RAJSHAHI UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Lab report: 04

Date of Experiment: 28.02.18
Date of Submission: 07.03.18

Submitted to:
Shyla Afroge
Assistant Professor,
Department of Computer Science and Engineering
Rajshahi University of Engineering and Technology

Submitted by:
Riyad Morshed Shoeb
Roll No: 1603013
Section: A
Department of Computer Science and Engineering
Rajshahi University of Engineering and Technology

Name of the experiment: Implementation of Newton's forward and backward difference interpolation method.

Theory:

Given the set of $(n+1)$ *valus, viz,* $(x_0, y_0), (x_1, y_1), (x_2, y_2) \dots\dots\dots, (x_n, y_n)$, *of* x *and* y, *it is required to find* $y_n(x)$, *a polynomial of the* $n-th$ *degree such that* y *and* $y_n(x)$ *agree at the tabulated points. Let the values of* x *be equidistant, i.e. let,*

$$x_i = x_0 + ih, \qquad i = 0,1,2, \dots\dots\dots\dots\dots\dots, n$$

Since $y_n(x)$ is a polynomial of the nth degree, it may be written as,

$$y_n(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + a_3(x - x_0)(x - x_1)(x - x_2) + \cdots \dots\dots\dots\dots\dots$$
$$+a_n(x - x_0)(x - x_1)(x - x_2) \dots\dots\dots\dots(x - x_{n-1})$$

Imposing now the condition that y and $y_n(x)$ should agree at the set of tabulated points, we obtain,

$$a_0 = y_0;\ a_1 = \frac{y_1 - y_0}{x_1 - x_0} = \frac{\Delta y_0}{h};\ \ a_2 = \frac{\Delta^2 y_0}{h^2 2!};\ a_3 = \frac{\Delta^3 y_0}{h^3 3!}; \dots\dots; a_n = \frac{\Delta^n y_0}{h^n n!};$$

Setting $x = x_0 + ph$ *and substituting for* $a_0, a_1, \dots\dots\dots, a_n$ *gives,*

$$y_n(x) = y_0 + p\Delta y_0 + \frac{p(p-1)}{2!}\Delta^2 y_0 + \frac{p(p-1)(p-2)}{3!}\Delta^3 y_0 + \cdots \dots + \frac{p(p-1)(p-2)\dots(p-n+1)}{n!}\Delta^n y_0$$

Which is Newton's Forward Difference Interpolation formula and is useful for interpolation near the beginning of a set of tabular values.

Instead of assuming $y_n(x)$, if we assume it in the form,

$$y_n(x) = a_0 + a_1(x - x_n) + a_2(x - x_n)(x - x_{n-1}) + a_3(x - x_n)(x - x_{n-1})(x - x_{n-2}) + \cdots \dots\dots\dots\dots$$
$$+a_n(x - x_n)(x - x_{n-1})(x - x_{n-2}) \dots\dots\dots\dots(x - x_1)$$

And then impose the condition that y and $y_n(x)$ should agree at the tabulated points $x_n, x_{n-1}, \dots\dots, x_2$, x_1, x_0, we obtain,

$$y_n(x) = y_n + p\nabla y_n + \frac{p(p+1)}{2!}\nabla^2 y_n + \cdots \dots\dots + \frac{p(p+1)\dots(p+n-1)}{n!}\nabla^n y_n$$

$where, p = \frac{x - x_n}{h}$.

This is Newton's Backward Difference Interpolation formula and it uses tabular values to the left of y_n. This formula is, therefore, useful for interpolation near the end of the tabular values.

Code:

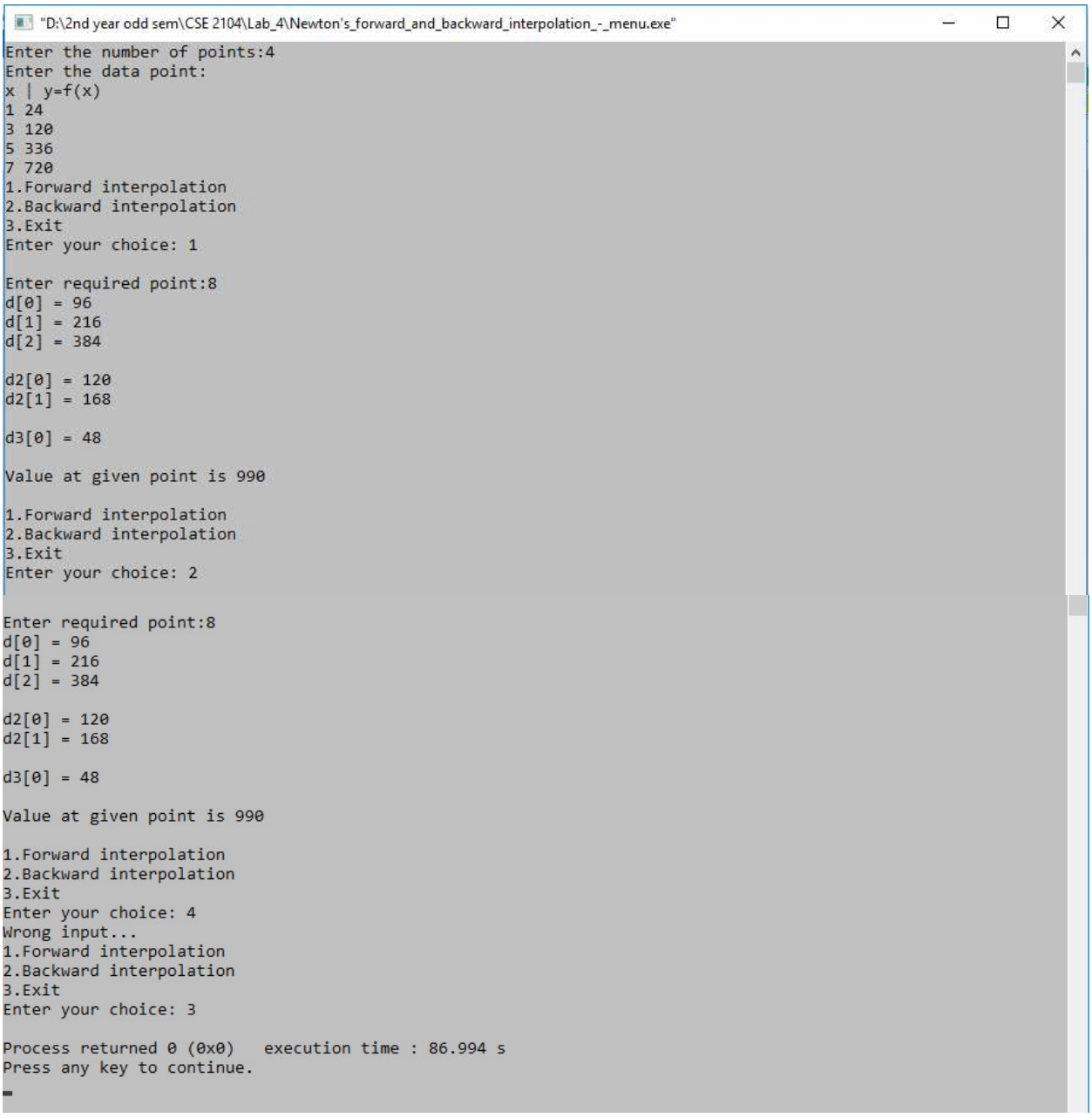
```
#include<iostream>
#include<cstdio>
#include<cstdlib>
#include<cmath>
using namespace std;

int main(void)
{
    double a,a1, x[100],y[100],d[100],d2[100],d3[100],d4[100],xa,p,h;
    int i,j,k,n,m,c;
    printf("Enter the number of points:");
    cin>>n;
    cout<<"Enter the data point:"<<endl;
    cout<<"x | y=f(x)"<<endl;
    for(i=0;i<n;i++)
        cin>>x[i]>>y[i];
    while(1)
    {
        printf("1.Forward interpolation \n2.Backward interpolation \n3.Exit
\n");
        printf("Enter your choice: ");
        cin>>c;
        switch(c)
        {
        case 1:
```

```
            {
                cout<<"\nEnter required point:";
                cin>>xa;
                h=(x[1]-x[0]);
                p=(xa-x[0])/h;
                for(i=0; i<n-1; i++)
                {
                    d[i]=y[i+1]-y[i];
                    printf("d[%d] = %d\n",i,(int)d[i]);
                }
                printf("\n");
                for(i=0; i<n-2; i++)
                {
                    d2[i]=d[i+1]-d[i];
                    printf("d2[%d] = %d\n",i,(int)d2[i]);
                }
                printf("\n");
                for(i=0; i<n-3; i++)
                {
                    d3[i]=d2[i+1]-d2[i];
                    printf("d3[%d] = %d\n",i,(int)d3[i]);
                }
                printf("\n");
                for(i=0; i<n-4; i++)
                {
                    d4[i]=d3[i+1]-d3[i];
                    printf("d4[%d] = %d\n",i,(int)d4[i]);
                }
                a=            y[0]+p*d[0]+((p*(p-1))*d2[0])/2+((p*(p-1)*(p-
2))*d3[0])/6+((p*(p-1)*(p-2)*(p-3))*d4[0])/24;
                printf("Value at given point is %d\n\n",(int)a);
            }
            break;
        case 2:
            {
                cout<<"\nEnter required point:";
                cin>>xa;
                h=(x[1]-x[0]);
                p=(xa-x[n-1])/h;
                for(i=0; i<n-1; i++)
                {
                    d[i]=y[i+1]-y[i];
                    printf("d[%d] = %d\n",i,(int)d[i]);
                }
                printf("\n");
                for(i=0; i<n-2; i++)
                {
                    d2[i]=d[i+1]-d[i];
                    printf("d2[%d] = %d\n",i,(int)d2[i]);
                }
                printf("\n");
                for(i=0; i<n-3; i++)
                {
                    d3[i]=d2[i+1]-d2[i];
                    printf("d3[%d] = %d\n",i,(int)d3[i]);
                }
                printf("\n");
                for(i=0; i<n-4; i++)
                {
                    d4[i]=d3[i+1]-d3[i];
                    printf("d4[%d] = %d\n",i,(int)d4[i]);
```

```
                }
                a1=y[n-1]+p*d[n-2]+((p*(p+1))*d2[n-
3])/2+((p*(p+1)*(p+2))*d3[n-4])/6+((p*(p+1)*(p+2)*(p+3))*d4[n-5])/24;
                printf("Value at given point is %d\n\n",(int)a);
            }
            break;
        case 3:
            return 0;
        default:
            printf("Wrong input...\n");
        }
    }
}
```

Output:



Discussion:

Newton's formula is of interest because it is the straightforward and natural differences-version of Taylor's polynomial. Taylor's polynomial tells where a function will go, based on its y value, and its derivatives (its rate of change, and the rate of change of its rate of change, etc.) at one particular x value. Newton's formula is Taylor's polynomial based on finite differences instead of instantaneous rates of change