Determinants:

If the determinant of *A* is zero

- *A* is singular (A^{-1} does not exist)
- The system of equations has no unique solution

In Matlab the determinant of a matrix can found using function *det*.

```
>> A = [ 4 2; 2 1 ];
>> det(A)

ans = 0

>> A = [ 2 5 6; -4 5 1; 7 6 -2 ];
>> det(A)

ans = -391
```

Determinants close to zero are an indication of problem cases.

Condition Numbers:

The condition number of a matrix is defined as

cond(A) = $||A|| * ||A^{-1}||$

where $||A||$ is a "norm" of A

A "norm" is as measure of the "size" of a matrix. For example:

$$\left\| \begin{bmatrix} 1 & 5 & 2 \end{bmatrix} \right\|_e = \sqrt{1^2 + 5^2 + 2^2}$$

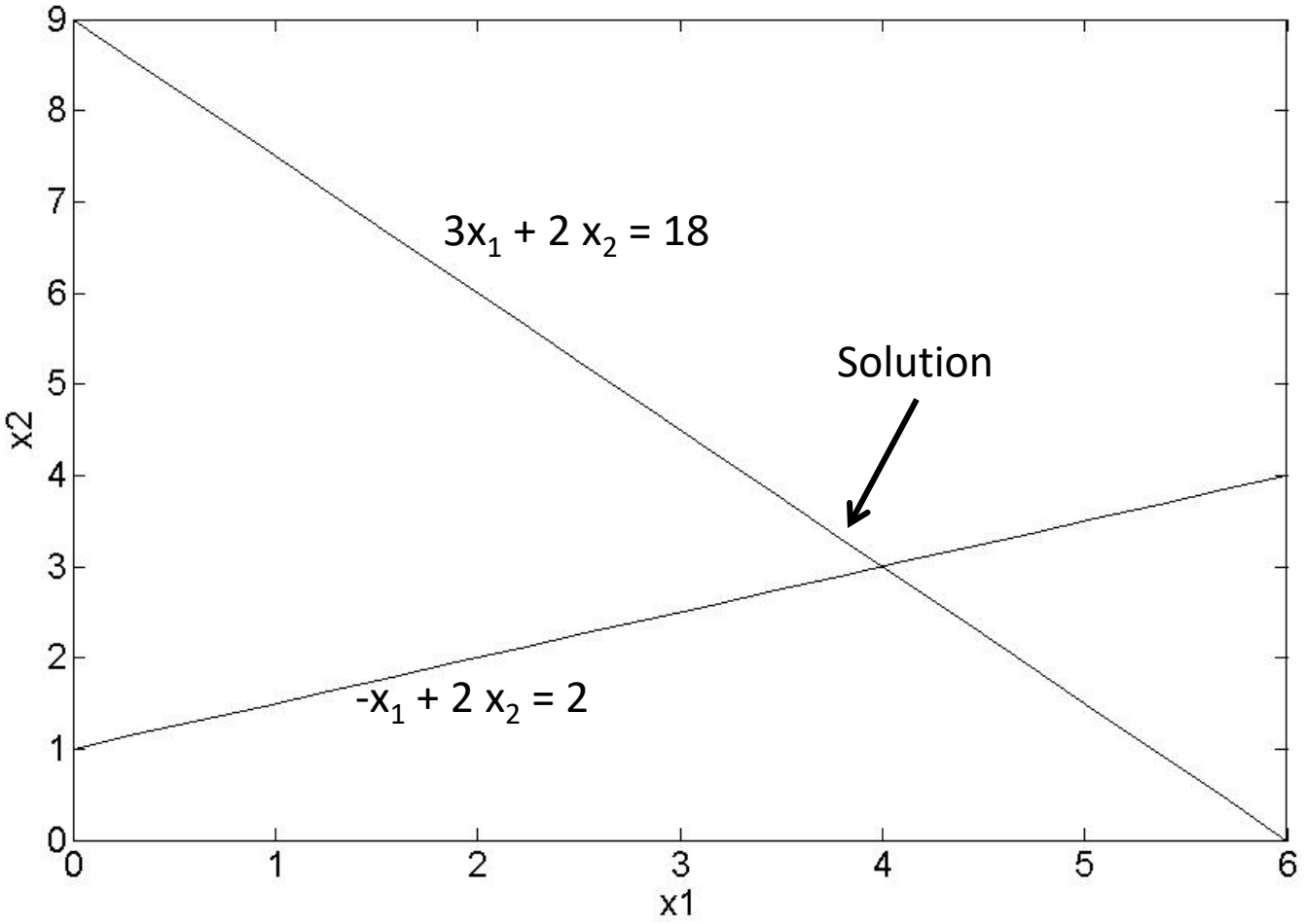
Multiple "norm" definitions exist. The "e" in the above equation indicates a "Euclidian norm". See text page 254 for more information.

Relatively high condition numbers (>> 1) are an indication of problem cases.

There is no hard and fast line.

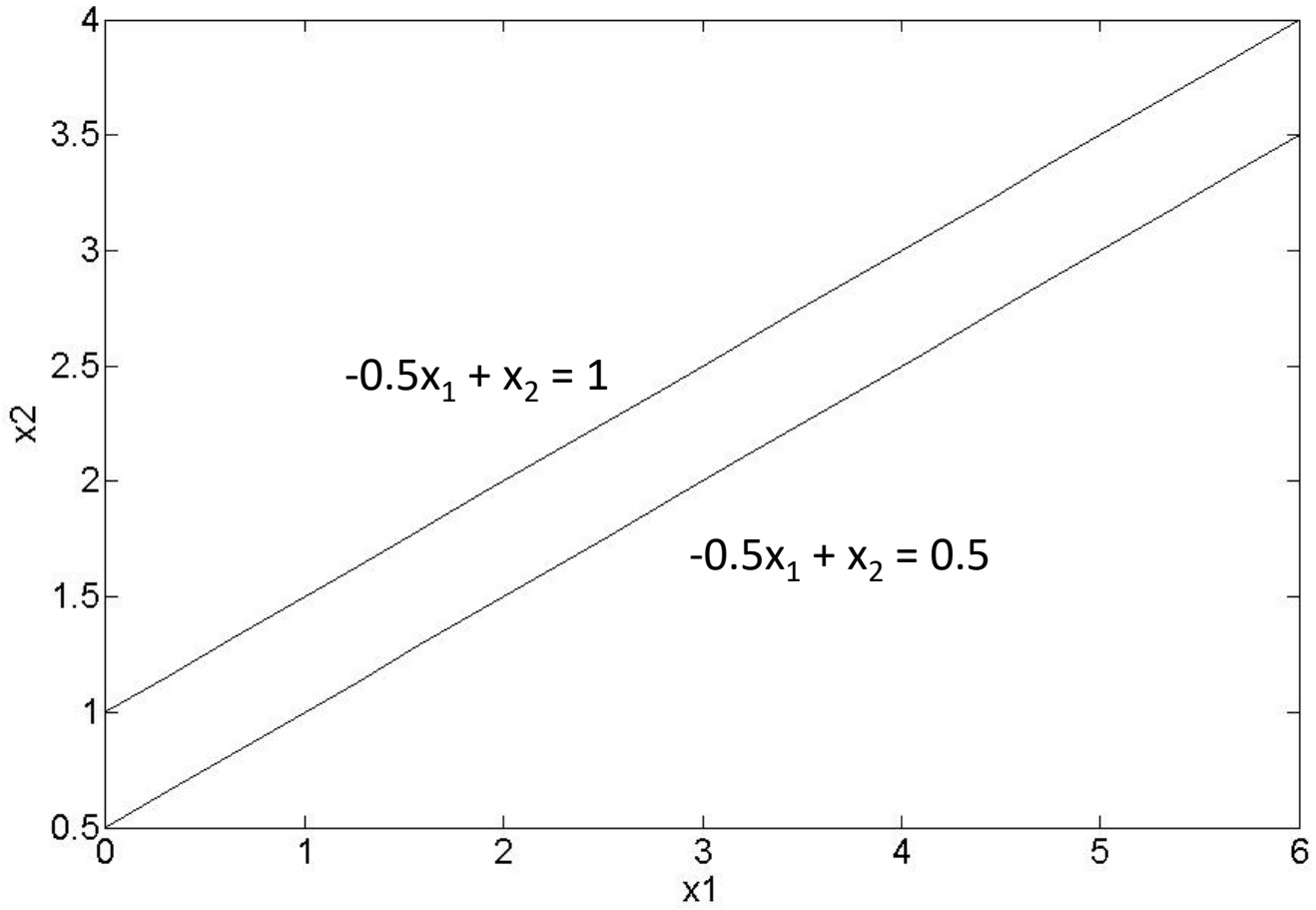
If the elements of A have t significant digits the elements of x (the solution) will have t – log(cond(A)) significant digits. Note: this rule is conservative.

Graphical representation of two simultaneous equations (ideal case):



cond(A) = 1.6404, det(A) = 8

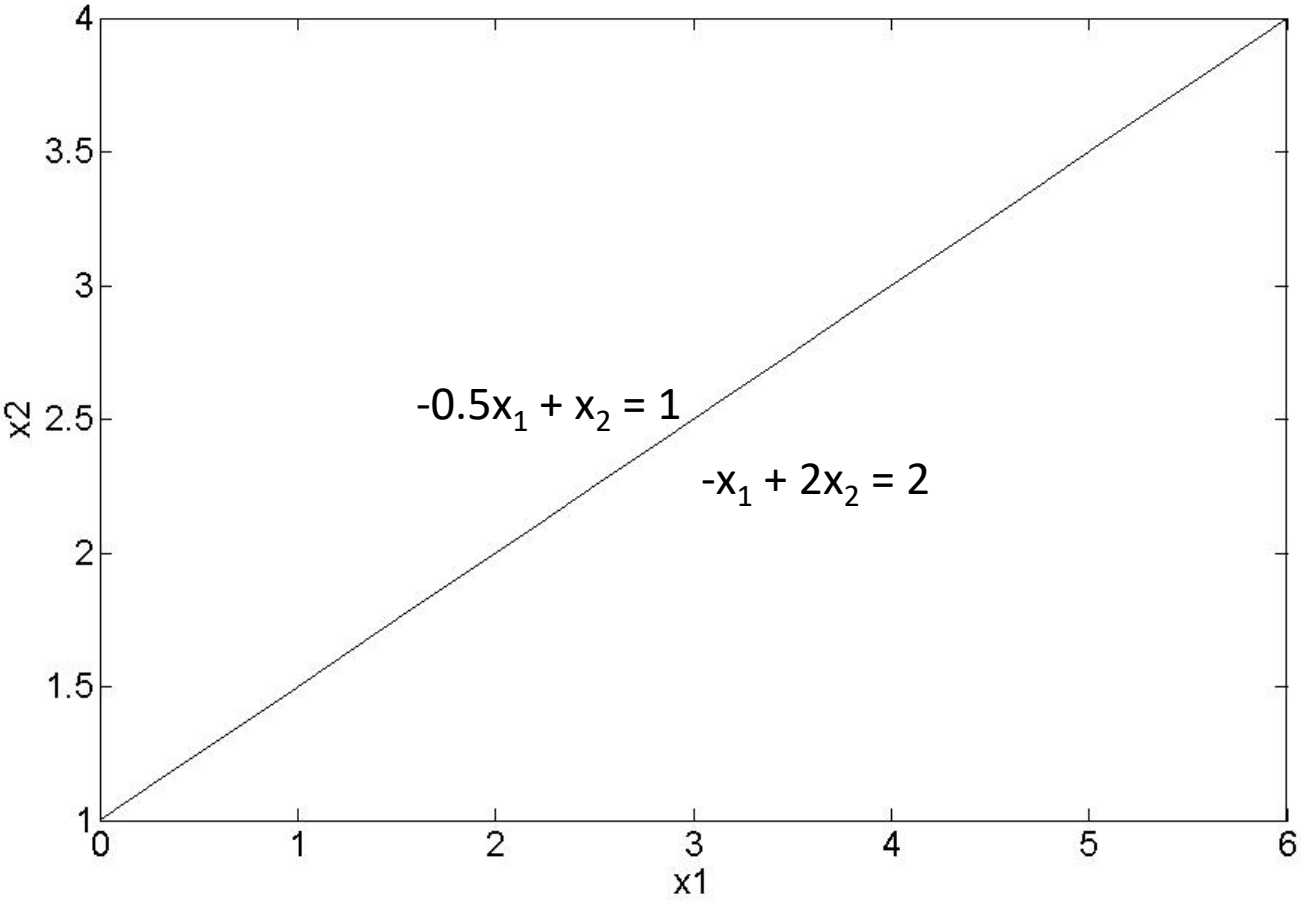
Graphical representation of two simultaneous equations (no solution):



The two equations are contradictory.

cond(A) = Inf, det(A) = 0

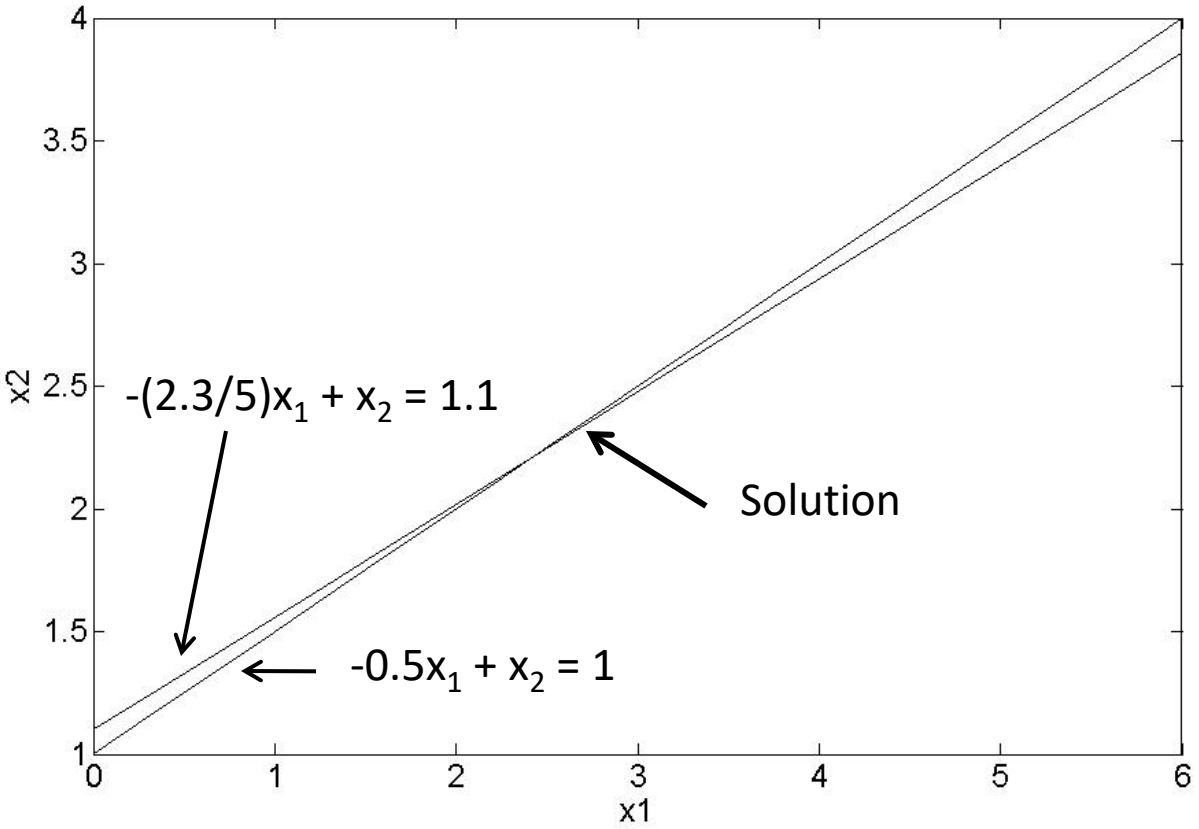
Graphical representation of two simultaneous equations (infinite solutions):



The two equations are not linearly independent (they give the same information).

cond(A) = 2.5176e+016, det(A) = 0

Graphical representation of two simultaneous equations (problem case):



A solution exists but it is very sensitive to small calculation errors.

The system is *ill conditioned*.

cond(A) = 61.5237, det(A) = 0.0400

Tests for ill conditioned systems:

Scale A so that the largest element in each row has a magnitude of 1. Then calculate the inverse. If A^{-1} has elements that are orders of magnitude greater than 1 it is likely that the system is ill conditioned.

Multiply A by A^{-1}. If the result is not close to the identity matrix it is likely that the system is ill conditioned.

Invert A^{-1}. If the result is not close to A it is likely that the system is ill conditioned.

Bottom Line:

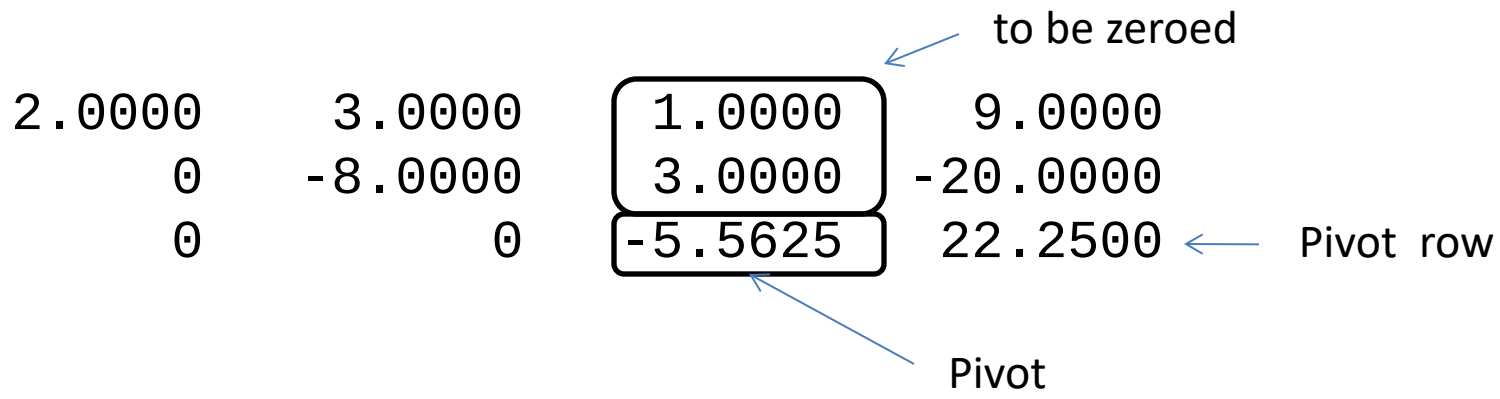
Be aware that solving a set of simultaneous equations will not always produce a good answer.

Check solutions obtained by multiplying A by x. The result should be very close to b but this will not always be the case.

Gauss- Jordan elimination: (** not in text ****)**

Forward elimination first used to make *A* upper triangular (exactly as for Gaussian elimination).

Backward elimination (same process but applied “backwards”) then used to zero all values above the diagonal.



```
>> P = C(3,3);
>> C(2,:) = C(2,:) - C(2,3)/P * C(3,:)
C =
    2.0000    3.0000    1.0000    9.0000
         0   -8.0000         0   -8.0000
         0         0   -5.5625   22.2500
>> C(1,:) = C(1,:) - C(1,3)/P * C(3,:)
C =
    2.0000    3.0000         0   13.0000
         0   -8.0000         0   -8.0000
         0         0   -5.5625   22.2500
>> P = C(2,2)
>> C(1,:) = C(1,:) - C(1,2)/P * C(2,:)
C =
    2.0000         0         0   10.0000
         0   -8.0000         0   -8.0000
         0         0   -5.5625   22.2500
```

Finally each row is *normalized* by dividing through by the diagonal element.

```
>> C(1,:) = C(1,:) / C(1,1);
>> C(2,:) = C(2,:) / C(2,2);
>> C(3,:) = C(3,:) / C(3,3)
C =
    1    0    0    5
    0    1    0    1
    0    0    1   -4
```

The system of equations is now x_1 = C(1,n+1), x_2 = C(2, n+1),…where n is the original size of the system.

What was originally *b* (the right hand side of the equations) is now *x* (the solution).

The method allows multiple right hand sides to be dealt with in parallel.

Example: Solve $Ax=b$ for $A=\begin{bmatrix}4 & 2 & -1\\ 8 & 2 & 5\\ 5 & -5 & 6\end{bmatrix}$ $b=\begin{bmatrix}0\\3\\2\end{bmatrix}, \begin{bmatrix}1\\0\\4\end{bmatrix}, \text{ and } \begin{bmatrix}6\\1\\5\end{bmatrix}$

Original C:

```
   4      2     -1      0      1      6
   8      2      5      3      0      1
   5     -5      6      2      4      5
```

Final C:

```
1.0000         0         0    0.0197    0.5592    1.8092
     0    1.0000         0    0.2039   -0.8882   -1.6382
     0         0    1.0000    0.4868   -0.5395   -2.0395
```

1st solution 2nd solution 3rd solution

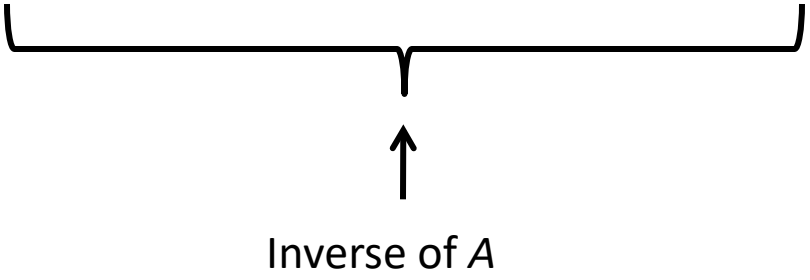
The inverse of *A* may be obtained by using an identity matrix as the “multiple right hand sides”.

Original C:

```
     4      2     -1      1      0      0
     8      2      5      0      1      0
     5     -5      6      0      0      1
```

Final C:

```
1.0000         0         0    0.2434   -0.0461    0.0789
     0    1.0000         0   -0.1513    0.1908   -0.1842
     0         0    1.0000   -0.3289    0.1974   -0.0526
```



Inverse of *A*

Code for backward elimination and normalization:

```
% np is n + 1, nm is n + number of right hand sides

% backward elimination
for i = n : -1 : 2
   for k = i - 1: -1 : 1 % for all rows above pivot
      % only RHS and pivot columns of row k will change
      C(k, np:nm) = C(k, np:nm) - C(k, i) / C(i, i) * C(i, np:nm);
      C(k, i) = 0;  % why calculate when the answer is known?
   end
end

% normalization
for i = 1 : n
   C(i, np:nm) = C(i, np:nm) / C(i, i);
   C(i, i) = 1;
end

x = C(:, np:nm);  % extract solution
```

Problem: A man has \$600. He intends to place \$100 bets on red at the roulette table until he either goes broke or has \$1000. What is the probability that the man will leave the casino with \$1000?

The probability of a bet on red winning is 0.4737. A winning bet pays 2:1 (i.e. one receives twice the amount bet).

If the man starts with \$600 he has a 0.4737 chance of moving to a state in which he has \$700 and a (1 – 0.4737) = 0.5263 chance of moving to a state in which he has \$500.

Let P_i be the probability that the man will leave with \$1000 if he starts with *i* dollars

Then $P_{600} = 0.4737 * P_{700} + 0.5263 * P_{500}$

Similar expressions can be written for P_{200}, P_{300}, P_{400}, P_{500}, P_{700}, and P_{800}

Finally we have

$P_{100} = 0.4737 * P_{200} + 0.5263 *(0)$ and $P_{900} = 0.4737 *(1) + 0.5263 * P_{800}$

This gives us nine equations in nine unknowns (P_{100} through P_{900})

The system of equations is:

$$\begin{bmatrix} 1 & -.4737 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -.5263 & 1 & -.4737 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -.5263 & 1 & -.4737 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -.5263 & 1 & -.4737 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -.5263 & 1 & -.4737 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -.5263 & 1 & -.4737 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -.5263 & 1 & -.4737 & \\ 0 & 0 & 0 & 0 & 0 & 0 & -.5263 & 1 & -.4737 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -.5263 & 1 \end{bmatrix} \begin{bmatrix} P_{100} \\ P_{200} \\ P_{300} \\ P_{400} \\ P_{500} \\ P_{600} \\ P_{700} \\ P_{800} \\ P_{900} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ .4737 \end{bmatrix}$$

```
A = zeros(9);  % creates 9x9 matrix
for i = 1 : 9
      A(i,i) = 1;
      if i ~= 1; A(i, i - 1) = -.5263; end
      if i ~= 9; A(i, i + 1) = -.4737; end
End
b = zeros(9,1); b(9) = .4737;
x = GaussJordan(a,b)  % other methods equally applicable
% answer (in x(6)) is 0.4721
```

We can take advantage of fact that A is *tridiagonal*.

$$\begin{bmatrix} f_1 & g_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ e_2 & f_2 & g_2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & e_3 & f_3 & g_3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & e_4 & f_4 & g_4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & e_5 & f_5 & g_5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & e_6 & f_6 & g_6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & e_7 & f_7 & g_7 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & e_8 & f_8 & g_8 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & e_9 & f_9 \end{bmatrix} \begin{bmatrix} P_{100} \\ P_{200} \\ P_{300} \\ P_{400} \\ P_{500} \\ P_{600} \\ P_{700} \\ P_{800} \\ P_{900} \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \\ r_7 \\ r_8 \\ r_9 \end{bmatrix}$$

Can save space by using three vectors in place of matrix *A*

f = diagonal

e = below diagonal (*e*(1) unused and set to zero)

g = above diagonal (*g*(*n*) unused and set to zero)

Can greatly reduce computational time by using the Gauss-Thomas method

Gauss-Thomas method:

Essentially just an optimized version of Gaussian elimination. The basic idea is identical.

Uses vector representation of A.

For each pivot position only the immediately following row need be modified.

Only two elements (other than the one that becomes zero) of this row need to be calculated.

The net effect is that the computational cost is $O(n)$ rather than $O(n^3)$.

Matlab code is in the text (p229).

Elements of e not actually set to zero as there is no point in doing this.

LU Factorization:

"Decompose" A into L (lower diagonal) and U (upper diagonal) such that $LU = A$

Once this has been done $Ax = b$ can be efficiently solved for any b of interest.

First we find d by solving $Ld = b$ $\quad d = L^{-1}b \Rightarrow Ld = b$

Then we find x by solving $Ux = d$ $\quad Ax = b \Rightarrow LUx = b \Rightarrow Ux = L^{-1}b \Rightarrow Ux = d$

Both solutions are fast because of nature of L and U.

In Matlab:

```
>> [L, U] = lu(A);     % decompose A

% solve for b1
>> d = L\b1;          % find d
>> x1 = U\d           % find x

% solve for b2
>> d = L\b2;          % find d
>> x2 = U\d           % find x
```

When "lu" is used in this way *L* may not actually be lower diagonal. Instead it is merely guaranteed to be a "psychologically lower triangular matrix" (to quote the documentation).

```
>> A = [2 -2 4; 4 1 6; 1 -2 3];

>>  [L, U] = lu(A)
L =
   0.5000    1.0000         0
   1.0000         0         0
   0.2500    0.9000    1.0000
U =
   4.0000    1.0000    6.0000
        0   -2.5000    1.0000
        0         0    0.6000
```

The scrambling of *L* is a result of the row switches (row pivoting) that occurred during the process of creating *U* from *A*. The *L* returned is in fact $P^{-1} L$, where *P* is a "permutation matrix" that reflects the row switches.

A strictly lower diagonal *L* may be obtained is shown below.

```
>> [L, U, P] = lu(A)
L =
    1.0000         0         0
    0.5000    1.0000         0
    0.2500    0.9000    1.0000
U =
    4.0000    1.0000    6.0000
         0   -2.5000    1.0000
         0         0    0.6000
P =
     0     1     0
     1     0     0
     0     0     1
```

In this case *LU = PA*

L and *U* are the *LU* decomposition of a permuted version of *A*.

To preserve the consistency of the equations, the same row switches must be applied to *b* (i.e. *Pb* must be used in place of *b*).

Alternate way of looking at things:

$$Ax = b$$

$$PAx = Pb \quad \% \text{ multiply both sides by P}$$

$$LUx = Pb \quad \% \text{ replace PA with LU (recall that LU = PA)}$$

This is what we began with but with *b* replaced with *Pb*.

In Matlab:

```
>> [L, U, P] = lu(A);     % decompose A

>> d = L\(P * b1);        % find d (permuting b1 to match LU)
>> x = U\d                % find x

>> d = L\(P * b2);        % find d (permuting b2 to match LU)
>> x = U\d                % find x
```

Cholesky factorization:

Can be thought of as a specialized form of LU factorization.

Only applicable when A is symmetric ($A = A^T$)

In this case A can be efficiently decomposed into $U^T U$

The procedure for solving Ax=b is exactly as for LU decomposition with U^T (which is lower triangular) serving as L.

In Matlab:

```
>> U = chol(A);     % decompose A

>> d = U'\b1;      % find d
>> x = U\d          % find x

>> d = U'\b2;      % find d
>> x = U\d          % find x
```

Errors on text p246: d = A'\b should be d = U'\b
x = A\y should be x = U\d