利用二分法和牛顿公式求解方程的根

第七小组：刘阳　任浩辰
科目：数值分析

日期：2020 年 5 月 24 日

1 实验目的

通过编程实践，熟练掌握牛顿公式求解方程的根，验证牛顿公式的局部收敛性，比较二分法与牛顿公式的收敛速度，并验证求解结果的正确性。

2 实验步骤

1. 验证牛顿公式的局部收敛性
2. 比较二分法与牛顿公式的收敛速度
3. 验证求解结果的正确性

3 实验内容

3.1 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。

3.2 牛顿公式

对于方程

$$f(x)=0 \tag{1}$$

已知它的近似根 x_k，则函数 $f(x)$ 在点 x_k 附近可用一阶泰勒展开多项式

$$p(x)=f(x_k)+f'(x_k)(x-x_k) \tag{2}$$

来近似，因此方程 $f(x)=0$ 可近似地表示为 $p(x)=0$。后者是个线性方程，求它的根是容易的，我们取 $p(x)=0$ 的根作为 $f(x)=0$ 的新的近似根，记 x_{k+1}，则有

$$x_{k+1}=x_k-\frac{f(x_k)}{f'(x_k)} \tag{3}$$

这就是牛顿公式。

4 实验过程及主要代码

4.1 验证牛顿公式的局部收敛性

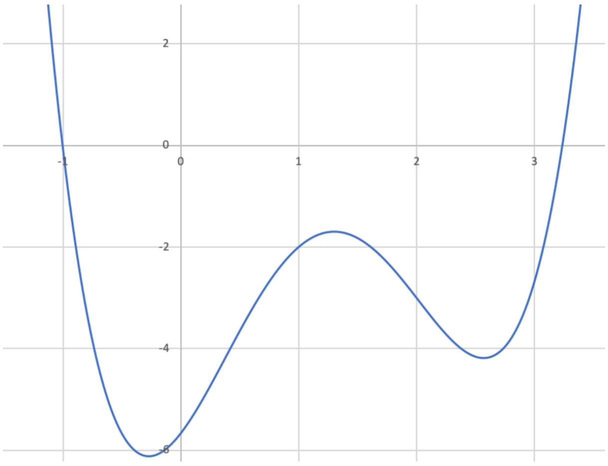
因为牛顿法是局部收敛的，收敛性依赖于 x_0 的选取。

这里我们选取

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

并求 $f(x) = 0$ 的根。

$f(x)$ 的函数图像为



从图像中我们可以发现，$f(x) = 0$ 存在两个根，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 $x_0 = 3$ 和 $x_0 = 2$，作为牛顿公式的起始点进行迭代求根（误差不超过 10^{-3}）。

4.2 比较二分法与牛顿公式的收敛速度

对于上述 $f(x)$，分别使用二分法和牛顿公式求解 $f(x) = 0$ 的根。

4.3 验证求解结果的正确性

对于上述 $f(x)$，使用牛顿公式迭代求解 $f(x) = 0$ 的根 x^*，并带入 $f(x)$ 求函数值，判断 $f(x^*)$ 是否等于或接近 0。

4.4 主要代码

```
#include <stdio.h>
#include <cmath>

#define MAX_TIMES 100
#define e 0.001
#define ROOT 3.2348365
```

```c
/// 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;
}

void Binary() {
    double left = 2.0;
    double right = 4.0;
    double mid;

    printf("Binary:\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);
    }
}

void Newton() {
//    double x0 = 2.0;
    double x0 = 3.0;
    double x1;

    printf("Newton:\n");
    printf("  k\t xk\n");
    for (int i = 1; i <= MAX_TIMES; i++) {
        x1 = x0 - f(x0) / f_d(x0);
        printf("%3d  %f\n", i, x1);
        if (fabs(x1 -x0) < e) { // 精度达到要求
            break;
        }
        x0 = x1;
    }
    printf("f(%f) = %f\n", x1, f(x1));
}
```

```
int main() {
    Binary();
    Newton();

    return 0;
}
```

5 实验结果及分析

5.1 验证牛顿公式的局部收敛性

对于 $x_0 = 3$

```
Newton:
  k  xk
  1  3.333333
  2  3.244547
  3  3.234943
  4  3.234837

Process finished with exit code 0
```

对于 $x_0 = 2$

```
Newton:
  k  xk
  1  1.000000
  2  2.000000
  3  1.000000
  4  2.000000
  5  1.000000
  6  2.000000
        ...
 95  1.000000
 96  2.000000
 97  1.000000
 98  2.000000
 99  1.000000
100  2.000000

Process finished with exit code 0
```

从 $x_0 = 3$ 和 $x_0 = 2$ 的迭代过程，我们可以看出，牛顿公式的收敛性依赖于 x_0 的选取，$x_0 = 3$ 时，通过迭代可以得出 $f(x) = 0$ 的近似根为 $x = 3.234837$，即 $x_0 = 3$ 收敛，而 $x_0 = 2$ 时，x_k 和 x_{k+1} 在 1 和 2 之间循环，最终超过迭代上限，即 $x_0 = 2$ 不收敛。

由上可知，牛顿公式具有局部收敛性。

5.2 比较二分法与牛顿公式的收敛速度

对于相同的方程 $f(x) = 0$，使用二分法（选取区间的左端点作为根的近似值）和牛顿法求解根（误差不超过 10^{-3}），迭代次数以及根输出如下

```
Binary:
  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

Process finished with exit code 0
```

```
Newton:
  k  xk
  1  3.333333
  2  3.244547
  3  3.234943
  4  3.234837

Process finished with exit code 0
```

由此可见，二分法 11 步求得近似根，而牛顿公式仅用 4 步求的近似根，牛顿公式的收敛速度远快于二分法。

5.3 验证求解结果的正确性

使用牛顿公式求得的近似根为 $x = 3.234873$，真实值为 $x = 3.234836$。

```
Newton:
  k  xk
  1  3.333333
  2  3.244547
  3  3.234943
  4  3.234837
f(3.234837) = 0.000000

Process finished with exit code 0
```

牛顿公式求得的近似根与真实值十分接近，并且近似根的函数值十分接近零，牛顿公式正确。

6 实验体会

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 $\varphi(x)$，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 x_0 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 x_0，再使用牛顿公式求解近似根。