实验报告

专业 ____学号____姓名 ______________

一、实验目的

通过编程，进一步理解非线性方程求解的各种方法——二分法、简单迭代法、埃特金迭代加速法、牛顿法。

二、实验题目

求方程 $f(x)=x^3-\cos x-5x-1=0$ 的全部根。

方案一 用二分法求解

方案二 用简单迭代求解

方案三 用埃特金迭代加速法求解

方案四 用牛顿法求解

通过四种方案分别求出方程的解并且比较各方法的收敛速度。

三、实验原理

方案一 二分法

原理：对于求方程 $f(x)=0$，设 $f(a)\cdot f(b)<0$，取 $x_0=\dfrac{a+b}{2}$，假如 $f(x_0)$ 是 $f(x)$ 的零点，那么输出 x_0，停止。否则，

若 $f(a)$ 与 $f(x_0)$ 同号，则 $a_1=x_0$， $b_1=b$；

否则 $a_1=a$， $b_1=x_0$。

……

故 $[a,b]\supset[a_1,b_1]\supset\ldots\supset[a_k,b_k]\ldots$， $x_k=(a_k+b_k)/2\to x^*$，

$\left|x_k-x^*\right|\le(b_k-a_k)/2=(b-a)/2^{k+1}$。

算法：

1）定义变量 p1、p2，误差限ε，并设置初值 p1 = a，p2 = b，（a、b 为初始左、右端点）。

2）设置x_0=a

3）设置 k=1

4）$x_k = \dfrac{p1 + p2}{2}$

5）如果$f(x_k)$=0，输出准确根x_k，迭代次数 k

6）如果$f(x_k)f(p1) < 0$，设置$p2 = x_k$

7）如果$f(x_k)f(p2) < 0$，设置$p1 = x_k$

8）如果$|x_k - x_{k-1}| \geq \varepsilon$，k = k + 1，跳至 4）

9）否则，输出近似根x_k，迭代次数 k，函数值与 0 的绝对误差$|f(x_k) - 0|$。

方案二 简单迭代法

将非线性方程$f(x) = 0$化为等价形式$x = \varphi(x)$，有$f(x^*) = 0 \Leftrightarrow x^* = \varphi(x^*)$，称$x^*$为函数$\varphi(x)$的一个不动点，即$f(x) = 0$的根。

算法：

1）选取一个初始近似值x_0，设定误差限ε。

2）设置 k=0

3）k = k+1

4）求$x_k = \varphi(x_{k-1})$。

5）如果$|x_k - x_{k-1}| \geq \varepsilon$，跳至 3）

6）否则，输出近似根x_k，迭代次数 k，函数值与 0 的绝对误差$\left|f(x_k)-0\right|$。

注：

所选取的迭代函数$\varphi(x)\in C[a,b]$需要满足以下两个条件：

1）$\forall x\in[a,b], a\le\varphi(x)\le b$。

2）$\exists 0\le L<1, s.t.\forall x\in[a,b], \left|\varphi^{'}(x)\right|\le L<1$。

满足上述条件的$\varphi(x)\in C[a,b]$收敛。

方案三 埃特金迭代加速法

该方法的理论基础为简单迭代法，相比简单迭代法，其收敛速度大大加快。

原理：对于收敛的迭代过程，由迭代公式校正一次得$x_1=\varphi(x_0)$，再校正一次得$x_2=\varphi(x_1)$，如果$\varphi^{'}(x)$变化不大，$\varphi^{'}(x)\approx L$，则借由拉格朗日中值定理可得：

$$\begin{cases} x_1-x^*=\varphi(x_0)-\varphi(x^*)\approx L(x_0-x^*) \\ x_2-x^*=\varphi(x_1)-\varphi(x^*)\approx L(x_1-x^*) \end{cases} \Rightarrow x^*=x_0-\frac{(x_1-x_0)^2}{x_2-2x_1+x_0}$$

于是得到埃特金迭代加速法：

$$\bar{x}_{k+1}=x_k-\frac{(x_{k+1}-x_k)^2}{x_{k+2}-2x_{k+1}+x_k}\triangleq x_k-(\Delta x_k)^2/\Delta^2 x_k$$

可以证明$\lim\limits_{k\to\infty}\frac{\bar{x}_{k+1}-x^*}{x_k-x^*}=0$，即该方法比简单迭代法收敛更快。

算法：

1）选取一个初始近似值x_0，运用迭代公式$x_k=\varphi(x_{k\text{-}1})$计算$x_1$、$x_2$、$x_3$。设定误差限$\varepsilon$。

2)运用公式计算$\bar{x}_1$、$\bar{x}_2$。

3)设置 k=2

4)如果$\left|\bar{x}_k-\bar{x}_{k-1}\right|\geq\varepsilon$

5)k = k + 1

6)计算x_{k+1}

7)计算$\bar{x}_k$，跳转至 4)

8)否则，输出近似根$\bar{x}_k$，迭代次数 k，函数值与 0 的绝对误差$\left|f(\bar{x}_k)-0\right|$。

方案四 牛顿法

原理：利用泰勒一阶展开对方程进行线性化。

已知方程 $f(x)=0$ 的近似根 x_k，并假定 $f^{'}(x)\neq 0$，做泰勒展开 $f(x)\approx f(x_k)+f^{'}(x_k)(x-x_k)$，则 $f(x)=0$ 近似表示为

$$f(x_k)+f^{'}(x_k)(x-x_k)=0,$$

记其根为x_{k+1},则有计算公式$x_{k+1}=x_k-\dfrac{f(x_k)}{f^{'}(x_k)}$,迭代公式为$\varphi(x)=x-\dfrac{f(x)}{f^{'}(x)}$

上式即为牛顿迭代法。

算法：

1)选取一个初始近似值x_0，设定误差限ε。

2)设置 k=0

3)k = k+1

4)计算 $f^{'}(x_{k-1})$，如果 $f^{'}(x_{k-1})$=0，宣布牛顿法失败

5)否则，求 $x_k=\varphi(x_{k-1})$。

6) 如果$|x_k - x_{k-1}| \ge \varepsilon$，跳至 3)

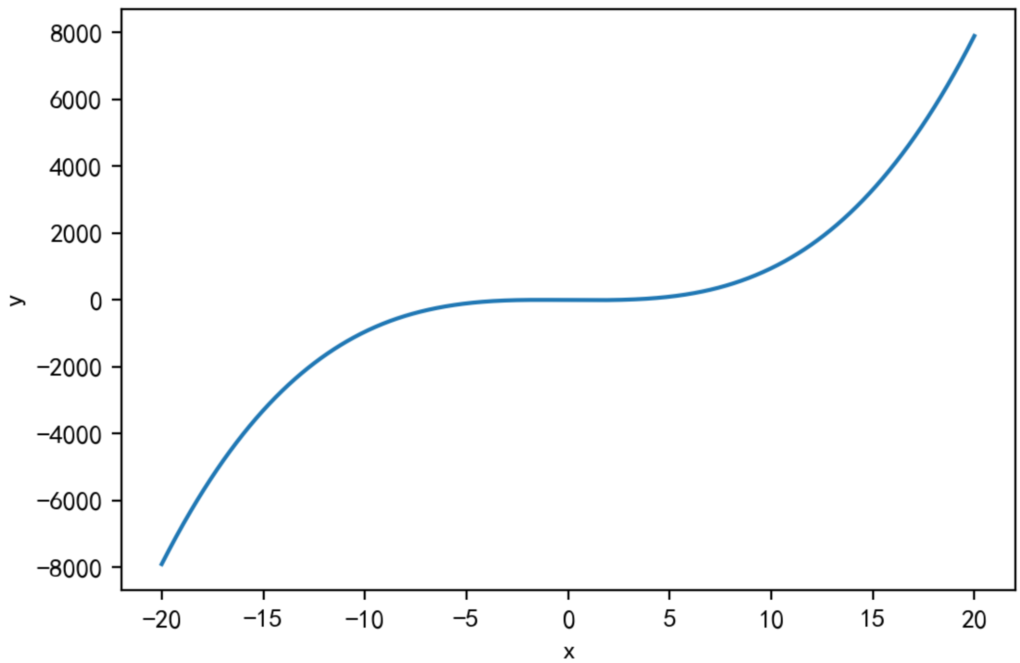
7) 否则，输出近似根x_k，迭代次数 k，函数值与 0 的绝对误差$|f(x_k) - 0|$。

四、实验内容

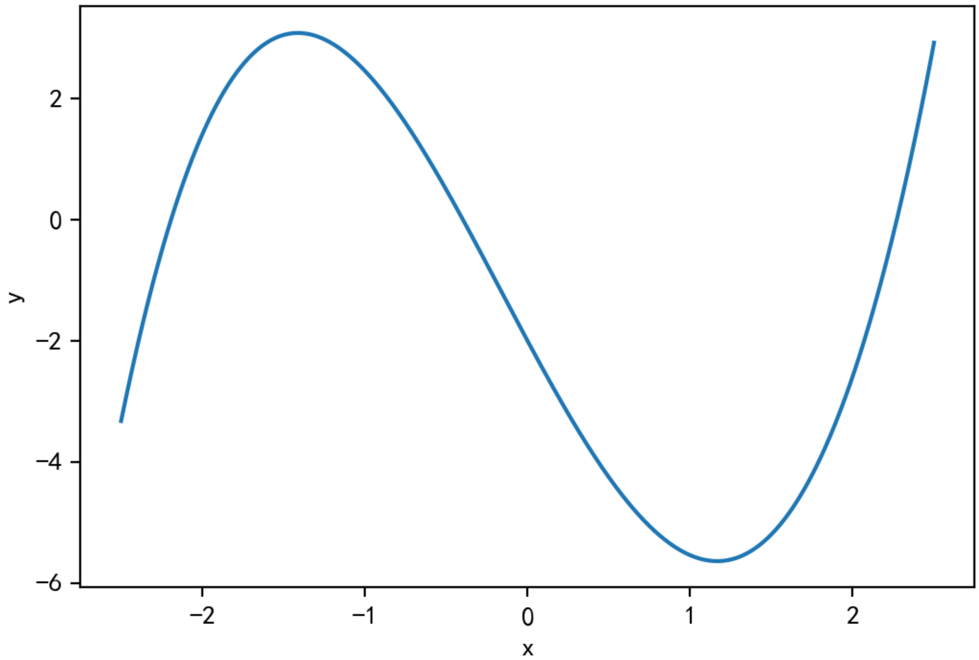
实验分析：

对于函数$f(x) = x^3 - \cos x - 5x - 1$，其导数为$f^{'}(x) = 3x^2 + \sin x - 5$，可见当$|x|$足够大时，$f^{'}(x) > 0$，即原函数$f(x)$持续递增。由此可见在$|x|$较大的区间上应该不存在$f(x) = 0$的实根。

首先尝试绘出一部分$f(x) = x^3 - \cos x - 5x - 1$的图像：



从上图中可以看出，$f(x) = 0$的实根应该只存在于$x \in [-10, 10]$上，借由目测逐步缩小范围，最后画出$x \in [-2.5, 2.5]$的图像：



至此，可以确定 $f(x)=0$ 有三个实数根 x_1,x_2,x_3， $x_1\in[-2.5,-1]$， $x_2\in[-1,0]$， $x_3\in[2.0,2.5]$。

现在分别使用四种方案求这三个根。误差限统一设置为 0.5*10e-8。

方案一 用二分法求解

```
#变量设置
a = -2.5#求 x1,x2,x3 时 a 分别为-2.5,-1,2
b = -1#求 x1,x2,x3 时 b 分别为-1,0,2.5
eps = 0.000000005

p = [a,b]
x = [a,b]#存放每次二分法所得的值，x_0 = a,x_1 = b 仅为方便初始化

#迭代开始
k = 1
while (abs(x[k] - x[k-1]) >= eps):
    temp_p, temp_x = erfen(p, fun)
    if temp_p == p:#说明找到精确解了
        print("精确解为：", temp_x)
        print("迭代次数为：", k)
        break
    else:
        p = temp_p#更新两个端点
        x.append(temp_x)
```

```
        k = k + 1
print("二分法近似根为：",x[k])
print("二分法迭代次数为：",k)
print("二分法迭代根的函数值与 0 的绝对误差：",abs(fun(x[k])))
```

方案二 用简单迭代求解

经过验证，对于 $x_1 \in [-2.5,-1]$， $x_3 \in [2.0,2.5]$， 使用 $\varphi_1(x)=\sqrt[3]{\cos x+5x+1}$，满足收敛条件。对于 $x_2 \in [-1,0]$，使用 $\varphi_2(x)=\dfrac{\cos x+1}{x^2-5}$ 满足收敛条件。

```
#变量设置
a = -2.5#求 x1,x2,x3 时 a 分别为-2.5,-1,2
b = -1#求 x1,x2,x3 时 b 分别为-1,0,2.5
eps = 0.000000005

x = [a]#作为初始近似解
x.append(diedai(a, phi_2)) #作为迭代一次的解,求 x1,x3 时使用 phi_1 函数，求 x2 时使用 phi_2 函数

#迭代开始
k = 1
while (abs(x[k] - x[k - 1]) >= eps):
    temp = diedai(x[k], phi_2) #作为迭代一次的解,求 x1,x3 时使用 phi_1 函数，求 x2 时使用 phi_2 函数
    x.append(temp)
    k = k + 1
print("简单迭代法近似根为：",x[k])
print("简单迭代法迭代次数为：",k)
print("简单迭代法迭代根的函数值与 0 的绝对误差：",abs(fun(x[k])))
```

方案三 用埃特金迭代加速法求解

```
#变量设置
a = -2.5#求 x1,x2,x3 时 a 分别为-2.5,-1,2
b = -1#求 x1,x2,x3 时 b 分别为-1,0,2.5
eps = 0.000000005

#初始化迭代解数组 x,并求 x_0,x_1,x_2
x = [a]#作为初始近似解
x.append(diedai(x[-1], phi_2))#作为迭代一次的解,求 x1,x3 时使用 phi_1 函
```

```
数，求 x2 时使用 phi_2 函数
x.append(diedai(x[-1], phi_2))#作为迭代一次的解,求 x1,x3 时使用 phi_1 函数，求 x2 时使用 phi_2 函数

x_ba = [0]#存储加速所得解，初始化 x_ba[0]定义为 0
x_ba.append(aitejin(x, 1))#先加一次速

#迭代开始
k = 1
while (abs(x_ba[k] - x_ba[k - 1]) >= eps):
    k = k + 1
    temp = diedai(x[k], phi_2)#作为迭代一次的解,求 x1,x3 时使用 phi_1 函数，求 x2 时使用 phi_2 函数
    x.append(temp)
    x_ba.append(aitejin(x, k))

print("埃特金迭代加速法近似根为：",x_ba[k])
print("埃特金迭代加速法加速次数为：",k)
print("埃特金迭代加速法迭代根的函数值与 0 的绝对误差：",abs(fun(x_ba[k])))
```

方案四 用牛顿法求解

```
#变量设置
a = -2.5#求 x1,x2,x3 时 a 分别为-2.5,-1,2
b = -1#求 x1,x2,x3 时 b 分别为-1,0,2.5
eps = 0.000000005

x = [a]#作为初始近似解
x.append(phi(x[-1], fun, dfun))#先迭代一次

#迭代开始
k = 1
while (abs(x[k] - x[k - 1]) >= eps):
    k = k + 1
    if dfun(x[k-1]) == 0:
        print("牛顿法失败。")
        break
    else:
        x.append(phi(x[k-1], fun, dfun))

print("牛顿法近似根为：",x[k])
print("牛顿法迭代次数为：",k)
print("牛顿法迭代根的函数值与 0 的绝对误差：",abs(fun(x[k])))
```

硬件：Personal Computer with Intel CPU

软件：Microsoft Windows, python3.6, Microsoft Office

五、实验结果

- 对于 $x_1 \in [-2.5, -1]$，结果如下：

方案一 二分法

二分法近似根为： -2.19313280005008
二分法迭代次数为： 30
二分法迭代根的函数值与 0 的绝对误差： 2.141842259106852e-10

方案二 用简单迭代求解

简单迭代法近似根为： -2.1931328014211187
简单迭代法迭代次数为： 21
简单迭代法迭代根的函数值与 0 的绝对误差： 1.2028394280605426e-08

方案三 用埃特金迭代加速法求解

埃特金迭代加速法近似根为： -2.193132800396751
埃特金迭代加速法加速次数为： 10
埃特金迭代加速法迭代根的函数值与 0 的绝对误差： 3.20144266652278e-09

方案四 用牛顿法求解

牛顿法近似根为： -2.1931328000252237
牛顿法迭代次数为： 5
牛顿法迭代根的函数值与 0 的绝对误差： 1.7763568394002505e-15

- 对于 $x_2 \in [-1, 0]$，结果如下：

方案一 用二分法求解

二分法近似根为： -0.39695845916867256
二分法迭代次数为： 29
二分法迭代根的函数值与 0 的绝对误差： 1.1996273974190785e-09

方案二 用简单迭代求解

简单迭代法近似根为： -0.39695845942101643
简单迭代法迭代次数为： 6
简单迭代法迭代根的函数值与 0 的绝对误差： 4.036193601564264e-11

方案三 用埃特金迭代加速法求解

埃特金迭代加速法近似根为： -0.3969584594128011
埃特金迭代加速法加速次数为： 4
埃特金迭代加速法迭代根的函数值与 0 的绝对误差： 7.327471962526033e-15

方案四 用牛顿法求解

牛顿法近似根为： -0.3969584594128026
牛顿法迭代次数为： 5
牛顿法迭代根的函数值与 0 的绝对误差： 0.0

• 对于 $x_3 \in [2.0, 2.5]$，结果如下：

方案一 用二分法求解

二分法近似根为： 2.2708289436995983
二分法迭代次数为： 28
二分法迭代根的函数值与 0 的绝对误差： 1.2804116877873639e-08

方案二 用简单迭代求解

简单迭代法近似根为： 2.2708289438347045
简单迭代法迭代次数为： 15
简单迭代法迭代根的函数值与 0 的绝对误差： 1.1286225287676643e-08

方案三 用埃特金迭代加速法求解

埃特金迭代加速法近似根为： 2.270828944901395
埃特金迭代加速法加速次数为： 7
埃特金迭代加速法迭代根的函数值与 0 的绝对误差： 6.978453370720672e-10

方案四 用牛顿法求解

牛顿法近似根为： 2.270828944839281
牛顿法迭代次数为： 5
牛顿法迭代根的函数值与 0 的绝对误差： 3.552713678800501e-15

六、实验结果分析

通过实验结果，可以看到，对于方程 $f(x)=0$ 的三个实数根 x_1, x_2, x_3，在相同的误差限要求下：

1. 牛顿法的收敛速度是最快的，仅需迭代 5 次，且其函数误差也是最低的，都能达到 e-15 的精确度。

2. 埃特金迭代加速法作为简单迭代法的改进算法，收敛速度仅次于

牛顿法。在求根x_2时，其收敛速度甚至快于牛顿法，所得根的精度也较为理想。

3.简单迭代法比二分法的收敛速度要高许多。

综上所述，针对此问题各方法的收敛速度比较结果为：

牛顿法>埃特金迭代加速法>简单迭代法>二分法。

附录：

程序清单：

#二分法

```
import numpy as np
import math

def fun(x):
    return x ** 3 - math.cos(x) - 5 * x - 1

#一次二分
def erfen(p, fun):#p为一个list，里面存放了区间两个端点的值，fun为函数fun(x)
    x_k = (p[0] + p[1]) / 2.0
    if fun(x_k) != 0:#未得到精确解
        if fun(x_k) * fun(p[0]) < 0:
            new_p = [x_k, p[0]]
        else:
            new_p = [x_k, p[1]]
    return new_p, x_k
```

#简单迭代法

```
import numpy as np
import math

def fun(x):
    return x ** 3 - math.cos(x) - 5 * x - 1

def phi_1(x):#用于求x1，x3的迭代函数
    temp = math.cos(x) + 5 * x + 1
    if temp >= 0:
        result = pow(temp, 1/3.0)
    else:
        temp = abs(temp)
        result = pow(temp, 1/3.0)
        result = result * (-1)
    return result

def phi_2(x):#用于求x2的迭代函数
    return (math.cos(x)+ 1) / (x ** 2 - 5)

#一次迭代
```

```
def diedai(x_temp, phi):#x_temp 为当前迭代解，phi 为迭代函数
    return phi(x_temp)
```

#埃特金迭代加速法

```
import numpy as np
import math

def fun(x):
    return x ** 3 - math.cos(x) - 5 * x - 1

def phi_1(x):#用于求 x1，x3 的迭代函数
    temp = math.cos(x) + 5 * x + 1
    if temp >= 0:
        result = pow(temp, 1/3.0)
    else:
        temp = abs(temp)
        result = pow(temp, 1/3.0)
        result = result * (-1)
    return result

def phi_2(x):#用于求 x2 的迭代函数
    return (math.cos(x)+ 1) / (x ** 2 - 5)

#一次迭代
def diedai(x_temp, phi):#x_temp 为当前迭代解，phi 为迭代函数
    return phi(x_temp)

#一次埃特金加速
def aitejin(x, k):#x 为迭代法所得解为一个 list，k 为迭代次数
    return x[k-1] - (x[k] - x[k-1]) ** 2 / (x[k+1] - 2 * x[k] + x[k-1])
```

#牛顿法

```
import numpy as np
import math

def fun(x):
    return x ** 3 - math.cos(x) - 5 * x - 1

def dfun(x):
    return 3 * x ** 2 + math.sin(x) - 5

def phi(x, fun, dfun):
    return x - fun(x) / dfun(x)
```