Lecture 10

Sparse Matrices, Iterative Methods

T. Gambill

Department of Computer Science
University of Illinois at Urbana-Champaign

March 15, 2011

An application

Latent semantic analysis (LSA) (also called LSI - Latent semantic indexing for information retrieval) analyzes two-mode data. Looks at relationships between documents and terms.

- natural language processing
- information retrieval
- information filtering
- textual machine learning

Document-term matrix:
Document1(d_1) = ”I love numerical analysis”
Document1(d_2) = ”I do not love numerical analysis, but I love linear algebra.”

	I	love	numerical	linear	algebra
d_1	1	1	1	0	0
d_2	2	2	1	1	1

An application

	I	love	numerical	linear	algebra
d_1	1	1	1	0	0
d_2	2	2	1	1	1

One method for weights: Term Count Model
Variation: Term Frequency-Inverse Document Frequency; weight the entries inversely, highlighting infrequent terms

Let X be the matrix of occurrences (or the inverse).

$$X = \begin{bmatrix} d_1 & \mid & d_2 & \mid & \dots & \mid & d_n \end{bmatrix} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_n \end{bmatrix}$$

Now each row t_i will be a vector relating a term to all documents. Each column d_i will be a vector relating a document to all terms.

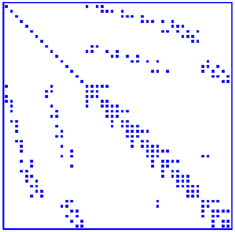
An application

$$X = \begin{bmatrix} x_{1,1} & \dots & x_{1,n} \\ \vdots & \ddots & \vdots \\ x_{m,1} & \dots & x_{m,n} \end{bmatrix}$$

- In general X has many zeros
- a dot product of the rows $t_i t_j^T$ gives the correlation between terms over the documents
- XX^T gives a cumulative view of the correlation
- same with $X^T X$
- singular value decompositions, eigenvalue analysis, etc give other information

Sparse Matrices

ack: Y. Saad



- Vague definition: matrix with few nonzero entries
- For all practical purposes: an $m \times n$ matrix is sparse if it has $\mathcal{O}(\min(m, n))$ nonzero entries.
- This means roughly a constant number of nonzero entries per row and column

Sparse Matrices

ack: Y. Saad

- Other definitions use a slow growth of nonzero entries with respect to n or m.
- Wilkinson's Definition: "..matrices that allow special techniques to take advantage of the large number of zero elements." (J. Wilkinson)"
- A few applications which lead to sparse matrices: Structural Engineering, Computational Fluid Dynamics, Reservoir simulation, Electrical Networks, optimization, data analysis, information retrieval (LSI), circuit simulation, device simulation, ...

Sparse Matrices: The Goal

- To perform standard matrix computations economically i.e., without storing the zeros of the matrix.
- For typical Finite Element /Finite difference matrices, number of nonzero elements is $\mathcal{O}(n)$.

Example

To add two square dense matrices of size n requires $\mathcal{O}(n^2)$ operations. To add two sparse matrices A and B requires $\mathcal{O}(nnz(A) + nnz(B))$ where $nnz(X) =$ number of nonzero elements of a matrix X.

remark

A^{-1} is usually dense, but L and U in the LU factorization may be reasonably sparse (if a good technique is used).

Iterative solution of $Ax = b$

- Principle goal: *solve*
$$Ax = b$$
where $A \in \mathbb{R}^{n\times n}$, $x, b \in \mathbb{R}^n$
- Assumption: A is very sparse
- General approach: iteratively improve the solution
- Given x_0, ultimate "correction" is
$$x_1 = x_0 + e_0$$
where $e_0 = x - x_0$, thus
$$\begin{aligned} Ae_0 &= Ax - Ax_0 \\ e_0 &= A^{-1}(Ax - Ax_0) \\ x_1 &= x_0 + e_0 = x_0 + A^{-1}(Ax - Ax_0) = x_0 + A^{-1}r_0 \end{aligned}$$
since $r_0 = b - Ax_0$.

Goal

- Principle difficulty: how do we “approximate” $A^{-1}r$ or reformulate the iteration?
- One simple idea:

$$x_1 = x_0 + \widehat{A^{-1}} r_0 \text{ where } \widehat{A^{-1}} \text{ is an approximation to } A^{-1}$$

- operation is inexpensive if r_0 is inexpensive
- requires very fast sparse mat-vec (matrix-vector multiply) Ax_0

Sparse Matrices

- So how do we store A?
- Fast mat-vec is certainly important; also ask
  - what type of access (rows, cols, diag, etc)?
  - dynamic allocation?
  - transpose needed?
  - inherent structure?
- Unlike dense methods, not a lot of standards for iterative
  - dense BLAS have been long accepted
  - sparse BLAS still iterating
- Even data structures for dense storage not as obvious
- Sparse operations have low operation/memory reference ratio

Sparse Matrix Qualification

Matrix Market attempts to classify the sparse matrix.

Matrix Market

`http://math.nist.gov/MatrixMarket/`

First Qualification (type of values and number of values):

identifier	description
Real	All entries are float
Complex	All entries are a pair of float
Integer	All entries are int
Pattern	Matrix is a pattern. Actual entries are omitted
Parallel	*Parallel structure is identified*

Sparse Matrix Qualification

Second Qualification (interpreting values):

identifier	description
General	A has no symmetry, no symmetry is utilized, or A is not square
Symmetric	$a_{ij} = a_{ji}$; only entries on the diagonal and below(or above) are stored.
Skew-Symmetric	$a_{ij} = -a_{ji}$; only entries below (or above) the diagonal ($= 0$) are stored.
Hermitian	$a_{ij} = \bar{a}_{ji}$; only entries on the diagonal and below (or above) are stored.

see “The Matrix Market Exchange Formats: Initial Design” by Boisvert, Pozo, Remington

Popular Storage Structures

DNS	Dense	**ELL**	Ellpack-Itpack
BND	Linpack Banded	**DIA**	Diagonal
COO	Coordinate	**BSR**	Block Sparse Row
CSR	Compressed Sparse Row	**SSK**	Symmetric Skyline
CSC	Compressed Sparse Column	**BSR**	Nonsymmetric Skyline
MSR	Modified CSR	**JAD**	Jagged Diagonal
LIL	Linked List		

note: CSR = CRS, CCS = CSC, SSK = SKS in some references

Matlab :: CSC

John R. Gilbert, Cleve, Moler and Robert Schreiber, Sparse Matrices in MATLAB: Design and Implementation, SIAM Journal on Matrix Analysis and Applications, volume 13, number 1, pages 333–356 (1992).

DNS (Dense)

$$A = \begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \\ 7.0 & 8.0 & 9.0 \end{bmatrix}$$

$$AA = \begin{bmatrix} 3 & 3 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 8.0 & 9.0 \end{bmatrix}$$

- simple
- row-wise
- easy blocked formats

COO (Coordinate)

$$A = \begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix}$$

$$\begin{array}{lcllllllllllllll} AA & = & [& 12.0 & 9.0 & 7.0 & 5.0 & 1.0 & 2.0 & 11.0 & 3.0 & 6.0 & 4.0 & 8.0 & 10.0 &] \\ JR & = & [& 5 & 3 & 3 & 2 & 1 & 1 & 4 & 2 & 3 & 2 & 3 & 4 &] \\ JC & = & [& 5 & 5 & 3 & 4 & 1 & 4 & 4 & 1 & 1 & 2 & 4 & 3 &] \end{array}$$

- simple, often used for entry

Question: Do you need this much storage?

CSR (Compressed Sparse Row)

$$A = \begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix}$$

$$\begin{array}{lclcccccccccccc} AA & = & [& 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 8.0 & 9.0 & 10.0 & 11.0 & 12.0 \;] \\ JA & = & [& 1 & 4 & 1 & 2 & 4 & 1 & 3 & 4 & 5 & 3 & 4 & 5 \;] \\ IA & = & [& 1 & 3 & 6 & 10 & 12 & 13 &] \end{array}$$

- Length of AA and JA is nnz; length of IA is $n+1$
- $IA(j)$ gives the index (offset) to the beginning of row j in AA and JA (one origin due to Fortran)
- no structure, fast row access, slow column access (why?)
- related: CSC, MSR

MSR (Modified CSR)

$$A = \begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix}$$

$$\begin{array}{lllllllllllll} AA = & [1.0 & 4.0 & 7.0 & 11.0 & 12.0 & * & 2.0 & 3.0 & 5.0 & 6.0 & 8.0 & 9.0 & 10.0] \\ JA = & [7 & 8 & 10 & 13 & 14 & 14 & 4 & 1 & 4 & 1 & 4 & 5 & 3] \end{array}$$

- places importance on diagonal (often nonzero and accessed frequently)
- first n entries are the diag
- $n+1$ is empty
- rest of AA are the nondiagonal entries
- first $n+1$ entries in JA give the index (offset) of the beginning of the row (the IA of CSR is in this JA)
- rest of JA are the columns indices

DIA (Diagonal)

or CDS

$$A = \begin{bmatrix} 1 & 0 & 2 & 0 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 0 & 6 & 7 & 0 & 8 \\ 0 & 0 & 9 & 10 & 0 \\ 0 & 0 & 0 & 11 & 12 \end{bmatrix} \quad DIAG = \begin{bmatrix} * & 1.0 & 2.0 \\ 3.0 & 4.0 & 5.0 \\ 6.0 & 7.0 & 8.0 \\ 9.0 & 10.0 & * \\ 11.0 & 12.0 & * \end{bmatrix} \quad IOFF = \begin{bmatrix} -1 & 0 & 2 \end{bmatrix}$$

- need to know the offset structure
- some entries will always be empty

ELL (Ellpack-Itpack)

$$A = \begin{bmatrix} 1 & 0 & 2 & 0 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 0 & 6 & 7 & 0 & 8 \\ 0 & 0 & 9 & 10 & 0 \\ 0 & 0 & 0 & 11 & 12 \end{bmatrix} \quad COEF = \begin{bmatrix} 1.0 & 2.0 & 0.0 \\ 3.0 & 4.0 & 5.0 \\ 6.0 & 7.0 & 8.0 \\ 9.0 & 10.0 & 0.0 \\ 11.0 & 12.0 & 0.0 \end{bmatrix} \quad JCOEF = \begin{bmatrix} 1 & 3 & 1 \\ 1 & 2 & 4 \\ 2 & 3 & 5 \\ 3 & 4 & 4 \\ 4 & 5 & 5 \end{bmatrix}$$

- Form columns from first non-zero in each row, repeat.
- used more on vector machines (what? why?)
- assumes low number of nnz per row (=number of columns in $COEFF$ and $JCOEFF$)

Blocked

$$A = \begin{bmatrix} 1.0 & 2.0 & 0.0 & 0.0 & 3.0 & 4.0 \\ 5.0 & 6.0 & 0.0 & 0.0 & 7.0 & 8.0 \\ 0.0 & 0.0 & 9.0 & 10.0 & 11.0 & 12.0 \\ 0.0 & 0.0 & 13.0 & 14.0 & 15.0 & 16.0 \\ 17.0 & 18.0 & 0.0 & 0.0 & 20.0 & 21.0 \\ 22.0 & 23.0 & 0.0 & 0.0 & 24.0 & 25.0 \end{bmatrix}$$

$$AA = \begin{bmatrix} 1.0 & 3.0 & 9.0 & 11.0 & 17.0 & 20.0 \\ 5.0 & 7.0 & 13.0 & 15.0 & 22.0 & 24.0 \\ 2.0 & 4.0 & 10.0 & 12.0 & 18.0 & 21.0 \\ 6.0 & 8.0 & 14.0 & 16.0 & 23.0 & 25.0 \end{bmatrix}$$

$$JA = \begin{bmatrix} 1 & 5 & 3 & 5 & 1 & 5 \end{bmatrix}$$
$$IA = \begin{bmatrix} 1 & 3 & 5 & 7 \end{bmatrix}$$

Blocked

- each column of AA is a 2×2 block
- $JA(k) =$ column index of $(1, 1)$ entries of the kth block
- declared as $AA(2, 2, 6)$
- blocks arise in *many* apps
- variant: variable block size

Blocked

Also row-wise

$$AA = \begin{bmatrix} 1.0 & 5.0 & 2.0 & 6.0 \\ 3.0 & 7.0 & 4.0 & 8.0 \\ 9.0 & 15.0 & 10.0 & 14.0 \\ 11.0 & 13.0 & 12.0 & 16.0 \\ 17.0 & 22.0 & 18.0 & 23.0 \\ 20.0 & 24.0 & 21.0 & 25.0 \end{bmatrix}$$

$$JA = \begin{bmatrix} 1 & 5 & 3 & 5 & 1 & 5 \end{bmatrix}$$
$$IA = \begin{bmatrix} 1 & 3 & 5 & 7 \end{bmatrix}$$

- each row of AA is a 2×2 block (can be a drawback)
- JA, IA same, $AA(6, 2, 2)$
- if elements of blocks are accessed at the same time: rows are better (C)
- if elements of similar positions in different blocks are accessed at the same time: columns are better (C)

try it...

$$A = \begin{bmatrix} 7 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 6 & 4 \end{bmatrix}$$

- CSR
- CSC
- COO

Example

$$A = \begin{bmatrix} 7 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 6 & 4 \end{bmatrix}$$

i	IA	JA	AA
1	2	2	1
2	3	4	2
3	4	5	5
4	2	3	2
5	5	6	4
6	1	1	7
7	5	5	6
8	3	2	2

COO

i	IA	JA	AA
1	1	1	7
2	2	2	1
3	4	3	2
4	6	2	2
5	7	4	2
6	9	5	5
7	-	5	6
8	-	6	4

CSR

Sparse Matrix-Vector Multiply

$z = Ax$, $A_{m\times n}$, $x_{n\times 1}$, $z_{m\times 1}$

```
input $A$, $x$

 for $i = 1$ to $m$
      $z(i) = A(i,:) * x$
  end
```

- CSR: rows are contiguous...(next slide)

Sparse Matrix-Vector Multiply

CSR

$z = Ax,\ A_{m\times n}, AA_{1\times nnz(A)},\ x_{n\times 1},\ z_{m\times 1}$

```
for i=1:m
  Z(i)=0
  K1 = IA(i)
  K2 = IA(i+1)-1
  for j=K1:K2
    z(i) = z(i) + AA(j)*x(JA(j))
  end
end
```

- $\mathcal{O}(nnz)$arithmetic operations
- marches down the rows
- very cheap

Some Python

$$A = \begin{bmatrix} 7 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 6 & 4 \end{bmatrix}$$

i	IA	JA	AA
1	2	2	1
2	3	4	2
3	4	5	5
4	2	3	2
5	5	6	4
6	1	1	7
7	5	5	6
8	3	2	2

COO

```
>>> import scipy
>>> import numpy as np
>> I = np.array([1.,2.,3.,1.,4.,0.,4.,2.])
>>> J = np.array([1.,3.,4.,2.,5.,0.,4.,1.])
>>> V =scipy.sparse.coo_matrix((A,(I,J)),shape=(5,6))
>>> V
<5x6 sparse matrix of type '<type numpy.float64>'
with 8 stored elements in COOrdinate format>
>>> V.todense()
matrix([[ 7.,   0.,   0.,   0.,   0.,   0.],
        [ 0.,   1.,   2.,   0.,   0.,   0.],
        [ 0.,   2.,   0.,   2.,   0.,   0.],
        [ 0.,   0.,   0.,   0.,   5.,   0.],
        [ 0.,   0.,   0.,   0.,   6.,   4.]])
```

Some Python

From COO to CSC:

```
>>> V =scipy.sparse.coo_matrix((A,(I,J)),shape=(5,6)).tocsr()
>>> V
<5x6 sparse matrix of type '<type numpy.float64>'
with 8 stored elements in Compressed Sparse Row format>
```

To view:

```
>>> V =scipy.sparse.coo_matrix((A,(I,J)),shape=(5,6)).tocsr()
>>> matplotlib.pylab.spy(V)
<matplotlib.lines.Line2D object at 0x1eba2d0>
>>> matplotlib.pylab.show()
```

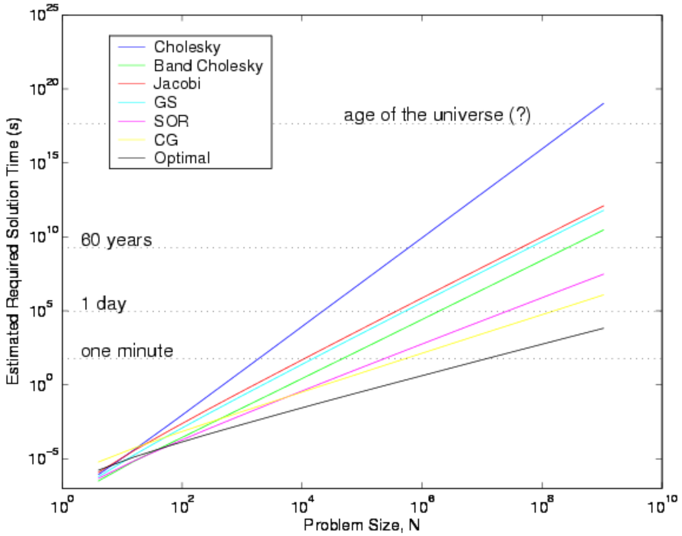
Simple Matrix Iterations

- Solve
$$Ax = b$$
- Assumption: A is very sparse
- Let $A = N + M$, then
$$\begin{aligned} Ax &= b \\ (N+M)x &= b \\ Nx &= b - Mx \end{aligned}$$
- Make this into an iteration:
$$\begin{aligned} Nx_k &= b - Mx_{k-1} \\ x_k &= N^{-1}(b - Mx_{k-1}) \end{aligned}$$
- Careful choice of N and M can give effective methods
- More powerful iterative methods exist

Summary: Complexity of Linear Solves

- $Ax = b$
- diagonal system: $\mathcal{O}(n)$
- upper or lower triangular system: $\mathcal{O}(n^2)$
- full system with GE: $\mathcal{O}(n^3)$
- partial pivoting adds $\mathcal{O}(n^2)$
- full system with LU: $\mathcal{O}(n^3)$
- LU back solve: $\mathcal{O}(n^2)$
- m different right-hand sides: $\mathcal{O}(mn^3)$ for GE or $\mathcal{O}(n^3 + mn^2)$ for LU
- tridiagonal system: $\mathcal{O}(n)$
- m-band system: $\mathcal{O}(m^2 n)$

Summary: Complexity



Approximate solutions

So far, we are seeking “exact” solutions x^* to

$$Ax = b$$

What if we only need an approximations $\hat{x}$ to x^*?

We would like some $\hat{x}$ so that $\|\hat{x} - x^*\| \leqslant \epsilon$, where ϵ is some tolerance.

The Residual

We can't actually evaluate

$$e = x^* - \hat{x}$$

We call $r = b - A\hat{x}$ the *residual*. It is way to measure the error. In fact

$$\begin{aligned} r &= b - A\hat{x} \\ &= Ax^* - A\hat{x} \\ &= Ae \end{aligned}$$

Residual versus Error

r = Ae

How big is the residual?

For a given approximation, $\hat{x}$ to x, how "big" is the residual $r = b - A\hat{x}$?

- $\|r\|$ gives a magnitude
- $\|r\|_1 = \sum_{j=1}^{n} |r_i|$
- $\|r\|_2 = \left(\sum_{j=1}^{n} r_i^2\right)^{1/2}$
- $\|r\|_\infty = \max_{1 \leqslant j \leqslant n} |r_i|$

Approximating x...

Suppose we made a wild guess to the solution x of $Ax = b$:

$$x^{(0)} \approx x$$

How do I improve $x^{(0)}$?

Ideally:

$$x^{(1)} = x^{(0)} + e^{(0)}$$

but to obtain $e^{(0)}$, we must know x. Not a viable method.

Ideally (another way):

$$\begin{aligned} x^{(1)} &= x^{(0)} + e^{(0)} \\ &= x^{(0)} + (x^* - x^{(0)}) \\ &= x^{(0)} + (A^{-1}b - x^{(0)}) \\ &= x^{(0)} + A^{-1}(b - Ax^{(0)}) \\ &= x^{(0)} + A^{-1}r^{(0)} \end{aligned}$$

An iteration

Again, the method

$$x^{(1)} = x^{(0)} + A^{-1} r^{(0)}$$

is nonsense since if we knew A^{-1} then we could compute the solution $A^{-1}b$.

What if we approximate A^{-1}? Suppose $Q^{-1} \approx A^{-1}$ and is cheap to construct, then

$$x^{(1)} = x^{(0)} + Q^{-1} r^{(0)}$$

is a good step.

continuing...

$$x^{(k)} = x^{(k-1)} + Q^{-1} r^{(k-1)}$$

An iteration, we've seen before...

The iterative formula we derived on the previous slide,

$$x^{(k)} = x^{(k-1)} + Q^{-1} r^{(k-1)}$$

is actually just the iteration we mentioned earlier,

$$x^{(k)} = N^{-1}(b - Mx^{(k-1)})$$

where $Q = N$ and $A = M + N$. To see this note that,

$$\begin{aligned}
x^{(k)} &= x^{(k-1)} + Q^{-1} r^{(k-1)} && (1)\\
&= x^{(k-1)} + N^{-1}(b - Ax^{(k-1)}) && (2)\\
&= x^{(k-1)} + N^{-1}(b - (M+N)x^{(k-1)}) && (3)\\
&= x^{(k-1)} + N^{-1}b - N^{-1}Mx^{(k-1)} - x^{(k-1)} && (4)\\
&= N^{-1}(b - Mx^{(k-1)}) && (5)\\
& && (6)
\end{aligned}$$

Two views of the solution

One form of the solution for $x^{(k)}$ is (remember Newton's method in higher dimensions?):

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

and a second form is:

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

In either form we do not compute Q^{-1} rather, where we solve the linear system.

Two Popular Choices

Example

Jacobi iteration approximates A with $Q = D = diag(A)$ where D has no zero values.

```
$x = x^{(0)}$

Q = D

for $k = 1$ to $k_{max}$
    $r = b - Ax$
    if $\|r = b - Ax\| \leqslant tol$, stop

    $x = x + Q^{-1}r$
end
```

Two Popular Choices

Example

Gauss-Seidel iteration approximates $A = D + L + U$ where L and U have zero as diagonal values. Choose $Q = D + L$.

```
$x = x^{(0)}$

Q = D + L

for $k = 1$ to $k_{max}$
    $r = b - Ax$
    if $\|r = b - Ax\| \leqslant tol$, stop

    $x = x + Q^{-1} r$
end
```

Why D and $D+L$?

Look again at the iteration

$$x^{(k)} = x^{(k-1)} + Q^{-1} r^{(k-1)}$$

Looking at the error:

$$x - x^{(k)} = x - x^{(k-1)} - Q^{-1} r^{(k-1)}$$

Gives

$$e^{(k)} = e^{(k-1)} - Q^{-1} A e^{(k-1)}$$

or

$$e^{(k)} = (I - Q^{-1}A) e^{(k-1)}$$

or

$$e^{(k)} = (I - Q^{-1}A)^k e^{(0)}$$

Why D and $D + L$?

We want

$$e^{(k)} = (I - Q^{-1}A)^k e^{(0)}$$

to converge.

When does $a_k = c^k$ converge? .....when $|c| < 1$

Likewise, our iteration converges

$$\begin{aligned} \|e^{(k)}\| &= \|(I - Q^{-1}A)^k e^{(0)}\| \\ &\leqslant \|I - Q^{-1}A\|^k \|e^{(0)}\| \end{aligned}$$

when $\|I - Q^{-1}A\| < 1$.

Matrix Norms

What is $\|I - Q^{-1}A\|$?

- $\|A\|_1 = \max_{1 \leqslant j \leqslant n} \sum_{i=1}^{n} |a_{ij}|$
- $\|A\|_2 = \sigma_{max} = \sigma_1$(the largest singular value of A)
- $\|A\|_\infty = \max_{1 \leqslant i \leqslant n} \sum_{j=1}^{n} |a_{ij}|$

Again, why do Jacobi and Gauss-Seidel work?

Jacobi, Gauss-Seidel (sufficient) Convergence Theorem

If A is diagonally dominant by rows, then the Jacobi and Gauss-Seidel methods converge for any initial guess $x^{(0)}$.

Definition: Diagonal Dominance

A matrix is *diagonally dominant by rows* if

$$|a_{ii}| > \sum_{j=1, j\neq i}^{n} |a_{ij}|$$

for all i.

Smart Jacobi Algorithm

The algorithm above uses the matrix representation:

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

The diagonal is decoupled from the $L+U$, so we have an update in the form of

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

So each sweep (from $k-1$ to k) uses $\mathcal{O}(n)$ operations per vector element.
If, for each row i, $a_{ij} = 0$ for all but m values of j, each sweep uses $\mathcal{O}(mn)$ operations.

Smart Gauss-Seidel Algorithm

The algorithm above uses the matrix representation:

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

Component-wise:

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

So again each sweep (from $k-1$ to k) uses $\mathcal{O}(n)$ operations per vector element.
If, for each row i, $a_{ij} = 0$ for all but m values of j, each sweep uses $\mathcal{O}(mn)$ operations.

In our iterative methods we would expect that $x^{(k)}$ to be closer to the solution than $x^{(k-1)}$. Note that the Gauss-Seidel method includes the previous values of $x_j^{(k)}$ in computing $x_i^{(k)}$ where $i > j$ whereas the Jacobi method does not. So we would expect the Gauss-Seidel method to converge faster than the Jacobi method.

Conjugate Gradients

- Suppose that A is $n \times n$ symmetric and positive definite.
- Since A is positive definite, $x^T A x > 0$ for all $x (\neq 0) \in \mathbb{R}^n$. (Why?)
- Define a quadratic function

$$\phi(x) = \frac{1}{2} x^T A x - x^T b$$

- It turns out that $-\nabla \phi = b - Ax = r$, or $\phi(x)$ has a minimum for x such that $Ax = b$.
- Optimization methods look in a "search direction" and pick the best step:

$$x_{k+1} = x_k + \alpha s_k$$

  Choose α so that $\phi(x_k + \alpha s_k)$ is minimized in the direction of s_k.
- Find α so that ϕ is minimized:

$$0 = \frac{d}{d\alpha}\phi(x_{k+1}) = \nabla\phi(x_{k+1})^T \frac{d}{d\alpha} x_{k+1} = -r_{k+1}^T \frac{d}{d\alpha}(x_k + \alpha s_k) = -r_{k+1}^T s_k.$$

Conjugate Gradients

- Find α so that ϕ is minimized:

$$0 = \frac{d}{d\alpha}\phi(x_{k+1}) = \nabla\phi(x_{k+1})^T \frac{d}{d\alpha} x_{k+1} = -r_{k+1}^T \frac{d}{d\alpha}(x_k + \alpha s_k) = -r_{k+1}^T s_k.$$

- We also know

$$r_{k+1} = b - Ax_{k+1} = b - A(x_k + \alpha s_k) = r_k - \alpha A s_k$$

- So, the optimal search parameter is

$$\alpha = \frac{r_k^T s_k}{s_k^T A s_k}$$

- This is CG: take a step in a search direction

Conjugate Gradients

- Neat trick: We can compute the r_{k+1} without explicitly forming $b - Ax_{k+1}$. Note from the previous slide:

$$r_{k+1} = b - Ax_{k+1} = b - A(x_k + \alpha s_k) = b - Ax_k - \alpha As_k = r_k - \alpha As_k$$

  and computing α (on the previous slide) already involves computing As_k.

Conjugate Gradients

- How should we pick s_k?
- Note that $-\nabla \phi = b - Ax = r$, so r is the negative gradient of ϕ (for *any* x), and this is a good direction.
- Thus, pick $s_0 = r = b - Ax_0$.
- What is s_1? This should be in the direction of r_1, but *conjugate* to s_0: $s_1^T A s_0 = 0$.
- (Two vectors u and v are A-*conjugate* if $u^T A v = 0$)
- So, if we let $s_1 = r_1 + \beta_1 s_0$, we can require

$$0 = s_1^T A s_0 = (r_1^T + \beta_1 s_0^T) A s_0 = r_1^T A s_0 + \beta_1 s_0^T A s_0$$

  or

$$\beta_1 = -r_1^T A s_0 / s_0^T A s_0.$$

- Holds for s_{k+1} in terms of $r_{k+1} + \beta_k s_k$
- Further simplification (which is *not* simple to carry out) yields a simple method that requires only one matrix-vector product per step:

Conjugate Gradients

```
$x_0 =$ initial guess $r_0 = b - Ax_0$  $s_0 = r_0$
 for $k = 0, 1, 2, \ldots$
    $\alpha_k = \frac{r_k^T r_k}{s_k^T A s_k}$
    $x_{k+1} = x_k + \alpha_k s_k$
    $r_{k+1} = r_k - \alpha_k A s_k$
    $\beta_{k+1} = r_{k+1}^T r_{k+1} / r_k^T r_k$
    $s_{k+1} = r_{k+1} + \beta_{k+1} s_k$
 end
```

Goal

- Find $x = Ax$ and the elements of x are Google's PageRank.
- For a matrix A, the scalar-vector pairs (λ, v) such that $Av = \lambda v$ are eigenvalue-eigenvectors.
- Power Method

Power Method

Suppose that A is $n \times n$ and that A is similar to a diagonal matrix D, that is,

$$D = S^{-1}AS$$

and further that the eigenvalues of A are ordered:

$$|\lambda_1| > |\lambda_2| \geqslant |\lambda_3| \geqslant \cdots \geqslant |\lambda_n|$$

The column vectors of S form a linearly independent set of of vectors Se_i where e_i is a standard unit vector in $\mathbb{R}$ such that $ASe_i = d_iSe_i$. That is, $\lambda_i = d_i$ and the columns s_i of S are the eigenvectors of A.

Goal

Computing the value of the largest (in magnitude) eigenvalue, λ_1.

Power Method

Take a guess $x^{(0)}$ at the associated eigenvector, $Ax = \lambda_1 x$. We know

$$x^{(0)} = c_1 s_1 + \cdots + c_n s_n$$

Since $c_j s_j$ is an eigenvector of A with eigenvalue λ_j (why?) rename the eigenvectors $v_j = c_j s_j$.

$$x^{(0)} = v_1 + \cdots + v_n$$

Then compute

$$\begin{aligned} x^{(1)} &= Ax^{(0)} \\ x^{(2)} &= Ax^{(1)} \\ x^{(3)} &= Ax^{(2)} \\ &\vdots \\ x^{(k+1)} &= Ax^{(k)} \end{aligned}$$

Power Method

Or $x^{(k)} = A^k x^{(0)}$. Or

$$\begin{aligned} x^{(k)} &= A^k x^{(0)} \\ &= A^k v_1 + \cdots + A^k v_n \\ &= \lambda_1^k v_1 + \ldots \lambda_n^k v_n \end{aligned}$$

And this can be written as

$$x^{(k)} = \lambda_1^k \left(v_1 + \left(\frac{\lambda_2}{\lambda_1}\right)^k v_2 + \cdots + \left(\frac{\lambda_n}{\lambda_1}\right)^k v_n \right)$$

So as $k \to \infty$, we are left with

$$x^{(k)} \to \lambda^k v_1$$

The Power Method (with normalization)

```
for $k = 1$ to $kmax$
    $y = Ax$
    $r = \phi(y)/\phi(x)$
    $x = y/\|y\|_\infty$
```

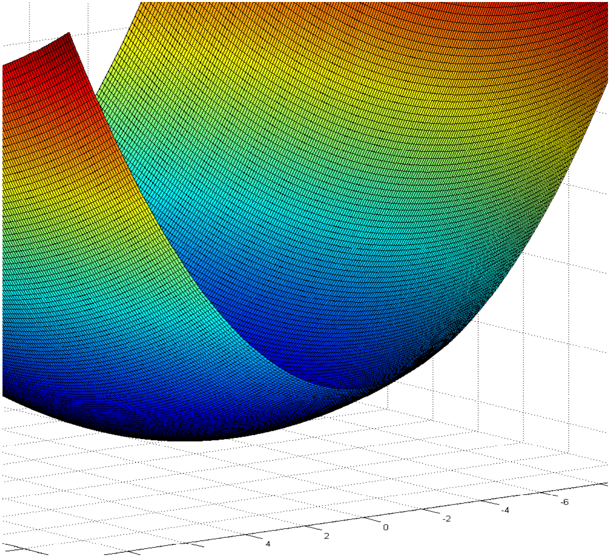
- often $\phi(x) = x_1$ is sufficient
- r is an estimate of the eigenvalue; x the eigenvector

Challenge: Why can't we use $r = \|y\|/\|x\|$?

Inverse Power Method

- We now want to find the smallest eigenvalue
- $Av = \lambda v \quad \Rightarrow \quad A^{-1}v = \frac{1}{\lambda}v$
- So "apply" power method to A^{-1} (assuming a distinct smallest eigenvalue)
- $x^{(k+1)} = A^{-1}x^{(k)}$
- Easier with $A = LU$
- Update RHS and backsolve with U:

$$Ux^{(k+1)} = L^{-1}x^{(k)}$$


-6
-4
-2
0
2
4

Instructor Notes

The quadratic function $q(x) = x^T Ax$ is called a quadratic form. If A is not symmetric then it can be converted to a new matrix B where $q(x) = x^T Ax = x^T Bx$ and B is symmetric. Define

$$b_{ij} = \begin{cases} a_{ij}, & \text{when } i = j \\ \frac{a_{ij}+a_{ji}}{2}, & \text{when } i \neq j \end{cases}$$