利用二分法和牛顿公式求解方程的根

姓名：岳宇轩 学号：19020011038 序号：14

实验目的：

1.实现牛顿公式，并分别找到收敛和发散的例子；

2.在相同精度和相同条件下，比较二分法与牛顿公式的迭代次数；

实验原理：

1. 二分法：

函数 f(x)在区间[a,b]内单调连续，且 f(a)f(b)<0，根据连续函数的性质可知方程在区间[a,b]内一定有唯一的实根。通过不断地把函数 f(x)的零点所在的区间一分为二，使区间的两个端点逐步逼近根，进而得到根的近似值。

2. 牛顿公式：

对于方程

$$f(x)=0$$

已知它的近似根 xk，则函数 f(x)在点 xk 附近可用一阶泰勒展开式

$$p(x)=f(xk)+f'(xk)(x-xk)$$

来近似，因此方程 f(x)=0 可近似的表示为 p(x)=0。p(x)=0 是一个线性方程，容易求根，因此取 p(x)=0 的根作为 f(x)=0 的新的近似根，记 xk+1，则有

$$x_{k+1} = x_k - \frac{f(x)}{f'(x)}$$

即为牛顿公式。

实验过程：

1. 实现牛顿公式，并分别找到收敛和发散的例子

选取函数：

$$f(x) = \frac{5}{6}x^4 - 4x^3 + \frac{23}{6}x^2 + 3x - \frac{17}{3}$$

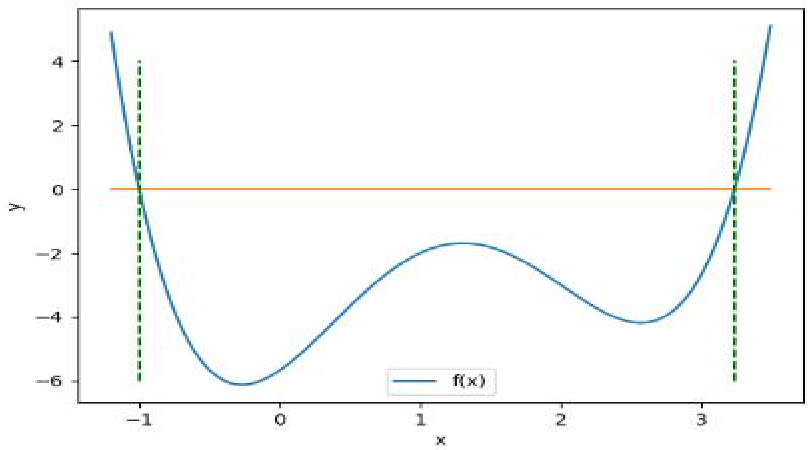
则有

$$f'(x) = \frac{10}{3}x^3 - 12x^2 + \frac{23}{3}x + 3$$

带入牛顿公式可得迭代式为:

$$x_{k+1} = \frac{5x_k^{\ 4} - 24x_k^{\ 3} + 23x_k^{\ 2} + 18x_k - 34}{20x_k^{\ 3} - 72x_k^{\ 2} + 46x_k + 18}$$

画出 f(x)的函数图：



求解 f(x)=0 的两个根，两个根分别是 x =-1 和 x = 3.2348365。为了验证牛顿公式的局部收敛性，我们分别选择 x0 =4 和 x0 =2，作为牛顿公式的起始点进行迭代求根（误差不超过 10-3)。

2.在相同精度和相同条件下，比较二分法与牛顿公式的迭代次数；

对于上述 f(x)=0 方程的根 3.2348365，取二分初始区间为[2, 4]，牛顿迭代的初始值 X0 设为 4，以保证公平性。精度设置：（误差不超过 10-3)。

代码实现：

全局变量设置：

```
#define MAX_TIMES 100 //牛顿迭代的最大迭代次数
#define e 0.001 //精度，即最小误差
#define ROOT 3.2348365 //根的真值
```

f(x）函数以及 f’(x)函数

```
/// f(x) = 5/6x^4 - 4x^3 + 23/6x^2 + 3x - 17/3
double f(double x) {
    return (5.0 / 6.0) * pow(x, 4) - 4 * pow(x, 3)
        + (23.0 / 6.0) * pow(x, 2) + 3 * pow(x, 1) - (17.0 / 3.0);
}

/// f’(x) = 10/3x^3 - 12x^2 + 23/3x + 3
double f_d(double x) {
    return (10.0 / 3.0) * pow(x, 3) - 12 * pow(x, 2)
        + (23.0 / 3) * pow(x, 1) + 3;
```

二分法：

```
void Binary() {
    int count = 0; //记录二分次数
    double left = 2.0; //二分初始区间左
```

```
    double right = 4.0; // 二分初始区间右
    double mid;

    printf("二分:\n");
    printf("  k\t xk\n");
    for (int i = 1; i <= MAX_TIMES && (right - left) >= e; i++) {
        mid = (left + right) / 2.0;
        if (f(left) * f(mid) < 0) {
            right = mid;
        }
        else {
            left = mid;
        }
        printf("%3d  %f\n", i, left); //最终取区间左值为结果
        count++;
    }
    printf("\n 二分法的迭代次数为:%d\n\n", count); //输出二分次数
}
```

牛顿公式：

```
void Newton() {

    int count = 0; //记录迭代次数
    double x0 = 4.0; //迭代初值
    double x1;

    printf("牛顿公式:\n");
    printf("  k\t xk\n");
    for (int i = 1; i <= MAX_TIMES; i++) {
        x1 = x0 - f(x0) / f_d(x0);  //牛顿公式
        printf("%3d  %f\n", i, x1);
        count++;
        if (fabs(x1 - x0) < e) { //收敛精度达到要求
            break;
        }
        x0 = x1;
    }
    printf("\n 牛顿公式的迭代次数为:%d\n\n", count);
}
```

主函数调用：

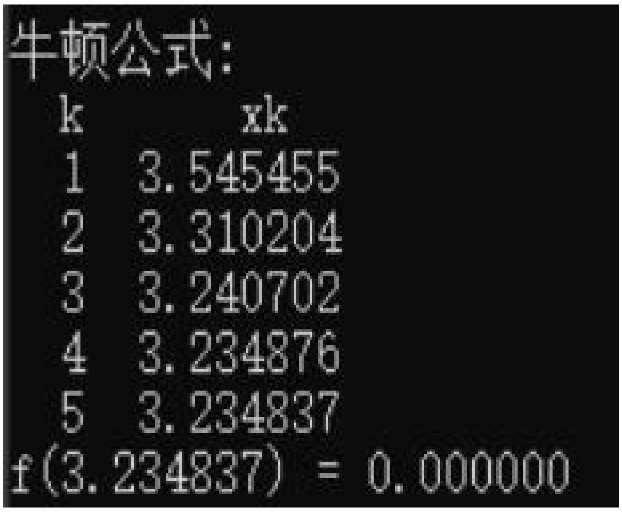
```
int main() {
```

```
    Binary();
    Newton();
    return 0;
}
```

实验结果

1. 实现牛顿公式，并分别找到收敛和发散的例子；

对于 x0=4, 结果如下：



```
牛顿公式:
 k        xk
 1   3.545455
 2   3.310204
 3   3.240702
 4   3.234876
 5   3.234837
f(3.234837) = 0.000000
```

可以看到，牛顿公式经过 5 次迭代之后达到精度 10^-3，输出 f(xk)结果为 0，表明实现了牛顿公式。

同时可以看到，对于初值 x0=4，牛顿公式是收敛的。

对于 x0=2，结果如下：

```
牛顿公式:
  k        xk
  1   1.000000
  2   2.000000
  3   1.000000
  4   2.000000
  5   1.000000
  6   2.000000
  7   1.000000
  8   2.000000
  9   1.000000
 10   2.000000
 11   1.000000
 12   2.000000
 13   1.000000
 14   2.000000
 15   1.000000
 16   2.000000
```

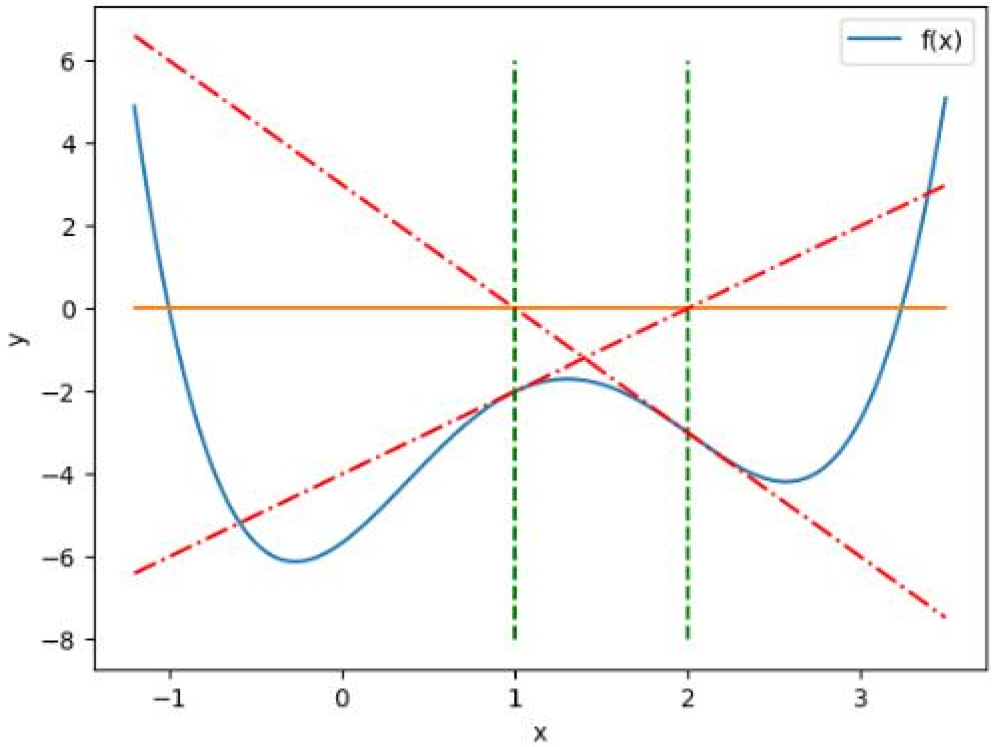
……………………

```
 86   2.000000
 87   1.000000
 88   2.000000
 89   1.000000
 90   2.000000
 91   1.000000
 92   2.000000
 93   1.000000
 94   2.000000
 95   1.000000
 96   2.000000
 97   1.000000
 98   2.000000
 99   1.000000
100   2.000000
f(2.000000) = -3.000000
```

从图中可以看到，经过最大的迭代次数 100 之后，该迭代仍然没有收敛，因此对于初值 x0=2 时，牛顿公式是发散的。

原因分析：根据牛顿公式迭代的原理，xk+1 实际上是取 f(x)在 xk

处的切线与 x 轴的交点，那么可以画出如下图像：



根据 f(x)和 f′(x)，我们可以求出 f(x)在 x=1 的切线方程为：

$$y = 2x - 4$$

f(x)在 x=2 处的切线方程为：

$$y = -3x + 3$$

切线在上图中用红线标出。

第一次迭代，对于初值 x0=2，它的下一次迭代值 x1 就是 f(x)在 x=2 处的切线与 x 轴交点处，将 y=0 带入 f(x)在 x=2 处的切线方程可求出：x=1，所以第一次迭代结果 x1=1.

第二次迭代，将 y=0 带入 f(x)在 x=1 处的切线方程，可求出 x=2，所以 x2=2.

第三次迭代，将重复第一次迭代的过程。

第四次迭代，将重复第二次迭代的过程。

……

可以发现，陷入了一个死循环之中，故对于初值 x0=2 的情况下，牛顿公式是不收敛的。

2.在相同精度和相同条件下，比较二分法与牛顿公式的迭代次数

实验结果如下：

```
二分:
  k        xk
  1   3.000000
  2   3.000000
  3   3.000000
  4   3.125000
  5   3.187500
  6   3.218750
  7   3.234375
  8   3.234375
  9   3.234375
 10   3.234375
 11   3.234375

二分法的迭代次数为:11

牛顿公式:
  k        xk
  1   3.545455
  2   3.310204
  3   3.240702
  4   3.234876
  5   3.234837

牛顿公式的迭代次数为:5
```

可以看到，对于相同的精度要求（10^-3），二分法需要 11 次求解，而牛顿公式只需 5 次迭代。通过对比可知，牛顿公式收

敛更快。

实验心得：

在这次的实验过程中，使用 C++编写了二分法和牛顿公式的相关代码，增强了编程能力，也进一步理解了这两种求近似根的方法。

牛顿公式给了我们一种可以机械化求近似根的方式，不需要我们花费更多时间寻找适合的迭代函数 p(，并且牛顿公式比二分法有更快的收敛速度，在精度一定的情况下，能更快地求解出近似根，但是牛顿公式是局部收敛的，也就是说对于某些函数 f(x)与特定的 xo 牛顿公式不一定收敛于根 x*，所以在实际使用中可以用其他方法找出处在收敛区域内的 xo，再使用牛顿公式求解近似根。