一. 实验目的

① 掌握二分法、牛顿迭代法等常用的非线性方程迭代算法；
② 了解迭代算法的设计原理及初值对收敛性的影响。

二.实验过程和结果

1. 二分法实现[1]：

二分法的思想是对区间进行二分，在左右两个区间中确定下一次求根搜索的区间，如此反复来得到方程的根。

该算法的实现为以下函数：

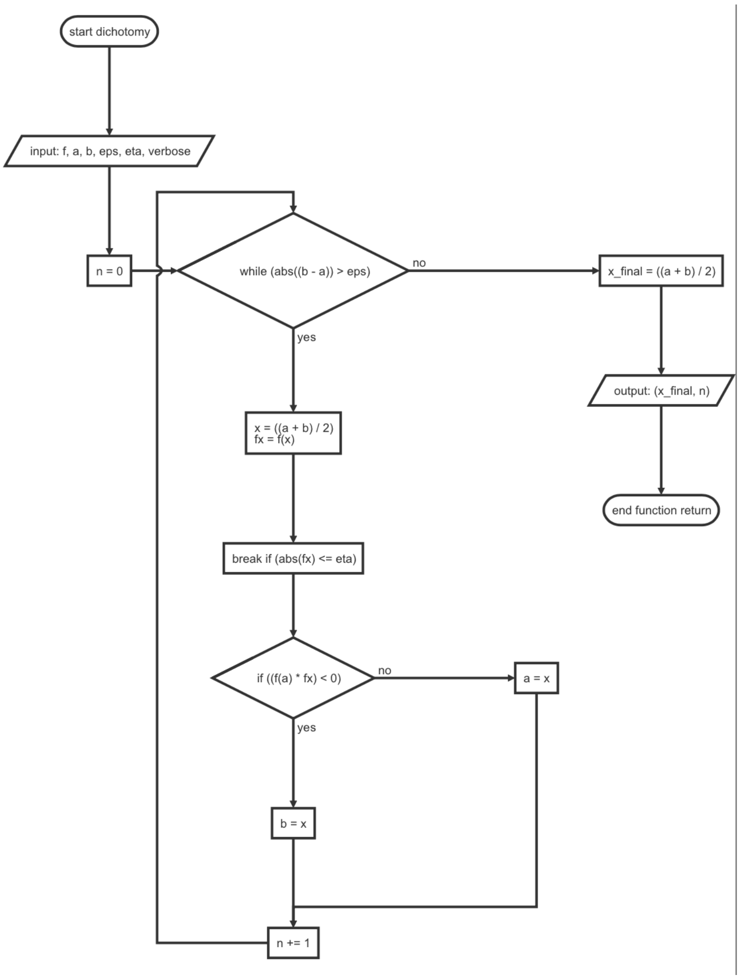
```
dichotomy(f, a, b, eps,
eta=1e-16, verbose=False)
```

输入：一元函数，表示要求根的方程: $f(x) = 0$， 有根区间 $[a, b]$ 的端点，以及给定精度eps。

输出：二分法求得的近似根 x_final 和迭代次数 N。

其核心代码如下：

```
while abs(b - a) > eps:
    x = (a + b) / 2
    if abs(f(x))<=eta:
        break
    if f(a) * f(x) < 0:
        b = x
    else:
        a = x
x_final = (a + b) / 2
```



【图1】二分法程序流程图

[1] 二分法完整代码实现见：https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex2/src/dichotomy.py

二分法实现测试:

```
In [45]: def f(x):
             return 2 * exp(-x) - sin(x)

         bisection(f, 0, 1, 0.0005, verbose=True)

         N = 10

         n        (a, b)          f(x_n)
         -----------------------------------
         0        (0, 1)          f(0.5)=0.733635780821064
         1        (0.5, 1)        f(0.75)=0.263094345458695
         2        (0.75, 1)       f(0.875)=0.0661805371209897
         3        (0.875, 1)      f(0.9375)=-0.0228698549070950
         4        (0.875, 0.9375)         f(0.90625)=0.0208763999947352
         5        (0.90625, 0.9375)       f(0.921875)=-0.00119109771321235
         6        (0.90625, 0.921875)     f(0.9140625)=0.00979401298210625
         7        (0.9140625, 0.921875)   f(0.91796875)=0.00428930379438952
         8        (0.91796875, 0.921875)          f(0.919921875)=0.00154606529293533
         9        (0.919921875, 0.921875)         f(0.9208984375)=0.000176724441991016
         10       (0.9208984375, 0.921875)        f(0.92138671875)=-0.000507376461447939
         11       (0.9208984375, 0.92138671875)   -

         result: x = (0.9208984375+0.92138671875)/2 = 0.921142578125
Out[45]: (0.921142578125, 11)
```

2. 不动点迭代[2]

不动点迭代的思想很简单，就是用一个函数反复作用于 x 来等到新的 x。

用如下函数来实现这个方法:

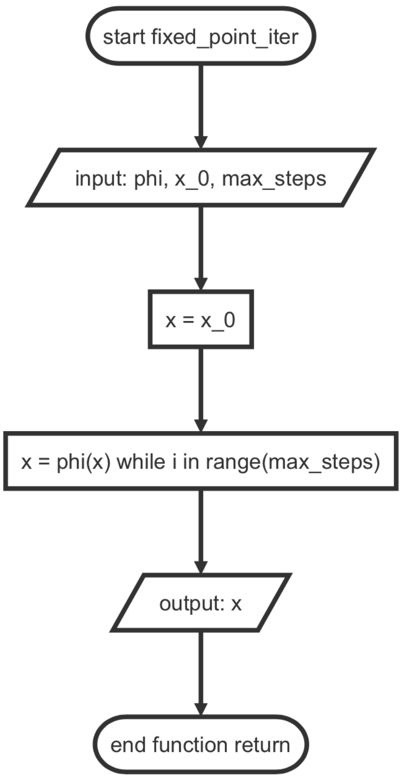
```
fixed_point_iter(phi, x_0, max_steps=25, verbose=False)
```

输入:

- phi: function, 迭代函数
- x_0: float, 初值
- max_steps: 最大迭代次数

输出:

- x_final: float, 最终的近似根 x



【图2】不动点迭代程序流程图

2 不动点迭代的完整代码见: https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex2/src/fixed_point_iter.py

不动点迭代的核心代码为：

```python
x = x_0
for i in range(max_steps):
    x = phi(x)
```

不动点迭代测试：

```
In [65]: fixed_point_iter(lambda x: 1 + 1 / x**2, 1.5, max_steps=10, verbose=True)

         x_0      1.5
         x_1      1.4444444444444444
         x_2      1.4792899408284024
         x_3      1.456976
         x_4      1.4710805833200253
         x_5      1.4620905354712408
         x_6      1.4677905760195855
         x_7      1.464164380462178
         x_8      1.4664663557170745
         x_9      1.465003040566855
Out[65]: 1.4659324390818347
```

3. 牛顿迭代法[3]

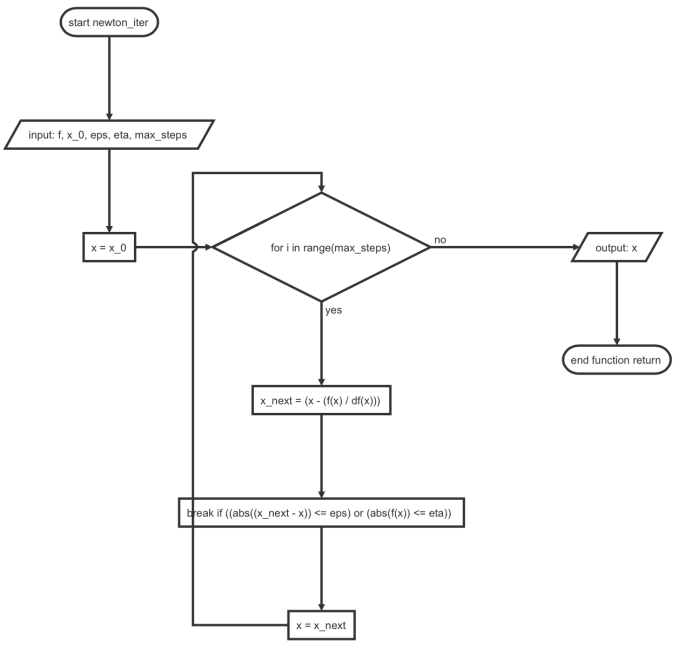
牛顿迭代法用函数的导数（切线）去迭代逼近曲线的根。这个方法的收敛速度明显优于二分法。

实现函数：

```python
newton_iter(f, x_0, eps=0, eta=0, df=None, max_steps=20, frac=False, verbose=False)
```

输入参数：

- f: function, 迭代函数
- x_0: float, 初值
- eps: float, 根的容许误差, default 0.
- eta: float, abs(f(x)) 的容许误差, default 0.



【图3】牛顿迭代程序流程图

---

[3] 牛顿迭代法的完整代码见：https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex2/src/newton_iter.py

- df: function, f 的导函数，默认 None 表示自动调用 sympy.diff 求导(会导致后续迭代中使用分数运算)。
- max_steps: int, 最大迭代次数, default 20.
- frac: bool, True 则输出分数(仅对 df=None 时生效)，否则使用 float, default False.
- verbose: bool, 打印出每一步的值, default False.

输出：

- x_final: float, 最终的近似根 x

该算法的核心代码是：

```
x = x_0
for i in range(max_steps):
    x_next = x - f(x) / df(x)
    if abs(x_next - x) <= eps or abs(f(x)) <= eta:
       break
    x = x_next
```

牛顿迭代法测试：

```
In [128]: def f(x):
              return x ** 3 - 2 * x - 5

          def df(x):
              return 3 * x ** 2 - 2

          # newton_iter(f, 2, eps=0.5e-3, frac=True, verbose=True)
          newton_iter(f, 2, eps=0.5e-3, df=df, frac=True, verbose=True)

          0 2
          1 2.1
          2 2.094568121104185
Out[128]: 2.094568121104185
```

4. 单点弦截法[4]

虽然牛顿法的收敛速度快，但每一步都需要计算导数值 $f'(x_{k-1})$。为了避开导数的计算，使用 $\dfrac{f(x_{k-1} - f(x_0))}{x_{k-1} - x_0}$ 来代替导数，就得到了单点弦截法：

$$x_k = x_{k-1} - \frac{x_{k-1} - x_0}{f(x_{k-1}) - f(x_0)} f(x_{k-1}) \qquad k = 2,3,\ldots$$

[4] 单点弦截法的完整代码见：https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex2/src/single_point_truncation.py

编程实现：

`single_point_truncation(f, x_0, x_1, eps=0, eta=0, max_steps=20, verbose=False)`

这个函数和牛顿迭代的实现 newton_iter 很类似，只需把迭代更新修改了一下：

```
x_next = x - f(x) / (f(x) - f_x0) * (x - x_0)
```

输入：

- f: function, 迭代函数
- x_0, x_1: float, 初值

输出：

- x_final: float, 最终的近似根 x

测试：

```
In [111]: def f(x):
              return x ** 3 - 2 * x - 5

          single_point_truncation(f, 2, 1, eps=0.5e-3, verbose=True)

          0 1
          1 2.2
          2 2.088967971530249
          3 2.094861151990966
Out[111]: 2.094861151990966
```

5. 两点弦截法（割线法）[5]

使用差商 $\dfrac{f(x_k)-f(x_{k-1})}{x_k-x_{k-1}}$ 来代替牛顿迭代法中的导数，就得到了两点弦截法（割线法）：

$$x_k = x_{k-1} - \frac{x_{k-1} - x_{x-2}}{f(x_{k-1}) - f(x_{k-2})} f(x_{k-1}) \qquad k = 2,3,4,\ldots$$

[5] 两点弦截法（割线法）的完整代码见：https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex2/src/secant_method.py

编程实现：

```
secant_method(f, x0, x1, eps=0, eta=0, max_steps=20, verbose=False)
```

代码中只需要修改迭代更新：

```
x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
x0, x1 = x1, x2
```

测试：

```
In [123]: secant_method(lambda x: x ** 2 - 612, 10, 30, max_steps=5, verbose=True)
          # Root: 24.738633748750722

0 30
1 22.8
2 24.545454545454547
3 24.746543778801843
4 24.73860275369709

Out[123]: 24.738633748750722
```

6. 题目 [6]

求方程 $f(x)=x^3+x^2-3x-3=0$ **在1.5 附近的根.（误差限为** $\varepsilon=1e-6, \eta=1e-9$ **）。参考答案：原方程的根为** $x=1.732051$

```
In [130]: def f(x):
              return x ** 3 + x ** 2 - 3 * x - 3

          x0 = 1.5
          x1 = 2
          eps = 1e-6
          eta = 1e-9

          result = {}
          print("牛顿迭代法:")
          result["牛顿迭代法"] = newton_iter(f, x0, eps=eps, eta=eta, verbose=True)
          print("单点弦截法:")
          result["单点弦截法"] = single_point_truncation(f, x0, x1, eps=eps, eta=eta, verbose=True)
          print("两点弦截法:")
          result["两点弦截法"] = secant_method(f, x0, x1, eps=eps, eta=eta, verbose=True)

          print("\nresult:")
          for k in result:
              print(f'{k}: {result[k]}')
```

输出结果:

```
牛顿迭代法:
0 1.5
1 1.77777777777778
2 1.73336066694000
3 1.73205192940947
4 1.73205080756970
单点弦截法:
0 2
1 1.6923076923076923
2 1.7390156515180086
3 1.7308625826467308
4 1.7322544663053168
5 1.732015928689518 7
6 1.7320567817872679
7 1.7320497843005194
8 1.732050982835706
两点弦截法:
0 2
1 1.6923076923076923
2 1.7257977285018928
3 1.7322172842612025
4 1.732050123979108

result:
牛顿迭代法: 1.7320508075697012
单点弦截法: 1.732050982835706
两点弦截法: 1.7320508074943775
```

[6] 具体实现见：https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex2/src/ex2.ipynb

三、思考题分析解答

二分法简单易行，但固定每次缩短一半的区间，只能求单实根，不能求复根或偶数重根。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。

虽然牛顿法的收敛速度快，但每一步都需要计算导数值 $f'(x_{k-1})$。为了避开导数的计算，使用 $\dfrac{f(x_{k-1}-f(x_0))}{x_{k-1}-x_0}$ 来代替导数，即单点弦截法；或者，也可以使用差商 $\dfrac{f(x_k)-f(x_{k-1})}{x_k-x_{k-1}}$ 来代替导数，即两点弦截法（割线法）。

四、重点难点分析

重点:

1. 掌握二分法、牛顿迭代法等常用的非线性方程迭代算法；
2. 了解迭代算法的设计原理及初值对收敛性的影响。

难点:

1. 牛顿迭代法的改进。