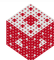
MATH 350: Introduction to Computational Mathematics

Chapter II: Solving Systems of Linear Equations

Greg Fasshauer

Department of Applied Mathematics
Illinois Institute of Technology

Spring 2011



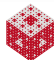
Outline

1. Applications, Motivation and Background Information
2. Gaussian Elimination = LU Decomposition
3. Forward and Back Substitution in MATLAB
4. Partial Pivoting
5. MATLAB Implementation of LU-Decomposition
6. Roundoff Error and the Condition Number of a Matrix
7. Special Matrices
8. An Application: Google's Page Rank

Where do systems of linear equations come up?

Everywhere!

- They appear straightforwardly in
  - analytic geometry (intersection of lines and planes),
  - traffic flow networks,
  - Google page ranks,
  - linear optimization problems (MATH 435),
  - Leontief's input-output model in economics,
  - electric circuit problems,
  - the steady-state analysis of a system of chemical or biological reactors,
  - the structural analysis of trusses,
  - and many other applications.
- They will appear as intermediate or final steps in many numerical methods such as
  - polynomial or spline interpolation (Ch. 3),
  - the solution of nonlinear systems (Ch. 4),
  - least squares fitting,
  - the solution of systems of differential equations (Ch. 6),
  - and many other advanced numerical methods.



- Equation form:

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 1 \\ 2x_1 + x_2 + 4x_3 &= 1 \\ 3x_1 + 4x_2 + x_3 &= 1 \end{aligned}$$

- Matrix form: $\mathsf{A}\boldsymbol{x} = \boldsymbol{b}$, with

$$\mathsf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 4 \\ 3 & 4 & 1 \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

Note: We will always think of vectors as column vectors. If we need to refer to a *row vector* we will use the notation $\boldsymbol{x}^T$.

Remark

Most of the material discussed in this chapter can be found in Chapter 2 of [NCM].

Never use A^{-1} to solve $A\boldsymbol{x} = \boldsymbol{b}$

In linear algebra we learn that the solution of

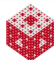
$$A\boldsymbol{x} = \boldsymbol{b}$$

is given by

$$\boldsymbol{x} = A^{-1}\boldsymbol{b}.$$

This is correct, but inefficient and more prone to roundoff errors.

Always solve linear systems (preferably with some decomposition method such as LU, QR or SVD).



Never use A^{-1} to solve $A\boldsymbol{x} = \boldsymbol{b}$ (cont.)

Example

Consider the trivial "system" $7x = 21$ and compare solution via the "inverse" and by straightforward division.

Solution

- Division immediately yields $x = \frac{21}{7} = 3$.
- Use of the "inverse" yields

$$x = 7^{-1} \times 21 = 0.142857 \times 21 = 2.999997.$$

Clearly, use of the inverse requires more work (first compute the inverse, then multiply it into the right-hand side), and it is less accurate. This holds even more so for larger systems of equations.

Note that MATLAB is "smarter" than this so that `7^(-1)*21` is still equal 3 (but slower than division, see `division.m`).

How to solve linear systems by "division" in MATLAB

In order to mimic what we do (naturally) for a single equation, MATLAB provides two very sophisticated *matrix division* operators:

- For systems $AX = B$, we have the backslash (or `mldivide`) operator, i.e.,

$$X = A \backslash B,$$

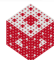
- and $XA = B$ is solved using a forward slash or (`mrdivide`) operator, i.e.,

$$X = B/A.$$

Remark

These operators provide black boxes for the solution of (possibly even non-square or singular) systems of linear equations.

We now want to understand a bit of what goes on "inside" the algorithms.



Example

Solve the 3×3 linear system

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 1 \\ 2x_1 + x_2 + 4x_3 &= 1 \\ 3x_1 + 4x_2 + x_3 &= 1 \end{aligned}$$

using a simple (easily programmable) algorithm.

Solution

The main idea is to *systematically* reduce the system to upper triangular form since triangular systems are easy to solve.
Thus, we will have

- an elimination phase (resulting in an upper triangular system),
- and a back substitution phase (during which we solve for the variables).

Solution (cont.)

$$x_1 + 2x_2 + 3x_3 = 1 \quad (1)$$
$$2x_1 + x_2 + 4x_3 = 1 \quad (2)$$
$$3x_1 + 4x_2 + x_3 = 1 \quad (3)$$

Use multiples of the first equation to eliminate x_1 from the other two equations:

$$x_1 + 2x_2 + 3x_3 = 1 \quad (1)$$
$$\xrightarrow{(2)-2\times(1)} \quad -3x_2 - 2x_3 = -1 \quad (2')$$
$$\xrightarrow{(3)-3\times(1)} \quad -2x_2 - 8x_3 = -2 \quad (3')$$

Use multiples of the new second equation to eliminate x_2 from the third equation:

$$x_1 + 2x_2 + 3x_3 = 1 \quad (1)$$
$$-3x_2 - 2x_3 = -1 \quad (2')$$
$$\xrightarrow{(3')-\frac{2}{3}\times(2')} \quad -\frac{20}{3}x_3 = -\frac{4}{3} \quad (3'')$$

Solution (cont.)

The system

$$x_1 + 2x_2 + 3x_3 = 1 \quad (1)$$

$$-3x_2 - 2x_3 = -1 \quad (2')$$

$$-\frac{20}{3}x_3 = -\frac{4}{3} \quad (3'')$$

is upper triangular.

Now we do the back substitution:

- From (3"): $x_3 = \frac{-4/3}{-20/3} = \frac{1}{5}$
- Insert x_3 into (2'): $x_2 = -\frac{1}{3}\left(-1 + 2\left(\frac{1}{5}\right)\right) = \frac{1}{5}$
- Insert x_3 and x_2 into (1): $x_1 = \left(1 - 3\left(\frac{1}{5}\right) - 2\left(\frac{1}{5}\right)\right) = 0$

Remark

In order to have an algorithm it is important to be consistent: in the elimination phase always subtract multiples of the pivot row from rows below it.

The algorithm from the previous example is known as Gaussian elimination (even though it can be found in Chinese writings from around 150 B.C., see "The Story of Maths", Episode 2)

Using matrix notation, the original system, $A\boldsymbol{x} = \boldsymbol{b}$, is

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

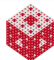
After the elimination phase we end up with the upper triangular matrix

$$U = \begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -2 \\ 0 & 0 & -\frac{20}{3} \end{bmatrix}.$$

Note that A can be written as $A = LU$, with U as above, and

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & \frac{2}{3} & 1 \end{bmatrix},$$

where the entries in the lower triangular part of L correspond to the multipliers used in the elimination phase.



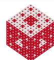
What is the advantage of LU factorization?

Consider the system $A\boldsymbol{x} = \boldsymbol{b}$ with LU factorization $A = LU$, i.e., we have

$$L \underbrace{U\boldsymbol{x}}_{=\boldsymbol{y}} = \boldsymbol{b}.$$

We can now solve the linear system by solving two easy triangular systems:

1. Solve the lower triangular system $L\boldsymbol{y} = \boldsymbol{b}$ for $\boldsymbol{y}$ by forward substitution.
2. Solve the upper triangular system $U\boldsymbol{x} = \boldsymbol{y}$ for $\boldsymbol{x}$ by back substitution.



An even greater advantage

Consider the problem $AX = B$ with many different right-hand sides associated with the same system matrix.
In this case we need to compute the factorization $A = LU$ only once, and then

$$AX = B \iff LUX = B.$$

We then proceed as before:

1. Solve $LY = B$ by many forward substitutions (in parallel).
2. Solve $UX = Y$ by many back substitutions (in parallel).

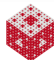
Remark

In order to appreciate the usefulness of this approach, note that one can show that the operations count for the matrix factorization is $\mathcal{O}(\frac{2}{3}n^3)$, while that for forward and back substitution is $\mathcal{O}(n^2)$.

We can easily solve a lower triangular system $L\boldsymbol{x} = \boldsymbol{b}$ by simple forward substitution.
In MATLAB this is done with the following function:

```
function x = forward(L,x)
% FORWARD. For lower triangular L,
%          x = forward(L,b) solves L*x = b.
[n,n] = size(L);
x(1) = x(1)/L(1,1);
for k = 2:n      % for each row, from the top down
   j = 1:k-1;    % all columns simultaneously
   x(k) = (x(k) - L(k,j)*x(j))/L(k,k);
end
```

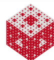
Note that `L(k,j)*x(j)` calculates a dot product. Also, note that the right-hand side vector, here `x`, is overwritten with the solution.
This function is included in `bslashtx.m` (see later).



An upper triangular system $U\boldsymbol{x} = \boldsymbol{b}$ is solved by back substitution.
The MATLAB code is:

```
function x = backsubs(U,x)
% BACKSUBS.  For upper triangular U,
%            x = backsubs(U,b) solves U*x = b.
[n,n] = size(U);
x(n) = x(n)/U(n,n);
for k = n-1:-1:1    % for each row, from the bottom up
   j = k+1:n;       % all columns simultaneously
   x(k) = (x(k) - U(k,j)*x(j))/U(k,k);
end
```

Again, `U(k,j)*x(j)` calculates a dot product, and the right-hand side vector is overwritten with the solution.
This function is also included in `bslashtx.m` (see later).



Example

Use Gaussian elimination to solve the linear system

$$\begin{aligned} 6x_1 + 2x_2 + 2x_3 &= -2 \\ 2x_1 + \frac{2}{3}x_2 + \frac{1}{3}x_3 &= 1 \\ x_1 + 2x_2 - x_3 &= 0. \end{aligned}$$

Solution

See the Maple worksheet `PartialPivoting.mw`.
Using simulated double-precision and standard Gaussian elimination we get the answer

$$\boldsymbol{x} = [1.333333333333335, 0, -5.000000000000003]^T.$$

Swapping rows to obtain a "good" pivot gives the correct solution

$$\boldsymbol{x} = [2.600000000000002, -3.800000000000001, -5.000000000000003]^T.$$

How to choose "good" pivots

The partial pivoting strategy is a simple one.
In order to avoid the tiny multipliers we encountered in the example above,

- at step k of the elimination algorithm we take that element in rows k through n of the k-th column which is largest in absolute value as the new pivot element.
- We also swap the k-th row with the new pivot row (or keep track of this swap in a permutation matrix P – see next slide).

Example

In step $k = 2$ of the previous example we need to check the elements in the second column of the last two rows to find that the new pivot is the element a_{32}, i.e., the larger of $|a_{22}|$ and $|a_{32}|$

$$A = \begin{bmatrix} 6 & 2 & 2 \\ 0 & 10^{-15} & -0.3333 \\ 0 & 1.6667 & -1.3333 \end{bmatrix} \xrightarrow{(2)\leftrightarrow(3)} \begin{bmatrix} 6 & 2 & 2 \\ 0 & 1.6667 & -1.3333 \\ 0 & 10^{-15} & -0.3333 \end{bmatrix}$$

Permutation Matrices

An identity matrix whose rows have been permuted is called a *permutation matrix*. For example,

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

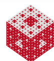
If we multiply a matrix A by a permutation matrix P, then we can either

- swap the rows of A (if we multiply from the left), i.e.,

$$PA = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 4 \\ 3 & 4 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 1 \\ 2 & 1 & 4 \end{bmatrix},$$

- or swap the columns (by multiplying from the right), i.e.,

$$AP = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 4 \\ 3 & 4 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 2 \\ 2 & 4 & 1 \\ 3 & 1 & 4 \end{bmatrix}.$$



Pivoting revisited

In the Maple worksheet `PartialPivoting.mw` we saw that Gaussian elimination (with partial pivoting) for the matrix

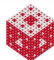
$$A = \begin{bmatrix} 6 & 2 & 2 \\ 2 & \frac{2}{3} & \frac{1}{3} \\ 1 & 2 & -1 \end{bmatrix}$$

leads to

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0.1667 & 1 & 0 \\ 0.3333 & 5.9999 \times 10^{-16} & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 6 & 2 & 2 \\ 0 & 1.6667 & -1.3333 \\ 0 & 0 & -0.3333 \end{bmatrix}.$$

With the permutation matrix

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



we can write the LU decomposition as $PA = LU$.

The Program `lutx.m` from [NCM]

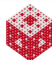
Most important statements of Gaussian elimination code:

```
for k = 1:n-1    % for each row
   :
   :
   % Compute multipliers
   i = k+1:n;                  % for all remaining rows
   A(i,k) = A(i,k)/A(k,k);
   % Update the remainder of the matrix
   j = k+1:n;                  % and all remaining columns
   A(i,j) = A(i,j) - A(i,k)*A(k,j);
end
```

Note: This would be quite a bit more involved in a programming language that does not use matrices and vectors as building blocks.

In fact, `A(i,j)` is a matrix, and `A(i,k)*A(k,j)` is (column vector)$\times$(row vector), i.e., a rank one matrix.

Both matrices are of size $(n-k) \times (n-k)$.



The Program `bslashtx.m` from [NCM]

The code represents a (simplified) readable version of the MATLAB backslash solver (which – as a built-in function – cannot be viewed by us).

It looks for special cases:

- lower triangular systems (for which it uses `forward.m` discussed earlier),
- upper triangular systems (for which it uses `backsubs.m` discussed earlier),
- symmetric positive definite systems (for which it uses MATLAB's built-in `chol` function).

For general systems it calls `lutx.m`.

Consider $A\boldsymbol{x} = \boldsymbol{b}$ with square nonsingular matrix A. We use the following notation: computed solution: $\boldsymbol{x}_*$ and exact solution $\boldsymbol{x} = A^{-1}\boldsymbol{b}$.
There are two ways to measure how good the solution is:

- *error*: $\boldsymbol{e} = \boldsymbol{x} - \boldsymbol{x}_*$
- *residual*: $\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}_*$

Note that error and residual are connected:

$$\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}_* = A\boldsymbol{x} - A\boldsymbol{x}_* = A(\underbrace{\boldsymbol{x} - \boldsymbol{x}_*}_{=\boldsymbol{e}}) = A\boldsymbol{e},$$

i.e., $\boldsymbol{e}$ is the (exact) solution of the linear system

$$A\boldsymbol{e} = \boldsymbol{r}.$$

Remark

Since A is nonsingular we clearly have: $\boldsymbol{e} = \boldsymbol{0} \iff \boldsymbol{r} = \boldsymbol{0}$.
However, a "small" residual need not imply a "small" error (or vice versa).

"Gaussian elimination with partial pivoting always produces small residuals" (see some qualifying remarks in [NCM])

Example

Solve the linear system

$$\begin{aligned} 0.780x_1 + 0.563x_2 &= 0.217 \qquad (4)\\ 0.913x_1 + 0.659x_2 &= 0.254 \qquad (5) \end{aligned}$$

using Gaussian elimination with partial pivoting on a three-digit decimal computer.

Solution

After swapping the two equations the multiplier is $\frac{0.780}{0.913} = 0.854$, and

$$0.913x_1 + 0.659x_2 = 0.254 \qquad (5)$$

$$\xrightarrow{(4)-0.854\times(5)} \qquad 0.001x_2 = 0.001 \qquad (4')$$

Solution (cont.)

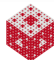
Back substitution yields

$$x_2 = \frac{0.001}{0.001} = 1.00, \qquad x_1 = \frac{0.254 - 0.659(1.00)}{0.913} = -0.443$$

so that $\boldsymbol{x}_* = [-0.443 \; 1.00]^T$.
We may now want to check the "accuracy" of our solution by computing the residual (let's do that with higher accuracy):

$$\begin{aligned} \boldsymbol{r} &= \boldsymbol{b} - \mathsf{A}\boldsymbol{x}_* = \begin{bmatrix} 0.217 \\ 0.254 \end{bmatrix} - \begin{bmatrix} 0.780 & 0.563 \\ 0.913 & 0.659 \end{bmatrix} \begin{bmatrix} -0.443 \\ 1.00 \end{bmatrix} \\ &= \begin{bmatrix} -0.000460 \\ -0.000541 \end{bmatrix}, \end{aligned}$$

so that we are led to believe that our solution is accurate.
However, the exact solution is $\boldsymbol{x} = [1 \; -1]^T$.



Solution (cont.)

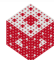
Thus, the error is

$$\boldsymbol{e} = \boldsymbol{x} - \boldsymbol{x}_* = \begin{bmatrix} 1 \\ -1 \end{bmatrix} - \begin{bmatrix} -0.443 \\ 1 \end{bmatrix} = \begin{bmatrix} 1.443 \\ -2 \end{bmatrix}$$

which is larger than the solution itself!

Q: When can we trust the residual, i.e., when does a small residual guarantee a small error?

A: When the condition number of A is small.



In order to understand the concept of the condition number of a matrix we need to introduce the concepts of vector and matrix norms.

Example

The most important vector norms are

- $\|\boldsymbol{x}\|_1 = \sum_{i=1}^{n} |\boldsymbol{x}_i|$, ℓ_1-norm or Manhattan norm.
- $\|\boldsymbol{x}\|_2 = \left(\sum_{i=1}^{n} |\boldsymbol{x}_i|^2\right)^{1/2}$, ℓ_2-norm or Euclidean norm.
- $\|\boldsymbol{x}\|_\infty = \max_{1\le i\le n} |\boldsymbol{x}_i|$, ℓ_∞-norm, maximum norm or Chebyshev norm.
- $\|\boldsymbol{x}\|_p = \left(\sum_{i=1}^{n} |\boldsymbol{x}_i|^p\right)^{1/p}$, ℓ_p-norm.

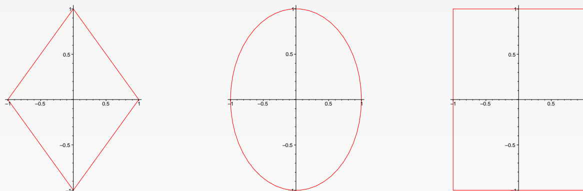
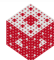


Figure: Unit “circles” for the ℓ_1, ℓ_2 and ℓ_∞ norms.

In general, any norm satisfies

1. $\|\boldsymbol{x}\| \geq 0$ for every vector $\boldsymbol{x}$, and $\|\boldsymbol{x}\| = 0$ only if $\boldsymbol{x} = \mathbf{0}$.
2. $\|\alpha \boldsymbol{x}\| = |\alpha| \|\boldsymbol{x}\|$ for every vector $\boldsymbol{x}$, and scalar α.
3. $\|\boldsymbol{x} + \boldsymbol{y}\| \leq \|\boldsymbol{x}\| + \|\boldsymbol{y}\|$ for all vectors $\boldsymbol{x}, \boldsymbol{y}$ (triangle inequality).



Example

Consider the vector $\boldsymbol{x} = [1,\ -2,\ 3]^T$. Compute $\|\boldsymbol{x}\|_p$ for $p = 1, 2, \infty$.

Solution

$$\|\boldsymbol{x}\|_1 = \sum_{i=1}^{3} |\boldsymbol{x}_i| = |1| + |-2| + |3| = 6.$$

$$\|\boldsymbol{x}\|_2 = \left(\sum_{i=3}^{n} |\boldsymbol{x}_i|^2\right)^{1/2} = \sqrt{1^2 + (-2)^2 + 3^2} = \sqrt{14}.$$

$$\|\boldsymbol{x}\|_\infty = \max_{1\le i\le n} |\boldsymbol{x}_i| = \max\{|1|, |-2|, |3|\} = 3.$$

For an $m \times n$ matrix A, the most popular matrix norms can be computed as follows:

- The maximum column sum norm is given by

$$\begin{aligned} \|A\|_1 &= \max_{1\le j\le n} \|A(:,j)\|_1 \\ &= \max_{1\le j\le n} \sum_{i=1}^{m} |A(i,j)|. \end{aligned}$$

- $\|A\|_2 = \max_{1\le j\le n} |\sigma_j|$, where σ_j is the j-th singular value of A.
    - We can compute $\sigma_j = \sqrt{\lambda_j}$ (where λ_j are the eigenvalues of $A^T A$),
    - or $\sigma_j = |\lambda_j|$ (with λ_j the eigenvalues of a real symmetric A).
- The maximum row sum norm is given by

$$\begin{aligned} \|A\|_\infty &= \max_{1\le i\le m} \|A(i,:)\|_1 \\ &= \max_{1\le i\le m} \sum_{j=1}^{n} |A(i,j)|. \end{aligned}$$

Example

Consider the *Hilbert matrix* $A = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}$. Compute $\|A\|_p$ for $p = 1, 2, \infty$.

Solution

- Column sum norm:

$$\begin{aligned} \|A\|_1 &= \max_{1\le j\le 3} \sum_{i=1}^{3} |A_{ij}| \\ &= \max_{1\le j\le 3} \{1 + \tfrac{1}{2} + \tfrac{1}{3}, \tfrac{1}{2} + \tfrac{1}{3} + \tfrac{1}{4}, \tfrac{1}{3} + \tfrac{1}{4} + \tfrac{1}{5}\} = \frac{11}{6} = 1.8333. \end{aligned}$$

- 2-norm:
$\|A\|_2 = \max_{1\le j\le 3} |\sigma_j| = \max_{1\le j\le 3} \{1.4083, 0.1223, 0.0027\} = 1.4083$
- Row sum norm: $\|A\|_\infty = \|A\|_1$ since A is symmetric.

In the previous example the three different norms of A were of similar magnitude.

Remark

All matrix norms are equivalent, i.e., comparable in size. In fact, for any $m \times n$ matrix A we have

$$\begin{aligned} \frac{1}{\sqrt{n}}\|A\|_\infty \le & \quad \|A\|_2 \quad \le \sqrt{n}\|A\|_1, \\ \frac{1}{\sqrt{m}}\|A\|_1 \le & \quad \|A\|_2 \quad \le \sqrt{m}\|A\|_\infty. \end{aligned}$$

Example

For the 3×3 Hilbert matrix this implies:

$$\frac{1}{\sqrt{3}}\|A\|_\infty \le \|A\|_2 \le \sqrt{3}\|A\|_1$$

$$\Longrightarrow \quad \underbrace{0.5774 \times 1.8333}_{=1.0585} \le \|A\|_2 \le \underbrace{1.7321 \times 1.8333}_{=3.1754}$$

Definition

The quantity $\kappa(A) = \|A\|\|A^{-1}\|$ is called the *condition number* of A.

Remark

The condition number depends on the type of norm used.

For the 2-norm of a nonsingular $n \times n$ matrix A we know $\|A\|_2 = \sigma_1$ (the largest singular value of A).
Similarly, $\|A^{-1}\|_2 = \frac{1}{\sigma_n}$ (with σ_n the smallest singular value of A).
Thus,

$$\kappa(A) = \|A\|_2\|A^{-1}\|_2 = \frac{\sigma_1}{\sigma_n}.$$

Note that $\kappa(A) \geq 1$. In fact, this holds for any norm.

Singular values and the SVD are discussed in detail in MATH 477.

Example

Consider the matrix $A = \begin{bmatrix} 0.780 & 0.563 \\ 0.913 & 0.659 \end{bmatrix}$ from our earlier example.
Compute $\kappa_2(A)$.

Solution

In MATLAB we compute the singular values of A with `s = svd(A)`.
This yields

$$\sigma_1 = 1.480952059, \quad \sigma_2 = 0.000000675,$$

so that

$$\kappa_2(A) = \frac{\sigma_1}{\sigma_2} = 2.193218999 \times 10^6.$$

This is very large for a 2×2 matrix.
Note that the latter can be obtained directly with the command `cond(A)` in MATLAB.

In order to see what effect the condition number has on the reliability of the residual for predicting the accuracy of the solution to the linear system we can refer to the following result.

Theorem

Consider the linear system $\mathsf{A}\boldsymbol{x} = \boldsymbol{b}$ *with computed solution* $\boldsymbol{x}_*$, *residual* $\boldsymbol{r} = \boldsymbol{b} - \mathsf{A}\boldsymbol{x}_*$, *and exact solution* $\boldsymbol{x}$. *Then*

$$\frac{1}{\kappa(\mathsf{A})}\frac{\|\boldsymbol{r}\|}{\|\boldsymbol{b}\|} \le \frac{\|\boldsymbol{x} - \boldsymbol{x}_*\|}{\|\boldsymbol{x}\|} \le \kappa(\mathsf{A})\frac{\|\boldsymbol{r}\|}{\|\boldsymbol{b}\|},$$

i.e., the relative error is bounded from above and below by the relative residual and the condition number of A.

In particular, if the condition number of A is small (close to 1), then a small residual will accurately predict a small error.

Earlier we used $A = \begin{bmatrix} 0.780 & 0.563 \\ 0.913 & 0.659 \end{bmatrix}$ with $\kappa_2(A) = 2.1932 \times 10^6$.

For $\boldsymbol{b} = \begin{bmatrix} 0.217 \\ 0.254 \end{bmatrix}$ (with $\|\boldsymbol{b}\|_2 = 0.3341$) we had $\boldsymbol{r} = \begin{bmatrix} -0.000460 \\ -0.000541 \end{bmatrix}$ (with $\|\boldsymbol{r}\|_2 = 7.1013 \times 10^{-4}$).

The theorem tells us

$$\frac{1}{\kappa(A)}\frac{\|\boldsymbol{r}\|}{\|\boldsymbol{b}\|} \le \frac{\|\boldsymbol{x}-\boldsymbol{x}_*\|}{\|\boldsymbol{x}\|} \le \kappa(A)\frac{\|\boldsymbol{r}\|}{\|\boldsymbol{b}\|}$$

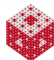
$$\Longrightarrow \quad \frac{1}{2.1932\times 10^6}\frac{7.1013\times 10^{-4}}{0.3341} \le \frac{\|\boldsymbol{x}-\boldsymbol{x}_*\|}{\|\boldsymbol{x}\|} \le 2.1932 \times 10^6 \frac{7.1013\times 10^{-4}}{0.3341}$$

$$\Longrightarrow \quad 9.6912 \times 10^{-10} \le \frac{\|\boldsymbol{x}-\boldsymbol{x}_*\|}{\|\boldsymbol{x}\|} \le 4.6617 \times 10^3.$$

This estimate ranges over 13 orders of magnitude and tells us that since our matrix is ill-conditioned, the residual is totally unreliable.

Note that our relative error was

$$\frac{\|\boldsymbol{x}-\boldsymbol{x}_*\|}{\|\boldsymbol{x}\|} = \frac{\|[-0.443, 1.00]^T - [1,-1]^T\|}{\|[1,-1]^T\|} = \frac{2.4662}{1.4142} = 1.7439.$$



Definition

A matrix A is called *sparse* if many of its entries are zero. Otherwise, A is called *dense* or *full*.

The MATLAB command `nnz` gives the number of nonzero entries in a matrix, and so we can compute

```
density = nnz(A)/prod(size(A))
sparsity = 1 - density
```

A sparse matrix has very small density (i.e., sparsity close to 1).

An $n \times n$ diagonal matrix has density $1/n$ and sparsity $1 - 1/n$.

Sparse Matrices in MATLAB

Since it would be a waste to store every single entry in a sparse matrix (many of which are zero), MATLAB has a special data structure to deal with sparse matrices.

The MATLAB command

```
S = sparse(i,j,x,m,n)
```

produces an $m \times n$ sparse matrix S whose nonzero entries (specified in the vector `x`) are located at the (i, j) positions (specified in the vectors `i` and `j` of row and column indices, respectively).

Sparse Matrices in MATLAB (cont.)

Example

```
i = [1 1 3], j = [1 2 3], x = [1 1 1]
S = sparse(i,j,x,3,3)
```

produces

$$S = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

A sparse matrix can be converted to the full format by `A=full(S)`, and a full matrix is converted to the sparse format by `S=sparse(A)`.

Algorithms for Sparse Linear Systems

There are two main categories of algorithms to solve linear systems $A\boldsymbol{x} = \boldsymbol{b}$ when A is sparse.

- *Direct methods*:
  - Especially for tridiagonal or banded systems (see below). Done with a custom implementation of Gaussian elimination.
  - MATLAB's built-in solvers (such as the backslash solver) work with the sparse matrix format.
- *Iterative methods* (discussed in detail in MATH 477):
  - Stationary ("classical") methods such as *Jacobi*, *Gauss-Seidel* or *SOR* iteration.
  - Krylov subspace methods such as *conjugate gradient* or *GMRES*. MATLAB has special routines for these and many others.

An efficient linear solver will always use as much structure of A as possible.

Jacobi Iteration

Example

For the system

$$\begin{aligned} 2x_1 + x_2 &= 6 \\ x_1 + 2x_2 &= 6 \end{aligned}$$

the Jacobi method looks like

$$\begin{aligned} x_1^{(k)} &= \left(6 - x_2^{(k-1)}\right)/2 \\ x_2^{(k)} &= \left(6 - x_1^{(k-1)}\right)/2. \end{aligned}$$

We start with an initial guess $[x_1^{(0)}, x_2^{(0)}]^T$ and then iterate to improve the answer.

Jacobi Iteration (cont.)

In general,

Jacobi iteration

Let $\boldsymbol{x}^{(0)}$ be an arbitrary initial guess

for $k = 1, 2, \ldots$

for $i = 1 : m$

$$x_i^{(k)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k-1)} - \sum_{j=i+1}^{m} a_{ij} x_j^{(k-1)} \right) / a_{ii}$$

end

end

Gauss-Seidel Iteration

Example

In order to improve the Jacobi method we notice that the value of $x_1^{(k-1)}$ used in the second equation of the example above is actually outdated since we already computed a newer version, $x_1^{(k)}$, in the first equation.
Therefore, we might consider

$$
\begin{aligned}
x_1^{(k)} &= \left(6 - x_2^{(k-1)}\right)/2 \\
x_2^{(k)} &= \left(6 - x_1^{(k)}\right)/2
\end{aligned}
$$

instead.
This is known as the Gauss-Seidel method.

Gauss-Seidel Iteration (cont.)

The general algorithm is of the form

Gauss-Seidel iteration

Let $\boldsymbol{x}^{(0)}$ be an arbitrary initial guess

for $k = 1, 2, \ldots$

for $i = 1 : m$

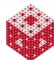
$$x_i^{(k)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{m} a_{ij} x_j^{(k-1)} \right) / a_{ii}$$

end

end

Classical Iterative Solvers

- The Jacobi or Gauss-Seidel method are conceptually very easy to understand, but are not very useful for solving linear systems.
- For example, MATLAB does not have any special code for them.
- They are useful, however, in preconditioning given linear systems for use with other — more powerful — iterative solvers.
- They are also frequently used as preconditioners in *domain decomposition* methods.



Example

The matrix (`A = gallery('poisson',3); full(A)`)

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

is a typical sparse matrix that arises in the finite difference solution of (partial) differential equations.

Note that all its nonzero entries are close to the diagonal, i.e., it is *banded* (see next slide). It is even *block tridiagonal*.

Definition

A matrix A is called banded if its nonzero entries are all located on the main diagonal as well as neighboring sub- and super-diagonals.

We can determine the bandwidth of A via

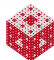
```
[i,j] = find(A)    % finds indices of nonzero entries
bandwidth = max(abs(i-j))
```

Example

The finite difference matrix of the previous slide has bandwidth 3.

An $n \times n$ diagonal matrix has bandwidth zero.

We will see banded matrices in the context of *spline interpolation*.



Tridiagonal Matrices

A matrix whose upper and lower bandwidth are both 1 is called tridiagonal. Tridiagonal matrices arise, e.g., when working with *splines*, *finite element*, or *finite difference methods*.

A tridiagonal (or more general banded) matrix is usually given by specifying its diagonals.

The MATLAB command `A = gallery('tridiag',a,b,c)` defines the matrix

$$A = \begin{bmatrix} b_1 & c_1 & 0 & \dots & 0 \\ a_1 & b_2 & c_2 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & a_{n-2} & b_{n_1} & c_{n-1} \\ 0 & \dots & 0 & a_{n-1} & b_n \end{bmatrix}$$

Note that the vectors $\boldsymbol{a}$ and $\boldsymbol{c}$ specifying the sub- and superdiagonals, respectively, have one entry fewer than the diagonal $\boldsymbol{b}$.

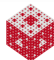
Other ways to create tridiagonal matrices in MATLAB

- In dense matrix format:
  - The command `diag` produces a diagonal matrix, i.e., if $\boldsymbol{a}$ is a vector of length n, then `A = diag(a)` produces

$$A = \begin{bmatrix} a_1 & 0 & \dots & 0 \\ 0 & a_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & a_n \end{bmatrix}.$$

    Note that `a = diag(A)` can also be used to return the diagonal of the matrix A in the vector $\boldsymbol{a}$.
  - A tridiagonal matrix T with subdiagonal $\boldsymbol{a}$, main diagonal $\boldsymbol{b}$, and superdiagonal $\boldsymbol{c}$ is given by

```
T = diag(a,-1) + diag(b,0) + diag(c,1)
```



Other ways to create tridiagonal matrices in MATLAB (cont.)

- In sparse matrix format:
  - An $n \times n$ tridiagonal matrix T with subdiagonal $\boldsymbol{a}$, main diagonal $\boldsymbol{b}$, and superdiagonal $\boldsymbol{c}$ is given by

    ```
    T = spdiags([a,b,c],[-1 0 1],n,n)
    ```
  - `spdiags` can also be used to extract the diagonals from a sparse matrix.
- In both cases, MATLAB's built-in backslash operator will solve the linear system $\mathsf{T}\boldsymbol{x} = \boldsymbol{d}$ efficiently using a direct method.

Solving a tridiagonal system in MATLAB (see `tridisolve.m`)

As for general systems $A\boldsymbol{x} = \boldsymbol{d}$ we use the usual two-phase approach.

- Gaussian elimination to convert to upper triangular form:

```
x = d;
n = length(x);
for j = 1:n-1
   mu = a(j)/b(j);                   % multipliers
   b(j+1) = b(j+1) - mu*c(j);   % diagonal elements
   x(j+1) = x(j+1) - mu*x(j);   % rhs
end
```

  Note that the superdiagonal elements are not touched, and that we did not use any pivoting.
  It can be shown that if A is diagonally dominant, i.e., each diagonal element is greater than the sum of the (absolute values of the) off-diagonal elements, then pivoting is not needed.

Solving a tridiagonal system in MATLAB (cont.)

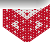
- Back substitution:

```
x(n) = x(n)/b(n);
for j = n-1:-1:1
    x(j) = (x(j)-c(j)*x(j+1))/b(j);
end
```

The code from this and the previous slide is combined in the function `tridisolve` from [NCM].

Remark

Recall that standard Gaussian elimination requires $\mathcal{O}(n^3)$ operations, and back substitution $\mathcal{O}(n^2)$. The specialized code `tridisolve` *requires only $\mathcal{O}(n)$ operations (see* `TridisolveDemo.m`*).*



A Mathematical Model for the Internet

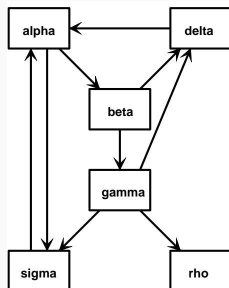
The following is from Sect. 2.11 of [NCM] (which is based on [Page *et al.*]).
Start with a connectivity matrix G of zeros and ones such that

$$g_{ij} = 1 \qquad \text{if page } \#j \text{ links to page } \#i.$$

G is $n \times n$, where — for the entire internet — n is huge, i.e., $\mathcal{O}(10^{12})$ (see this Google blog).

Example

Consider a tiny web consisting of only $n = 6$ webpages:



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

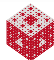
A Mathematical Model for the Internet (cont.)

In order to simulate web browsing, we use as a mathematical model a *random walk* or *Markov chain* with transition matrix A such that

$$a_{ij} = \begin{cases} \frac{pg_{ij}}{c_j} + \delta, & c_j \neq 0 \\ 1/n, & c_j = 0 \end{cases}$$

where a_{ij} denotes the probability of someone going to page $\#i$ when they're visiting page $\#j$, and

- p is the probability that an existing link is followed (typical value $p = 0.85$). Then $1 - p$ is the probability that a random page is visited instead.
- $c_j = \sum_{j=1}^{n} g_{ij}$, the column sums of G (called the *out-degree* of page $\#j$, i.e., how many pages the j-th page links to).
- $\delta = (1 - p)/n$ is the (tiny) probability of going to a particular random page.



A Mathematical Model for the Internet (cont.)

Example

$$A = \begin{bmatrix} 0.0250 & 0.0250 & 0.0250 & 0.8750 & 0.1667 & 0.8750 \\ 0.4500 & 0.0250 & 0.0250 & 0.0250 & 0.1667 & 0.0250 \\ 0.0250 & 0.4500 & 0.0250 & 0.0250 & 0.1667 & 0.0250 \\ 0.0250 & 0.4500 & 0.3083 & 0.0250 & 0.1667 & 0.0250 \\ 0.0250 & 0.0250 & 0.3083 & 0.0250 & 0.1667 & 0.0250 \\ 0.4500 & 0.0250 & 0.3083 & 0.0250 & 0.1667 & 0.0250 \end{bmatrix}, \quad \boldsymbol{\lambda} = \begin{bmatrix} 1.0000 \\ -0.5867 \\ -0.1341 + 0.3882i \\ -0.1341 - 0.3882i \\ 0.1465 \\ 0.0000 \end{bmatrix}$$

All our transition matrices have

- positive entries, i.e., $a_{ij} > 0$ for all i, j,
- column sums equal to one, i.e., $\sum_{i=1}^{n} a_{ij} = 1$ for all j,
- their largest eigenvalue equal to one, i.e., $\lambda_1 = 1$.

For such matrices the *Perron-Frobenius theorem* ensures that

$$A\boldsymbol{x} = \boldsymbol{x} \quad \Longleftrightarrow \quad (A - I)\boldsymbol{x} = \boldsymbol{0}$$

has a unique solution $\boldsymbol{x}$ such that $\sum_{i=1}^{n} x_i = 1$. The vector $\boldsymbol{x}$ is called the *state vector* of A. It is also Google's page rank vector.

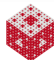
A Mathematical Model for the Internet (cont.)

In linear algebra jargon we are trying to find the eigenvector $\boldsymbol{x}$ associated with the maximum eigenvalue $\lambda_1 = 1$.
The simplest algorithm for doing this (and for huge matrices, such as the Google matrix, the only feasible method) is:

Power iteration

Initialize $\boldsymbol{x}^{(0)}$ with arbitrary nonzero vector (e.g., $\boldsymbol{x}^{(0)} = [\frac{1}{n}, \ldots, \frac{1}{n}]$)
for $k = 1, 2, \ldots$
$\quad \boldsymbol{x}^{(k)} = \mathsf{A}\boldsymbol{x}^{(k-1)}$
end

Run the MATLAB script `TinyWeb.m` to see the example of this section worked through.
The [NCM] program `surfer` can be used to compute page ranks starting at any URL. Note that this program is a bit buggy and may even crash MATLAB.



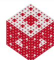
References I

J. W. Demmel.
Applied Numerical Linear Algebra.
SIAM, Philiadelphia, 1997.

G. H. Golub and C. Van Loan.
Matrix Computations.
Johns Hopkins University Press (3rd ed.), Baltimore, 1996.

C. D. Meyer.
Matrix Analysis and Applied Linear Algebra .
SIAM, Philadelphia, 2000.
Also http://www.matrixanalysis.com/.

C. Moler.
Numerical Computing with MATLAB.
SIAM, Philadelphia, 2004.
Also http://www.mathworks.com/moler/.



References II

G. Strang.
Introduction to Linear Algebra.
Wellesley-Cambridge Press (3rd ed.), Wellesley, MA, 2003.

G. Strang.
Linear Algebra and Its Applications.
Brooks Cole (4th ed.), 2005.

L. N. Trefethen and D. Bau, III.
Numerical Linear Algebra.
SIAM, Philadelphia, 1997.

L. Page, S. Brin, R. Motwani, and T. Winograd.
The PageRank Citation Ranking: Bringing Order to the Web.
http://ilpubs.stanford.edu:8090/422/

M. du Sautoy.
The Story of Maths — BBC The Genius of the East.
click here

References III

G. Strang.
Video lectures of Gilbert Strang teaching MIT's Linear Algebra 18.06.
http://ocw.mit.edu/OcwWeb/Mathematics/18-06Spring-2005/VideoLectures/index.htm