Introduction to Programming Language (C)

(EEF 110E)

Dr. Emre DİNCEL

LECTURE NOTES

(WEEK 12)

Mathematics Applications in C

In this section, it is aimed to give the theoretical background for mathematical applications such as matrix operations, numerical solutions to the linear/non-linear equations, curve fitting, etc. Furthermore, the given theoretical background is desired to be implemented in C environment via programming; thus, it will be shown that many engineering problems can be formulated and solved easily by taking advantage of the programming tools.

- Applications of Linear Algebra:

Matrix Operations:

As shown earlier, matrices are actually multi-dimensional arrays. It is possible to perform different operations with matrices such as addition, subtraction, multiplication, inverse, transpose, etc.

Let us define some matrices (and vectors) in C programming environment.

```
double T[2][2] = {{4, 1},
                  {2, 2}};
double X[3][3] = {{-1,  3.0,   5},
                  {2.5, 1.6,  -7},
                  {1,   1.1, 4.5}};
double Y[3][3] = {{5.0, 2.2, 0.4},
                  {1.5, 5.1, 0.0},
                  {0.2, 0.8, 3.0}};
double Z[4][5] = {{1,  2,  0, -5, -2},
                  {0,  1,  3,  2,  5},
                  {2, -5,  4,  4, -1},
                  {1,  0, -2,  6, -1}};
double U[3] = {1, 2, 4};
double V[3] = {-2, 0, 3.2};
double *RESULT;
complex *cResult;   // It is a struct with double Re, Im and defined in the .h file given to you
```

It is possible to write suitable functions to perform matrix addition, subtraction and multiplication. It should be noted that the matrix dimensions should match to perform the matrix operations.

The functions, written in C language to perform matrix operations, are given follow:

```c
double* Add(double *A, double *B, int n, int m)
{
	int i,j;
	double *C=malloc(n*m*sizeof(double));

	for(i=0;i<n;i++)
		for(j=0;j<m;j++)
			*(C+i*m+j)=*(A+i*m+j)+*(B+i*m+j);
	return C;
}

double* Subs(double *A, double *B, int n, int m)
{
	int i,j;
	double *C=malloc(n*m*sizeof(double));

	for(i=0;i<n;i++)
		for(j=0;j<m;j++)
			*(C+i*m+j)=*(A+i*m+j)-*(B+i*m+j);
	return C;
}
```

Here, the user is responsible to send valid matrices to the functions (it is also possible to implement a validation system in the functions). The functions can be used as follows to calculate the addition and subtraction:

```c
// Perform X+Y
printf("X+Y:\n");
RESULT = Add((double*)X,(double*)Y,3,3);
ShowMatrix((double*)RESULT,3,3);
free(RESULT);

// Perform X-Y
printf("X-Y:\n");
RESULT = Subs((double*)X,(double*)Y,3,3);
ShowMatrix((double*)RESULT,3,3);
free(RESULT);
```

Here, the “ShowMatrix” function is used to print the resulting matrices to the screen as given as follows:

```
void ShowMatrix(double *A, int n, int m)
{
      int i,j;

      for(i=0;i<n;i++)
      {
            for(j=0;j<m;j++)
                  printf("%.3lf\t",*(A+i*m+j));
            printf("\n");
      }
      printf("\n");
}
```

Matrix multiplication can be tricky; nevertheless, it can easily be implemented in C language as follows (n x m matrix is multiplied by m x p matrix):

```
double* Multiply(double *A, double *B, int n, int m, int p)
{
      int i,j,k;
      double *C=calloc(n*p,sizeof(double));

     for(i=0;i<n;i++)
        for(j=0;j<p;j++)
            for(k=0;k<m;k++)
                 *(C+i*p+j)+=*(A+i*m+k)**(B+k*p+j);
    return C;
}
```

Please, however, note that the multiplication operation is not commutative in matrices (i.e. $AB \neq BA$). You can check this issue with the help of the following code segment.

```
// Perform X*Y
printf("X*Y:\n");
RESULT = Multiply((double*)X,(double*)Y,3,3,3);
ShowMatrix((double*)RESULT,3,3);
// Perform Y*X
printf("Y*X:\n");
RESULT = Multiply((double*)Y,(double*)X,3,3,3);
ShowMatrix((double*)RESULT,3,3);
free(RESULT);
```

Identity Matrix:

$$I_n = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & 0 & \ddots & 0 & \vdots \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix}$$

An identity matrix is *n*×*n* square matrix with ones on the main diagonal and zeros elsewhere. An identity matrix can also be defined and used in C language as necessary.

Transpose:

When a transpose operation is applied to a matrix, a new matrix, whose columns are the rows of the original matrix (and rows are the columns of the original matrix), is created.

Transpose function in C language can be written as follows:

```
double* Transpose(double* A, int n, int m)
{
      int i,j;
      double *C=malloc(n*m*sizeof(double));


      for(i=0;i<m;i++)
            for(j=0;j<n;j++)
                  *(C+i*n+j)=*(A+j*m+i);
      return C;
}
```

The written transpose code can also be used as follows:

```c
// Transpose
printf("Z^T:\n");
RESULT = Transpose((double*)Z,4,5);
ShowMatrix((double*)RESULT,5,4);
free(RESULT);
```

Determinant:

An $n\,x\,n$ matrix actually describes a linear transformation in n-dimensional space ($T\colon R^n \to R^n$). Determinant of a matrix determines the scaling factor in this transformation. It means that an n-dimensional content (length if n = 1, area if n = 2, volume if n = 3, etc.) will be scaled by the value of the determinant during such transformation.

In order to calculate the determinant, the cofactors should be calculated at first. In linear algebra, a minor of a matrix *A* is the determinant of some smaller square matrix, cut down from *A* by removing one or more of its rows and columns. Minors obtained by removing just one row and one column from square matrices (first minors) are required for calculating matrix ***cofactors***, which in turn are useful for computing both the determinant and inverse of square matrices.

The following code can be used to find the cofactor matrix:

```c
double* CoFactor(double *A, int n, int m, int p, int q)
{
	int i,j,k=0,t=0;
	double *C=malloc((n-1)*(m-1)*sizeof(double));

	for(i=0;i<n;i++)
		for(j=0;j<m;j++)
			if(i!=p && j!=q)
			{
				*(C+k*(m-1)+(t++))=pow(-1,i+j)**(A+i*m+j);
				if(t==m-1)
				{
					t=0;
					k++;
				}
			}
	return C;
}
```

Determinant of an n-by-n matrix can then be calculated with the following formula:

$$\det(A) = \sum_{i=1}^{n} a_{ij} C_{ij}$$

Finally, the following recursive function in C language is able to calculate the determinant of the given n x n matrix.

```c
double Determinant(double *A, int n)
{
	int i;
	double det=0.0, *coFact;

	if(n==1) return *A;
	for(i=0;i<n;i++)
	{
		coFact=CoFactor(A,n,n,0,i);
		det+=*(A+i)*Determinant(coFact,n-1);
	}
	free(coFact);
	return det;
}
```

The usages of the functions are then;

```c
// CoFactors
RESULT = CoFactor((double*)Z,4,5,1,2);
ShowMatrix((double*)RESULT,3,4);
free(RESULT);

// Determinant
printf("det(T) = %.3lf\n\n",Determinant((double*)T,2,2));
printf("det(X) = %.3lf\n\n",Determinant((double*)X,3,3));
```

Matrix Trace:

Trace of a square matrix *A* is defined to be the sum of elements on the main diagonal. It is easy to calculate the trace of the given matrix with the help of the following code segment in C language:

```
double Trace(double *A, int n)
{
      int i,j;
      double tr=0.0;

      for(i=0;i<n;i++)
            for(j=0;j<n;j++)
                  if(i==j)
                        tr+=*(A+i*n+j);
      return tr;
}
```

Please note that, trace also gives the sum of the eigenvalues of the considered matrix.

Dot and Cross Product:

Dot product (also known as scalar product) is a vector operation, in which two equal-length sequences of numbers are taken and a single number is returned. The operation is mathematically given as follows:

$$A.B = a_1b_1 + \cdots + a_nb_n$$

Cross product, on the other hand, is another vector operation on two vectors in three-dimensional space. Consider two linearly independent vectors A and B, the cross product is a vector which is perpendicular to both A and B (i.e. normal to the plane containing them).

$$A = [a_1\ a_2\ a_3\ ],\ \ B = [b_1\ b_2\ b_3]$$

$$AxB = [a_2b_3 - a_3b_2 \quad a_3b_1 - a_1b_3 \quad a_1b_2 - a_2b_1]$$

It is possible to use the following functions to find the dot and cross products of the given two vectors:

```
double DotProduct(double *U, double *V, int n)
{
      int i;
      double sum=0.0;

      for(i=0;i<n;i++)
            sum+=*(U+i)**(V+i);
      return sum;
}
```

```
double* CrossProduct(double *U, double *V)
{
      int i;
      double *W=malloc(3*sizeof(double));

      for(i=0;i<3;i++)
            *(W+i)=*(U+(i+1)%3)**(V+(i+2)%3)-*(U+(i+2)%3)**(V+(i+1)%3);
      return W;
}
```

The functions can then be used as illustrated below.

```
// Dot Product
printf("U.V:\n");
printf("%.3lf\n\n",DotProduct(U,V,3));
// Cross Product
printf("UxV:\n");
RESULT = CrossProduct(U,V);
ShowMatrix(RESULT,1,3);
free(RESULT);
```

Matrix Inverse:

In linear algebra, an n x n square matrix A is called invertible if there exists an n x n square matrix B such that

$$A\,B = I_n$$

where I_n denotes the n x n identity matrix. So the matrix $B = A^{-1}$ is called as the inverse of the matrix *A*. The inverse is calculated with the help of the cofactor matrix and the determinant of the *A* matrix.

The following code segment calculates the inverse of a square matrix.

```
// Matrix Inverse
printf("X^-1:\n");
RESULT = Inverse((double*)X,3);
ShowMatrix((double*)RESULT,3,3);
free(RESULT);
```

where the corresponding function, written in C language, is given as follows:

```
double* Inverse(double *A, int n)
{
	int i,j;
	double *C=malloc(n*n*sizeof(double));
	double *coFact, detA;

	detA=Determinant(A,n);
	for(i=0;i<n;i++)
		for(j=0;j<n;j++)
		{
			coFact=CoFactor(A,n,n,i,j);
			*(C+j*n+i)=Determinant(coFact,n-1)/detA;
			free(coFact);
		}
	return C;
}
```

Eigenvalues:

Eigenvalues and eigenvector have a significant importance in control theory. The eigenvalues give us the largest effect on input in magnitude and the eigenvectors define its direction. It means that if a specific input is applied in the direction of eigenvector, the input will be scaled (i.e. gain) by the value of corresponding eigenvalue.

The following C code is able to find the eigenvalues of any 2x2 matrix. For higher order matrices, it is possible to use several eigenvalue algorithms exist in the literature.

```
typedef struct Complex {
	double Re, Im;
} complex;


complex* Eigenvalues2x2(double *A)
{
	double tr, det, D;
	complex *eigs=malloc(2*sizeof(complex));

	tr=Trace(A,2);
	det=Determinant(A,2);
	D=pow(tr,2)-4*det;
```

```
      if(D>0)
      {
            eigs->Re=(tr+sqrt(D))/2;
            eigs->Im=0;
            (eigs+1)->Re=(tr-sqrt(D))/2;
            (eigs+1)->Im=0;
      }
      else
      {
            eigs->Re = tr/2;
            eigs->Im = sqrt(-1*D)/2;
            (eigs+1)->Re = tr/2;
            (eigs+1)->Im = -sqrt(-1*D)/2;
      }
      return eigs;
}
```

Eigenvalues also correspond to the system poles in s-plane! The written function can be used as follows:

```
// Eigenvalues
printf("eigs(T):\n");
cResult = Eigenvalues2x2((double*)T);
printf("%.3lf+j%.3lf\n",cResult->Re,cResult->Im);
printf("%.3lf+j%.3lf\n\n",(cResult+1)->Re,(cResult+1)->Im);
free(cResult);
```

Systems of Linear Equations:

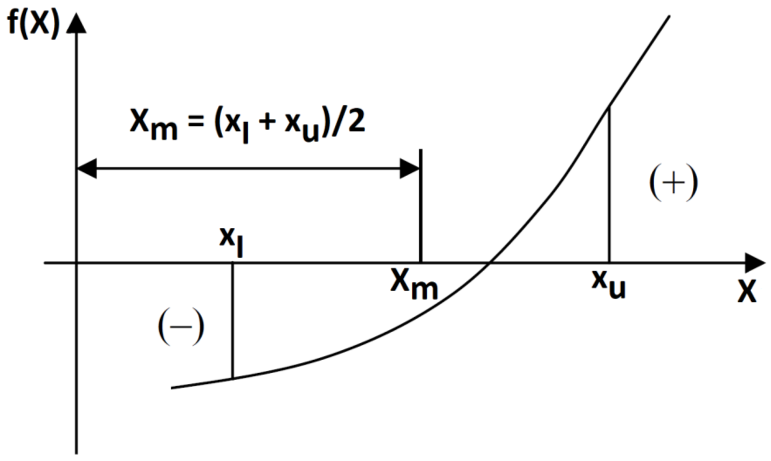
When you deal with two or more linear equations together, it is called as a system of linear equations. Linear algebra is a powerful tool to solve such equation systems with the help of matrix operations. Row or column operations can be used to solve the linear equations in following general form.

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2$$
$$\dots$$
$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m$$

In addition, it is possible to solve such equation systems with the help of matrix inversion if the resulting A matrix is invertible ($Ax = b \;\rightarrow\; x = A^{-1}b$).

- Solution to Linear/Nonlinear Equations with One Unknown:

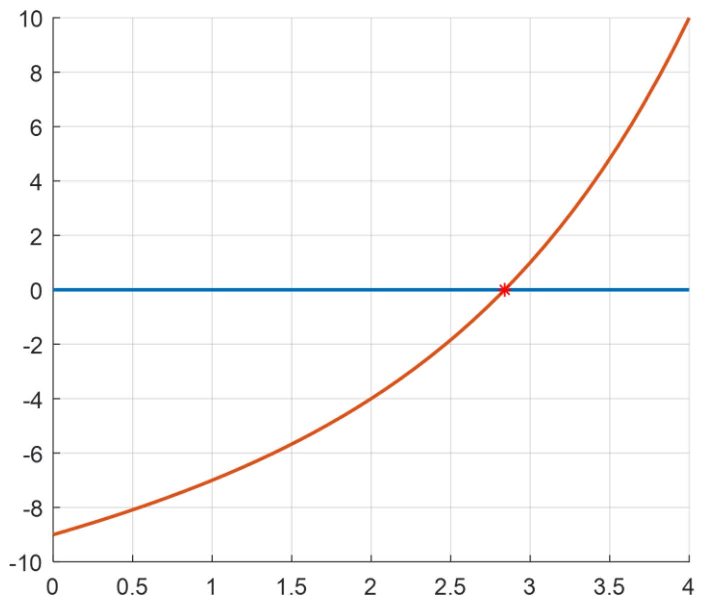
Midpoint Method:



If $f(x_l)f(x_u) < 0$ then the equation $f(x)$ has at least one root in (x_l, x_u). Therefore,

- If $f(x_l)f(x_m) < 0$ then $x_u = x_m$
- If $f(x_l)f(x_m) > 0$ then $x_l = x_m$
- If $f(x_l)f(x_m) = 0$ then x_m is a root of the $f(x)$

Thus, it is possible to find the root(s) of the equation using an iterative approach. Let us show this via an example. Assume that it is desired to find the root of the equation $f(x) = 2^x + x - 10$ in the interval of $x \in (0,4)$.



It is possible to find the root in the desired interval with the help of the following program written in C language:

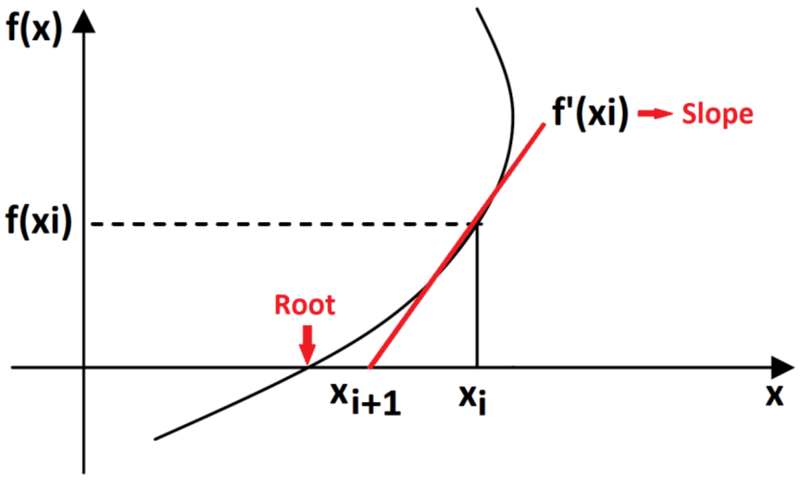
```c
#include <stdio.h>
#include <math.h>
#define EPS 0.001

double fx(double);
double Midpoint(double (*f)(double), double xL, double xU);

int main()
{
	printf("Solution is x=%.4lf\n",Midpoint(fx,0,4));
	return 0;
}

double Midpoint(double (*f)(double), double xL, double xU)
{
	double xm = (xU+xL)/2;

	if (f(xL)*f(xU) > 0)
	{
		printf("There is no root in (%.2lf,%.2lf)\n",xL,xU);
		return 0;
	}

	while(fabs(f(xm)) > EPS)
	{
		if(f(xL)*f(xm)<0)
			xU = xm;
		else if(f(xL)*f(xm)>0)
			xL = xm;
		else
			break;
		xm = (xL+xU)/2;
	}

	return xm;
}

double fx(double x)
{
	return pow(2,x)+x-10;
}
```

Fixed Point Iteration Method:

In this method, the $f(x)$ function is divided into two parts $f_1(x) = x = g(x)$. It is possible to separate the parts as $x_{n+1} = g(x_n)$ and then the solution is obtained via iterative approach.

Note that this method does not always converge to the solution, the initial point selection is important.

Newton-Raphson Method:



Newton-Raphson method can actually be obtained from the Taylor series expansion which is given as,

$$f(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i) + \cdots$$

If the higher order terms are ignored and by taking $f(x_{i+1}) = 0$, we can construct the iteration rule as follows:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$