6.1 Manipulating Matrices

- We will review how Matlab works with matrices and introduce some new functions to help us do linear algebra.

EXAMPLE 1 Create a 3 x 3 matrix and get its transpose.

```
>> A = [1 5 6;7 4 2;3 6 7]
A =
     1     5     6
     7     4     2
     3     6     7
>> A'
ans =
     1     7     3
     5     4     6
     6     2     7
```

EXAMPLE 2 Compare **matrix multiplication** and **element-wise multiplication**.

```
>> A*A
ans =
     54     61     58
     41     63     64
     66     81     79
```

```
>> A.*A
ans =
      1     25     36
     49     16      4
      9     36     49
```

EXAMPLE 3 Calculate the inverse of ***A***.

```
>> Ainv = inv(A)
Ainv =
   -0.8421   -0.0526    0.7368
    2.2632    0.5789   -2.1053
   -1.5789   -0.4737    1.6316
```

EXAMPLE 4 View A^{-1} as rational numbers.

```
>> format rat
>> Ainv
Ainv =
     -16/19          -1/19           14/19
      43/19          11/19          -40/19
     -30/19          -9/19           31/19
```

EXAMPLE 5 Save A^{-1} as a character array of rational numbers.

```
>> Ainv_rational = rats(Ainv)
Ainv_rational =
  3×42 char array
    '    -16/19          -1/19           14/19    '
    '     43/19          11/19          -40/19    '
    '    -30/19          -9/19           31/19    '
```

Uses of Rational Formats

- The rational number functions are useful for human-usable output.
- When we code a method to solve a large linear system it is often useful to see the numbers in rational form since they are generally easier to comprehend than decimal numbers.
- The string variant is useful for printing the output to screen or to a data file.

EXAMPLE 6 Create a 4 x 4 identity matrix.

```
>> I = eye(4)
I =
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

EXAMPLE 7 Create a 3 x 3 matrix of zeros.

```
>> B = zeros(3)
B =
     0     0     0
     0     0     0
     0     0     0
```

Row Operations with Elementary Matrices

- When we come to solve linear systems we will store the system information in matrices.

- Solving such systems requires performing Elementary Row Operations (EROs):

  1. Multiply a row by a scalar
  2. Swap rows
  3. Add multiples of rows to other rows

- We can do each of these by multiplying the matrix by an elementary matrix.

- An elementary matrix is any matrix which is exactly one ERO away from the identity matrix.

$$E_1 = \begin{pmatrix} 4 & 0 \\ 0 & 1 \end{pmatrix} \qquad E_2 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad E_3 = \begin{pmatrix} 1 & 0 \\ -2 & 1 \end{pmatrix}$$

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \longrightarrow E_1 A = \begin{pmatrix} 4 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 4 & 8 \\ 3 & 4 \end{pmatrix}$$

- Multiplying a matrix by these performs the row operations in sequential order:

$$E_1 E_2 E_3 A$$

means,

$R_2 \rightarrow R_2 - 2R_1$ then Swap R_1 with R_2 then $R_1 \rightarrow 4R_1$.

EXAMPLE 7 Create an elementary matrix that swaps rows 1 & 3 of ***A***.

```
>> A = [1 5 6; 7 4 2; 3 6 7]
A =
        1                   5                   6
        7                   4                   2
        3                   6                   7

>> E = [0 0 1; 0 1 0; 1 0 0]
E =
        0                   0                   1
        0                   1                   0
        1                   0                   0
>> E*A
ans =
        3                   6                   7
        7                   4                   2
        1                   5                   6
```

Or go to www.pollev.com/jsands601

Which one is an elementary matrix?

$$\begin{pmatrix} 0 & 1 \\ 1 & -3 \end{pmatrix}$$

$$\begin{pmatrix} 1 & -3 \\ 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 \\ 2 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$$

Tc 0

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Which one is not an elementary matrix?

$$\begin{pmatrix} 1 & 0 & 0 \\ 14 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -5 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 2 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

EXAMPLE 8 Augment the matrix ***A*** with the 3 x 3 identity matrix and get its dimensions.

```
>> aug = [A eye(3)]
aug =
     1     5     6     1     0     0
     7     4     2     0     1     0
     3     6     7     0     0     1
```

```
>> size(aug)
ans =
     3     6
```

EXAMPLE 9 Calculate the determinant of the matrix ***A***.

```
>> det(A)
ans =
   -19
```

6.2 Solving Linear Systems

- We can write a linear system in matrix form:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2$$
$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3$$

$$\longrightarrow \quad [A]\{x\} = \{b\}$$

$$[A] = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

- It's possible to solve easily if ***A*** is invertible (unique solution).

$$[A]^{-1}[A]\{x\} = [A]^{-1}\{b\} \longrightarrow \{x\} = [A]^{-1}\{b\}$$

6.3 Left-Division vs. Inverse Matrix

- We can program this in ourselves or use MATLAB's built-in **left-division** operator.

EXAMPLE 10 Solve the following linear system in MATLAB.

$$\begin{bmatrix} 150 & -100 & 0 \\ -100 & 150 & -50 \\ 0 & -50 & 50 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 588.6 \\ 686.7 \\ 784.8 \end{Bmatrix}$$

```
>> A = [150 -100 0;-100 150 -50;0 -50 50]
A =
   150  -100     0
  -100   150   -50
     0   -50    50
```

```
>> b = [588.6; 686.7; 784.8]
b =
   588.6000
   686.7000
   784.8000
```

```
>> x = inv(A)*b
x =
    41.2020
    55.9170
    71.6130
```

```
>> x = A\b
x =
    41.2020
    55.9170
    71.6130
```

- The left division operator in Matlab is a bit more complicated than simply multiplying by the inverse so in general it is more robust. We will look at how it works later.

6.4 Gaussian Elimination

- The methods in the previous example will not work for systems with infinite solutions or no solution.
- Inverse matrix method fails since there is no inverse for such systems.
- Left-division will return a single answer, but we cannot get the other infinity of solutions.
- We resort to Gaussian elimination and use the echelon form to tell us the solutions. We can then make a basis for the solution space.

Echelon Form

- Start at the top left and find the first non-zero number (called the first pivot). Use a loop to move between columns.
- Make sure everything below that is 0 (use EROs).
- Move to next row and find first non-zero entry (second pivot). Use another loop to move between rows.
- Make everything below that 0.
- Continue until the last row is reached.
- Any “all-zero rows” should be at the bottom.

- If all pivots are equal to 1 and there are 0’s both above and below we call this (Row) Reduced Echelon Form (RREF).

Echelon form:

$$\begin{bmatrix} \blacksquare & * & * & * \\ 0 & \blacksquare & * & * \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad \begin{bmatrix} 0 & \blacksquare & * & * & * & * & * & * & * & * \\ 0 & 0 & 0 & \blacksquare & * & * & * & * & * & * \\ 0 & 0 & 0 & 0 & \blacksquare & * & * & * & * & * \\ 0 & 0 & 0 & 0 & 0 & \blacksquare & * & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \blacksquare & * \end{bmatrix}$$

Reduced Echelon form (REF or RREF):

$$\begin{bmatrix} 1 & 0 & * & * \\ 0 & 1 & * & * \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad \begin{bmatrix} 0 & 1 & * & 0 & 0 & 0 & * & * & 0 & * \\ 0 & 0 & 0 & 1 & 0 & 0 & * & * & 0 & * \\ 0 & 0 & 0 & 0 & 1 & 0 & * & * & 0 & * \\ 0 & 0 & 0 & 0 & 0 & 1 & * & * & 0 & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & * \end{bmatrix}$$

Is the matrix in echelon form?

$$\begin{pmatrix} 2 & -1 & -1 & 1 & 4 \\ 0 & 4 & 0 & 3 & 7 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Yes

No

Is the matrix in echelon form?

$$\begin{pmatrix} 0 & -1 & 7 & 5 & 0 & -2 \\ 0 & 0 & 0 & 3 & 7 & 0 \\ 0 & 0 & 0 & 0 & 1 & -4 \\ 0 & 0 & 0 & 0 & 0 & 6 \end{pmatrix}$$

Yes

No

To 0

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Is the matrix in rref?

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \end{pmatrix}$$

Yes

No

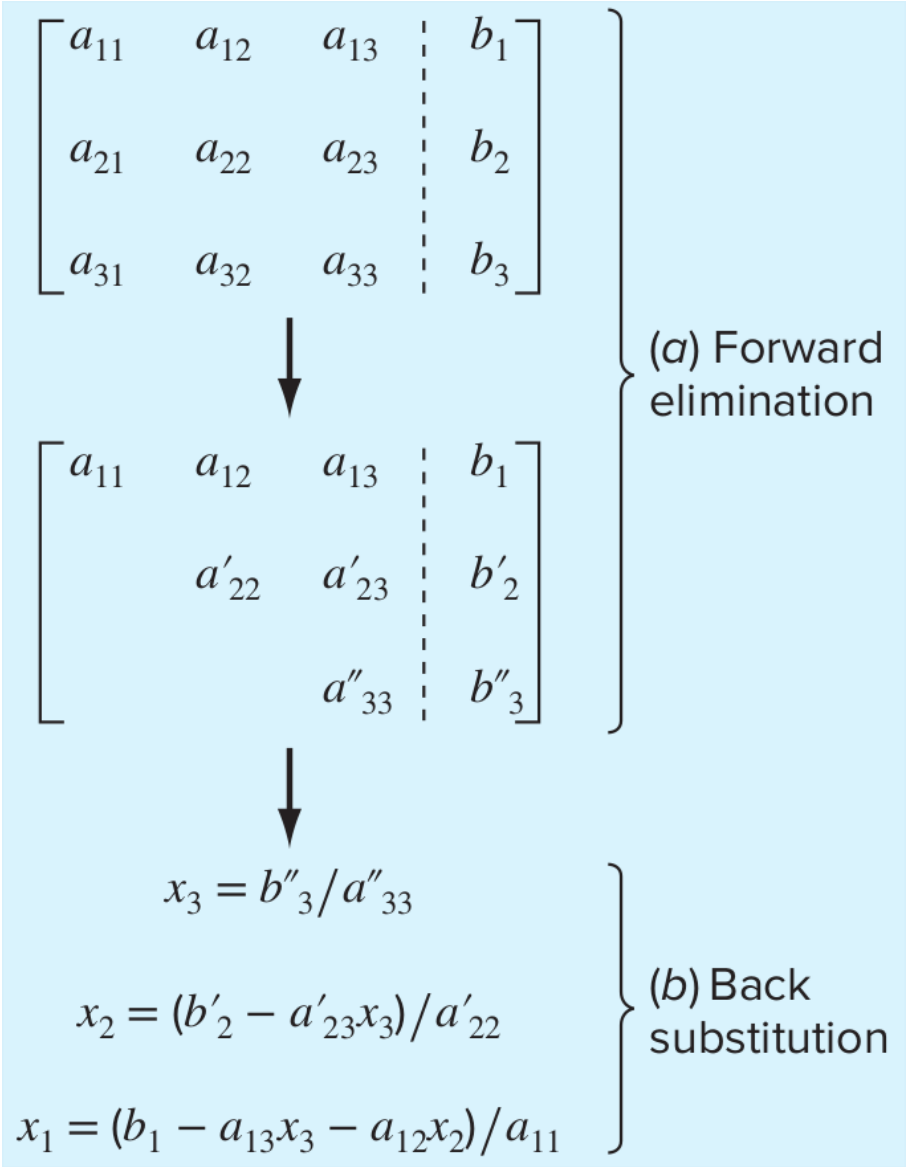
Is the matrix in rref?

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Yes

No

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**



- Make sure top row has non-zero entry (first pivot).
- Create zeros below first pivot.
- Repeat for all rows below to finish the forward elimination phase.

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n = b_1$$
$$a'_{22}x_2 + a'_{23}x_3 + \cdots + a'_{2n}x_n = b'_2$$
$$a''_{33}x_3 + \cdots + a''_{3n}x_n = b''_3$$
$$\ddots \qquad \vdots$$
$$a_{nn}^{(n-1)}x_n = b_n^{(n-1)}$$

- Solve for x_i's using back substitution:

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

6.5 Pivoting

- Problems can occur when the pivot is close to 0 compared with the other elements since we will get round-off errors.

- **Partial pivoting** means we find the largest value in a column and swap rows so it is in the pivot position before proceeding with the algorithm.

- **Full pivoting** is when the largest value of the rows and columns is selected for the pivot position. This method is not used often since the efficiency gain over partial pivoting is small but it introduces complexity by switching the order of the *x*'s.

EXAMPLE 11 Compare Gaussian elimination with and without partial pivoting for the following system.

$$0.0003x_1 + 3.0000x_2 = 2.0001$$
$$1.0000x_1 + 1.0000x_2 = 1.0000$$

No pivoting:

$$x_1 + 10{,}000x_2 = 6667$$
$$-9999x_2 = -6666$$

$$x_2 = 2/3$$
$$x_1 = \frac{2.0001 - 3(2/3)}{0.0003}$$

- Since we are using a computer to do the sum we must approximate the fraction with a decimal.

- Our value of x_1 depends strongly on how many significant figures we use for 2/3.

Significant Figures	x_2	x_1	Absolute Value of Percent Relative Error for x_1
3	0.667	−3.33	1099
4	0.6667	0.0000	100
5	0.66667	0.30000	10
6	0.666667	0.330000	1
7	0.6666667	0.3330000	0.1

With partial pivoting:

$$1.0000x_1 + 1.0000x_2 = 1.0000$$
$$0.0003x_1 + 3.0000x_2 = 2.0001$$

$$x_1 + x_2 = 1$$
$$2.9997x_2 = 1.9998$$

$$\longrightarrow \quad x_2 = 2/3$$

$$x_1 = \frac{1 - (2/3)}{1}$$

Significant Figures	x_2	x_1	Absolute Value of Percent Relative Error for x_1
3	0.667	0.333	0.1
4	0.6667	0.3333	0.01
5	0.66667	0.33333	0.001
6	0.666667	0.333333	0.0001
7	0.6666667	0.3333333	0.0000

6.6 Comparison of Procedures

- The number of operations (addition/subtraction, multiplication/division) is known as the **number of flops** (floating point operations).
- To keep our algorithms efficient for solving linear systems we should try to minimise the number of flops ("cost of the solution").
- For small systems it is common to use less efficient algorithms in order to keep the method simple, however as the system gets larger we must employ more sophisticated algorithms in order to maintain efficiency.

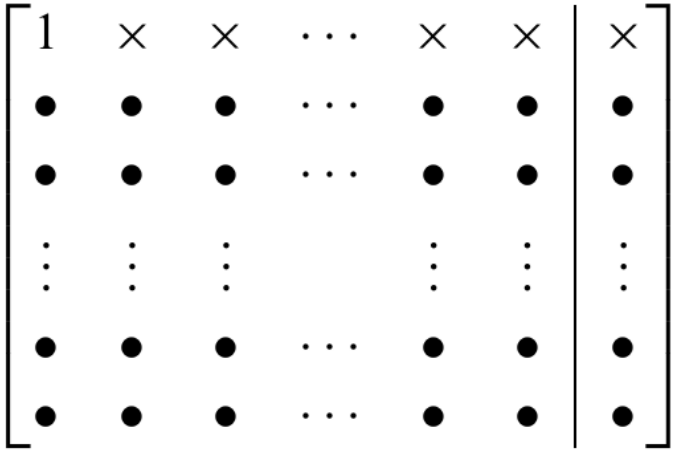
Value in scientific notation	Metric prefix	
	Prefix	Symbol
10^0		
10^1	deca	da
10^2	hecto	h
10^3	kilo	k
10^6	mega	M
10^9	giga	G
10^{12}	tera	T
10^{15}	peta	P
10^{18}	exa	E
10^{21}	zetta	Z
10^{24}	yotta	Y

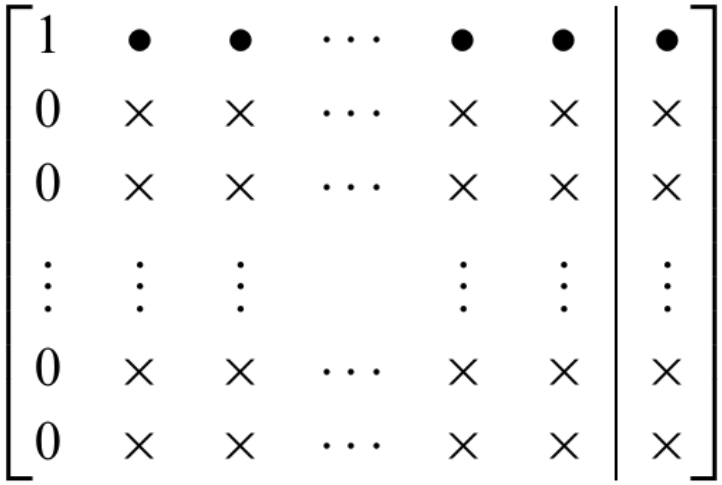
Source: https://en.wikipedia.org/wiki/Long_and_short_scale

- Let's count the number of flops required to solve a system with Gaussian elimination and use the simple case of an invertible *n* x *n* matrix, **A**, that requires no row changes.

- We will also add the stronger condition that the leading entries must be 1.

- It takes *n* flops to make the first row.





- We need *n* multiplications and another *n* additions to create a 0 below the 1.

- We then need to do that to each of the *n* - 1 rows beneath which gives $2n(n-1)$.

- The number of flops for the 1st column is therefore:

  1st column: $n + 2n(n - 1) = 2n^2 - n$

- The 2nd column follows similarly but with 1 less row and 1 less column so we just replace n with $n - 1$ in the previous count:

  2nd column: $2(n - 1)^2 - (n - 1)$

- We can continue in this fashion to get:

$$(2n^2 - n) + [2(n - 1)^2 - (n - 1)] + [2(n - 2)^2 - (n - 2)] + \cdots + (2 - 1)$$

- Factoring:

$$2[n^2 + (n - 1)^2 + \cdots + 1] - [n + (n - 1) + \cdots + 1]$$

- Remembering the following partial sum formulae:

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} \qquad \sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

we have,

$$2\frac{n(n+1)(2n+1)}{6} - \frac{n(n+1)}{2} = \frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{1}{6}n$$

- Taking the highest order term gives us a suitable handle on the error for large *n* and we say that:

No. of flops ≈ ⅔n^3 = $O(n^3)$

Summary of Linear System Solver Costs

Approximate Cost for an $n \times n$ Matrix A with Large n	
Algorithm	**Cost in Flops**
Gauss–Jordan elimination (forward phase)	$\approx \frac{2}{3}n^3$
Gauss–Jordan elimination (backward phase)	$\approx n^2$
LU-decomposition of A	$\approx \frac{2}{3}n^3$
Forward substitution to solve $L\mathbf{y} = \mathbf{b}$	$\approx n^2$
Backward substitution to solve $U\mathbf{x} = \mathbf{y}$	$\approx n^2$
A^{-1} by reducing $[A \mid I]$ to $[I \mid A^{-1}]$	$\approx 2n^3$
Compute $A^{-1}\mathbf{b}$	$\approx 2n^3$

EXAMPLE 12 Approximate the time required to perform both the forward and backward phases of Gauss-Jordan elimination for a system of 1 million equations with 1 millions unknowns if the computer can execute 10 petaflops per second.

- Forward phase: $\approx \frac{2}{3}n^3 \times 10^{-16} = 66.67 \text{ s}$
- Backward phase: $\approx n^2 \times 10^{-16} = 0.0001 \text{ s}$
- Clearly the forward phase is significantly more costly.

6.7 Gauss-Jordan Elimination

- If we continue the row reduction until the left matrix becomes the identity matrix then the augmented vector simply becomes the solution of the system.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 & 0 & x_1 \\ 0 & 1 & 0 & x_2 \\ 0 & 0 & 1 & x_3 \end{bmatrix}$$

- This is known as Gauss-Jordan elimination.

- Matlab has a built-in function for this called **rref**.

```
>> rref([1 -1; -2 2])
```

→

```
ans =
     1    -1
     0     0
```

6.8 LU Factorisation

- This method is generally favoured over Gauss-Jordan elimination with back substitution for square matrices.

- First row reduce to get an upper triangular and a lower triangular matrix:

$$[A]\{x\} = \{b\}$$

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} d_1 \\ d_2 \\ d_3 \end{Bmatrix} \qquad [L] = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \rightarrow [L][U] = [A]$$

- Solve ***Ly = b*** to get ***y***.

- Solve ***Ux = y*** to get ***x***.

- ***U*** is found from row reduction:

$$[U] = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a'_{22} & a'_{23} \\ 0 & 0 & a''_{33} \end{bmatrix}$$

- To find ***L*** note that the procedure to get ***U*** was:

$$a'_{21} = 0 = a_{21} - \frac{a_{21}}{a_{11}} a_{11} \qquad a'_{22} = a_{22} - \frac{a_{21}}{a_{11}} a_{12}$$

$$f_{21} = \frac{a_{21}}{a_{11}}$$

$$a'_{31} = 0 = a_{31} - \frac{a_{31}}{a_{11}} a_{11} \qquad a'_{32} = a_{32} - \frac{a_{31}}{a_{11}} a_{12}$$

$$f_{31} = \frac{a_{31}}{a_{11}}$$

$$f_{32} = \frac{a'_{32}}{a'_{22}}$$

$$a''_{32} = 0 = a'_{32} - \frac{a'_{32}}{a'_{22}} a'_{22}$$

- Using basic matrix algebra it can be shown that:

$$[L] = \begin{bmatrix} 1 & 0 & 0 \\ f_{21} & 1 & 0 \\ f_{31} & f_{32} & 1 \end{bmatrix}$$

- Using this method to calculate ***L*** and ***U*** allows for the efficient solution of linear systems.

- The row reduction used to produce ***U*** can also include partial pivoting to make it more robust. Note the factorisation is not unique.

- MATLAB also has a built-in function **lu** that creates an LU decomposition:

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

Advantages of LU-Factorisation

- Gauss-Jordan elimination requires a new computation for each value of ***b*** whereas the costly phase of obtaining ***LU*** is done independently so any ***b*** can be used with the second, less costly phase.

- For very large systems we can save on computer storage memory by recognising encoding the ***LU*** matrices into a single matrix (getting rid of the 0's).

- The method can be modified slightly to produce the inverse matrix the system without much further work.

Generalising the Method

- An **LU**-factorisation is only guaranteed to exist if we don't exchange rows (linear algebra theorem) so we generalise the method using only the other EROs.

- Note we now create the **U** matrix with 1's on the diagonal.

Procedure for Constructing an LU-Decomposition

Step 1. Reduce A to a row echelon form U by Gaussian elimination without row interchanges, keeping track of the multipliers used to introduce the leading 1's and the multipliers used to introduce the zeros below the leading 1's.

Step 2. In each position along the main diagonal of L, place the reciprocal of the multiplier that introduced the leading 1 in that position in U.

Step 3. In each position below the main diagonal of L, place the negative of the multiplier used to introduce the zero in that position in U.

Step 4. Form the decomposition $A = LU$.

EXAMPLE 13 Find an **LU**-factorisation of,

$$A = \begin{bmatrix} 6 & -2 & 0 \\ 9 & -1 & 1 \\ 3 & 7 & 5 \end{bmatrix}$$

$$A = \begin{bmatrix} 6 & -2 & 0 \\ 9 & -1 & 1 \\ 3 & 7 & 5 \end{bmatrix} \qquad \begin{bmatrix} \bullet & 0 & 0 \\ \bullet & \bullet & 0 \\ \bullet & \bullet & \bullet \end{bmatrix}$$

$$\begin{bmatrix} \textcircled{1} & -\frac{1}{3} & 0 \\ 9 & -1 & 1 \\ 3 & 7 & 5 \end{bmatrix} \longleftarrow \text{multiplier} = \frac{1}{6} \qquad \begin{bmatrix} 6 & 0 & 0 \\ \bullet & \bullet & 0 \\ \bullet & \bullet & \bullet \end{bmatrix}$$

$$\begin{bmatrix} 1 & -\frac{1}{3} & 0 \\ \textcircled{0} & 2 & 1 \\ \textcircled{0} & 8 & 5 \end{bmatrix} \begin{matrix} \\ \longleftarrow \text{multiplier} = -9 \\ \longleftarrow \text{multiplier} = -3 \end{matrix} \qquad \begin{bmatrix} 6 & 0 & 0 \\ 9 & \bullet & 0 \\ 3 & \bullet & \bullet \end{bmatrix}$$

$$\begin{bmatrix} 1 & -\frac{1}{3} & 0 \\ 0 & \textcircled{1} & \frac{1}{2} \\ 0 & 8 & 5 \end{bmatrix} \longleftarrow \text{multiplier} = \frac{1}{2} \qquad \begin{bmatrix} 6 & 0 & 0 \\ 9 & 2 & 0 \\ 3 & \bullet & \bullet \end{bmatrix}$$

$$\begin{bmatrix} 1 & -\frac{1}{3} & 0 \\ 0 & 1 & \frac{1}{2} \\ 0 & \textcircled{0} & 1 \end{bmatrix} \longleftarrow \text{multiplier} = -8 \qquad \begin{bmatrix} 6 & 0 & 0 \\ 9 & 2 & 0 \\ 3 & 8 & \bullet \end{bmatrix}$$

$$U = \begin{bmatrix} 1 & -\frac{1}{3} & 0 \\ 0 & 1 & \frac{1}{2} \\ 0 & 0 & \textcircled{1} \end{bmatrix} \longleftarrow \text{multiplier} = 1 \qquad L = \begin{bmatrix} 6 & 0 & 0 \\ 9 & 2 & 0 \\ 3 & 8 & 1 \end{bmatrix}$$

$$A = LU = \begin{bmatrix} 6 & 0 & 0 \\ 9 & 2 & 0 \\ 3 & 8 & 1 \end{bmatrix} \begin{bmatrix} 1 & -\frac{1}{3} & 0 \\ 0 & 1 & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

PLU-Factorisation

- Since the **LU**-factorisation is not guaranteed to exist if we exchange rows, however if row changing is necessary to minimise round-off errors (pivoting) then we can apply the row changes before the **LU**-factoring process begins to guarantee its existence.

- The row exchanges are represented by the product of elementary matrices, **P**.

- The factorisation is then: **A** = **PLU**

```
>> [L,U,p] = lu(A)
```

```
>> P*L*U % Returns A
```

```
>> L*U == P'*A % Returns True
```

6.9 Calculating Matrix Inverses

- We can augment the matrix with the identity matrix then bring to reduced echelon form:

$$[\boldsymbol{A} \quad \boldsymbol{I}] \sim [\boldsymbol{I} \quad \boldsymbol{A}^{-1}]$$

- Alternatively we can use ***LU***-factorisation. The columns of $\boldsymbol{A}^{-1}$ come from solving the following systems:

$$\boldsymbol{Ax} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad \boldsymbol{Ax} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \qquad \boldsymbol{Ax} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

1st column of $\boldsymbol{A}^{-1}$ 2nd column of $\boldsymbol{A}^{-1}$ 3rd column of $\boldsymbol{A}^{-1}$

6.10 Newton Method for Nonlinear Systems

- If we have a nonlinear system of equations:

$$\left.\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, \ldots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, \ldots, x_n) &= 0 \end{aligned}\right\} \quad \boldsymbol{F(x) = 0}$$

- We can use the multidimensional Newton-Raphson method to search for roots.

$$\boldsymbol{x_{i+1} = x_i - J^{-1} F}$$

Note that all of these are vector and matrix variables

- The matrix ***J*** is the Jacobian matrix:

Matrix of 1st order partial derivatives

$$[J] = \begin{bmatrix} \frac{\partial f_{1,i}}{\partial x_1} & \frac{\partial f_{1,i}}{\partial x_2} & \cdots & \frac{\partial f_{1,i}}{\partial x_n} \\ \frac{\partial f_{2,i}}{\partial x_1} & \frac{\partial f_{2,i}}{\partial x_2} & \cdots & \frac{\partial f_{2,i}}{\partial x_n} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial f_{n,i}}{\partial x_1} & \frac{\partial f_{n,i}}{\partial x_2} & \cdots & \frac{\partial f_{n,i}}{\partial x_n} \end{bmatrix}$$

- Note that the matrix multiplication $\boldsymbol{J^{-1}F}$ is the same as the MATLAB left-division operator:

```
>> J\F
```

- Note that when looping through a system of equations with an iterative method we need a way to measure the error between each iteration.

- There are many norms that exist that could do this for us (we will look at these next time).

- For now use the magnitude of the difference vector:

$$\text{Error} = \sqrt{\boldsymbol{x_k} - \boldsymbol{x_{k-1}}}$$

6.11 Nonlinear Systems with fsolve

- MATLAB has a built-in function for nonlinear systems:

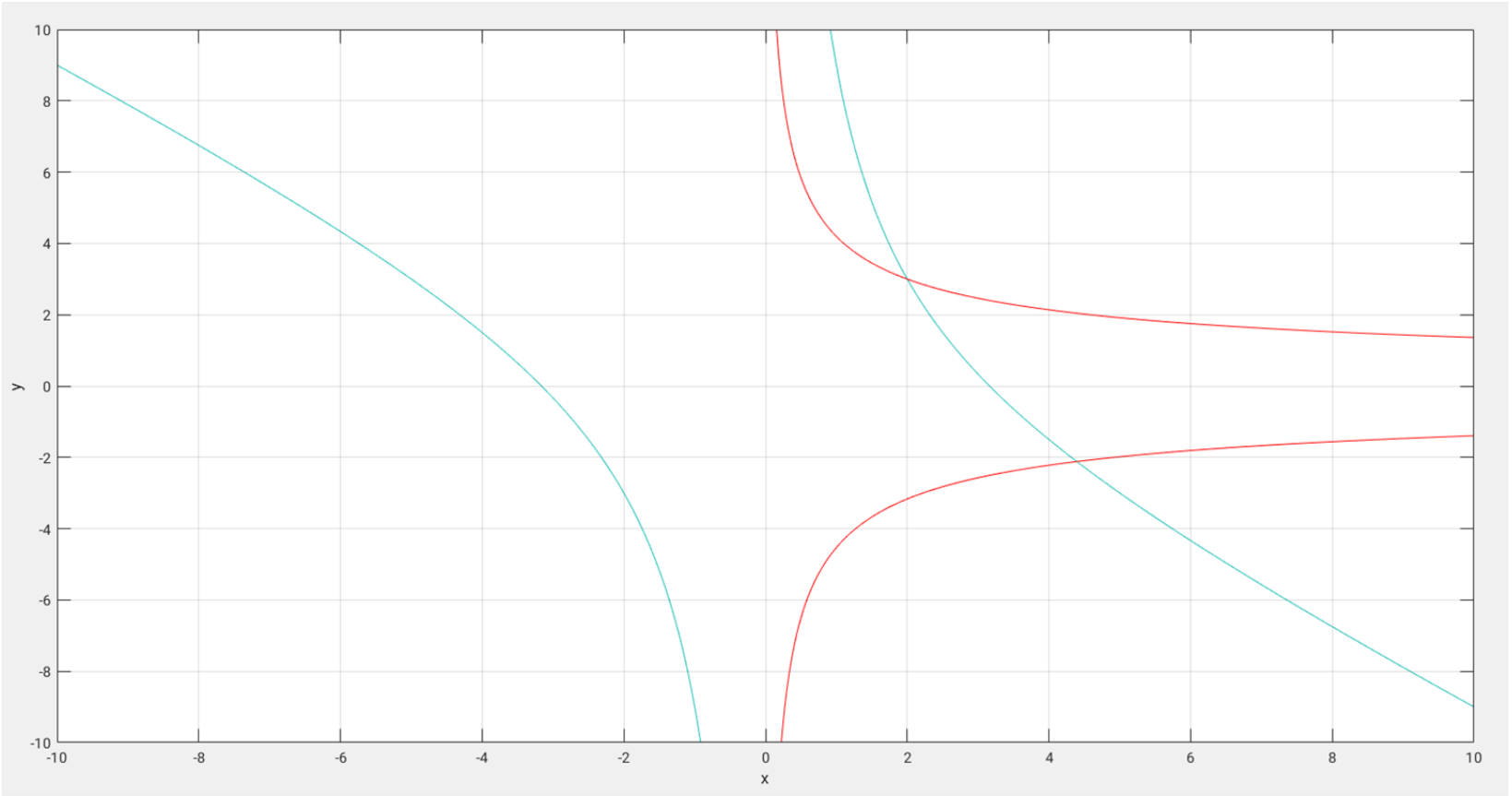
EXAMPLE 14 Solve the following nonlinear system in MATLAB.

$$f(x_1, x_2) = 2x_1 + x_1x_2 - 10$$
$$f(x_1, x_2) = x_2 + 3x_1x_2^2 - 57$$

Important: Make sure functions = 0!!!

Plot graphs to get idea of initial values

```
>> f1 = @(x,y) x.^2+x.*y - 10;
>> f2 = @(x,y) y+3*x.*y.^2 - 57;
>> fimplicit(f1,[-10 10])
>> hold on
>> fimplicit(f2,[-10 10])
>> grid on
```



- It looks like roots are around roughly (2,3) and (4,-2).

- We create a function file to store the nonlinear system:

```
function f = nonlin(x)
f = [x(1)^2+x(1)*x(2) - 10;
     x(2)+3*x(1)*x(2)^2 - 57];
```

- Then solve the system from the command window for different initial values.

```
>> [x,fx] = fsolve(@nonlin, [2,3])

Equation solved at initial point.

x =
     2     3
fx =
     0
     0
```

```
>> [x,fx] = fsolve(@nonlin, [4,-2])
```

Equation solved.

```
x =
        4.3937       -2.1178
fx =
   3.5527e-15
   7.1054e-15
```

- If we want to use anonymous functions we need to write it in terms of a variable vector and slightly different syntax for the solver.

```
>> f1 = @(x) x(1).^2+x(1).*x(2) - 10;
>> f2 = @(x) x(2)+3*x(1).*x(2).^2 - 57;

>> [x,fx] = fsolve(@(x)[f1(x);f2(x)], [4,-2])
```

```
Equation solved.

x =
        4.3937       -2.1178
fx =
   3.5527e-15
   7.1054e-15
```

- Finally if we want to write a script to solve a system like this we can embed the function to hold the system in the file.

- To do this we put a function at the end of our script the defines the system of equations and we finish the function with the keyword **end**.

- Note that all lines to be executed in Matlab must be **before** that the function.

myscript.m

```
x = linspace(-pi,pi);
y = myfunc(x)
plot(x,y)

function z = myfunc(t)
      z = t.^2;
end
```