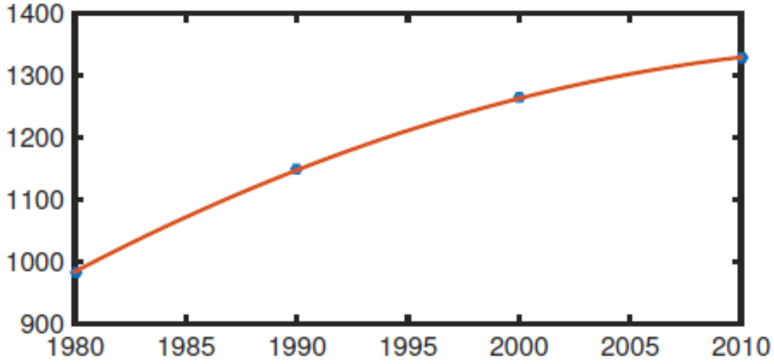
Chapter 2

Square linear systems: $Ax = b$



Square linear systems: objectives

- Where these systems may arise
- Efficient representation in MATLAB
- Learn increasingly sophisticated algorithms to solve these systems
- Learn tools to measure the performance of these algorithms
- Learn to recognize where difficulties may arise in solving linear systems
- Learn to recognize where structure of the problem can be exploited for fast and accurate solutions

2.1

Polynomial interpolation

- The idea of interpolation is determine a function that passes through, or recovers, given data
- Say we have (t_i, y_i) for $i = 1,2, \ldots, n$.
- We assume that no two t_i are the same
- If we consider putting a polynomial through this data, we choose an n-1 degree polynomial because there are n constants to find, which matches the number of data points available:

$$f(t) = a_1 t^{n-1} + a_2 t^{n-2} + \cdots + a_{n-1} t + a_n.$$

- To make this pass through the data, enforce $y_i = f(t_i)$ at each i

Polynomial interpolation

- Enforcing $y_i = f(t_i)$ at each i gives →
- The unknowns are the coefficients a_i
- This is n equations and n unknowns, and the a_i appear linearly: linear system!

$$
\begin{gathered}
t_1^{n-1}a_1 + t_1^{n-2}a_2 + \cdots + t_1 a_{n-1} + a_n = y_1 \\
t_2^{n-1}a_1 + t_2^{n-2}a_2 + \cdots + t_2 a_{n-1} + a_n = y_2 \\
t_3^{n-1}a_1 + t_3^{n-2}a_2 + \cdots + t_3 a_{n-1} + a_n = y_3 \\
\vdots \\
t_n^{n-1}a_1 + t_n^{n-2}a_2 + \cdots + t_n a_{n-1} + a_n = y_n.
\end{gathered}
$$

Polynomial interpolation

- It is convenient to write the system in matrix-vector form →
- The (known) matrix has a special name: Vandermonde matrix V
- The right hand side (rhs) vector y is known
- The unknown vector a must satisfy $Va = y$

$$\begin{bmatrix} t_1^{n-1} & t_1^{n-2} & \cdots & t_1 & 1 \\ t_2^{n-1} & t_2^{n-2} & \cdots & t_2 & 1 \\ t_3^{n-1} & t_3^{n-2} & \cdots & t_3 & 1 \\ \vdots & \vdots & & \vdots & \vdots \\ t_n^{n-1} & t_n^{n-2} & \cdots & t_n & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}$$

An example: Fitting population data

- Census data is often measured in 10 year intervals.
- How to get population estimates in other years?
- May affect policy decisions ranging from social programs like health care to education, business planning, and many other things
- We can apply interpolation to available data to compute values in intervening years
- Our data is 4 populations for China at 10 year intervals
- The idea is to put a cubic polynomial through this data: we have to find the 4 coefficients such that this happens.

Population example

- Input years and population →
- The rhs vector y is now pop →
- The Vandermonde matrix V has columns in decreasing powers of t
- The method works better for measuring t as years since 1980 →

```
year = (1980:10:2010)'
pop = [
     984.736
     1148.364
     1263.638
     1330.141
]
```

[Example 2.1.1]

```
y = pop;
```

```
V = zeros(4,4);
for i = 1:4
    V(i,:) = [t(i)^3 t(i)^2 t(i) 1];
end
V
```

```
V =
           0           0           0           1
        1000         100          10           1
        8000         400          20           1
       27000         900          30           1
```

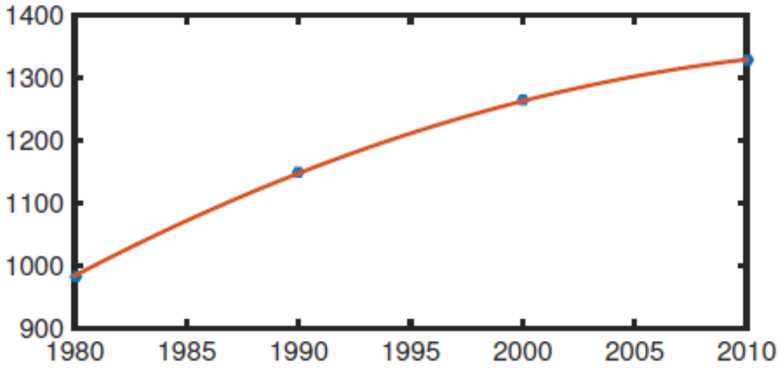
Population example

- Solve linear system for coefficients a using the backslash (\)
- Use the command polyval to calculate polynomial
- Plot the populations (dots) and cubic polynomial fit (red curve)
- Note how curve passes through the data

```
a = V \ y
```

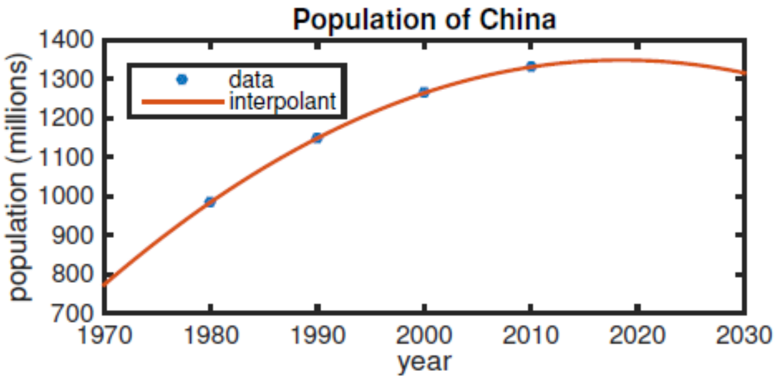
```
a =
    -0.0001
    -0.2397
    18.7666
   984.7360
```

```
tt = linspace(0,30,300)';
yy = polyval(a,tt);
plot(1980+tt,yy)
```



Population example

- A better plot with labels
- Using the cubic polynomial between the data points is interpolation: solid results
- Using the cubic outside the data interval is extrapolation: use with care!!



2.2

Computing with matrices in ~~MATLAB~~ Julia

- MATLAB originally stood for “Matrix Laboratory”
- It is particularly easy to use MATLAB for this kind of computing
- Section 2.2 contains a tutorial on creating and manipulating matrices
- Try it!

Needed operations for matrices

- Consider the j-th column of the identity matrix:

$e_1 = [1\ 0\ 0 \ldots 0]^{\mathrm{T}}$, $e_2 = [0\ 1\ 0 \ldots 0]^{\mathrm{T}}$, etc, $I = [e_1\ e_2 \ldots e_n] = \begin{bmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \end{bmatrix}$

- Then if $A = [a_1\ \ a_2 \ldots a_n]$ shows the columns of A…
- Right-multiplying by e_j gives the j-th column of A: $Ae_j = a_j$
- Left multiplying by e_i^T gives the i-th row of A: $e_i^T A = [A_{i1} A_{i2} \ldots A_{in}]$
- Combining the two can pick out a single element: $e_i^T A e_j = A_{ij}$
- We will use these soon!

Linear Systems

- Consider the system
- The compact form is $\boldsymbol{Ax} = \boldsymbol{b}$
- The exact solution from theory is $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$ provided that $\boldsymbol{A}$ is non-singular, i.e., $\boldsymbol{A}^{-1}$ exists
- This is great, but for computing, we almost *never* want to solve this way!!

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
&\vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n.
\end{aligned}
$$

Linear Systems

- A convenient and powerful capability for solving $\boldsymbol{Ax} = \boldsymbol{b}$ in MATLAB is \
- An example is at left
- Is the answer good? Look at residual
- Small residual indicates a good answer here: how much we missed exactly satisfying the equation is close to `eps`

[Example 2.2.1]

```
A = magic(3)
b = [1;2;3];
x = A\b
```

```
A =
     8     1     6
     3     5     7
     4     9     2
x =
    0.0500
    0.3000
    0.0500
```

```
residual = b - A*x
```

```
residual =
   1.0e-15 *
         0
         0
    0.4441
```

2.3)

Linear Systems

- Care and understanding is still needed though
- Note that even for singular A, one still gets an answer from \

```
A = [0 1; 0 0];
b = [1;2];
x = A\b
```

```
Warning: Matrix is singular to working precision.
x =
  -Inf
   Inf
```

- We will understand why later
- If you see a warning, investigate!!

[Example 2.3.2]

Triangular Systems

- Let's begin with simple systems to understand algorithms for solving $\boldsymbol{Ax} = \boldsymbol{b}$
- Consider *lower triangular* systems where elements above a_{ii} are zero
- Said another way, $a_{ij} = 0, j > i$
- Solve by starting with $x_1 = \frac{8}{4} = 2$
- Continue with $x_2 = \frac{5-(3)(2)}{-1} = 1$
- Then find x_3, then x_4, to get solution
- This is called *forward substitution*

$$\begin{bmatrix} 4 & 0 & 0 & 0 \\ 3 & -1 & 0 & 0 \\ -1 & 0 & 3 & 0 \\ 1 & -1 & -1 & 2 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 8 \\ 5 \\ 0 \\ 1 \end{bmatrix}$$

$$\mathbf{x} = \begin{bmatrix} 2 \\ 1 \\ 2/3 \\ 1/3 \end{bmatrix}$$

Triangular Systems

- For the more general system $\boldsymbol{Lx} = \boldsymbol{b}$ where $\boldsymbol{L}$ is lower triangular, we can still do forward substitution
- L_{ij} are elements of $\boldsymbol{L}$
- No trouble here unless $L_{ii} = 0$, then we can divide by it
- Theorem 2.1: If at least one of the $L_{ii} = 0$, then $\boldsymbol{L}$ is singular
- How to implement in MATLAB?

$$x_1 = \frac{b_1}{L_{11}}$$

$$x_2 = \frac{b_2 - L_{21}x_1}{L_{22}}$$

$$x_3 = \frac{b_3 - L_{31}x_1 - L_{32}x_2}{L_{33}}$$

$$x_4 = \frac{b_4 - L_{41}x_1 - L_{42}x_2 - L_{43}x_3}{L_{44}}.$$

L is nonsingular $\Longleftrightarrow$ $L_{ii} \neq 0, \ \forall i$

Triangular Systems

- At right is an implementation using two nested `for` loops
- For the i loop, we work with x(i)
- For the j loop, we step through the columns $j = 1, \ldots, i$
- For $i = 1$, there is only one term (b(1)); for larger i, previous x's are used
- As a user, you can remove the semi-colons and see how the loops progress
- We can be a little more elegant however

```
for i = 1:4
    x(i) = b(i);
    for j = 1:i-1
        x(i) = x(i) - L(i,j)*x(j);
    end
    x(i) = x(i)/L(i,i);
end
```

Triangular Systems

```
for i = 1:4
    x(i) = b(i);
    for j = 1:i-1
        x(i) = x(i) - L(i,j)*x(j);
    end
    x(i) = x(i)/L(i,i);
end
```

- At right is two nested `for` loops, consider $i = 4$
- Considering the last unknown, we end up with

$$b_4 - (L_{41}x_1 + L_{42}x_2 + L_{43}x_3) = b_4 - \begin{bmatrix} L_{41} & L_{42} & L_{43} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

- The last vector product is a dot product, and we can eliminate the inner loop:

```
for i = 1:n
  x(i) = ( b(i) - L(i,1:i-1)*x(1:i-1) ) / L(i,i);
end
```

- Function next….

Lower Triangular Systems

- Pass in **L** and **b**; return solution **x**
- Note comments; they appear in `help forwardsub`
- Detect system size **n** automatically
- Single loop implementation
- No test for singular **L**

```
function x = forwardsub(L,b)
% FORWARDSUB   Forward substitution for lower-tri
% Input:
%    L     lower triangular square matrix (n by n)
%    b     right-hand side vector (n by 1)
% Output:
%    x     solution of Lx=b (n by 1 vector)

n = length(L);
x = zeros(n,1);
for i = 1:n
  x(i) = ( b(i) - L(i,1:i-1)*x(1:i-1) ) / L(i,i);
end
```

Lower Triangular Example

- Easy to make a lower triangular matrix
- Make up rhs
- Solve system by calling function `forwardsub`
- Compute the residual to check the solution; it is the size of ϵ_M

```
A = magic(5)
L = tril(A)
```

```
A =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
L =
    17     0     0     0     0
    23     5     0     0     0
     4     6    13     0     0
    10    12    19    21     0
    11    18    25     2     9
```

```
b = ones(5,1);
x = forwardsub(L,b)
```

```
x =
    0.0588
   -0.0706
    0.0914
   -0.0228
   -0.0684
```

```
b - L*x
```

```
ans =
   1.0e-15 *
         0
         0
         0
         0
   -0.2220
```

Upper Triangular Systems

- For the upper triangular system $\boldsymbol{Ux} = \boldsymbol{b}$ where $\boldsymbol{U}$ has $U_{ij} = 0, j < i$
- That is elements below the main diagonal are zero this time
- Solve for last variable first this time and work backward
- No trouble here unless $U_{ii} = 0$, then we cannot divide by it
- Theorem 2.1: If at least one of the $U_{ii} = 0$, then $\boldsymbol{U}$ is singular
- Implementation similar

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ 0 & U_{22} & U_{23} & U_{24} \\ 0 & 0 & U_{33} & U_{34} \\ 0 & 0 & 0 & U_{44} \end{bmatrix} \mathbf{x} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

$$x_4 = \frac{b_4}{U_{44}}$$

$$x_3 = \frac{b_3 - U_{34}x_4}{U_{33}}$$

$$x_2 = \frac{b_2 - U_{23}x_3 - U_{24}x_4}{U_{22}}$$

$$x_1 = \frac{b_1 - U_{12}x_2 - U_{13}x_3 - U_{14}x_4}{U_{11}}$$

Upper Triangular Systems

- Pass in **U** and **b**; return solution **x**
- Note comments; they appear in `help backsub`
- Detect system size **n** automatically
- Single loop counts down now
- No test for singular **U**

```
function x = backsub(U,b)
% BACKSUB   Backward substitution for upper-triang
% Input:
%    U     upper triangular square matrix (n by n)
%    b     right-hand side vector (n by 1)
% Output:
%    x     solution of Ux=b (n by 1 vector)

n = length(U);
x = zeros(n,1);
for i = n:-1:1
  x(i) = ( b(i) - U(i,i+1:n)*x(i+1:n) ) / U(i,i);
end
```

Triangular Systems

- Let’s look at a made-up system where we know exact answer
- For $\beta = 2.2$ and $\alpha = 0.3$, there is no problem and the residual is the size of machine epsilon, $\texttt{eps}$
- However, we can create difficult systems with $\beta \gg \alpha$ with terrible residual!
- We’ll improve this later

$$\begin{bmatrix} 1 & -1 & 0 & \alpha-\beta & \beta \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

```
alpha = 0.3;
beta = 1e12;
U = eye(5) + diag([-1 -1 -1 -1],1);
U(1,[4 5]) = [ alpha-beta, beta ];
b = [alpha;0;0;0;1];

x = backsub(U,b);
err = x - x_exact
```

```
err =
   1.0e-04 *
   -0.4883
         0
         0
         0
         0
```

[Example 2.3.3]

2.4)

Gaussian elimination and LU factorization

- A general purpose algorithm for solving linear systems is *Gaussian elimination* (GE)
- We will use this algorithm to adapt it so that it yields an *LU facutorization*; this is just our first example of factorization
- We will further adapt the method to improve its stability and robustness a bit later
- You should have seen the basic GE method prior to this class
- We incorporate MATLAB in discussing the method here

GE example

- First, create a 4x4 linear system
- Then, we append the right hand side to the coefficient matrix to obtain the augmented matrix
- GE is then performed on this augmented matrix

```
Ab = [A, b]

Ab =
    2.0000         0    4.0000    3.0000    4.0000
   -4.0000    5.0000   -7.0000  -10.0000    9.0000
    1.0000   15.0000    2.0000   -4.5000   29.0000
   -2.0000         0    2.0000  -13.0000   40.0000
```

GE example

- We now want to make Ab(2:4,1)=0
- That is, elements below row 1 (R1) become zero in first column
- Compute multiplier mult, then do R2 <- R2- mult*R1
- Repeat this process for R3 and R4, recomputing mult for each row

```
mult = Ab(2,1)/Ab(1,1);
Ab(2,:) = Ab(2,:) - mult*Ab(1,:)
```

```
Ab =
    2.0000         0    4.0000    3.0000    4.0000
         0    5.0000    1.0000   -4.0000   17.0000
    1.0000   15.0000    2.0000   -4.5000   29.0000
   -2.0000         0    2.0000  -13.0000   40.0000
```

```
mult = Ab(3,1)/Ab(1,1);
Ab(3,:) = Ab(3,:) - mult*Ab(1,:);
mult = Ab(4,1)/Ab(1,1);
Ab(4,:) = Ab(4,:) - mult*Ab(1,:);
Ab
```

```
Ab =
     2     0     4     3     4
     0     5     1    -4    17
     0    15     0    -6    27
     0     0     6   -10    44
```

GE example

- We now want to make Ab(3:4,2)=0
- That is, elements below R2 become zero in 2nd column
- Compute multipliers mult, for R3 and R4, recomputing mult for each row, in a loop

```
for i = 3:4
    mult = Ab(i,2)/Ab(2,2);
    Ab(i,:) = Ab(i,:) - mult*Ab(2,:);
end
Ab
```

```
Ab =
     2     0     4     3     4
     0     5     1    -4    17
     0     0    -3     6   -24
     0     0     6   -10    44
```

GE example

- We now want to make Ab(4:4,3)=0
- That is, repeat the process for column 3, below R3
- This time loop only runs once, but can still be loop structure
- The first four columns are a 4x4 upper triangular matrix U, and the last column is a rhs z so that Ux=z
- Now use back substitution!

```
for i = 4
    mult = Ab(i,3)/Ab(3,3);
    Ab(i,:) = Ab(i,:) - mult*Ab(3,:);
end
Ab
```

```
Ab =
     2     0     4     3     4
     0     5     1    -4    17
     0     0    -3     6   -24
     0     0     0     2    -4
```

```
U = Ab(:,1:4)
z = Ab(:,5)
```

GE example

- Solve for x with back substitution
- Error is quite small!

```
x = backsub(U,z)
```

```
x =
    -3
     1
     4
    -2
```

```
b - A*x
```

```
ans =
     0
     0
     0
     0
```

[Example 2.4.1]

- This is a general strategy to solve systems
- How can we use this to get a factorization? And why?

GE or algebra rules

- There are three rules for manipulating systems (augmented matrix) and not changing the answer. We can:
  1. Switch two rows: Ri $\leftrightarrow$ Rj
  2. Multiply a row by a scalar: c Ri $\rightarrow$ Ri $c \neq 0$!!
  3. Form a linear combination of two rows and replace one of them with the linear combo: a Ri + b Rj $\rightarrow$ Rj $b \neq 0$!!
- We use these for GE, particularly the last, but we can write them as matrix vector operations
- This is a different take than linear algebra class; revisit our example

GE matrix ops

- We now want to make Ab(2:4,1)=0
- We needed mult = -2, then do R2 <- R2- mult*R1
- Let e_j be jth column of thet identity matrix (col vector)
- Multiplication on the left with e_j^T gives jth row of A
- We can write $e_2^T - (-2)e_1^T$ to subtract (-2)*R1 from R2 and obtain a zero element in A_{21}
- Matrix form!

```
mult = Ab(2,1)/Ab(1,1);
Ab(2,:) = Ab(2,:) - mult*Ab(1,:)
```

```
Ab =
    2.0000         0    4.0000    3.0000    4.0000
         0    5.0000    1.0000   -4.0000   17.0000
    1.0000   15.0000    2.0000   -4.5000   29.0000
   -2.0000         0    2.0000  -13.0000   40.0000
```

$$\begin{bmatrix} \mathbf{e}_1^T \\ \mathbf{e}_2^T - 2\mathbf{e}_1^T \\ \mathbf{e}_3^T \\ \mathbf{e}_4^T \end{bmatrix} \mathbf{A} = \left(\mathbf{I} - \begin{bmatrix} 0\mathbf{e}_1^T \\ -2\mathbf{e}_1^T \\ 0\mathbf{e}_1^T \\ 0\mathbf{e}_1^T \end{bmatrix} \right) \mathbf{A} = \left(\mathbf{I} + 2 \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \mathbf{e}_1^T \right) \mathbf{A}.$$

$e_2 e_1^T$

$$I + 2\begin{bmatrix}0\\1\\0\\0\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 0\end{bmatrix}$$

elementary matrix L_1

$$I + 2\begin{bmatrix}0&0&0&0\\1&0&0&0\\0&0&0&0\\0&0&0&0\end{bmatrix} = \begin{bmatrix}1&&&\\2&1&&\\&&1&\\&&&1\end{bmatrix}$$

$$L_1 A = \begin{bmatrix}1&&&\\2&1&&\\&&1&\\&&&1\end{bmatrix}\begin{bmatrix}2&0&4&3\\-4&5&-7&-10\\1&15&2&-4.5\\-2&0&2&-13\end{bmatrix}$$

$$= \begin{bmatrix}2&0&4&3\\0&5&1&-4\\1&15&2&-4.5\\-2&0&2&-13\end{bmatrix}$$

$$L_1^{-1} = \begin{bmatrix}1&&&\\-2&1&&\\&&1&\\&&&1\end{bmatrix}$$

GE matrix ops

- Use matrix ops to make Ab(2:4,1)=0
- Elements below row 1 (R1) become zero in first column again
- Compute multiplier mult, then do R2 <- R2- mult*R1
- Repeat this process for R3 and R4, recomputing mult for each row

[Example 2.4.2]

```
A = [2 0 4 3 ; -4 5 -7 -10 ; 1 15 2 -4.5 ; -2 0 2 -13];
I = eye(4);
mult = A(2,1)/A(1,1);
L1 = I - mult*I(:,2)*I(:,1)';
A = L1*A
```

```
A =
    2.0000         0    4.0000    3.0000
         0    5.0000    1.0000   -4.0000
    1.0000   15.0000    2.0000   -4.5000
   -2.0000         0    2.0000  -13.0000
```

```
mult = A(3,1)/A(1,1);
L2 = I - mult*I(:,3)*I(:,1)';
A = L2*A;

mult = A(4,1)/A(1,1);
L3 = I - mult*I(:,4)*I(:,1)';
A = L3*A
```

```
A =
     2     0     4     3
     0     5     1    -4
     0    15     0    -6
     0     0     6   -10
```

Elementary Matrices for GE

- To change the A_{21} element, we used $\mathbf{I} + 2\mathbf{e}_2\mathbf{e}_1^T$
- Call that matrix L_{21}
- To change A_{31}, use $L_{31} = I - 0.5\cancel{e_1^T e_3}$ $e_1 e_3^T$
- To change A_{41}, use $L_{41} = I - (-1)\cancel{e_1^T e_4}$ $e_1 e_4^T$
- So, to change the first column under A_{11} , we can write steps in the matrix form: $A_1 = L_{41}L_{31}L_{21}A$
- We can also write $L_{j1} = I - m_{j1}e_j e_1^T$ and we had three different multipliers

Elementary Matrices, toward LU

- To change the second column, we use $L_{42}L_{32}$, with the right one done first, and multipliers m_{42} and m_{32}
- The cumulative operations are then $A_1 = L_{42}L_{32}L_{41}L_{31}L_{21}A$
- To change A_{43}, use $L_{43} = I - m_{43}e_4e_3^T$
- So, combining all of these gives us the upper triangular form we sought: $U = L_{43}L_{42}L_{32}L_{41}L_{31}L_{21}A$

$U = \hat{L}A$

- Note that applying these to the right side b will give z for $Ux = z$
- We get something useful for recovering A...

$A = LU$

$A = L_{21}^{-1}L_{31}^{-1}\cdots L_{43}^{-1}U$

Elementary Matrices, toward LU

- These elementary matrices L_{ij} and their inverses have remarkably useful properties
- Multiply two that differ only in sign together:

$$
\begin{aligned}
(\mathbf{I} + \alpha \mathbf{e}_i \mathbf{e}_j^T)(\mathbf{I} - \alpha \mathbf{e}_i \mathbf{e}_j^T) &= \mathbf{I} + \alpha \mathbf{e}_i \mathbf{e}_j^T - \alpha \mathbf{e}_i \mathbf{e}_j^T - \alpha^2 \mathbf{e}_i \mathbf{e}_j^T \mathbf{e}_i \mathbf{e}_j^T \\
&= \mathbf{I} - \alpha^2 \mathbf{e}_i (\mathbf{e}_j^T \mathbf{e}_i) \mathbf{e}_j^T = \mathbf{I},
\end{aligned}
$$

- This says that the two matrices on the left are inverses! So,

$$
(\mathbf{I} + \alpha \mathbf{e}_i \mathbf{e}_j^T)^{-1} = \mathbf{I} - \alpha \mathbf{e}_i \mathbf{e}_j^T,
$$

and think of α as a multiplier m_{ij}

- We then have $L_{ij}^{-1} = I + m_{ij} e_i e_j^T$, and we know all those inverses!

Elementary Matrices, toward LU

- We also need to know about the produce of all of them
- Multiply two general elementary matrices together:

$$(\mathbf{I} + \alpha \mathbf{e}_2 \mathbf{e}_1^T)(\mathbf{I} + \beta \mathbf{e}_3 \mathbf{e}_1^T) = \mathbf{I} + \alpha \mathbf{e}_2 \mathbf{e}_1^T + \beta \mathbf{e}_3 \mathbf{e}_1^T + \alpha\beta \mathbf{e}_2 \mathbf{e}_1^T \mathbf{e}_3 \mathbf{e}_1^T.$$

- Second term puts α in (2,1) location and β in (3,2) location
- Last term?

$$\begin{bmatrix} 1 & & & \\ \alpha & 1 & & \\ & & 1 & \\ & & & 1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ \beta & & 1 & \\ & & & 1 \end{bmatrix} = \begin{bmatrix} 1 & & & \\ \alpha & 1 & & \\ \beta & & 1 & \\ & & & 1 \end{bmatrix}$$

LU factorization

- The product of the inverses is then of the form

$$L = L_{21}^{-1} L_{31}^{-1} L_{32}^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ m_{21} & 1 & 0 \\ m_{31} & m_{32} & 1 \end{bmatrix}$$

- Those multipliers that we used to create zero in the columns can go right into the lower triangular matrix L!
- And, finally, we have

$$\boxed{LU = A}$$

- This shows that A can factored or factorized into a lower triangular L and upper triangular U
- Just our first instance of factorization…

Using LU factorization

- We can rewrite the system $Ax = b$ as $L(Ux) = b$
- Substitute to get $L(\underbrace{Ux}_{z}) = b$;
- then let $Ux = z$;
- Then $Lz = b$
- This suggests how to use LU:

1. Compute the factorization $LU = A$ using Gaussian elimination
2. Solve $Lz = b$ using forward substitution
3. Solve $Ux = z$ using backward substitution
4. For same A, repeat 2 and 3 for different b without step 1

$Lz = b$
$Ux = z$

$Ax_1 = b_1$
$Ax_2 = b_2$
$\vdots$

Function for LU

```
function [L,U] = lufact(A)
% LUFACT   LU factorization (demo only--not stable!)
% Input:
%    A     square matrix
% Output:
%    L,U   unit lower triangular and upper triangular such that LU=A

n = length(A);
L = eye(n);    % ones on diagonal

% Gaussian elimination
for j = 1:n-1
  for i = j+1:n
    L(i,j) = A(i,j) / A(j,j);    % row multiplier
    A(i,j:n) = A(i,j:n) - L(i,j)*A(j,j:n);
  end
end

U = triu(A);
```

LU example: compute factors

Example 2.8.
```
A = [2 0 4 3; -4 5 -7 -10; 1 15 2 -4.5; -2 0 2 -13];
[L,U] = lufact(A)
```

```
L =
    1.0000         0         0         0
   -2.0000    1.0000         0         0
    0.5000    3.0000    1.0000         0
   -1.0000         0   -2.0000    1.0000
U =
     2     0     4     3
     0     5     1    -4
     0     0    -3     6
     0     0     0     2
```

```
LtimesU = L*U
```

```
LtimesU =
    2.0000         0    4.0000    3.0000
   -4.0000    5.0000   -7.0000  -10.0000
    1.0000   15.0000    2.0000   -4.5000
   -2.0000         0    2.0000  -13.0000
```

```
A - LtimesU
```

```
ans =
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
```

- Lufact computes L and U
- Checking the product: looks like A
- Error shows that it works

LU example: solve a system

Example 2.8. `A = [2 0 4 3; -4 5 -7 -10; 1 15 2 -4.5; -2 0 2 -13];`
`[L,U] = lufact(A)`

```
L =
    1.0000         0         0         0
   -2.0000    1.0000         0         0
    0.5000    3.0000    1.0000         0
   -1.0000         0   -2.0000    1.0000
U =
     2     0     4     3
     0     5     1    -4
     0     0    -3     6
     0     0     0     2
```

```
b = [4;9;29;40];
z = forwardsub(L,b);
x = backsub(U,z)
```

```
x =
    -3
     1
     4
    -2
```

```
b - A*x
```

```
ans =
     0
     0
     0
     0
```

- Lufact computes L and U
- Forward solve to get z
- Backward solve for x

[Example 2.4.3]

Performance of Linear System Solutions

- Sections 2.5 and later are in Part 2