# RAJSHAHI UNIVERSITY OF ENGINEERING AND TECHNOLOGY


**Lab report: 03**

**Date of Experiment: 20.02.18**
**Date of Submission: 28.02.18**

**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

**Experiment no: 01**

**Name of the Experiment: Implementation of Newton-Raphson Method**

**Theory:**

*Let, $x_0$ be an approximate root of $f(x) = 0$ and $x_1 = x_0 + h$ be the correct root so that, $f(x_1) = 0$.*
*Expanding $f(x_0 + h)$* by Taylor's series, we obtain,

$$f(x_0) + hf'(x_0) + \frac{h^2}{2!}f"(x_0) + \cdots \ldots \ldots \ldots \ldots = 0$$

Neglecting the second and higher order derivatives, we have,

$$f(x_0) + hf'(x_0) = 0$$

Which gives,

$$h = -\frac{f(x_0)}{f'(x_0)}$$

Hence,

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

For successive iterations,

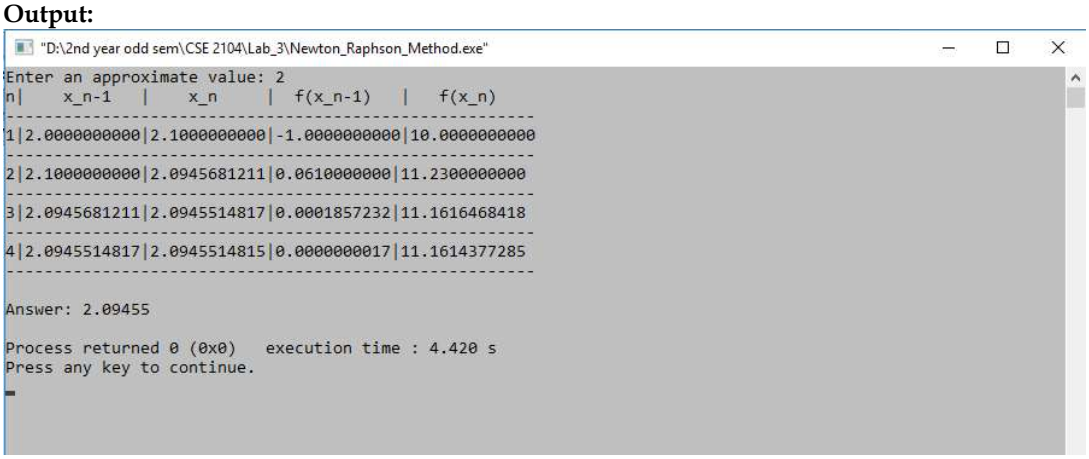
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

**Code:**

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

double func_val(double x)
{
    return ((x*x*x)-(2*x)-5);
}
double diff_func_val(double x)
{
    return ((3*x*x)-2);
}
int main(void)
{
    double x0,x;
    int n=1;
    cout<<"Enter an approximate value: ";
    cin>>x0;
    printf("n|    x_n-1   |    x_n     |  f(x_n-1)   |   f(x_n)\n");
    printf("-----------------------------------------------------\n");
    while(1)
    {
        x=x0-(func_val(x0)/diff_func_val(x0));
        printf("%d|%3.10f|%3.10f|%3.10f|%3.10f\n",n,x0,x,func_val(x0),
diff_func_val(x0));
        printf("-----------------------------------------------------
\n");
        n++;
        if(abs(x0-x)<=0.000001)
            break;
        else
            x0=x;
    }
    printf("\nAnswer: ");
```

```
    cout<<x<<endl;
}
```

**Output:**




**Discussion:**

Newton's method is an extremely powerful technique—in general the convergence is quadratic: as the method converges on the root, the difference between the root and the approximation is squared (the number of accurate digits roughly doubles) at each step. However, there are some difficulties with the method. Newton's method requires that the derivative be calculated directly. An analytical expression for the derivative may not be easily obtainable and could be expensive to evaluate. In these situations, it may be appropriate to approximate the derivative by using the slope of a line through two nearby points on the function. Using this approximation would result in something like the secant method whose convergence is slower than that of Newton's method. If a stationary point of the function is encountered, the derivative is zero and the method will terminate due to division by zero. It is important to review the proof of quadratic convergence of Newton's Method before implementing it. Specifically, one should review the assumptions made in the proof. For situations where the method fails to converge, it is because the assumptions made in this proof are not met.

**Experiment no: 02**

**Name of the Experiment: Implementation of Ramanujan's Method**

**Theory:**

This method is used to determine the smallest root of the equation,

$$f(x) = 0$$

Where, f(x) is of the form

$$f(x) = 1 - (a_1x + a_2x^2 + a_3x^3 + \cdots \ldots \ldots \ldots \ldots$$

Let,

$[1 - (a_1x + a_2x^2 + a_3x^3 + \cdots \ldots \ldots \ldots \ldots)]^{-1} = b_1 + b_2x + b_3x^2 + \cdots$

$\Rightarrow 1 + (a_1x + a_2x^2 + a_3x^3 + \cdots) + (a_1x + a_2x^2 + a_3x^3 + \cdots)^2 + \cdots = b_1 + b_2x + b_3x^2 + \cdots$

Equating the co-efficient on both sides, we obtain,

$b_1 = 1$

$b_2 = a_1b_1$

$b_3 = a_1b_2 + a_2b_1$

.. .. .. .. ..

.. .. .. .. ..

$b_n = a_1b_{n-1} + a_2b_{n-2} + \cdots \ldots \ldots \ldots + a_{n-1}b_1$

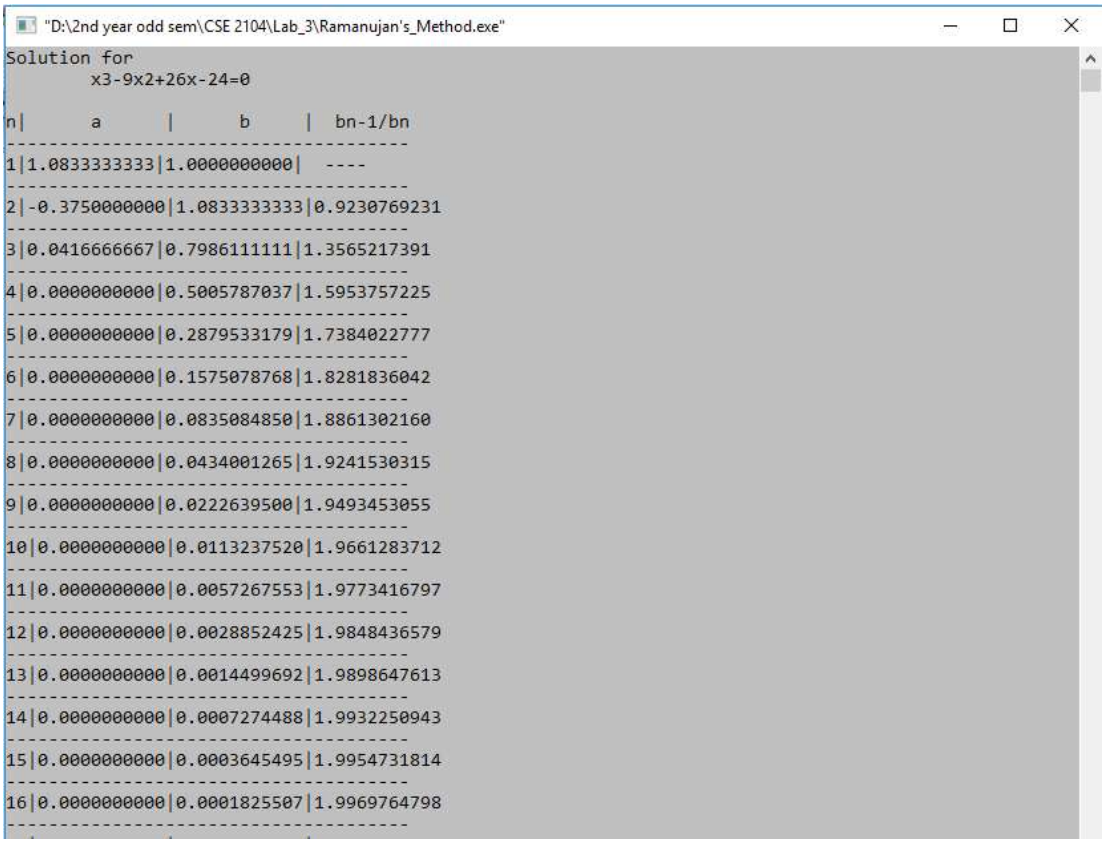
The ratios $\frac{b_{i-1}}{b_i}$, called the convergent, approach, In the limit, the smallest root of f(x)=0.

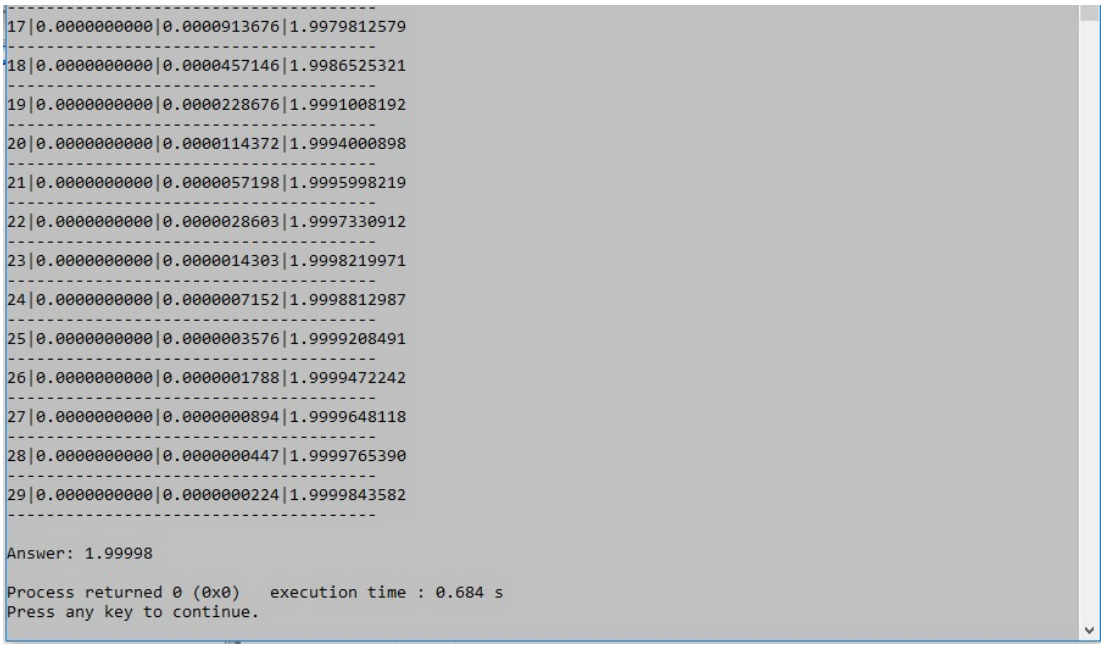
**Code:**

```
#include<iostream>
#include<cstdio>
#include<cstdlib>
#include<cmath>
#include<vector>

using namespace std;



int main(void)
{
    double a[100]={0};
    double temp;
    vector<double>b;
    int i,n;
    int iter=1;

    a[1]=(13/12.0);
    a[2]=(-3/8.0);
    a[3]=(1/24.0);

    b.push_back(0);
    b.push_back(1);
    temp=a[1]*b[1];
    b.push_back(temp);

    printf("Solution for\n\tx3-9x2+26x-24=0\n\n");
    n=1;
    printf("n|      a      |      b     |  bn-1/bn\n");
    printf("---------------------------------------\n");
    printf("%d|%3.10f|%3.10f|  ----  \n",iter++,a[n],b[n]);
    printf("---------------------------------------\n");
    n++;
    printf("%d|%3.10f|%3.10f|%3.10f\n",iter++,a[n],b[n],b[n-1]/b[n]);
```

```
    printf("---------------------------------------\n");
    n++;


    while(1)
    {
        b.push_back(0);
        for(i=1;i<n;i++)
            b[n]+=(a[i]*b[n-i]);

        printf("%d|%3.10f|%3.10f|%3.10f\n",iter++,a[n],b[n],b[n-1]/b[n]);
        printf("---------------------------------------\n");

        if(abs((b[n-2]/b[n-1])-(b[n-1]/b[n]))<=0.00001)
            break;
        else
            n++;
    }


    printf("\nAnswer: ");
    cout<<(b[n-1]/b[n])<<endl;

    return 0;
}
```

**Output:**






**Discussion:**
This method is efficient in the sense that it does not need any initial value. But the problem is, it might iterate more times than the other methods. Also, this method does not work on every function.

**Discussion:**

The method of false position is more efficient than bisection method, but the calculation for each iteration is pretty lengthy. The Iteration method is even more efficient, but not every time. The initial guess depends a lot on this. As we have to find derivative, the calculation process is bit more complex.