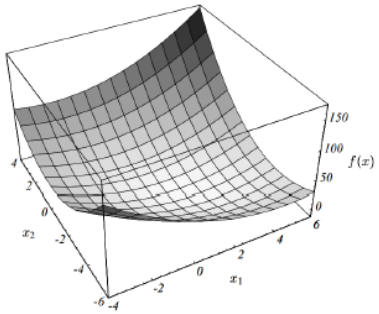
$$f(x) = x^TAx + b^Tx + c$$

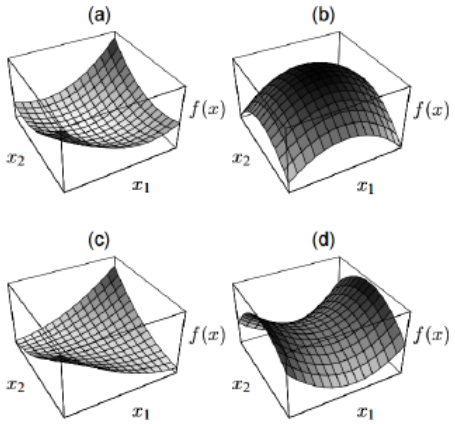
where x,b,c are vectors.

For 1D quadratic, $f(x) = ax^2 + bx + c$ describes a parabola. Positive *a* implies concave up. Negative implies concave down.



In higher dimensions, positive definite A in $f(x) = x^TAx + b^Tx + c$ implies function is concave up.





(a) Positive definite ($x^TAx > 0$)
(b) Negative definite ($x^TAx < 0$)
(c) Positive semi-definite ($x^TAx \geq 0$)
(d) Indefinite (positive and negative eigenvalues)

Will revisit these ideas when we discuss *iterative* methods for Ax = b

Cholesky Factorization – Derivation

Viewed building the Cholesky factorization as a recursive process, one row/column at a time using:

$$A = \begin{bmatrix} \alpha & v^T \\ v & B \end{bmatrix} = \begin{bmatrix} \sqrt{\alpha} & 0 \\ v/\sqrt{\alpha} & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & B - vv^T/\alpha \end{bmatrix} \begin{bmatrix} \sqrt{\alpha} & v/\sqrt{\alpha} \\ 0 & I \end{bmatrix}$$

The final result must be of the form:

$$A = GG^T = \begin{bmatrix} g_{11} & 0 \\ G_{21} & G_{22} \end{bmatrix} \begin{bmatrix} g_{11} & G_{21}^T \\ 0 & G_{22}^T \end{bmatrix} = \begin{bmatrix} g_{11}^2 & g_{11}G_{21}^T \\ g_{11}G_{21} & G_{22}G_{22}^T + G_{21}G_{21}^T \end{bmatrix}$$

Therefore

$$A = \begin{bmatrix} \alpha & v^T \\ v & B \end{bmatrix} = \begin{bmatrix} g_{11}^2 & g_{11}G_{21}^T \\ g_{11}G_{21} & G_{22}G_{22}^T + G_{21}G_{21}^T \end{bmatrix}$$

Which implies:

$$g_{11} = \sqrt{\alpha}$$
$$G_{21} = v/g_{11} = v/\sqrt{\alpha}$$
$$B - G_{21}G_{21}^T = B - vv^T/\alpha = G_{22}G_{22}^T$$

Cholesky Factorization – Example

$$A = \begin{bmatrix} 9 & 3 & 0 \\ 3 & 5 & 2 \\ 0 & 2 & 17 \end{bmatrix}$$

Find the Cholesky factors of A. <example: 05_C, p1>

Special Matrices

We've seen:

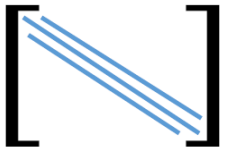
- Symmetric
- Symmetric positive definite
- Banded

Two more common cases:

- Tridiagonal
- General sparse

Tridiagonal Matrices

A special case of band matrices: p = q = 1



One can show that flops(LU(tridiag)) = O(n)
Example will be shown later

General Sparse Matrices

For most problems, number of non-zeroes per row is ***constant***.
e.g. number of non-zeroes is O(n)
store only the non-zero entries.

For GE/LU, main computation is: $a_{ij} = a_{ij} - a_{ik}a_{kj}/a_{kk}$
Since most entries are zero, we try to skip operating on them.

Sparse Matrix Storage

Various storage formats exist (we'll mostly let MATLAB deal with it)
A common example is Compressed Row Storage (CRS or CSR)

- Array of non-zero entries ("val"). Length = nnz
- Array of column indices ("collnd"). Length = nnz
- Array of indices where each row starts ("rowPtr"). Length = # of rows.

Sparse Matrix Storage – CSR example

$$\begin{bmatrix} 2 & & 5 & \\ & 3 & & \\ & 6 & -3 & \\ & & 10 & 2 \end{bmatrix}$$

$$val = [2, 5, 3, 6, -3, 10, 2]$$
$$colInd = [1, 3, 2, 2, 3, 3, 4]$$
$$rowPtr = [1, 3, 4, 6]$$

Val: array of the non-zero entries, ordered by smallest to largest, using ij
Collnd: array of column indices, 1:1 correspondence with val; i-th entry of val in column specified by i-th entry of collnd
rowPtr: array of indices where row starts; e.g. second entry of rowPtr is 3, so row 2 starts at 3rd entry of val/collnd

LU on Sparse Matrices – Fill-in/Fill

Key point: even if A is sparse, its factorization **may not** be.
Classic example: "arrow matrix". LU factors are fully dense.

$$A = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & & & \\ \times & & \times & & \\ \times & & & \times & \\ \times & & & & \times \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ \times & 1 & & 0 & \\ \times & \times & \ddots & & \\ \times & \times & \times & \ddots & \\ \times & \times & \times & \times & 1 \end{bmatrix} \begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & 0 & & \times & \times \\ & & & & \times \end{bmatrix}$$

1. Why?
2. What is the storage cost of factors, and complexity of factorization?

$O(n^2)$ $O(n^3)$

A re-ordering of the system's rows/columns yields a matrix whose LU factorization suffers **no fill-in** (new non-zeros)

Partial Differential Equations (PDE)

Differential equations involving multi-variable functions and (partial) derivatives
A common source of sparse linear systems is numerical solutions of PDEs (e.g. finite difference/finite volume/finite element methods).

Describe many phenomena:

- Fluid flow
- Electromagnetism
- Sound
- Financial problems
- Etc.

Application Problem: Heat Conduction

At steady state, the heat distribution in a uniform material can be modeled using a *Poisson equation*:

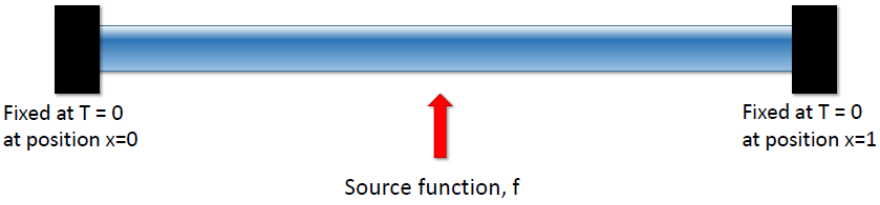
$$-\left(\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}+\frac{\partial^2 T}{\partial z^2}\right)=f \text{ or equivalently } -\Delta T = f$$

Where *x*, *y*, *z* are spatial coordinates, T(x,y,z) is temperature, and f(x,y,z) is a given heat source function.

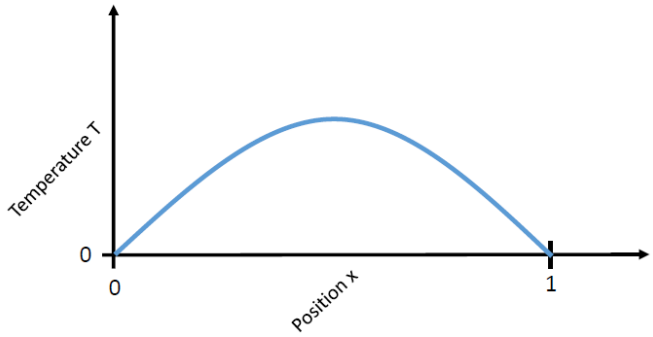
Heat Conduction in a 1D Bar

Interior Equation: $-\frac{\partial^2 T}{\partial x^2}$ = f on (0,1)

Boundaries: T(0) = 0, T(1) = 0



Temperature Distribution in a 1D Bar



Finding a Numerical Solution

We want to find an approximate *numerical solution*, given *f* and the boundary temperatures.

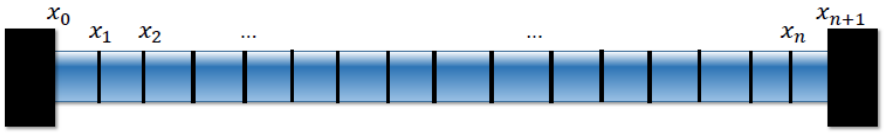
Approach: *discretize* the material into subintervals, and *approximate* the derivatives with finite differences

Discretizing the Domain

Dice the length of the bar into chunks:

$0 = x_0 < x_1 < x_2 < \ldots < x_n < x_{n+1} = 1$

The $\{x_i\}$ positions are called gridpoints.



We let T_i be the numerical approximations of $T(x_i)$.

Assume T = 0 at boundaries, so $T_0 = T_{n+1} = 0$

Unknowns are the *n* temperature values $T_1, T_2, \ldots, T_n$ at gridpoints.

Assume evenly spaced intervals. Define grid spacing *h* as:

$$h = x_i - x_{i-1} = \frac{1}{n+1}$$
$$= domain\ length\ /\ \#\ of\ gridpoints$$

Recall: finite differences are discrete approximations of derivatives.

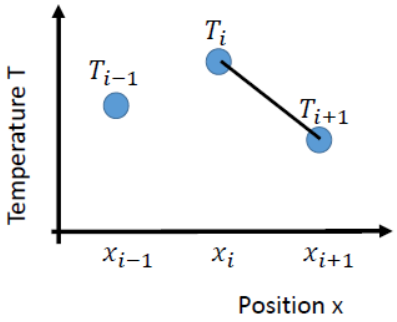
$$\text{e.g., } \frac{\partial f}{\partial x} \approx \frac{f(x_i)-f(x_{i-1})}{x_i-x_{i-1}} = \frac{f(x_i)-f(x_{i-1})}{h}.$$

Use finite differences to approximate $-\frac{\partial^2 T}{\partial x^2}$

Resulting relationships between gridpoints determine *T*.

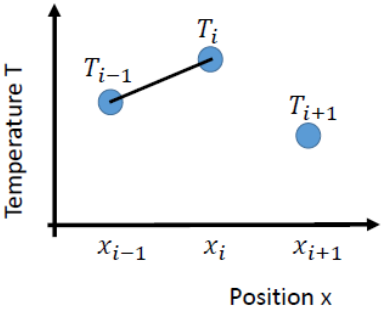
Finite Differences – Gradient

Consider point x_i with value T_i and its neighbours.



Forward Difference (1st deriv.):

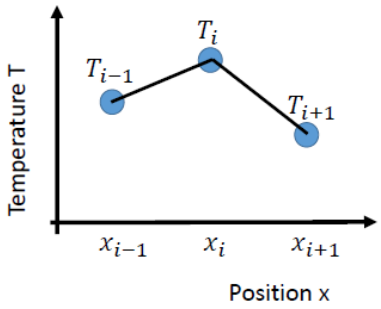
$$\frac{\partial T}{\partial x}(x_i^+) \approx \frac{T_{i+1} - T_i}{h}$$



Backward Difference (1st deriv.):

$$\frac{\partial T}{\partial x}(x_i^-) \approx \frac{T_{i+1} - T_i}{h}$$

Finite Differences – Laplacian (Δ)



Centred difference (2nd deriv.):

$$\frac{\partial^2 T}{\partial x}(x_i) \approx \frac{\frac{\partial T}{\partial x}(x_i^+) - \frac{\partial T}{\partial x}(x_i^-)}{h}$$

$$= \frac{\frac{T_{i+1} - T_i}{h} - \frac{T_i - T_{i-1}}{h}}{h}$$

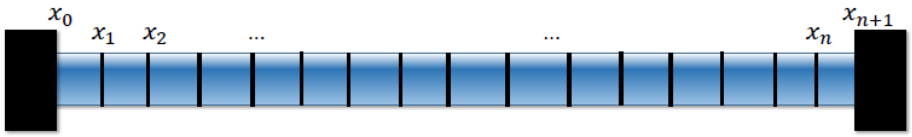
$$= \frac{T_{i+1} - 2T_i + T_{i-1}}{h^2}$$

Linear System of **n** equations

Each grid point i∈[1,n] has one equation relating to its neighbours:

$$-\frac{\partial^2 T}{\partial x^2} = f \quad \Rightarrow \quad -\frac{T_{i+1} - 2T_i + T_{i-1}}{h^2} = f_i$$

Continuous Equation — Discrete Equation



Example

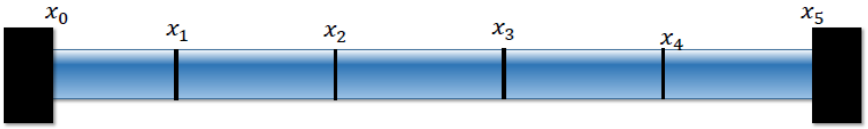
Consider the case $n = 4$ grid points.

Boundaries are: $T_0 = 0$, $T_5 = 0$

Source function at each point given by f_i

Grid spacing $h = \frac{1}{n+1} = 1/5$

What is the matrix?



General Form – 1D Laplacian

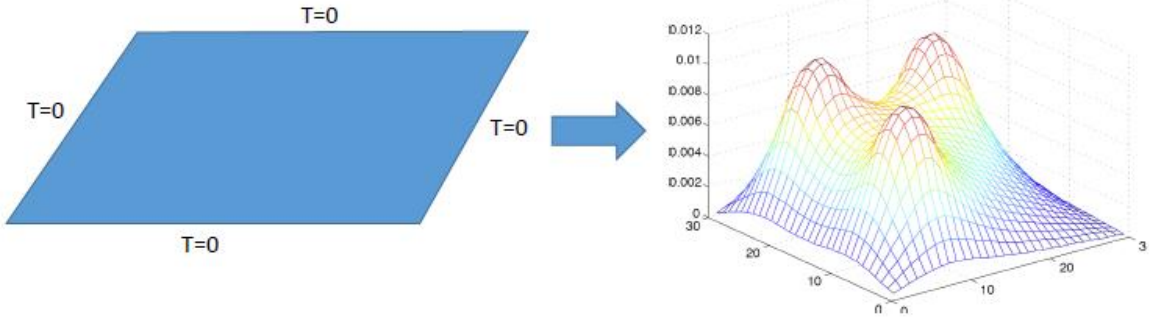
$$\frac{1}{h^2}\begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & \dots & & \\ & \dots & \dots & \dots & \\ & & \dots & 2 & -1 \\ & & & -1 & 2 \end{bmatrix}\begin{bmatrix} T_1 \\ T_2 \\ \dots \\ T_{n-1} \\ T_n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \dots \\ f_{n-1} \\ f_n \end{bmatrix}$$

What can we say about the matrix structure?

Band matrix, but more specifically symmetric, tridiagonal.

Heat conduction in a 2D plate

Given a rectangular plate, with zero temperature boundaries and a source function f, determine the heat distribution.

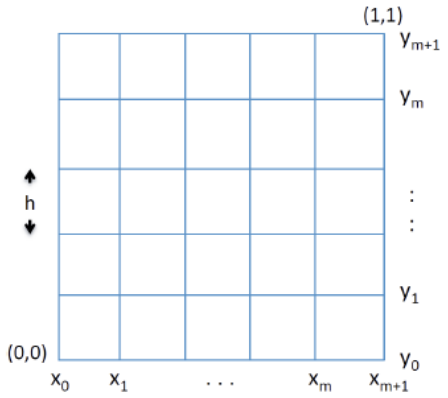


2D Computational Grid

Gridpoints now indexed by (i,j)

e.g. $T(x_{i,j}) \approx T(i,j)$

We need to approximate the 2D equation at each point: $-\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) = f$



2D Discrete Poisson equation

Approximate the 2nd derivative in each direction separately, and sum.

$$-\left(\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}\right)=f$$

$$\downarrow$$

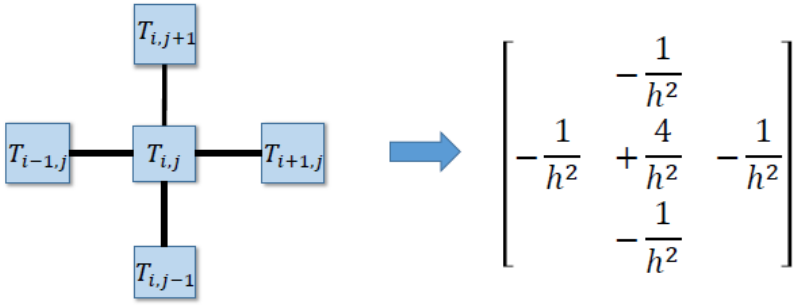
$$-\left(\frac{T_{i+1,j}-2T_{i,j}+T_{i-1,j}}{h^2}+\frac{T_{i,j+1}-2T_{i,j}+T_{i,j-1}}{h^2}\right)=f_{i,j}$$

$$\downarrow$$

$$\frac{4}{h^2}T_{i,j}-\frac{1}{h^2}T_{i+1,j}-\frac{1}{h^2}T_{i-1,j}-\frac{1}{h^2}T_{i,j+1}-\frac{1}{h^2}T_{i,j-1}=f_{i,j}$$

The Finite Difference ***Stencil***

A convenient visual notation for the equation centered at each point.



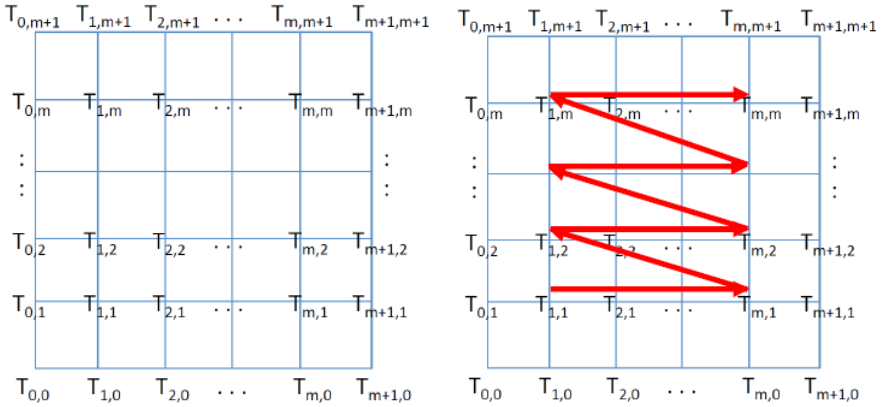
Numbering the Unknowns

To put into matrix for, need to "flatten" the indices from 2D (i,j) to 1D (k)

Boundary values are 0.

Unknowns are $T_{i,j}$ for all $i,j \in [1,m]$. Therefore there are m^2 unknowns.

How to index them?



Natural ordering: number points in x-axis first, then y-axis.

$T_{1,1}$, $T_{2,1}$, …, $T_{m,1}$; $T_{1,2}$, $T_{2,2}$, … etc.

If $T_k = T_{i,j}$ then we can index with:

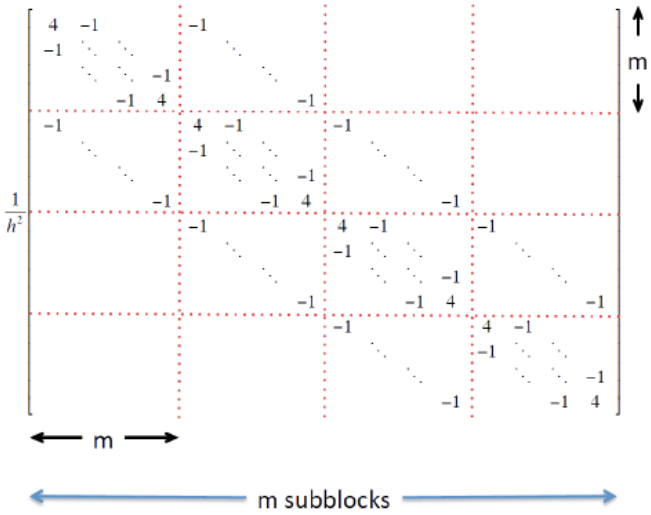
$k = i + (j - 1)m$

Example: 05_C, p2

Resulting 2D Matrix Structure

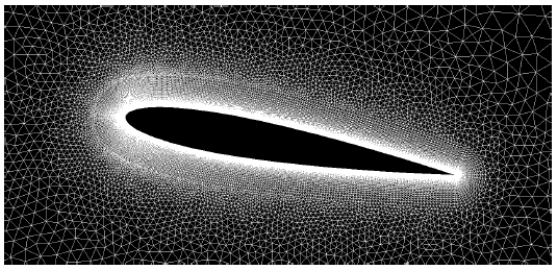
5 bands:

- 1 diagonal
- 2 immediately above/below the diagonal
- 2 separated by *m* entries



Finite Element/Finite Volume

Other types of PDE problems, discretizations and meshes give rise to different matrix structures and properties.



A triangular mesh for finite volume study of flow around an airfoil.

Summary

PDEs often give rise to sparse linear systems.

We considered the specific case of steady state heat distribution, governed by a particular Poisson equation.

In 1D -> a tridiagonal matrix system
In 2D -> a particular sparse matrix with 5 bands.

Next time: the natural ordering may not produce minimal fill.
Can we re-index or **re-order** the unknowns to reduce the fill suffered during factorization?

A **graph** view of matrix structure can help us.

[Module 06 – Matrix Graphs and Reordering - 05.13]

Brief Application Example

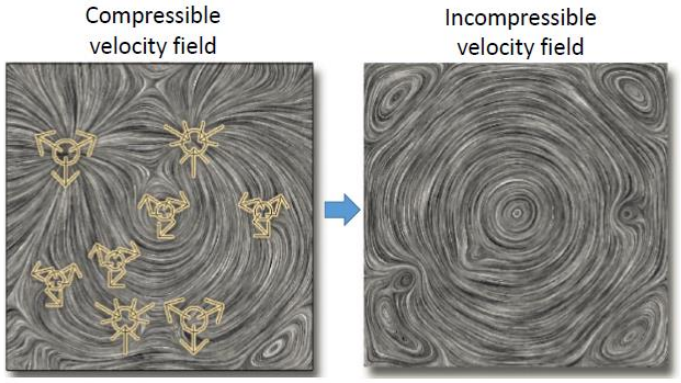
For each frame of animation, we need to solve a Poisson equation to find the fluid's pressure p:

$$\Delta p = \nabla \cdot u^*$$

Where u* is the current fluid velocity.

The pressure force eliminates compression, to preserve air volume and yield nice "swirly" flow.

Incompressibility



Smoke animation

The grid structure doesn't change, so we can factor the matrix once, and just do forward/backward solve at each step with a different RHS.

Much faster:

$O(n^2)$ vs $O(n^3)$

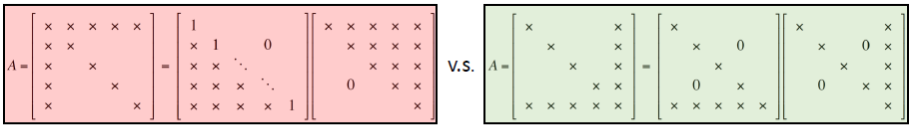
Moral: understanding the numeric of your problem can be very useful in practice

Last Time

We saw the *Poisson* equation $-\Delta T = f$ modeling heat conduction.

- An example of a PDE that gives rise to (a particular class of) sparse matrices

Mentioned that general sparse matrices may have **dense LU factors**.



Complete fill! Zero fill!

Today

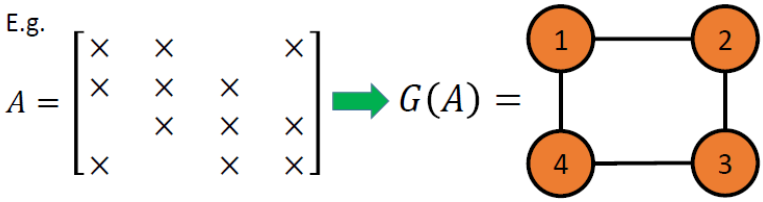
- Graph representations of (symmetric) matrices
- Effect of factorization/fill on the graph view
- Common *matrix reordering* methods that reduce fill

Graph Structure of Matrices

Given sparse, *symmetric* A, create an undirected graph G(A) with:

- A node *i*, for each row *i*
- An edge *i* <-> *j*, for each $a_{ij} \neq 0$

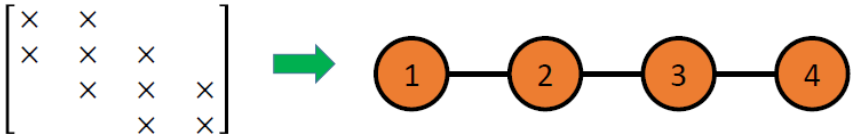
E.g.



Graph structure often has a **physical** or **geometric interpretation**.

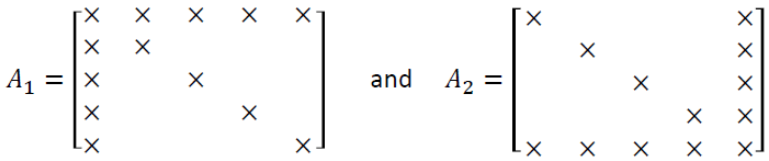
e.g. Laplacian matrix from the Poisson problem recovers the underlying grid structure.

1D Laplacian:

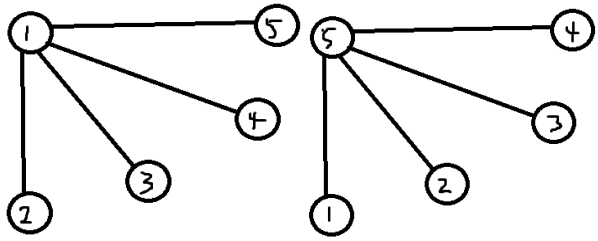


2D Laplacian:

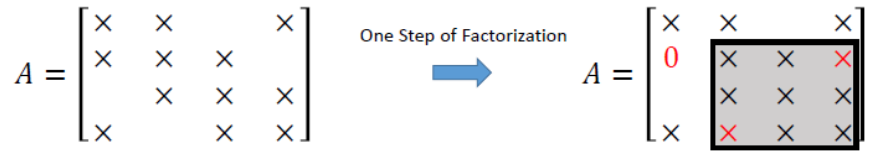
What are the graphs of these "arrow" matrices?

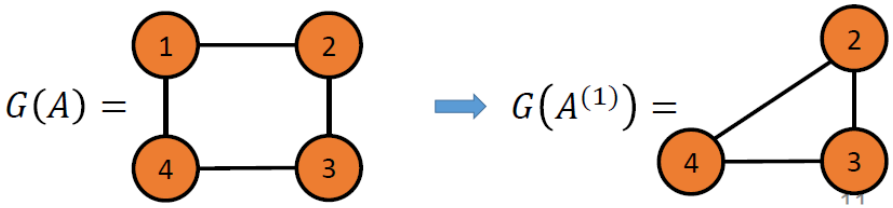


A_1 A_2



Graph Relationship to Fill During Factorization





What happened to the graph?

Elimination of node *i* yields a new graph with:

1. Node *i* and all its edges deleted
2. New edge j <-> k added between all node pairs incident on *i* in the old graph (corresponds to fill-in)

Reordering – Key Principle

Graph structure of a symmetric matrix is clearly unchanged by renumbering its nodes.
But, the resulting (symmetric) permutation of the matrix may suffer vastly different levels of fill-in during factorization.

Goal: how to **renumber** the nodes to **minimize fill-in**?

Ordering Algorithms

Finding the true optimum is NP-complete.
But many effective heuristic strategies exist.

Look at the common ones:

- Envelope/level-set methods
- (reverse) Cuthill-McKee
- Markowitz
- Approximate Minimum Degree

Reordering in Matrix Form

Make use of *permutation matrices*, *P*.
Copies of identity matrix *I*, with (some) rows swapped around.

$$\text{e.g. } \begin{bmatrix} & 1 & \\ & & 1 \\ 1 & & \end{bmatrix} \begin{bmatrix} 3 & 2 & 5 \\ 2 & 4 & 1 \\ 5 & 1 & 3 \end{bmatrix} = \begin{bmatrix} 2 & 4 & 1 \\ 5 & 1 & 3 \\ 3 & 2 & 5 \end{bmatrix}$$

To preserve symmetry, we symmetrically swap rows *and* columns.

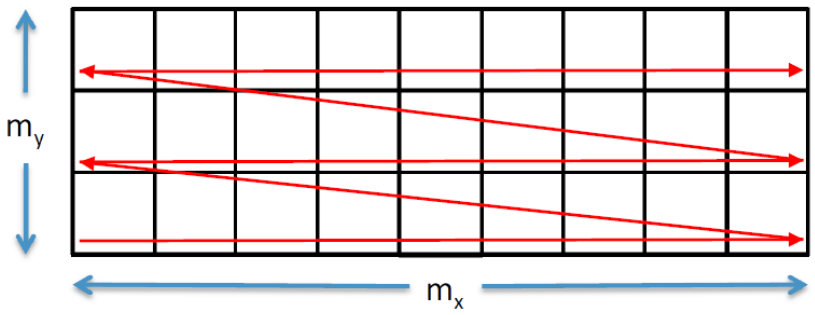
$$A^{new} = PAP^T$$

$$\text{e.g. } \begin{bmatrix} & 1 & \\ & & 1 \\ 1 & & \end{bmatrix} \begin{bmatrix} 3 & 2 & 5 \\ 2 & 4 & 1 \\ 5 & 1 & 3 \end{bmatrix} \begin{bmatrix} & & 1 \\ 1 & & \\ & 1 & \end{bmatrix} = \begin{bmatrix} 4 & 1 & 2 \\ 1 & 3 & 5 \\ 2 & 5 & 3 \end{bmatrix}$$

**Can undo the row/column swaps by P1'*A^{new}*P2'
This corresponds to renumbering graph nodes.
The *system* we solve is $PAP^TPx = Pb$. Same solution, different variable order.
(i.e. letting $B = PAP^T$, $c = Pb$, and $y = Px$, we see we have a new system $By = c$ with the symmetrically permuted matrix)

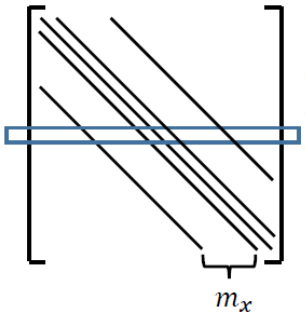
Band Matrices

Consider a matrix with this graph (e.g. 2D Laplacian), with $m_x >> m_y$.



Band Matrices – Natural Ordering

We saw the *natural* ordering last class: first along x-axis, then y-axis.
What is the bandwidth of this matrix?



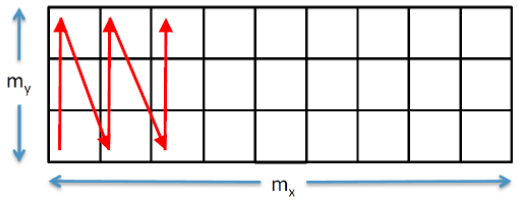
Row i has entries at columns…

- i (diagonal)
- $i-1, i+1$ (inner bands)
- $i-m_x, i+m_x$ (outer bands)

Bandwidth is m_x.

Band Matrices – A Different Ordering

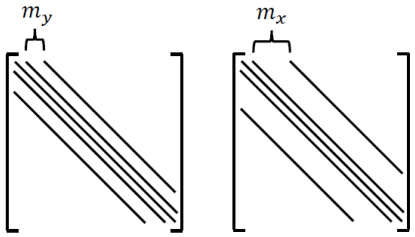
What if we number along y-axis, and then x-axis?



Bandwidth becomes m_y instead.

Comparison of Two Orderings

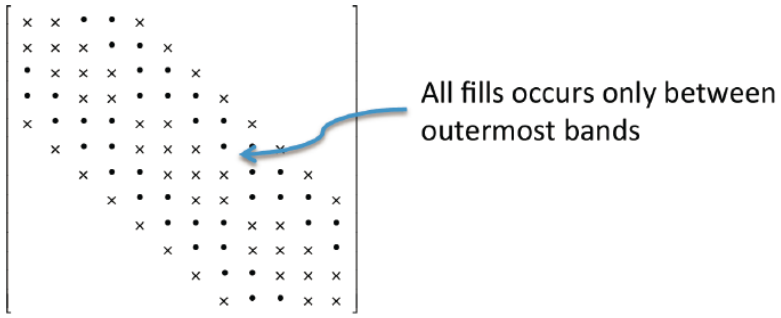
Numbering along the y-axis first, gives narrower bandwidth (in this case).
Which produces less fill? Why?



m_y produces less fill because – narrower.

Factoring Band Matrices

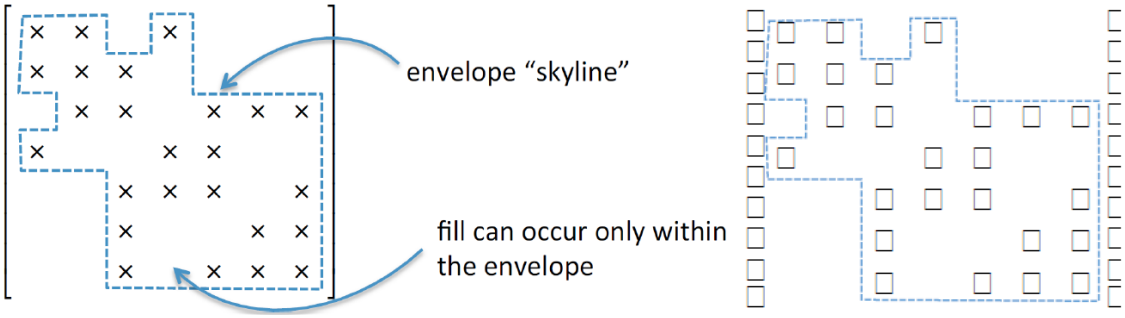
Recall: the *L* and *U* factors of a band matrix have the same upper and lower bandwidths, respectively, as the input A.
i.e. the band structure is preserved.



We saw that flops(band GE) ≈ O(npq) for bandwidths p and q. Therefore cost $O(m^2n)$ for bandwidth m and n gridpoints.
X-first order: flops = $O(m_x^2 n)$
Y-first order: flops = $O(m_y^2 n)$
But what can we do for more general sparsity patterns?

Envelope Methods

In practice, bandwidth may vary a lot *per* row.

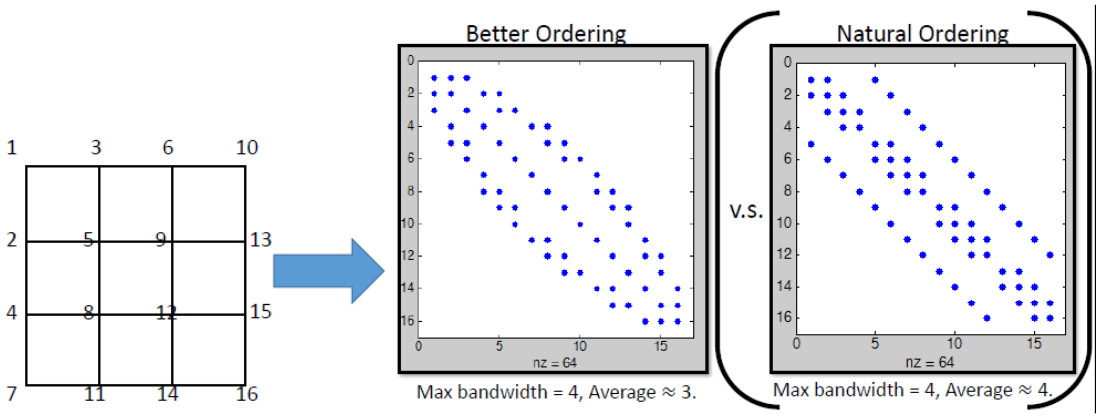


- In each row of *L*, fill can only occur between 1st non-zero entry and the diagonal.
- Aim to limit fill by keeping the envelope close to the diagonal

Q: what does this imply about a "good" numbering of nodes?
A: graph neighbours should have numbers as close together as possible.

Envelope Methods – Example

Graph neighbours should have numbers close together as possible to minimize the envelope.



Envelope Methods

Based on "level sets", S_i

S_1 = the single starting node

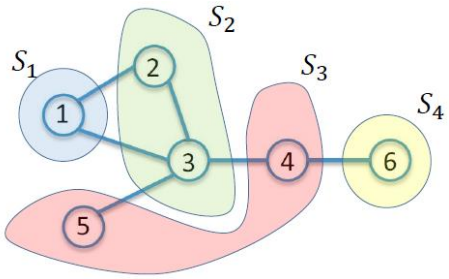
S_2 = all graph neighbours of the node in S_1

S_3 = all graph neighbours of nodes in S_2

…

S_i = all graph neighbours of nodes in S_{i-1}, node in S_1, S_2, …, S_{i-1}

Ordering: nodes in S_1, then nodes in S_2,…



Basically a breadth-first traversal.

How to order nodes *within* each level set?

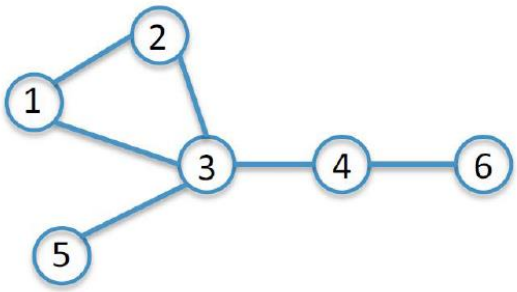
Definition: Degree

Degree of a node = number of adjacent nodes (or # of incident edges)

Deg(node 3) = 4

Deg(node 5) = 1

Etc.



Cuthill-McKee (1969)

Within each set, order the nodes in *increasing* order of their degree.

Cuthill-McKee algorithm:

1. Pick a starting node (arbitrary, or use some heuristic), number it 1
2. For *i* = 1 .. n, find all unnumbered neighbours of node *i*, and number them in increasing order of degree.

Reverse Cuthill-McKee

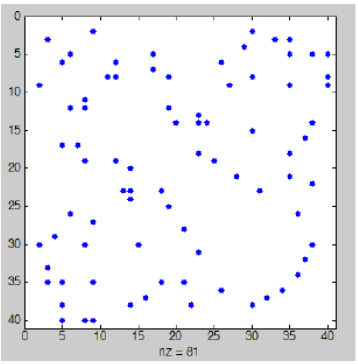
Exactly what it sounds like; take the CM numbering and reverse it.

$$node_i^{RCM} = node_{n-i+1}^{CM} \text{ for } i = 1 \dots n$$

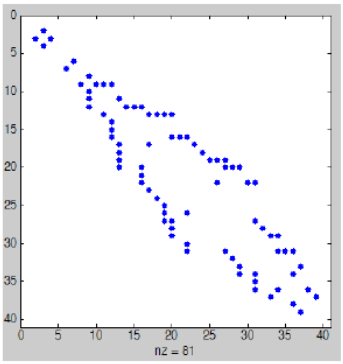
But why?

1. Better observed behaviour in practice.
2. Same envelope, but has patterns more like the low-fill downward arrow matrix, rather than the upward arrow.

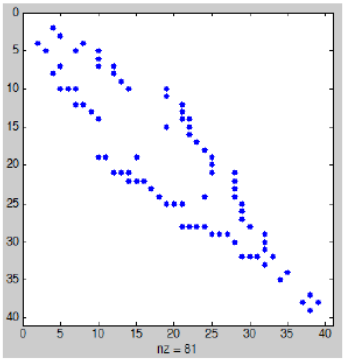
Example



Random Symmetric Matrix



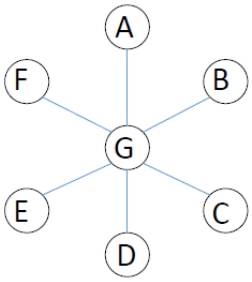
Cuthill-McKee



Reverse Cuthill-McKee

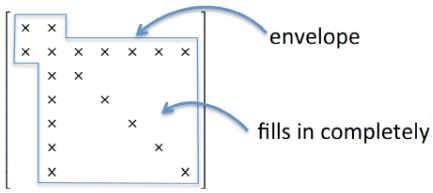
Example #1

Perform CM and RCM on this graph, using A as the start node, and breaking ties alphabetically.

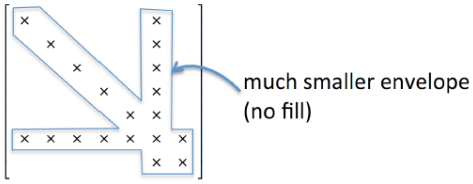


Node #	Node	Unnumbered neighbours (deg)
1	A	G(5)
2	G	B(1), C(1), D(1), E(1), F(1)
3	B	-
4	C	-
5	D	-
6	E	-
7	F	-

CM:



RCM:

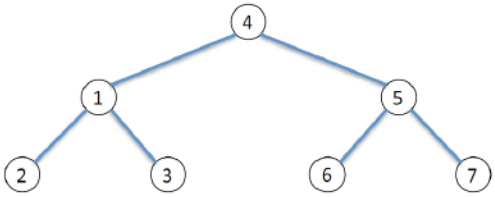


Reverse Cuthill-McKee on Trees

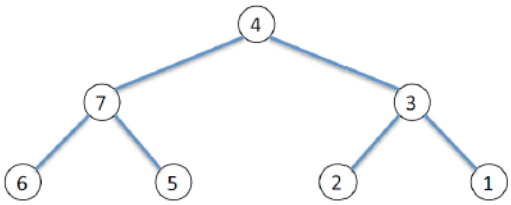
Nice property: if the graph is a tree, RCM produces no fill (not necessarily true for CM)

e.g. the arrow matrix from earlier, or:

CM ordering of a tree:

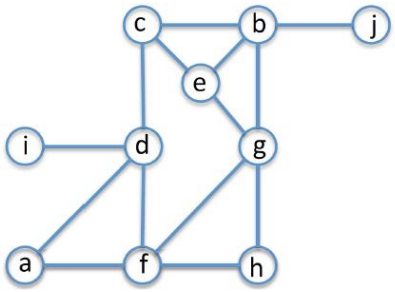


RCM ordering of a tree:



Example #2

Perform CM and RCM on this graph, using G as the start node, and breaking ties alphabetically.



Node #	Node	Unnumbered neighbours	Node #	Node	Unnumbered neighbours
1	G	H(2), E(3), B(4), F(4)	1	I	-
2	H	-	2	D	
3	E	C(3)	3	A	
4	B	J(1)	4	J	
5	F	A(2), D(4)	5	C	
6	C	-	6	F	
7	J	-	7	B	
8	A	-	8	E	
9	D	I(1)	9	H	
10	I	-	10	G	

Summary

A graph provides a useful abstraction of the structure of a symmetric matrix: offers insight into the fill process.

(Reverse) Cuthill-McKee is one family of reorderings that can be helpful in reducing fill, by minimizing the envelope.

Next: local strategy (Markowitz), (approximate) minimum degree orderings

[Module 07 – Minimum Degree Reordering – 05.20]

Example Application: cloth simulation

Solving cloth physics often requires solving a large linear system, with a complex mesh/graph structure.

Quote: "the linear system for implicit time integration is solved using the **sparse Cholesky-based solver** in the TAUCS library." [Narain et al. 2012]

Reorderings – Key Principle

Graph structure of a symmetric matrix is clearly unchanged by renumbering its nodes.

But, the resulting (symmetric) permutation of the matrix may suffer vastly different levels of fill-in during factorization.

Goal: how to **renumber** the nodes to **minimize fill-in**?

Ordering Algorithms

Finding the true optimum is NP-complete.
But many effective heuristic strategies exist.

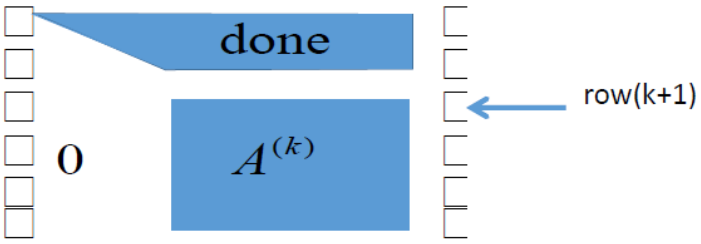
We'll look at some of the common ones:

- Envelope/level-set methods
- (reverse) Cuthill-McKee
- Markowitz
- Minimum Degree

First two from last time, last two today.

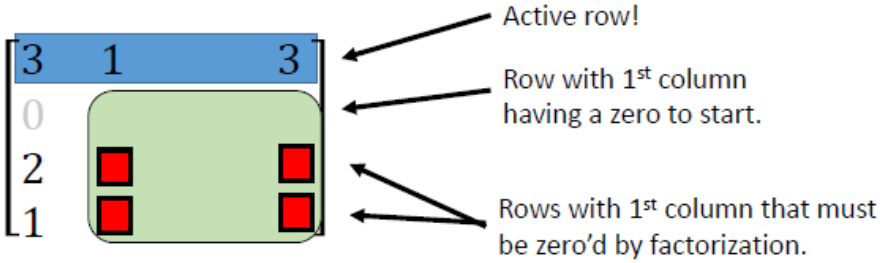
Local Strategy (Markowitz 1957)

Try to (approximately) minimize fill on the current step only.
e.g. after k steps of GE (i.e. LU) we have:



Fill During a Step of LU

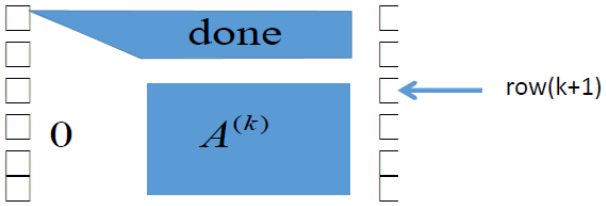
At each step, we subtract multiples of the current row from those below, *if they have a non-zero entry in the column*.



Worst case fill at this step (using this pivot)?
4 entries = (2 other non-zeros in row) x (2 other non-zeros in column)

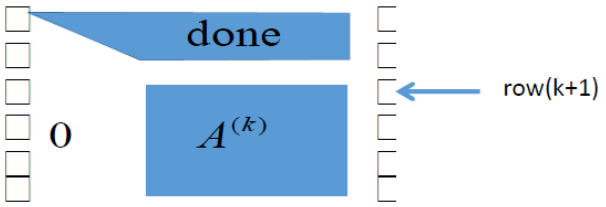
Swapping to Reduce Fill

We can swap rows *and* columns (on the fly) to reduce the resulting fill.



Consider all entries $a_{ij}^{(k)}$ in the lower right block.
Find the one that would minimize the (worst-case) fill.
Swap it into the top-left position of $A^{(k)}$.

Markowitz Product



Let $r_i^{(k)}$ = number of non-zero entries in row *i* of $A^{(k)}$.
$c_j^{(k)}$ = number of non-zero entries in column j of $A^{(k)}$.

Maximum fill = $(r_i^{(k)} - 1)(c_j^{(k)} - 1)$ (The "Markowitz product")
Select $a_{ij}^{(k)}$ that minimizes the Markowitz product, $(r_i^{(k)} - 1)(c_j^{(k)} - 1)$.
Then swap.

Example

What are the Markowitz products for each of the (non-zero) entries of this matrix?

$$\begin{bmatrix} a & b & c & \\ d & e & & \\ & f & g & \\ & h & & i \end{bmatrix}$$

Recall:

$r_i^{(k)}$ = number of non-zero entries in row *I* of $A^{(k)}$
$c_j^{(k)}$ = number of non-zero entries in column *j* of $A^{(k)}$
Markowitz product: $(r_i^{(k)} - 1)(c_j^{(k)} - 1)$

A = (3-1)(2-1) = 2 B = (3-1)(4-1) = 6 C = (3-1)(2-1) = 2 D = (2-1)(2-1) = 1 E = (2-1)(4-1) = 3
F = (2-1)(4-1) = 3 G = (2-1)(2-1) = 1 H = (2-1)(4-1) = 3 I = (2-1)(1-1) = 0

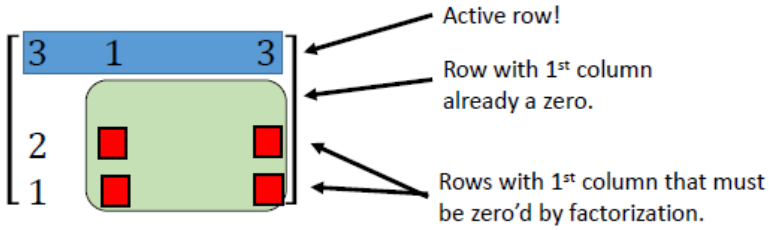
Example – Solution

$$\begin{bmatrix} a & b & c & \\ d & e & & \\ & f & g & \\ & h & & i \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 6 & 2 & \\ 1 & 3 & & \\ & 3 & 1 & \\ & 3 & & 0 \end{bmatrix}$$

Which would we choose to swap and why? How much fill results?
Entry $a_{4,4} = i$, since using it as pivot introduces no fill at all
Its column is already zeroed.

Approximate the fill

Note: why is this just an approximation?



Just estimates the *worst case* fill.

Generally, some of the [red] entries may already be non-zero, so no *new* non-zeros are created.

Local Strategy – Symmetric Case

For symmetric case, min $r_i^{(k)}$ = min $c_j^{(k)}$

So only consider diagonals:

- At each step, find nodes *i*, k + 1 ≤ i ≤ n giving min $r_i^{(k)} - 1$
- Use $a_{ii}^{(k)}$ as the pivot (by symmetrically swapping both rows and columns)

Features:

- Preserves symmetry and diagonal dominance
- Corresponds to node reordering

Symmetric Case – Graph View

What are the $r_i^{(k)} - 1$ for diagonal entries of this symmetric matrix?

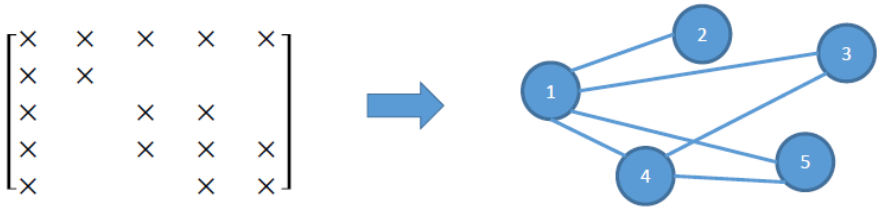
$$\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & & & \\ \times & & \times & \times & \\ \times & & \times & \times & \times \\ \times & & & \times & \times \end{bmatrix}$$

Results: $a_{11} = 4$, $a_{22} = 1$, $a_{33} = 2$, $a_{44} = 3$, $a_{55} = 2$

So we would swap to use a_{22} as a pivot.

What do these values indicate/correspond to in the graph view?

# of off-diagonal entries in the row = degree of corresponding node.



This leads to:

Minimum Degree Ordering

- At each step we are choosing the node with (current) minimum degree
- This is called "minimum degree ordering"
- We can do this up front by renumbering nodes of the graph structure

Recall that a step of factorization corresponds (in the graph) to:

1. Delete the chosen node and all its edges
2. Connect its incident neighbours together with a new edge

Cholesky and Fill – Concrete Example

Matrix:

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

Elimination Graph:

First step of Cholesky:

$\alpha = 2, v = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$.

$$B - \frac{vv^T}{\alpha} = \begin{bmatrix} 2 & \\ & 2 \end{bmatrix} - \frac{1}{2}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 3/2 & -1/2 \\ -1/2 & 3/2 \end{bmatrix}.$$

Result after one step:

$$\begin{bmatrix} \sqrt{2} & 0 & 0 \\ -\frac{1}{\sqrt{2}} & 3/2 & -1/2 \\ -\frac{1}{\sqrt{2}} & -1/2 & 3/2 \end{bmatrix}$$

The zeros have filled in, corresponding to nodes B and C connecting.

Minimum Degree Ordering

- Construct the elimination graph
- At each step:
  - Number the node with (current) least degree
  - Remove the node and its edges
  - Add new edges connecting all its neighbours together
- Relabel the graph accordingly

Tie-Breaking

When multiple nodes have same degree, which to choose?

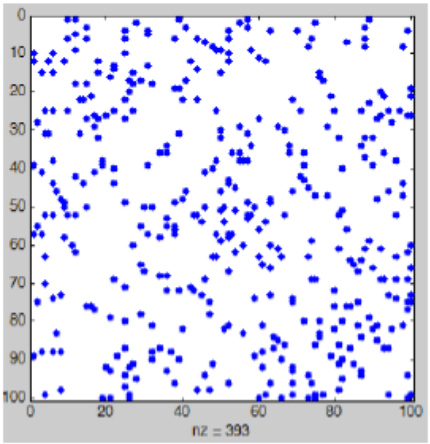
Possible strategies:

1. Select the node with smallest node number in the original ordering
2. Pre-order with RCM: selecting the node that is numbered earlier according to a RCM ordering (computed in advance).
3. Various others (e.g. “multiple minimum degree”: choose multiple nodes that don’t interact and eliminate them at once)

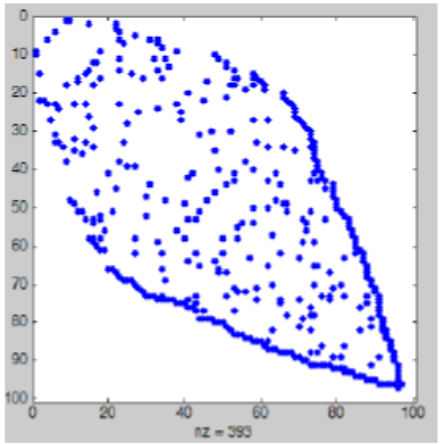
Can have a significant impact in practice.

Example

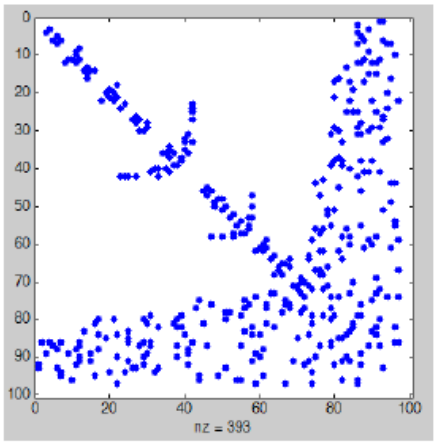
Matlab’s more advanced variant of MD is “symand”, for symmetric approximate minimum degree.



Random Sparse Matrix



Reverse Cuthill-McKee
nnz(L) = 955



Matlab’s symamd
nnz(L) = 800

Example:

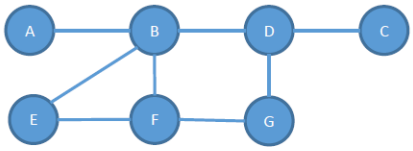
1. Construct the graph for the matrix below
2. Determine its minimum degree re-ordering, breaking ties alphabetically.

MARKOWITZ EX ROW/COL SWAP – HOW??

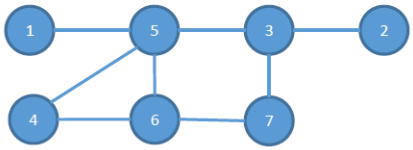
	A	B	C	D	E	F	G
A	×	×					
B	×	×		×	×	×	
C			×	×			
D		×	×	×			×
E		×			×	×	
F		×			×	×	×
G				×		×	×

Solution

Graph:



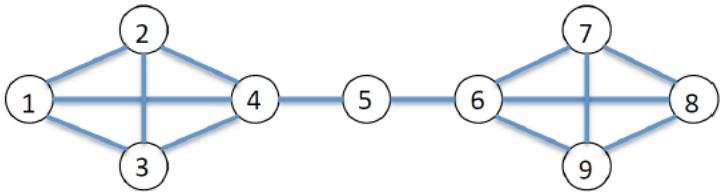
Relabeling:

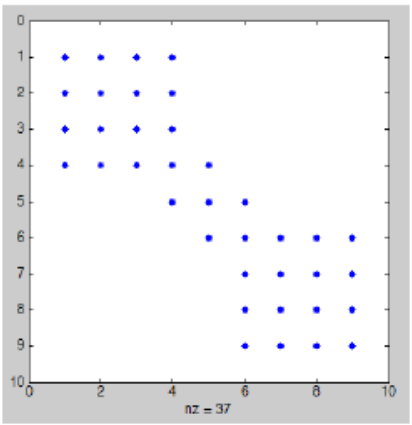


Node # replaces the original letter of the node (refer to notes)

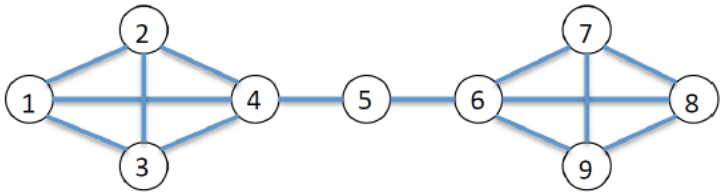
Minimum Degree Ordering – Notes

- A tree graph will always be ordered correctly
- Still a *local* strategy – no guarantee of absolute minimum total fill (which is NP-complete)
  - Counter example showing non-optimality





Counterexample



The above ordering suffers no fill. How can we tell this by examining the matrix?
Factorization can only fill in the "envelope", but its envelope is already totally filled.

	5	6	4
5	×	×	×
6	×	×	•
4	×	•	×

fill in

How can we see that the minimum degree ordering would lead to fill here?

MD ordering would eliminate node 5 first (degree 2).
This would immediately connect nodes 4 and 6, corresponding to **fill**.

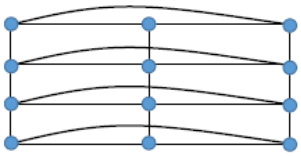
Interesting Extensions of Minimum Degree

Many improvements are possible:

- "supervariables" / indistinguishable nodes: nodes with identical adjacency structure (neighbours) can be eliminated simultaneously
- Multiple elimination: *non-adjacent* nodes of same degree can also be safely eliminated simultaneously
- *Approximate* minimum degree: use an *approximation* to the degree updates of neighbours, which improves the run-time
- Quotient graph: smarter graph representation to reduce storage

Problem – periodic domain, RCM and MD

Consider a matrix with the following periodic/wrapped graph/grid structure:



1. Number the nodes with a natural ordering, starting from bottom-left, proceeding first alone X, then Y.
2. Compute a (regular) Cuthill-McKee ordering, starting from node 5.
3. Determine the minimum degree ordering.
4. Sketch the resulting matrix sparsity structure in each case.

Break ties using the ordering of part (1).

Summary

- Remember: our overall goal is to minimize computation and storage costs of factorization on sparse matrices, by limiting fill
- *Minimum degree ordering* tries to greedily minimize fill at each step by eliminating the node with least degree
- Often outperforms RCM, but still just a heuristic

Next time

Stability of factorization:

- What if diagonal entries are (near-) zero?
- Pivoting, and when it is not required.

Application: image denoising

- Given an image with unwanted "noise"/error, how can we try to reduce/remove it?

[Module 08 – Stability of factorization and Image Denoising – 05.25]

So far

- ***direct methods*** for solving linear systems rely on matrix factorization
- matrices often have useful properties or structure (e.g. sparse, banded, symmetric, positive definite)
- we can design faster factorization algorithms by exploiting these properties
- matrix reordering can reduce cost of factorization for sparse matrices by (heuristically) reducing *fill*
- as an application, we say a finite difference method for Poisson problems arising in heat conduction

Today

- discuss some stability issues for factorization
- explore a second application: removing/reducing noise in images

Next time: start on ***iterative methods*** for linear systems

Slides Correction

Consider a (symmetric) matrix reordering from A to $B = PAP^T$
How do row and column swaps affect the linear system? (i.e. x and b)
{Identity $I = P^TP$}
The system becomes $(PAP^T)(Px) = (Pb)$
Letting $y = Px$ and $c = Pb$ we have the new system: $By = c$
(an earlier slide erroneously stated: $PAP^T = Pb$)

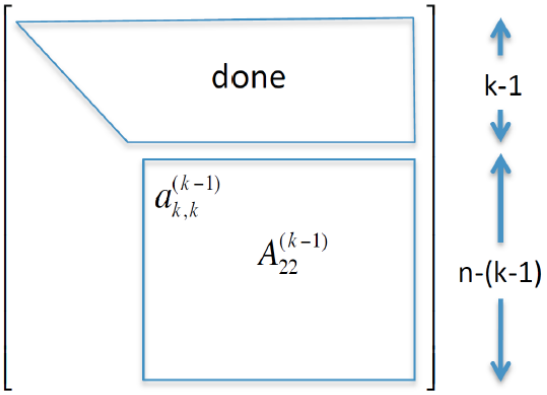
Also: fixed an indexing typo in the Band LU pseudocode (module 4)

Stability of Factorization

The factorizations considered so far assume that the diagonal entry $a_{kk}^{(k-1)}$ (i.e. the "pivot") at each step is non-zero.

Numerical issues arise when pivot is zero ($a_{kk}^{(k-1)} = 0$) or very near-zero ($a_{kk}^{(k-1)} \approx 0$) (i.e. division by zero, large rounding errors)

Usually addressed through *pivoting*.



Pivoting

Permute to get a large magnitude element into position $a_{kk}^{(k-1)}$.
Complete pivoting: use large element in $a_{kk}^{(k-1)}$.
Partial/row pivoting: use largest element in column k.

Example where partial pivoting is inadequate

$$\begin{bmatrix} 1 & & & & 1 \\ -1 & 1 & & & 1 \\ -1 & -1 & 1 & & 1 \\ -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ -1 & 1 & & & \\ -1 & -1 & 1 & & \\ -1 & -1 & -1 & 1 & \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & & & & 1 \\ & 1 & & & 2 \\ & & 1 & & 4 \\ & & & 1 & 8 \\ & & & & 16 \end{bmatrix}$$

Each step induces magnification so the matrix entries grow exponentially.
Fortunately:

Partial Pivoting

Partial/row pivoting is essentially always sufficient in practice

Partial pivoting yields a modified factorization PA = LU, where P is the permutation matrix due to swaps.

Partial Pivoting – Review Example

Consider the following small system:

$$\begin{bmatrix}1 & -1 & 2\\ 1 & -1 & 1\\ 2 & 3 & -1\end{bmatrix} x = \begin{bmatrix}2\\1\\4\end{bmatrix}$$

Factor to get PA = LU, compute b' = Pb, then solve LUx = b' with triangular solves. (s9)

Swap R_1 and R_3

$$P_1 = \begin{bmatrix}0 & 0 & 1\\ 0 & 1 & 0\\ 1 & 0 & 0\end{bmatrix}$$

$$A \rightarrow \begin{bmatrix}2 & 3 & -1\\ 1 & -1 & 1\\ 1 & -1 & 2\end{bmatrix} R_3,\ R_2 - 0.5R_1 \begin{bmatrix}2 & 3 & -1\\ 0 & -2.5 & 1.5\\ 0 & -2.5 & 2.5\end{bmatrix}$$, no swap since |-2.5| = |-2.5|, $P_2 = I$

$$R_3 - R_2 \begin{bmatrix}2 & 3 & -1\\ 0 & -2.5 & 1.5\\ 0 & 0 & 1\end{bmatrix} = U,\ L = \begin{bmatrix}1 & & \\ 0.5 & 1 & \\ 0.5 & 1 & 1\end{bmatrix},\ P = P_2 * P_1 = \begin{bmatrix}0 & 0 & 1\\ 0 & 1 & 0\\ 1 & 0 & 0\end{bmatrix}$$ PA = LU

$$b' = Pb = \begin{bmatrix}0 & 0 & 1\\ 0 & 1 & 0\\ 1 & 0 & 0\end{bmatrix}\begin{bmatrix}2\\1\\4\end{bmatrix} = \begin{bmatrix}4\\1\\2\end{bmatrix}$$

can solve L(Ux) = b'

Pivoting is unnecessary for SPD matrices

Pivoting improves stability, but tends to wreck sparsity. But for certain matrices, pivoting is never necessary.

Theorem: suppose A is symmetric positive definite. Then, $a_{kk}^{(k-1)} > 0$ for all *k*.

Proof: 08_F

Pivoting is also unnecessary for:

1. row diagonally-dominant matrices

$$|a_{k,k}| > \sum_{j \neq k} |a_{k,j}| \text{ for k} = 1 \ldots \text{n}$$

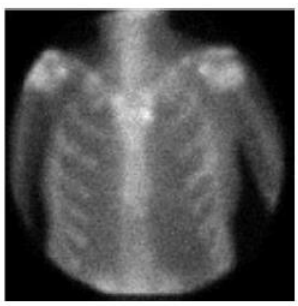
2. column diagonally dominant matrices

$$|a_{k,k}| > \sum_{j \neq k} |a_{j,k}| \text{ for k} = 1 \ldots \text{n}$$

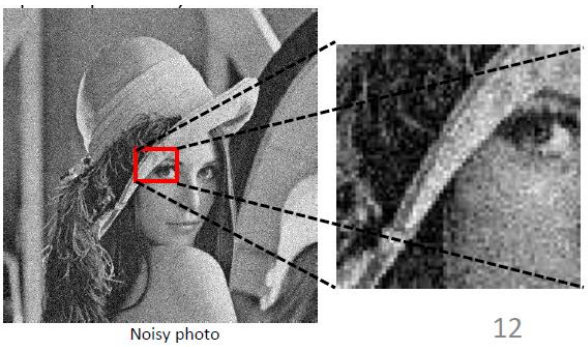
i.e. the diagonal entry is bigger in magnitude than the rest of the entries in its row/column (respectively) combined.

Noise in captured images

Images often contain random "noise" (small errors), which may result from (e.g.) the sensors, the capture process, or the conditions under which it was captured.



Noisy medical image



Noisy photo

Noise in generated images

Synthetic images generated by various computer graphics techniques (e.g. ray tracing) often also possess noise.

Image Denoising

Often there is enough "signal" amidst the noise that we can try to recover a version with the noise reduced/removed.

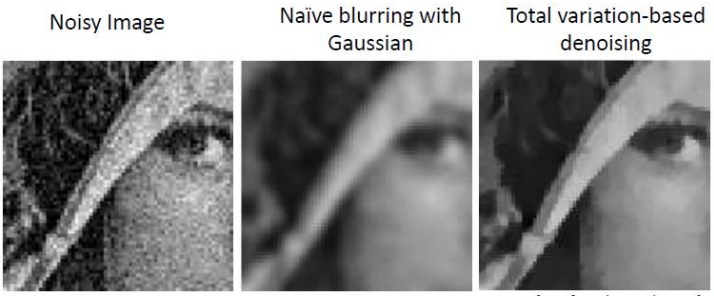


Image denoising is an *inverse* problem: given some observations, reconstruct the source/factors that generated them.

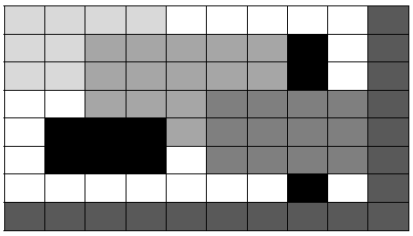
Given:

1. Observed image is $u^0 = u^* + n$, where *n* is noise, u^* is the true underlying image.
2. Estimate of variance of the noise: $||n||^2 = \sigma^2$ for known parameter σ (e.g. from the camera.)

Find an approximation of u^*.

We treat (grayscale) images as 2D scalar functions:

$u_{i,j}$ = pixel intensity value at row *i*, column *j*



Denoising: PDE Approach

Treat as an optimization problem:

Minimize "*fluctuation of pixel value*"

Subject to "*constraint on noise level*"

1. The objective function seeks to eliminate/reduce noise in the solution
2. The constraint ensures that the deviation of our solution from the observed data matches the expected noise level

$$||u - u^0||^2 \approx \sigma^2 = ||u^* - u^0||^2$$

Numerical solution — Observed image — True solution

Denoising: An Ill-Posed Problem

Problem is ill-posed: there are many possible images that would satisfy the constraint on the noise level.

Need a criterion to pick out a specific solution: "**regularization**".

Regularization Models

Noise-level constraint optimization:

$\min_u R(u)$ ← 1. "fluctuation of pixel value"

subject to $||u - u^0||^2 = \sigma^2$ ← 2. "constraint on noise level"

Can be shown equivalent to:

$\min_u f(u) = \alpha R(u) + ||u - u^0||^2$

α parameter controls trade-off between **regularity** and **fit**.

- α ≈ 0, implies $u \approx u^0$ (fit to the data)
- α → ∞, implies $u \approx$ *constant* (regularity of solution)

what is $R(u)$? we want to minimize fluctuations in u.

Attempt #1: *Tikhonov* Regularization

$$R(u) = \int |u|^2 dx = ||u||^2$$

So our optimization becomes:

$$\min_u \alpha||u||^2 + ||u - u^0||^2$$

The Euler-Lagrange equations give us:

$\alpha u + (u - u^0) = 0$

$(\alpha + 1)u = u^0$

$$u = \frac{u^0}{\alpha+1}$$

Tikhonov Regularization

$$u = \frac{u^0}{\alpha + 1}$$

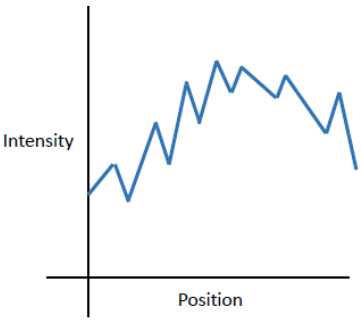
What is the solution for various α?

- α ≈ 0, implies $u \approx u^0$
- α ≈ 0, implies $u \approx 0$

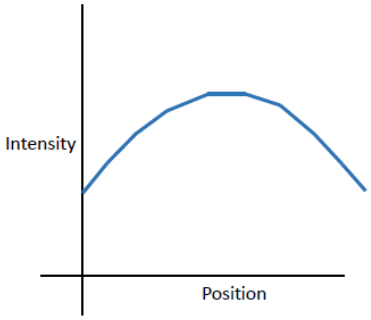
α trades off matching the input data, and being close to 0.

Not quite what we want.

Noise "image" in 1D



"Noisy" function



"Smoother" function

Attempt #2: Laplacian Regularization

Try to penalize slopes, ∇u, e.g. not pixel values, u!

$$R(u) = \int |\nabla u|^2 dx$$

So our optimization becomes:

$$\min_u \alpha ||\nabla u||^2 + ||u - u^0||^2$$

The Euler-Lagrange equations give us:

$-\alpha\Delta u - (u - u^0) = 0$

$-\alpha\Delta u + u = u^0$

Finite Difference Discretization

$-\alpha\Delta u + u = u^0$

We can apply finite differences to approximate this (linear) equation at each pixel:

$$\frac{\alpha}{h^2}\left(4u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1}\right) + u_{i,j} = u^0_{i,j}$$

We can write this in matrix form like so:

$\alpha(A + I)u = u^0$

if the solution is too new noisy, we can try iterating:

for k=0,1,…K

Solve $(\alpha A + I)u^{k+1} = u^k$

end

Drawback: tends to smear out edges!

Attempt #3: total variation regularization

Idea: still minimize the slopes, but with a different norm that doesn't punish them too much.

$$R(u) = \int |\nabla u| dx$$

So our optimization becomes:

$$\min_u \alpha \int |\nabla u| dx + ||u - u^0||^2$$

Equations for Total Variation Regularization

Euler-Lagrange equation gives:

$$-\alpha\nabla \cdot \left(\frac{1}{|\nabla u|}\right)\nabla u + (u - u^0) = 0$$

$$-\alpha\nabla \cdot \left(\frac{1}{|\nabla u|}\right)\nabla u + u = u^0$$

Matrix coefficients (degree of smoothing) depend on gradients in the solution.

(Previous approach is the roughly the same, but with 1 in place of $\frac{1}{|\nabla u|}$)

Total variation regularization – behaviour

$$-\alpha\nabla\cdot\left(\frac{1}{|\nabla u|}\right)\nabla u + u = u^0$$

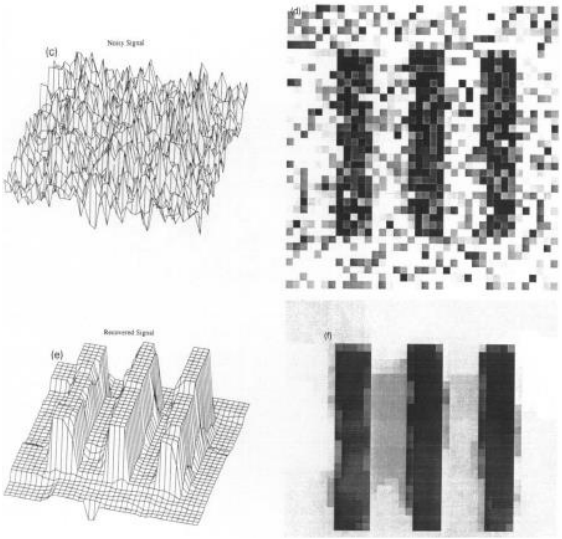
Near edges: $|\nabla u_{i,j}|$ is large → $\frac{1}{|\nabla u_{i,j}|}$ is small!

1st term becomes negligible, leaving $u \approx u^0$ (stay close to data)

In "flat" regions: $|\nabla u_{i,j}|$ is small → $\frac{1}{|\nabla u_{i,j}|}$ is large! We get roughly:

$-C\Delta u_{i,j} + u_{i,j} = u^0_{i,j}$ with C some large value.

Implies more diffusion at (i,j), making these regions flatter/smoother.



Total variation regularization – behaviour

That is:

- flatter regions are smoothed more
- Edge-like regions are smoothed less.

Solution method

We can apply a finite difference discretization again of the form:

$$\alpha A(u) + u = u^0$$
$$(\alpha A(u) + I)u = u^0$$

But this equation is ***nonlinear***, the coefficients depend on the solution.
i.e. matrix entries of *A* depend on *u*.

A simple approach to nonlinear equations is fixed point iteration:
Freeze the coefficients to make the equations linear, solve, update, and repeat.

Fixed point iteration approach

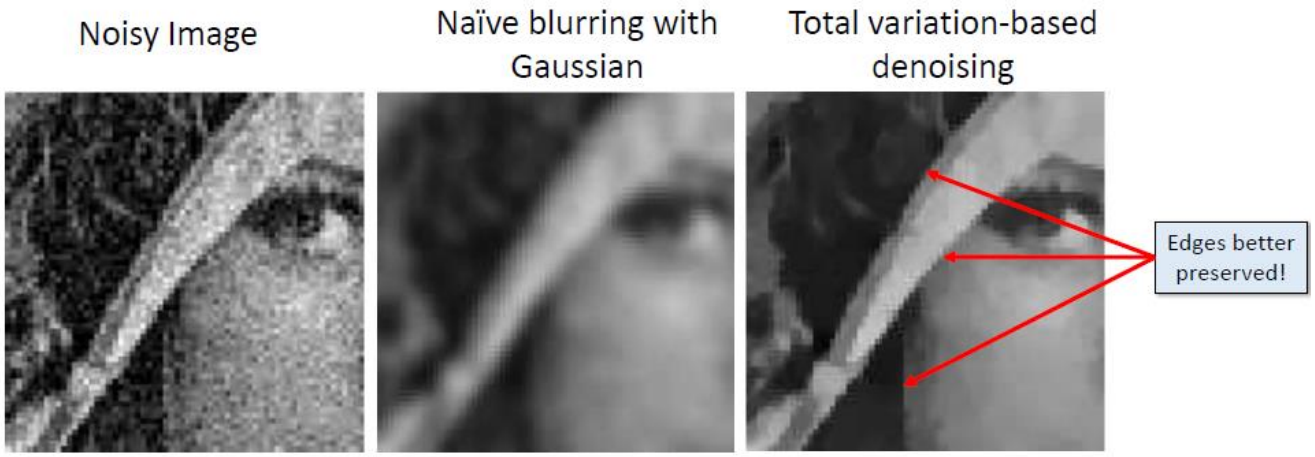
for k=0,1,…K

Solve $(\alpha A(u^k) + I)u^{k+1} = u^k$

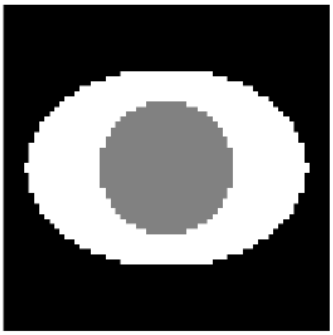
end

We pick an initial guess, compute an approximate solution, recompute A, and repeat the process iteratively.

Comparison



Denoising



Initial synthetic image

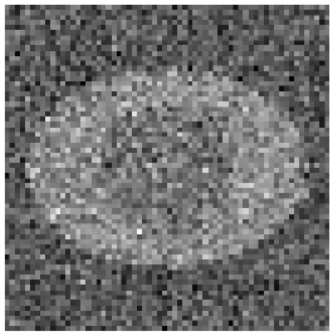
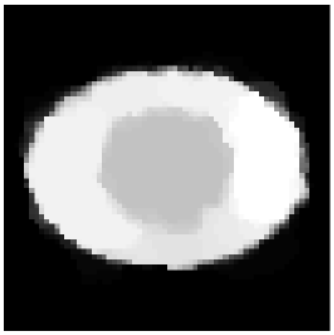


Image with substantial noise added.



Recovered image by TV-based denoising.

Direct Methods

This concludes our look at ***direct methods*** for linear systems.

- These solve the system in a finite sequence of steps

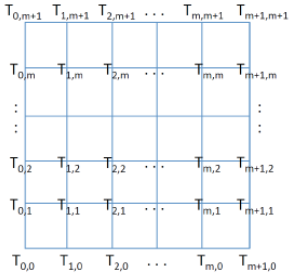
Next time: ***iterative methods***

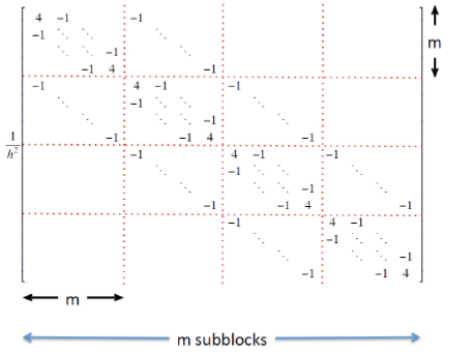
- Iteratively refine a solution, stopping when a tolerance is satisfied

[Module 09 – (Stationary) Iterative Methods – 05.27]

5-Point Laplacian Structure

Laplace matrix structure with natural ordering:
Notice bands AND the "block" structure!
Each grid row corresponds to one matrix block





Reminder: Conditioning

Roughly dictates how small error/changes in input to a (matrix) problem affect the output (even in exact arithmetic)
Condition number: $\kappa = ||A||||A^{-1}|| \geq 1$
κ(A) provides worst-case bounds on relative change in *x* due to relative change in *b* or A:

$$\frac{||\Delta x||}{||x||} \leq \kappa(A) \frac{||\Delta b||}{||b||}$$

And

$$\frac{||\Delta x||}{||x + \Delta x||} \leq \kappa(A) \frac{||\Delta A||}{||A||}$$

Stability

A property of the numerical algorithm, distinct from conditioning.
How changes in input to the *numerical algorithm* affect the output.

Note:
A perfectly stable algorithm cannot prevent issues to due to a poorly conditioned problem.
An unstable algorithm can give useless results even for a well-conditioned problem.

Goal: algorithm to generate *L* and *U* whose "size" remains under control.

- Massive entries will also inflate any floating point errors, producing useless answers.

e.g. re-multiplying LU together can give a new $\tilde{A}$ that may be very far from the original.

LU factorization algorithm with partial pivoting satisfies:

$$\hat{L}\hat{U} = \hat{P}A + \delta A, \ \frac{||\delta A||}{||A||} = O(\rho \epsilon_{machine})$$

Where ρ is a "growth factor"$\approx \frac{||U||}{||A||}$

$$\begin{bmatrix} 1 & & & & 1 \\ -1 & 1 & & & 1 \\ -1 & -1 & 1 & & 1 \\ -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ -1 & 1 & & & \\ -1 & -1 & 1 & & \\ -1 & -1 & -1 & 1 & \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & & & & 1 \\ & 1 & & & 2 \\ & & 1 & & 4 \\ & & & 1 & 8 \\ & & & & 16 \end{bmatrix}$$

$$||A|| \approx 3.145 \qquad ||L|| \approx 2.736 \qquad ||U|| \approx 18.473$$

In this example, the *norm* of U is much larger than that of A.
Growth factor here is $\rho \approx 2^{n-1}$
For large *n*, we run out of precision quickly.

Iterative Methods for Linear Systems

Iterative Methods

Generate a sequence of increasingly accurate *approximations* to the solution by iterating some procedure.
Stop only once some tolerance is satisfied.
Simple "stationary" iterative methods based on "relaxation" include:

- Richardson iteration
- Jacobi
- Gauss-Seidel
- Successive-Over-Relaxation

Possible benefits:

- Can be faster for some (typically large, sparse, 3D) problems
- Tend to be much less memory intensive (no "fill)
- For applications needing only approximate solutions, can "quit early"

Instead of $O(n^3)$ (where n is matrix size), operation count depends on number of non-zeros.

Termination Criterion

Error is $e = x - \hat{x}$, where:

- $\hat{x}$ is the (current, approximate) numerical solution
- X is the true solution

But: we don't know x
A more practical indicator is the residual: $r = b - A\hat{x}$
"By how much does the current solution fail to satisfy Ax = b?"

Residual vs. Error

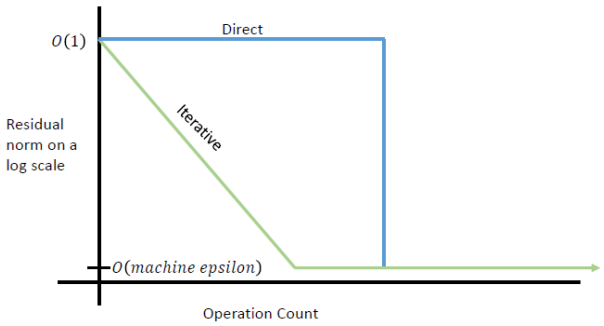
The residual and error satisfy: Ae = r

$$Ax = b \;\rightarrow\; A(\underbrace{x - \hat{x}}_{e}) = \underbrace{b - A\hat{x}}_{r} \;\rightarrow\; Ae = r$$

Matrix condition number relates the size of error e and residual, r.

$$\frac{||e||}{||x||} \leq \kappa(A) \frac{||r||}{||b||}$$

Qualitative Comparison with Direct



(Ideally) iterative approaches makes gradual progress in the solution quality, up to the limit of the floating point representation.
Direct provides no solution until it completes.
Depending on the problem, one or the other may get to a (satisfactory) solution first.

Splitting Approach

Equation to solve is still $Ax = b$

We split A into A = M – N, for some choice of matrices M and N.

Then rearrange our equation into a *fixed point iteration*.

rAx = b → (M – N)x = b → $Mx = Nx + b$

Now define our iterative method by:

$$Mx^{k+1} = Nx^k + b$$

Rearrange this, we see:

$$\begin{aligned} x^{k+1} &= M^{-1}Nx^k + M^{-1}b \\ &= M^{-1}(M - A)x^k + M^{-1}b \\ &= x^k + M^{-1}Ax^k + M^{-1}b \end{aligned}$$

$$x^{k+1} = x^k + M^{-1}(b - Ax^k)$$

Fixed Point Iteration

Generic iteration:
$$x^{k+1} = x^k + M^{-1}(b - Ax^k)$$

How do we choose M?

1. M^{-1} should exist and be cheap/easy to compute.
   (or, My = z should be easy to solve)
2. M ≈ A as closely as possible

Consider if $M = A$ exactly.

Then:

$$x^{k+1} = x^k + A^{-1}(b - Ax^k) = x^k + x - x^k = x$$

One step to the solution.

Computing actual A^{-1} is expensive, so instead try to pick M ≈ A.

Iterative Methods – Preview

Generic iteration:
$$x^{k+1} = x^k + M^{-1}(b - Ax^k)$$

Basic iterative methods amount to different choices of M.

- Richardson iteration: M = $\frac{1}{\Theta} I$
- Jacobi: $M = D$
- Gauss-Seidel: M = D – L
- Successive-Over-Relaxation (SQR): $\frac{1}{\omega} D - L$

Richardson iteration: M = scaled identity

Generic iteration:
$$x^{k+1} = x^k + M^{-1}(b - Ax^k)$$

Let $M = \frac{1}{\Theta} I$, or equivalently $M^{-1} = \Theta I$, for some "appropriate" Θ

Hence $x^{k+1} = x^k + \Theta(b - Ax^k)$

For a particular entry, $x_i^{k+1} = x_i^k + \theta\left(b_i - \sum_{j=1}^{n} a_{i,j} x_j^k\right)$

i.e. new value is a weighted sum of old value (x^k) and residual ($b - Ax^k$)

What is an "appropriate" Θ?

We need $||I - \Theta A|| < 1$; smaller norm implies faster convergence.

Algorithm: Richardson iteration

$x^0 = \text{initial guess}$

Note: need to store 2 separate vectors at a time, x^k and x^{k+1}

for k = 0, 1, 2, …

  for i = 1, 2, 3, …, n

$$x_i^{k+1} = x_i^k + \theta(b_i - \sum_{j=1}^{n} a_{i,j} x_j^k)$$

  end

End

Next: Jacobi, Gauss-Seidel, SOR

These methods will rely on the following sub-matrixes of A:

- D = main diagonal
- -L = below diagonal
- -U = above diagonal

$$A = \begin{bmatrix} \ddots & & -U \\ & D & \\ -L & & \ddots \end{bmatrix}$$

Jacobi: M = D

Intuition is to "exactly" solve each row *independently* for the corresponding entry, using the current x^k estimate.

e.g. if row 7 looks like:

$$2x_5 - 5x_6 + 10x_7 + 3x_9 = 14$$

At step *k*, set

$$x_7^{k+1} = \frac{1}{10}(14 - 2x_5^k + 5x_6^k - 3x_9^k)$$

Jacobi

In general, we can think of zeroing the residual entry for each row.

$$||r^k||_2 \approx 0 \quad \rightarrow \quad x - x^k \approx 0$$

$$r_i = b_i - \sum_{j=1}^{n} a_{i,j} x_j^k = b_i - \sum_{j \neq i} a_{i,j} x_j^k - a_{i,i} x_i^{k+1} = 0$$

We've separated the diagonal element on its own to be updated.

Solving for x_i^{k+1} gives our iterative update rule:

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

Jacobi – Matrix Form

Generic iteration:

$$x^{k+1} = x^k + M^{-1}(b - Ax^k)$$

Our update rule is:

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

In matrix form this is:

$$x^{k+1} = x^k + D^{-1}(b - Ax^k)$$

As claimed, Jacobi is the choice M = D = $\begin{bmatrix} a_{1,1} & & & \\ & a_{2,2}, & & \\ & & \dots & \\ & & & a_{n,n} \end{bmatrix}$

D^{-1} is indeed easy to compute, and (very) roughly approximates A.

Algorithm: Jacobi Iteration

$x^0 = \text{initial guess}$

for k = 0, 1, 2, …

  for i = 1, 2, 3, …, n

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

  end

End

"Old" data

note: still need to store 2 vectors, x^k and x^{k+1}

Example

Solve via Jacobi iteration: $\begin{bmatrix} 3 & 2 \\ 1 & 3 \end{bmatrix} x = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$

$x_1^1 = 1/3(3 - 2*0) = 1$, $x_2^1 = 1/3(1 - 1*0) = 1/3$

$x^1 = \begin{bmatrix} 1 \\ 1/3 \end{bmatrix}$

$x_1^2 = 1/3(3-2*1/3) = 7/9$, $x_2^2 = 1/3(1 - 1*1) = 0$

$x^2 = \begin{bmatrix} 7/9 \\ 0 \end{bmatrix}$

Gauss-Seidel: ***M = D – L***

Same general concept as Jacobi; try to satisfy each row, one at a time.

Difference: instead of "old" data, x^k, use "new" x^{k+1} data for entries that have already been solved

Jacobi:

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

Gauss-Seidel:

$$x_i^{k+1} = \frac{1}{a_{i,i}}\left(b_i - \sum_{j<i} a_{i,j} x_j^{k+1} - \sum_{j>i} a_{i,j} x_j^k\right)$$

Updated entries (below diagonal matrix entries)

Previous step entries (above diagonal matrix entries)

Gauss-Seidel – Matrix Form

Our update rule is:

$$x_i^{k+1} = \frac{1}{a_{i,i}}\left(b_i - \sum_{j<i} a_{i,j} x_j^{k+1} - \sum_{j>i} a_{i,j} x_j^k\right)$$

Generic splitting:

$$x^{k+1} = x^k + M^{-1}(b - Ax^k)$$

In matrix form this is:

$M = D - L$ } Lower triangle of A

Let's verify this…

$$x^{k+1} = x^k + (D - L)^{-1}(b - Ax^k)$$

[Notes] confirm that M = D – L gives our Gauss-Seidel iteration.

Example

Solve via Gauss-Seidel iteration: $\begin{bmatrix} 3 & 2 \\ 1 & 3 \end{bmatrix} x = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$

$x_1^1 = (1/3)*(3 - 2*0) = 1$, $x_2^1 = (1/3)*(1 - 1*1) = 0$ $\quad x^1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

Algorithm: Gauss-Seidel Iteration

$x^0 = \text{initial guess}$

for k = 0, 1, 2, …

  for i = 1, 2, 3, …, n

$$x_i^{k+1} = \frac{1}{a_{i,i}}\left(b_i - \sum_{j<i} a_{i,j}x_j^{k+1} - \sum_{j>i} a_{i,j}x_j^k\right)$$

  end

End

"Already "updated" data." — "Old" data

Note: store only **one** vector; update/overwrite x^k as you go!

Variants of Gauss-Seidel

Is there anything special about proceeding from top to bottom? i.e. inner loop runs from *i* = 1 to *n*.

Not really: reverse ordering gives ***backwards Gauss-Seidel***.

$$M = D - U \quad \} \text{ Upper triangle of A}$$

$$x^{k+1} = x^k + (D-U)^{-1}(b - Ax^k)$$

We can go forwards or backwards – why not both?

Symmetric Gauss-Seidel:

$$x^{k+1/2} = x^k + (D-L)^{-1}(b - Ax^k) \quad \text{Forward}$$

$$x^{k+1} = x^{k+1/2} + (D-U)^{-1}(b - Ax^{k+1/2}) \quad \text{Backward}$$

Which eventually simplifies to:

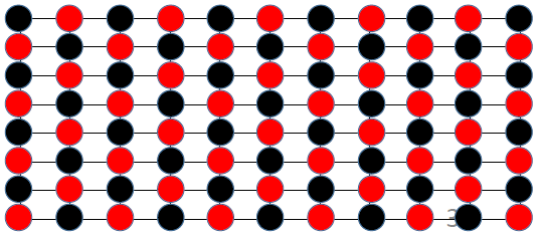
$$x^{k+1} = x^k + (D-U)^{-1}D(D-L)^{-1}(b - Ax^k)$$

What happens if A is symmetric?

Convergence vs. Parallelism

- Jacobi usually converges slower than GS, but easier to parallelize
- Red-black Gauss-Seidel ordering is a compromise: alternate sweeps updating (1) only red nodes and (2) only black nodes

We can update all nodes in parallel since they only use (old) black data, and vice versa.



Successive Over-Relaxation

Add a weight ω to Gauss-Seidel that "over" or "under" relaxes

i.e. weighted average of old data, x_i^k, with GS, x_{GS}^{k+1} = GS(x_i^k)

$$x_i^{k+1} = (1-\omega)x_i^k + \omega x_{GS}^{k+1}$$

$$x_i^{k+1} = (1-\omega)x_i^k + \frac{\omega}{a_{i,i}}\left(b_i - \sum_{j<i} a_{i,j}x_j^{k+1} - \sum_{j>i} a_{i,j}x_j^k\right)$$

Generic iteration:

$$x^{k+1} = x^k + M^{-1}(b - Ax^k)$$

And in matrix form:

$$M = \frac{1}{\omega}D - L$$

$$x^{k+1} = x^k - \left(\frac{1}{\omega}D - L\right)^{-1}(b - Ax^k)$$

$$\omega = \begin{cases} 1 & \text{Regular Gauss-Seidel} \\ > 1 & \text{Over-relaxation} \\ < 1 & \text{Under-relaxation} \end{cases}$$

For certain choices of ω(>1), SOR may converge substantially faster than GS.f

[Module 10 – Convergence of Iterative Methods – 06.01]

Stationary Iterative Methods: Recap

We chose a splitting $A = M - N$. this led to an iterative scheme:

Generic iteration:
$$x^{k+1} = x^k + M^{-1}(b - Ax^k)$$

Different choices of M yield different schemes.

- Jacobi: M = D
- Gauss-Seidel: M = D – L
- SOR: M = $\frac{1}{\omega}D - L$
- Etc.

Convergence

Basic questions:

1. Under what conditions does the iteration converge? (i.e. home on the right solution, in the limit)
2. If it does converge, how fast does it do so?

Need a few definitions first.

Eigenvalues, Eigenvectors, and Spectral Radius

Definition:

v is an ***eigenvector***, and λ its corresponding ***eigenvalue*** if:

$Av = \lambda v$ and $v \neq 0$

Definition:

The ***spectral radius*** of A, denoted ρ(A) = {max |λ|: λ eig of A} is the largest absolute value of the eigenvalues of A.

Iteration Matrix

Our usual iteration can also be written:

$$x^{k+1} = x^k + M^{-1}(b - Ax^k) = (I - M^{-1}A)x^k + M^{-1}b$$

$I - M^{-1}A$ is called the ***iteration matrix***.

Convergence: a sufficient condition

Theorem: let solution x* satisfy $x^* = (I - M^{-1}A)x^* + M^{-1}b$

The stationary iterative method convergences, i.e. for any initial guess x^0,

$$\lim_{k\to\infty} x^k = x^*$$

If $||I - M^{-1}A|| < 1$ for some (induced) matrix norm.

(proof – sketch)

Using normal iteration equation, x^{k+1} and equation for x*, subtract x* from x^{k+1}.

Take the norm of both equations; LHS <= RHS.

Then LHS = error at *k+1*, RHS = iteration matrix * error at *k*. then if the norm of the iteration matrix < 1, then error decreases at each step.

Convergence – Necessary and Sufficient

Theorem: the iterative method $x^{k+1} = x^k - M^{-1}(b - Ax^k)$ converges for any x^0 and b ***if and only if*** $\rho(I - M^{-1}A) < 1$

i.e. converges if the spectral radius of the iteration matrix < 1

- won't prove this

Convergence Speed

The error satisfies $||x^{k+1} - x|| \leq \rho(I - M^{-1}A)||x^k - x||$

So we will call $\rho(I - M^{-1}A)$ the ***convergence rate***.
Smaller ρ (closer to zero) implies faster convergence.

[more on convergence: Saad text, 1.8.4 and 4.2.1]

[Steepest Descent]

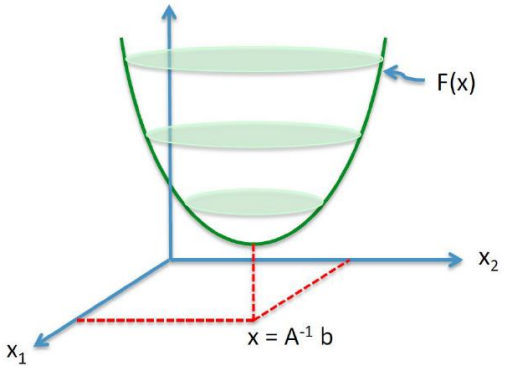
Minimization interpretation

Assume A is symmetric positive definite (SPD). Consider the quadratic form:

$$F(x) = \frac{1}{2}x^T Ax - b^T x, \qquad x \in \mathbb{R}^n$$

The solution of Ax = b is equivalent to the solution of the minimization problem:

$$\min_x F(x)$$



e.g. n = 2 case
A paraboloid with the minimum at $x = A^{-1}b$
(recall since A is SPD, the function is convex)

Minimization Interpretation

The solution of the linear system and minimization form are the same

Proof:
Solution of the minimization satisfies F'(x) = 0, i.e. $\frac{\partial F}{\partial x_k} = 0$ ∀k
Since F'(x) = Ax – b, then Ax = b is a stationary point.
But, F(x) is (strictly) convex, so any local min is the global min.

Solution by Steepest Descent

If we can find the minimum, we have also solved the linear system.
We will try to "walk downhill."

Idea: given a ***search direction*** vector $p \neq 0$, find the minimum point along it.

Let $x^{k+1} = x^k + \alpha p$
Find α that gives the minimum value of $F(x^{k+1}) = F(x^k + \alpha p)$
Think of alpha as how far we are going on p before our next vector
(Proof – Optimal α gives search direction p - sketch)
Reduce $f(\alpha) = F(x^k + \alpha p)$, differentiate with respect to alpha, set to zero, solve for α.

Solution for α

Given update $x^{k+1} = x^k + \alpha p$, and search vector $p \neq 0$, optimal α is:

$$\alpha = \frac{p^T(b - Ax^k)}{p^T Ap} = \frac{p^T r^k}{p^T Ap}$$

Note: since A is SPD, $p^T Ap > 0$
So we can find the minimum along a given search vector.

Choosing Search Directions

What is the optimal search direction?
A vector pointing from x^k towards solution x: $p = x - x^k$
Would find the solution in one step. But we don't know x.

Picking Search Directions

Choose a *locally optimal* direction.
What direction reduces *F* as rapidly as possible at the current point?

We saw that $f'(\alpha) = p^T(Ax^k + b) + \alpha p^T Ap$ (from "Solution for α", taking derivative w.r.t. α)
This gives the rate of change along the search vector.

f'(0) gives rate of change along *p* at the *current* position (i.e. x^k)

Picking Search Directions

f'(0) gives rate of change along p at the current position (i.e. x^k)

Idea: pick *p* to make f'(0) as negative as possible.
This gives the direction of fastest decrease.

(Proof – page 3)(s17)
Given $f'(\alpha) = p^T(Ax^k + b) + \alpha p^T Ap$
Then $f'(0) = p^T(Ax^k + b) = p^T F'(x^k)$ gives rate of change along p at x^k, since $F'(x^k) = Ax^k - b$
Fastest decrease → most negative f'(0)
f'(0) is maximized when $p = \frac{F\prime(x^k)}{||F'(x^k)||}$ (we assume $||p|| = 1$)
f'(0) is minimized when $p = \frac{-F\prime(x^k)}{||F'(x^k)||}$ = steepest descent direction
Therefore steepest descent uses $p = r^k$

$x^{k+1} = x^k + \alpha r^k$

…
Instead of recomputing residual, we can derive a simple update rule.

- $X^{k+1} = x^k + \alpha r^k$

Steepest Descent Algorithm

Given x^0, compute $r^0 = b - Ax^0$.
for k = 0, 1, 2, …

$$\alpha_k = \frac{(r^k)^T r^k}{(r^k)^T Ar^k}$$
$$x^{k+1} = x^k + \alpha r^k$$
$$r^{k+1} = r^k - \alpha A r^k$$

end

Observations:

1. One matrix-vector product required per iteration: Ar^k.
2. Can view as a ***nonlinear*** iterative method:
   $x^{k+1} = x^k + \alpha_k(b - Ax^k)$
   with $M = M^k = \frac{1}{\alpha_k} I$.
   (i.e. iteration matrix changes.)

Example

Apply one step of steepest descent on the problem.

$$\begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix} x = \begin{bmatrix} 2 \\ -8 \end{bmatrix}$$

Starting from $x^0 = [-2, -2]^T$

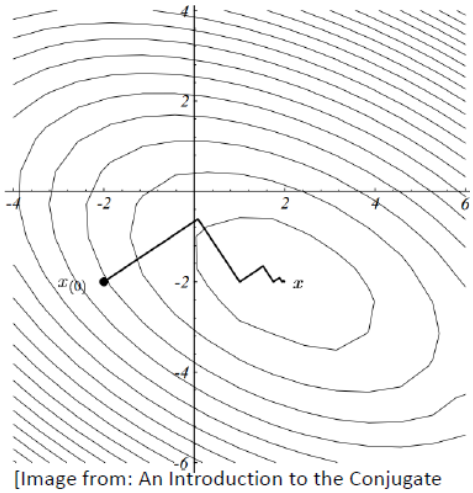
(Eventual) solution is $[2, -2]^T$ (10_H – p5)

Example: Poor Behaviour of Steepest Descent

Since we assumed A was SPD, steepest descent will converge.

But, it can be somewhat short-sighted.

May yield slow (zig-zag-like) convergence, as seen here.



[Image from: An Introduction to the Conjugate Gradient Method Without the Agonizing Pain]

Convergence Behaviour of Steepest Descent

For a SPD matrix A, the error vectors $e^k = x^k - x^*$ for steepest descent satisfy:

$$||e^{k+1}||_A \leq \left(\frac{\lambda_{max} - \lambda_{min}}{\lambda_{max} + \lambda_{min}}\right) ||e^k||_A$$

Where $||\bullet||_A$ indicates the "A-norm" or energy norm: $||x||_A = \sqrt{x^T A x}$

Proof: [Saad, s5.3.1], [Shewchuk 1994]

Next time

The **conjugate gradient method** chooses a different sequence of steps that will generally perform better.

[Module 11 – Conjugate Gradient – 06.03]

Method of Conjugate Directions

Choose a new search direction so that it is A-orthogonal to all previous search directions. This avoids searching in the same direction more than once.

Definition: suppose A is SPD. The A-inner product is defined as: $(p,q)_A = p^T Aq$

A-norm: $||p||_a = \sqrt{(p,p)_A}$

Two vectors are A-orthogonal (or conjugate) if $(p,q)_A = 0$.

Gram-Schmidt Process

Construct a set of A-orthogonal vectors.

Suppose the previous search directions $p^0, p^1, \ldots, p^{k-1}$ are A-orthogonal. Given the current r^k, construct p^k.

The idea is to subtract off the components of r^k that are *not* A-orthogonal to earlier directions, leaving behind a vector that *is*.

Let $p^k = r^k + \sum_{i=0}^{k-1} \beta_i p^i$

$$(p^k, p^j)_A = 0 \rightarrow (r^k, p^j)_A + \left(\sum_{i=0}^{k-1} \beta_i p^i, p^j \right)_A = 0$$

$$(r^k, p^j)_A + \beta_j (p^j, p^j)_A = 0$$

Therefore:

$$\beta_j = -\frac{(r^k, p^j)_A}{(p^j, p^j)_A}$$

(pk, pj) -> result of taking a projection of it and bunch of other vectors; thus orthogonality achieved

- ➔ k-1 only vector saved; others are not needed
- ➔ when you expand the sum, all other p_i disappears because we have made them orthogonal with pj

Conjugate Gradient Method

Intelligently construct a set of A-orthogonal search vectors $\{p^k\}$ from the residuals, $\{r^k\}$, and use them in the same manner (conjugate directions).

$$\text{i.e., } p^k = r^k + \sum_{i=0}^{k-1} \beta_i p^i = r^k - \sum_{i=0}^{k-1} \frac{(r^k, p^i)_A}{(p^i, p^i)_A} p^i$$

Conjugate Gradient Algorithm (version 1)

$x^0 = \text{initial guess}$

$r^0 = b - Ax^0$

for k = 0,1,2,...n-1

Compute p^k as on prev. slide.

$x^{k+1} = x^k + \alpha_k p^k$

$r^{k+1} = r^k - \alpha_k A p^k$

end

Notes:

1. $\alpha_k = \frac{(r^k, p^k)}{(p^k, p^k)_A}$
2. $r^{k+1} = b - Ax^{k+1}$

Conjugate Gradient

Let's work towards a more efficient algorithm, by making some observations about the various vectors involved.

In particular, we will exploit orthogonality or A-orthogonality in a few places.

Some useful facts

• $span\{p^0, p^1, \dots, p^{k-1}\} = span\{r^0, r^1, \dots, r^{k-1}\}$ (UF #1)
$= span\{r^0, Ar^0, \dots, A^{k-1}r^0\}$
$= \kappa_k(A, r^0)$ = k-dimensional ***Krylov subspace***.

• $r^k \perp span\{r^0, r^1, \dots, r^{k-1}\}, i.e. (r^k, r^j) = 0 \text{ for } j = 0,1, \dots k-1$ (UF #2)

Hence by the above, also $r^k \perp$ span$\{p^0, \dots, p^{k-1}\}$
So residual is orthogonal to previous residuals and search vectors.

• $(r^k, p^k) = (r^k, r^k)$ (UF #3)

Proof: $(r^k, p^k) = (r^k, r^k + \sum_{i=0}^{k-1} \beta_i p^i) = (r^k, r^k)$

i.e. since r^k is orthogonal to previous search directions p^i, the latter terms dropped out.

So we can compute step length as:

$$\alpha_k = \frac{(r^k, p^k)}{(p^k, p^k)_A} = \frac{(r^k, r^k)}{(p^k, Ap^k)}$$

• $(r^k, p^i)_A = 0$, for $i = 0,1, \dots k-2$ (UF #4)

Proof: $p^i \in span\{p^0, \dots, p^i\} = span\{r^0, Ar^0, \dots A^i r^0\}$
$\rightarrow Ap^i \in span\{Ar^0, A^2 r^0, \dots, A^{i+1} r^0\}$
$\subset span\{r^0, Ar^0, \dots, A^{i+1} r^0\}$
$= span\{r^0, r^1, r^2, \dots r^{i+1}\}$

But… $r^k \perp span\{r^0, r^1, \dots, r^{i+1}\}\ \forall\ i+1 \le k-1$

$\therefore (r^k, p^i)_A = (r^k, Ap^i) = 0\ \forall\ i \le k-2$

Constructing step directions

• $p^k = r^k + \sum_{i=0}^{k-1} \beta_i p^i = r^k - \sum_{i=0}^{k-1} \frac{(r^k, p^i)_A}{(p^i, p^i)_A} p^i$

Previous identity reduces this to: (β_{k-1})

$$p^k = r^k - \frac{(r^k, p^{k-1})_A}{(p^{k-1}, p^{k-1})_A} p^{k-1}$$

r^k orthogonal to all p^i, i = 0 r… k – 2

So only the current residual and previous step direction are needed to construct p^k.
Next we'll try to simplify the computation of β_{k-1}.

More Useful Facts: Numerator of β

Let's work out the numerator of beta.

• $r^k = r^{k-1} - \alpha_{k-1} Ap^{k-1}$ (from Conjugate Gradient algorithm version 1)

So $(r^k, r^k) = (r^k, r^{k-1}) - \alpha_{k-1}(r^k, Ap^{k-1})$ ((r^k, r^{k-1}) → 0) (taking the inner product of every term w.r.t. r^k)

Rearranging gives…

$$(r^k, Ap^{k-1}) = (r^k, p^{k-1})_A = -\frac{(r^k, r^k)}{\alpha_{k-1}}$$

More Useful Facts: Denominator of β

- $0 = (r^k, p^{k-1}) = (r^{k-1}, p^{k-1}) - \alpha_{k-1}(Ap^{k-1}, p^{k-1})$
  $= (r^{k-1}, r^{k-1}) - \alpha_{k-1}(p^{k-1}, p^{k-1})_A$

(CG Algorithm V1 – inner product on p^{k-1})
(UF #3) left term, $\langle x,y \rangle = x^T y = \langle p,p \rangle_A$

Thus $(p^{k-1}, p^{k-1})_A = \frac{1}{\alpha_{k-1}}(r^{k-1}, r^{k-1})$

So using the last the two identities, we can determine β via

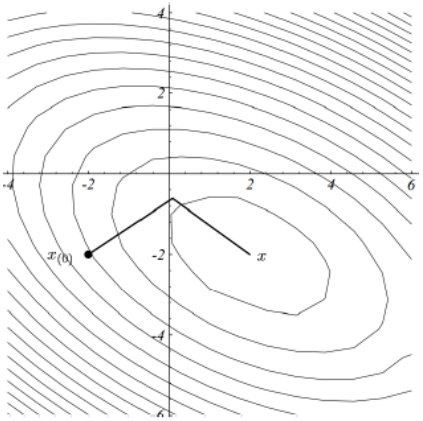
$$\beta_{k-1} = -\frac{(r^k, p^{k-1})_A}{(p^{k-1}, p^{k-1})_A} = -\left(-\frac{1}{\alpha_{k-1}}(r_k, r_k)\right)\left(\frac{\alpha_{k-1}}{(r^{k-1}, r^{k-1})}\right)$$
$$= \frac{(r^k, r^k)}{(r^{k-1}, r^{k-1})}$$

Conjugate Gradient Algorithm (version 2)

$x^0 = \text{initial guess}$
$r^0 = b - Ax^0$
for k = 0,1,…n-1
  $\beta_{k-1} = (r^k, r^k)/(r^{k-1}, r^{k-1})$ $(\beta_{-1} = 0)$
  $p^k = r^k + \beta_{k-1}p^{k-1}$
  $\alpha_k = (r^k, r^k)/(p^k, Ap^k)$
  $x^{k+1} = x^k + \alpha_k p^k$
  $r^{k+1} = r^k - \alpha_k Ap^k$
end

Conjugate Gradient algorithm notes

1. only 1 matrix-vector multiply, and 2 inner-products required
2. at most *n* A-orthogonal vectors in R^n. therefore the algorithm will terminate in at most n steps with an exact solution (assuming exact arithmetic).



Example

Conjugate gradient on the example we considered for steepest descent.
Converges in two steps, since we're in R^2.

Conjugate Gradient – Optimality

At each iteration, the conjugate gradient solution has the minimum energy norm of the error over the subspace it has been allowed to explore.

$$x^k = \underset{x \in \kappa_k}{\text{argmin}} \, ||x - x^*||_A^2$$

Linear Systems

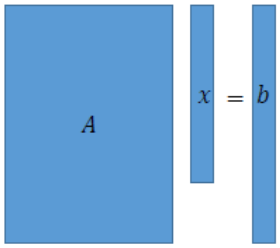
This concludes, for now, our look at direct and iterative methods for standard linear systems, *Ax* = *b*.
Next up: least squares problems, where we have more equations that unknown.

[Module 12 – Least Squares – 06.08]

General Problem

Given a problem with more equations than variables, try to find a solution that satisfies the equations "as well as possible."

i.e. $Ax = b$ where A is taller than it is wide. The solution is "over-determined."



Least Squares

First posted and formulated by Gauss around 1795 (Legendre first published in 1905).

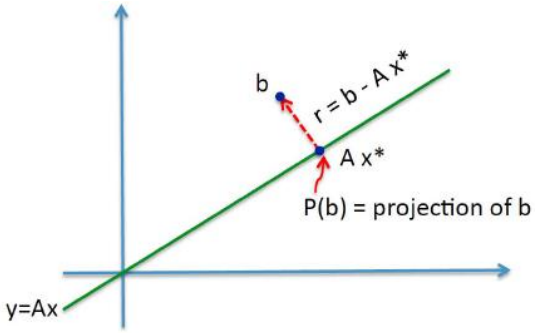
Arises often in applications.

e.g. trying to fit a line or polynomial to a (large) set of data/observations.

Mathematically

Approach: minimize the magnitude of the residual vector $r = b - Ax$

$$\min_{x \in \mathbb{R}^n} ||b - Ax||_2^2, \qquad \text{for } A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m, m \geq n$$



Geometric Interpretation

Find the closest point to b on $y = Ax$

Notice that the residual r is orthogonal to y.

AKA find the "projection" of b onto the range of A.

Theorem

Let $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $m \geq n$, and A full rank. A vector $x \in \mathbb{R}^n$ minimizes

$$||r||^2 = ||b - Ax||^2$$

If and only if $r \perp$ range(A)

Hence $r^T A = 0 \leftrightarrow A^T r = 0 \leftrightarrow A^T(b - Ax) = 0 \leftrightarrow A^T Ax = A^T b$

This gives the "least squares solution."

Pseudo-inverse

Definition: $A^+ = (A^T A)^{-1} A^T$ is called the **pseudoinverse** of A.

The least squares solution is given by:

$x = A^+ b = (A^T A)^{-1} A^T b$

Let's see that perturbations of x away from this solution yields higher residual norm.

Let $x' = x + e$

$||b - Ax'||_2^2 = (b - Ax')^T (b - Ax')$

…

Method 1: Normal Equations

Solve $A^TAx = A^Tb$ directly.

- Compute Cholesky factorization $A^TA = GG^T$, with G lower triangular.
- Compute x by forward/backward solves.

Complexity:

- Flops to form $A^TA \approx mn^2$ flops to factor into $GG^T \approx \frac{1}{3}n^3$
- Total flops $\approx mn^2 + \frac{1}{3}n^3$

Method 2: QR Factorization

Definition: a square matrix Q is orthogonal if $Q^{-1} = Q^T$, i.e. $Q^TQ = QQ^T = I$

Theorem: $||Qx||_2 = ||x||_2$

Proof: $||Qx||^2 = (Qx)^T(Qx) = x^TQ^TQx = x^Tx = ||x||^2$

Note: Multiplication by Q = $\begin{cases} \text{rotation if } \det(Q) = 1 \\ \text{reflection if } \det(Q) = -1 \end{cases}$

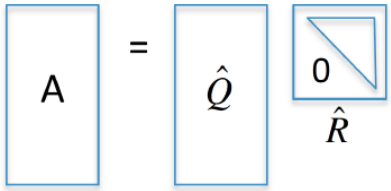
Orthonormal vectors

A set of vectors are ***orthonormal*** if they have norm = 1, and are mutually orthogonal.

e.g. the columns of an orthogonal matrix are orthogonal matrix are orthonormal.

Theorem:

Suppose $A \in R^{m \times n}$ has full rank. Then there exists a unique matrix $\hat{Q} \in R^{m \times n}$ satisfying $\hat{Q}^T\hat{Q} = I$ (i.e. with orthonormal columns) and a unique upper triangular matrix $\hat{R} \in R^{n \times n}$ with positive diagonals ($r_{i,i} > 0$) such that $A = \hat{Q}\hat{R}$



QR and Least Squares

Consider the LS problem:

$$\min ||Ax - b||^2$$

Ax – b = $\hat{Q}\hat{R}x - b$

$= \hat{Q}\hat{R}x - (\hat{Q}\hat{Q}^T - \hat{Q}\hat{Q}^T + I)b$

$= \hat{Q}(\hat{R}x - \hat{Q}^Tb) - (I - \hat{Q}\hat{Q}^T)b$

Note: Since $\hat{Q}$ is non-square here, $\hat{Q}\hat{Q}^T$ is not necessarily equal to the identity, although we do have $\hat{Q}^T\hat{Q} = I$.

The two vectors are orthogonal.

We can show that $(\hat{Q}(\hat{R}x - \hat{Q}^Tb)) \perp -(I - \hat{Q}\hat{Q}^T)b$

orthogonal == perpendicular

$$\left(\hat{Q}(\hat{R}x - \hat{Q}^Tb)\right)^T (I - \hat{Q}\hat{Q}^T)b$$
$$= (\hat{R}x - \hat{Q}^Tb)^T(\hat{Q})^T(I - \hat{Q}\hat{Q}^T)b$$
$$= (\hat{R}x - \hat{Q}^Tb)^T(\hat{Q}^T - \hat{Q}^T\hat{Q}\hat{Q}^T)b$$
$$= (\hat{R}x - \hat{Q}^Tb)^T(\hat{Q}^T - \hat{Q}^T)b = 0$$

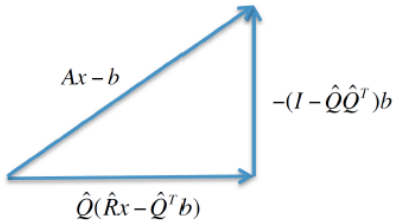
(Because $\hat{Q}^T\hat{Q} = I$)

0

Visually

Pythagoras:

$$||Ax-b||^2 = ||\hat{Q}(\hat{R}x-\hat{Q}^Tb)||^2 + ||(I-\hat{Q}\hat{Q}^T)b||^2$$
$$= ||(\hat{R}x-\hat{Q}^Tb)||^2 + ||(I-\hat{Q}\hat{Q}^T)b||^2$$



The RHS is minimized when the first term is 0 (observe that we can only modify x).

So the solution is:

$$\hat{R}x = \hat{Q}^Tb \rightarrow x = \hat{R}^{-1}\hat{Q}^Tb$$

Relation to pseudoinverse and normal equations

Pseudoinverse in terms of QR factors:

$$A^+ = \hat{R}^{-1}\hat{Q}^T$$

Consider the normal equations:

$$A^TAx = A^Tb \leftrightarrow (\hat{R}^T\hat{Q}^T)(\hat{Q}\hat{R})x = (\hat{R}^T\hat{Q}^T)b$$
$$\hat{R}^T\hat{R}x = (\hat{R}^T\hat{Q}^T)b$$
$$\hat{R}x = \hat{Q}^Tb$$
$$x = \hat{R}^{-1}\hat{Q}^Tb$$

Two sizes of QR factorization

We've seen the reduced (or "economy-size") version, A = $\hat{Q}\hat{R}$, where $\hat{Q} \in R^{m\,x\,n}$ with orthonormal columns, and $\hat{R} \in R^{n\,x\,n}$ upper triangular.

The full version adds extra orthonormal columns to make Q square (and so genuinely orthogonal), and extra zero rows in R to match dimensions.

Full QR Factorization

Append additional *m* – *n* orthogonal columns to Q, i.e.

$$[q_{n+1} \quad q_{n+2} \quad \cdots \quad q_m] \equiv \hat{Q}_{m-n}$$

Then we have:

$$\left[\begin{array}{c} A \end{array}\right]_{m\times n} = \left[\begin{array}{c|c} \hat{Q} & \hat{Q}_{m-n} \end{array}\right]_{m\times m} \left[\begin{array}{c} \hat{R} \\ 0 \end{array}\right]_{m\times n} \Big\} \; m-n \text{ zero rows}$$

Often useful for theoretical purposes.

Next time

- We will look at different ways to construct QR factorizations

[Module 13 – QR Factorization – 06.10]

Least Squares

Last time, we saw least squares problems.

We outlined two solution methods.

1. Form the ***normal equations***, $A^TAx = A^Tb$, and solve directly (via e.g. Cholesky)
   Must forward solve then backward solve; G, G^T need not be like LU where there is identity, just being a lower and upper triangular matrix is fine
2. Construct a ***QR factorization*** of the matrix, $A = QR$, and solve $Rx = Q^Tb$ (matrix-vector multiply, then backward solve).

Application Example: polynomial fitting with least squares

Find a polynomial of the form:

$p(t) = a_0 + a_1t + a_2t^2 + \dots a_{n-1}t^{n-1}$

That best fits a set of 2D points given by (t_i, y_i) for $i = 1 \dots m$, with $m > n$

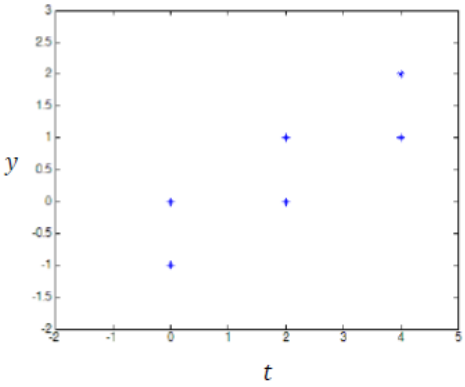
Each data point yields one equation. The coefficients $a_0, a_1, \dots a_{n-1}$ are the unknowns. Matrix problem is:

$$\begin{bmatrix} 1 & t_1 & t_1^2 & \dots & t_1^{n-1} \\ 1 & t_2 & t_2^2 & \dots & t_2^{n-1} \\ \dots & \dots & \dots & \dots & \\ 1 & t_m & t_m^2 & \dots & t_m^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \dots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_m \end{bmatrix}$$

E.g. line fitting

Given the set of (t_i, y_i) points {(0,0), (0,-1), (2,1), (2,0), (4,2), (4,1)}, find the best fit line $y = a_0 + a_1t$ using the normal equations.

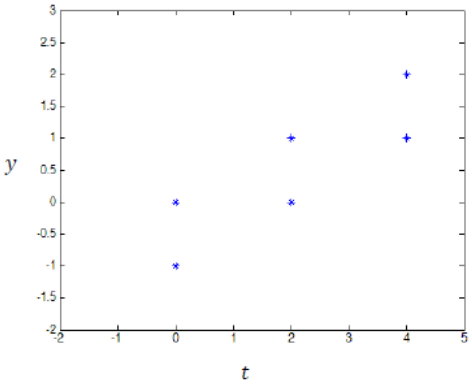
$$A = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 2 \\ 1 & 2 \\ 1 & 4 \\ 1 & 4 \end{bmatrix}, x = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}, b = \begin{bmatrix} 0 \\ -1 \\ 1 \\ 0 \\ 2 \\ 1 \end{bmatrix}$$

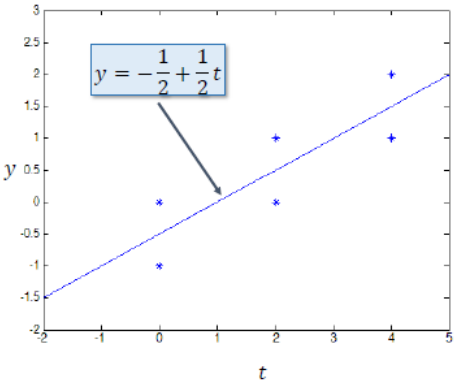


$\Rightarrow A^TA = \begin{bmatrix} 6 & 12 \\ 12 & 40 \end{bmatrix}, A^Tb = \begin{bmatrix} 3 \\ 14 \end{bmatrix}$

$\Rightarrow$ Solution is: $a_0 = -\frac{1}{2}, a_1 = \frac{1}{2}$, so

$$y = -\frac{1}{2} + \frac{1}{2}t.$$

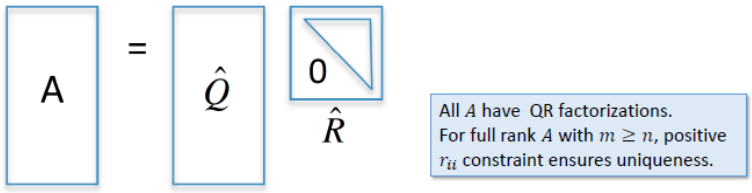




[Finding a QR Factorization]

Reduced QR

Suppose $A \in R^{m\,x\,n}$ has full rank. Then there exists a unique matrix $\hat{Q} \in R^{m\,x\,n}$ satisfying $\hat{Q}^T\hat{Q} = I$ (i.e. with orthonormal columns) and a unique upper triangular matrix $\hat{R} \in R^{n\,x\,n}$ with positive diagonals ($r_{i,I} > 0$) such that A = $\hat{Q}\hat{R}$



All A have QR factorizations.
For full rank A with $m \geq n$, positive r_{ii} constraint ensures uniqueness.

Full QR

Append *m* – *n* orthonormal columns to Q, i.e.:

$$[q_{n+1} \quad q_{n+2} \quad \dots \quad q_m] \equiv \hat{Q}_{m-n}$$

Then we have:

$$\left[\begin{array}{c} A \end{array}\right]_{m\times n} = \left[\begin{array}{c|c} \hat{Q} & \hat{Q}_{m-n} \end{array}\right]_{m\times m} \left[\begin{array}{c} \hat{R} \\ 0 \end{array}\right]_{m\times n} \} \; m-n \text{ zero rows}$$

QR Factorization process (reduced)

Let A = [a_1 a2 … a_n], where a_i are the columns of A.

Want to find a set of *orthonormal* vectors, {q_i} that span the same space, *span*{q_1, q_2, …, q_j} = *span*{a_1, …, a_j} for j = 1 … n.

$$\left[\begin{array}{cccc} a_1 & a_2 & \cdots & a_n \end{array}\right] = \left[\begin{array}{cccc} q_1 & q_2 & \cdots & q_n \end{array}\right] \left[\begin{array}{ccc} r_{11} & \cdots & r_{1n} \\ & \ddots & \vdots \\ & & r_{nn} \end{array}\right]$$

QR Factorization

Written out fully, this is:

$a_1 = r_{11}q_1$

$a_2 = r_{12}q_1 + r_{22}q_2$

$a_3 = r_{13}q_1 + r_{23}q_2 + r_{33}q_3$

…

$a_n = r_{1n}q_1 + r_{2n}q_2 + \cdots + r_{nn}q_n$

Gram-Schmidt Orthogonalization

We already saw a variant of Gram-Schmidt earlier, for (naively) constructing A-orthogonal search directions in conjugate gradient.

Idea is to iteratively build each new vector q_j by *orthogonalizing* a_j w.r.t. all previous *q* vectors, {q_1, q_2, …, q_{j-1}} then normalizing it.

Orthonormalization

For a vector a_j we can orthogonalize it against previous vectors q_i for i = 1 … j – 1 by finding:

$$v_j = a_j - (q_1^T a_j)q_1 - (q_2^T a_j)q_2 \dots - (q_{j-1}^T a_j)q_{j-1}$$

This strips out a_j's components in all the orthogonal directions we constructed so far.

Then we can directly normalize the remainder, v_j.

One way to see this

Let $v_j = a_j + \sum_{i=1}^{j-1} \beta_i q_i$, with v_j the new orthogonal vector.

Since we want v_j to be orthogonal to all previous q_i's, we have:

$$0 = q_k^T v_j = q_k^T a_j + \sum_{i=1}^{j-1} \beta_i (q_k^T q_i)$$

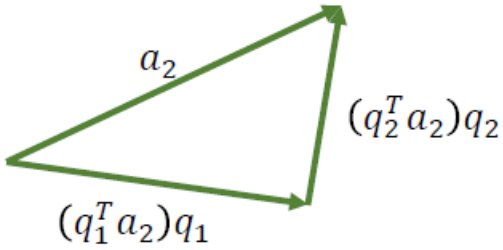
$$= q_k^T a_j + \beta_k (q_k^T q_k) \text{ for } k = 1 \ldots j-1$$

So $\beta_k = -q_k^T a_j$ and therefore:

$$v_j = a_j - \sum_{i=1}^{j-1} (q_i^T a_j) q_i$$

Visual example in 2D

Given a_2 and (previous) unit vector q_1, find q_2.



1. Orthogonalize: $v_2 = a_2 - (q_1^T a_2)q_1$
2. Normalize: $q_2 = \frac{v_2}{||v_2||}$

So $q_2 = \frac{a_2 - (q_1^T a_2)q_1}{||a_2 - (q_1^T a_2)q_1||}$.

What are the r coefficients in our factorization?

QR Factorization

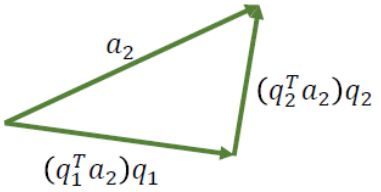
How do we get the R factor's components from this? Observe:

$$a_1 = r_{11} q_1$$
$$a_2 = r_{12} q_1 + r_{22} q_2$$
$$a_3 = r_{13} q_1 + r_{23} q_2 + r_{33} q_3$$
$$\ldots$$
$$a_n = r_{1n} q_1 + r_{2n} q_2 + \cdots + r_{nn} q_n$$

$$\Rightarrow$$

$$q_1 = a_1 / r_{11}$$
$$q_2 = (a_2 - r_{12} q_1)/r_{22}$$
$$q_3 = (a_3 - r_{13} q_1 - r_{23} q_2)/r_{33}$$
$$q_n = \frac{a_n - \sum_{i=1}^{n-1} r_{in} q_i}{r_{nn}}$$

Visual Example in 2D – R Entries



We have $q_2 = \frac{v_j}{||v_j||} = \frac{a_2 - (q_1^T a_2)q_1}{||a_2 - (q_1^T a_2)q_1||}$

Factorization is $q_2 = \frac{a_2 - r_{12} q_1}{r_{22}}$

What are the *r* coefficients in our factorization?

$r_{ij} = (q_i^T a_j) \rightarrow$ length for components of a_j in previous directions

$r_{jj} = ||a_j - \sum_{i=1}^{j-1} r_{ij} q_i||_2 \rightarrow$ length of v_j, required to normalize to q_j.

QR Summary

$q_1 = a_1/r_{11}$

$q_2 = (a_2 - r_{12}q_1)/r_{22}$

$q_3 = (a_3 - r_{13}q_1 - r_{23}q_2)/r_{33}$

...

$q_n = \frac{a_n - \sum_{i=1}^{n-1} r_{in}q_i}{r_{nn}}$

with

$r_{ij} = (q_i^T a_j)$

$r_{jj} = ||a_j - \sum_{i=1}^{j-1} r_{ij}q_i||_2$

Gram-Schmidt algorithm (classic)

for $j = 1, 2, \ldots n$

  $v_j = a_j$ — Get next column

  for $i = 1, 2, \ldots j-1$ — Orthogonalize

    $r_{ij} = q_i^T a_j$

    $v_j = v_j - r_{ij}q_i$

  end

  $r_{jj} = ||v_j||_2$ — Normalize

  $q_j = v_j/r_{jj}$

end

Classical Gram-Schmidt is numerically *unstable;* i.e., sensitive to round-off error. (e.g., it can yield non-orthogonal q's!)

Example – QR Factorization

Find the reduced QR factorization of A = $\begin{bmatrix} 1 & -4 \\ 2 & 3 \\ 2 & 2 \end{bmatrix}$ via Gram-Schmidt.

Solution is: $\begin{bmatrix} 1 & -4 \\ 2 & 3 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & -\frac{14}{15} \\ \frac{2}{3} & \frac{1}{3} \\ \frac{2}{3} & \frac{2}{15} \end{bmatrix} \begin{bmatrix} 3 & 2 \\ 0 & 5 \end{bmatrix}$

if we didn't calculate a r-value, then it is 0

A = QR, $Q^TA = R$ (Q is orthogonal)

Example 2 – QR Factorization

Find the reduced QR factorization of A = $\begin{bmatrix} 1 & 2 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$ via Gram-Schmidt.

Solution is: $\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} \\ 0 & \frac{1}{\sqrt{3}} & \frac{2}{\sqrt{6}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} \end{bmatrix} \begin{bmatrix} \sqrt{2} & \sqrt{2} & \frac{1}{\sqrt{2}} \\ 0 & \sqrt{3} & 0 \\ 0 & 0 & \frac{\sqrt{6}}{2} \end{bmatrix}$

Modified Gram-Schmidt

The sequence of operations can be altered to yield a more stable algorithm called "modified Gram-Schmidt".

In the inner i-loop, rather than use the original a_j to determine r_{ij}, use the (current) v_j instead.

$r_{ij} = q_i^T a_j$ ⇒ $r_{ij} = q_i^T v_j$

Since the components being removed from a_j are orthogonal, the result is identical in exact arithmetic.

Modified Gram-Schmidt

In the inner i loop, v_j is updated at each step with $v_j = v_j - r_{ij}q_i$

i=1: $v_j^{(1)} = a_j - r_{1j}q_1$

i=2: $v_j^{(2)} = v_j^{(1)} - r_{2j}q_2 = a_j - r_{1j}q_1 - r_{2j}q_2$

...

i=k-1: $v_j^{(k-1)} = a_j - \sum_{i=1}^{k-1} r_{ij}q_j$

Then at iteration

i=k: $r_{kj} = q_k^T a_j$

$= q_k^T\left(a_j - \sum_{i=1}^{k-1} r_{ij}q_j\right) = q_k^T v_j^{(k-1)}$

Contributes 0 due to orthogonality of q_k against $\{q_1, q_2, \dots, q_{k-1}\}$.

So $q_k^T a_j = q_k^T v_j^{(k-1)}$, we can replace a_j with current v_j in the inner step.

Modified Gram-Schmidt algorithm

for $j = 1, 2, \dots n$

  $v_j = a_j$

  for $i = 1, 2, \dots j-1$

    $r_{ij} = q_i^T v_j$

    $v_j = v_j - r_{ij}q_i$

  end

  $r_{jj} = ||v_j||_2$

  $q_j = v_j / r_{jj}$

end

Using v_j in place of a_j improves stability. (In exact arithmetic result is identical.)

Complexity of Gram-Schmidt

The inner i-loop involves:

$r_{ij} = q_i^T a_j$ or $q_i^T v_j$ → m (scalar) multiplies, m – 1 additions

$v_j = v_j - r_{ij}q_i$ → m multiplies, m subtractions

∴ flops per inner loop ≈ 4m

Total flops:

$$\sum_{j=1}^{n}\sum_{i=1}^{j-1} 4m = 4m\sum_{j=1}^{n}(j-1) \approx 4m\sum_{j=1}^{n} j = 4m\frac{n(n+1)}{2} \approx 2mn^2$$

Total flops is ≈ $2mn^2$

When m = n, flops(GS) = $2n^3 + O(n^2)$ ≈ 3 x flops(LU)

(So we *could* use QR to solve a standard linear system, but at ~3x cost)

Householder Triangularization

Another (more stable) approach to QR Factorization.

Gram-Schmidt is a "triangular orthogonalization": make the columns orthonormal via operations that amount to right-multiplication by a triangular matrix.

Householder proceeds by "orthogonal triangularization": make a matrix triangular by applying orthogonal matrix operations.

Premise: $Q_nQ_{n-1} \dots Q_2Q_1A = R$, where $Q_k \in R^{m\,x\,m}$ are orthogonal matrices.

Similar to Gaussian Elimination, each Q_k will zero the entries column *j*.

$$\begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} \xrightarrow{Q_1} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{bmatrix} \xrightarrow{Q_2} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & \times \end{bmatrix} \xrightarrow{Q_3} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & 0 \end{bmatrix}$$

$$A \qquad Q_1A \qquad Q_2Q_1A \qquad Q_3Q_2Q_1A$$

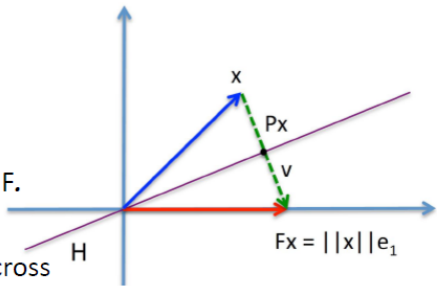
Householder Reflections

We'll be building orthogonal matrices of the following form.

Define $Q_k = \begin{bmatrix} I & 0 \\ 0 & F \end{bmatrix}$ } k-1 rows; } m-(k-1) rows

Basic tool is the ***Householder reflector*** F.

It performs a reflection of a vector x across a (specific) hyperplane H.



Suppose x = $\begin{bmatrix} x \\ x \\ x \\ \dots \\ x \end{bmatrix}$. We want to produce Fx = $\begin{bmatrix} ||x|| \\ 0 \\ \dots \\ 0 \end{bmatrix} = ||x||e_1$

F reflects *x* across the hyperplane *H* orthogonal to *v* = ||x||e_1 – x

(i.e. we've produced a new vector of the same length aligned with the axis e_1 so that all but the first entry are zeros)

The ***orthogonal projection P*** of *x* onto the hyperplane *H* (orthogonal to the vector *v*) is:

$$Px = x - \left(\left(\frac{v}{||v||}\right)^T x\right)\frac{v}{||v||} = x - v\left(\frac{v^T x}{v^T v}\right)$$

To instead reflect across H, go twice as far:

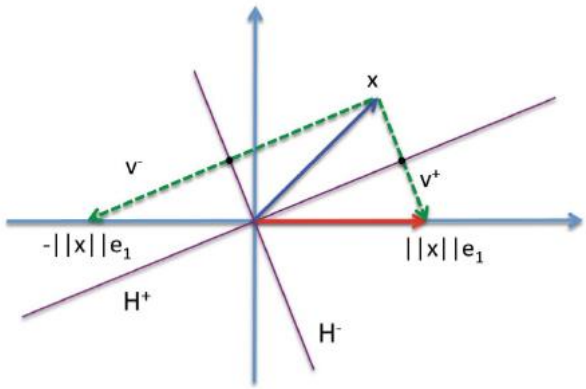
$$Fx = x - 2v\left(\frac{v^T x}{v^T v}\right)$$

An Alternative Reflection

We could also choose the other reflection, with a negative sign.

Fx = $\begin{bmatrix} -||x|| \\ 0 \\ \dots \\ 0 \end{bmatrix} = -||x||e_1$

Either one zeros out the desired entries of the column.



Which to choose?

For stability, we select the one that is farther away:

x > 0, choose left one

x < 0, choose right one

So $v = -\text{sign}(x_1)||x||e_1 - x$

Or $v = \text{sign}(x_1)||x||e_1 + x$

Avoids subtracting nearby quantities (i.e. cancellation error)

Alternative Derivation

<13_J, notes 06.10>

[Module 14 – Householder and Givens-based QR – 06.15]

QR Factorization

Three common algorithms for QR Factorization.

1. Gram-Schmidt orthogonalization.
2. **Householder reflections**
3. Givens rotations

Householder triangularization

Premise: $Q_nQ_{n-1} \dots Q_2Q_1A = R$, where $Q_k \in R^{m\,x\,m}$ are orthogonal matrices.
Similar to Gaussian Elimination, each Q_k will zero the entries column *j*.

$$\begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} \xrightarrow{Q_1} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{bmatrix} \xrightarrow{Q_2} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & \times \end{bmatrix} \xrightarrow{Q_3} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & 0 \end{bmatrix}$$

A $\quad Q_1A \quad Q_2Q_1A \quad Q_3Q_2Q_1A$

Householder Reflections

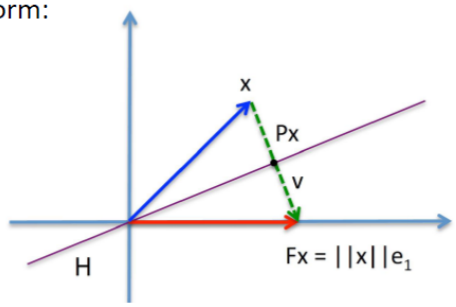
We apply orthogonal matrices of the form:

$$Q_k = \begin{bmatrix} I & 0 \\ 0 & F \end{bmatrix} \begin{matrix} \}\text{k-1 rows} \\ \}\text{m-(k-1) rows} \end{matrix}$$

We use ***Householder reflectors*** F*:*

$$Fx = x - 2v\left(\frac{v^T x}{v^T v}\right)$$

where $v = \text{sign}(x_1)||x||e_1 + x.$



Householder Reflector Example

Given vector x = $[1, 2, 2]^T$, find the Householder reflector F and the product Fx.

Householder QR factorization algorithm

for $k = 1, 2, \dots, n$

- $x = A(k{:}m, k)$ — Get current column.
- $v_k = x + sign(x_1)||x||e_1$ — Form the reflection vector
- $v_k = v_k/||v_k||$ — Normalize it.
- for $j = k, k+1, \dots, n$
  - $A(k{:}m, j) = A(k{:}m, j) - 2v_k(v_k^T A(k{:}m, j))$ — Apply the reflection F to the remaining lower-right block.
- end

end

The algorithm reduces *A* to R.
Notation: A(k:m, j) = j^{th} column of A from row k to row m (as in MATLAB).
Could further *vectorize* inner loop to: A(k:m, k:n) = A(k:m, k:n) – $2v_k(v_k^TA(k{:}m, k{:}n))$

Implicit Products with Q

But: the algorithm did not build Q.
Only the v_k's are built (and stored). Why?
Often don't need Q, but just the products: $Q^T b$ or Qx.
(e.g. for least squares, we solve $Rx = Q^T b$)

$Q^T = Q_n Q_{n-1} \dots Q_2 Q_1$

$Q = Q_1 Q_2 \dots Q_{n-1} Q_n$

We can efficiently find $Q^T b$ or Qx using the v_k's.

Implicit Calculation of **Q** products

Implicit $Q^T b$:

for $k = 1, 2, \dots, n$

$$b(k\!:\!m) = b(k\!:\!m) - 2v_k \left(v_k^T b(k\!:\!m)\right)$$

end

Implicit Qx:

for $k = n, n-1, \dots, 1$

$$x(k\!:\!m) = x(k\!:\!m) - 2v_k (v_k^T x(k\!:\!m))$$

end

Recovering **Q**

How can we use these implicit products with *Q* to recover *Q* itself?

Compute the product *QI*, by applying *Q* to *I*'s columns, e_1, e_2, …

For the reduced QR, $A = \hat{Q}\hat{R} =$ $\begin{bmatrix} & \\ & \end{bmatrix}_{m\times n} \begin{bmatrix} & \\ & \end{bmatrix}_{n\times n}$

$\hat{Q}$ is given by $[Qe_1\ Qe_2\ Qe_3 \dots Qe_n]$ (i.e. just the first *n* columns of *I*)

Example: QR Factorization via Householder

Perform QR factorization using Householder reflections on the matrix $A = \begin{bmatrix} 1 & -4 \\ 2 & 3 \\ 2 & 2 \end{bmatrix}$

How many Q's to calculate? The n in A. Q

Q_1 = initial F, Q_2 = identity with F as submatrix

in our example, v_+ is incorrect; negatives reversed

Complexity of Householder-based QR

Work is dominated by inner loop.

$$A(k\!:\!m, j) = A(k\!:\!m, j) - 2v_k(v_k^T A(k\!:\!m, j))$$

Let's tally the cost:

$v_k^T A(k\!:\!m, j) \approx 2(m-k+1)$ flops (dot product)

multiply and add for each vector element => number

$2v_k(v_k^T A(k\!:\!m, j) \approx (m-k+1)$ flops (scalar multiply)

scalar multiply each element of v_k^T (actually + 1 this)

$A(k\!:\!m, j) - 2v_k \left(v_k^T A(k\!:\!m, j)\right) \approx (m-k+1)$ flops (subtraction)

so $\approx 4(m-k+1)$ flops per inner step.

4(m – k + 1) flops per inner-most step
This is done (n – k + 1) times (j loop)
4(m – k + 1)(n – k + 1) flops per outer iteration.
The outer k-loop runs from 1 to n.

Total flops: $\sum_{k=1}^{n} 4(m-k+1)(n-k+1) \approx 2mn^2 - \frac{2}{3}n^3$

(This *does not include* forming Q)

For *m* = *n* (square), flops(QR) $\approx \frac{4}{3}n^3$ = 2 x flops(LU)

```
for k = 1, 2, ..., n
    for j = k, k + 1, ..., n
        4(m − k + 1) flops
    end
end
```

QR Factorization

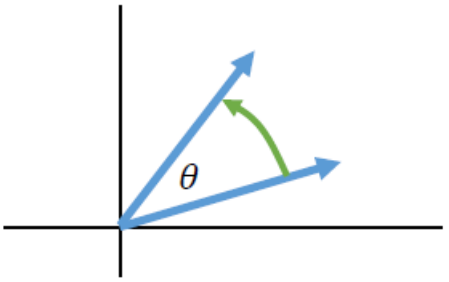
Three common algorithms for QR factorization.

1. Gram-Schmidt orthogonalization
2. Householder reflections
3. **Givens rotations**

Rotation matrices in 2D

Rotations of a vector in 2D can be described via 2x2 matrix:

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$



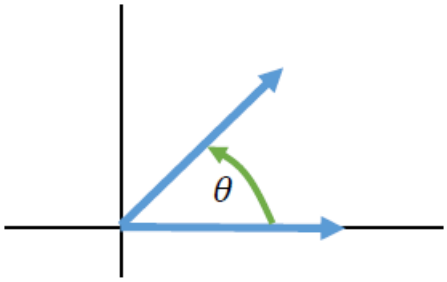
y = *R*x rotates the vector *y* counter-clockwise by Θ

Easy to see that R is orthogonal:
Transpose gives clockwise rotation (i.e. the inverse)
$\cos^2\Theta + \sin^2\Theta = 1$ and $\cos\Theta\sin\Theta - \cos\Theta\sin\Theta = 0$

Rotation Matrix

E.g. $\Theta = \frac{\pi}{4}$ (45 degrees)

$$R = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$



If $x = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, then $y = Rx = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$

How might we use rotations to zero matrix entries?

Givens Rotations

Zero individual elements "selectively", by orthogonal matrix operations that perform rotation in the (i,k)-plane only.
Givens rotations have the form:

$$G(i,k,\theta)^T = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & c & & -s & & \\ & & & 1 & & & \\ & & & & \ddots & & \\ & & & & & 1 & \\ & & s & & c & & \\ & & & & & & \ddots \\ & & & & & & & 1 \end{bmatrix} \begin{matrix} \\ \\ \text{row(i)} \\ \\ \\ \\ \text{row(k)} \\ \\ \end{matrix} \qquad \begin{matrix} c = \cos\theta \\ \\ s = \sin\theta \end{matrix}$$

col(i) col(k)

G(i,k,Θ) is orthogonal.
Consider $y = G(i,k,\Theta)^T x$. then

$$y_j = \begin{cases} cx_i - sx_k, & \text{for } j = i \\ sx_i + cx_k, & \text{for } j = k \\ x_j & \text{for } j \neq i,k \end{cases}$$

$$G(i,k,\theta)^T = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & c & & -s & & \\ & & & 1 & & & \\ & & & & \ddots & & \\ & & & & & 1 & \\ & & s & & c & & \\ & & & & & & \ddots \\ & & & & & & & 1 \end{bmatrix} \begin{matrix} \\ \\ \text{row(i)} \\ \\ \\ \\ \text{row(k)} \\ \\ \end{matrix}$$

col(i) col(k)

To force $y_k = 0$, let's

$$c = \frac{x_i}{\sqrt{x_i^2 + x_k^2}} \text{ and } s = \frac{-x_k}{\sqrt{x_i^2 + x_k^2}}$$

Notes:

1. *c* and *s* can be computed in 5 flops.
2. Value of Θ itself is not needed.
3. Computing product $G(i,k,\Theta)^T A$ affects only row(i) and row(k)

To perform QR factorization, zero entries one at a time, working upwards along columns.

Givens QR Process

e.g.

$$\begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} \xrightarrow{(3,4)} \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ 0 & \times & \times \end{bmatrix} \xrightarrow{(2,3)} \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{bmatrix} \xrightarrow{(1,2)}$$

$$\begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{bmatrix} \xrightarrow{(3,4)} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \end{bmatrix} \xrightarrow{(2,3)} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & \times \end{bmatrix} \xrightarrow{(3,4)} R$$

Givens Example

Given the vector $x = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$ apply a Givens rotation with i = 2 and k = 4, to zero out the bottom element. What is the corresponding matrix $G(2,4,\Theta)^T$ and result $G^T x$?

Givens QR Process

Let G_k^T be the j^{th} Givens rotation.

Then we can assemble Q like so:

$G_k^T G_{k-1}^T \dots G_2^T G_1^T A = R$

$A = QR$ with $Q = G_1 G_2 \dots G_{k-1} G_k$

Flops(Givens QR) $\approx 3mn^2 - n^3 = 1.5 \times$ flops(Householder QR)

Why bother?

More flexible than Householder; useful when only few elements need to be eliminated.

Hessenberg QR via Givens

QR applied to an upper Hessenberg matrix looks like:

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix} \xrightarrow{(1,2)} \begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix} \xrightarrow{(2,3)} \begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ & 0 & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix} \xrightarrow{(3,4)}$$

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ & 0 & \times & \times & \times \\ & & 0 & \times & \times \\ & & & \times & \times \end{bmatrix} \xrightarrow{(4,5)} \begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ & 0 & \times & \times & \times \\ & & 0 & \times & \times \\ & & & 0 & \times \end{bmatrix} \qquad flops \approx 3n^2$$

Next topic

Algorithms for solving eigenvalue problems, i.e. finding λ and v such that $Av = \lambda v$

In practice, we never solve the characteristic polynomial.

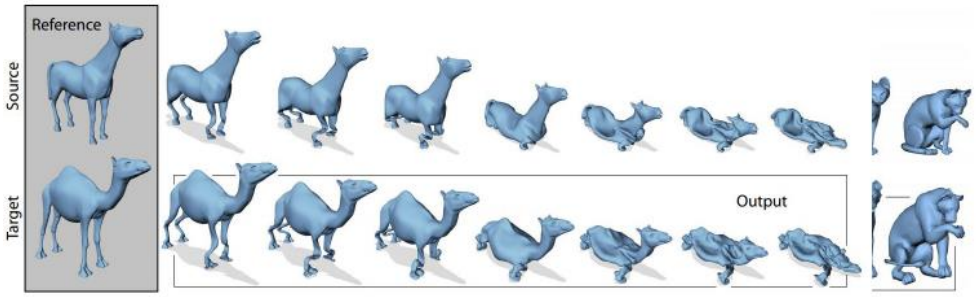
[Module 15 – Eigenproblems – 06.17]

Graphics Application of the Day

Deformation transfer: given 3D geometry and deformed poses, copy the deformations to a new object, *as closely as possible*.

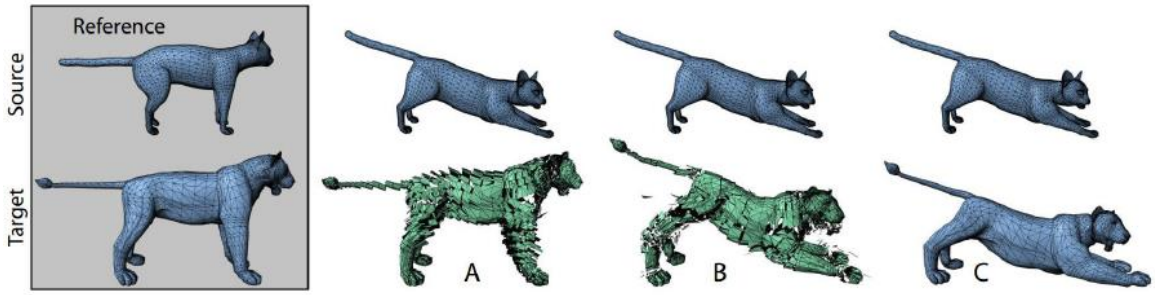
➔ Involves a **least squares** problem

e.g. given a cat and several poses, copy the poses onto a lion



Intuition

Apply the same linear transform to each triangle as in the source. Then, find a least squares fit that "glues" the vertices back together.



The authors used the *normal equations* approach, solving the linear system with a *sparse direct (LU) solver*.

[Eigenproblems]

Eigenvalue problems

Definition: let $A \in R^{nxn}$. A non-zero vector $x \in R^n$ is a (right) eigenvector and λ its corresponding **eigenvalue** if:

$Ax = \lambda x$

If x is an eigenvector, then so is αx, for $\alpha \neq 0$

(i.e. eigenvectors are determined only up to a multiplicative constant)

Definition: the set of A's eigenvalues is its spectrum, denoted Λ(A).

Eigendecomposition – matrix form

An eigendecomposition of A is:

$\mathbf{A = X\Lambda X^{-1}}$

Where

$$X = \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \dots & x_n \\ | & | & & | \end{bmatrix}, \Lambda = \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \dots & \\ & & & \lambda_n \end{bmatrix}$$

And $Ax_i = \lambda_i x_i$ for $i = 1, 2, \dots, n$

i.e. X's columns are eigenvectors. Λ's diagonal entries are eigenvalues.

Eigendecomposition

Equivalently, $AX = X\Lambda$

$$\begin{bmatrix} & A & \end{bmatrix}\begin{bmatrix} | & | & & | \\ x_1 & x_2 & \dots & x_n \\ | & | & & | \end{bmatrix} = \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \dots & x_n \\ | & | & & | \end{bmatrix}\begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \dots & \\ & & & \lambda_n \end{bmatrix}$$

Characteristic Polynomial

Definition: the characteristic polynomial of A, $p_A(z)$ is the degree n (monic) polynomial given by

$p_A(z) = \det(zI - A)$

Theorem: λ is an eigenvalue of A iff $p_A(\lambda) = 0$

Proof:

λ is an eigenvalue

$\Leftrightarrow \lambda x - Ax = 0$ (for some $x \neq 0$)

$\Leftrightarrow \lambda I - A$ is singular

$\Leftrightarrow \det(\lambda I - A) = 0$

The fundamental theorem of algebra tells us that the degree *n* polynomial $p_A(z)$ has *n* (possibly complex) roots.
So A as *n* (possibly complex) eigenvalues given by the roots.
We can write:

$p_A(z) = (z - \lambda_1)(z - \lambda_2)\dots(z - \lambda_n)$

Given an eigenvalue λ, the corresponding eigenvector(s) are given by solving $(\lambda I - A)x = 0$ for *x*
(i.e. the nullspace of $\lambda I - A$)

Conversely, for every *monic polynomial* of degree *n*,

$p(z) = z^n + a_{n-1}z^{n-1} + \dots + a_1 z + a_0$

There always exists a matrix whose eigenvalues are the roots of p(z).

e.g. companion matrix, $C = \begin{bmatrix} 0 & & & & -a_0 \\ 1 & 0 & & & -a_1 \\ & 1 & 0 & & -a_2 \\ & & \dots & \dots & \dots \\ & & & 1 & -a_{n-1} \end{bmatrix}$

Example

Find the eigenvalues and eigenvectors for the matrix

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 2 & -1 & 2 \\ 4 & -4 & 5 \end{bmatrix}$$

MATLAB's eigenvalue function outputs the correct eigenvalues but the outputted eigenvectors are of different form.

Multiplicity of Eigenvalues

The ***algebraic multiplicity*** of λ is the number of times it appears as a root of $p_A(z)$.
The ***geometric multiplicity*** of λ is the dimension of the nullspace of the matrix $\lambda I - A$.
If algebraic multiplicity exceeds geometric, λ is a defective eigenvalues, and the matrix is ***defective***.
Only non-defective matrices have eigenvalue decompositions.

Example: Multiplicity

$$A = \begin{bmatrix} 2 & & \\ & 2 & \\ & & 2 \end{bmatrix}, B = \begin{bmatrix} 2 & 1 & \\ & 2 & 1 \\ & & 2 \end{bmatrix}$$

$p_A(z) = (z - 2)^3$ for both, so $\lambda = 2$. Algebraic multiplicity 3.

$(2I - A) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ Any 3 linearly independent vectors span the nullspace. (Geometric multiplicity of 3)

$(2I - B) = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$ Only $x = [1, 0, 0]^T$ satisfies. **Defective!** (geometric multiplicity of 1)

Finding Eigenvalues

No closed form solutions exist for degree 5 or higher polynomials.

The roots cannot be found (exactly) using a finite number of rational operations! We must settle for approximations.

Does this suggest *direct* or *iterative* methods?

- ⇨ **Iterative**

Naïve Approach

1. Form the characteristic polynomial, $p_A(z)$
2. Use a root-finding method to extract the approximate roots/eigenvalues (e.g. bisection, Newton, etc.)

Root-finding tends to be ill-conditioned:

- ⇨ Small change/error in input gives large change in roots

More effective strategies exist.

Rayleigh Quotient

Let's first try to recover eigenvalues:

We'll assume $A \in R^{nxn}$ and symmetric ($A^T = A$). Such matrices (1) have real eigenvalues, and (2) have a complete set of *orthogonal* eigenvectors.

$\{\lambda_1, \lambda_2, \ldots, \lambda_n\}, \{q_1, q_2, \ldots, q_n\}$ with $||q_i|| = 1$

So

$A = Q\Lambda Q^T$

Definition: the ***Rayleigh quotient*** of a nonzero vector x with respect to A is

$$r(x) \equiv \frac{x^T A x}{x^T x}$$

If *x* is an eigenvector of A, what is r(x), and why?

$r(x) = \lambda$, since $\frac{x^T A x}{x^T x} = \lambda \frac{x^T x}{x^T x} = \lambda$

Otherwise: r(x) gives the scalar α that behaves "most like" an eigenvalue for a given vector *x*.

Rayleigh Quotient – Another Justification

Consider the following *n* x 1 least squares problem for unknown α:

$$\min_{\alpha} \left\| \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix} \alpha - Ax \right\|_2$$

What are the normal equations for this problem?

$(x^T x)\alpha = x^T(ax)$

$\alpha = \frac{x^T Ax}{x^T x} = r(x)$

Rayleigh Quotient

Theorem: let q_j be an eigenvector, and $x \approx q_j$. Then

$r(x) - r(q_j) = O(||x - q_j||^2)$ as $x \to q_j$

That is: as $x \to q_j$, error in the estimate of λ decreases quadratically.

Rayleigh Quotient example

$A = \begin{bmatrix} 3 & 2 & 5 \\ 2 & 7 & 5 \\ 0 & 2 & 8 \end{bmatrix}$ has an approximate eigenvector $v \approx [0.7, -0.7, 0.3]^T$

Rayleigh quotient gives:

$\alpha = \frac{v^T Av}{v^T v} \approx 3.028$

Which is close to the true value of λ = 3

Finding Eigenvectors – Stay Tuned!

But we need (a reasonable guess at) an eigenvector to make the Rayleigh quotient useful.

We'll look at ways to find eigenvectors next time.

First up: ***power iteration*** (e.g. underpins Google's original PageRank algorithm)

[Module 16 – Iterative Methods for Eigenproblems – 06.24]

Eigenproblems

Given a square matrix A, find its eigendecomposition, find its eigendecomposition, $A = X\Lambda X^{-1}$
Columns of X are eigenvectors, diagonal of Λ contains eigenvalues.

Before midterm, we:

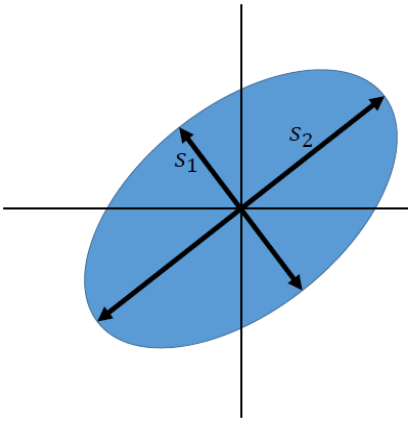
- Noted that an *iterative method* is required; no closed form solutions exist for (high degree) characteristic polynomials.
- Saw the ***Rayleigh quotient***, $r(x) \equiv \frac{v^T A v}{v^T v}$, which approximates the eigenvalue given an (approximate) eigenvector.

Eigenproblems: Geometric intuition

For symmetric matrices, the region given by

$$\varepsilon = \{x \mid x^T A x \leq 1\}$$

is an ellipsoid in R^n centred at the origin.



Its semi-axes are $s_i = \lambda_i^{-1/2} q_i$

Eigenvalues λ_i dictate *lengths* of semi-axes.
(Normalized) eigenvectors q_i dictate *directions* of semi-axes.

Bounding eigenvalues

Gershgorin Circle Theorem:
Let A be any square matrix. The eigenvalues λ of A are located in the union of the *n* disks (on the complex plane) given by:

$$|\lambda - a_{ii}| \leq \sum_{j \neq i} |a_{ij}|$$

Disks are denoted by $D(a_{ii}, R_i)$, where $R_i = \Sigma_{j \neq i} |a_{ij}|$

Proof
Consider (λ,x) s.t. $Ax = \lambda x$, $x \neq 0$
Scale x s.t. $||x||_\infty = 1 = x_i$, for some i
Then $\lambda x_i = (Ax)_i = \sum_{j=1}^{n} a_{ij} x_j = \sum_{j \neq i} a_{ij} x_j$ move $a_{ii}x_i$ to the left

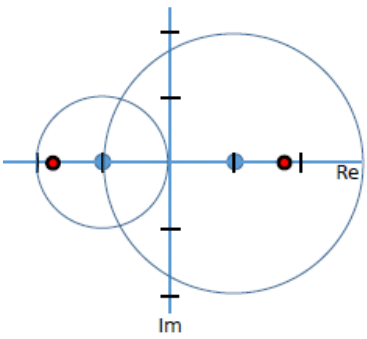
$$|(\lambda - a_{ii}) x_i| = \left|\sum_{j \neq i} a_{ij} x_j\right|$$

$$|\lambda - a_{ii}| \cancel{|x_i|} \leq \left|\sum_{j \neq i} a_{ij} x_j\right|$$

$$\leq \sum_{j \neq i} |a_{ij}|$$

Gershgorin disks: $|\lambda - a_{ii}| \leq \sum_{j \neq i} |a_{ij}|$

Essentially: if off-diagonal entries in a row are small, the corresponding eigenvalue must be close to the diagonal entry.



e.g. $A = \begin{bmatrix} 1 & 2 \\ 1 & -1 \end{bmatrix}, \Lambda(A) = \{\sqrt{3}, -\sqrt{3}\}$
Disk 1: centred at (1,0), radius 2. D(1,2)
Disk 2: centred at (-1,0), radius 1. D(-1,1).

- True eigenvalues.
- Diagonal matrix entries.

Example

What are the Gershgorin disks for?

$$A = \begin{bmatrix} 5 & 2 & 1 \\ 2 & 4 & -1 \\ 1 & -1 & 2 \end{bmatrix}?$$

Can we determine any of the eigenvalues exactly?

Disks are: D(5,3), D(4,3), D(2,2). (No, we cannot pin any down exactly)

True eigenvalues are: $\Lambda(A) \approx \{0.7956, 3.6356, 6.5688\}$

What about diagonal matrices?

[Finding Eigenvectors and Eigenvalues]

Power Iteration

Start with an initial vector $v^{(0)}$. Repeatedly multiply A and normalize.

$v^{(1)} = Av^{(0)}/||Av^{(0)}||$

$v^{(2)} = Av^{(1)}/||Av^{(1)}||$

…

In the limit, this approaches the eigenvector, q_1, associated with the *largest* magnitude eigenvalue.

$$\lim_{k\to\infty} v^{(k)} \to q_1$$

Proof: Power Iteration

Let $v^{(0)}$ = approximate eigenvector if known, $||v^{(0)}|| = 1$

Let $\{q_i\}$ be the set of eigenvectors (normalized)

Then $v^{(0)} = c_1q_1 + c_2q_2 + \ldots + c_nq_n$

$Av^{(0)} = c_1\lambda_1q_1 + c_2\lambda_2q_2 + \ldots + c_n\lambda_nq_n$

Likewise $A^kv^{(0)} = c_1\lambda_1^kq_1 + c_2\lambda_2^kq_2 + \ldots + c_n\lambda_n^kq_n$

Example

$$A = \begin{bmatrix} 21 & 7 & -1 \\ 5 & 7 & 7 \\ 4 & -4 & 20 \end{bmatrix}, v^{(0)} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

r(x) = Rayleigh quotient

$w = Av^{(0)} = (27,19,20)^T$
$v^{(1)} = \frac{w}{||w||} \approx (0.7, 0.49, 0.52)$
$\lambda^{(1)} = r(v^{(1)}) \approx 23.3235$

$w = Av^{(1)} = (17.62, 10.57, 11.19)^T$
$v^{(2)} = \frac{w}{||w||} \approx (0.75, 0.45, 0.48)^T$
$\lambda^{(2)} = r(v^{(2)}) \approx 23.7250$

$w = Av^{(2)} = (18.50, 10.28, 10.71)^T$
$v^{(3)} = \frac{w}{||w||} \approx (0.78, 0.43, 0.45)^T$
$\lambda^{(3)} = r(v^{(3)}) \approx 23.8670$

True solution:
$q_1 \approx (0.8165, 0.4082, 0.4082)$
$\lambda_1 = 24$

Power Iteration Algorithm

$v^{(0)}$ = initial guess, s.t. $||v^{(0)}|| = 1$

for $k = 1, 2, \ldots$

$w = Av^{(k-1)}$ (Apply A)

$v^{(k)} = w/||w||$ (Normalize)

$\lambda^{(k)} = (v^{(k)})^TAv^{(k)}$ (Rayleigh quotient)

end

Comments on Power Iteration

We normalize at each step; keeps things in a reasonable range.

We (can) only recover q_1 (what about other eigenvectors?)

Let's consider convergence.

Power Iteration Convergence

Theorem: suppose $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \ldots \geq |\lambda_n|$, $q_1^T v^{(0)} \neq 0$

Then:

$$||v^{(k)} - (\pm q_1)|| = O\left(\left|\frac{\lambda_2}{\lambda_1}\right|^k\right), \text{and } |\lambda^{(k)} - \lambda_1| = O\left(\left|\frac{\lambda_2}{\lambda_1}\right|^{2k}\right)$$

As $k \rightarrow \infty$

Linear convergence. Convergence rate is $\left|\frac{\lambda_2}{\lambda_1}\right|$. When is this slow?

Slow if $|\lambda_2| \approx |\lambda_1|$ (i.e. 1st two eigenvalues close in magnitude).

Another Iteration

Let's try to find another eigenvector.

Can we use the same idea to recover the eigenvector for the *smallest* magnitude eigenvalue?

Consider the eigenvalues of A^{-1}

Inverse Iteration

$Ax = \lambda x$

$x = \lambda A^{-1} x$

$(1/\lambda)x = A^{-1}x$

We have $\Lambda(A^{-1}) = \{1/\lambda_i\}$ if $\Lambda(A) = \{\lambda_i\}$

…

This is called ***inverse iteration***.

Shifting

Inverse iteration still only lets us get q_n. "shifting" lets us find more.

Basic idea:

The smallest possible magnitude eigenvalue is zero.

Try to modify A so it has a "target" eigenvector with its eigenvalue near zero, so inverse iteration will find it.

Consider $B = A - \mu I$ with μ not an eigenvalue of A. What are its eigenvalues and eigenvectors?

Eigenvectors are the same. Eigenvalues are $\{\lambda_j - \mu\}$, for $\lambda_j \in \Lambda(A)$

So if is μ is close to λ_j, $\lambda_j - \mu$ is the smallest eigenvalue of B. Apply inverse iteration.

Example – Shifted Inverse Iteration

$$A = \begin{bmatrix} 21 & 7 & -1 \\ 5 & 7 & 7 \\ 4 & -4 & 20 \end{bmatrix}, \Lambda(A) = \{8, 16, 24\}$$

Try $\mu = 15$, with $v^{(0)} = (1, 1, 1)^T$. So $B = A - 15I$

$$w = B^{-1}v^{(0)} = (0.032, 0.16, 0.30)^T$$
$$v^{(1)} = \frac{w}{||w||} \approx (0.093, 0.46, 0.88)$$
$$\lambda^{(1)} = r\left(v^{(1)}\right) \approx 19.2$$

$$w = B^{-1}v^{(1)} = (-0.33, 0.40, 0.76)^T$$
$$v^{(2)} = \frac{w}{||w||} \approx ((-0.36, 0.44, 0.83)^T$$
$$\lambda^{(2)} = r\left(v^{(2)}\right) \approx 15.9749$$

$$w = B^{-1}v^{(2)} = (-0.39, 0.40, 0.79)^T$$
$$v^{(3)} = \frac{w}{||w||} \approx (-0.40, 0.41, 0.82)^T$$
$$\lambda^{(3)} = r\left(v^{(3)}\right) \approx 16.0290$$

True solution:
$q_2 \approx (-0.4082, 0.4082, 0.8165)$
$\lambda_2 = 16$

(Shifted) Inverse Iteration Algorithm

$v^{(0)}$ = initial guess, s.t. $||v^{(0)}|| = 1$
for $k = 1, 2, \ldots$
$w = (A - \mu I)^{-1} v^{(k-1)}$
$v^{(k)} = w/||w||$
$\lambda^{(k)} = (v^{(k)})^T A v^{(k)}$
end

Actually, solve linear system: $(A - \mu I)w = v^{(k-1)}$

Comments on (shifted) inverse iteration

Unlike power iteration, we can select the eigenvector we recover if we can choose μ close to the corresponding λ_J.

Like power iteration, inverse iteration has linear convergence.

Convergence of inverse iteration

Theorem: suppose λ_J is closest to μ, and λ_L is next closest.
i.e., $|\mu - \lambda_J| < |\mu - \lambda_L| \leq |\mu - \lambda_j|$, for $j \neq J$, and $q_J^T v^{(0)} \neq 0$.

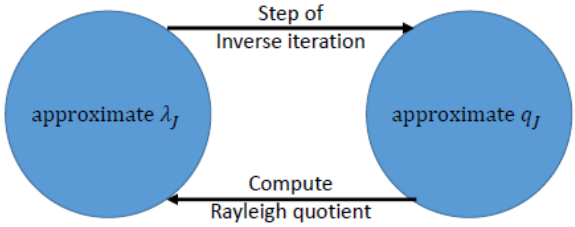
Then...

$$||v^{(k)} - (\pm q_J)|| = O\left(\left|\frac{\mu - \lambda_J}{\mu - \lambda_L}\right|^k\right), \text{and } |\lambda^{(k)} - \lambda_J| = O\left(\left|\frac{\mu - \lambda_J}{\mu - \lambda_L}\right|^{2k}\right)$$

as $k \to \infty$.

Rayleigh quotient iteration

Rayleigh quotient, r(v), gives eigenvalue estimate from eigenvector estimate (w/ quadratic convergence).
Inverse iteration gives eigenvector estimate from eigenvalue estimate μ (with linear convergence).
Combine the two. Inverse iteration, updating μ with latest guess for λ at each step.



Rayleigh quotient iteration algorithm

$v^{(0)} = \text{initial guess, s.t. } ||v^{(0)}|| = 1$

$\lambda^{(0)} = \left(v^{(0)}\right)^T A v^{(0)} \left(= r\left(v^{(0)}\right)\right)$

for $k = 1, 2, \ldots$

Solve $\left(A - \lambda^{(k-1)} I\right) w = v^{(k-1)}$.

$v^{(k)} = w/||w||$

$\lambda^{(k)} = \left(v^{(k)}\right)^T A v^{(k)}$

end

Using current λ estimate instead of fixed initial μ.

Rayleigh quotient iteration convergence

Theorem: RQI converges cubically for "almost all" starting vectors $v^{(0)}$.

$$||v^{(k+1)} - (\pm q_J)|| = O\left(||v^{(k)} - (\pm q_J)||^3\right)$$

And

$$\left|\lambda^{(k+1)} - \lambda_J\right| = O\left(\left|\lambda^{(k)} - \lambda_J\right|^3\right)$$

Each iteration roughly triples number of digits of accuracy.

Example – Rayleigh Quotient iteration

$$A = \begin{bmatrix} 21 & 7 & -1 \\ 5 & 7 & 7 \\ 4 & -4 & 20 \end{bmatrix}, v^{(0)} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

$\lambda^{(0)} = 22$

$\lambda^{(1)} = 24.0812$

$\lambda^{(2)} = 24.0013$

$\lambda^{(3)} = 24.00000017$

Operation Counts

Power iteration: each step involves $Av^{(k-1)}$ → $O(n^2)$ flops.

Inverse iteration: each step requires solving $(A - \mu I)w = v^{(k-1)}$. Would be $O(n^3)$ per step, but can pre-factor into L and U and reuse: $O(n^2)$ flops per step.

Rayleigh quotient iteration: matrix $A - \lambda^{(k-1)}I$ changes at each step, so $O(n^3)$ flops.

For all three, if A is tridiagonal, O(n) flops.

[Module 17 – QR Iteration – 06.24/07.06] (module 16/17)

Similarity

Our next goal is to find more than one eigenpair at a time. Need some definitions first.
Definition: $X \in R^{nxn}$ is nonsingular, then A → $X^{-1}AX$ is called a ***similarity transformation*** of A.
Definition: A and B are ***similar*** if $B = X^{-1}AX$ for some nonsingular X.

Theorem: if *A* and *B* are similar, then they have the same characteristic polynomial, hence the same eigenvalues.

Proof: $p_b(z) = \det(zI - X^{-1}AX)$
$= \det(X^{-1}(zI - A)X)$
$= \det(X^{-1})\det(zI - A)\det(X)$
$= \det(zI - A)$
$= p_A(z)$

QR Iteration

Idea: apply a sequence of similarity transformations to *A* that converge to a *diagonal* matrix, consisting of the eigenvalues. (Recall, we're assuming only real symmetric matrices)
We will rely on QR factorization again.

Given $A^{(k-1)}$, factor into $Q^{(k)}$ and $R^{(k)}$.
Hence $R^{(k)} = (Q^{(k)})^T A^{(k-1)}$

Multiplying by $Q^{(k)}$ on the left, gives a similarity transformation:
$A^{(k)} \equiv R^{(k)}Q^{(k)} = (Q^{(k)})^T A^{(k-1)} Q^{(k)}$
$A^{(k-1)}$ and $A^{(k)}$ are similar.
QR iteration applies this process repeatedly.

Basic QR iteration

$A^{(0)} = A$
for *k* = 1, 2, …
$Q^{(k)}R^{(k)} = A^{(k-1)}$ (Compute QR factors of $A^{(k-1)}$)
$A^{(k)} = R^{(k)}Q^{(k)}$ ("Recombine" in reverse order)
End

Eventually $A^{(k)}$ becomes diagonal (consisting of eigenvalues).
But how does it work?

QR Iteration example

$$A = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 3 & 1 \\ 1 & 1 & 4 \end{bmatrix} = A^{(0)}$$

$A^{(0)} = Q^{(1)}R^{(1)}$

$$A^{(1)} = R^{(1)}Q^{(1)} = \begin{bmatrix} 4.17 & 1.1 & -1.27 \\ 1.1 & 2 & 0 \\ -1.27 & 0 & 2.83 \end{bmatrix}$$

$A^{(1)} = Q^{(2)}R^{(2)}$

$$A^{(2)} = R^{(2)}Q^{(2)} = \begin{bmatrix} 5.09 & 0.16 & 0.62 \\ 0.16 & 1.96 & 0 \\ 0.62 & 0 & 2.05 \end{bmatrix}$$

Etc. ⇒

$$A^{(3)} = \begin{bmatrix} 5.09 & 0.16 & 0.62 \\ 0.16 & 1.96 & 0 \\ 0.62 & 0 & 2.05 \end{bmatrix}$$

True solution:
Λ(A) = {5.2143, 2.4608, 1.3249}

Extracting Eigenvectors

Since $A^{(k)}$ converges to eigenvalues on the diagonal, products of Q's gives the set of eigenvectors.
i.e. with

$\underline{Q}^{(k)} = Q^{(1)}Q^{(2)}\ldots Q^{(k)}$

We have the relation:

$A^{(k)} = (\underline{Q}^{(k)})^T A \underline{Q}^{(k)}$

Simultaneous iteration/Block power iteration

First consider the simpler (and related) notion of ***simultaneous iteration***. Then argue that it is the *equivalent* as QR iteration.

Simultaneous iteration:
Apply power iteration to *several* vectors at once, while maintaining linear independence among them.

Simultaneous iteration

Start with: $v_1^{(0)}, v_2^{(0)}, \ldots, v_p^{(0)}$
$A^k v^{(0)}$ converges to q_1 where $|\lambda_1|$ is largest, as usual.
But also, *span*$\{A^k v_1^{(0)}, \ldots, A^k v_p^{(0)}\}$ converges to *span*$\{q_1, \ldots, q_p\}$, where $\lambda_1, \ldots, \lambda_p$ are the *p* largest eigenvalues.

In matrix form, $V^{(0)} = \left[v_1^{(0)} v_2^{(0)} \cdots v_p^{(0)}\right]$ and $V^{(k)} = A^k V^{(0)}$

But all the vectors are converging to (multiples of) q_1, so they provide a *very* ill-conditioned basis for the space of eigenvectors.

Solution: orthonormalize the vectors at each step, using QR factorization.

Simultaneous iteration algorithm

Pick initial $\hat{Q}^{(0)} \in R^{n\,x\,p}$ with orthonormal columns.
for *k* = 1, 2, …

$Z^{(k)} = A\hat{Q}^{(k-1)}$ (Block) power iteration step
$\hat{Q}^{(k)}\hat{R}^{(k)} = Z^{(k)}$ (Reduced) QR factorization (as indicated by the "hats")

end

Column spaces of $\hat{Q}^{(k)}$ and $Z^{(k)}$ are the same, and both are equal to $A\hat{Q}^{(0)}$.

Assumptions

This relies on two assumptions:

1. The leading *p*+1 eigenvalues are distinct in absolute value.
   i.e., $|\lambda_1| > |\lambda_2| > \ldots > |\lambda_p| > |\lambda_{p+1}| \geq |\lambda_{p+2}| \geq \ldots \geq |\lambda_n|$
2. All the leading principal minors of $\hat{Q}^T V^{(0)}$ are nonsingular.

Convergence of block power iteration

Theorem:

Suppose block power iteration is applied and the preceding two assumptions are satisfied. Then as k → ∞,

$$||q_j^{(k)} - (\pm q_j)|| = O(c^k),\ j = 1, 2, \ldots, p$$

Where $c = \max_{1 \le k \le p} \left|\frac{\lambda_{k+1}}{\lambda_k}\right| < 1$

(i.e. linear convergence)

Simultaneous Iteration vs QR Iteration

QR can be viewed as simultaneous iteration with $\hat{Q}^{(0)} = I$ and $p = n$.

Since the matrices are square, we can drop hats on $\hat{Q}$ and $\hat{R}$.

Notation:

- $Q^{(k)}$ for Q's from QR iteration
- $\underline{Q}^{(k)}$ for Q's from simultaneous iteration, (i.e. product of $Q^{(k)}$'s)

We will now try to show equivalence. (s17)

Simultaneous iteration (algorithm)

Can be written as:

$Q^{(0)} = I$

for k = 1, 2, …

$$Z^{(k)} = A\underline{Q}^{(k-1)} \quad \text{(A)}$$

$$Z^{(k)} = \underline{Q}^{(k)} R^{(k)} \quad \text{(B)}$$

$$A^{(k)} = \left(\underline{Q}^{(k)}\right)^T A\underline{Q}^{(k)} \quad \text{(C)}$$

$$\underline{R}^{(k)} = R^{(k)} R^{(k-1)} \ldots R^{(1)} \quad \text{(D)}$$

} Added just for the proof!

end

QR Iteration

Can be written:

$A^{(0)} = A$

for k = 1, 2, …

$$A^{(k-1)} = Q^{(k)} R^{(k)} \quad \text{(A)}$$

$$A^{(k)} = R^{(k)} Q^{(k)} \quad \text{(B)}$$

$$\underline{Q}^{(k)} = Q^{(1)} Q^{(2)} \ldots Q^{(k)} \quad \text{(C)}$$

$$\underline{R}^{(k)} = R^{(k)} R^{(k-1)} \ldots R^{(1)} \quad \text{(D)}$$

} Added just for the proof!

end

Theorem

The two algorithms generate identical sequences of matrices, $\underline{R}^{(k)}$, $\underline{Q}^{(k)}$, $A^{(k)}$ satisfying:

$$A^k = \underline{Q}^{(k)} \underline{R}^{(k)} \quad (1)$$ QR factorization of ***k^th^* power** of A

$$A^{(k)} = (\underline{Q}^{(k)})^T A \underline{Q}^{(k)} \quad (2)$$ Similarity transform of A

Proof by induction.

<notes>

Convergence of QR

Observe:

(1) $A^k = \underline{Q}^{(k)}\underline{R}^{(k)}$ implies QR iteration is computing QR factors of A^k, i.e. an orthogonal basis for A^k.

(2) $A^k = (\underline{Q}^{(k)})^T A \underline{Q}^{(k)}$ implies diagonal of $A^{(k)}$ are Rayleigh quotients of column vectors $\underline{Q}^{(k)}$.

So columns of $\underline{Q}^{(k)}$ approach eigenvectors, these Rayleigh quotients approach eigenvalues.

What about off-diagonal entries of $A^{(k)}$?

$A_{ij}^{(k)} = (\underline{q_i}^{(k)})^T A \underline{q_j}^{(k)}$, where $\underline{q_i}^{(k)}$, $\underline{q_j}^{(k)}$ are columns of $\underline{Q}^{(k)}$.

As they converge to eigenvectors q_i, q_j, (which will be orthogonal) then

$$A_{ij}^{(k)} \approx q_i(\lambda q_j) \approx 0 \text{ for } i \neq j$$

Therefore, $A^{(k)}$ converges to diagonal.

Making QR Practical

QR factorization of dense square matrix is costly: $\sim \frac{4}{3}n^3$ flops.

Instead, increase sparsity first!

- If non-symmetric, reduce to upper Hessenberg → $O(n^2)$ for QR factorization
- If symmetric, reduce to tridiagonal → O(n) for QR factorization.

Reduction to Hessenberg (or tridiagonal)

In the general case, A can be non-symmetric.

But why reduce to Hessenberg, not triangular? Would that not be cheaper?

Attempt #1

First attempt: try to reduce A to triangular via Householder.

Apply Householder Q_1 to A.

$$A = \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix} \xrightarrow{Q_1^T \times} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} = Q_1^T A$$

But to maintain similarity, need to multiply by Q_1 on the right.

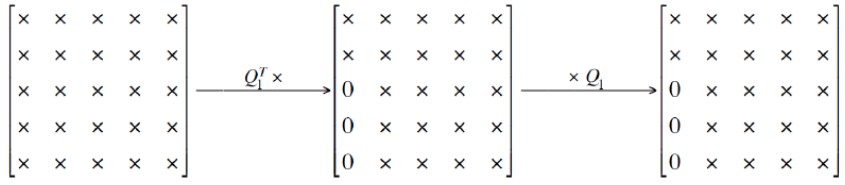
$$\begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} \xrightarrow{\times Q_1} \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix} = Q_1^T A Q_1$$

Newly created zeros are destroyed again!

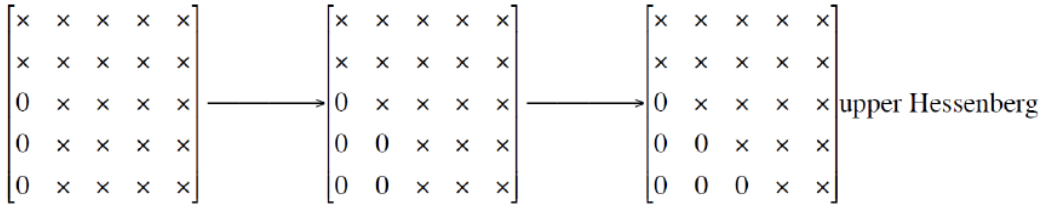
Attempt #2

Be a bit less ambitious: choose a different Q_1^T that leaves the **whole first row untouched**.

Then, when we multiply by Q_1 on the right, it won't wreck our progress (i.e. it will leave the 1st column unchanged)



A $\quad$ $Q_1^T A$ $\quad$ $Q_1^T A Q_1$



$Q_1^T A Q_1$ $\quad$ $Q_2^T Q_1^T A Q_1 Q_2$ $\quad$ $Q_3^T Q_2^T Q_1^T A Q_1 Q_2 Q_3$

$Q = Q_1 Q_2 \ldots Q_{n-2}$ and $Q^T A Q$ = upper Hessenberg.

Algorithm: Reduction to Hessenberg

Cost:
flops(reduction to Hessenberg) $\approx \frac{10}{3} n^3$
flops(reduction to tridiagonal) $\approx \frac{4}{3} n^3$

for k = 1, 2, …

$x = A(k+1:n, k)$

$v_k = v_k / ||v_k||$

for j = k, k+1,…,n

$A(k+1:n, j) = A(k+1:n, j) - 2v_k \left(v_k^T A(k+1:n, j) \right)$ } $Q_k^T \times$

end

for i = 1, 2,…,n

$A(i, k+1:n) = A(i, k+1:n) - 2(A(i, k+1:n) v_k) v_k^T$ } $\times Q_k$

end

end

Symmetric Case

If $A = A^T$, then

$(Q^T A Q)^T = Q^T A Q$

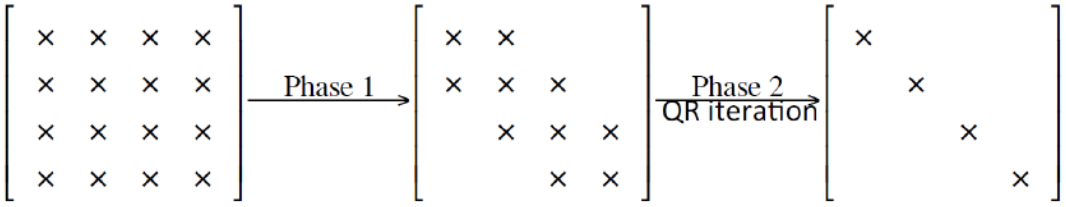
Is also symmetric.

A matrix that is both symmetric ***and*** Hessenberg is necessarily tridiagonal.

Two-Phase Process (symmetric case)

Phase 1: reduce A to tridiagonal via Householder operations (direct)

Phase 2: perform QR algorithm until convergence (iterative)



A $\quad$ $T = Q^T A Q$ $\quad$ D

More improvements

QR iteration can additionally be improved by:

- Applying ***shifting*** to achieve cubic convergence rates (similar to RQI)
- Breaking $A^{(k)}$ into sub-matrices when an eigenvalue is found.

Will not be explored further.

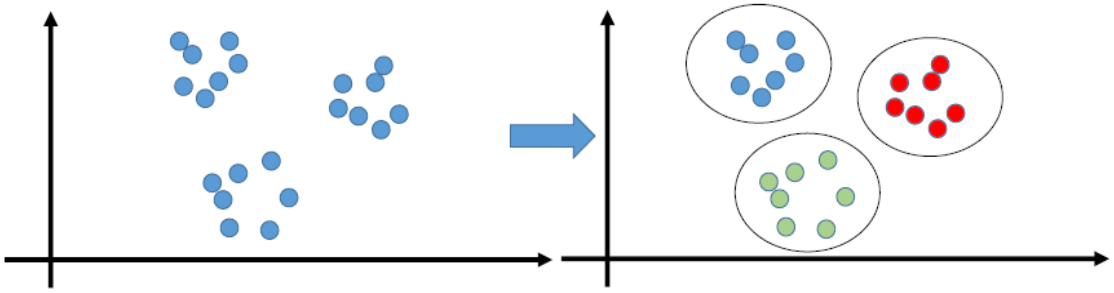
Summary

- QR iteration recovers all eigenvalues and eigenvectors, by repeating QR factorization and matrix multiplication
- It is equivalent to *block* power iteration (simultaneous iteration), in its basic form
- Reducing to Hessenberg/tridiagonal in pre-processing reduces cost.

[Module 18 – Application: Spectral Clustering for Image Segmentation – 07.08]

Spectral Clustering

A family of techniques that use the eigendecomposition of a matrix to identify clusters of "similar" or related elements in a data set.



Segmentation

Segmentation tasks (identifying distinct parts of an image or shape) often relies on clustering.

- E.g. image segmentation tries to group similar and nearby pixels.

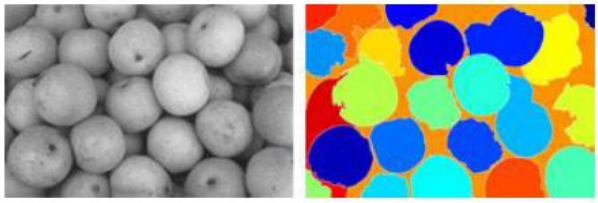
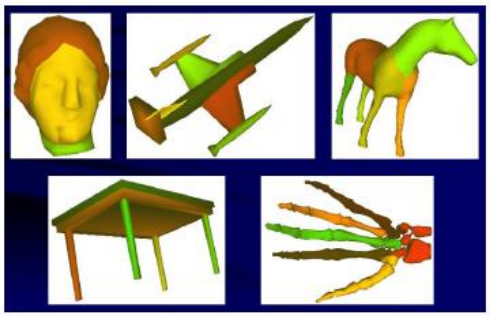


Image Segmentation



Shape Segmentation

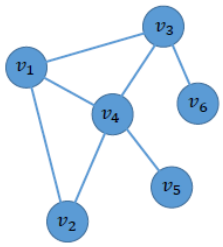
Spectral Clustering – Definitions

Definitions:

Let G = (V, E) be an undirected graph where $V = \{v_1, \dots, v_n\}$ set of vertices and $E = \{e_{ij}\}$ set of edges with e_{ij} = edge between v_i and v_j.

G is a *weighted* graph if edge e_{ij} has a weight $w_{ij} \geq 0$

W = (w_{ij}) = *weighted* adjacency matrix of the graph. Node has weight 0 on itself.



Degree of a vertex v_i: $d_i = \sum_{j=1}^{n} w_{ij}$

D = diag(d_i) = the "degree matrix"

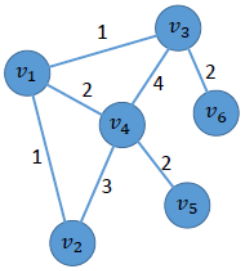
(Generalizing the usual notion of vertex degree)

Examples – Degrees and Weights

What is the weight matrix W?

What is the degree of v_4?

What is the degree matrix D?



$$W = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 0 \\ 1 & 0 & 0 & 3 & 0 & 0 \\ 1 & 0 & 0 & 4 & 0 & 2 \\ 2 & 3 & 4 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}, v_4 = 11, D = \begin{bmatrix} 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 7 & 0 & 0 & 0 \\ 0 & 0 & 0 & 11 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

Spectral Clustering - Definitions

Definitions:

Given A ⊂ V, indicator vector $1_A = (x_1, \ldots, x_n)$ is defined such that:

$$x_i = \begin{cases} 1, & if\ v_i \in A \\ 0, & if\ v_i \notin A \end{cases}$$

Given two subsets A, B define

$$W(A,B) = \sum_{i \in A, j \in B} w_{ij}$$

Examples – Weights and Indicators

For the two subsets of vertices, $A = \{v_1, v_2, v_6\}, B = \{v_3, v_4, v_5\}$:

What is the indicator vector 1_A, 1_B?

What is $W(A,B) = \sum_{i \in A, j \in B} w_{ij}$?

$1_A = [1, 1, 0, 0, 0, 1]^T$, $1_B = [0, 0, 1, 1, 1, 0]^T$

$W(A, B) = w_{12} + w_{13} + w_{14} + w_{24} + w_{34} + w_{36} + w_{45} = w_{13} + w_{14} + w_{24} + w_{36} = 1 + 2 + 3 + 2 = 8$

- j can be from A, i from B

Spectral Clustering - Definitions

We will consider two ways to measure the size of a subset A ⊂ V:

|A| = number of vertices in A

vol(A) = $\sum_{i \in A} d_i$ = sum of degrees of vertices in A

Example – Subset sizes

For the two subsets of vertices $A = \{v_1, v_2, v_6\}, B = \{v_3, v_4, v_5\}$

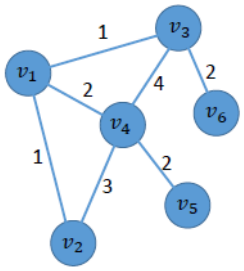
What is |A|? |B|?

What is vol(A)? vol(B)?

|A| = 3, |B| = 3

vol(A) = 4 + 4 + 2 = 10

vol(B) = 7 + 11 + 2 = 20



Graph Laplacians

A generalization of our finite difference Laplacian operator to arbitrary graphs. Consider two variants:

Unnormalized graph Laplacian:

L = D – W

Normalized graph Laplacian:

$$\hat{L} = I - D^{-\frac{1}{2}} W D^{-\frac{1}{2}}$$

Example – Graph Laplacians

Find L = D – W.

$$L = \begin{bmatrix} 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 7 & 0 & 0 & 0 \\ 0 & 0 & 0 & 11 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix} - \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 0 \\ 1 & 0 & 0 & 3 & 0 & 0 \\ 1 & 0 & 0 & 4 & 0 & 2 \\ 2 & 3 & 4 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 4 & -1 & -1 & -2 & 0 & 0 \\ -1 & 4 & 0 & -3 & 0 & 0 \\ -1 & 0 & 7 & -4 & 0 & -2 \\ -2 & -3 & -4 & 11 & -2 & 0 \\ 0 & 0 & 0 & -2 & 2 & 0 \\ 0 & 0 & -2 & 0 & 0 & 2 \end{bmatrix}$$

Unnormalized graph Laplacian **L = D – W**

Theorem: the unnormalized graph Laplacian L satisfies

1. for any vector x,

$$x^T L x = \frac{1}{2}\sum_{i,j=1}^{n} w_{ij}(x_i - x_j)^2$$

2. the smallest eigenvalue of *L* is 0, with corresponding eigenvector being the constant one vector, **1** = {1, …, 1}
3. *L* is symmetric and positive *semi*-definite, and has *n* non-negative eigenvalues $0 = \lambda_1 \le \lambda_2 \le \lambda_3 \le \ldots \le \lambda_n$

Normalized graph Laplacian $\hat{\boldsymbol{L}} = \boldsymbol{I} - \boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{W}\boldsymbol{D}^{-\frac{1}{2}}$

Theorem:

1. for any vector x,

$$x^T \hat{L} x = \frac{1}{2}\sum_{i,j=1}^{n} w_{ij}\left(\frac{x_i}{\sqrt{d_i}} - \frac{x_j}{\sqrt{d_j}}\right)^2$$

2. The smallest eigenvalue of $\hat{L}$ is 0 and the corresponding eigenvalue is $D^{1/2}\mathbf{1}$.
3. $\hat{L}$ is positive semi-definite and has *n* non-negative eigenvalues, $0 = \lambda_1 \le \lambda_2 \le \lambda_3 \le \ldots \le \lambda_n$

Graph Laplacians

Note: suppose the graph is a 2D grid (e.g. an image) using $w_{ij} = 1$. Then the unnormalized graph Laplacian *L* becomes (a multiple of) the usual 2D finite difference Laplacian.

Connected Components

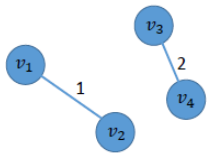
Theorem:

The multiplicity *k* of the eigenvalues 0 for both *L* and $\hat{L}$ equals the number of connected components $A_1, A_2, \ldots, A_k$ in the graph.

e.g.

$$L = \begin{bmatrix} 1 & -1 & & \\ -1 & 1 & & \\ & & 2 & -2 \\ & & -2 & 2 \end{bmatrix}$$

$$\Lambda = \{0,0,2,4\}$$



Multiplicity: number of times something appears in a set, i.e. number of times eigenvalue 0 appears in Λ

Overview

Show how graph Laplacians are used for clustering data.

1. Consider the problem of finding minimally weighted cuts in the graph.
2. This leads to NP-hard problems, so we "relax" them, to yield our spectral clustering algorithms.

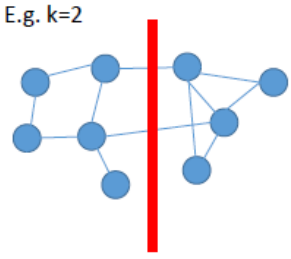
Graph Cuts

Problem: given a graph G with adjacency matrix W, find a partition of G such that the edges between the partitions have a very low weight.

mincut: find a partition $A_1, \ldots, A_k$ that minimizes

$$cut(A_1, A_2, \ldots, A_k) = \frac{1}{2}\sum_{i=1}^{k} W(A_i, \bar{A}_i)$$

Where $\bar{A}_i$ is the complement of A_i (vertices not in A_i).

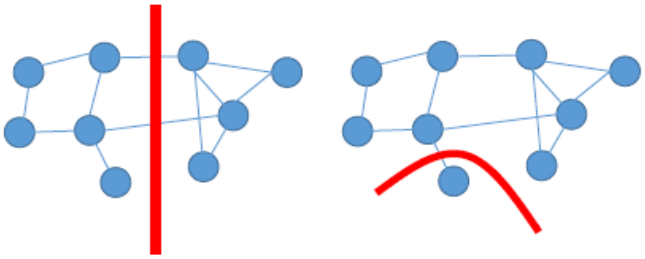


Mincut

Easy to solve, but doesn't give great results.

Minimal solution often separates individual vertices from the rest. Not a meaningful clustering.

e.g. k = 2



Preferred cut Computed cut

Better Cuts

Encourage the size of partitions to be larger, or rather more balanced.

Divide by "size" ($|A_i|$ or $vol(A_i)$)

Define: RatioCut, Ncut

$$RatioCut(A_1, A_2, \dots, A_k) = \frac{1}{2}\sum_{i=1}^{k}\frac{W(A_i, \overline{A_i})}{|A_i|} = \sum_{i=1}^{k}\frac{cut(A_i, \overline{A_i})}{|A_i|}$$

Recall: Number of vertices in the set.

$$Ncut(A_1, A_2, \dots, A_k) = \frac{1}{2}\sum_{i=1}^{k}\frac{W(A_i, \overline{A_i})}{vol(A_i)} = \sum_{i=1}^{k}\frac{cut(A_i, \overline{A_i})}{vol(A_i)}$$

Recall: Sum of "degrees" of vertices in the set.

Minimizing these is NP-hard.

RatioCut for k = 2

First, consider the case of partitioning into 2 subsets.

$$\min_{A} RatioCut\ (A, \bar{A})$$

We will rewrite it in terms of graph Laplacian.

Given a subset A ⊂ V, define $x = \{x_1, \dots, x_n\}$ where

$$x = \begin{cases} +\sqrt{\frac{|\bar{A}|}{|A|}} if\ v_i \in A \\ -\sqrt{\frac{|A|}{|\bar{A}|}}\ if\ v_i \in \bar{A} \end{cases}$$

It is possible to prove that:

$$x^T A x = |V| \cdot RatioCut(A, \bar{A})$$

$$\sum_{i=1}^{n} x_i = 0, i.e., x^T \mathbf{1} = 0$$

$$\|x\|^2 = n$$

So the minimization problem becomes

$$\min_{A \subset V} x^T L x \text{ subject to } x \perp \mathbf{1} \text{ and } \|x\| = \sqrt{n}$$

Still discrete and NP-hard. Vertices are in either A or $\bar{A}$. But relax…

Relax: allow x to consist of real numbers instead.

$$\min_{x \in R^n} x^T L x \text{ subject to } x \perp \mathbf{1} \text{ and } \|x\| = \sqrt{n}$$

The solution turns out to be the eigenvector corresponding to the 2nd smallest eigenvalue (AKA the "Fiedler vector").
Recover the separation into two clusters by thresholding:

$v_i \in A$ if $x_i \geq 0$
$v_i \in \bar{A}$ if $x_i < 0$

Ncut for k = 2

Same idea, different measure of set size.

$$\min_A Ncut\ (A, \bar{A})$$

Rewrite using normalized graph Laplacian.
Given a subset A ⊂ V, define *x* = {x_1, …, x_n} where

$$x = \begin{cases} +\sqrt{\dfrac{vol(\bar{A})}{vol(A)}} if\ v_i \in A \\ -\sqrt{\dfrac{vol(A)}{vol(\bar{A})}}\ if\ v_i \in \bar{A} \end{cases}$$

It is possible to prove that:

$$x^T A x = vol(V) \cdot Ncut(A, \bar{A})$$

$$\sum_{i=1}^{n} d_i x_i = 0, i.e., (Dx)^T \mathbf{1} = 0$$

$$x^T D x = vol(V)$$

So the minimization becomes

$$\min_{A \subset V} x^T L x \text{ subject to } Dx \perp \mathbf{1} \text{ and } x^T D x = vol(V)$$

Still discrete, NP-hard. But again, we can relax.

Relaxed problem:

$$\min_{x \in R^n} x^T L x \text{ subject to } Dx \perp \mathbf{1} \text{ and } x^T D x = vol(V)$$

Defining y = $D^{1/2}$x, the problem becomes:

$$\min_{y \in R^n} y^T \hat{L} y \text{ subject to } y \perp D^{1/2}1 \text{ and } \|y\|^2 = vol(V)$$

Solution is again the Fiedler vector, for $\hat{L}$ this time. Threshold y_i to determine the 2 clusters.

k > 2

What about k > 2 clusters? Cannot simply threshold at zero.
Make use of *k-means* clustering on data drawn from *several* eigenvectors.
First, the basic k-means algorithm.

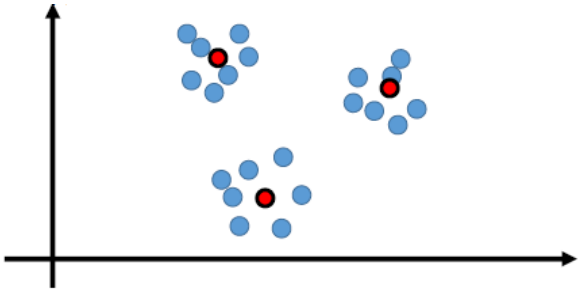
k-means

Given a set of *n* data points/vectors {p_j}, find a partition of the points A_1, …, A_k, such that each point is assigned to the set whose mean μ_i is closest to it.

k-means aims to solve the problem:

$$\min_{A_i} \sum_{i=1}^{k} \sum_{p \in A_i} \|p - \mu_i\|^2$$

E.g. in 2D, given blue data points, try to find k = 3 means, and an assignment of particles corresponding to 3 clusters.

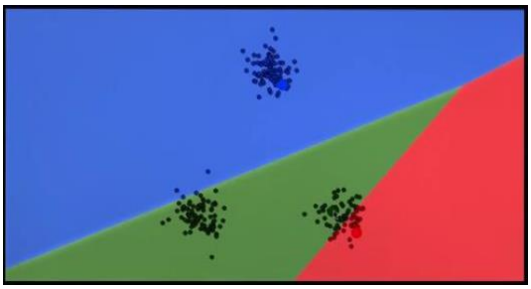


We expect something like the red points as the means of the 3 clusters.

k-means algorithm

1. Start with an initial guess for the *k* means $\{\mu_i\}$
2. Assign each point *p* to cluster A_i if *p* is closer to μ_i than any other mean.
3. Re-compute new mean positions $\{\mu_i\}$ for all partitions $\{A_i\}$
4. Repeat

A visualization



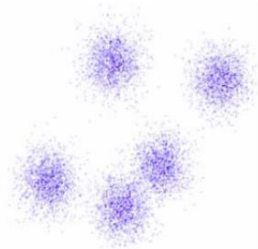
https://www.youtube.com/watch?v=fQXXa-CAoS0

- Compactness, e.g., k-means, mixture models
- Connectivity, e.g., spectral clustering

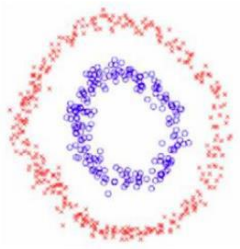
k-means vs spectral clustering

If k-means can do clustering, why not use it directly for our original clustering problem?

Spectral clustering allows more general similarity measures, and non-convex clusters.



Compactness



Connectivity

Back to *spectral* clustering

In the k = 2 case, we can use k-means rather than thresholding at zero, to assign points into two clusters.

Specifically, consider the entries of the eigenvector $\{x_i\}$ as *n* data points in R. Apply k-means with k = 2.

This ***can*** extend to k > 2 clusters.

Unnormalized Spectral Clustering Algorithm

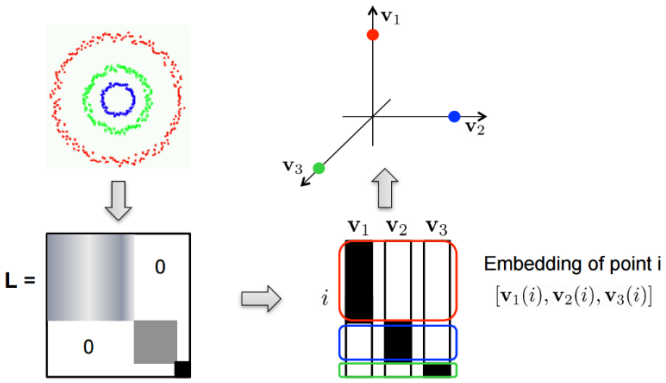
- Construct the unnormalized graph Laplacian *L*
- Compute the first *k* eigenvectors $q_1, q_2, \ldots, q_k$ of *L* (corresponding to ***smallest*** magnitude eigenvalues)
- Consider $Q_k = \{q_1, q_2, \ldots, q_k\}$. let $p_i \in R^k$ be the vector of row *I* of Q_k (i.e. $p_i = Q_k(i,:)$ in MATLAB notation)
- For the *n* points $\{p_i\}$ in R^k, apply *k*-means to cluster them into *k* groups: $\{A_1, \ldots, A_k\}$

Normalized Spectral Clustering

- Use $\hat{L}$ instead of L
- Use normalized rows, $\overline{p_i} = \frac{p_i}{\|p_i\|}$ in place of p_i

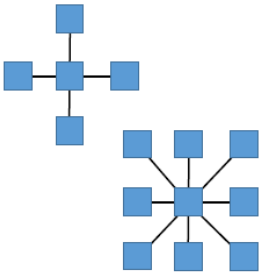
Intuition

Eigenvectors provide a new representation of the data (based on similarity) as points in R^k, on which standard k-means clustering is very effective.



Choosing Weighting W

Choice of the weight matrix *W* measures similarity between vertices of the graph; it is *problem dependent*.
Usually, we consider non-zero weights only between a small set of local neighbours, e.g. within graph distance of 1 or 2.
More neighbours implies more non-zeros in *W*.
Few neighbours ensures sparsity of the graph Laplacian.



Choosing W for images

For image segmentation: view pixels as graph vertices.
Neighbour relationships imply graph edges.
Considering the 4 adjacent pixels, *W* has a non-zero structure similar to usual finite difference Laplacian.
Including the four diagonal neighbours too gives 8 neighbours, W has at most 8 non-zeros per row.

w_{ij} measures the similarity between pixels *i* and *j* using:

- Distance between pixels i and j
- Intensity difference between pixels *i* and *j*

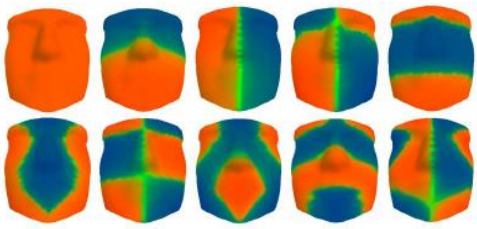
We will use:

$$w_{ij} = (e^{-\frac{\|x_i - x_j\|^2}{\sigma_{dist}^2}})(e^{-\frac{\|I_i - I_j\|^2}{\sigma_{int}^2}})$$

Where pixel *i* is at position x_i with intensity I_i, likewise for pixel *j*.
Define positions as x_i = (r, c) if pixel *i* is at row *r*, column *c*.
σ_{dist}^2 and σ_{int}^2 are parameters to vary the relative importance of the terms.

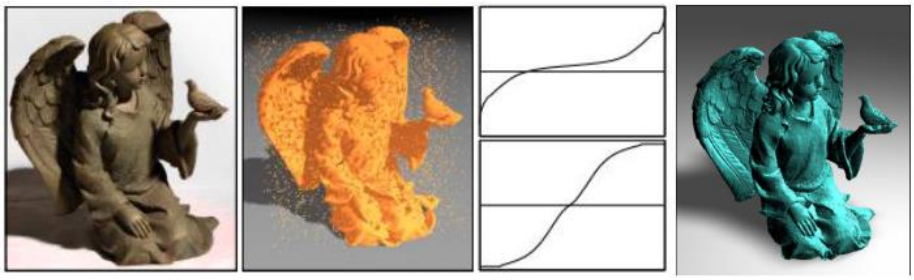
Spectral Processing

Spectral approaches find many other applications.
E.g. in geometric mesh processing. These are visualizations of several eigenvectors associated to Laplacians defined on 3D triangle meshes.



https://www.cs.sfu.ca/~haoz/pubs/zhang_eg07star_spectral.pdf

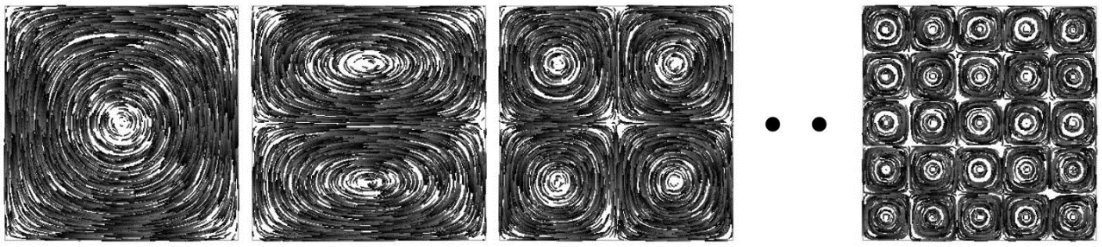
E.g. mesh reconstruction from noisy scanned point data (segmenting interior region from exterior).



http://graphics.berkeley.edu/papers/Kolluri-SSR-2004-07/Kolluri-SSR-2004-07.pdf

E.g. extracting the dominant motion "modes" in solids or fluids for analysis or efficient simulation.

https://en.wikipedia.org/wiki/Vibrations_of_a_circular_membrane



Basis of divergence free fields that are eigenfunctions of the vector Laplacian and satisfy a free-slip boundary condition (equation)
Basis fields have correspondence with spatial scales of vorticity.
Their coefficients form a discrete spectrum.

➔ Increasing eigenvalue magnitude (figure showing bar graph of spectrum)

Spectral Graph Theory/Additional material

- Tutorial on course website

[Module 19 – Singular Value Decomposition – 07.13]

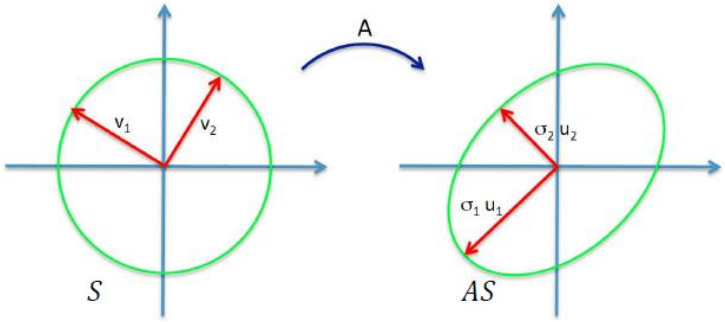
Singular Value Decomposition

The final of the course's four main decompositions:

- LU/Cholesky
- QR
- Eigendecomposition
- **Singular Value Decomposition (SVD)**

Geometric Motivation

The *image* of the unit (hyper)sphere *S* in R^n under a m x n matrix transformation is a hyperellipse in R^m.



Singular Value Decomposition

The *n* ***singular values***, σ_i, of A are the lengths of the principal semi-axes of the ellipse *AS*.
By convention, we order them such that:

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$$

The *n* ***left singular vectors***, u_i, of A are the unit vectors in the directions of the principal semi-axes of the ellipse.

The *n* ***right singular vectors***, v_i, are the unit vectors in S, such that:

$$Av_j = \sigma_j u_j$$

i.e. the v_i's are the pre-images of u_i's under *A*.

Reduced SVD

$$Av_j = \sigma_j u_j \text{ for } i = 1,2,\dots,n$$

$$\begin{bmatrix} A \end{bmatrix}_{m\times n} \begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix}_{n\times n} = \begin{bmatrix} u_1 & u_2 & \dots & u_n \end{bmatrix}_{m\times n} \begin{bmatrix} \sigma_1 & & & \\ & \dots & & \\ & & & \sigma_n \end{bmatrix}_{n\times n}$$

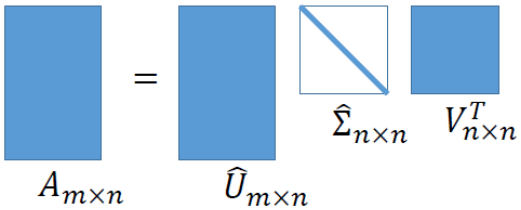
$$A \qquad V \qquad = \qquad \hat{U} \qquad \hat{\Sigma}$$

$\hat{\Sigma}$ is diagonal. $\hat{U}$ and *V* have orthonormal columns.
(Hat notation indicates reduced SVD)

$$AV = \hat{U}\hat{\Sigma}$$

In fact, V is orthogonal, so multiplying by V^T on the right, we can equivalently write:

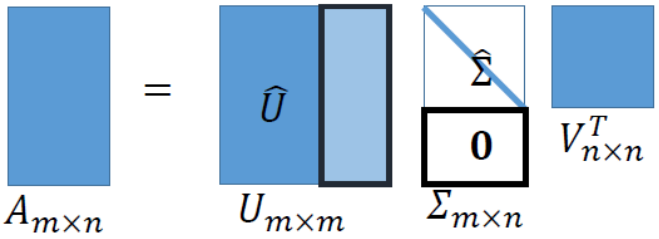
$$A = \hat{U}\hat{\Sigma}V^T$$



Full SVD

As for (full) QR, we can also define a ***full SVD***, by adding m – n more orthonormal columns to $\hat{U}$ to give square, orthogonal U.

Replace $\hat{U} \rightarrow U$ and add extra empty rows to $\hat{\Sigma} \rightarrow \Sigma$.



Theorem

Every matrix $A \in R^{m\,x\,n}$ has a singular value decomposition.
The singular values are uniquely determined.
If A is square and σ_j are distinct, then the left and right singular vectors are unique (*up to signs*).

SVD vs Eigendecomposition

$$\mathrm{A} = \mathrm{U}\Sigma\mathrm{V}^T \; vs \; A = X\Lambda X^{-1}$$

- Both act to diagonalize a matrix
  - SVD uses two bases: U and V, the left and right singular vectors
  - Eigendecomposition uses one basis, the set of eigenvectors.
- SVD uses orthonormal vectors. Eigenvectors are not orthogonal in the general case (though for real 55 matrices they are)
- Not all matrices have an eigendecomposition, *all matrices* have a SVD.

Properties of SVD

Let $A \in R^{m\,x\,n}$, *p* = min(m, n), and *r* = # of non-zero singular values.

Theorem: *rank*(A) = *r* (proved)
Theorem: $range(A) = span\{u_1, u_2, \dots, u_r\}, null(A) = \{v_{r+1}, \dots, v_n\}$ (!proved)
Theorem: $\|A\|_2 = \sigma_1$ and $\|A\|_F = \sqrt{\sigma_1 + \sigma_2 + \cdots + \sigma_r}$ (proved)
Theorem: non-zero singular values of A are the square roots of non-zero eigenvalues of AA^T or A^TA. (proved)
Theorem: if A = A^T, then $\sigma(A) = \{|\lambda| : \lambda \in \Lambda(A)\}$. In particular, if A is SPD, $\sigma(A) = \Lambda(A)$ (proved)
Theorem: the condition number $k_2(A) = \sigma_1/\sigma_n$, for $A \in R^{m\,x\,n}$ (proved)
Proofs in notes

Computing the SVD

$\mathrm{A} = \mathrm{U}\Sigma\mathrm{V}^T$ so in general,

$$\begin{aligned} A^TA &= (U\Sigma V^T)^T(U\Sigma V^T) \\ &= V\Sigma U^T U\Sigma V^T \\ &= V\Sigma^2 V^T \end{aligned}$$

← An eigendecomposition of A^TA!

Therefore:
Eigenvalues of A^TA are the *squares* of the singular values of *A*.
Eigenvectors of A^TA are the right singular vectors of *A*.

Algorithm

This gives us a (naïve) methods:

1. Form A^TA.
2. Compute eigendecomposition $A^TA = V\Lambda V^T$
3. Compute $\Sigma = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \dots & \\ & & & \sigma_n \end{bmatrix}, \sigma_i = \sqrt{\lambda_i}, \Lambda = \begin{bmatrix} \lambda_1 & & & \\ & \dots & & \\ & & & \lambda_n \end{bmatrix}$
4. Solve $\mathrm{U}\Sigma = \mathrm{AV}$ for orthogonal *U*. (e.g. by QR factorization)

Naïve Algorithm

Unfortunately, this method is unstable/inaccurate; the error satisfies

$$|\widetilde{\sigma_k} - \sigma_k| = O(\frac{\varepsilon\|A\|^2}{\sigma_k})$$

Which can be very bad for smaller singular values.

(*Conceptually* similar to how normal equations use A^TA, effectively "squaring the condition number", and making it more susceptible to numerical error than approaches based on QR)

Later, will discuss an alternative.

Example

Find the SVD of $A = \begin{bmatrix} 0 & -1/2 \\ 3 & 0 \\ 0 & 0 \end{bmatrix}$

Alternative Formulation

Assume A is square, *n* x *n*, and consider the 2n x 2n symmetric matrix:

$$H = \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix}$$

Compute the eigendecomposition of *H*. Then $\sigma_A = |\lambda_H|$, and U, V can be recovered from the eigenvectors.

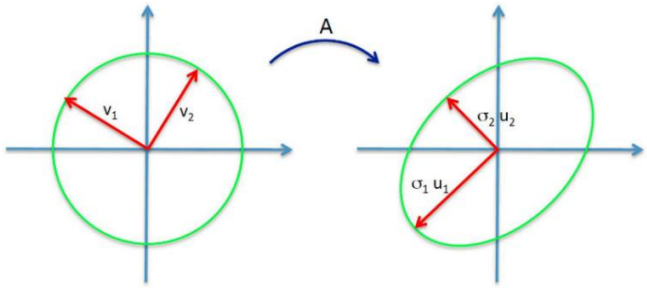
Let's see why (and how).

This algorithm is more stable: $|\widetilde{\sigma_k} - \sigma_k| = O(\varepsilon\|A\|)$

[Module 20 – More Fun with SVD – 07.15]

Singular Value Decomposition

Decomposition of any matrix A into $U\Sigma V^T$, where Σ is diagonal with non-negative entries, and U, V are orthogonal.



The SVD can be found from the eigendecomposition of A^TA or AA^T, or more stably from $H = \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix}$

Proof of Existence of SVD

We claimed last time that every matrix $A \in R^{m\,x\,n}$ has a singular value decomposition.
Proof: <notes>

Alternative Formulation

Assume A is square, *n* x *n*, consider the 2n x 2n symmetric matrix:

$$H = \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix}$$

Compute the eigendecomposition of $H = Q\Lambda Q^T$, giving:

$$\begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix}\begin{bmatrix} V & V \\ U & -U \end{bmatrix} = \begin{bmatrix} V & V \\ U & -U \end{bmatrix}\begin{bmatrix} \Sigma & 0 \\ 0 & -\Sigma \end{bmatrix}$$

We can then extract the singular values and vectors.
This algorithm is more stable: $|\widetilde{\sigma_k} - \sigma_k| = O(\varepsilon\|A\|)$

Alternative Formulation – Algorithm

1. Form $H = \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix}$
2. Compute eigendecomposition *HQ* = *Q*Λ
3. Set $\sigma_A = |\lambda_H|$
4. Extract *U*, *V* from Q (normalizing for orthogonality)

Can be extended to non-square matrices too.
Practical algorithms are based on this premise, but without explicitly forming the (large) matrix *H*.

Alternative Formulation – Example

$$A = \begin{bmatrix} 0 & -1/2 \\ 3 & 0 \end{bmatrix}$$

Stability Comparison

Assume we have a (backward) stable algorithm for finding eigenvalues, such that

$$|\widetilde{\lambda_k} - \lambda_k| = O(\varepsilon_{machine}\|A\|)$$

This satisfies $\widetilde{\lambda_k} = \lambda_k(A + \delta A)$, with $\frac{\|\delta A\|}{\|A\|} = O(\varepsilon_{machine})$

i.e. we get the exact eigenvalues for a slightly perturbed matrix, A + δA.

Applying this to $H = \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix}$, we can get the singular values with:

$$|\widetilde{\sigma_k} - \sigma_k| = |\widetilde{\lambda_k} - \lambda_k| = O(\varepsilon_{machine}\|H\|) = O(\varepsilon_{machine}\|A\|)$$

But let's apply our stable eigenvalue routine to A^TA:

$$|\widetilde{\lambda_k} - \lambda_k| = O(\varepsilon_{machine}\|A^TA\|) = O(\varepsilon_{machine}\|A\|^2)$$

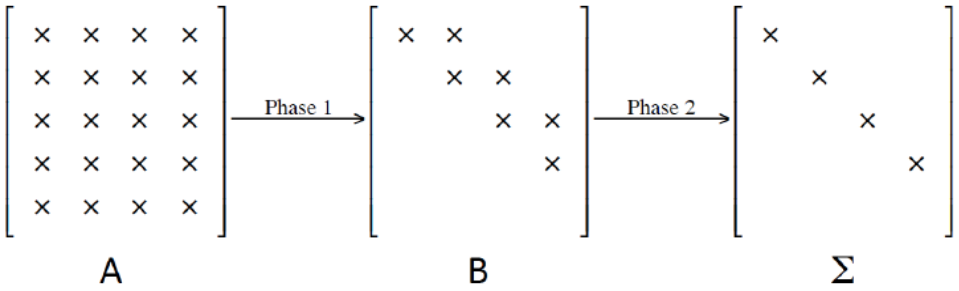
Taking square roots to get σ_k gives

$$|\widetilde{\sigma_k} - \sigma_k| = O\left(\frac{|\widetilde{\lambda_k} - \lambda_k|}{\sqrt{\lambda_k}}\right) = O\left(\frac{\varepsilon_{machine}\|A\|^2}{\sigma_k}\right)$$

As claimed, this behaves badly for $\sigma_k \ll \|A\|$

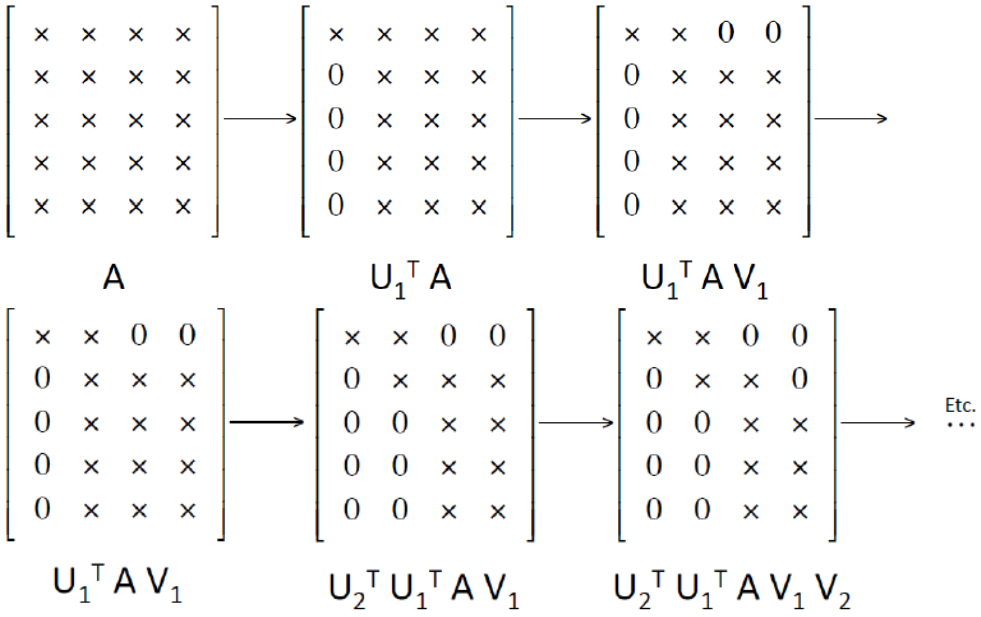
A Two-Phase Process

As for QR iteration (to find eigendecompositions), we can pre-process the matrix to reduce cost of computing the SVD.
Idea: convert to *bidiagonal*, then extract SVD.



Golub-Kahan Bidiagonalization – Why bidiagonal?

Don't have to maintain similarity (as we did for eigendecomposition), so we can apply *different* Householder reflectors on left and right to introduce zeros.



Golub-Kahan Bidiagonalization

Cost:

Uses *n* reflectors on the left, *n* – 2 on the right.

$$flops(bidiagonalization) \approx 2\ x\ flops(QR) \approx 4mn^2 - \frac{4}{3}n^3$$

Low rank approximation with SVD

Theorem: A is the sum of *r* rank-one matrices:

$$A = \sum_{j=1}^{r} \sigma_j\, u_j v_j{}^T$$

Proof: from definition of SVD:

$$A = [u_1 \dots u_m] \begin{bmatrix} \sigma_1 & & \\ & \cdots & \\ & & \sigma_r \\ & 0 & \end{bmatrix} \begin{bmatrix} v_1^T \\ \dots \\ v_n^T \end{bmatrix} = [u_1 \dots u_m] \begin{bmatrix} \sigma_1 v_1^T \\ \dots \\ \sigma_r v_r^T \\ 0 \end{bmatrix}$$

$$\therefore\ A = \sigma_1 u_1 v_1^T + \cdots + \sigma_r u_r v_r^T$$

Low rank approximation

Define an *approximate* version of *A* using the first *k* singular values.

$$A_k = \begin{bmatrix} u_1 & \dots & u_m \end{bmatrix} \begin{bmatrix} \sigma_1 & & \\ & \cdots & \\ & & \sigma_k \\ & 0 & \end{bmatrix} \begin{bmatrix} v_1^T \\ \dots \\ v_n^T \end{bmatrix} = \underbrace{\begin{bmatrix} u_1 & \dots & u_k \end{bmatrix}}_{U_k} \underbrace{\begin{bmatrix} \sigma_1 & & \\ & \dots & \\ & & \sigma_k \end{bmatrix}}_{\Sigma_k} \underbrace{\begin{bmatrix} v_1^T \\ \dots \\ v_k^T \end{bmatrix}}_{V_k^T}$$

So we may write $A_k = U_k \Sigma_k V_k^T$

Low rank approximation with SVD

Theorem: for any k, $0 \le k \le r$, define:

$$A_k = \textstyle\sum_{j=1}^{k} \sigma_j\, u_j v_j{}^T$$

Then $\|A - A_k\|_2 = \inf_{rank(B) \le k} \|A - B\| = \sigma_{k+1}$

i.e. over all matrices B with the same or lesser rank, A_k minimizes $\|A - A_k\|_2$, i.e. it provides the best rank *k* approximation of *A*, with an approximation error of σ_{k+1}

Proof part 1

We can see $\|A - A_k\|_2 = \sigma_{k+1}$ using the definition of SVD:

$$A - A_k = \sum_{j=k+1}^{r} \sigma_j u_j v_j^T = [u_1 \quad \dots \quad u_m] \begin{bmatrix} 0 & & \\ & \sigma_{k+1} & \\ & & \dots \\ & & \sigma_r \\ & 0 & \end{bmatrix} \begin{bmatrix} v_1^T \\ \dots \\ v_n^T \end{bmatrix}$$

i.e. this gives the SVD for $A - A_k$.

We showed that $||A||_2 = \sigma_1$, so we have $\|A - A_k\|_2 = \sigma_{k+1}$ (the largest remaining singular value).

Proof part 2

We will show $\|A - A_k\|_2 = \inf_{rank(B)\leq k} \|A - B\|_2$ using a proof by contradiction.

<Notes, 07.15>

Suppose there exists B such that rank(B) <= k and $\|A - B\|_2 < \sigma_{k+1}$, i.e. a better approximation of A.

The null(B) has dimensions >= n – k (by rank nullity theorem) and contains vectors v such that $B_v = 0$.

Since dim(null(B)) >= n – k and dim(span{v1,…,v_{k+1}}) = k + 1, therefore (n – k) + (k + 1) > n, so the spaces must "overlap", i.e. there exists z != 0 such that z is in the intersection, $null(B) \cap span\{v_1, \dots, v_{k+1}\}$

Let ||z|| = 1. We will show that ||A – B|| >= σ_{k+1} to give a contradiction.

Note that $||A - B||_2^2 \geq ||(A - B)_z||^2$ (by definition of matrix 2-norm $||A||_2 = \max_{\|x\|=1} \|Ax\|$

Since $z \in null(B), B_z = 0$

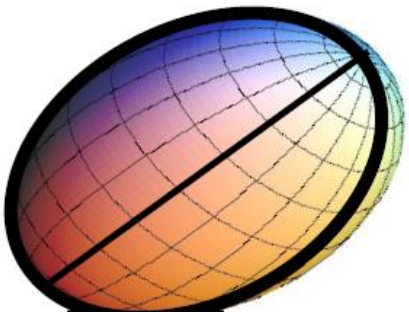
Therefore $||(A - B)_z||^2 = \|Az\|^2$

But $z \in span\{v_1, \dots, v_{k+1}\}$ $\|Az\|^2 = \|\sum_{i=1}^{n} \sigma_i u_i {v_i}^T z\|^2 = \sum_{i=1}^{k+1} {\sigma_i}^2 ({v_i}^T z)^2 \geq \sigma_{k+1}^2 \sum_{i=1}^{k+1} {v_i}^T z = \sigma_{k+1}^2$

$||A - B||_2^2 \geq \sigma_{k+1}^2$ contradicting $||A - B||_2^2 < \sigma_{k+1}^2$

So there can exist no such B.

Geometric interpretation of low-rank approximation



What is the line segment that best approximates an ellipsoid?

A segment along the longest axis.

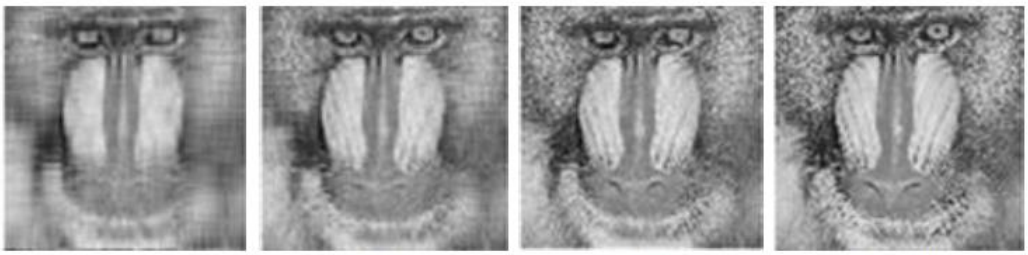
Ellipse that best approximates an ellipsoid?

An ellipse spanning the two longest axes.

Etc.

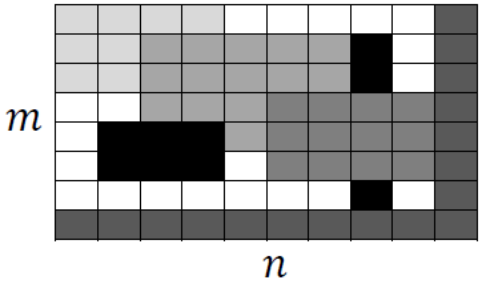
Application of SVD to Image Compression

SVD can be used to produce ae cheaper approximate version of an image that captures the "most important" parts.



K=10 K=20 K=50 Original

Images



Consider a *m* x *n* pixel (greyscale) image as an *m* x *n* matrix *A* where A_{ij} is the intensity of pixel (*i*, *j*).

If we can store fewer than *mn* entries, we have a compressed representation.

Let $A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$, the best rank-*k* approximation of *A*.
A_k gives a compressed version of the image *A* using the first k singular values.

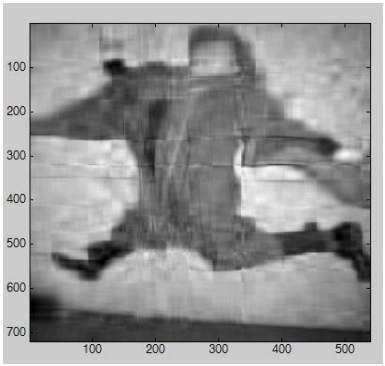
E.g. *m* = 320, *n* = 200.
For A_k, need only store vectors $u_1, u_2, \dots, u_k$ and $\sigma_1 v_1, \sigma_2 v_1, \dots, \sigma_k v_k$.
Thus we have (*m* + *n*)*k* entries.
This gives a compression ratio of $\frac{(m+n)k}{mn}$
For our example, $\frac{(320+200)k}{320 \cdot 200} \approx \frac{k}{123}$

k	Rel. err. σ_{k+1}/σ_1	Comp. ratio
3	0.155	2.4%
10	0.077	8.1%
20	0.04	16.3%

Image Compression – Example

For example, compress this image (in grayscale):

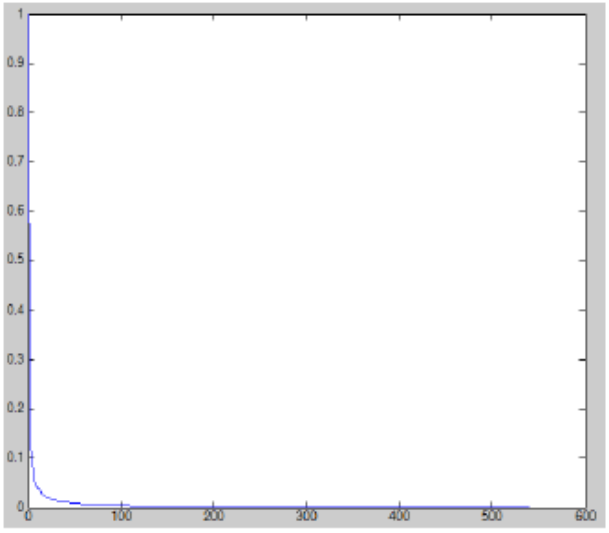
```
A=rgb2gray(imread('jump.jpg'));
A=double(A);
[U,S,V] = svd(A);
k=30; %try different choices
Ak=U(:,1:k)*S(1:k,1:k)*V(:,1:k)';
colormap('gray');
imagesc(Ak);
```

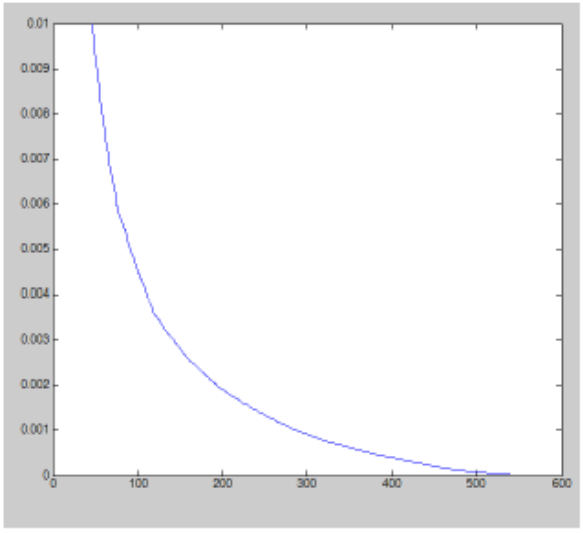


Your very dignified professor upon receiving his PhD.

Singular values as error measures

We said that $\|A - A_k\|_2 = \sigma_{k+1}$ gives the approximation error. So we can plot the relative error as σ_{k+1}/σ_1 against the choice of *k*.





[Module 21 – Convergence of Iterative Schemes – 07.20]

Convergence of Iterative Schemes

We'll take a closer look at this topic, starting with Richardson, Jacobi, and Gauss-Seidel.

Then consider the specific case of a Laplacian matrix.

Stationary Iterative Methods – Recall

Basic iterative methods amount to different choices of M, in $A = M - N$.

Generic iteration:
$x^{k+1} = x^k + M^{-1}(b - Ax^k)$

$$A = \begin{bmatrix} \ddots & & -U \\ & D & \\ -L & & \ddots \end{bmatrix}$$

- Richardson iteration: $M = \frac{1}{\Theta} I$ for scalar Θ
- Jacobi: $M = D$
- Gauss-Seidel: $M = D - L$
- Successive-Over-Relaxation ("SOR"): $\frac{1}{\omega} D - L$ for scalar ω

We can write this (the generic iteration as):

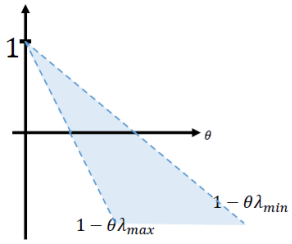
$$x^{k+1} = (I - M^{-1}A)x^k + M^{-1}b$$

Then $I - M^{-1}A$ is the iteration matrix for the scheme.
The method converges if and only if $\rho(I - M^{-1}A) < 1$
Lower ρ implies faster convergence to solution.

We will consider their behaviour for SPD matrices.

Optimal Θ for Richardson

Eigenvalues of $A \in [\lambda_{min}, \lambda_{max}]$, so eigenvalues of Richardson iteration matrix $I - \Theta A$ are in $[1 - \Theta\lambda_{min}, 1 - \Theta\lambda_{max}]$.

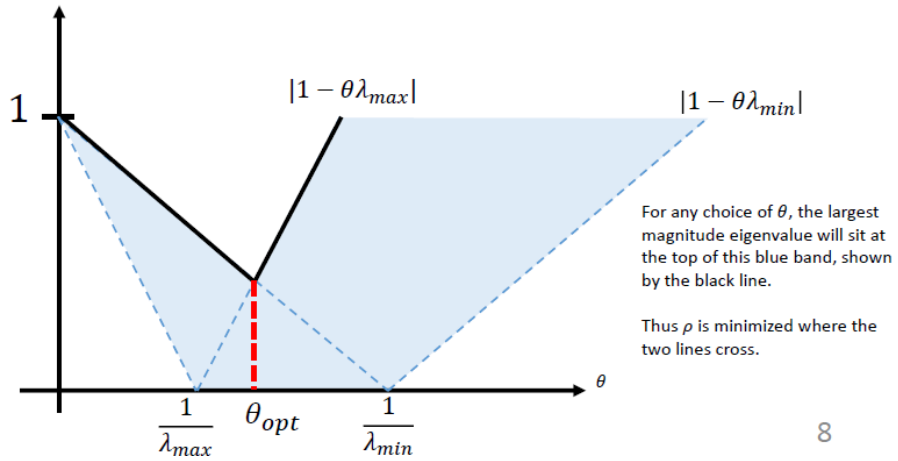


Plotting this range gives us the light blue band.

But to get the minimum spectral radius, we are interested in the absolute value. So reflecting over x-axis gives us the diagram on the next page.

Choosing Optimal Θ for Richardson

Find intersection of $|1 - \Theta\lambda_{min}|$ and $|1 - \Theta\lambda_{max}|$



For any choice of θ, the largest magnitude eigenvalue will sit at the top of this blue band, shown by the black line.

Thus ρ is minimized where the two lines cross.

[Module 22 – Convergence: Iterative Schemes and Laplacian Matrices – 07.22]

Convergence of Iterative Schemes

Analytical expressions can be found for the eigenvectors/eigenvalues of certain matrices.
This can give us sense for how our iterative schemes fare in practice.

We will consider the familiar 2D finite difference Laplacian matrix for the Poisson equation:

$$-\nabla \cdot \nabla u = f$$

Why the negative sign?

Example: 2D Laplacian

A terse way of building our 2D Laplacian in Matlab is the following:

```
I = speye(m, m);
E = sparse(2:m, 1:m – 1, 1, m, m);
D = 2*I – E – E';
A = (kron(D, I) + kron(I, D))/h^2;
```

Uses the Kronecker product: $A \otimes B = \begin{bmatrix} a_{11}B & \dots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \dots & a_{mn}B \end{bmatrix}$

Conjugate Gradient – Convergence

Theorem:

$$\left\|x^k - x\right\|_A^2 \leq 2\left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1}\right)^k \|x^0 - x\|_A$$

With $\kappa(A) = \frac{\lambda_{max}}{\lambda_{min}}$

But it's just an *upper bound*. Convergence can be much better.

Conjugate Gradient convergence – example

Real convergence behaviour depends on all the λ, not just max and min.
e.g. consider an A with 3 distinct eigenvalues $\lambda_1 < \lambda_2 < \lambda_3$.
We can pick a good polynomial to show faster convergence!

First, write the error in terms of eigenvectors v_j of A.

$e^0 = \sum_{i=1}^{n} \xi_j v_j$ for some ξ_j

Note also that $P_k(A)v = P_k(\lambda)v$ will hold if v is an eigenvector.

Fast convergence example

Construct a Lagrange polynomial $P_3(x)$ with degree ≤ 3 such that:

$P_3(0) = 1$ and $P_3(\lambda) = 0$, for $j = 1, 2, 3$

Then some algebra shows that

$$||e^3||_A \leq ||P_3(A)e^0||_A^2 = \sum_{j=1}^{3} \xi_j^2 P_3^2(\lambda_j)\lambda_j = 0$$

i.e. converges in 3 iterations, independent of k(A) (since CG's error has to be at least as good as that produced by any other possible polynomial)

CG vs SOR

For the Poisson equation, asymptotic convergence rates are the same for CG and SOR.

However, CG doesn't require pre-determining/guessing an optimal relaxation parameter, ω.

Summary

- Explored convergence of various iterative methods for linear systems in more depth
- Examined the Laplacian matrix in particular

[Module 23 – Convergence: CG & Preconditioning – 07.27]

Last Time

We concluded by arguing that CG at each step finds the polynomial P_k that minimizes the error in the A-norm:

$$||e^k||_A = \min\{\, ||P_k(A)e^0||_A : P_k(x) = \text{poly of deg} \le k \text{ and } P_k(0) = 1\}$$

So for any poly $\tilde{P}_k$ with degree ≤ *k* and $\tilde{P}_k(0) = 1$, we have

$$\|e^k\|_A \le \|\tilde{P}_k(A)e^0\|_A$$

This can be used to derive bounds on convergence.

Conjugate Gradient – Convergence

One particular choice of polynomial leads to the following bound.

Theorem:

$$\|x^k - x\|_A^2 \le 2\left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1}\right)^k \|x^0 - x\|_A$$

With $\kappa(A) = \frac{\lambda_{max}}{\lambda_{min}}$

But this is just an upper bound. Convergence can be much better.

[Preconditioning]

Preconditioning – Idea

We want to solve $Ax = b$ with an iterative scheme (e.g. CG)

Solve a modified linear system, with better conditioning, κ(A) = $\frac{\lambda_{max}}{\lambda_{min}}$, therefore faster convergence (fewer iterations).

This system will look like

$$M^{-1}Ax = M^{-1}b$$

Where the goal is to choose M such that κ($M^{-1}A$) << κ(A).

Preconditioning:

$M^{-1}Ax = M^{-1}b$ vs *Ax* = b

x is a solution to both problems, so we prefer to solve the "easier" one.

The matrix *M* is called a **preconditioner.**

Similar to splitting in stationary iterative methods, we want:

1. M ≈ A
2. M^{-1} is cheap to compute, or *My* = *c* is cheap to solve.

Symmetric Preconditioning

If A is SPD, as desired for CG, we also want our modified system to be SPD. (Note that M and A being SPD does not imply $M^{-1}A$).

Let M be SPD. Then it has a Cholesky factorization M = LL^T with *L* lower triangular.

Form the modified system as:

$$\underbrace{L^{-1}AL^{-T}}_{\tilde{A}}\underbrace{L^Tx}_{\tilde{x}} = \underbrace{L^{-1}b}_{\tilde{b}}$$

Notice $\tilde{A}$ is symmetric positive definite by construction.
The preconditioner is effectively *split* into left and right parts.

Preconditioning CG

- Naïve approach: form modified system, apply standard CG, transform solution to recover x.
- Better approach: do not form $\tilde{A}$ explicitly. Just modify CG algorithm to include a preconditioning step.

Preconditioned CG Algorithm

$x^0 = \text{initial guess}, r^0 = b - Ax^0$
for k = 0,1,2…n-1
$z^k = M^{-1}r^k$ (or preferably solve $Mz^k = r^k$)
$\beta^k = (z^k, r^k)/(z^{k-1}, r^{k-1})$
$p^k = z^k + \beta^k p^{k-1}$
$\alpha^k = (z^k, r^k)/(p^k, Ap^k)$
$x^{k+1} = x^k + \alpha^k p^k$
$r^{k+1} = r^k - \alpha^k Ap^k$
end

Essentially one extra line here:
For $M = I$, we recover basic CG.

Common Preconditioners

- Jacobi preconditioning: *M* = *D*
- Symmetric Gauss-Seidel/SOR: $M = (D - L)D^{-1}(D - L)^T$ (SOR for ω = 1 case)
- "Incomplete" Cholesky preconditioning:
  - Try to find $LL^T \approx A$, approximately
  - Proceed like Cholesky factorization, but skip most or all steps that would introduce fill
  - The sparsity pattern of *L* stays close to *A*'s for reduced cost.

Matlab

- MATLAB's CG routine is **pcg** for preconditioned conjugate gradient
- Accepts preconditioner(s) as extra arguments
- Incomplete Cholesky preconditioning is supported via ichol

e.g. 2D Laplacian for *m* = 14 case.
$\kappa(A) \approx 90.5, \lambda_{max} \approx 1780, \lambda_{min} \approx 19.7$, 23 CG iterations for *tol* = 10^{-7}
$\kappa(\tilde{A}) \approx 8.85, \lambda_{max} \approx 1.2, \lambda_{min} \approx 0.135$, 14 PCG iterations for *tol* = 10^{-7}