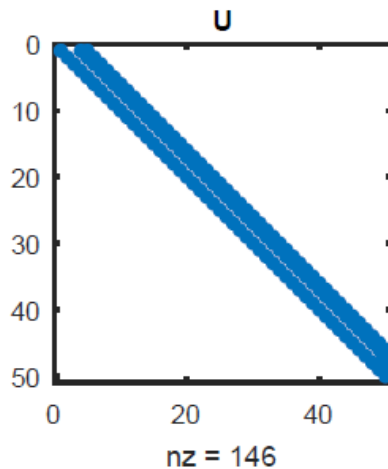
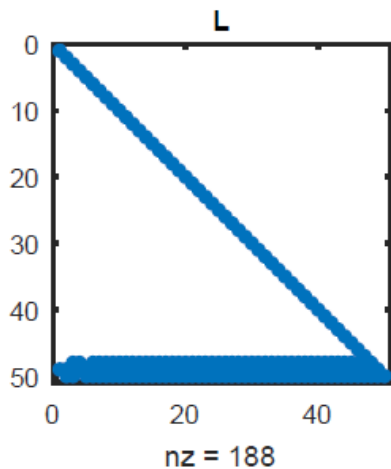


# Chapter 8

## Krylov Methods in Linear Algebra



# Section 8.1

## Sparsity and Structure

# Methods for large matrices/systems

- We could also have called this chapter “Iterative methods”
- Depending on your computer, you may find it is difficult to store an  $n \times n$  matrix where  $n = 10^4$
- We need a way to compute at all for larger matrices
- We need ways to compute efficiently for larger matrices

# Sparse matrices

- One common case is when large matrices has many *structural zeros*: many elements are exactly zero
- In these *sparse matrices*, the fraction of nonzero entries may be quite small
- One uses much less storage to store only nonzero entries, and to do computations only for nonzero values
- Each nonzero entry  $\{a_{ij}\}$  in the matrix  $\mathbf{A}$  can be stored as a triple  $(i, j, a_{ij})$
- One then has to write code that can correctly use only the nonzero entries; we will not do this
- But, we will exploit Matlab's capabilities in this area

## Example: large adjacency matrix

### Example 8.1.1

A is an adjacency matrix for web pages making about Roswell, NM, and how they link to each other.

```
load roswelladj    % get from the book's website
a = whos('A')
```

```
a =
    name: 'A'
    size: [2790 2790]
    bytes: 158120
    class: 'double'
    global: 0
    sparse: 1
    complex: 0
    nesting: [1x1 struct]
    persistent: 0
```

- It's large
- The elements are double precision
- It's sparse!
- Real valued

# Example: large adjacency matrix

## Example 8.1.1

Each nonzero entry in A corresponds to an edge; we can compute fraction of nonzeros: call that the density

```
sz = size(A); n = sz(1);  
density = nnz(A) / prod(sz)
```

```
density =  
    0.0011
```

```
F = full(A);  
f = whos('F');  
a.bytes/f.bytes
```

```
ans =  
    0.0025
```

`size` returns the number of rows and columns

`nnz` returns the number of nonzero elements

Sparse!

`full` creates the full matrix storage version

Compare storage of sparse vs full matrix versions

How about computation time?

## Example: large adjacency matrix

### Example 8.1.1

Computation time for sparse vs full? Sparse uses only nonzeros

```
x = randn(n,1);  
tic, for i = 1:200, A*x; end  
sparse_time = toc
```

```
sparse_time =  
    0.0107
```

```
tic, for i = 1:200, F*x; end  
dense_time = toc
```

```
dense_time =  
    0.6207
```

tic and toc for timing

Matrix vector products first

Create random vector,  
repeat 200x for timing

Sparse first, then full

60x slower for full!



## Example: large adjacency matrix

### Example 8.1.1

Computation time for row operations can be slower: Matlab's sparse storage is column oriented

```
v = A(:,1000);  
tic, for i = 1:n, A(:,i)=v; end  
column_time = toc  
r = v';  
tic, for i = 1:n, A(i,:)=r; end  
row_time = toc
```

Column oriented

Row oriented

```
column_time =  
    0.0079  
row_time =  
    0.0630
```

8x slower for rows

[Example 8.1.1]



## Matrix fill-in

- Arithmetic operations  $*, +, -, ^$  can exploit sparsity
- But, matrix operations can reduce the sparsity: *matrix fill-in*

### Example 8.1.2

Here is the buckyball adjacency matrix again.

```
[A,v] = bucky;
```

The number of vertex pairs on a soccer ball connected by a path of length  $k > 1$  grows with  $k$ , as can be seen here for  $k = 3$ .

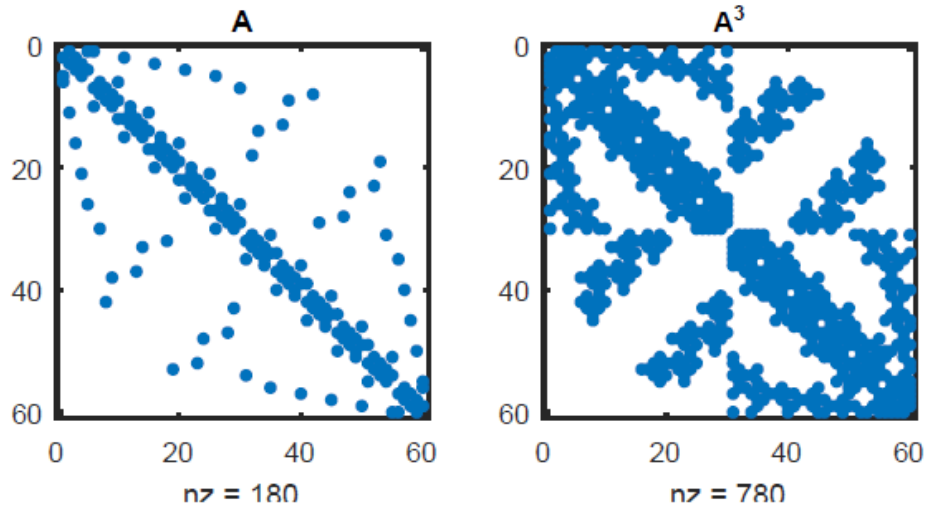
```
subplot(1,2,1), spy(A)  
subplot(1,2,2), spy(A^3)
```

The `spy` command plots nonzero elements and gives the number of nonzeros...

# Matrix fill-in

*[Example 8.1.2]*

Example 8.1.2

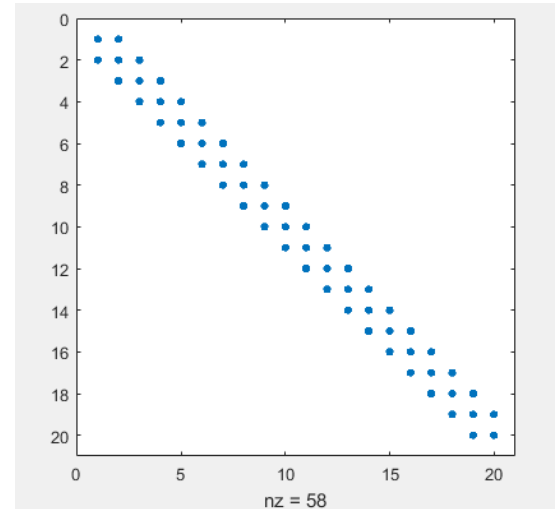


The number of nonzeros increased by more than 4x

# Banded matrices

- This is an important class of matrices that comes up in applications
- The simplest is the diagonal matrix: bandwidth 1
- Another is tridiagonal, with main diagonal, as well as one superdiagonal and one subdiagonal: bandwidth is 3
- More generally, if  $p$  nonzero superdiagonals, and  $q$  nonzero subdiagonals, then bandwidth is  $p+q-1$

```
>> n=20;  
>> v = ones(n,1);  
>> d = [-v 2*v -0.5*v];  
>> pos = [-1 0 1];  
>> A = spdiags(d,pos,n,n);  
>> spy(A)  
>> |
```



## Example: solving a banded system

### Example 8.1.3

```
n = 50;  
d = [ n*ones(n,1), ones(n,1), -(1:n)']; % diagonal entries  
pos = [-3 0 1]; % which diagonals  
A = spdiags(d,pos,n,n);  
full( A(1:7,1:7) )
```

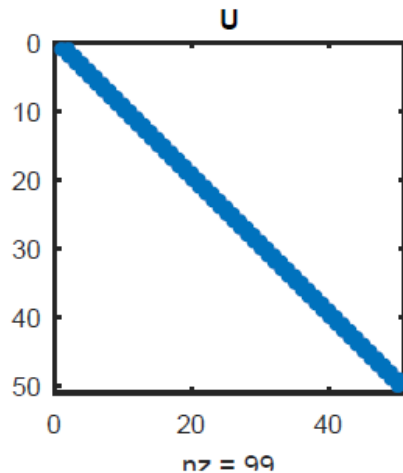
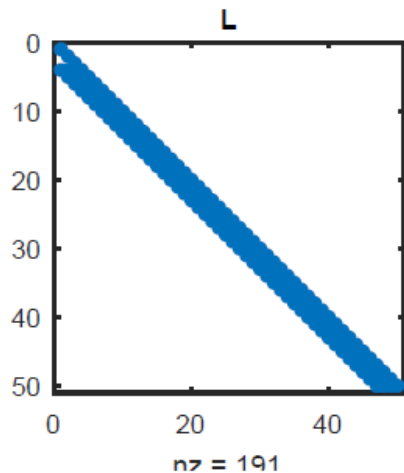
```
ans =  
    1    -2     0     0     0     0     0  
    0     1    -3     0     0     0     0  
    0     0     1    -4     0     0     0  
   50     0     0     1    -5     0     0  
    0    50     0     0     1    -6     0  
    0     0    50     0     0     1    -7  
    0     0     0    50     0     0     1
```

- Sparse tridiagonal matrix created
- Matlab trims the size of the vectors put into the diagonals
- Full version displayed
- Solving a system?

## Example: solving a banded system

### Example 8.1.3

```
[L,U] = lufact(A);  
subplot(1,2,1), spy(L), title('L')  
subplot(1,2,2), spy(U), title('U')
```



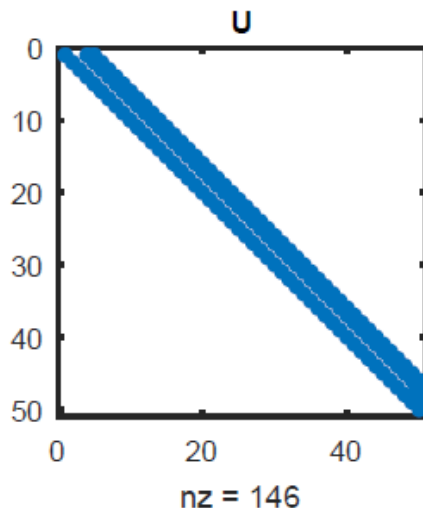
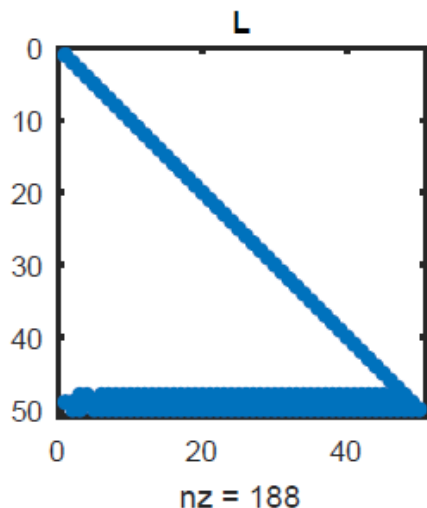
- No pivoting here
- Small bandwidth preserved

# Example: solving a banded system

## Example 8.1.3

```
[L,U,P] = lu(A);  
subplot(1,2,1), spy(L), title('L')  
subplot(1,2,2), spy(U), title('U')
```

*[Example 8.1.3]*



- Pivoting used here
- Bandwidth grows a little in U, but a lot in L to roughly  $n/2$

## Linear systems and eigenvalues

- Say one starts with a sparse matrix  $\mathbf{A}$  and compatible right-hand-side  $\mathbf{b}$
- In Matlab,  $\mathbf{A} \backslash \mathbf{b}$  will automatically try a sparse-aware form of Cholesky or pivoted LU factorization
- This approach could beat the  $O(n^3)$  cost for the general case with a full matrix, but this depends on the sparsity pattern of  $\mathbf{A}$
- For a very large  $\mathbf{A}$ , it is unlikely that one would compute all of the eigenvalues and eigenvectors of  $\mathbf{A}$ , that is, one would typically not use `eig` for very large matrices
- For very large  $\mathbf{A}$ , one would use `eigs` (Sec 8.4) to find the a selected number of eigenvalues, often the largest or those nearest a selected complex number (the `s` at the end indicates sparse; there is a similar `svds` command)

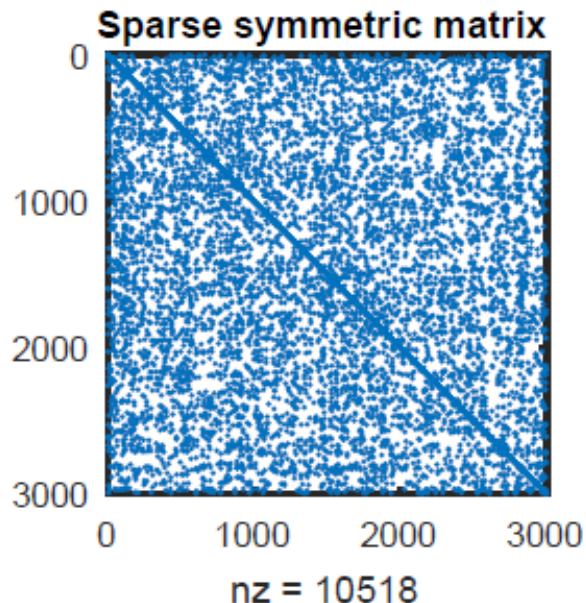
## Example: sparse matrix eigenvalues

### Example 8.1.4

The `sprandsym` command generates a random sparse matrix with prescribed eigenvalues.

```
n = 3000;  
density = 1.23e-3;  
lambda = 1./(1:n);  
A = sprandsym(n,density,lambda);  
spy(A)  
eigs(A,5)    % largest magnitude
```

```
ans =  
    1.0000  
    0.5000  
    0.3333  
    0.2500  
    0.2000
```





# Example: sparse matrix eigenvalues

## Example 8.1.4

```
eigs(A,5,0) % closest to zero
```

```
ans =  
 1.0e-03 *  
 0.3338  
 0.3337  
 0.3336  
 0.3334  
 0.3333
```

- Predicting order of FLOps is difficult without info re **A**
- But sparse calculation is often faster

```
x = 1./(1:n)'; b = A*x;  
tic, sparse_err = norm(x - A\b), sparse_time = toc
```

```
sparse_err =  
 2.1074e-14  
sparse_time =  
 0.0110
```

```
A = full(A);  
tic, dense_err = norm(x - A\b), dense_time = toc
```

```
dense_err =  
 7.2768e-14  
dense_time =  
 0.2792
```

## Section 8.2

### Power Iteration

## Example: repeated matrix multiplication

Let's use that fast matrix-vector multiplication

### Example 8.2.1

```
A = magic(5)/65;  
x = randn(5,1)
```

```
x =  
    1.7491  
    0.1326  
    0.3252  
   -0.7938  
    0.3149
```

```
y = A*x
```

```
y =  
    0.4864  
    0.5707  
    0.0473  
    0.1467  
    0.4770
```

```
z = A*y
```

```
z =  
    0.4668  
    0.3701  
    0.2987  
    0.2634  
    0.3291
```

- Nothing to good happening yet
- Now try more factors of **A**

# Example: repeated matrix multiplication

## Example 8.2.1

```
for j = 1:8, x = A*x; end  
[x,A*x]
```

```
ans =  
    0.3457    0.3457  
    0.3457    0.3456  
    0.3455    0.3456  
    0.3455    0.3456  
    0.3456    0.3456
```

- After 8 times,  
we are getting  
 $Ax \approx x$

- But this seems  
to happen lots  
of initial  
vectors

```
x = randn(5,1)  
for j = 1:8, x = A*x; end  
[x,A*x]
```

```
x =  
   -0.5273  
    0.9323  
    1.1647  
   -2.0457  
   -0.6444  
ans =  
   -0.2240   -0.2241  
   -0.2239   -0.2241  
   -0.2239   -0.2241  
   -0.2242   -0.2241  
   -0.2243   -0.2240
```

# Example: repeated matrix multiplication

## Example 8.2.1

- Using eig, we find that the eigenvalues are  $1, \pm 0.327, \pm 0.202$
- It turns out that we would be right to think that the process at left is converging to  $Ax = \lambda x$  with  $\lambda = 1$
- It turns out that this process converges to the largest eigenvalue and its associated eigenvalue
- Why?

```
x = randn(5,1)
for j = 1:8, x = A*x; end
[x,A*x]
```

```
x =
-0.5273
 0.9323
 1.1647
-2.0457
-0.6444
ans =
-0.2240 -0.2241
-0.2239 -0.2241
-0.2239 -0.2241
-0.2242 -0.2241
-0.2243 -0.2240
```

# Dominant eigenvalue

- Suppose that we have an  $n \times n$  diagonalizable matrix  $A$
- The eigenvalues are  $\lambda_1, \lambda_2, \dots, \lambda_n$  with associated eigenvectors  $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$

- Suppose also that

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \dots \geq |\lambda_n|.$$

- $\lambda_1$  is the dominant eigenvalue
- The  $\mathbf{v}_k$  are an LI set, so we can express the general initial vector as a linear combo of them:

$$\mathbf{x} = c_1 \mathbf{v}_1 + c_2 \mathbf{v}_2 + \dots + c_n \mathbf{v}_n.$$

- We know that  $A\mathbf{v}_k = \lambda_k \mathbf{v}_k$ ; use that with repeated mult by  $A$

# Dominant eigenvalue

- First,

$$\begin{aligned}Ax &= c_1 Av_1 + c_2 Av_2 + \cdots + c_n Av_n \\ &= c_1 \lambda_1 v_1 + c_2 \lambda_2 v_2 + \cdots + c_n \lambda_n v_n.\end{aligned}$$

- Now use this repeatedly, factor out dominant eigenvalue

$$\begin{aligned}A^k x &= \lambda_1^k c_1 v_1 + \lambda_2^k c_2 v_2 + \cdots + \lambda_n^k c_n v_n \\ &= \lambda_1^k \left[ c_1 v_1 + \left( \frac{\lambda_2}{\lambda_1} \right)^k c_2 v_2 + \cdots + \left( \frac{\lambda_n}{\lambda_1} \right)^k c_n v_n \right].\end{aligned}$$

- Does the left side approach  $c_1 v_1$ ? How fast?

# Dominant eigenvalue

- Rewrite the equation a bit: divide by  $\lambda_1^k$ , and subtract  $c_1 v_1$  from both sides to get

$$\frac{A^k x}{\lambda_1^k} - c_1 v_1 = \left(\frac{\lambda_2}{\lambda_1}\right)^k c_2 v_2 + \cdots + \left(\frac{\lambda_n}{\lambda_1}\right)^k c_n v_n$$

- Now take a norm of both sides, and use the triangle inequality:

$$\left\| \frac{A^k x}{\lambda_1^k} - c_1 v_1 \right\| \leq |c_2| \cdot \left| \frac{\lambda_2}{\lambda_1} \right|^k \|v_2\| + \cdots + |c_n| \cdot \left| \frac{\lambda_n}{\lambda_1} \right|^k \|v_n\| \rightarrow 0 \text{ as } k \rightarrow \infty.$$

- The right side tends to zero because the ratios of eigenvalues are all less than one:  $A^k x$  becomes almost parallel to dominant eigenvector
- We need  $c_1 \neq 0$ , essentially guaranteed with random initial vector
- Important to think of this as repeated matrix vector multiplication



# Power iteration

- We do repeated matrix-vector multiplication
- To make a more practical method, we also renormalize the vector with it's largest element each iteration
- Let  $|y_{k,m}| = ||y_k||_\infty$  ( $m$  is  $m$ th component)
- Note  $||x_{k+1}||_\infty = 1$

Power iteration algorithm:

1. Choose  $x_1$ .
2. For  $k = 1, 2, \dots$

$$y_k = Ax_k,$$

$$\alpha_k = \frac{1}{y_{k,m}}, \text{ where } |y_{k,m}| = \|y_k\|_\infty,$$

$$x_{k+1} = \alpha_k y_k.$$

# Dominant eigenvalue

- What about the eigenvalue?
- $k$ th approximation is

$$\gamma_k = \frac{y_{k,m}}{x_{k,m}} = \lambda_1 \frac{1 + r_2^{k+1}b_2 + \cdots + r_n^{k+1}b_n}{1 + r_2^k b_2 + \cdots + r_n^k b_n},$$

where  $r_j = \lambda_j/\lambda_1$  and the  $b_j$  are constants.

- With our ordering,  $r_j < 1$ , and we have convergence to the dominant eigenvalue as  $k \rightarrow \infty$
- Function `poweriter` implements this in Matlab

# Function poweriter

```
1 function [gamma,x] = poweriter(A,numiter)
2 % POWERITER    Power iteration for the dominant eigenvalue.
3 % Input:
4 %   A          square matrix
5 %   numiter    number of iterations
6 % Output:
7 %   gamma      sequence of eigenvalue approximations (vector)
8 %   x          final eigenvector approximation
9
10 n = length(A);
11 x = randn(n,1);
12 x = x/norm(x,inf);
13 for k = 1:numiter
14     y = A*x;
15     [normy,m] = max(abs(y));
16     gamma(k) = y(m)/x(m);
17     x = y/y(m);
18 end
```

- max returns max value and location in vector
- Sparsity used automatically

# Convergence of power iteration

- Return to  $k$ th approximation:

$$\gamma_k = \frac{y_{k,m}}{x_{k,m}} = \lambda_1 \frac{1 + r_2^{k+1}b_2 + \cdots + r_n^{k+1}b_n}{1 + r_2^k b_2 + \cdots + r_n^k b_n},$$

where  $r_j = \lambda_j/\lambda_1$  and the  $b_j$  are constants.

- Consider the denominator:

$$\begin{aligned} r_2^k b_2 + \cdots + r_n^k b_n &= r_2^k \left[ b_2 + \left( \frac{r_3}{r_2} \right)^k b_3 + \cdots + \left( \frac{r_n}{r_2} \right)^k b_n \right] \\ &= r_2^k \left[ b_2 + \left( \frac{\lambda_3}{\lambda_2} \right)^k b_3 + \cdots + \left( \frac{\lambda_n}{\lambda_2} \right)^k b_n \right]. \end{aligned}$$

- For simplicity, assume that

$$|\lambda_2| > |\lambda_3| \geq \cdots \geq |\lambda_n|.$$

## Convergence of power iteration

- With this assumption all of the ratios  $\left(\frac{r_j}{r_2}\right)^k \rightarrow 0$  for  $k \rightarrow \infty$ , so that the term approaches just the leading term,  $b_2 r_2^k$ , for large  $k$
- With the simpler denominator, it is relatively easy to use a geometric series for it
- Using that in the eigenvalue approximation gives

$$\begin{aligned}\gamma_k &\rightarrow \lambda_1 (1 + b_2 r_2^{k+1}) (1 - b_2 r_2^k + O(r_2^{2k})), \\ \gamma_k - \lambda_1 &\rightarrow \lambda_1 b_2 (r_2 - 1) r_2^k.\end{aligned}$$

- The next term has  $\gamma_{k+1} - \lambda_1 \rightarrow \lambda_1 b_2 (r_2 - 1) r_2^{k+1}$
- Dividing the last two expressions tells us what happens each iteration...

# Convergence of power iteration

- We find that the ratio is

$$\frac{\gamma_{k+1} - \lambda_1}{\gamma_k - \lambda_1} \rightarrow r_2 = \frac{\lambda_2}{\lambda_1} \quad \text{as } k \rightarrow \infty.$$

- The new approximation is roughly a factor of  $r_2$  closer than the old one
- This is linear convergence: the error drops by a constant factor each time
- It is usually true once one has done a few iterations, so that the computed values start to come from “large  $k$ ”
- Thinking of taking this limit for a (potentially) large number of iterations is what we mean when we say:

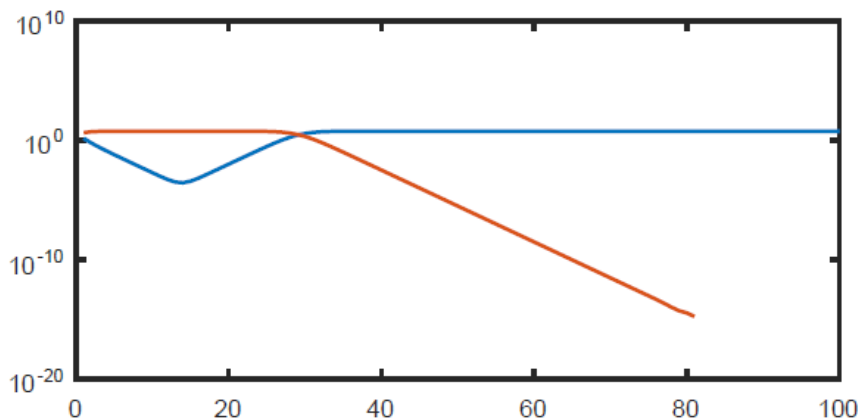
*The error in the eigenvalue estimates  $\gamma_k$  of power iteration is reduced asymptotically by a constant factor  $\lambda_2/\lambda_1$  on each iteration.*

# Example: convergence

## Example 8.2.2

```
A = [ 6 3 3; 1 10 1; 2 5 5];  
[V,D] = eig(A);  
x = [2;-1;2]+1e-8;  
for n = 1:100  
    y = A*x;  
    [~,m] = max(abs(y));  
    gamma(n) = y(m)/x(m);  
    x = y/y(m);  
end  
semilogy(abs(gamma-6))  
hold on, semilogy(abs(gamma-12))
```

- How fast do iterates gamma converge to eigenvalues?
- Blue:  $\lambda_2 = 6$ ; red:  $\lambda_1 = 12$
- Linear convergence to  $\lambda_1$



## Convergence of power iteration

- This is not a practical implementation
- We will continue to improved methods



## Section 8.3

# Inverse Iteration

## Inverse iteration

- Power iteration only finds the dominant eigenvalue
- We can extend power iteration to find other eigenvalues
- The idea is to use some simple linear algebra results to change the relative sizes of the eigenvalues

## Inverse iteration

- Here are those linear algebra results

### Theorem 8.3.1

Let  $A$  be an  $n \times n$  matrix with eigenvalues  $\lambda_1, \dots, \lambda_n$  (possibly with repeats), and let  $s$  be a complex scalar. Then:

1. The eigenvalues of the matrix  $A - sI$  are  $\lambda_1 - s, \dots, \lambda_n - s$ .
2. If  $s$  is not an eigenvalue of  $A$ , the eigenvalues of the matrix  $(A - sI)^{-1}$  are  $(\lambda_1 - s)^{-1}, \dots, (\lambda_n - s)^{-1}$ .
3. The eigenvectors associated with the eigenvalues in the first two parts are the same as those of  $A$ .

## Inverse iteration

- Great options for moving around eigenvalues, if true
- Consider item 1: start with  $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$ ; subtract  $s\mathbf{I}\mathbf{v}$  from both sides

$$\begin{aligned}\mathbf{A}\mathbf{v} - s\mathbf{I}\mathbf{v} &= \lambda\mathbf{v} - s\mathbf{I}\mathbf{v} \rightarrow \\ (\mathbf{A} - s\mathbf{I})\mathbf{v} &= (\lambda - s)\mathbf{v}\end{aligned}$$

- We have that a matrix times a vector gives a scalar times the same vector, as in part 1: proven.
- For part 2, we assumed that  $s \neq \lambda_k$ , so  $\mathbf{A} - s\mathbf{I}$  is non-singular, then straightforward rearrangement gives

$$(\mathbf{A} - s\mathbf{I})^{-1}\mathbf{v} = (\lambda_k - s)^{-1}\mathbf{v}$$

- This proves that item 2 is true.
- Item 3 follows too.

## Inverse iteration

- Consider item 2 with  $s = 0$ : start with  $A$  having a *smallest* eigenvalue:

$$|\lambda_n| \geq |\lambda_{n-1}| \geq \cdots > |\lambda_1|.$$

- Then

$$|\lambda_1^{-1}| > |\lambda_2^{-1}| \geq \cdots \geq |\lambda_n^{-1}|,$$

so that the reciprocal of the smallest eigenvalue is now the dominant one

- We can use power iteration to find the reciprocal value, and thus approximate the smallest eigenvalue (the one closest to zero)
- This is *inverse iteration*
- The convergence rate is linear again, with rate

$$\frac{\lambda_2^{-1}}{\lambda_1^{-1}} = \frac{\lambda_1}{\lambda_2}.$$

## Shifted inverse iteration

- Consider  $s \neq 0$ : order eigenvalues of  $\mathbf{A} - s\mathbf{I}$  with distance from

$$|\lambda_n - s| \geq \cdots \geq |\lambda_2 - s| > |\lambda_1 - s|,$$

- Then

$$|\lambda_1 - s|^{-1} > |\lambda_2 - s|^{-1} \geq \cdots \geq |\lambda_n - s|^{-1}.$$

- We can use power iteration to find the first reciprocal value, and thus approximate the eigenvalue closest to  $s$
- This is *shifted inverse iteration*
- Power iteration on  $(\mathbf{A} - s\mathbf{I})^{-1}$  converges to  $(\lambda_1 - s)^{-1}$  provided that  $\lambda_1$  is closest to  $s$

## Shifted inverse iteration

- There is a new computational wrinkle here
- We want to find:  $y_k = (A - sI)^{-1}x_k$ . for a sequence of  $k$  values
- However, we do NOT want to compute  $(A - sI)^{-1}$  to do it!
- Instead, we could write what is needed as

$$\text{Solve } (A - sI)y_k = x_k \text{ for } y_k.$$

- The solve is the computational version of the inverse, but is faster

## Shifted inverse iteration

- We need

$$\text{Solve } (A - sI)y_k = x_k \text{ for } y_k.$$

- To begin with, we use PLU factorization, and live with any fill-in that may result for sparse matrices
- Function 8.3.1 does inverse iteration
- Note that the iteration is for  $\beta_k = (\gamma_k - s)^{-1}$ ; the approximation to the original eigenvalue is  $\gamma_k = s + \beta_k^{-1}$



# Function inviter

---

Function 8.3.1 (inviter) Shifted inverse iteration for the closest eigenvalue.

---

```
1 function [gamma,x] = inviter(A,s,numiter)
2 % INVITER   Shifted inverse iteration for the closest eigenvalue.
3 % Input:
4 %   A       square matrix
5 %   s       value close to targeted eigenvalue (complex scalar)
6 %   numiter  number of iterations
7 % Output:
8 %   gamma   sequence of eigenvalue approximations (vector)
9 %   x       final eigenvector approximation
10
11 n = length(A);
12 x = randn(n,1);
13 x = x/norm(x,inf);
14 B = A - s*eye(n);
15 [L,U] = lu(B);
16 for k = 1:numiter
17     y = U \ (L\x);
18     [normy,m] = max(abs(y));
19     gamma(k) = x(m)/y(m) + s;
20     x = y/y(m);
21 end
```

- Random x ensures some component in eigenvector direction
- Should be  $s * \text{speye}(n)$  here

# Example: inverse iteration

## Example 8.3.1

```
lambda = [1 -0.75 0.6 -0.4 0];  
A = triu(ones(5),1) + diag(lambda);  
format long
```

```
[gamma,x] = inviter(A,0.7,30);  
eigval = gamma(end)
```

```
eigval =  
0.5999999999999998
```

- Create a  $5 \times 5$  matrix with known eigenvalues
- Use  $s = 0.7$  and do 30 iterations
- Convergence is to closest eigenvalue with value 0.6

## Example: inverse iteration

### Example 8.3.1

```
observed_rate = err(26)/err(25)
```

```
observed_rate =  
-0.327983951855567
```

```
theoretical_rate = (lambda(3)-0.7) / (lambda(1)-0.7)
```

```
theoretical_rate =  
-0.333333333333333
```

- We could also compute the ratio  $(\gamma_{k+1} - s)/(\gamma_k - s)$  in Matlab using  $s=0.7$  and

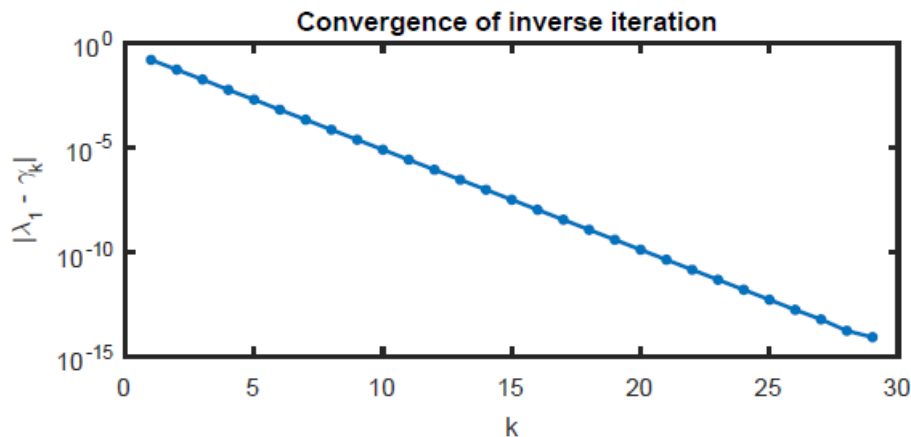
```
seq_of_rate = (gamma(2:end)-s) ./ (gamma(1:end-1)-s)
```

and see that the sequence of ratios approaches  $1/3$

# Example: inverse iteration

## Example 8.3.1

```
err = eigval - gamma;  
semilogy(abs(err), 'b.-')
```



- Convergence rate is linear: drops by constant factor of about 0.33 each time
- How do we know this?

## Dynamic shifting

- In our last example, we left  $s$  as a single value throughout the iteration
- This works reasonably well, but one can do better
- This time, let  $s = \gamma_k$  when we compute  $\gamma_{k+1}$
- So, we are adjusting  $s$  on the fly
- How well does it work?

## Example: inverse iteration

### Example 8.3.2

```
lambda = [1 -0.75 0.6 -0.4 0];  
A = triu(ones(5),1) + diag(lambda);  
format long  
I = eye(5); s = 0.7;  
x = ones(5,1);  
y = (A-s*I)\x; gamma = x(1)/y(1) + s
```

```
gamma =  
0.703481392557023
```

```
s = gamma;  
x = y/y(1);  
y = (A-s*I)\x; gamma = x(1)/y(1) + s
```

```
gamma =  
0.561276140617300
```

- Same  $5 \times 5$  matrix as previous example
- Use  $s = 0.7$  and do 1 iteration
- Change  $s$  to  $\gamma$  and repeat
- Doesn't look too good yet, but...

# Example: inverse iteration

## Example 8.3.2

```
for k = 1:4
    s = gamma; x = y/y(1);
    y = (A-s*I)\x; gamma = x(1)/y(1) + s
end
```

```
gamma =
    0.596431288475387
gamma =
    0.599971709182010
gamma =
    0.599999997855635
gamma =
    0.600000000000000
```

- Now repeat the process a few times
- Boom!
- Convergence is very rapid: roughly double the number of digits each time
- This is *quadratic convergence*
- Like Newton's method near a root

## Dynamic shifting

- There is a price for this increased rate of convergence
- Before, with constant shift, we could factor  $\mathbf{A} - s\mathbf{I}$  once, and each iterate is faster, but there can be many more
- With dynamic shifting, we need to factor  $\mathbf{A} - s_k\mathbf{I}$  for each iteration but there may be many fewer iterations
- For our examples, 30 iterates for constant shift, only 6 with dynamic shift
- In many cases it pays off to dynamically shift.



## Section 8.4

# Krylov Subspaces

## Krylov subspaces

- Say we start with a seed vector  $\mathbf{u}$
- If we repeatedly multiply by a matrix  $\mathbf{A}$ , we can keep all of the products, and try to use that set as a basis for approximating answers.
- Thus, we are creating the set  $\mathbf{u}, \mathbf{A}\mathbf{u}, \mathbf{A}^2\mathbf{u}, \dots$
- In the power or inverse iterations we only kept the latest on each time
- We get the  $n \times m$  *Krylov matrix* if the columns are  $\mathbf{u}, \mathbf{A}\mathbf{u}, \mathbf{A}^2\mathbf{u}, \dots$

$$\mathbf{K}_m = [\mathbf{u} \quad \mathbf{A}\mathbf{u} \quad \mathbf{A}^2\mathbf{u} \quad \dots \quad \mathbf{A}^{m-1}\mathbf{u}]$$

## Krylov subspaces

- The columns of the  $n \times m$  Krylov matrix are important

$$K_m = [u \quad Au \quad A^2u \quad \cdots \quad A^{m-1}u]$$

- The columns form the  $m$ th Krylov subspace  $\mathcal{K}_m$  of  $\mathbb{C}^n$
- Each column is  $n$  long, but there are only  $m$  columns
- We want to approximate answers from this subspace; said another way, we want to find answers that are a linear combo of the columns
- The dimension of  $\mathcal{K}_m$  is the same as the rank of  $K_m$ , which is often  $m$  but may be smaller

## Properties

- Generating columns of Krylov matrix is relatively easy for sparse  $A$

$$K_m = [u \quad Au \quad A^2u \quad \cdots \quad A^{m-1}u]$$

- And, there are some nice properties...

### Lemma 8.4.1

Suppose  $A$  is  $n \times n$ ,  $0 < m < n$ , and a vector  $u$  is used to generate Krylov subspaces. If  $x \in \mathcal{K}_m$ , then the following hold:

1.  $x = K_m z$  for some  $z \in \mathbb{C}^m$
2.  $x \in \mathcal{K}_{m+1}$
3.  $Ax \in \mathcal{K}_{m+1}$

## Properties

- Part 1 simply says that if  $\mathbf{x} \in \mathcal{K}_m$ , then for some coefficients  $c_1, c_2, \dots, c_m$ ,  $\mathbf{x}$  can be written as

$$\mathbf{x} = c_1 \mathbf{u} + c_2 A \mathbf{u} + \dots + c_m A^{m-1} \mathbf{u}.$$

- Then, let  $\mathbf{z} = [c_1 \ c_2 \ \dots \ c_m]^T$ , and recall that

$$\mathbf{K}_m = [\mathbf{u} \ A \mathbf{u} \ A^2 \mathbf{u} \ \dots \ A^{m-1} \mathbf{u}]$$

- For Part 3, just multiply the first equation by  $A$ , and one has

$$A \mathbf{x} = c_1 A \mathbf{u} + c_2 A^2 \mathbf{u} + \dots + c_m A^m \mathbf{u} \in \mathcal{K}_{m+1}.$$

## Reducing dimension and solving linear systems

- Consider the linear system  $\mathbf{A}x = \mathbf{b}$  for  $n \times n$  nonsingular  $\mathbf{A}$
- The original idea for using Krylov subspaces was to create a space with  $n$  basis vectors,  $\mathcal{K}_n$
- This could come with a predetermined number of steps, making it a direct method ( $m = n$  here for multiplications by  $\mathbf{A}$ )

$$K_m = [u \quad Au \quad A^2u \quad \cdots \quad A^{m-1}u]$$

- However, the columns of  $K_m$  effectively come from power method
- The columns are converging to the dominant eigenvector
- This makes the *direct method* with  $m = n$  ill-conditioned and impractical

## Reducing dimension and solving linear systems

- An alternative view is to say that most of the approximation or information is in the first few columns; the farther out columns are nearly dependent
- Why not stop early and perhaps get a good approximation with only  $m < n$  basis vectors, that is,  $\mathcal{K}_m$
- The problem changes then to one where we don't know in advance when to stop: it becomes an iterative method.
- Also, we need to get the best answer for the system  $Ax = b$  with  $n$  unknowns: this suggests a minimization of the error:

$$\min_{x \in \mathcal{K}_m} \|Ax - b\| = \min_{z \in \mathbb{C}^m} \|A(K_m z) - b\| = \min_{z \in \mathbb{C}^m} \|(AK_m)z - b\|.$$

## Reducing dimension and solving linear systems

- For the linear system  $Ax = b$ , the natural choice to start the Krylov iteration is  $u = b$

$$K_m = [u \quad Au \quad A^2u \quad \cdots \quad A^{m-1}u]$$

- We do adjust the size of each column with a normalization to prevent wildly different sizes of the elements
- Let's do an example



# Example: Krylov iteration

## Example 8.4.1

```
% Create a triangular matrix with
% known eigenvalues and a random vector b
lambda = 10 + (1:100) ;
A = diag( lambda ) + triu( rand(100),1 );
b = rand(100 ,1) ;

% Next we build up the first thirty Krylov
% matrices iteratively, using renormalization
% after each matrix-vector multiplication.
Km = b;
for m = 1:29
    v = A*Km(:,m);
    Km(:,m+1) = v/norm (v);
end
```

- $m$ th column of  $K_m$  created each time
- Normalize the column and append it

# Example: Krylov iteration

## Example 8.4.1

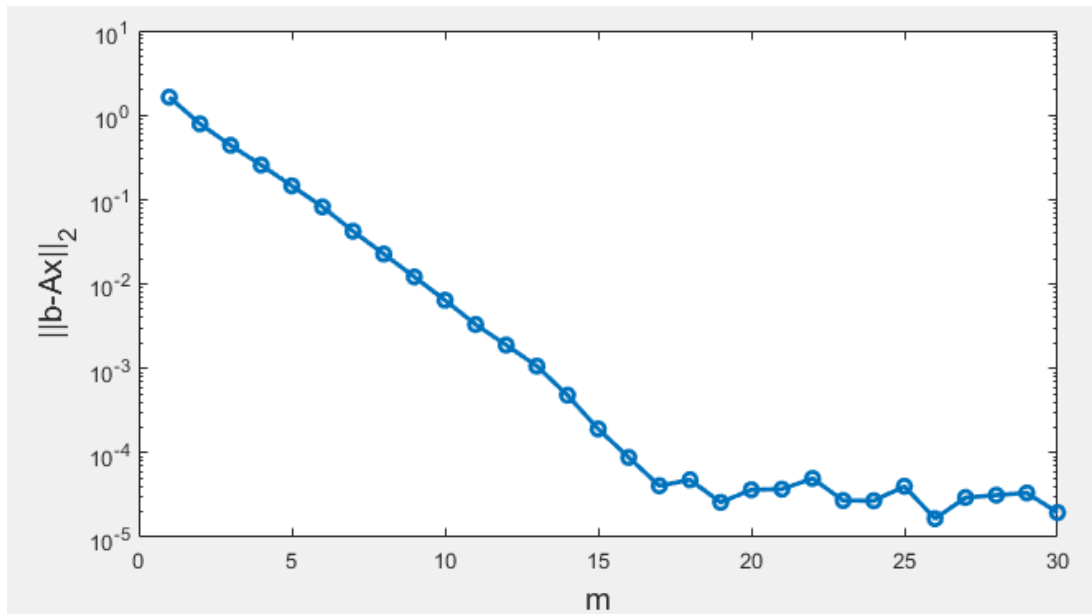
```
% Now we solve a least squares problem for
% Krylov matrices of increasing dimension.
for m = 1:30
    z = ( A*Km(:,1:m) ) \ b;
    x = Km(:,1:m)*z;
    resid(m) = norm( b-A*x );
end

% The linear system approximations show smooth
% linear convergence at first, but
% the convergence stagnates after only a
% few digits have been found.
semilogy(resid,'o-','LineWidth',2)
ylabel('||b-Ax||_2','FontSize',14)
xlabel('m','FontSize',14)
```

- Use the first  $m$  columns of  $K_m$  to approximate the solution
- $\backslash$  does least squares solution here
- Compute and save the residual

# Example: Krylov iteration

Example 8.4.1



- Residual “stalls” at  $m=17$
- Matlab throws warnings at that point because  $K_m$  is nearly singular
- We need a better approach!

## Fixing the problem

- The Krylov matrix we were generating was for  $m = 30$  with

$$K_m = [u \quad Au \quad A^2u \quad \cdots \quad A^{m-1}u]$$

- After column 17, any new columns were nearly dependent on the previous ones: warnings for  $m \geq 17$
- The opposite of this problem is to use an orthogonal matrix, where the columns are far from dependent
- Try QR factorization:

$$K_m = Q_m R_m = [q_1 \quad q_2 \quad \cdots \quad q_m] \begin{bmatrix} R_{11} & R_{12} & \cdots & R_{1m} \\ 0 & R_{22} & \cdots & R_{2m} \\ \vdots & & \ddots & \\ 0 & 0 & \cdots & R_{mm} \end{bmatrix}$$

## Fixing the problem

- The columns  $\mathbf{q}_j, j = 1, \dots, m$  of  $\mathbf{Q}$  are orthonormal
- And, they are a basis for  $\mathcal{K}_m$ !
- From Lemma 8.4.1, we know that multiplying a vector in  $\mathcal{K}_m$  with  $\mathbf{A}$  will produce a vector in  $\mathcal{K}_{m+1}$
- Using the last column of  $\mathbf{Q}$ , we know that

$$\mathbf{A}\mathbf{q}_m \in \mathcal{K}_{m+1}$$

- It's then a linear combo of the  $m + 1$  columns of another  $\mathbf{Q}$ :

$$\mathbf{A}\mathbf{q}_m = H_{1m} \mathbf{q}_1 + H_{2m} \mathbf{q}_2 + \cdots + H_{m+1,m} \mathbf{q}_{m+1},$$

## Fixing the problem

- Because the columns  $\mathbf{q}_j$  are orthonormal,

$$\mathbf{q}_i^* \mathbf{q}_j = 0, \quad i \neq j$$

$$\mathbf{q}_i^* \mathbf{q}_j = 1, \quad i = j$$

- Premultiply by  $\mathbf{q}_i^*$ , and only one term on the right survives:

$$\mathbf{q}_i^* (A \mathbf{q}_m) = H_{1m} \mathbf{q}_1 + H_{2m} \mathbf{q}_2 + \cdots + H_{m+1,m} \mathbf{q}_{m+1}$$

- We get  $\mathbf{q}_i^* (A \mathbf{q}_j) = H_{im}, i = 1, \dots, m$
- Orthonormality of the  $\mathbf{q}_j$  makes this easy!!
- The first  $m$   $\mathbf{q}_j$  are known, and the  $H_{im}$  are then determined.
- What about the last term on the right?

## Fixing the problem: Arnoldi iteration

- With the first  $m$   $\mathbf{q}_j$  are known, and the  $H_{im}$  found, we can rearrange:

$$H_{m+1,m}\mathbf{q}_{m+1} = A\mathbf{q}_m - \sum_{i=1}^m H_{im}\mathbf{q}_i$$

- Now, we only know  $H_{m+1,m}\mathbf{q}_{m+1}$
- However, we know that we want  $\mathbf{q}_{m+1}$  to be a *unit vector*
- So, we can make  $H_{m+1,m}$  the norm of the result, and make  $\mathbf{q}_{m+1}$  its direction.
- This is called *Arnoldi iteration*

# Arnoldi iteration

- Here's an algorithm

1. Let  $q_1 = u/\|u\|$ .

2. For  $m = 1, 2, \dots$

i. Use (8.4.4) to find  $H_{im}$  for  $i = 1, \dots, m$ .

ii. Let

$$v = (Aq_m) - H_{1m} q_1 - H_{2m} q_2 - \dots - H_{mm} q_m.$$

iii. Let  $H_{m+1,m} = \|v\|$ .

iv. Let  $q_{m+1} = v/H_{m+1,m}$ .

- The big improvement here is that the Arnoldi algorithm finds an orthonormal basis for the Krylov subspace



# Example: Arnoldi iteration

## Example 8.4.2

Make a few steps of Arnoldi iteration for a small matrix

```
% A few steps of Arnoldi iteration using a small matrix
```

```
A = magic(6) ;
```

```
% The seed vector determines the first member of the orthonormal basis.
```

```
u = randn(6 ,1);
```

```
Q = u/norm(u)
```

```
% Multiplication by A gives us a new vector in K2.
```

```
Aq = A*Q(:,1)
```

```
% We subtract off its projection in the previous direction.
```

```
% The remainder is rescaled to give us the next orthonormal column.
```

```
v = Aq - (Q(:,1)'*Aq)*Q(:,1)
```

```
Q(:,2) = v/norm(v)
```

- The first column was the unit vector in the  $u$  direction
- The second vector is  $A$  times that first vector, then normalized

## Example: Arnoldi iteration

Make a few steps of Arnoldi iteration for a small matrix


### Example 8.4.2

```
% On the next pass, we have to subtract off  
% the projections in two previous directions.
```

```
Aq = A*Q(:,2);
```

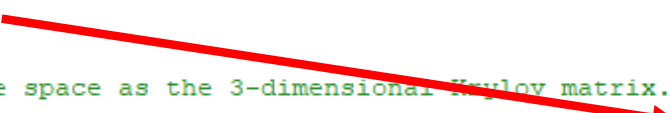
```
v = Aq - (Q(:,1)'*Aq)*Q(:,1) - (Q(:,2)'*Aq)*Q(:,2)
```

```
Q(:,3) = v/norm(v)
```



```
% At every step, Qm is an ONC matrix.
```

```
norm ( Q'*Q - eye(3) )
```



```
% And Qm spans the same space as the 3-dimensional Raylov matrix.
```

```
K = [ u A*u A*A*u ];
```

```
rank ( [Q,K] )
```



Q =

-0.7836	-0.3993	0.0892
0.2807	0.2386	0.5703
0.2801	-0.4678	0.0839
0.1849	0.2795	-0.2166
0.3077	-0.3680	-0.6645
-0.3161	0.5927	-0.4138

ans =

2.5117e-16

ans =

3

## A Key Identity

- We focused on the  $\mathbf{q}_j$  so far, but the  $H_{im}$  are important too
- They can be assembled into an *upper Hessenberg* matrix:

$$\begin{aligned}
 A\mathbf{Q}_m &= [\mathbf{A}\mathbf{q}_1 \quad \cdots \quad \mathbf{A}\mathbf{q}_m] \\
 &= [\mathbf{q}_1 \quad \mathbf{q}_2 \quad \cdots \quad \mathbf{q}_{m+1}] \begin{bmatrix} H_{11} & H_{12} & \cdots & H_{1m} \\ H_{21} & H_{22} & \cdots & H_{2m} \\ & H_{32} & \ddots & \vdots \\ & & \ddots & H_{mm} \\ & & & H_{m+1,m} \end{bmatrix} = \mathbf{Q}_{m+1}H_m,
 \end{aligned}$$

- This is a fundamental identity for Krylov subspace methods
- We will study practical and widely used methods based on it

---

**Function 8.4.1 (arnoldi)** Arnoldi iteration for Krylov subspaces.

---

```
1 function [Q,H] = arnoldi(A,u,m)
2 % ARNOLDI    Arnoldi iteration for Krylov subspaces.
3 % Input:
4 %   A       square matrix (n by n)
5 %   u       initial vector
6 %   m       number of iterations
7 % Output:
8 %   Q       orthonormal basis of Krylov space (n by m+1)
9 %   H       upper Hessenberg matrix, A*Q(:,1:m)=Q*H (m+1 by m)
10
11 n = length(A);
12 Q = zeros(n,m+1);
13 H = zeros(m+1,m);
14 Q(:,1) = u/norm(u);
15 for j = 1:m
16     % Find the new direction that extends the Krylov subspace.
17     v = A*Q(:,j);
18     % Remove the projections onto the previous vectors.
19     for i = 1:j
20         H(i,j) = Q(:,i)'*v;
21         v = v - H(i,j)*Q(:,i);
22     end
23     % Normalize and store the new basis vector.
24     H(j+1,j) = norm(v);
25     Q(:,j+1) = v/H(j+1,j);
26 end
```

## Section 8.5

### GMRES

## GMRES: Using Arnoldi iteration

- Arnoldi iteration may be most widely used to solve the square linear system  $\mathbf{A}\mathbf{x} = \mathbf{b}$  for  $n$  unknowns  $\mathbf{x}$
- The Krylov basis (columns of  $K_m$ ) was bad because some of the basis vectors were nearly dependent
- We tried to solve

$$\min_{\mathbf{x} \in \mathcal{K}_m} \|\mathbf{A}\mathbf{x} - \mathbf{b}\| = \min_{\mathbf{z} \in \mathbb{C}^m} \|\mathbf{A}K_m\mathbf{z} - \mathbf{b}\|,$$

- But, we found trouble when  $m$  got bigger
- We then went to an orthonormal basis from the QR factorization

$$\mathbf{K}_m = \mathbf{Q}_m\mathbf{R}_m$$

## GMRES: Using Arnoldi iteration

- Use the orthonormal columns of  $Q_m$  as the basis for the approximate answer:  $x = Q_m z$  for  $n$  unknowns  $x$
- Then we have converted the problem to

$$\min_{z \in \mathbb{C}^m} \|AQ_m z - b\|.$$

- At this point, there are  $n$  equations for the  $m$  coefficients in  $z$
- To make the method efficient, we can make the system smaller: use  $AQ_m = Q_{m+1}H_m$
- Then we have

$$\min_{z \in \mathbb{C}^m} \|Q_{m+1}H_m z - b\|.$$

## GMRES: Using Arnoldi iteration

- This system still has  $n$  equations for the  $m$  coefficients in  $\mathbf{z}$

$$\min_{\mathbf{z} \in \mathbb{C}^m} \|\mathbf{Q}_{m+1} \mathbf{H}_m \mathbf{z} - \mathbf{b}\|.$$

- Note that  $\mathbf{b}$  is a multiple of the unit vector  $\mathbf{q}_1$ ; we can then write

$$\mathbf{b} = \|\mathbf{b}\| \mathbf{Q}_{m+1} \mathbf{e}_1$$

- Now substitute for  $\mathbf{b}$ , to get the new problem

$$\min_{\mathbf{z} \in \mathbb{C}^m} \|\mathbf{Q}_{m+1} (\mathbf{H}_m \mathbf{z} - \|\mathbf{b}\| \mathbf{e}_1)\|.$$

- Is this better? It is still  $n$  equations for the  $m$  coefficients in  $\mathbf{z}$
- But, for  $\mathbf{w} \in \mathbb{C}^{m+1}$ , we have

$$\|\mathbf{Q}_{m+1} \mathbf{w}\|^2 = \mathbf{w}^* \mathbf{Q}_{m+1}^* \mathbf{Q}_{m+1} \mathbf{w} = \mathbf{w}^* \mathbf{w} = \|\mathbf{w}\|^2.$$



## GMRES: Using Arnoldi iteration

- These orthogonal matrices don't change the norm!
- We need only minimize the smaller system without the  $Q_{m+1}$

Thus

$$\min_{z \in \mathbb{C}^m} \|Q_{m+1}(H_m z - \|b\|e_1)\|.$$

becomes

$$\min_{z \in \mathbb{C}^m} \|H_m z - \|b\|e_1\|,$$

- Is this better? Now only  $(m + 1)$  eqns for the  $m$  coefficients in  $z$
- The solution for this system is  $z_m$
- The solution to the original system is approximately  $x_m = Q_m z_m$

## GMRES: Using Arnoldi iteration

- This method is called GMRES, for Generalized Minimum RESidual

$$\min_{z \in \mathbb{C}^m} \|H_m z - \|b\|e_1\|,$$

$$x_m = Q_m z_m$$

- GMRES uses the output of the Arnoldi iteration to minimize the residual of  $Ax = b$  over successive Krylov subspaces.
- Let's look at an example

# Example: GMRES

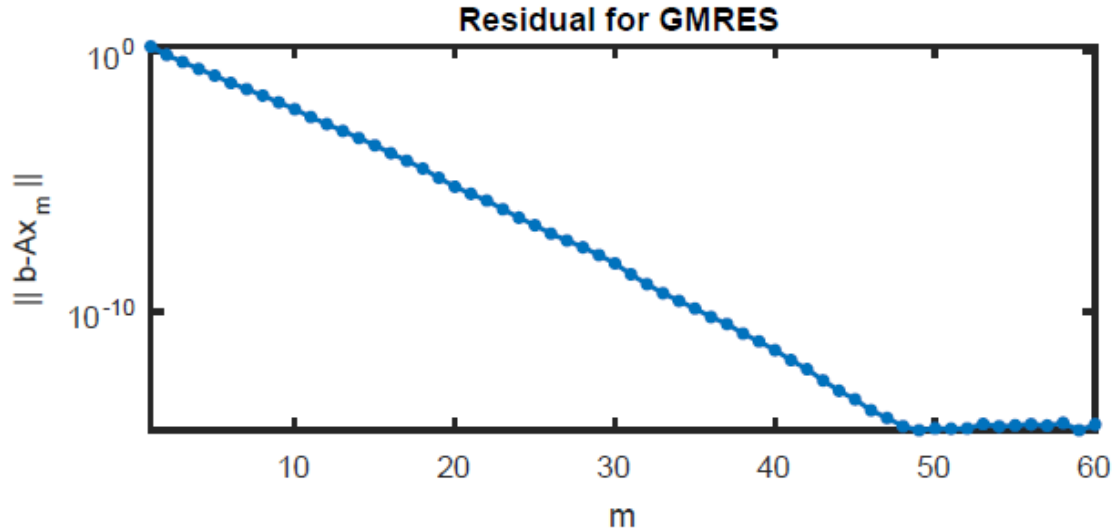
## Example 8.5.1

```
lambda = 10 + (1:100);  
A = diag(lambda) + triu(rand(100),1);  
b = rand(100,1);  
  
[Q,H] = arnoldi(A,b,60);  
  
for m = 1:60  
    s = [norm(b); zeros(m,1)];  
    z = H(1:m+1,1:m)\s;  
    x = Q(:,1:m)*z;  
    resid(m) = norm(b-A*x);  
end  
  
semilogy(resid, '.-')
```

- Matrix with known e-values and random ***b***
- $K_m$  not used, created ***Q, H*** instead
- Create a sequence of 60 approximations using one more column each time
- Plot all the residuals

# Example: GMRES

## Example 8.5.1



- Error drops to nearly machine epsilon before 50 columns
- No stalling of the error here

## GMRES

- A basic version of the algorithm is implemented in Function 8.5.1

---

### Function 8.5.1 (arngmres) GMRES for a linear system.

---

```
1 function [x,residual] = arngmres(A,b,M)
2 % ARNGMRES    GMRES for a linear system (demo only).
3 % Input:
4 %   A        square matrix (n by n)
5 %   b        right-hand side (n by 1)
6 %   M        number of iterations
7 % Output:
8 %   x        approximate solution (n by 1)
9 %   r        history of norms of the residuals
10
11 n = length(A);
12 Q = zeros(n,M);
13 Q(:,1) = b/norm(b);
14 H = zeros(M,M-1);
15
16 % Initial "solution" is zero.
17 residual(1) = norm(b);
18
```

# GMRES

- A basic version of the algorithm is implemented in Function 8.5.1

```
18
19  for m = 1:M
20
21      % Next step of Arnoldi iteration.
22      v = A*Q(:,m);
23      for i = 1:m
24          H(i,m) = Q(:,i)'*v;
25          v = v - H(i,m)*Q(:,i);
26      end
27      H(m+1,m) = norm(v);
28      Q(:,m+1) = v/H(m+1,m);
29
30      % Solve the minimum residual problem.
31      r = norm(b)*eye(m+1,1);
32      z = H(1:m+1,1:m) \ r;
33      x = Q(:,1:m)*z;
34      residual(m+1) = norm( A*x - b );
35
36  end
```

- Matlab has a more sophisticated version:  
gmres

## GMRES: convergence and restarting

- The residual  $||\mathbf{r}_m|| = ||\mathbf{b} - \mathbf{A}\mathbf{x}_m||$  can't increase during Arnoldi iteration because minimization over the largest space includes the previous ones
- But, it is difficult to say more than that.
- The previous example showed the cleanest behavior.
- It turns out that phases of sublinear and superlinear convergence can happen in (typically larger) systems. This depends on the spectrum of the matrix.
- Also, the number of columns of  $\mathbf{Q}$  and the number of entries in  $\mathbf{H}_m$ : work and storage grow like  $m^2$ , which can be too much
- Using GMRESS with *restarting* can help

## GMRES: convergence and restarting

- Good things about restarting:
  - one does not lose the previous gain of getting close to the answer
  - one uses low dimensional approximations again so that memory used is never allowed to get too large
- However, the low dimensional approximations may retard or even stagnate progress
- Matlab's GMRES function allows restarting
- Number of restarts (including initial): *outer iterations*
- Number of iterations after restarting: *inner iterations*



# Example: GMRES with restarting

## Example 8.5.2

```
maxit = 120;  rtol = 1e-8;  
d = 50;  
A = d^2*gallery('poisson',d);  
n = size(A,1)  
b = ones(n,1);
```

```
n =  
    2500
```

- Matrix discretizing a PDE that will be used not uncommonly
- The name comes from the PDE problem: Poisson equation
- From built-in `gallery` function
- Gives a large matrix

## Example: GMRES with restarting

### Example 8.5.2

- We use these five input arguments

```
>> help gmres
```

```
gmres    Generalized Minimum Residual Method.
```

```
X = gmres(A,B) attempts to solve the system of linear equations  $A \cdot X = B$  for X. The N-by-N coefficient matrix A must be square and the right hand side column vector B must have length N. This uses the unrestarted method with MIN(N,10) total iterations.
```

```
X = gmres(A,B,RESTART) restarts the method every RESTART iterations. If RESTART is N or [] then gmres uses the unrestarted method as above.
```

```
X = gmres(A,B,RESTART,TOL) specifies the tolerance of the method. If TOL is [] then gmres uses the default, 1e-6.
```

```
X = gmres(A,B,RESTART,TOL,MAXIT) specifies the maximum number of outer iterations. Note: the total number of iterations is RESTART*MAXIT. If MAXIT is [] then gmres uses the default, MIN(N/RESTART,10). If RESTART is N or [] then the total number of iterations is MAXIT.
```

# Example: GMRES with restarting

Example 8.5.2

- We use these five input arguments

```
X = gmres(A,B,RESTART,TOL,MAXIT)
```

Matrix A

RHS b

Restart after  
RESTART *inner*  
iterations

Specifies tolerance  
for method (1e-6  
default)

MAXIT specifies  
limit for *outer*  
iterations

# Example: GMRES with restarting

## Example 8.5.2

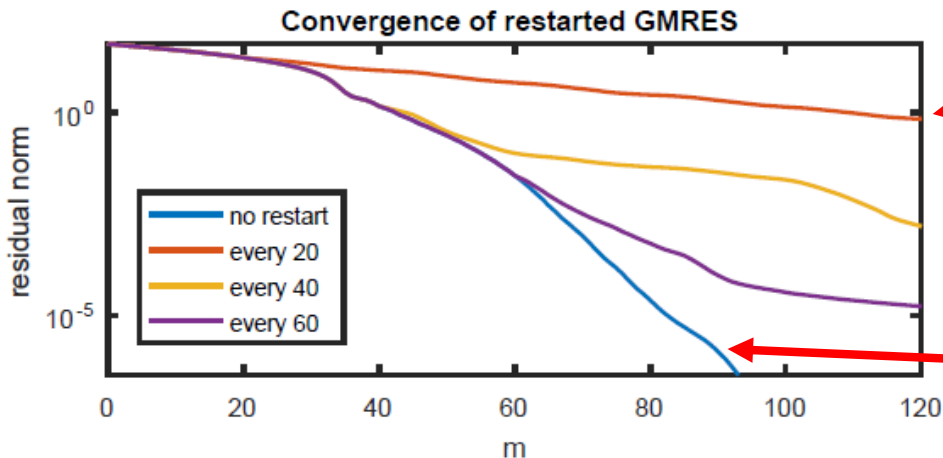
`[X, FLAG, RELRES, ITER, RESVEC] = gmres(A, B, ...)` also returns a vector of the residual norms at each inner iteration, including  $\text{NORM}(B - A \cdot X_0)$ . Note with preconditioners  $M_1, M_2$ , the residual is  $\text{NORM}(M_2 \setminus (M_1 \setminus (B - A \cdot X)))$ .

- We use only the last of these five outputs here
- $X$  is approximate answer is  $X$
- RELRES gives relative residual of  $X$ ,  $\|B - A \cdot X\| / \|B\|$
- FLAG states whether we achieved the tolerance
- ITER returns the outer and inner iteration number of the solution
- **RESVEC** is the list of residual values over iterations  $\|B - A \cdot X\|$

# Example: GMRES with restarting

## Example 8.5.2

```
rest = [maxit 20 40 60];  
for j = 1:4  
    [~,~,~,~,rv] = gmres(A,b,rest(j),rtol,maxit/rest(j));  
    semilogy(0:length(rv)-1,rv, '-'), hold on  
end
```



Most restarts:  
GMRES had not  
got to rapidly  
converging part  
before restart

No restarts

## GMRES: comments

- There are a lot of variations for solving  $A\mathbf{x} = \mathbf{b}$  using Krylov based methods
- These include
  - QMR: quasi-minimal residual
  - CGS: conjugate gradient stabilized
  - BiCGStab: bi-conjugate gradient stabilized
  - And more...
- We will discuss two special cases in the next section

## Section 8.6

### MINRES and conjugate gradients

## Special cases of GMRES

- Some really nice simplifications can be made in special cases of solving  $\mathbf{Ax} = \mathbf{b}$ , depending on the properties of  $\mathbf{A}$
- Consider Hermitian  $\mathbf{A}$ , where  $\mathbf{A}^* = \mathbf{A}$
- For square systems, we know that  $\mathbf{AQ}_m = \mathbf{Q}_{m+1}\mathbf{H}_m$
- Use the nice properties of an orthogonal matrix: pre-multiply by  $\mathbf{Q}_m^*$  to get

$$\mathbf{Q}_m^* \mathbf{AQ}_m = \mathbf{Q}_m^* \mathbf{Q}_{m+1} \mathbf{H}_m = \widetilde{\mathbf{H}}_m,$$

- $\widetilde{\mathbf{H}}_m$  is first  $m$  rows of  $\mathbf{H}_m$ , and it's Hessenberg form
- The lhs is Hermitian; the rhs must be too:  $\widetilde{\mathbf{H}}_m$  is tridiagonal!



## Special cases of GMRES: MINRES

- This is news we can use!
- The Arnoldi iteration has only a few components at each step:

$$Aq_m = H_{m-1,m} q_{m-1} + H_{mm} q_m + H_{m+1,m} q_{m+1}.$$

- We only need to find  $H_{m-1,m}$ ,  $H_{mm}$ ,  $H_{m+1,m}$ ,  $q_{m+1}$
- This truncated version of Arnoldi iteration is called *Lanczos iteration*
- Perhaps the best thing about this is that we need only a couple of previous vectors and a few coefficients each step
- This is only  $O(1)$  steps per iteration: No need for restarts!

## MINRES

- Matlab has a built-in function `minres`
- It also is relatively tractable for convergence analysis.
- If  $A$  is hermitian, then it has real eigenvalues
- Let  $s_+ = \{\lambda_i | \lambda > 0\}$  and define  $\kappa_+ = \max \lambda_i / \min \lambda_i$
- Let  $s_- = \{\lambda_i | \lambda < 0\}$  and define  $\kappa_- = \min \lambda_i / \max \lambda_i$

$$\frac{\|r_m\|_2}{\|b\|_2} \leq \left( \frac{\sqrt{\kappa_+ \kappa_-} - 1}{\sqrt{\kappa_+ \kappa_-} + 1} \right)^{\lfloor m/2 \rfloor},$$

- $\lfloor m/2 \rfloor$  is the floor function (round  $m/2$  down to nearest integer)
- If  $\kappa_+ \kappa_-$  is large, the ratio inside the parens tends to unity, and the convergence is slow

## MINRES

- Matlab has a built-in function `minres`
- It also is relatively tractable for convergence analysis.
- If  $A$  is hermitian, then it has real eigenvalues
- Let  $s_+ = \{\lambda_i | \lambda > 0\}$  and define  $\kappa_+ = \max \lambda_i / \min \lambda_i$
- Let  $s_- = \{\lambda_i | \lambda < 0\}$  and define  $\kappa_- = \min \lambda_i / \max \lambda_i$

$$\frac{\|r_m\|_2}{\|b\|_2} \leq \left( \frac{\sqrt{\kappa_+ \kappa_-} - 1}{\sqrt{\kappa_+ \kappa_-} + 1} \right)^{\lfloor m/2 \rfloor},$$

- $\lfloor m/2 \rfloor$  is the floor function (round  $m/2$  down to nearest integer)
- If  $\kappa_+ \kappa_-$  is large, the ratio inside the parens tends to unity, and the convergence is slow

## The conjugate gradient method

- Let's specify that nonsingular  $\mathbf{A}$  from  $\mathbf{Ax} = \mathbf{b}$  is HPD
- Then  $\mathbf{A}$  has a Cholesky factorization:  $\mathbf{A} = \mathbf{R}^* \mathbf{R}$
- For any vector  $u$ , we have

$$u^* \mathbf{A} u = (\mathbf{R} u)^* (\mathbf{R} u) = \|\mathbf{R} u\|^2,$$

- This is great because it is non-negative, and zero iff  $\mathbf{u} = \mathbf{0}$
- We can then define

$$\|u\|_{\mathbf{A}} = (u^* \mathbf{A} u)^{1/2}.$$

- The conjugate gradient method minimizes the error measured with the  $\mathbf{A}$ -norm over the sequence of Krylov subspaces
- That is, it computes  $\min_{\mathcal{K}_m} \|x_m - x\|_{\mathbf{A}}$

## The conjugate gradient method

- The conjugate gradient (CG) computes directions for each iterate that are “A-orthogonal” and estimates how long they should be.
- Another take on CG based on this view on the Sakai site
- It began being designed for  $n$  steps, i.e., as a direct method, but suffered from the same problem as Krylov iteration.
- Only later did it become a useful iterative method.
- The classical CG method is somewhat limited by the need for an HPD matrix
- But, that same list of modified functions applied to the CG method
- Matlab has a built-in function `pcg`

## Convergence of MINRES and CG

- Let  $\mathbf{A}$  be real and nonsingular with  $\kappa = \|A^{-1}\|_2 \|A\|_2$
- We have the following theorem:

### Theorem 8.6.1

Let  $\mathbf{A}$  be real and SPD with 2-norm condition number  $\kappa$ . For MINRES define  $R(m) = \|r_m\|_2 / \|b\|_2$ , and for CG define  $R(m) = \|x_m - x\|_{\mathbf{A}} / \|x\|_{\mathbf{A}}$ , where  $r_m$  and  $x_m$  are the residual and solution approximation associated with the space  $\mathcal{K}_m$ . Then

$$R(m) \leq 2 \left( \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^m. \quad (8.6.4)$$

- MINRES: bound on *relative residual*; CG: on *relative error*
- If  $\kappa$  is large, convergence is slow: spectrum of  $\mathbf{A}$  matters!
- Bigger exponent than indefinite case

## Convergence of MINRES and CG

- Using this result, we can estimate how many steps are needed.
- Let  $\kappa$  be a parameter (i.e., given)
- We want to specify a tolerance, say  $\epsilon$ ; to get the bound under this value, we need

$$2 \left( \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^m \approx \epsilon$$

- Solve for  $m$ :
- $$m \log \left( \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right) \approx \log \left( \frac{\epsilon}{2} \right).$$

- Simplify assuming large  $\kappa$ :
- $$\begin{aligned} \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} &= (1 - \kappa^{-1/2})(1 + \kappa^{-1/2})^{-1} \\ &= (1 - \kappa^{-1/2})(1 - \kappa^{-1/2} + \kappa^{-1} + \dots) \\ &= 1 - 2\kappa^{-1/2} + O(\kappa^{-1}), \quad \text{as } \kappa \rightarrow \infty. \end{aligned}$$

## Convergence of MINRES and CG

- Now use the last result inside the log function and Taylor expand to get

$$\log(1 + x) = x - (x^2/2) + \dots,$$

- Finally, we obtain

$$-2m\kappa^{-1/2} \approx \log\left(\frac{\epsilon}{2}\right), \text{ or } m = O(\sqrt{\kappa}),$$

- It is easy to see that the number of iterations  $m$  increases with  $\kappa^{1/2}$  and with decreasing tolerance  $\epsilon$
- Let's do an example



## Example: MINRES and CG

### Example 8.6.1

```
n = 1000;  
density = 0.008;  
A = sprandsym(n,density,1e-2,2);  
x = (1:n)'/n;  
b = A*x;
```

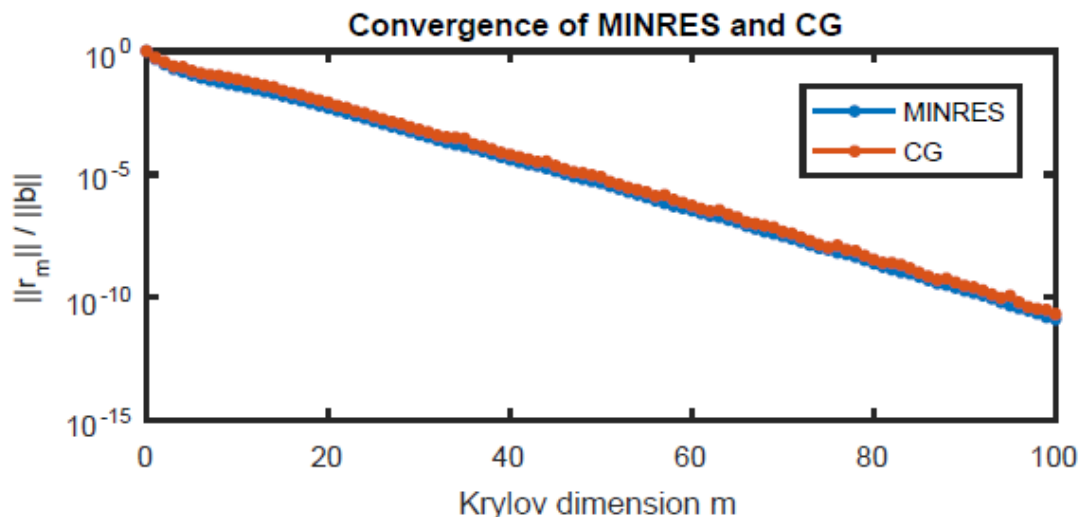
```
[xMR,~,~,~,residMR] = minres(A,b,1e-12,100);  
[xCG,~,~,~,residCG] = pcg(A,b,1e-12,100);  
semilogy(0:100,residMR/norm(b),'.-')  
hold on, semilogy(0:100,residCG/norm(b),'.-')
```

- Create sparse random matrix
- Condition number 100
- Known solution
- `minres` used first
- `pcg` second
- Same tolerance
- Plot relative residuals

# Example: MINRES and CG

## Example 8.6.1

```
[xMR,~,~,~,residMR] = minres(A,b,1e-12,100);  
[xCG,~,~,~,residCG] = pcg(A,b,1e-12,100);  
semilogy(0:100,residMR/norm(b),'.-')  
hold on, semilogy(0:100,residCG/norm(b),'.-')
```



- `minres` and `pcg` residuals are very similar here
- What about error?
- We know exact solution here

## Example: MINRES and CG

### Example 8.6.1

```
errorMR = norm( xMR - x ) / norm(x)
errorCG = norm( xCG - x ) / norm(x)
```

```
errorMR =  
    1.1192e-10  
errorCG =  
    7.9932e-11
```

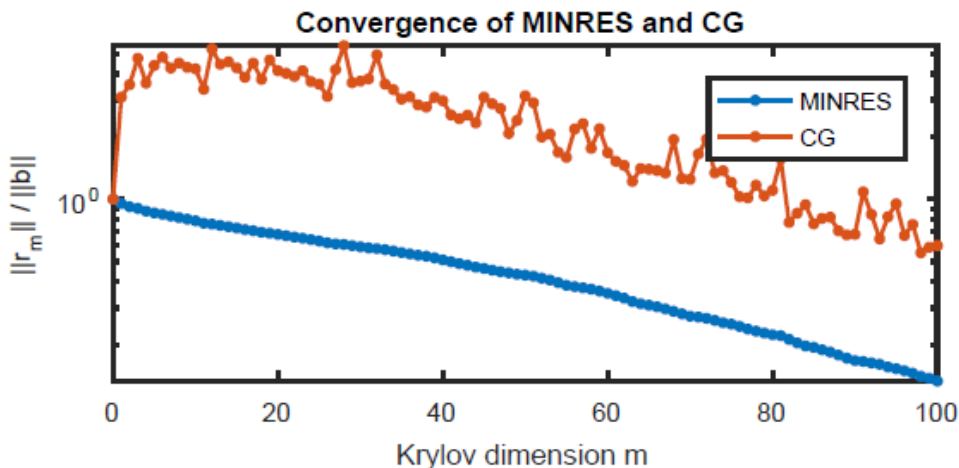
```
A = sprandsym(n,density,1e-4,2);
```

- Using the exact solution, we little difference when the condition number is  $\kappa = 100$
- Now let's up the ante and use a condition number of  $\kappa = 10^4$

# Example: MINRES and CG

## Example 8.6.1

```
[xMR,~,~,~,residMR] = minres(A,b,1e-12,100);  
[xCG,~,~,~,residCG] = pcg(A,b,1e-12,100);  
clf  
semilogy(0:100,residMR/norm(b),'.-')  
hold on, semilogy(0:100,residCG/norm(b),'.-')
```



- `minres` and `pcg` residuals' decay *rates* are similar, but residual bigger for CG
- Very little progress toward answer
- What about error?

## Example: MINRES and CG

### Example 8.6.1

```
errorMR = norm( xMR - x ) / norm(x)
errorCG = norm( xCG - x ) / norm(x)
```

```
errorMR =
    922.3604
errorCG =
    1.0061e+03
```

- Yep, that error is bad!
- Theory:  $\kappa = 100$ ,  $2 \left( \frac{9}{11} \right)^{100} \approx 3.9 \times 10^{-9}$
- Theory:  $\kappa = 10^4$ ,  $2 \left( \frac{99}{101} \right)^{100} \approx 0.27$
- Larger  $\kappa$  has strong effect!

$$R(m) \leq 2 \left( \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^m$$

## About stopping criteria

- $A\mathbf{x} = \mathbf{b}$ ; in  $m$ th iteration is  $\mathbf{x}_m$ ;  $\mathbf{r}_m = \mathbf{b} - A\mathbf{x}_m$
- We can think of solution as  $\mathbf{x} = \mathbf{x}_m + \boldsymbol{\delta}_m$
- Substitution gives  $A(\mathbf{x}_m + \boldsymbol{\delta}_m) = \mathbf{b}$ , or  $A\boldsymbol{\delta}_m = \mathbf{b} - A\mathbf{x}_m = \mathbf{r}_m$
- Approximately solving  $A\boldsymbol{\delta}_m = \mathbf{r}_m$  for the “correction” gives us an updated solution  $\mathbf{x}_{m+1} = \mathbf{x}_m + \boldsymbol{\delta}_m$
- Iteration can be stopped when  $\|\boldsymbol{\delta}_m\|$  or  $\|\mathbf{r}_m\|$ , or both, are smaller than preset tolerances
- When  $A$  is ill-conditioned,  $\|\mathbf{r}_m\|$  may be very small, while  $\|\boldsymbol{\delta}_m\|$  isn't: be careful

$$\frac{\|\mathbf{x} - \tilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq \kappa(A) \frac{\|\mathbf{r}\|}{\|\mathbf{b}\|}. \quad (2.8.4)$$

## Section 8.7

# Matrix-free Iterations

## Linear transformations, and undoing them

- We can think of  $A\mathbf{x} = \mathbf{b}$  as a linear transformation of  $\mathbf{x}$  to  $\mathbf{b}$
- The solution process undoes this transformation,  $\mathbf{x} = A^{-1}\mathbf{b}$
- Interestingly, we can undo a linear transformation without having to explicitly form the matrix  $A$ !
- How can we do this? Let's try an example.



## Linear transformations, and undoing them

- We can think of  $A\mathbf{x} = \mathbf{b}$  as a linear transformation of  $\mathbf{x}$  to  $\mathbf{b}$
- The solution process undoes this transformation,  $\mathbf{x} = A^{-1}\mathbf{b}$
- Interestingly, we can undo a linear transformation without having to explicitly form the matrix  $A$ !
- How can we do this? Let's try an example.

## Blurring images

- We can think of an image as an  $m \times n$  matrix  $\mathbf{X}$ , as we've already done
- Let's consider a simple model process for blurring
- Let the tridiagonal  $m \times n$  matrix  $\mathbf{B}$  be given by

$$B_{ij} = \begin{cases} \frac{1}{2}, & \text{if } i = j \\ \frac{1}{4}, & \text{if } |i - j| = 1 \\ 0, & \text{otherwise.} \end{cases}$$

- Pre-multiplication forms weighted combos or averages of elements in each column:  $\mathbf{BX}$
- This smears column elements

## Blurring images

- For rows, we can make a matrix with elements as it B, but make it  $n \times n$
- Call this tridiagonal matrix  $C$
- We can accomplish row blurring with transposes

$$(CX^T)^T = XC^T = XC$$

- We can get blurring of both rows and columns with

$$\text{blur}(X) = B^k X C^k$$

- Integer  $k \geq 1$

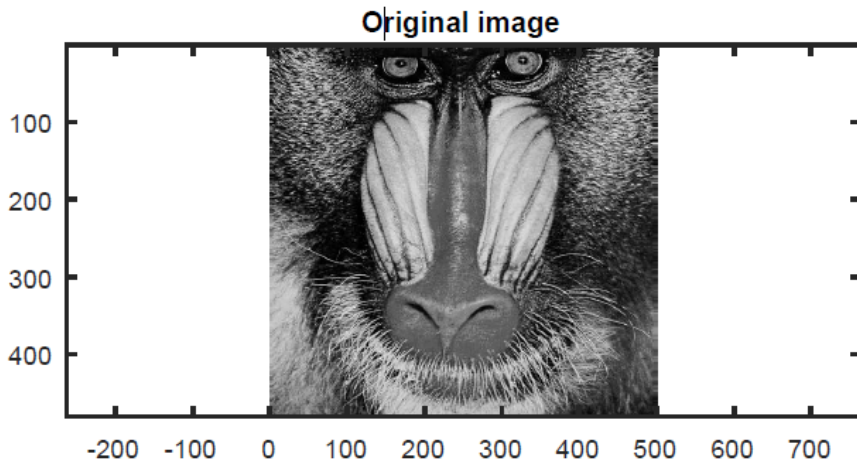
# Blurring example

## Example 8.7.1

```
load mandrill
[m,n] = size(X)
image(X), colormap(gray(256))
```

```
m =
    480
n =
    500
```

- We use our best old ex-friend the mandrill
- Now make some blurring matrices, and combine them for a blur function



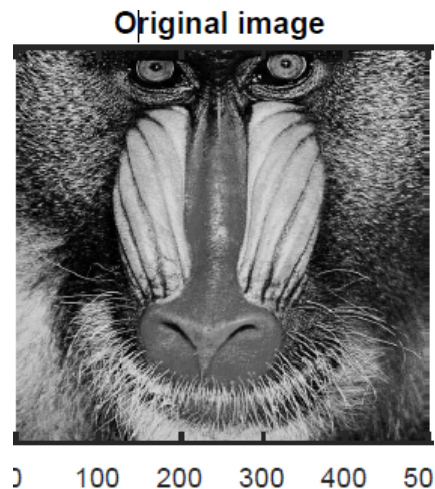
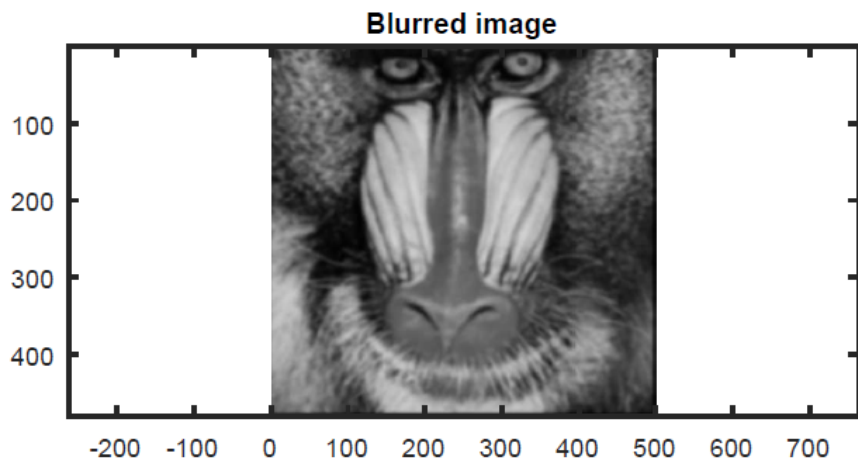
```
v = [1/4 1/2 1/4];
B = spdiags( repmat(v,m,1), -1:1, m,m);
C = spdiags( repmat(v,n,1), -1:1, n,n);
blur = @(X) B^12 * X * C^12;
```

# Blurring example

Example 8.7.1

This is what the blur did at left

```
image(blur(X))
```



## Deblurring images

- An important task is to try to deblur poorly focused images
- One way to treat this process is as linear transformation of the image
- To facilitate this, covert the  $m \times n$  matrix  $\mathbf{X}$  to an  $mn \times 1$  vector  $\mathbf{x}$
- Denote this by  $\text{vec}(\mathbf{X}) = \mathbf{x}$
- Converting from vector to matrix will be  $\text{unvec}(\mathbf{x}) = \mathbf{X}$
- We use blurring process from before  $\mathbf{Z} = \text{blur}(\mathbf{X})$

## Deblurring images

- Because the blurring is a linear transformation, we can write it as  $A \text{vec}(X) = \text{vec}(Z)$ .
- The matrix  $A$  is  $mn \times mn$
- For a 12 megapixel image,  $1.4 \times 10^4$  entries!!!
- The matrix is very sparse but it is not necessary
- Given an input vector  $u$ , we can compute  $v = Au$  via

$$U = \text{unvec}(u)$$

$$V = \text{blur}(U)$$

$$v = \text{vec}(V).$$

- Our blur recipe can be used without constructing  $A$

# Deblurring example

## Example 8.7.2

```
load mandrill
[m,n] = size(X)
v = [1/4 1/2 1/4];
B = spdiags( repmat(v,m,1), -1:1, m,m);
C = spdiags( repmat(v,n,1), -1:1, n,n);
blur = @(X) B^12 * X * C^12;
Z = blur(X);
```

```
m =
    480
n =
    500
```

```
vec = @(X) reshape(X,m*n,1);
unvec = @(x) reshape(x,m,n);
T = @(x) vec( blur(unvec(x)) );
```

- Mandrill again
- Imagine blurred image Z is given and we want to get back unblurred version X
- Make vec, unvec and blur
- T defines the action of the matrix A but does not create it

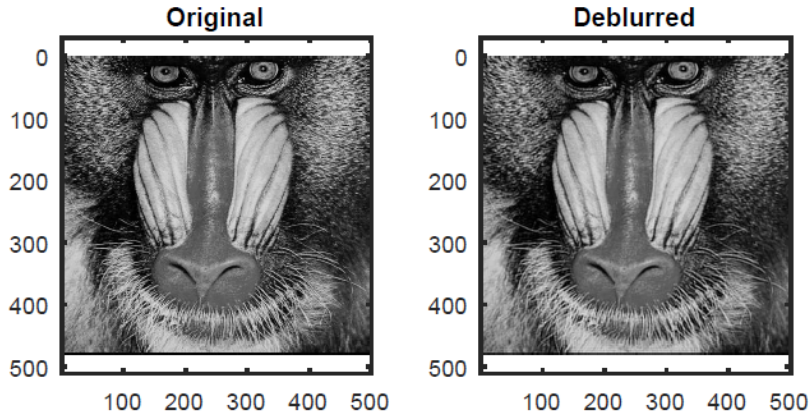


# Deblurring example

## Example 8.7.2

```
y = gmres(T,vec(Z),50,1e-5);  
Y = unvec(y);  
subplot(121)  
image(X), colormap(gray(256))  
subplot(122)  
image(Y), colormap(gray(256))
```

gmres(50) converged at outer iteration 2 (inner iteration 45) to a solution with relative residual  $1e-05$ .



- Apply GMRES to finding the inverse of the blur transformation
- The function  $T$  is passed in rather than the matrix: this effectively says how to compute the matrix vector product  $Au$
- Not a perfect deblurring

## Section 8.8

# Preconditioning

## Preconditioning

- As condition number increases, convergence of Krylov methods deteriorates
- Preconditioning can help get around this
- Say we are solving  $\mathbf{Ax} = \mathbf{b}$
- The idea is to use an easily inverted matrix that is close the original  $\mathbf{A}$  that gives a better conditioned system
- Say we are choosing  $\mathbf{M}$
- Then we are trying to find it so that  $(\mathbf{M}^{-1}\mathbf{A})\mathbf{x} = (\mathbf{M}^{-1}\mathbf{b})$  is easier to solve
- Do we compute  $\mathbf{M}^{-1}$ ? NO!!!!!!

## Preconditioning

- We want  $\mathbf{M}$  so that  $(\mathbf{M}^{-1}\mathbf{A})\mathbf{x} = (\mathbf{M}^{-1}\mathbf{b})$  is easier to solve
- Instead of  $\mathbf{M}^{-1}$ , we do the following two step process to compute any  $\mathbf{y} = (\mathbf{M}^{-1}\mathbf{A})\mathbf{v}$

1. Set  $\mathbf{u} = \mathbf{A}\mathbf{v}$ .

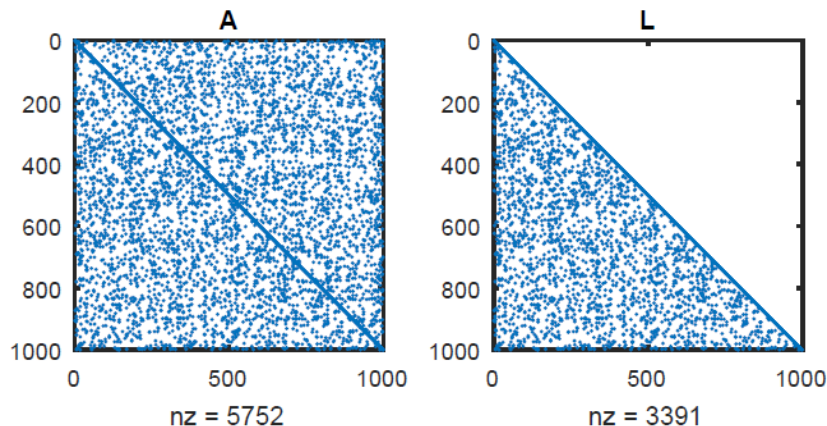
2. Solve  $\mathbf{M}\mathbf{y} = \mathbf{u}$  for  $\mathbf{y}$ .

- We use a numerical solve which is more efficient than inverting a matrix
- How to get  $\mathbf{M}$ ? One approach is *incomplete LU factorization*, or ILU factorization

## Preconditioning: ILU

- We know that LU factorization can ruin sparsity
- ILU says to only save nonzero elements that are similar to the original matrix
- Alternatively, one can set threshold and set to zero any element smaller than that

```
[L,U] = ilu(A);  
subplot(121), spy(A)  
subplot(122), spy(L)
```

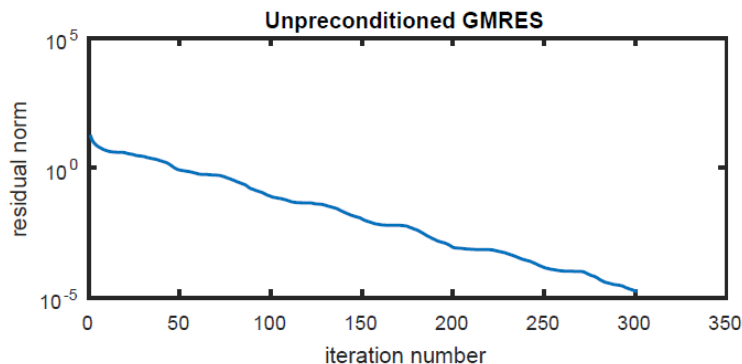


# Preconditioning example

## Example 8.8.1

Here is a  $1000 \times 1000$  matrix of density around 0.5%.

```
A = 0.6*speye(1000) + sprand(1000,1000,0.005,1/10000);  
b = rand(1000,1);  
[x,~,~,~,resid_plain] = gmres(A,b,50,1e-10,6); % restart  
at 50  
clf, semilogy(resid_plain, '-')
```

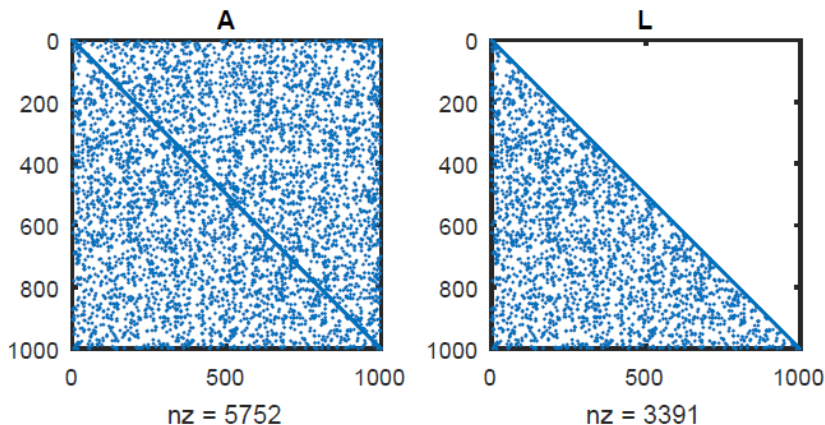


- Slow convergence for this system

# Preconditioning example

## Example 8.8.1

```
[L,U] = ilu(A);  
subplot(121), spy(A)  
subplot(122), spy(L)
```



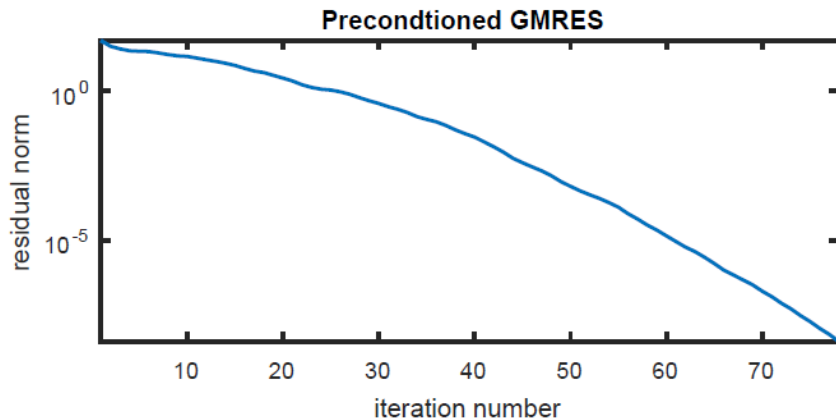
It does *not* produce a true factorization of  $A$ .

```
norm( full(A - L*U) )
```

# Preconditioning example

## Example 8.8.1

```
[x,~,~,~,resid_prec] = gmres(A,b,[],1e-10,300,L,U);  
clf, semilogy(resid_prec, '-')
```



- Better now



## Preconditioning

- We want  $\mathbf{M}$  so that  $(\mathbf{M}^{-1}\mathbf{A})\mathbf{x} = (\mathbf{M}^{-1}\mathbf{b})$  is easier to solve
- Not always easy to find
- Our last example found  $\mathbf{M} = \mathbf{LU}$
- If we were doing this manually we would need to solve  $(\mathbf{M}^{-1}\mathbf{b}) = \mathbf{c}$ , or  $\mathbf{c} = \mathbf{U} \setminus (\mathbf{L} \setminus \mathbf{b})$  to get the rhs
- Then use our previous two-step procedure to solve the preconditioned system

1. Set  $\mathbf{u} = \mathbf{A}\mathbf{v}$ .

2. Solve  $\mathbf{M}\mathbf{y} = \mathbf{u}$  for  $\mathbf{y}$ .