

作业二：非线性方程求根

英才 1701 赵鹏威 U201710152

2019 年 9 月 21 日

目录

1 引言	2
2 问题描述	2
3 程序实现	2
3.1 辅助模块	2
3.1.1 result 类型	3
3.1.2 数值导数	3
3.1.3 判断是否一定收敛	3
3.2 求根的主要算法	5
3.3 接口子程序与区间扫描	7
4 运行时结果	8
4.1 二分法	8
4.2 Jacobi 迭代法	9
4.3 牛顿下山法	11
4.4 事后加速法	12
4.5 Atiken 法	14
5 各方法的比较	15

1 引言

在物理中会遇到很多方程求根的问题，一般这些方程是非线性的，甚至是超越方程。另外，方程求根问题实际上就是求函数的零点，这也对应着求函数极值点的问题。因此开发有效的数值求根方法是很有必要的。常用的算法有：二分法、Jacobi 迭代法、牛顿下山法。为了让 Jacobi 迭代法收敛更快，又有人提出了事后加速法、Atiken 加速法。这次作业就是通过 Fortran 来实现这些算法。

2 问题描述

问题 1. 使用不同的算法求非线性方程

$$f(x) = \frac{x^3}{3} - x = 0$$

的根，并比较它们的性能，包括：结果的准确性、误差大小、迭代次数

这是一个一元三次方程，很容易得到这个方程的解析解

$$x_1 = -\sqrt{3} \approx -1.732051 \quad x_2 = 0 \quad x_3 = \sqrt{3} \approx 1.732051.$$

这里将解析解四舍五入到了精确到 6 位小数的数值，之后会将由算法得到的结果与这个结果来比较，验证算法的准确性。

3 程序实现

希望程序达到的效果是：向程序提供函数 $f(x)$ （和迭代式 $\varphi(x)$ ，如果有的话），区间 $[a, b]$ ，程序可以调用某种指定的算法，在指定的精度下，找出这个区间内所有可能的根。

为了达到这个目的，通过两步来实现整个程序。第一步是将每种算法分别单独写成一个 subroutine，这些 subroutine 以函数 f ，迭代式 φ ，初值（对二分法来说是初始的区间）和要求的精度为输入参数，返回最多一个找到的根。第二步是另写一个 subroutine 作为调用这些算法的接口程序，它的作用是将输入的区间等分成若干个小区间，然后调用第一步中的子程序在这些小区间内寻找根，并且对于方便预先判断收敛性的算法，在调用之前会自动判断这个区间内的迭代是否一定会收敛，跳过不一定收敛的区间，这样可以省去很多不必要的计算。

下面详细阐述实现细节。

3.1 辅助模块

这个模块存放一些对程序实现有帮助但不是必须的东西，包括一些常数、新的类型定义和一些函数。

3.1.1 result 类型

由于几乎在所有的情况下，数值计算的结果具有一定的误差，所以可以定义一个 `result` 类型，将计算结果和误差封装在一起储存。

Listing 1: `result` 类型

```
1 type result
2     real(8) :: value
3     real(8) :: error
4 end type
```

3.1.2 数值导数

一些程序里需要求导数，比如牛顿下山法，因此写一个 `function` 来实现数值导数的功能。这样计算的导数必然会有误差，但是可以证明这些误差不会累加到最后的結果上。

Listing 2: 数值导数

```
1 real(8) function numerical_derivative(func, x) result(dfdx)
2 !-----
3 ! This function computes the numerical derivative of func at x.
4 ! It returns the results in real(8).
5 !
6 ! Arguments:
7 !     func: a real(8) function
8 !     x: a real(8) number
9 !-----
10 implicit none
11 real(8), external :: func
12 real(8), intent(in) :: x
13
14 real(8) :: h
15 h = 1e-5
16
17 dfdx = (func(x+h) - func(x-h)) / (2*h)
18
19 return
20 end function
```

3.1.3 判断是否一定收敛

Jacobi 迭代法以及对应的加速方法都可以提前判断迭代是否会收敛。为了节约计算资源，在迭代前对迭代是否收敛进行判断。根据理论，只要迭代函数 $\varphi(x)$ 在区间 $[a, b]$ 满足下面两个条件就可以保证迭代一定会收敛：

- $\forall x \in [a, b], \varphi(x) \in [a, b]$
- $\exists L < 1 \forall x \in [a, b], |\varphi'(x)| \leq L$

要实现收敛性的判断需要知道 $\varphi(x)$ 和它的导数 $\varphi'(x)$ 在区间 $[a, b]$ 上的取值范围，这里使用最直接的方法，也就是将区间均匀分成一些各点，计算 $\varphi(x)$ 和它的导数 $\varphi'(x)$ 在这些点上的值，取最大值和最小值作为函数在这个区间的上下限。

Listing 3: 收敛性预判断

```

1  subroutine convergence_check(phi, a, b, stat)
2  !-----
3  ! This subroutine check whether the iteration x = phi(x) converges in [a, b].
4  ! It return the results in stat. If converges, stat is true; if not, false.
5  !
6  ! Arguments:
7  !     phi: a real(8) function
8  !     a: real(8), the left boundary of the interval
9  !     b: real(8), the right boundary of the interval
10 !     stat: logical, save the state of convergence
11 !-----
12     implicit none
13     real(8), external :: phi
14     real(8), intent(in) :: a, b
15     logical, intent(out) :: stat
16
17     real(8), dimension(EVAL_NUM) :: xs, phis, dphis
18     real(8) :: eval_step
19     integer :: i
20
21     eval_step = abs(a - b) / EVAL_NUM
22
23     do i = 1, EVAL_NUM
24         xs(i) = a + eval_step * i
25         phis(i) = phi(xs(i))
26         dphis(i) = numerical_derivative(phi, xs(i))
27     end do
28
29     stat = .false.
30     if (maxval(phis) <= maxval((/a, b/)) .and. minval(phis) >= minval((/a, b/))) then
31         if (maxval(abs(dphis)) < 1) then
32             stat = .true.
33         end if
34     end if
35
36     return

```

3.2 求根的主要算法

将二分法、Jacobi 迭代法、牛顿下山法、事后加速法和 Atiken 加速法封装到一个模块里。每种方法都用 subroutine 实现，如二分法，输入函数、区间和精度，返回满足精度要求的结果。

Listing 4: 二分法

```

1  subroutine bisection(f, left_init, right_init, epsilon1, epsilon2, file_name, root, iter,
   stat)
2      implicit none
3      real(8), external :: f
4      real(8), intent(in) :: left_init, right_init, epsilon1, epsilon2
5      character(50), intent(in) :: file_name
6      type(result), intent(out) :: root
7      integer, intent(out) :: iter
8      logical, intent(out) :: stat
9
10     real(8), dimension(MAX_ITER_NUM) :: history
11     real(8) :: left, middle, right, lv, mv, rv
12     integer :: i
13     logical :: stop_condition
14
15     left = left_init
16     right = right_init
17     middle = (left + right) / 2
18     i = 1
19     history(i) = middle
20     stat = .true.
21
22     stop_condition = .false.
23     do while (.not. stop_condition)
24         lv = f(left)
25         mv = f(middle)
26         rv = f(right)
27
28         if (abs(lv - 0.0d0) <= epsilon2) then
29             right = left + 2 * epsilon1
30         else if (abs(rv - 0.0d0) <= epsilon2) then
31             left = right - 2 * epsilon1
32         end if
33
34         if (lv * rv < 0) then

```

```

35         if (lv * mv < 0) then
36             right = middle
37         else if (mv * rv < 0) then
38             left = middle
39         end if
40     else
41         stat = .false.
42         exit
43     end if
44
45     middle = (left + right) / 2
46     i = i + 1
47     history(i) = middle
48
49     stop_condition = (abs(right - left) <= epsilon1) &
50                     .and. (abs(f(middle)) <= epsilon2)      &
51                     .and. (i <= max_iter_num)
52
53 end do
54
55 root%value = middle
56 root%error = right - left
57 iter = i
58
59 if (i == MAX_ITER_NUM) then
60     stat = .false.
61 end if
62
63 open(file=file_name, unit=10, action="write")
64 do i = 1, iter
65     write (10, "(i8, a4, f8.6)") i, " ", history(i)
66 end do
67 close(unit=10)
68
69 return
70 end subroutine

```

其他方法的实现大同小异，为了文章的简洁，源码见附录（Jacobi 迭代法：第 169 行，牛顿下山法：第 216 行，事后加速法：第 272 行，Atiken：第 320 行）。需要注意的是，这些子程序都最多只能找到一个根。另外有一些值得说明一下的细节

- 牛顿下山法中的参数 λ 在每次迭代前会恢复为 1；
- 牛顿下山法中计算导数使用数值导数（3.1.2 节）；

- 事后加速法中第 n 次迭代使用的 L 取的是当前位置导数值 $\varphi'(x_n)$ ，同样用数值导数.

3.3 接口子程序与区间扫描

上面实现的方法只能找到一个根，但是我们希望的是尽可能一次性找出所有可能的根. 可以采用这样的做法，给定一个区间 $[a, b]$ ，将这个区间等分成若干个小区间，在这些小区间里调用上面写的各种求根算法的子程序. 如果使用的方法为 Jacobi 迭代、事后加速法和 Atiken 法，那么在调用求根子程序前先判断在这个小区间内能否一定收敛，如果是的话才调用程序. 这样就可以找出一个较大的区间内的所有可能的根. 将上面所述的区间扫描功能写在一个 subroutine 内，这个 subroutine 作为调用求根函数的接口. 它的功能可以用下面的流程图表示，见图1. 代码见附录第 532 行.

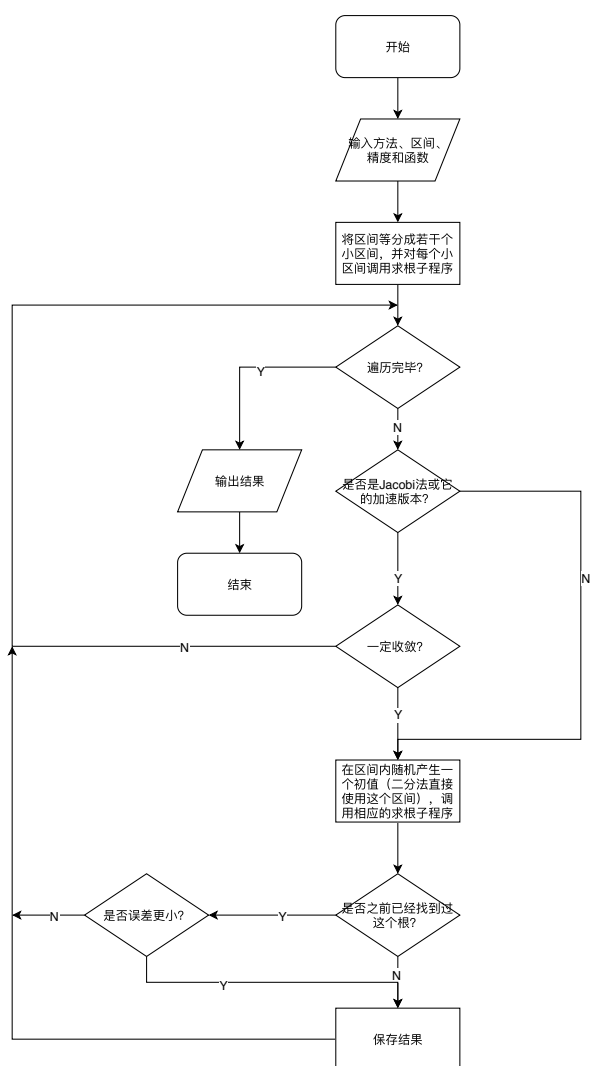


图 1: 接口子程序的流程图

4 运行时结果

下面各方法要求的精度为

$$|x_k - x_{k-1}| \leq 10^{-5}$$

$$|f(x_k)| \leq 10^{-5}$$

输入区间为 $[-10, 10]$.

4.1 二分法

主程序代码如下

Listing 5: 二分法主程序

```
1 program main
2   use utils
3   use roots_seeker
4
5   implicit none
6   real(8), external :: f, phi1, phi2
7   real(8), parameter :: step_size = 0.1d0, epsilon1 = 1e-5, epsilon2 = 1e-5
8   type(result), dimension(MAX_ROOTS_NUM) :: roots
9   integer, dimension(MAX_ROOTS_NUM) :: iters
10  real(8) :: left, right
11  character(50) :: file_name = "history.dat"
12  character(20) :: seeker = "bisection"
13  integer :: iter, root_num, i
14  logical :: stat
15
16  left = -10.0d0
17  right = 10.0d0
18
19  print *, "method: "//trim(seeker)
20  call multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1, epsilon2,
21    file_name, iters, f)
22  print "(a35, i3)", "The number of roots that is found: ", root_num
23  do i = 1, root_num
24    print "(a8, f32.16)", "root: ", roots(i)%value
25    print "(a8, f32.16)", "error: ", roots(i)%error
26    print "(a8, i32)", "iter: ", iters(i)
27    print *, " "
28  end do
29 end program
```

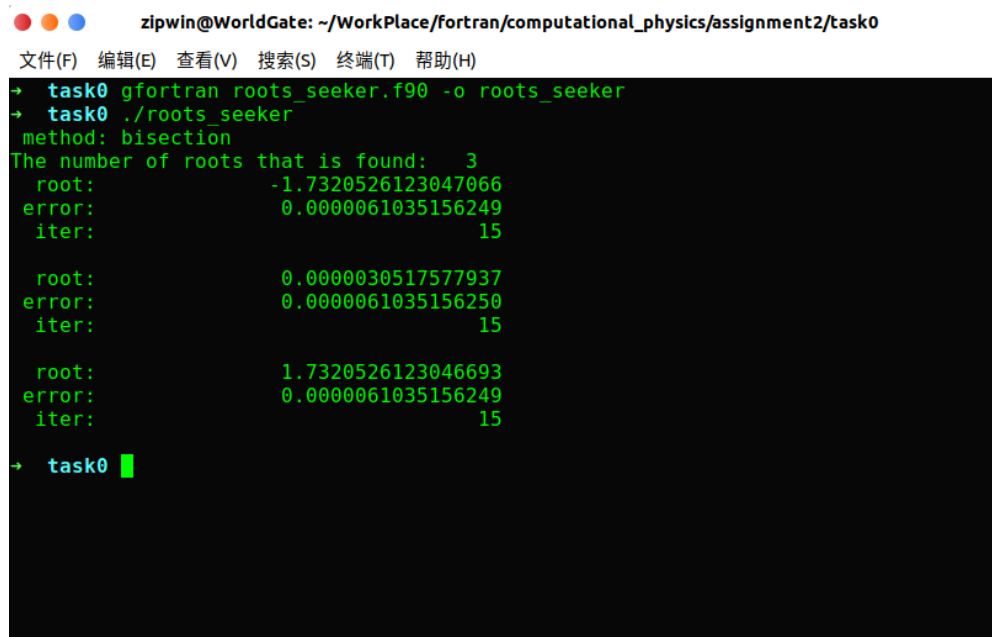

结果如图2所示. 求得的结果为 (取到第 6 位小数)

$$x_1 = -1.732053(6)$$

$$x_2 = 0.000003(6)$$

$$x_3 = 1.732053(6)$$

括号内为最后一位误差. 可见与准确的结果符合.



```
zipwin@WorldGate: ~/WorkPlace/fortran/computational_physics/assignment2/task0
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
→ task0 gfortran roots_seeker.f90 -o roots_seeker
→ task0 ./roots_seeker
method: bisection
The number of roots that is found: 3
root: -1.7320526123047066
error: 0.0000061035156249
iter: 15

root: 0.0000030517577937
error: 0.0000061035156250
iter: 15

root: 1.7320526123046693
error: 0.0000061035156249
iter: 15
→ task0
```

图 2: 二分法

4.2 Jacobi 迭代法

主程序如下

Listing 6: Jacobi 迭代法主程序

```
1 program main
2   use utils
3   use roots_seeker
4
5   implicit none
6   real(8), external :: f, phi1, phi2
7   real(8), parameter :: step_size = 0.1d0, epsilon1 = 1e-5, epsilon2 = 1e-5
8   type(result), dimension(MAX_ROOTS_NUM) :: roots
9   integer, dimension(MAX_ROOTS_NUM) :: iters
10  real(8) :: left, right
11  character(50) :: file_name = "history.dat"
```

```

12     character(20) :: seeker = "jacobi"
13     integer :: iter, root_num, i
14     logical :: stat
15
16     left = -10.0d0
17     right = 10.0d0
18
19     print *, "method: "//trim(seeker)
20     print *, "phi: (3*x)^{1/3}"
21     call multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1, epsilon2,
22         file_name, iters, f, phi1)
23     print "(a35, i3)", "The number of roots that is found: ", root_num
24     do i = 1, root_num
25         print "(a8, f32.16)", "root: ", roots(i)%value
26         print "(a8, f32.16)", "error: ", roots(i)%error
27         print "(a8, i32)", "iter: ", iters(i)
28         print *, " "
29     end do
30
31     print *, "phi: x^3/3"
32     call multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1, epsilon2,
33         file_name, iters, f, phi2)
34     print "(a35, i3)", "The number of roots that is found: ", root_num
35     do i = 1, root_num
36         print "(a8, f32.16)", "root: ", roots(i)%value
37         print "(a8, f32.16)", "error: ", roots(i)%error
38         print "(a8, i32)", "iter: ", iters(i)
39         print *, " "
40     end do
41 end program

```

Jacobi 迭代法使用了两个迭代函数

$$\varphi_1(x) = (3x)^{\frac{1}{3}}, \quad \varphi_2(x) = \frac{x^3}{3}$$

运行的结果如图3所示. 结果为

$$x_1 = -1.732053(5)$$

$$x_2 = 0.000000(0)$$

$$x_3 = 1.732048(5)$$

其中 x_1, x_3 是使用 $\varphi_1(x) = (3x)^{1/3}$ 得到的, x_2 是使用 $\varphi_2(x) = x^3/3$ 得到的.

```
zipwin@WorldGate: ~/WorkPlace/fortran/computational_physics/assignment2/task0
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
→ task0 gfortran roots_seeker.f90 -o roots_seeker
→ task0 ./roots_seeker
method: jacobi
phi: (3*x)^(1/3)
The number of roots that is found: 2
root: -1.7320530958850135
error: 0.0000045766413421
iter: 11

root: 1.7320484760871435
error: 0.0000046629540520
iter: 10

phi: x^3/3
The number of roots that is found: 1
root: -0.0000000000000000
error: 0.0000000000000014
iter: 4
→ task0 █
```

图 3: Jacobi 迭代法

4.3 牛顿下山法

主程序如下

Listing 7: 牛顿法主程序

```
1 program main
2   use utils
3   use roots_seeker
4
5   implicit none
6   real(8), external :: f, phi1, phi2
7   real(8), parameter :: step_size = 0.1d0, epsilon1 = 1e-5, epsilon2 = 1e-5
8   type(result), dimension(MAX_ROOTS_NUM) :: roots
9   integer, dimension(MAX_ROOTS_NUM) :: iters
10  real(8) :: left, right
11  character(50) :: file_name = "history.dat"
12  character(20) :: seeker = "downhill"
13  integer :: iter, root_num, i
14  logical :: stat
15
16  left = -10.0d0
17  right = 10.0d0
18
19  print *, "method: "//trim(seeker)
```

```

20     call multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1, epsilon2,
        file_name, iters, f)
21     print "(a35, i3)", "The number of roots that is found: ", root_num
22     do i = 1, root_num
23         print "(a8, f32.16)", "root: ", roots(i)%value
24         print "(a8, f32.16)", "error: ", roots(i)%error
25         print "(a8, i32)", "iter: ", iters(i)
26         print *, " "
27     end do
28
29 end program

```

运行结果如图4所示. 结果为

$$x_1 = -1.732051(0)$$

$$x_2 = 0.000000(0)$$

$$x_3 = 1.732051(0)$$

可见牛顿下山法的精度非常高.

```

zipwin@WorldGate: ~/WorkPlace/fortran/computational_physics/assignment2/task0
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
→ task0 gfortran roots_seeker.f90 -o roots_seeker
→ task0 ./roots_seeker
method: downhill
The number of roots that is found: 3
root:          -1.7320508075688772
error:          0.000000001055960
iter:           8

root:           1.7320508075688772
error:          0.000000000893869
iter:           6

root:           -0.0000000000000000
error:          0.0000000000000299
iter:           6
→ task0 █

```

图 4: 牛顿下山法

4.4 事后加速法

主程序如下

Listing 8: 事后加速法主程序

```

1  program main
2      use utils
3      use roots_seeker
4
5      implicit none
6      real(8), external :: f, phi1, phi2
7      real(8), parameter :: step_size = 0.1d0, epsilon1 = 1e-5, epsilon2 = 1e-5
8      type(result), dimension(MAX_ROOTS_NUM) :: roots
9      integer, dimension(MAX_ROOTS_NUM) :: iters
10     real(8) :: left, right
11     character(50) :: file_name = "history.dat"
12     character(20) :: seeker = "post"
13     integer :: iter, root_num, i
14     logical :: stat
15
16     left = -10.0d0
17     right = 10.0d0
18
19     print *, "method: "//trim(seeker)
20     print *, "phi: (3*x)^(1/3)"
21     call multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1, epsilon2,
22         file_name, iters, f, phi1)
23     print "(a35, i3)", "The number of roots that is found: ", root_num
24     do i = 1, root_num
25         print "(a8, f32.16)", "root: ", roots(i)%value
26         print "(a8, f32.16)", "error: ", roots(i)%error
27         print "(a8, i32)", "iter: ", iters(i)
28         print *, " "
29     end do
30
31     print *, "phi: x^3/3"
32     call multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1, epsilon2,
33         file_name, iters, f, phi2)
34     print "(a35, i3)", "The number of roots that is found: ", root_num
35     do i = 1, root_num
36         print "(a8, f32.16)", "root: ", roots(i)%value
37         print "(a8, f32.16)", "error: ", roots(i)%error
38         print "(a8, i32)", "iter: ", iters(i)
39         print *, " "
40     end do
41 end program

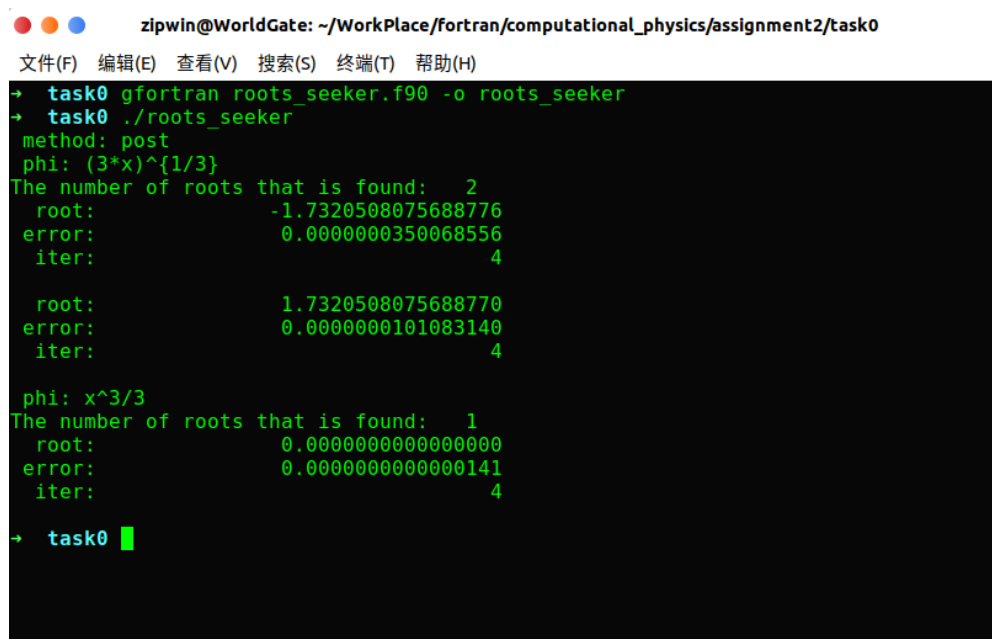
```

运行结果如图5所示. 结果为

$$x_1 = -1.732051(0)$$

$$x_2 = 0.000000(0)$$

$$x_3 = 1.732051(0)$$



```
zipwin@WorldGate: ~/WorkPlace/fortran/computational_physics/assignment2/task0
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
→ task0 gfortran roots_seeker.f90 -o roots_seeker
→ task0 ./roots_seeker
method: post
phi: (3*x)^(1/3)
The number of roots that is found: 2
root: -1.7320508075688776
error: 0.000000350068556
iter: 4

root: 1.7320508075688770
error: 0.000000101083140
iter: 4

phi: x^3/3
The number of roots that is found: 1
root: 0.0000000000000000
error: 0.000000000000141
iter: 4

→ task0
```

图 5: 事后加速法

4.5 Atiken 法

主程序如下

Listing 9: Atiken 法主程序

```
1 program main
2   use utils
3   use roots_seeker
4
5   implicit none
6   real(8), external :: f, phi1, phi2
7   real(8), parameter :: step_size = 0.1d0, epsilon1 = 1e-5, epsilon2 = 1e-5
8   type(result), dimension(MAX_ROOTS_NUM) :: roots
9   integer, dimension(MAX_ROOTS_NUM) :: iters
10  real(8) :: left, right
11  character(50) :: file_name = "history.dat"
```

```

12     character(20) :: seeker = "atiken"
13     integer :: iter, root_num, i
14     logical :: stat
15
16     left = -10.0d0
17     right = 10.0d0
18
19     print *, "method: "//trim(seeker)
20     print *, "phi: (3*x)^{1/3}"
21     call multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1, epsilon2,
22         file_name, iters, f, phi1)
23     print "(a35, i3)", "The number of roots that is found: ", root_num
24     do i = 1, root_num
25         print "(a8, f32.16)", "root: ", roots(i)%value
26         print "(a8, f32.16)", "error: ", roots(i)%error
27         print "(a8, i32)", "iter: ", iters(i)
28         print *, " "
29     end do
30
31     print *, "phi: x^3/3"
32     call multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1, epsilon2,
33         file_name, iters, f, phi2)
34     print "(a35, i3)", "The number of roots that is found: ", root_num
35     do i = 1, root_num
36         print "(a8, f32.16)", "root: ", roots(i)%value
37         print "(a8, f32.16)", "error: ", roots(i)%error
38         print "(a8, i32)", "iter: ", iters(i)
39         print *, " "
40     end do
41 end program

```

运行结果如图6所示. 结果为

$$x_1 = -1.732051(0)$$

$$x_2 = 0.000000(0)$$

$$x_3 = 1.732051(0)$$

5 各方法的比较

从上面的运行结果可以发现, 牛顿下山法的精度最高, 要求的精度为 10^{-5} , 而牛顿下山法达到了 10^{-11} 以上的精度. 事后加速法和 Atiken 法次之. 而二分法可以很好地控制精度. 至于

```

zipwin@WorldGate: ~/WorkPlace/fortran/computational_physics/assignment2/task0
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
→ task0 gfortran roots_seeker.f90 -o roots_seeker
→ task0 ./roots_seeker
method: atiken
phi: (3*x)^(1/3)
The number of roots that is found: 2
root: -1.7320508075688772
error: 0.0000000011577819
iter: 4

root: 1.7320508075688774
error: 0.0000000045921038
iter: 4

phi: x^3/3
The number of roots that is found: 1
root: -0.0000000000000000
error: 0.0000001748737858
iter: 3
→ task0 █

```

图 6: Atiken 法

收敛速度，从上面的结果也可以发现，二分法最慢，Jacobi 法稍快一些，牛顿下山法、事后加速法和 Atiken 法收敛速度较快。进一步比较收敛速度，让这几种方法都以 3.0 为初值（二分法为 [1.0, 4.0]），迭代 30 次，收敛曲线如图 7 所示。可见最快的算法是 Atiken 法，而二分法出现震荡，收敛最慢。

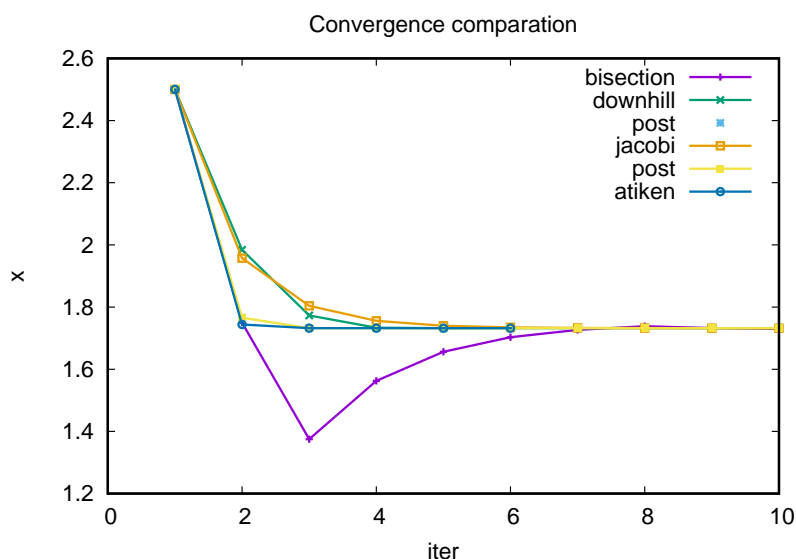


图 7: 各方法收敛速度的比较

附录

代码可在https://github.com/ZipWin/computational_physics/tree/master/assignments/assignment2找到.

Listing 10: roots__seeker.f90

```
1 module utils
2   implicit none
3   integer, parameter :: EVAL_NUM = 1000
4
5   type result
6     real(8) :: value
7     real(8) :: error
8   end type
9
10  contains
11  real(8) function numerical_derivative(func, x) result(dfdx)
12    !-----
13    ! This function computes the numerical derivative of func at x.
14    ! It returns the results in real(8).
15    !
16    ! Arguments:
17    !     func: a real(8) function
18    !     x: a real(8) number
19    !-----
20    implicit none
21    real(8), external :: func
22    real(8), intent(in) :: x
23
24    real(8) :: h
25    h = 1e-5
26
27    dfdx = (func(x+h) - func(x-h)) / (2*h)
28
29    return
30  end function
31
32  subroutine convergence_check(phi, a, b, stat)
33    !-----
34    ! This subroutine check whether the iteration  $x = \text{phi}(x)$  converges in  $[a, b]$ .
35    ! It return the results in stat. If converges, stat is true; if not, false.
36    !
37    ! Arguments:
38    !     phi: a real(8) function
```

```

39      !          a: real(8), the left boundary of the interval
40      !          b: real(8), the right boundary of the interval
41      !          stat: logical, save the state of convergence
42      !-----
43      implicit none
44      real(8), external :: phi
45      real(8), intent(in) :: a, b
46      logical, intent(out) :: stat
47
48      real(8), dimension(EVAL_NUM) :: xs, phis, dphis
49      real(8) :: eval_step
50      integer :: i
51
52      eval_step = abs(a - b) / EVAL_NUM
53
54      do i = 1, EVAL_NUM
55          xs(i) = a + eval_step * i
56          phis(i) = phi(xs(i))
57          dphis(i) = numerical_derivative(phi, xs(i))
58      end do
59
60      stat = .false.
61      if (maxval(phis) <= maxval((/a, b/)) .and. minval(phis) >= minval((/a, b/))) then
62          if (maxval(abs(dphis)) < 1) then
63              stat = .true.
64          end if
65      end if
66
67      return
68  end subroutine
69
70 end module
71
72
73 module roots_seeker
74     use utils
75
76     implicit none
77     integer, parameter :: MAX_ITER_NUM = 1000000
78     integer, parameter :: MAX_ROOTS_NUM = 100
79
80     contains
81     !-----
82     ! This following subroutines finds only one root in a given interval or initial point

```

```

83      !
84      ! Arguments:
85      !      f: real(8), the original function
86      !      phi: a real(8) function (not need for some methods)
87      !      left_init: the lower limit of the interval
88      !      right_init: the greater limit of the interval
89      !      x_init: real(8), the initial point
90      !      epsilon1: real(8), the expected error between x_n and x_{n-1}
91      !      epsilon2: real(8), the expected error between f(x_n) and 0
92      !      file_name: a string, the iteration history will be save into a file named as
      file_name
93      !      root: save the found root
94      !      error: save the error
95      !      iter: save the number of times of iteration
96      !      stat: if converges, true; else, false
97      !-----
98      subroutine bisection(f, left_init, right_init, epsilon1, epsilon2, file_name, root, iter,
      stat)
99      implicit none
100     real(8), external :: f
101     real(8), intent(in) :: left_init, right_init, epsilon1, epsilon2
102     character(50), intent(in) :: file_name
103     type(result), intent(out) :: root
104     integer, intent(out) :: iter
105     logical, intent(out) :: stat
106
107     real(8), dimension(MAX_ITER_NUM) :: history
108     real(8) :: left, middle, right, lv, mv, rv
109     integer :: i
110     logical :: stop_condition
111
112     left = left_init
113     right = right_init
114     middle = (left + right) / 2
115     i = 1
116     history(i) = middle
117     stat = .true.
118
119     stop_condition = .false.
120     do while (.not. stop_condition)
121         lv = f(left)
122         mv = f(middle)
123         rv = f(right)
124

```

```

125         if (abs(lv - 0.0d0) <= epsilon2) then
126             right = left + 2 * epsilon1
127         else if (abs(rv - 0.0d0) <= epsilon2) then
128             left = right - 2 * epsilon1
129         end if
130
131         if (lv * rv < 0) then
132             if (lv * mv < 0) then
133                 right = middle
134             else if (mv * rv < 0) then
135                 left = middle
136             end if
137         else
138             stat = .false.
139             exit
140         end if
141
142         middle = (left + right) / 2
143         i = i + 1
144         history(i) = middle
145
146         stop_condition = ((abs(right - left) <= epsilon1) &
147             .and. (abs(f(middle)) <= epsilon2)) &
148             .or. (i >= max_iter_num)
149
150     end do
151
152     root%value = middle
153     root%error = right - left
154     iter = i
155
156     if (i == MAX_ITER_NUM) then
157         stat = .false.
158     end if
159
160     open(file=file_name, unit=10, action="write")
161     do i = 1, iter
162         write (10, "(i8, a4, f8.6)") i, " ", history(i)
163     end do
164     close(unit=10)
165
166     return
167 end subroutine
168

```

```

169  subroutine jacobi(f, phi, x_init, epsilon1, epsilon2, file_name, root, iter, stat)
170      implicit none
171      real(8), external :: phi, f
172      real(8), intent(in) :: x_init, epsilon1, epsilon2
173      character(50), intent(in) :: file_name
174      type(result), intent(out) :: root
175      integer, intent(out) :: iter
176      logical, intent(out) :: stat
177
178      real(8), dimension(MAX_ITER_NUM) :: history
179      real(8) :: x
180      integer :: i
181      logical :: stop_condition
182
183      x = x_init
184      i = 1
185      history(i) = x
186
187      stop_condition = .false.
188      do while (.not. stop_condition)
189          x = phi(x)
190          i = i + 1
191          history(i) = x
192          stop_condition = ((x - history(i-1) <= epsilon1) &
193                          .and. (abs(f(x)) <= epsilon2)) &
194                          .or. (i >= max_iter_num)
195      end do
196
197      iter = i
198      root%value = history(i)
199      root%error = abs(history(i) - history(i-1))
200
201      if (i == max_iter_num) then
202          stat = .false.
203      else
204          stat = .true.
205      end if
206
207      open(file=file_name, unit=10, action="write")
208      do i = 1, iter
209          write (10, "(i8, a4, f8.6)") i, " ", history(i)
210      end do
211      close(unit=10)
212

```

```

213     return
214 end subroutine
215
216 subroutine newton_downhill(f, x_init, epsilon1, epsilon2, file_name, root, iter, stat)
217     implicit none
218     real(8), external :: f
219     real(8), intent(in) :: x_init, epsilon1, epsilon2
220     character(50), intent(in) :: file_name
221     type(result), intent(out) :: root
222     integer, intent(out) :: iter
223     logical, intent(out) :: stat
224
225     real(8), dimension(MAX_ITER_NUM) :: history
226     real(8) :: x, tmp, lambda
227     integer :: i
228     logical :: stop_condition
229
230     x = x_init
231     i = 1
232     history(i) = x
233
234     stop_condition = .false.
235     do while (.not. stop_condition)
236         lambda = 1.0d0 ! reset the value of lambda
237         do while (.true.)
238             tmp = x - lambda * f(x) / numerical_derivative(f, x)
239             if (abs(f(tmp)) < abs(f(x))) then
240                 exit
241             else
242                 lambda = lambda / 2
243             end if
244         end do
245         x = tmp
246         i = i + 1
247         history(i) = x
248         stop_condition = ((x - history(i-1)) <= epsilon1) &
249             .and. (abs(f(x)) <= epsilon2) &
250             .or. (i >= max_iter_num)
251     end do
252
253     iter = i
254     root%value = history(i)
255     root%error = abs(history(i) - history(i-1))
256

```

```

257     if (i == max_iter_num) then
258         stat = .false.
259     else
260         stat = .true.
261     end if
262
263     open(file=file_name, unit=10, action="write")
264     do i = 1, iter
265         write (10, "(i8, a4, f8.6)") i, " ", history(i)
266     end do
267     close(unit=10)
268
269     return
270 end subroutine
271
272 subroutine post_accelerating(f, phi, x_init, epsilon1, epsilon2, file_name, root, iter,
    stat)
273     implicit none
274     real(8), external :: f, phi
275     real(8), intent(in) :: x_init, epsilon1, epsilon2
276     character(50), intent(in) :: file_name
277     type(result), intent(out) :: root
278     integer, intent(out) :: iter
279     logical, intent(out) :: stat
280
281     real(8), dimension(MAX_ITER_NUM) :: history
282     real(8) :: x, L
283     integer :: i
284     logical :: stop_condition
285
286     x = x_init
287     i = 1
288     history(i) = x
289
290     stop_condition = .false.
291     do while (.not. stop_condition)
292         L = numerical_derivative(phi, x)
293         x = (phi(x) - L * x) / (1.0d0 - L)
294         i = i + 1
295         history(i) = x
296         stop_condition = ((x - history(i-1) <= epsilon1) &
297             .and. (abs(f(x)) <= epsilon2)) &
298             .or. (i >= max_iter_num)
299     end do

```

```

300
301     iter = i
302     root%value = history(i)
303     root%error = abs(history(i) - history(i-1))
304
305     if (i == max_iter_num) then
306         stat = .false.
307     else
308         stat = .true.
309     end if
310
311     open(file=file_name, unit=10, action="write")
312     do i = 1, iter
313         write (10, "(i8, a4, f8.6)") i, " ", history(i)
314     end do
315     close(unit=10)
316
317     return
318 end subroutine
319
320 subroutine atiken(f, phi, x_init, epsilon1, epsilon2, file_name, root, iter, stat)
321     implicit none
322     real(8), external :: f, phi
323     real(8), intent(in) :: x_init, epsilon1, epsilon2
324     character(50), intent(in) :: file_name
325     type(result), intent(out) :: root
326     integer, intent(out) :: iter
327     logical, intent(out) :: stat
328
329     real(8), dimension(MAX_ITER_NUM) :: history
330     real(8) :: x
331     integer :: i
332     logical :: stop_condition
333
334     x = x_init
335     i = 1
336     history(i) = x
337
338     stop_condition = .false.
339     do while (.not. stop_condition)
340         x = phi(phi(x)) - (phi(phi(x)) - phi(x))**2 / (phi(phi(x)) - 2*phi(x) + x)
341         i = i + 1
342         history(i) = x
343         stop_condition = ((x - history(i-1)) <= epsilon1) &

```



```

344         .and. (abs(f(x)) <= epsilon2)) &
345         .or. (i >= max_iter_num)
346     end do
347
348     iter = i
349     root%value = history(i)
350     root%error = abs(history(i) - history(i-1))
351
352     if (i == max_iter_num) then
353         stat = .false.
354     else
355         stat = .true.
356     end if
357
358     open(file=file_name, unit=10, action="write")
359     do i = 1, iter
360         write (10, "(i8, a4, f8.6)") i, " ", history(i)
361     end do
362     close(unit=10)
363
364     return
365 end subroutine
366
367 subroutine multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1,
368     epsilon2, file_name, iters, f, phi)
369 !-----
370 ! This subroutine search all the possible roots in [left, right].
371 ! It divides the interval into several adjoint small intervals, whose length is step_size
372 !
373 ! Arguments:
374 !     seeker: the roots solver to use, including bisection, Jacobi, Newton-downhill,
375 !             post acceleration, Aitken
376 !     f: real(8), the original function
377 !     phi: a real(8) function
378 !     left: real(8), the left boundary of the interval
379 !     right: real(8), the right boundary of the interval
380 !     step_size: real(8), the small interval length
381 !     epsilon1: real(8), the expected error between  $x_n$  and  $x_{n-1}$ 
382 !     epsilon2: real(8), the expected error between  $f(x_n)$  and 0
383 !     file_name: a string, the iteration history will be save into a file named as
384 !               file_name
385 !     root_num: the number of roots that has been found
386 !     roots: save the found roots

```

```

384      !           errors: save the errors
385      !           iters: save the numbers of times of iteration
386      !-----
387      implicit none
388      real(8), external :: f
389      real(8), external, optional :: phi
390      character(20), intent(in) :: seeker
391      real(8), intent(in) :: left, right, step_size, epsilon1, epsilon2
392      character(50), intent(in) :: file_name
393      integer, intent(out) :: root_num
394
395      integer, parameter :: MAX_ROOTS_NUM = 100
396      type(result), dimension(MAX_ROOTS_NUM) :: roots
397      integer, dimension(MAX_ROOTS_NUM) :: iters
398      type(result) :: root
399      real(8) :: a, b, x_init
400      integer :: iter, i, idx
401      logical :: can_converge, is_converge, is_found
402
403      if (trim(seeker) == "bisection") then
404          ! Bisection
405          i = 1
406          a = left
407          b = left + step_size
408          do while (b <= right)
409              call bisection(f, a, b, epsilon1, epsilon2, file_name, root, iter, is_converge)
410              if (is_converge) then
411                  is_found = .false.
412                  do idx = 1, i-1
413                      if (abs(root%value - roots(idx)%value) < 2*epsilon1) then
414                          is_found = .true.
415                          exit
416                      end if
417                  end do
418                  if (.not. is_found) then
419                      roots(i) = root
420                      iters(i) = iter
421                      i = i + 1
422                  else if (root%error <= roots(idx)%error) then
423                      roots(idx) = root
424                      iters(idx) = iter
425                  end if
426              end if
427              a = a + step_size

```

```

428         b = b + step_size
429     end do
430     root_num = i - 1
431 end if
432
433 if (trim(seeker) == "downhill") then
434     ! Newton downhill
435     i = 1
436     a = left
437     b = left + step_size
438     do while (b <= right)
439         call random_seed()
440         call random_number(x_init)
441         x_init = a + x_init * (a - b)
442         call newton_downhill(f, x_init, epsilon1, epsilon2, file_name, root, iter,
443             is_converge)
444         if (is_converge) then
445             is_found = .false.
446             do idx = 1, i-1
447                 if (abs(root%value - roots(idx)%value) < 2*epsilon1) then
448                     is_found = .true.
449                     exit
450                 end if
451             end do
452             if (.not. is_found) then
453                 roots(i) = root
454                 iters(i) = iter
455                 i = i + 1
456             else if (root%error <= roots(idx)%error) then
457                 roots(idx) = root
458                 iters(idx) = iter
459             end if
460         end if
461         a = a + step_size
462         b = b + step_size
463     end do
464     root_num = i - 1
465 end if
466
467 if (trim(seeker) == "jacobi" .or. trim(seeker) == "post" .or. trim(seeker) == "atiken"
468     ) then
469     i = 1
470     a = left
471     b = left + step_size

```

```

470     do while (b <= right)
471         call convergence_check(phi, a, b, can_converge)
472         if (can_converge) then
473             call random_seed()
474             call random_number(x_init)
475             x_init = a + x_init * (a - b)
476             if (trim(seeker) == "jacobi") then
477                 call jacobi(f, phi, x_init, epsilon1, epsilon2, file_name, root, iter,
478                     is_converge)
479             else if (trim(seeker) == "post") then
480                 call post_accelerating(f, phi, x_init, epsilon1, epsilon2, file_name,
481                     root, iter, is_converge)
482             else if (trim(seeker) == "atiken") then
483                 call atiken(f, phi, x_init, epsilon1, epsilon2, file_name, root, iter,
484                     is_converge)
485             end if
486             if (is_converge) then
487                 is_found = .false.
488                 do idx = 1, i-1
489                     if (abs(root%value - roots(idx)%value) < 2*epsilon1) then
490                         is_found = .true.
491                         exit
492                     end if
493                 end do
494                 if (.not. is_found) then
495                     roots(i) = root
496                     iters(i) = iter
497                     i = i + 1
498                 else if (root%error <= roots(idx)%error) then
499                     roots(idx) = root
500                     iters(idx) = iter
501                 end if
502             end if
503             a = a + step_size
504             b = b + step_size
505         end do
506         root_num = i - 1
507     end if
508     return
509 end subroutine
510 end module

```

```

511
512
513 program main
514     use utils
515     use roots_seeker
516
517     implicit none
518     real(8), external :: f, phi1, phi2
519     real(8), parameter :: step_size = 0.1d0, epsilon1 = 1e-5, epsilon2 = 1e-5
520     type(result), dimension(MAX_ROOTS_NUM) :: roots
521     integer, dimension(MAX_ROOTS_NUM) :: iters
522     real(8) :: left, right
523     character(50) :: file_name = "history.dat"
524     character(20) :: seeker = "post"
525     integer :: iter, root_num, i
526     logical :: stat
527
528     left = -10.0d0
529     right = 10.0d0
530
531     print *, "method: "//trim(seeker)
532     print *, "phi: (3*x)^{1/3}"
533     call multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1, epsilon2,
534         file_name, iters, f, phi1)
535     print "(a35, i3)", "The number of roots that is found: ", root_num
536     do i = 1, root_num
537         print "(a8, f32.16)", "root: ", roots(i)%value
538         print "(a8, f32.16)", "error: ", roots(i)%error
539         print "(a8, i32)", "iter: ", iters(i)
540         print *, " "
541     end do
542
543     print *, "phi: x^3/3"
544     call multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1, epsilon2,
545         file_name, iters, f, phi2)
546     print "(a35, i3)", "The number of roots that is found: ", root_num
547     do i = 1, root_num
548         print "(a8, f32.16)", "root: ", roots(i)%value
549         print "(a8, f32.16)", "error: ", roots(i)%error
550         print "(a8, i32)", "iter: ", iters(i)
551         print *, " "
552     end do
553 end program

```

```

553
554 !-----
555 ! This is the function for iteration x = phi(x)
556 ! To find different roots, different phi should be used
557 ! The form of function phi is chosen manually
558 !
559 ! Arguments:
560 !         x: a real(8) number
561 !-----
562 real(8) function f(x)
563     implicit none
564     real(8), intent(in) :: x
565
566     f = sign(abs(x)**3/3, x) - x
567
568     return
569 end function
570
571 real(8) function phi1(x) result(phi)
572     implicit none
573     real(8), intent(in) :: x
574
575     phi = sign(abs(3*x)**(1.0d0/3.0d0), x)
576
577     return
578 end function
579
580 real(8) function phi2(x) result(phi)
581     implicit none
582     real(8), intent(in) :: x
583
584     phi = x**3 / 3
585
586     return
587 end function

```