Introduction to Programming Language (C)

(EEF 110E)

Dr. Emre DİNCEL

LECTURE NOTES

(WEEK 13)

Mathematics Applications in C

- Numerical Integration:

Trapezoidal Rule:

There exist several methods to calculate the integral of a function. One of the mostly used methods is called as trapezoidal rule and can be given as follows:

$$S = \int_{x_1}^{x_n} f(x)dx = \sum_{k=1}^{n-1} \int_{x_k}^{x_{k+1}} f(x)dx \qquad \text{(Dividing integrals into “n” parts)}$$

Thus, we have the following expression to calculate the given integral numerically;

$$S = \sum_{k=1}^{n-1} \frac{h}{2}[f(x_{k+1}) + f(x_k)]$$

where $h = x_{k+1} - x_k$.

Example:

Let us calculate the following integral with the help of trapezoidal rule:

$$I = \int_0^2 \sin(x) + x^2$$

Please note that the analytical solution is given as $I = \frac{x^3}{3} - \cos(x) \Big|_0^2 \cong 4.0828.$

It is now possible to write a code segment and calculate the integral of any given function as follows:

```c
// Numerical integration via trapezoidal rule
int main()
{
	int i;

	for(i=5;i<100;i=i+10)
		printf("Solution is x=%.4lf for n=%d\n",Integral(fx,0,2,i),i);
	return 0;
}
```

where the integral function is given as below:

```
double Integral(double (*f)(double), double a, double b, int n)
{
	int i;
	double xk, xk_1, step, sum=0.0;

	step=(b-a)/n;
	xk=a;
	for(i=0;i<n;i++)
	{
		xk_1=xk+step;
		sum+=(f(xk_1)+f(xk))*step/2;
		xk=xk_1;
	}
	return sum;
}
```

Simpson's Rule:

Another method to calculate an integral numerically is the Simpson's rule. The rule can be given as follows:

$$S = \int_{x_1}^{x_n} f(x)dx = \sum_{k=1}^{\frac{n-1}{2}} \int_{x_{2k-1}}^{x_{2k+1}} f(x)dx$$

$$S = \sum_{k=1}^{\frac{n-1}{2}} \frac{h}{3}[f(x_{2k-1}) + 4f(x_{2k}) + f(x_{2k+1})]$$

where $h = x_{2k+1} - x_{2k} = x_{2k} - x_{2k-1},\ n > 3,\ n$ is odd number.

Boole's Rule:

$$S = \int_{x_1}^{x_n} f(x)dx = \sum_{k=1}^{\frac{n-1}{4}} \int_{x_{4k-2}}^{x_{4k+2}} f(x)dx$$

$$S = \sum_{k=1}^{\frac{n-1}{4}} \frac{h}{45}[14f(x_{4k-2}) + 64f(x_{4k-1}) + 24f(x_{4k}) + 64f(x_{4k+1}) + 14f(x_{4k+2})]$$

where $h = x_{4k+2} - x_{4k+1} = \cdots = x_{4k-1} - x_{4k-2},\ \frac{n-1}{4}$ are positive integers.

- Numerical Derivative:

Numerical differentiation describes the algorithms for estimating the derivative of a function using values of the function and perhaps other knowledge about the function. There are several methods available to calculate the derivative of a function at a given point, numerically.

Forward Difference Method:

$$f'(x_i) = \frac{f(x_i + h) - f(x_i)}{h}$$

Backward Difference Method:

$$f'(x_i) = \frac{f(x_i) - f(x_i - h)}{h}$$

Central Difference Method:

$$f'(x_i) = \frac{f(x_i + h) - f(x_i - h)}{2h}$$

Example: Consider the function

$$f(x) = \sin(x) + x^2$$

Using forward and backward difference formulas, calculate the derivative $f'(1)$ for $h = 0.1$, 0.01 and 0.001, respectively.

Note that the analytical solution is given as $F = 2x + \cos(x) \bigg|_{x\,=\,1.0} = 2.5403$.

```
int main()
{
	int i;
	double h=0.1;

	for(i=0;i<3;i++)
	{
		printf("f'(1)=%.4lf for h=%lf (Forward)\n",DForward(fx,1,h),h);
		printf("f'(1)=%.4lf for h=%lf (Backward)\n\n",DBackward(fx,1,h),h);
		h/=10;
	}
	return 0;
}
```

where the corresponding functions are written as follows:

```
double DForward(double (*f)(double), double x, double h)
{
	return (f(x+h)-f(x))/h;
}

double DBackward(double (*f)(double), double x, double h)
{
	return (f(x)-f(x-h))/h;
}
```

- Curve Fitting:

In many engineering applications, it is required to find a suitable function/polynomial which can be used to represent a data set well enough. Thus, it may be possible to find the other values, which are not indicated in the data set, using this function/polynomial.

Linear Regression:

Linear regression is one of the most used types of predictive analysis. Here, a linear equation is aimed to be fit to the given data set by minimizing the quadratic error.

The linear equation including the error term is $y = a_0 + a_1 x + e$; therefore, the error term is expressed as follows.

$$e = y - a_0 - a_1 x$$

The quadratic error sum,

$$E = \sum_{i=1}^{n} (y_i - a_0 - a_1 x_i)^2$$

In order to find the a_0 and a_1 which minimizes the error, partial derivatives should be equalized to zero as given below.

$\frac{\partial}{\partial a_0} E = -2 \sum_{i=1}^{n} (y_i - a_0 - a_1 x_i) = 0$ and $\frac{\partial}{\partial a_1} E = -2 \sum_{i=1}^{n} (y_i - a_0 - a_1 x_i)\, x_i = 0$

If the required simplifications are done, we have,

$$a_0 n + \sum_{i=1}^{n} a_1 x_i = \sum_{i=1}^{n} y_i$$

$$\sum_{i=1}^{n} a_0 x_i + \sum_{i=1}^{n} a_1 x_i^2 = \sum_{i=1}^{n} x_i y_i$$

If these equations are solved together, it is possible to obtain the corresponding coefficients. For this special case (linear approximation), the solution to the given equation set is given as follows:

$$a_1 = \frac{n\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}$$

$$a_0 = \frac{\sum_{i=1}^{n} y_i}{n} - a_1 \frac{\sum_{i=1}^{n} x_i}{n}$$

Example:

Let us find a linear equation which represents the following data set.

Time	0	3	5	9	11	14	18	20	[s]
Position	0	10	24	33	36	40	44	50	[m]

First of all, we require some new functions to be included in our previously written “matrix library” such as vector power and vector sum. The written functions can be found below:

```c
double VectorSum(double *U, int n)
{
      int i;
      double sum=0.0;

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

double* VectorPow(double *U, int n, int p)
{
      int i;
      double *W=malloc(n*sizeof(double));

      for(i=0;i<n;i++)
            *(W+i)=pow(*(U+i),p);
      return W;
}
```

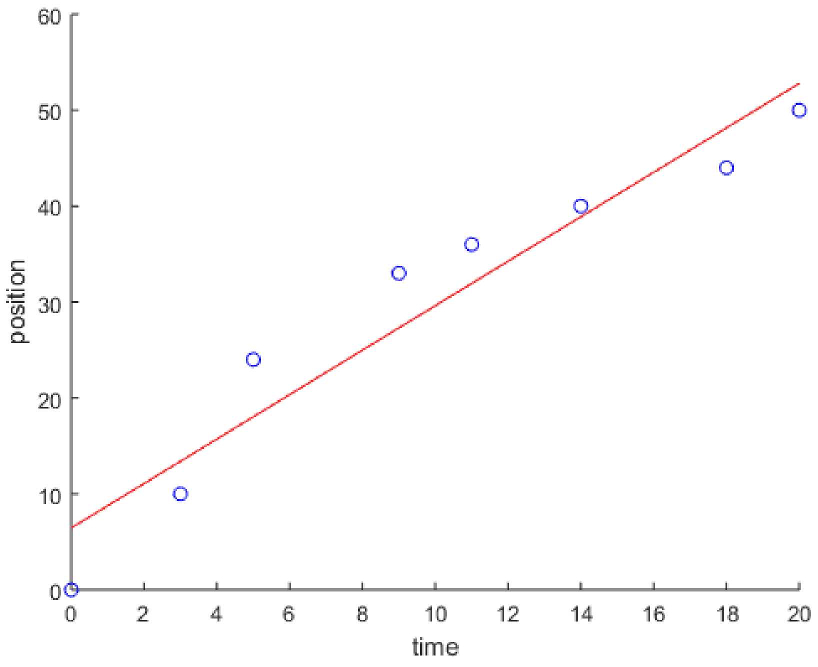
It is now possible to fit a linear equation to the given data set with the help of the following code in C language.

```c
#include <stdio.h>
#include <stdlib.h>
#include "matrix.h"
#define DATA_LENGTH 8

int main()
{
	int i, j;
	double time[DATA_LENGTH] = {0,  3,  5,  9, 11, 14, 18, 20};
	double pos[DATA_LENGTH] =  {0, 10, 24, 33, 36, 40, 44, 50};
	double *A=malloc(4*sizeof(double));
	double *b=malloc(2*sizeof(double));
	double *x;

	for(i=0;i<2;i++)
	   for(j=0;j<2;j++)
	      *(A+2*i+j)=VectorSum(VectorPow(time,DATA_LENGTH,i+j),DATA_LENGTH);

	for(j=0;j<2;j++)
	   *(b+j)=DotProduct(VectorPow(time,DATA_LENGTH,j),pos,DATA_LENGTH);

	x=Multiply(Inverse(A,2),b,2,2,1); // x=A^-1*b
	free(A);
	free(b);

	printf("Best fit: x=%.3lf*t+%.3lf\n",*(x+1),*x);
	free(x);
	return 0;
}
```

If the above code is compiled and run, we get the following results:

$$x = 2.317t + 6.451$$

We can plot the data points and the line described by the equation given below, with the help of any 3rd party program as follows:



Let us also find the position at the time $t = 15s$ (which is not given in the data set).

$$f(15) = 2.317 \text{ x } 15 + 6.451 = 41.206$$

Polynomial Fitting:

The same approach is used in polynomial fitting; however, this time the error term is given as follows:

$e = y - a_0 - a_1x - a_2x^2 - \cdots - a_mx^m$

Thus, the quadratic error sum is

$$E = \sum_{i=1}^{n}(y_i - a_0 - a_1x_i - a_2x_i^2 - \cdots - a_mx_i^m)^2$$

If the partial derivatives are calculated,

$$a_0n + a_1\sum_{i=1}^{n}x_i + a_2\sum_{i=1}^{n}x_i^2 + \cdots + a_m\sum_{i=1}^{n}x_i^m = \sum_{i=1}^{n}y_i$$

$$a_0\sum_{i=1}^{n}x_i + a_1\sum_{i=1}^{n}x_i^2 + \cdots + a_m\sum_{i=1}^{n}x_i^{m+1} = \sum_{i=1}^{n}x_iy_i$$

…

$$a_0\sum_{i=1}^{n}x_i^m + a_1\sum_{i=1}^{n}x_i^{m+1} + \cdots + a_m\sum_{i=1}^{n}x_i^{2m} = \sum_{i=1}^{n}x_i^my_i$$

and the resulting equations are all solved together, the polynomial coefficients can be found.

Example:

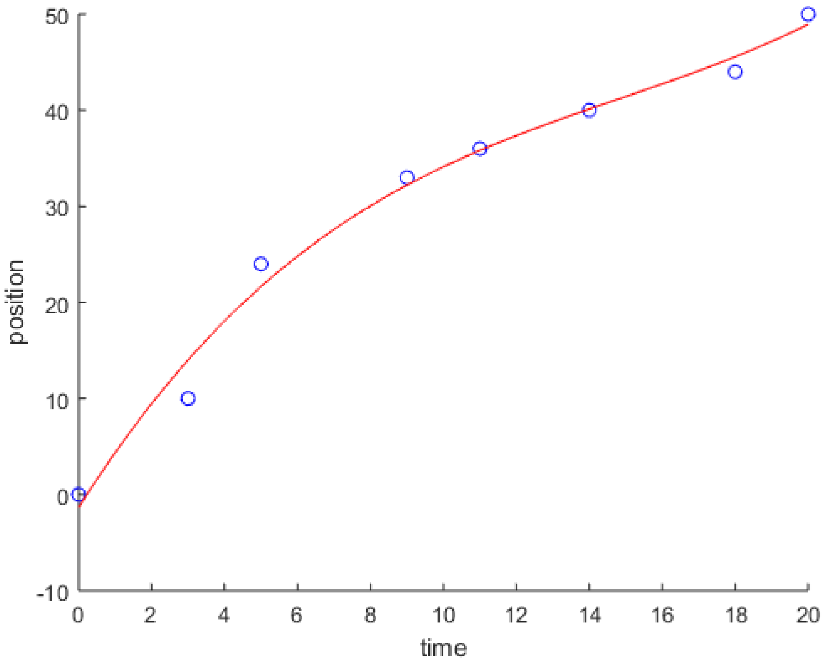
Consider the previous example. This time it is desired to fit a polynomial of 3rd order to the given data set. Let us generalize the given code for nth order polynomial fit.

```
#include <stdio.h>
#include <stdlib.h>
#include "matrix.h"
#define DATA_LENGTH 8
#define POLY_ORDER 3

int main()
{
      int i, j;
      double time[DATA_LENGTH] = {0,  3,  5,  9, 11, 14, 18, 20};
      double pos[DATA_LENGTH] =  {0, 10, 24, 33, 36, 40, 44, 50};
      double *A=malloc((POLY_ORDER+1)*(POLY_ORDER+1)*sizeof(double));
      double *b=malloc((POLY_ORDER+1)*sizeof(double));
      double *x, aVal;

      for(i=0;i<(POLY_ORDER+1);i++)
         for(j=0;j<(POLY_ORDER+1);j++)
            *(A+(POLY_ORDER+1)*i+j)=VectorSum(VectorPow(time, ...
DATA_LENGTH,i+j),DATA_LENGTH);

      for(j=0;j<(POLY_ORDER+1);j++)
         *(b+j)=DotProduct(VectorPow(time,DATA_LENGTH,j),pos,DATA_LENGTH);

      // x=A^-1*b
      x=Multiply(Inverse(A,POLY_ORDER+1),b,(POLY_ORDER+1),(POLY_ORDER+1),1);
      free(A);
      free(b);

      printf("Best fit: x=");
      for(i=0;i<(POLY_ORDER+1);i++)
      {
            aVal=*(x+POLY_ORDER-i);
            if(aVal>0) printf("+");
            printf("%.4lf*t^%d",aVal,POLY_ORDER-i);
      }
      free(x);
      return 0;
}
```

We have the result of

$$x = 0.0071t^3 - 0.3175t^2 + 6.006t - 1.3346$$

It is seen from the below figure that the 3rd order approximation is clearly better than the first order (linear) approximation!


position
time