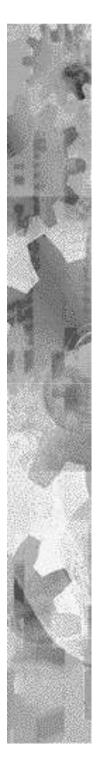
## Computers in Engineering COMP 208

Linear Algebra Michael A. Hawker

## **Representing Vectors**

- A vector is a sequence of numbers (the components of the vector)
- If there are n numbers, the vector is said to be of dimension n
- To represent a vector in C, we use an array of size n, indexed from 0 to n-1
- In Fortran we use an array indexed from
   1 to n



## **Vector Operations**

#### Scaling

 Multiply each element by a given scalar factor

#### Adding and Subtracting

 Given two vectors of the same dimension add the components to get a new vector of the same dimension



#### Dot Product

- Sum the Products of Vector Components
- Vector Norm
  - Length of the Vector, Square-root of the sum of squares of Components



### **Dot Product**

```
#include <math.h>
double vector dot(double v1[], double v2[],
                   int size) {
  int i;
  double dot = 0.0;
  for(i = 0; i < size; i++)</pre>
    dot += v1[i] * v2[i];
  return dot;
}
double vector norm(double v[], int size)(
  return sqrt(vector dot(v, v, size)
}
```

# Read a Vector

```
void fscan_vector(FILE * in, double v[], int size){
    int i;
```

```
for(i = 0; i < size; i++) {
   fscanf(in, "%lf", &v[i]);
}</pre>
```

```
void scan_vector(double v[], int size){
  fscan vector(stdin, v, size);
```

## Output a Vector

```
void fprint_vector(FILE * out, double v[], int size){
    int i;
```

```
fprintf(out, "{");
for(i = 0; i < size - 1; i++)
    fprintf(out, "%g, ", v[i]);
fprintf(out, "%g}\n", v[i]);</pre>
```

```
return;
```

}

```
void print_vector(double v[], int size)
{
```

```
fprint_vector(stdout, v, size);
```

```
return;
```

Nov. 29th, 2007

## **Possible Confusion**

```
for(i = 0; i < size - 1; i++)
    fprintf(out, "%g, ", v[i]);
fprintf(out, "%g}\n", v[i]);</pre>
```

#### Does Indentation Always Dictates Meaning?

```
for(i = 0; i < size - 1; i++)
fprintf(out, "%g, ", v[i]);
fprintf(out, "%g}\n", v[i]);</pre>
```

for(i = 0; i < size - 1; i++)
fprintf(out, "%g, ", v[i]);
fprintf(out, "%g}\n", v[i]);</pre>

#### Same Results

## Output a Vector

```
void fprint_vector(FILE * out, double v[], int size){
    int i;
```

```
fprintf(out, "{");
for(i = 0; i < size - 1; i++) {
   fprintf(out, "%g, ", v[i]);
}
fprintf(out, "%g}\n", v[i]);</pre>
```

```
return;
```

```
}
```

```
void print_vector(double v[], int size)
{
   fprint_vector(stdout, v, size);
   return;
}
```

## **Representing Matrices**

A matrix with m rows and n columns can be represented as a two dimensional array in C (or Fortran).

# In C the declaration could be double voltage[m][n];

The first dimension is the number of rows and the second the number of columns

A specific value in row i, column j is referenced as voltage[i][j]



## Initialization

We can initialize a matrix (or any array) when it is declared:

int val[3][4] = {{8,16,9,24}, {3,7,19,25},  $\{42,2,4,12\}$ ;

## Row Major Ordering

#### What happens if we write

int val[3][4] =
{{8,16,9,24,3,7,19,25,42,2,4,12}};

We begin filling in values starting with v[0][0] and continue.

If the array is stored in row major order, this has the same effect as the previous example

## Implementing Row Major Order

- We can simulate a matrix using a one dimensional array by taking the two indices and finding the position in row major order.
- We have to know how many columns there are, that is the number of elements in each row.

```
int in2d(int row, int col, int n) {
   return col + row * n;
}
```

#### Simulating Matrices in One Dimension

- In the previous example we showed how to simulate a matrix by a one dimensional vector.
- This may be done in some applications to make highly computational intensive programs more efficient
- We could also simulate a matrix with a one dimensional array that stores the values in column major order
- Imagine adding one to every element?
- Used with other Data Structures as well

## Input of Matrix

```
return;
```

}

}

```
void scan_matrix(double **m, int h, int w){
  fscan_matrix(stdin, m, h, w);
  return;
```

## \*\*, What about [][]?

Why can't we use [][] in our function arguments:

- C needs to know the length of the second dimension!
- Need to use a double pointer if we want to allow for completely dynamic matrices

# How do we allocate a dynamic matrix?

```
double ** make matrix(int h, int w) {
    int i;
    double **array2 = (double **)malloc(h * sizeof(double *));
    if (array2) {
        array2[0] = (double *)malloc(h * w * sizeof(double));
        if(array2[0]) {
            for (i = 1; i < h; i++)
                array2[i] = array2[0] + i * w;
            return array2;
        } else {
            free(array2);
        }
    }
    return NULL;
```

## Matrix Output

```
void fprint matrix(FILE * out, double **m, int h, int w) {
  int i, j;
  fprintf(out, "{\n");
  for (i = 0; i < h - 1; ++i) {
    fprintf(out, " {");
    for (j = 0; j < w - 1; ++j)
      fprintf(out, "%g, ", m[i][j]);
    fprintf(out, "%g},\n", m[i][j]);
  fprintf(out, " {");
  for (j = 0; j < w - 1; ++j)
    fprintf(out, "%g, ", m[i][j]);
  fprintf(out, "%g}\n}\n", m[i][j]);
  return;
void print matrix(double **m, int h, int w) {
  fprint matrix(stdout, m, h, w);
  return;
Nov. 29th, 2007
                           Linear Algebra
```

## Matrix Transposition

- A common operation is to compute the transpose of a matrix
- We could do this in place and overwrite the contents of the matrix
- In the following algorithm, we compute a new matrix containing the transposed matrix

## **Matrix Transposition**

```
double ** matrix_transpose(double ** m1, int h, int w) {
    int i, j;
```

```
double ** mr = make_matrix(w, h);
```

```
if (mr) {
  for(i = 0; i < h; ++i)
    for(j = 0; j < w; ++j)
    mr[j][i] = m1[i][j];</pre>
```

```
return mr;
} else {
  return NULL;
```

Nov. 29th, 2007

}



#### Example

```
int main() {
  //double m[4][3] = \{ \{0, 1, 2\}, \{2, 3, 4\}, \{5, 6, 7\}, \{9, 1, 0\} \};
  int h = 2, w = 3;
  double ** m = make_matrix(h, w);
  scan matrix(m, h, w);
  print matrix(m, h, w);
  double ** mt = matrix transpose(m, h, w);
  if (mt) {
    print matrix(mt, w, h);
    free matrix(mt);
  free matrix(m);
  return 0;
Nov. 29th, 2007
                                Linear Algebra
```

# Matrix Multiplication

- Matrix multiplication is a fundamental operation that occurs in many applications
- Given two matrices A, a matrix with h1 rows and w1 columns and B a matrix with w1 rows and h2 columns, we can compute their product matrix C
- Note that the number of columns of A must equal the number of rows of B

## Matrix Multiplication

- The element c[i][j] is computed as the dot product of the ith row of A and the jth column of B
- The overall algorithm computes has two nested loops that vary i and j, computing each dot product
- The computation of the dot product is done in another loop nested inside those two

## **Matrix Multiplication**

```
double ** matrix mult(double **m1, double **m2,
                      int hm1, int wm1, int wm2) {
  int i, j, k;
  double sum;
  double ** mr = make matrix(hm1, wm2);
  if (mr) {
     for(i = 0; i < hm1; ++i) {</pre>
         for (j = 0; j < wm2; ++j) {
           sum = 0;
           for (k = 0; k < wm1; ++k) {
                  sum += m1[i][k] * m2[k][j];
           }
           mr[i][j] = sum;
     }
  return mr;
```

## Solving Linear Systems

- One of the most widespread applications of computers is the solving of systems of linear equations
- These systems arise in numerous application areas
- There is a large body of literature and research on how to solve these systems efficiently and accurately
- We examine two simple approaches

Nov. 29th, 2007

Linear Algebra

## An Easy Example

If the system of equations is triangular, we can solve it by a process called back substitution:

w - 1.5x + y + 2.5z = 1.5 x + 0y - z = -1 y + 0z = -2z = 7

## Matrix Representation

We can represent this system of equations using an upper triangular matrix, A and a vector b. The equations can be written Ax=b, where x is a vector of length 4 representing the values of (w,x,y,z)



## Matrix Representation

$$A = 1 - 1.5 \quad 1 \ 2.5 \quad b = (1.5) \\ 0 \quad 1 \quad 0 \ -1 \quad -1 \\ 0 \quad 0 \quad 1 \quad 0 \quad -2 \\ 0 \quad 0 \quad 0 \quad 1 \quad 7)$$

## **Back Substitution**

First solve for z and then substitute in the previous equation to solve for y. Continue until all of the variables have been

solved.

$$z = 7$$

$$y = -2 - 0*7 = -2$$
  

$$x = -1 - 0*-2 + 7 = 6$$
  

$$w = 1.5 + 1.5*6 - (-2) - 2.5*7 = -5$$

Linear Algebra

## Gaussian Elimination

- The Gaussian elimination algorithm attempts to transform a system of linear equations into a triangular system
- As we have seen by example, a triangular system is easy to solve by back substitution
- We transform the system by eliminating one variable at each step

## A Linear System Example

Consider the system of equations:

$$2w - 3x + 2y + 5z = 3$$
  
 $w - x + y + 2z = 1$   
 $3w + 2x + 2y + z = 0$   
 $w + x - 3y - z = 0$ 

## A Linear System Example

Again we can write this in the form Ax=b where A is a 4x4 matrix, x is a 1x4 vector and b is a 1x4 vector:

A	b:				
2	-3	2	5		3
1	-1	1	2		1
3	2	2	1		0
1	1	3	-1		0

We first eliminate the first entry in the second row, by multiplying the first row by 1.0/2.0 and subtracting the rows.

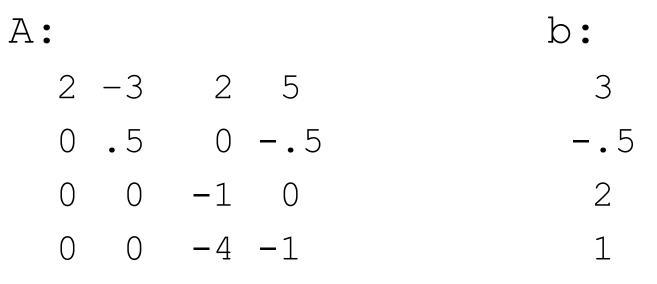
We do the same to the second entry in b.

Α:				b:
2	-3	2	5	3
0	• 5	0	<b>-</b> .5	<b></b> 5
3	2	2	1	0
1	1	3	-1	0

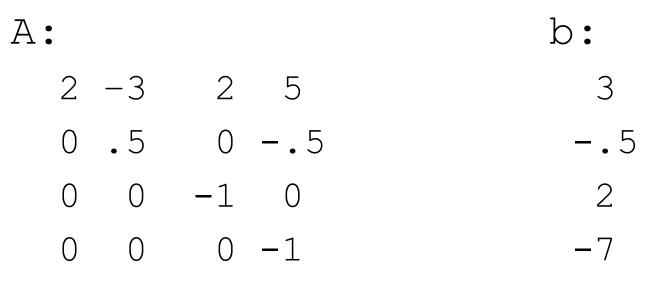
Repeat this process for each row

A:				b:
2	-3	2	5	3
0	• 5	0	5	5
0	6.5	-1	-6.5	-4.5
0	2.5	-4	-3.5	-1.5

Now eliminate the second non-zero entries in the second column below the diagonal in the same way



Do the same for the third column. Notice that it is not necessary to continue with the last column



## **Gaussian Elimination**

```
void genp(double **m, double v[], int h, int w){
    int row, next_row, col;
    double factor;
```

```
for(row = 0; row < (h - 1); ++row) {
   for(next_row = row + 1; next_row < h; ++next_row) {
     factor = m[next row][row] / m[row][row];
</pre>
```

```
for(col = 0; col < w; ++col)
    m[next row][col] -= factor * m[row][col];</pre>
```

```
v[next row] -= factor * v[row];
```

}

#### Problems with Gaussian Elimination

- If there is a zero on the diagonal that of the row we are processing, there will be an attempt to divide by zero, causing an error
- Even if there isn't a zero, dividing by a small number causes large roundoff errors and inaccurate results.
- These problems can be reduced by pivoting
- We rearrange the rows at each step so that the largest possible value is the next one we chose to eliminate

### Gaussian Elimination with Partial Privoting

void gepp(double \*\*m, double v[], int h, int w){
 int row, next\_row, col, max\_row;
 double tmp, factor;

for(row = 0; row < (h - 1); ++row) {

// Find row with largest pivot.

// Swap rows.

// Rest like Gaussian Elimination without Pivoting.

}



#### Finding a Pivot

```
max_row = row;
for(next_row = row + 1; next_row < h; ++next_row)
if(m[next_row][row] > m[max_row][row])
max row = next row;
```



## Swapping Two Rows

```
if(max_row != row) {
  for(col = 0; col < w; ++col) {
    tmp = m[row][col];
    m[row][col] = m[max_row][col];
    m[max_row][col] = tmp;
    }
  tmp = v[row];
  v[row] = v[max_row];
  v[max_row] = tmp;
}</pre>
```

## **Back Substitution**

- Once we have an upper triangular matrix, we can solve the system of equations by back substitution
- We first solve for the last variable and use the solution to solve for the second last and so on.

## **Back Substitution**

```
void back substitute(double **m, double v[],
                     int h, int w) {
  int row, next row;
  for (row = h - 1; row >= 0; --row) {
    v[row] /= m[row] [row];
   m[row][row] = 1;
    for(next row = row - 1; next row >= 0; --next row)
      v[next row] -= v[row] * m[next row][row];
      m[next row][row] = 0;
```