实验报告2

姓名：吴毅龙 学号：PB19111749

1. 问题

- 分别编写**Newton迭代法和对分法**的程序，并利用它们去计算如下非线性方程的根
  $f(x) = 2^{-x} + e^x + 2cosx - 6$

  其中，对分法的初始区间$[0, 10]$，Newton迭代法的初始点分别取0和10.
- 取误差限$\varepsilon = 10^{-8}$，即当$|f(x_k)| < \varepsilon$时，停止迭代
- 列表给出每步的迭代结果或者前后5步的迭代结果（如果迭代步数超过10步时）以及迭代总步数；比较并分析两种方法的优劣。

2. 计算过程及计算结果

2.1 算法与程序实现

Newton迭代法的基本流程为：确定迭代、变量建立迭代关系式、对迭代过程进行控制。

基于**二分法求方程的根**的原理是介值定理，即通过给定两个初始值a和b，使f(a)f(b)<0。如果f(x)是连续的，则必然在(a,b)内存在一个值c，使f(c)=0。

首先计算c=(a+b)/2;如果f(c)=0，则直接返回c即可，否则，进行下面的判断：如果f(a)f(c)>0，则令a=c，否则令b=c。

使用C语言对上述算法进行程序实现

```
#include<stdio.h>
#include<math.h>
#define e 2.718281828
#define epslion 1e-8

double PrimitiveFunction(double x)
{
    return pow(2, -x) + pow(e, x) + 2 * cos(x) - 6;
}

double DerivativeFunction(double x)
{
    return -1 * pow(2, -x) * log(2) + pow(e, x) - 2 * sin(x);
}

void NewtonMethod(double x)
{
    int k = 0;
    while (fabs(PrimitiveFunction(x)) >= epslion)
    {
        printf("k=%d    x%d=%0.10f      f(x)=%0.10f\n", k, k, x,
PrimitiveFunction(x));
        x = x - PrimitiveFunction(x) / DerivativeFunction(x);
        k++;
    }
```

```
    printf("k=%d    x%d=%0.10f      f(x)=%0.10f\n", k, k, x,
PrimitiveFunction(x));
}

double Half(double x, double y)
{
    x = x * 10000000;
    y = y * 10000000;
    return (x + y) / 20000000;
}

void BisectionMethod(double low, double high)
{
    int k = 0;
    double root = (low + high) / 2;
    double left = PrimitiveFunction(low);
    double right = PrimitiveFunction(high);
    double middle = PrimitiveFunction(root);
    while (fabs(middle) >= epslion)
    {
        printf("k=%d    [%0.10f,%0.10f]     root=%0.10f     f(x)=%0.10f\n", k,
low, high, root, middle);
        if ((middle <epslion && right > epslion)||(middle > epslion && right <
epslion))
        {
            low = root;
            root = Half(low, high);
            left = PrimitiveFunction(low);
            middle = PrimitiveFunction(root);
            k++;
        }
        else
        {
            if ((middle <epslion && left > epslion) || (middle > epslion&& left
< epslion))
            {
                high = root;
                root = Half(low, high);
                right = PrimitiveFunction(high);
                middle = PrimitiveFunction(root);
                k++;
            }
            else
            {
                printf("ERROR!\n");
                break;
            }
        }
    }
    printf("k=%d    [%0.10f,%0.10f]     root=%0.10f     f(x)=%0.10f\n", k, low,
high, root, middle);
}

int main()
{
    double x1, x2;
    double low, high;
    scanf("%lf %lf", &x1, &x2);
```

```
    scanf("%lf %lf", &low, &high);

    printf("Newton Method for x1\n");
    NewtonMethod(x1);
    printf("\n");

    printf("Newton Method for x2\n");
    NewtonMethod(x2);
    printf("\n");


    printf("Bisection Method\n");
    BisectionMethod(low, high);
    return 0;
}
```

2.2 计算结果

2.2.1 Newton迭代法

- 初值$x_0=0$的Newton迭代结果

迭代步数k	x_k	$f(x_k)$
$k=0$	0.0000000000	-2.0000000000
$k=1$	6.5177827065	673.0315732306
$k=2$	5.5230611395	245.8716230513
$k=3$	4.5464628865	88.0107359569
$k=4$	3.6319814821	30.1039795420
$k=5$	2.8535707847	9.5703611468
$k=6$	2.2800010916	2.6801276990
$k=7$	1.9497821270	0.5460548492
$k=8$	1.8403446404	0.0453715661
$k=9$	1.8294833422	0.0004091171
$k=10$	1.8293836107	0.0000000342
$k=11$	1.8293836024	0.0000000000

- 初值$x_0=10$的Newton迭代结果

迭代步数k	x_k	$f(x_k)$
$k=0$	10.0000000000	22018.7885911142
$k=1$	9.0003978895	8098.4880529237
$k=2$	8.0012609805	2978.4296918299
$k=3$	7.0027054717	1095.1161260613
$k=4$	6.0055867983	401.6279733817
$k=5$	5.0169063711	145.5741686140
$k=6$	4.0643840546	51.0816097141
$k=7$	3.2099219086	16.8898913273
$k=8$	2.5299415504	5.0885160278
$k=9$	2.0790045507	1.2599570918
$k=10$	1.8719524207	0.1809311653
$k=11$	1.8308468174	0.0060087418
$k=12$	1.8293853949	0.0000073519
$k=13$	1.8293836024	0.0000000000

2.2.2 对分法

迭代步数k	区间	区间中点	$f(x_k)$
$k=0$	[0.0000000000,10.0000000000]	5.0000000000	143.0117333482
$k=1$	[0.0000000000,5.0000000000]	2.5000000000	4.7569834198
$k=2$	[0.0000000000,2.5000000000]	1.2500000000	-1.4585641109
$k=3$	[1.2500000000,2.5000000000]	1.8750000000	0.1943790391
$k=4$	[1.2500000000,1.8750000000]	1.5625000000	-0.8741104693
$k=5$	[1.5625000000,1.8750000000]	1.7187500000	-0.4134649413
....	....	....	....
$k=25$	[1.8293833733,1.8293836713]	1.8293835223	-0.0000003287
$k=26$	[1.8293835223,1.8293836713]	1.8293835968	-0.0000000231
$k=27$	[1.8293835968,1.8293836713]	1.8293836340	0.0000001297
$k=28$	[1.8293835968,1.8293836340]	1.8293836154	0.0000000533
$k=29$	[1.8293835968,1.8293836154]	1.8293836061	0.0000000151
$k=30$	[1.8293835968,1.8293836061]	1.8293836014	-0.0000000040

3. 计算结果分析

从上述实验结果可以看出，对于不同的起始数据，牛顿迭代法的收敛速度是不一样的。越靠近根的起始数据的收敛速度越快。横向对比两个求根算法，可以看出牛顿迭代法的收敛速度比对分法快很多，牛顿迭代法只需要十几次就可以得出目标精度的结果，而使用对分法却需要进行多大三十次的计算。

总体来看，虽然少数情况下牛顿迭代法不能收敛，但是大多数情况下它效果都非常好。对分法固定每次缩短一半的区间，而牛顿迭代法的迭代效率往往更高，一般情况下使用牛顿迭代法可以获得更快的收敛速度。

4. 实验总结

通过这个实验，我再一次体会到算法从理论落地到实践的过程，这其中需要解决许多在理论推演过程中无法想象到的问题。比如在实验过程中，我遇到最大的问题就是double数据类型精度的损失问题。在对分法求根的程序编写过程中，我最初使用的是，区间两端的函数值相乘是否小于0作为下一次计算区间选择的判据。但是在程序调试中发现，这样的判断依据会使程序陷入无限循环。究其原因就是double双精度数据在相乘的过程中精度发生了损失。因此我针对性的修改了判据，从而解决了程序的问题。