CS1010E Lecture 9
Arrays and Matrices (Part II)

Joxan Jaffar

Block COM1, Room 3-11, +65 6516 7346
`www.comp.nus.edu.sg/∼joxan`
`cs1010e@comp.nus.edu.sg`

Semester II, 2016/2017

Lecture Outline

- Two-Dimensional Arrays

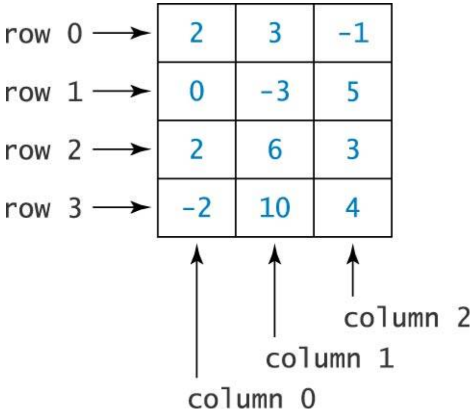
- Matrix Algorithms
  - Addition, Subtraction, Dot Product, Transpose
  - Solving Linear Equations (non-examinable)

- Higher-Dimensional Arrays (non-examinable)

Two-Dimensional Arrays

- A set of data values that is visualized as a row or column is easily represented by a one-dimensional array.
- However, there are many examples in which the best way to visualize a set of data is with a grid or a table of data, which has both rows and columns.
- A two-dimensional array with four rows and three columns is shown in the next slide.

Two-Dimensional Arrays



Two-Dimensional Arrays

- In C, a grid or table of data is represented with a **two-dimensional array**.
- Each element in a two-dimensional array is referenced using an identifier followed by two subscripts—a row subscript and a column subscript.
- The subscript values for both rows and columns begin with 0, and each subscript has its own set of brackets.

Two-Dimensional Arrays

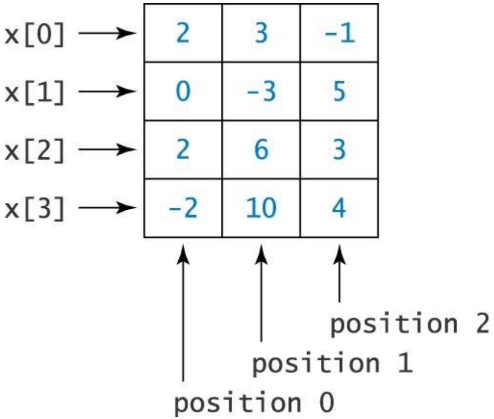
- Thus, assuming that the previous array has an identifier `x`, the value in position `x[2][1]` is 6.

- Common errors in array references include using parentheses instead of brackets as in `x(2)(3)`, or using only one set of brackets or parentheses as in `x[2,3]` or `x(2,3)`.

Two-Dimensional Arrays

- We can also visualize this grid or table of data as a one-dimensional array, where each element is also an array.
- Thus, the array in the previous diagram can be interpreted as a one-dimensional array with four elements, each of which is a one-dimensional array with three elements, as shown on the next slide.

Two-Dimensional Arrays



Two-Dimensional Arrays

- All values in an array must have the same type.
- An array cannot have a column of integers followed by a column of floating-point numbers, and so on.

Definition and Initialization

- To define a two-dimensional array, we specify the number of rows and the number of columns in the declaration statement.
- The row number is written first.
- Both the row number and the column number are in brackets, as shown in this statement:
  - `int x[4][3];`

Definition and Initialization

- A two-dimensional array can be initialized with a declaration statement.
- The values are specified in a sequence separated by commas, and each row is contained in braces.
- An additional set of braces is included around the complete set of values, as shown in the following statement:

  -
    ```
    int x[4][3]={{2,3,-1},{0,-3,5},
                 {2,6,3},{-2,10,4}};
    ```

Definition and Initialization

- If the initializing sequence is shorter than the array, then the rest of the values are initialized to zero.
- If the array is specified with the first subscript empty, but with an initializing sequence, the size is determined by the sequence.
- Thus, the array `x` can also be defined with the following statement

  -
    ```
    int x[][3]={{2,3,-1},{0,-3,5},
                {2,6,3},{-2,10,4}};
    ```

Definition and Initialization

- Arrays can also be initialized with program statements.

- For two-dimensional arrays, two nested `for` loops are usually required to initialize an array; `i` and `j` are commonly used as subscripts.

- To define and initialize an array such that each row contains the row number, use the following statements:

  -
    ```
    int i, j, t[5][4];
    ...
    for (i=0; i<=4; i++)
        for (j=0; j<=3; j++)
            t[i][j] = i;
    ```

Definition and Initialization

- After these statements are executed, the values in the array t are as follows:

    -
      ```
      0   0   0   0
      1   1   1   1
      2   2   2   2
      3   3   3   3
      4   4   4   4
      ```

Definition and Initialization

- Two-dimensional arrays can also be initialized with values read from a data file.
- In the next set of statements, we assume that a data file contains 50 temperature values that we read and store in the array.
- Symbolic constants `NROWS` and `NCOLS` are used to represent the number of rows and columns.
- Changing the size of an array is easier to do when the numbers of rows and columns are specified as symbolic constants; otherwise the change requires modifications to several statements.

Definition and Initialization

```
#define NROWS 10
#define NCOLS 5
#define FILENAME "engine1.txt"
...
/*  Declare variables.  */
int i, j;
double temps[NROWS][NCOLS];
FILE *sensor;
...
/*  Open file and read data into array.  */
sensor = fopen(FILENAME,"r");
for (i=0; i<=NROWS-1; i++)
    for (j=0; j<=NCOLS-1; j++)
        fscanf(sensor,"%lf",&temps[i][j]);
```

Computations and Output

- Computations and output with two-dimensional arrays must always specify two subscripts when referencing an array element.
- Consider a program that reads a data file containing power output for an electrical plant for an eight-week period.
- Each line of the data file contains seven values representing the daily power output for a week.
- The data are stored in a two-dimensional array.
- Then a report provides the average power for the first day of the week during the period, the average power for the second day of the week during the period, and so on.

Computations and Output

```
/*-----------------------------------------------------------------*/
/*  Program chapter5_6                                             */
/*                                                                 */
/*  This program computes daily power averages over eight          */
/*  weeks.                                                         */

#include <stdio.h>
#define NROWS 8
#define NCOLS 7
#define FILENAME "power1.txt"

int main(void)
{
    /*  Declare variables.  */
    int i, j;
    int power[NROWS][NCOLS], col_sum;
    FILE *file_in;
```

Code on Next Slide

```
   /* Exit program. */
   return 0; }
```

Computations and Output

```
/*  Read information from a data file.  */
file_in = fopen(FILENAME,"r");
if (file_in == NULL)
    printf("Error opening input file. \n");
else
{
    for (i=0; i<=NROWS-1; i++)
        for (j=0; j<=NCOLS-1; j++)
            fscanf(file_in,"%d",&power[i][j]);

    /*  Compute and print daily averages.  */
    for (j=0; j<=NCOLS-1; j++)
    {
        col_sum = 0;
        for (i=0; i<=NROWS-1; i++)
            col_sum += power[i][j];
        printf("Day %d: Average = %.2f \n",
                j+1,(double)col_sum/NROWS);
    }
    /*  Close file.  */
    fclose(file_in);
}
```

Computations and Output

- Note that the daily averages are computed by adding each column and then dividing the column sum by the number of rows (which is also the number of weeks).

- The column number is then used to compute the day number.

- A sample output from this program is as follows:

  -
    ```
    Day 1:  Average = 253.75
    Day 2:  Average = 191.50
    Day 3:  Average = 278.38
    Day 4:  Average = 188.62
    Day 5:  Average = 273.12
    Day 6:  Average = 321.38
    Day 7:  Average = 282.50
    ```

Computations and Output

- Writing information from a two-dimensional array to a data file is similar to writing the information from a one-dimensional array.
- In both cases, a newline indicator must be used to specify when the values are to begin a new line.
- To write a set of distance measurements to a data file named `dist1.txt` with five values per line, see the next slide.
- Note that the space after the conversion specifier in the `fprintf` statement is necessary in order to have the values separated by a space.

Computations and Output

```
/*  Declare variables.  */
int i, j;
double dist[20][5];
FILE *file_out;
...
/*  Write information from the array to a  */
/*  file.                                  */
file_out = fopen("dist1.txt","w");
for (i=0; i<=19; i++) {
    for (j=0; j<=4; j++)
        fprintf(file_out,"%f ",dist[i][j]);
    fprintf(file_out,"\n");
}
fclose(file_out);
```

Function Arguments

- When arrays are used as function parameters, the references are call-by-address instead of call-by-value.
- When discussing one-dimensional arrays, we learned that array references in a function refer to the original array and not to a copy of the array.
- Thus, we must be careful so we do not unintentionally change values in the original array.
- Of course, an advantage of a call-by-address reference is that we can make changes in the array values, in addition to returning a value from the function call.

Function Arguments

- When using a one-dimensional array as a function argument, the function needs only the address of the array, which is specified by the array name.
- When using a two-dimensional array as a function argument, the function also needs information about the size of the array.
- In general, the function declaration and prototype statement should give complete information about the size of a two-dimensional array.

Function Arguments

- Suppose that we need to write a program that computes the sum of the elements in an array containing four rows and four columns.

- Here is the function prototype statement:
  - `int sum(int x[4][4]);`

- The program can then reference the function with a single statement:

  -
    ```
    int a[4][4];
    ...
    printf("Array sum = %i \n",sum(a));
    ```

Function Arguments

- We can use this function for several two-dimensional arrays of the same size:

  -
    ```
    int a[4][4], b[4][4];
    ...
    printf("Sum of a = %i \n",sum(a));
    printf("Sum of b = %i \n",sum(b));
    ```
- Note that the actual parameter consists of simply `a`.
- Anything else (for example, `int a[4][4]` or `a[4][4]` or `a[][]`) is incorrect.

Function Arguments

```
/*-----------------------------------------------*/
/*  This function returns the sum of the values   */
/*  in an array with four rows and four columns.  */

int sum(int x[4][4])
{
    /*  Declare and initialize variables.  */
    int i, j, total=0;

    /*  Compute a sum of the array values.  */
    for (i=0; i<=3; i++)
        for (j=0; j<=3; j++)
            total += x[i][j];

    /*  Return sum of array values.  */
    return total;
}
```

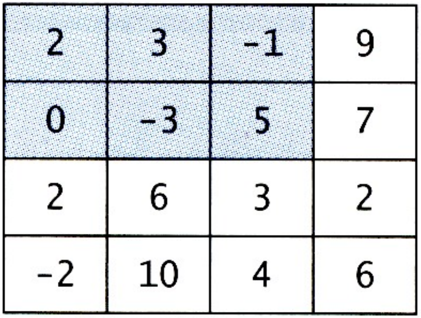
Function Arguments

- We can use this function for several two-dimensional arrays of the same size:

  -
    ```
    int a[4][4], b[4][4];
    ...
    printf("Sum of a = %i \n",sum(a));
    printf("Sum of b = %i \n",sum(b));
    ```
- Note that the actual parameter consists of simply `a`.
- Anything else (for example, `int a[4][4]` or `a[4][4]` or `a[][]`) is incorrect.

Partial Sum

Now suppose we want to compute only a *partial sum* of the elements in an array. The subarray shall be at the *top-left* corner of the original array.



2	3	-1	9
0	-3	5	7
2	6	3	2
-2	10	4	6

Partial Sum

```
/*-----------------------------------------------------------*/
/*  This function returns the sum of the values in an      */
/*  SUB-array of an array with four rows and               */
/*  four columns.  The subarray has r rows and c columns. */

int partial_sum(int x[4][4], int r, int c)
{
    /*  Declare and initialize variables.  */
    int i, j, total = 0;

    /*  Compute a sum of the array values.  */
    for (i = 0; i <= r-1; i++)
        for (j = 0; j <= c-1; j++)
            total += x[i][j];

    /*  Return sum of SUB-array values.  */
    return total;
}
```

Array Size Declarations

- With one-dimensional arrays (eg. `foo`) we could abbreviate the prototype definition as well as the function definition such as

  ```
  int foo(int a[100]);
  ```

  by

  ```
  int foo(int a[], int n);
  ```

- For two-dimensional arrays, we can also abbreviate

  ```
  int foo(int a[100][200]);
  ```

  by

  ```
  int foo(int a[][200], int n);
  ```

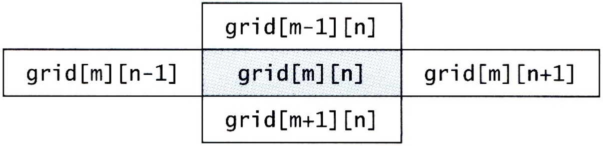
  Note that only the *first size* is abbreviated.

Main advantage: can use function over arrays of different sizes.

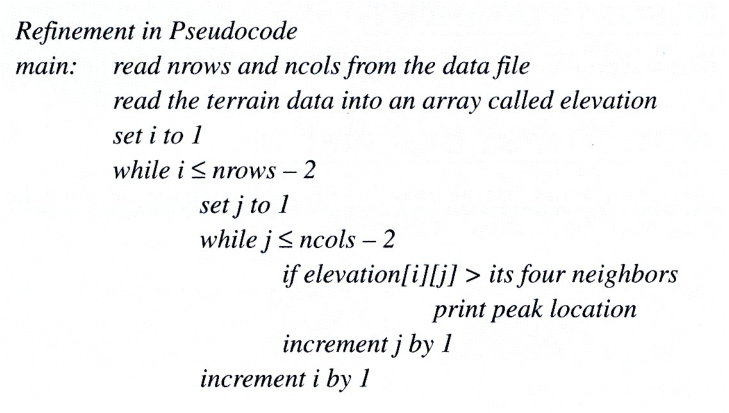
In order to use the function over two-dimensional arrays of varying sizes, we require pointers, covered later.

Example: Terrain Navigation

- Given: a *grid* representation of rectangular piece of land, and for each *grid element*, the *elevation* of the land at the the corresponding point.
- Represent the grid as a two dimensional array `grid[M][N]`.
- Problem: determine all *peaks*, ie. all grid positions whose elevation is higher than that of its neighbours.



Pseudocode



Refinement in Pseudocode

```
main:   read nrows and ncols from the data file
        read the terrain data into an array called elevation
        set i to 1
        while $i \leq nrows - 2$
            set j to 1
            while $j \leq ncols - 2$
                if elevation[i][j] > its four neighbors
                        print peak location
                increment j by 1
            increment i by 1
```

C Program Code

```
#include <stdio.h>
#define N 25

int main(void) {

int nrows, ncols, i, j;
double elev[N][N];
FILE *grid;

    if ((grid = fopen("grid.txt", "r")) == NULL)
        printf("Error opening file\n");
    else {
        fscanf(grid, "%d %d", &nrows, &ncols);
        for (i = 0; i <= nrows-1; i++)
            for (j = 0; j <= ncols-1; j++)
                fscanf(grid, "%lf", &elev[i][j]);
```

Code on Next Slide

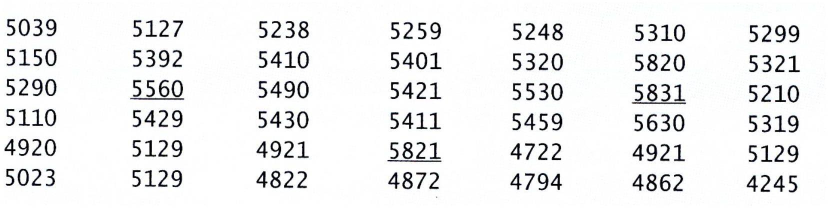
```
   /* Exit program. */
    fclose(grid);
    return 0;
}
```

Body code

```
printf("Top left point is row 0 column 0\n");
for (i = 1; i <= nrows-2; i++)
    for (j = 1; j <= ncols-2; j++)
        if (elev[i-1][j] < elev[i][j] &&
            elev[i+1][j] < elev[i][j] &&
            elev[i][j-1] < elev[i][j] &&
            elev[i][j+1] < elev[i][j])
            printf("Peak at row %d col %d\n", i, j);
```

Sample Data

Input:



5039	5127	5238	5259	5248	5310	5299
5150	5392	5410	5401	5320	5820	5321
5290	5560	5490	5421	5530	5831	5210
5110	5429	5430	5411	5459	5630	5319
4920	5129	4921	5821	4722	4921	5129
5023	5129	4822	4872	4794	4862	4245

Output:

```
Top left point defined as row 0 column 0
Peak at row:  2 column 1
Peak at row:  2 column 5
Peak at row:  4 column 3
```

Matrices and Vectors

- A *matrix* is a set of numbers arranged in a rectangular grid with rows and columns. Example of a 4×3 matrix:

$$\mathbf{A} = \begin{bmatrix} -1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & -2 & 3 \\ 0 & 2 & 1 \end{bmatrix}$$

- A matrix with one row is a *row vector*.
  A matrix with one column is a *column vector*.
  A matrix with the same number of rows and columns is a *square matrix*.

- A two-dimesional array is naturally used to store a matrix.

- We next exemplify some *operations* on matrices.

Dot Product

- The dot product[a] is a number computed from two vectors **A, B** of the same size.
  Suppose **A** is $[a_1, a_2, \cdots, a_n]$ and **B** is $[b_1, b_2, \cdots, b_n]$.
  Then the dot product **A . B** of **A** and **B** is:

$$\mathbf{A}.\mathbf{B} = \sum_{i=1}^{n} a_i b_i$$

- Example: suppose $\mathbf{A} = [4 \quad -1 \quad 3]$ and $\mathbf{B} = [-2 \quad 5 \quad 2]$, then:

$$\begin{aligned}\mathbf{A}.\mathbf{B} = \ & 4.(-2) + (-1).5 + 3.2 \\ & (-8) + (-5) + 6 \\ & -7\end{aligned}$$

[a] Also called an *inner product.*

C Program code

```
double dot_product(double a[], double b[], int n)
{
    int k;
    double sum = 0;

    for (k = 0; k <= n-1; k++)
        sum += a[k] * b[k];

    return sum;
}
```

IMPORTANT NOTE ABOUT NOTATION:

Matrix entries are denoted using subscripts starting from 1.
Arrays in C however use subscripts starting from 0.

Transpose

- The *tranpose* of a matrix is a new matrix in which the rows of the original matrix are columns of the new matrix.
  Notation: the tranpose of a matrix **B** is written $\mathbf{B}^T$.

- Example:

$$\mathbf{B} = \begin{bmatrix} 2 & 5 & 1 \\ 7 & 3 & 8 \\ 4 & 5 & 21 \\ 16 & 13 & 0 \end{bmatrix} \qquad \mathbf{B}^T = \begin{bmatrix} 2 & 7 & 4 & 16 \\ 5 & 3 & 5 & 13 \\ 1 & 8 & 21 & 0 \end{bmatrix}$$

C Program code

```c
/*-----------------------------------------------------*/
/* Transpose a matrix.                                 */
/*      Symbolic constants NROWS and NCOLS are declared */
/*      in the calling program                         */

void transpose(int b[NROWS][NCOLS], int bt[NCOLS][NROWS])
{
    int i, j;

    for (i = 0; i <= NROWS-1; i++)
        for (j = 0; j <= NCOLS-1; j++)
            bt[j][i] = b[i][j];

    return;
}
```

Matrix Addition and Subtraction

- Addition/Subtraction for matrices of the *same size*.
- operations defined by adding/subtracting elements in corresponding positions.
- Examples:

$$\mathbf{A} = \begin{bmatrix} 2 & 5 & 1 \\ 0 & 3 & -1 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 1 & 0 & 2 \\ -1 & 4 & -2 \end{bmatrix}$$

$$\mathbf{A}+\mathbf{B} = \begin{bmatrix} 3 & 5 & 3 \\ 0 & 3 & -1 \end{bmatrix} \qquad \mathbf{A}-\mathbf{B} = \begin{bmatrix} 1 & 5 & -1 \\ 1 & -1 & 1 \end{bmatrix}$$

$$\mathbf{B}-\mathbf{A} = \begin{bmatrix} -1 & -5 & 1 \\ -1 & 1 & -1 \end{bmatrix}$$

Matrix Multiplication

- The product **C** = **AB** of two matrices **A** and **B** is defined as follows: $c_{i,j}$ is the dot product of row i of **A** and column j of **B**:

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k}\, b_{k,j}$$

- Thus the row size of **A** must equal the column size of **B**: if **A** is $M \times K$ and **B** is $L \times N$, then $K = L$ and **AB** is $M \times N$.

- Example:

$$\mathbf{A} = \begin{bmatrix} 2 & 5 & 1 \\ 0 & 3 & -1 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 1 & 0 & 2 \\ -1 & 4 & -2 \\ 5 & 2 & 1 \end{bmatrix}$$

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} 2 & 22 & 5 \\ -8 & 10 & -7 \end{bmatrix}$$

- The product **BA** *does not exist*.

C Program code

```
/*-----------------------------------------------------------*/
/* Matrix Multiplication of two NxN matrices.                */
/* N is a symbolic constant defined in the calling program. */

void matrix_mult(int a[N][N], int b[N][N], int c[N][N])
{
    int i, j, k;

    for (i = 0; i <= N-1; i++)
        for (j = 0; j <= N-1; j++) {
            c[i][j] = 0;
            for (k = 0; k <= N-1; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
    return;
}
```

Exercise: Modify this function for matrices A[N][K] and B[K][M].

Solving Simultaneous Linear Equations (Non-Examinable)

- A *linear* equation is of the form

  $$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n$$

  where the x_i are variables and the a_i are *coefficients*.

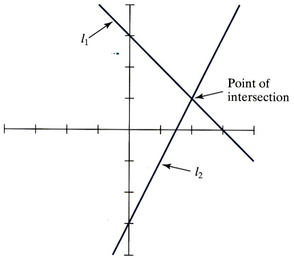
- A linear equation with 2 variables defines a *line*.
  Eg. $y = 2x + 3$ is a line whose *slope* is 2 and *y-intercept* is 3.

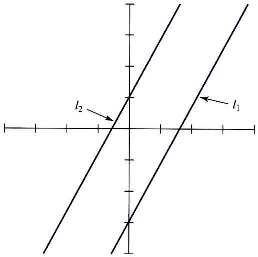
- A linear equation with 3 or more variables defines a *hyperplane*.

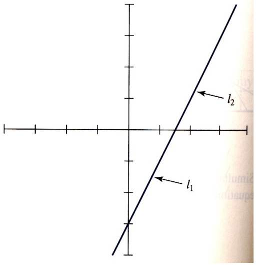
- Given a system of m equations in n unknowns (variables).
  If $m < n$: system is *underspecified* and there is no unique solution.
  If $m = n$: unique solution if none of the equations define *parallel* hyperplanes.
  If $m > n$: system is *overspecified* and there is no solution if none of the equations define parallel hyperplanes.
  A system with a unique solution is called *nonsingular*.

Solving Simultaneous Linear Equations

Intersecting, Parallel and Overlapping Lines

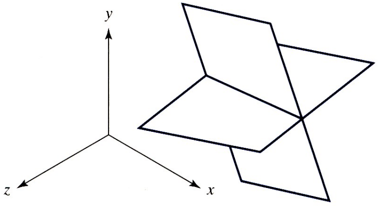




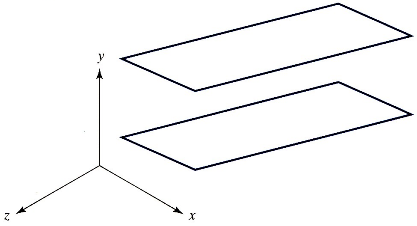


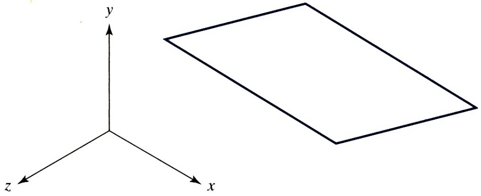
Solving Simultaneous Linear Equations

Intersecting, Parallel and Overlapping Planes



(a)





Solving Simultaneous Linear Equations

- Example: $$\begin{array}{rcrcrcr} 3x & + & 2y & - & z & = & 10 \\ -x & + & 3y & + & 2z & = & 5 \\ x & - & y & - & z & = & -1 \end{array}$$

- Can represent as the matrix product **AX** = **B** where

$$\mathbf{A} = \begin{bmatrix} 3 & 2 & -1 \\ -1 & 3 & 2 \\ 1 & -1 & -1 \end{bmatrix} \quad \mathbf{X} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 10 \\ 5 \\ -1 \end{bmatrix}$$

- We will describe the *Gaussian Method*.

Gauss Elimination (Non-Examinable)

- $$\begin{array}{rcrcrcrl} 3x & + & 2y & - & z & = & 10 & \text{(eqn 1)} \\ -x & + & 3y & + & 2z & = & 5 & \text{(eqn 2)} \\ x & - & y & - & z & = & -1 & \text{(eqn 3)} \end{array}$$

- $$\begin{array}{rcrcrcrl} x & + & \frac{2}{3}y & - & \frac{1}{3}z & = & \frac{10}{3} & \text{(eqn 1 times } \frac{1}{3}\text{)} \\ -x & + & 3y & + & 2z & = & 5 & \text{(eqn 2)} \\ \hline 0x & + & \frac{11}{3}y & - & \frac{5}{3}z & = & \frac{25}{3} & \text{sum} \end{array}$$

- $$\begin{array}{rcrcrcr} 3x & + & 2y & - & z & = & 10 \\ 0x & + & \frac{11}{3}y & - & \frac{5}{3}z & = & \frac{25}{3} \\ x & - & y & - & z & = & -1 \end{array}$$

Gauss Elimination

- $$\begin{array}{rcrcrcr} 3x & + & 2y & - & z & = & 10 \\ 0x & + & \frac{11}{3}y & - & \frac{5}{3}z & = & \frac{25}{3} \\ x & - & y & - & z & = & -1 \end{array}$$

- $$\begin{array}{rcrcrcrl} -x & - & \frac{2}{3}y & + & \frac{1}{3}z & = & -\frac{10}{3} & \text{(eqn 1 times } -\frac{1}{3}) \\ x & - & y & - & z & = & -1 & \text{(eqn 3)} \\ \hline 0x & - & \frac{5}{3}y & - & \frac{2}{3}z & = & -\frac{13}{3} & \text{sum} \end{array}$$

- $$\begin{array}{rcrcrcr} 3x & + & 2y & - & z & = & 10 \\ 0x & + & \frac{11}{3}y & - & \frac{5}{3}z & = & \frac{25}{3} \\ 0x & - & \frac{5}{3}y & - & \frac{2}{3}z & = & -\frac{13}{3} \end{array}$$

- Variable x is now *eliminated* from all but the first equation.

Gauss Elimination

- $$\begin{array}{rcrcrcrl} 3x & + & 2y & - & z & = & 10 & \text{(eqn 1)} \\ 0x & + & \frac{11}{3}y & - & \frac{5}{3}z & = & \frac{25}{3} & \text{(eqn 2)} \\ 0x & - & \frac{5}{3}y & - & \frac{2}{3}z & = & -\frac{13}{3} & \text{(eqn 3)} \end{array}$$

- $$\begin{array}{rcrcrcrl} 0x & + & \frac{5}{3}y & + & \frac{25}{33}z & = & \frac{125}{33} & \text{(eqn 2 times } \frac{5}{11}\text{)} \\ 0x & - & \frac{5}{3}y & - & \frac{2}{3}z & = & -\frac{13}{3} & \text{(eqn 3)} \\ \hline 0x & + & 0y & + & \frac{3}{33}z & = & -\frac{18}{33} & \text{sum} \end{array}$$

- $$\begin{array}{rcrcrcr} 3x & + & 2y & - & z & = & 10 \\ 0x & + & \frac{11}{3}y & + & \frac{5}{3}z & = & \frac{25}{3} \\ 0x & + & 0y & + & \frac{3}{33}z & = & -\frac{18}{33} \end{array}$$

- Variables x, y is now *eliminated* from all but the first and second equations.

Back Substitution (Non-Examinable)

- In the last equation $\frac{3}{33}z = -\frac{18}{33}$, ie $z = -6$.
- *Substituting* this into the second last equation $\frac{11}{3}y - \frac{5}{3}z = \frac{25}{3}$ we get $y = \frac{165}{33}$, ie. $y = 5$.
- Substituting both $z = -6$ and $y = 5$ into the first equation $3x + 2y - z = 10$ we get $x = -2$.
- So we have our solution: $x = -2, y = 5, z = -6$.

Summary on Gauss Elimination and Back Substitution

- The Gaussian Method consisted of two parts: Gauss Elimination E and BS.

- **Gauss Elimination** Equations are modified so that the k^{th} variable is eliminated from all equations *after* the k^{th} equation.

- **Back Substitution** Starting with the last equation, we compute the value of the last variable. Using this last value and the *second-last* equation, we compute the value of the second-last variable. This process terminates successfully if the system has a unique solution, in which case all the variable values are computed.

- The above process *fails* when *all* the coefficients of one variable is zero. We also say that the system of equations is *ill-conditioned*.

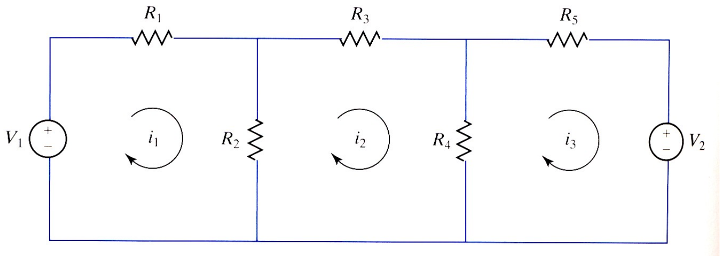
Example: Electrical Circuit Analysis

Analysis of an electrical circuit often involves solving a set of simultaneous equations. These equations are either

- current equations describing currents entering and leaving a node, or
- voltage equations describing voltages around mesh loops in the circuit.

We next provide an example using voltages:

Example: Electrical Circuit Analysis



The voltages around the three loops can be described with the following equations[a] :

$$\begin{aligned}
-V_1 &+ R_1 i_1 + R_2(i_1 - i_2) = 0,\\
R_2(i_2 - i_1) &+ R_3 i_2 + R_4(i_2 - i_3) = 0,\\
R_4(i_3 - i_2) &+ R_5 i_3 + V_2 = 0.
\end{aligned}$$

[a] You are not required to understand why this is so.

Example: Electrical Circuit Analysis

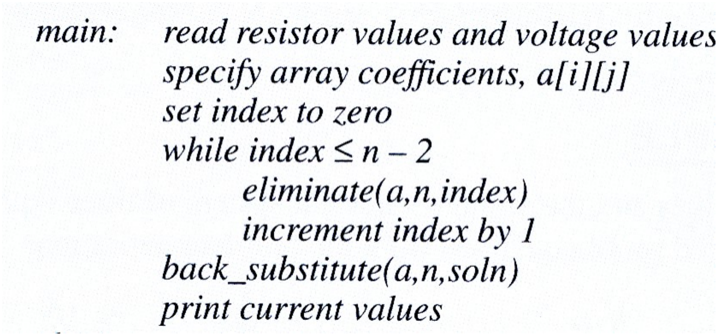
Assuming the values of the resistors (R_1, R_2, R_3, R_4, R_5) are known and the voltage sources (V_1, V_2) are known, we can rearrange the equations as follows:

$$\begin{aligned}
(R_1 + R_2)i_1 &- R_2 i_2 &+ 0 i_3 &= V_1, \\
-R_2 i_1 &+ (R_2 + R_3 + R_4) i_2 &- R_4 i_3 &= 0, \\
0 i_1 &- R_4 i_2 &+ (R_4 + R_5) i_3 &= -V_2.
\end{aligned}$$

For example, suppose that each of the resistor values is 1 ohm, and assume the both the voltage sources are 5 volts. Then the corresponding set of equations is:

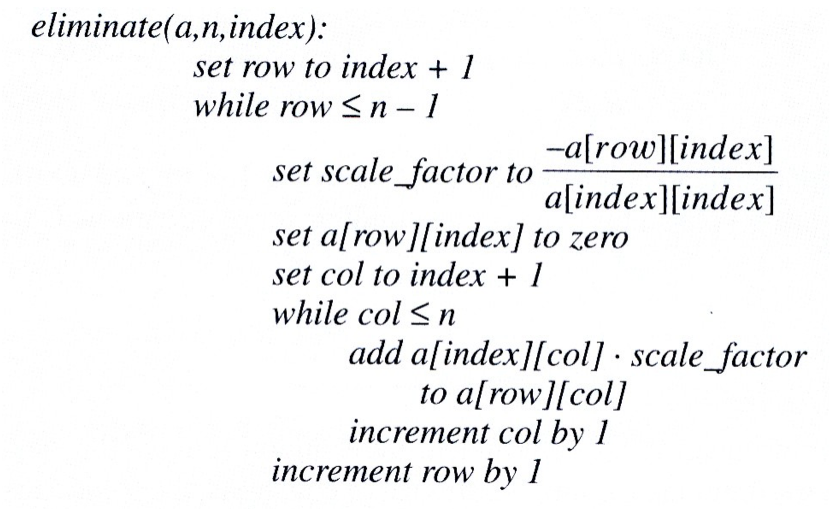
$$\begin{aligned}
2i_1 &- i_2 &+ 0i_3 &= 5, \\
-i_1 &+ 3i_2 &- i_3 &= 0, \\
0i_1 &- i_2 &+ 2i_3 &= -5.
\end{aligned}$$

Pseudocode



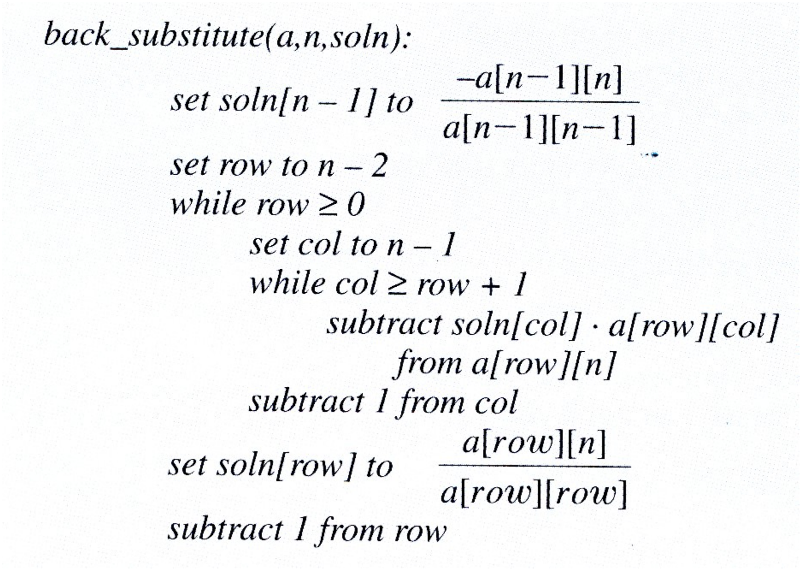
```
main:   read resistor values and voltage values
        specify array coefficients, a[i][j]
        set index to zero
        while index $\leq n - 2$
            eliminate(a,n,index)
            increment index by 1
        back_substitute(a,n,soln)
        print current values
```

Pseudocode



```
eliminate(a,n,index):
        set row to index + 1
        while row $\leq n - 1$
            set scale_factor to $\frac{-a[row][index]}{a[index][index]}$
            set a[row][index] to zero
            set col to index + 1
            while col $\leq n$
                add a[index][col] $\cdot$ scale_factor
                    to a[row][col]
                increment col by 1
            increment row by 1
```

Pseudocode

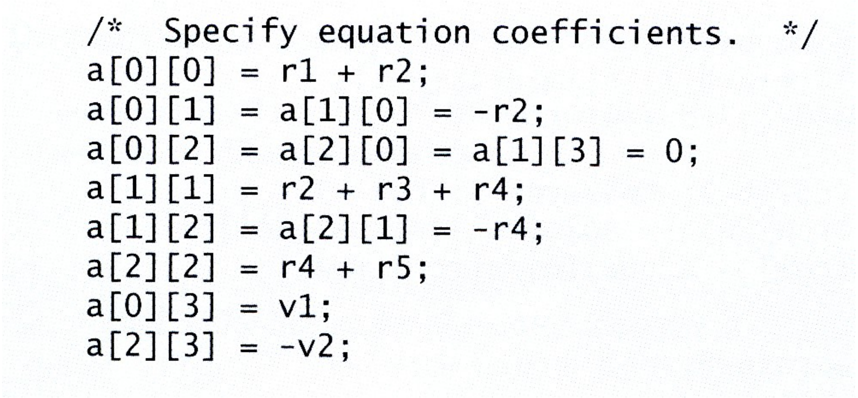


```
back_substitute(a,n,soln):
        set soln[n – 1] to $\frac{-a[n-1][n]}{a[n-1][n-1]}$
        set row to n – 2
        while row ≥ 0
                set col to n – 1
                while col ≥ row + 1
                        subtract soln[col] · a[row][col]
                            from a[row][n]
                subtract 1 from col
        set soln[row] to $\frac{a[row][n]}{a[row][row]}$
        subtract 1 from row
```

C Program Code

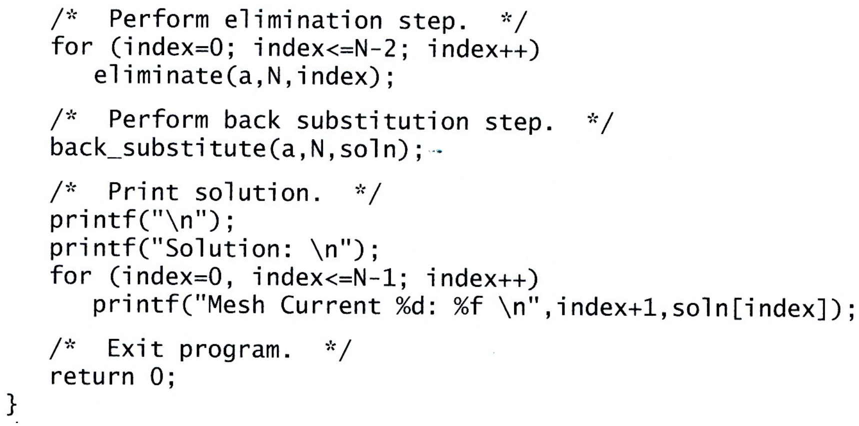
```
/*-----------------------------------------------------------------*/
/*  Program chapter5_8                                             */
/*                                                                 */
/*  This program uses Gauss elimination to determine the           */
/*  mesh currents for a circuit.                                   */

#include <stdio.h>
#define N 3  /*  number of unknown currents  */

int main(void)
{
   /*  Declare variables and function prototypes.  */
   int index;
   double r1, r2, r3, r4, r5, v1, v2, a[N][N+1], soln[N];
   void eliminate(double a[N][N+1],int n,int index);
   void back_substitute(double a[N][N+1],int n,
                        double soln[N]);

   /*  Get user input.  */
   printf("Enter resistor values in ohms: \n");
   printf("(R1, R2, R3, R4, R5) \n");
   scanf("%lf %lf %lf %lf %lf",&r1,&r2,&r3,&r4,&r5);
   printf("Enter voltage values in volts: \n");
   printf("(V1, V2) \n");
   scanf("%lf %lf",&v1,&v2);
```

C Program Code



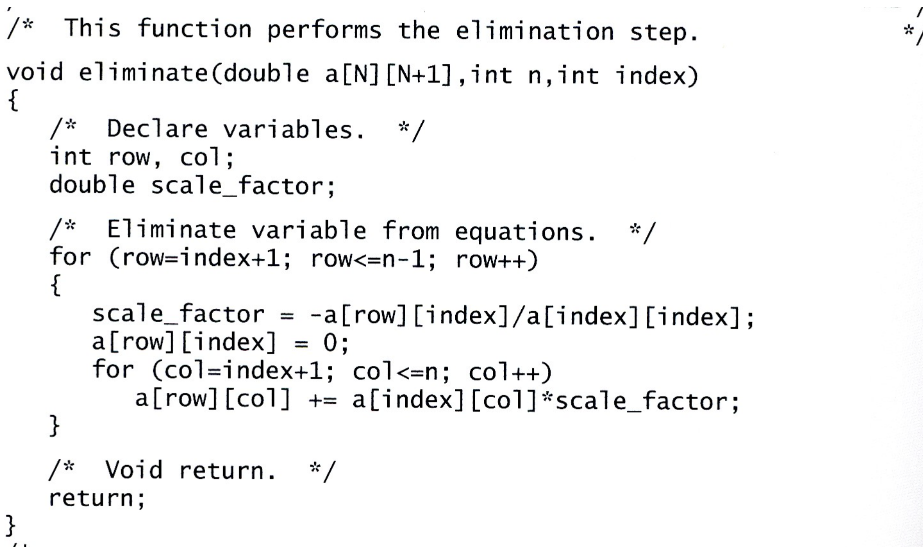
```
/*  Specify equation coefficients.  */
a[0][0] = r1 + r2;
a[0][1] = a[1][0] = -r2;
a[0][2] = a[2][0] = a[1][3] = 0;
a[1][1] = r2 + r3 + r4;
a[1][2] = a[2][1] = -r4;
a[2][2] = r4 + r5;
a[0][3] = v1;
a[2][3] = -v2;
```

C Program Code



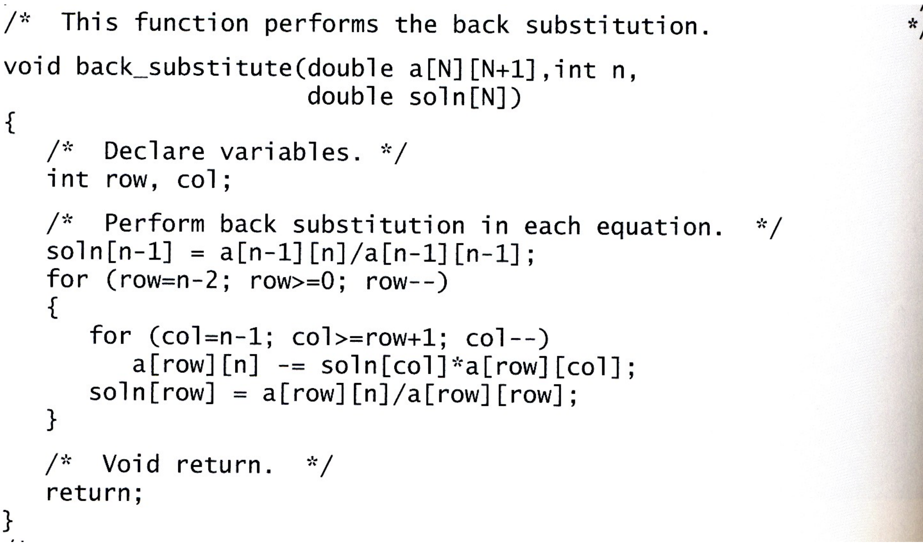
```
    /*  Perform elimination step.  */
    for (index=0; index<=N-2; index++)
       eliminate(a,N,index);

    /*  Perform back substitution step.  */
    back_substitute(a,N,soln);

    /*  Print solution.  */
    printf("\n");
    printf("Solution: \n");
    for (index=0, index<=N-1; index++)
       printf("Mesh Current %d: %f \n",index+1,soln[index]);

    /*  Exit program.  */
    return 0;
}
```

C Program Code



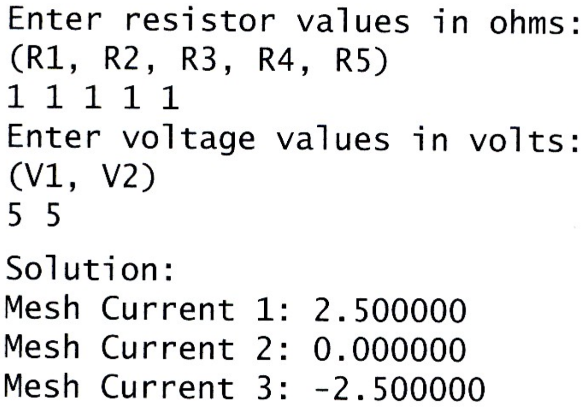
```
/*  This function performs the elimination step.                     */

void eliminate(double a[N][N+1],int n,int index)
{
   /*  Declare variables.  */
   int row, col;
   double scale_factor;

   /*  Eliminate variable from equations.  */
   for (row=index+1; row<=n-1; row++)
   {
      scale_factor = -a[row][index]/a[index][index];
      a[row][index] = 0;
      for (col=index+1; col<=n; col++)
         a[row][col] += a[index][col]*scale_factor;
   }

   /*  Void return.  */
   return;
}
```

C Program Code



```
/*  This function performs the back substitution.                  */

void back_substitute(double a[N][N+1],int n,
                     double soln[N])
{
   /*  Declare variables. */
   int row, col;

   /*  Perform back substitution in each equation.  */
   soln[n-1] = a[n-1][n]/a[n-1][n-1];
   for (row=n-2; row>=0; row--)
   {
      for (col=n-1; col>=row+1; col--)
         a[row][n] -= soln[col]*a[row][col];
      soln[row] = a[row][n]/a[row][row];
   }

   /*  Void return.  */
   return;
}
```

Testing



```
Enter resistor values in ohms:
(R1, R2, R3, R4, R5)
1 1 1 1 1
Enter voltage values in volts:
(V1, V2)
5 5

Solution:
Mesh Current 1: 2.500000
Mesh Current 2: 0.000000
Mesh Current 3: -2.500000
```

Higher Dimensional Arrays (Non-Examinable)

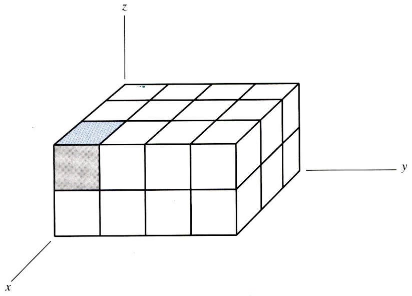
C allows arrays to be defined with more than two subscripts.

For example, the following declares a *three-dimensional* array:

```
int b[3][4][2];
```

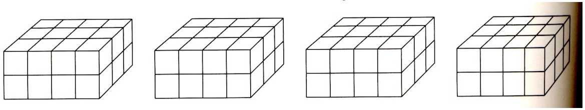
The three subscripts correspond to the *x*, *y*, *z* coordinates.

Positioning the array at the origin, the shaded area corresponds to `b[2][0][1]`:

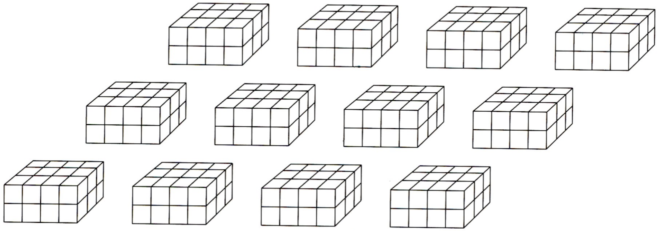


Higher Dimensional Arrays

A *four-dimensional* area is depicted as follows:



A *five-dimensional* area is depicted as follows:



Note: Arrays with over three subscripts are seldom used. Instead, *structures* are used – this is covered later.

References

Etter Sections 5.8 to 5.13

Next Lecture

Programming with Pointers