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

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**

```fortran
type result
    real(8) :: value
    real(8) :: error
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结果上.

Listing 2: **数值导数**

```fortran
real(8) function numerical_derivative(func, x) result(dfdx)
!------------------------------------------------------------------------------
! This function computes the numerical derivative of func at x.
! It returns the results in real(8).
!
! Argments:
!         func: a real(8) function
!         x: a real(8) number
!------------------------------------------------------------------------------
    implicit none
    real(8), external :: func
    real(8), intent(in) :: x

    real(8) :: h
    h = 1e-5

    dfdx = (func(x+h) - func(x-h)) / (2*h)

    return
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**

```fortran
subroutine convergence_check(phi, a, b, stat)
!-----------------------------------------------------------------------------
! This subroutine check whether the iteration x = phi(x) converges in [a, b].
! It return the results in stat. If converges, stat is true; if not, false.
!
! Argments:
!         phi: a real(8) function
!         a: real(8), the left boundary of the interval
!         b: real(8), the right boundary of the interval
!         stat: logical, save the state of convergence
!-----------------------------------------------------------------------------
   implicit none
   real(8), external :: phi
   real(8), intent(in) :: a, b
   logical, intent(out) :: stat

   real(8), dimension(EVAL_NUM) :: xs, phis, dphis
   real(8) :: eval_step
   integer :: i

   eval_step = abs(a - b) / EVAL_NUM

   do i = 1, EVAL_NUM
      xs(i) = a + eval_step * i
      phis(i) = phi(xs(i))
      dphis(i) = numerical_derivative(phi, xs(i))
   end do

   stat = .false.
   if (maxval(phis) <= maxval((/a, b/)) .and. minval(phis) >= minval((/a, b/))) then
      if (maxval(abs(dphis)) < 1) then
         stat = .true.
      end if
   end if

   return
```

```
end subroutine
```

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**

```
subroutine bisection(f, left_init, right_init, epsilon1, epsilon2, file_name, root, iter, stat)
    implicit none
    real(8), external :: f
    real(8), intent(in) :: left_init, right_init, epsilon1, epsilon2
    character(50), intent(in) :: file_name
    type(result), intent(out) :: root
    integer, intent(out) :: iter
    logical, intent(out) :: stat

    real(8), dimension(MAX_ITER_NUM) :: history
    real(8) :: left, middle, right, lv, mv, rv
    integer :: i
    logical :: stop_condition

    left = left_init
    right = right_init
    middle = (left + right) / 2
    i = 1
    history(i) = middle
    stat = .true.

    stop_condition = .false.
    do while (.not. stop_condition)
        lv = f(left)
        mv = f(middle)
        rv = f(right)

        if (abs(lv - 0.0d0) <= epsilon2) then
            right = left + 2 * epsilon1
        else if (abs(rv - 0.0d0) <= epsilon2) then
            left = right - 2 * epsilon1
        end if

        if (lv * rv < 0) then
```

```
            if (lv * mv < 0) then
                right = middle
            else if (mv * rv < 0) then
                left = middle
            end if
        else
            stat = .false.
            exit
        end if

        middle = (left + right) / 2
        i = i + 1
        history(i) = middle

        stop_condition = (abs(right - left) <= epsilon1) &
                       .and. (abs(f(middle)) <= epsilon2)      &
                       .and. (i <= max_iter_num)

    end do

    root%value = middle
    root%error = right - left
    iter = i

    if (i == MAX_ITER_NUM) then
        stat = .false.
    end if

    open(file=file_name, unit=10, action="write")
    do i = 1, iter
        write (10, "(i8, a4, f8.6)") i, " ", history(i)
    end do
    close(unit=10)

    return
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.

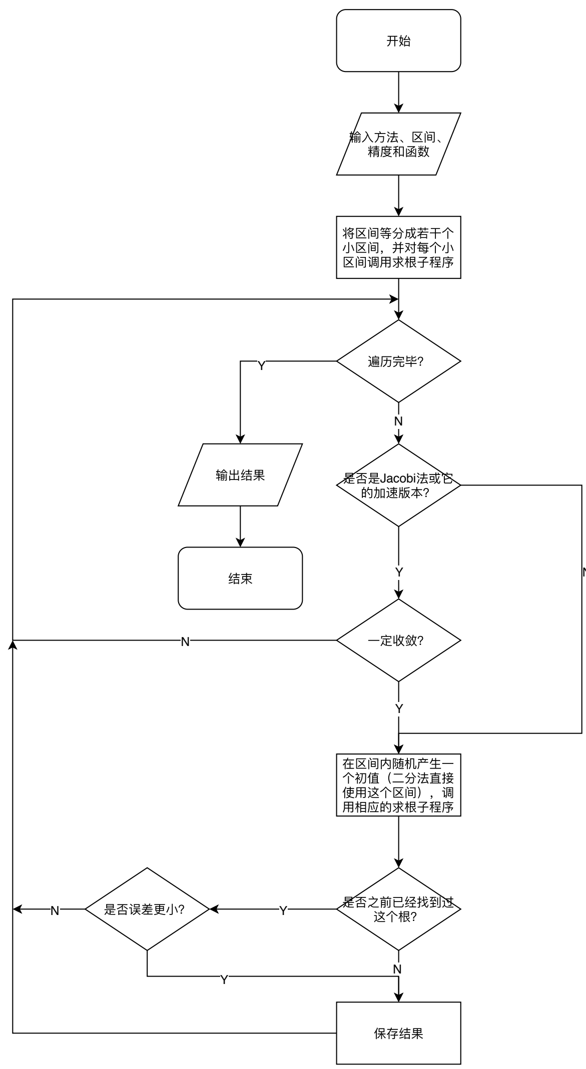

图 1: 接口子程序的流程图

4 运行时结果

下面各方法要求的精度为

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

输入的区间为 $[-10, 10]$.

4.1 二分法

主程序代码如下

Listing 5: 二分法主程序

```
program main
    use utils
    use roots_seeker

    implicit none
    real(8), external :: f, phi1, phi2
    real(8), parameter :: step_size = 0.1d0, epsilon1 = 1e-5, epsilon2 = 1e-5
    type(result), dimension(MAX_ROOTS_NUM) :: roots
    integer, dimension(MAX_ROOTS_NUM) :: iters
    real(8) :: left, right
    character(50) :: file_name = "history.dat"
    character(20) :: seeker = "bisection"
    integer :: iter, root_num, i
    logical :: stat

    left = -10.0d0
    right = 10.0d0

    print *, "method: "//trim(seeker)
    call multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1, epsilon2,
        file_name, iters, f)
    print "(a35, i3)", "The number of roots that is found: ", root_num
    do i = 1, root_num
        print "(a8, f32.16)", "root: ", roots(i)%value
        print "(a8, f32.16)", "error: ", roots(i)%error
        print "(a8, i32)", "iter: ", iters(i)
        print *, " "
    end do

end program
```

结果如图2所示. 求得的结果为（取到第 6 位小数）

$$
\begin{aligned}
x_1 &= -1.732053(6)\\
x_2 &= 0.000003(6)\\
x_3 &= 1.732053(6)
\end{aligned}
$$

括号内为最后一位误差. 可见与准确的结果符合.

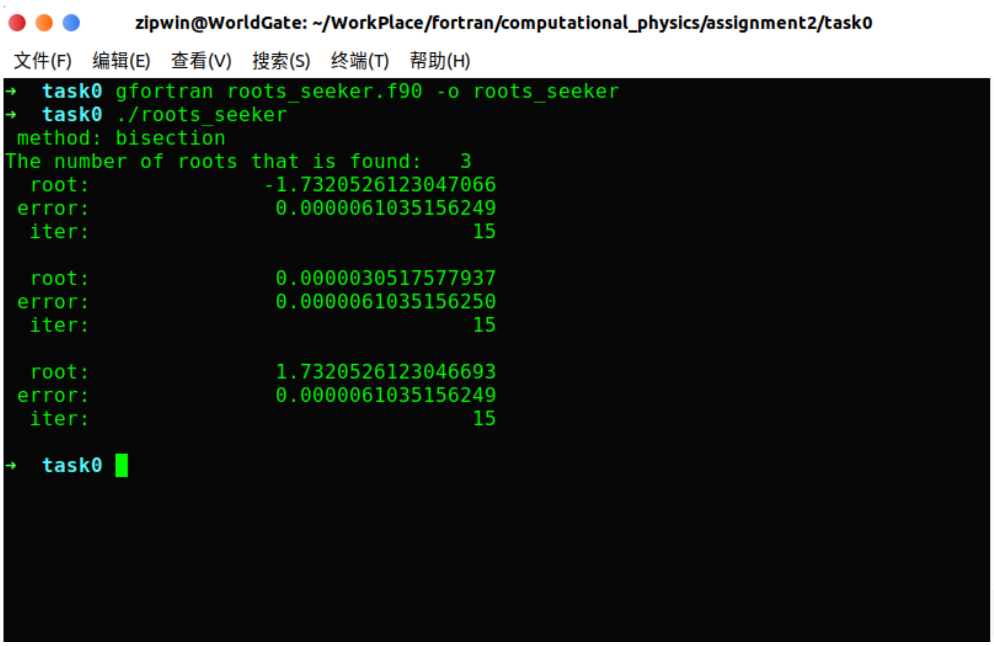

图 2: 二分法

4.2 Jacobi 迭代法

主程序如下

Listing 6: **Jacobi 迭代法主程序**

```fortran
program main
    use utils
    use roots_seeker

    implicit none
    real(8), external :: f, phi1, phi2
    real(8), parameter :: step_size = 0.1d0, epsilon1 = 1e-5, epsilon2 = 1e-5
    type(result), dimension(MAX_ROOTS_NUM) :: roots
    integer, dimension(MAX_ROOTS_NUM) :: iters
    real(8) :: left, right
    character(50) :: file_name = "history.dat"
```

```
    character(20) :: seeker = "jacobi"
    integer :: iter, root_num, i
    logical :: stat

    left = -10.0d0
    right = 10.0d0

    print *, "method: "//trim(seeker)
    print *, "phi: (3*x)^{1/3}"
    call multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1, epsilon2,
        file_name, iters, f, phi1)
    print "(a35, i3)", "The number of roots that is found: ", root_num
    do i = 1, root_num
        print "(a8, f32.16)", "root: ", roots(i)%value
        print "(a8, f32.16)", "error: ", roots(i)%error
        print "(a8, i32)", "iter: ", iters(i)
        print *, " "
    end do

    print *, "phi: x^3/3"
    call multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1, epsilon2,
        file_name, iters, f, phi2)
    print "(a35, i3)", "The number of roots that is found: ", root_num
    do i = 1, root_num
        print "(a8, f32.16)", "root: ", roots(i)%value
        print "(a8, f32.16)", "error: ", roots(i)%error
        print "(a8, i32)", "iter: ", iters(i)
        print *, " "
    end do

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.

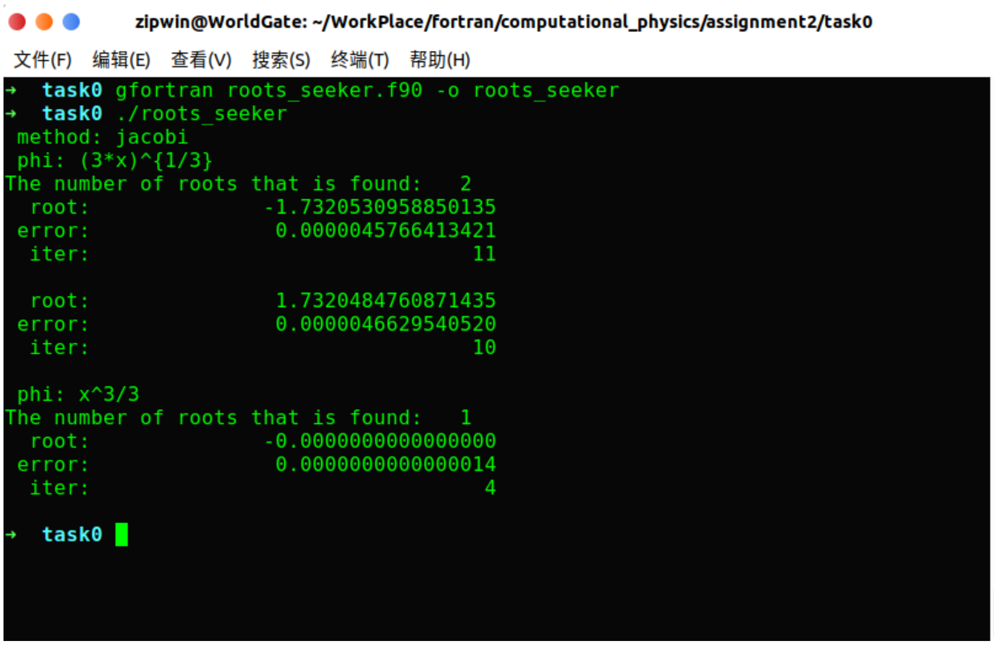

图 3: Jacobi 迭代法

4.3 牛顿下山法

主程序如下

Listing 7: **牛顿法主程序**

```fortran
program main
    use utils
    use roots_seeker

    implicit none
    real(8), external :: f, phi1, phi2
    real(8), parameter :: step_size = 0.1d0, epsilon1 = 1e-5, epsilon2 = 1e-5
    type(result), dimension(MAX_ROOTS_NUM) :: roots
    integer, dimension(MAX_ROOTS_NUM) :: iters
    real(8) :: left, right
    character(50) :: file_name = "history.dat"
    character(20) :: seeker = "downhill"
    integer :: iter, root_num, i
    logical :: stat

    left = -10.0d0
    right = 10.0d0

    print *, "method: "//trim(seeker)
```

```fortran
    call multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1, epsilon2,
        file_name, iters, f)
    print "(a35, i3)", "The number of roots that is found: ", root_num
    do i = 1, root_num
        print "(a8, f32.16)", "root: ", roots(i)%value
        print "(a8, f32.16)", "error: ", roots(i)%error
        print "(a8, i32)", "iter: ", iters(i)
        print *, " "
    end do

end program
```

运行结果如图4所示. 结果为

$$
\begin{aligned}
x_1 &= -1.732051(0) \\
x_2 &= 0.000000(0) \\
x_3 &= 1.732051(0)
\end{aligned}
$$

可见牛顿下山法的精度非常高.

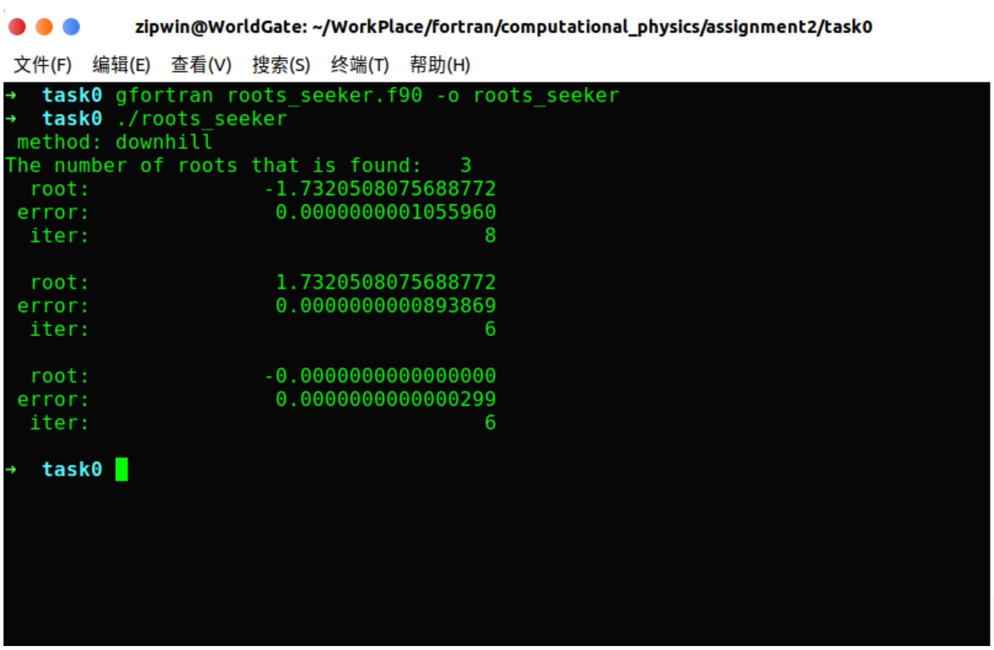

图 4: 牛顿下山法

4.4 事后加速法

主程序如下

Listing 8: 事后加速法主程序

```
program main
    use utils
    use roots_seeker

    implicit none
    real(8), external :: f, phi1, phi2
    real(8), parameter :: step_size = 0.1d0, epsilon1 = 1e-5, epsilon2 = 1e-5
    type(result), dimension(MAX_ROOTS_NUM) :: roots
    integer, dimension(MAX_ROOTS_NUM) :: iters
    real(8) :: left, right
    character(50) :: file_name = "history.dat"
    character(20) :: seeker = "post"
    integer :: iter, root_num, i
    logical :: stat

    left = -10.0d0
    right = 10.0d0

    print *, "method: "//trim(seeker)
    print *, "phi: (3*x)^{1/3}"
    call multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1, epsilon2,
        file_name, iters, f, phi1)
    print "(a35, i3)", "The number of roots that is found: ", root_num
    do i = 1, root_num
        print "(a8, f32.16)", "root: ", roots(i)%value
        print "(a8, f32.16)", "error: ", roots(i)%error
        print "(a8, i32)", "iter: ", iters(i)
        print *, " "
    end do

    print *, "phi: x^3/3"
    call multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1, epsilon2,
        file_name, iters, f, phi2)
    print "(a35, i3)", "The number of roots that is found: ", root_num
    do i = 1, root_num
        print "(a8, f32.16)", "root: ", roots(i)%value
        print "(a8, f32.16)", "error: ", roots(i)%error
        print "(a8, i32)", "iter: ", iters(i)
        print *, " "
    end do

end program
```

运行结果如图5所示. 结果为

$$x_1 = -1.732051(0)$$
$$x_2 = 0.000000(0)$$
$$x_3 = 1.732051(0)$$

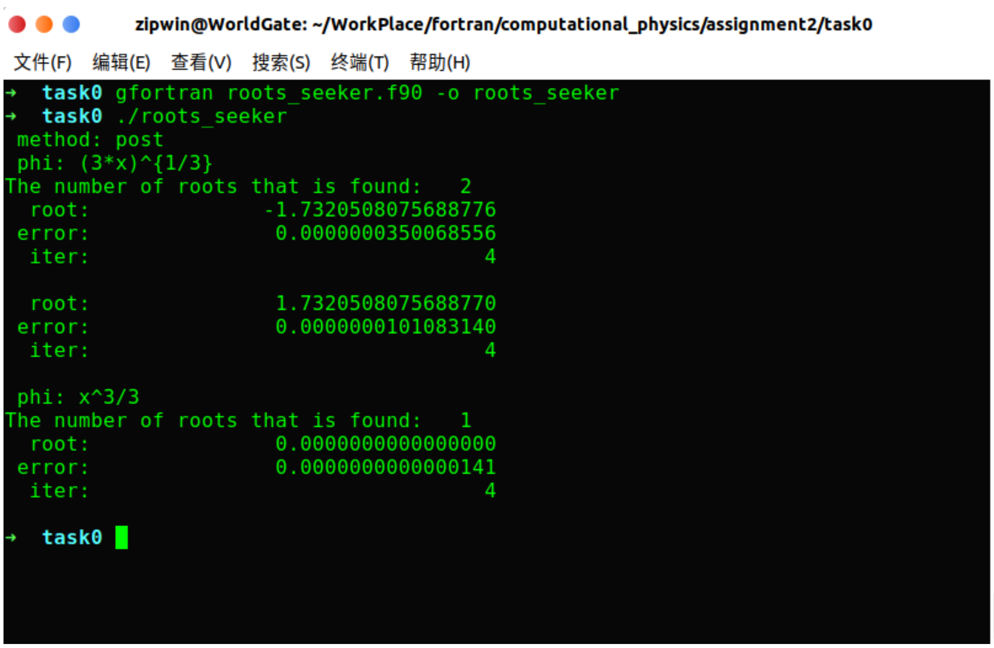

图 5: 事后加速法

4.5 Atiken 法

主程序如下

Listing 9: **Atiken 法主程序**

```
program main
    use utils
    use roots_seeker

    implicit none
    real(8), external :: f, phi1, phi2
    real(8), parameter :: step_size = 0.1d0, epsilon1 = 1e-5, epsilon2 = 1e-5
    type(result), dimension(MAX_ROOTS_NUM) :: roots
    integer, dimension(MAX_ROOTS_NUM) :: iters
    real(8) :: left, right
    character(50) :: file_name = "history.dat"
```

```fortran
    character(20) :: seeker = "atiken"
    integer :: iter, root_num, i
    logical :: stat

    left = -10.0d0
    right = 10.0d0

    print *, "method: "//trim(seeker)
    print *, "phi: (3*x)^{1/3}"
    call multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1, epsilon2,
        file_name, iters, f, phi1)
    print "(a35, i3)", "The number of roots that is found: ", root_num
    do i = 1, root_num
       print "(a8, f32.16)", "root: ", roots(i)%value
       print "(a8, f32.16)", "error: ", roots(i)%error
       print "(a8, i32)", "iter: ", iters(i)
       print *, " "
    end do

    print *, "phi: x^3/3"
    call multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1, epsilon2,
        file_name, iters, f, phi2)
    print "(a35, i3)", "The number of roots that is found: ", root_num
    do i = 1, root_num
       print "(a8, f32.16)", "root: ", roots(i)%value
       print "(a8, f32.16)", "error: ", roots(i)%error
       print "(a8, i32)", "iter: ", iters(i)
       print *, " "
    end do

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

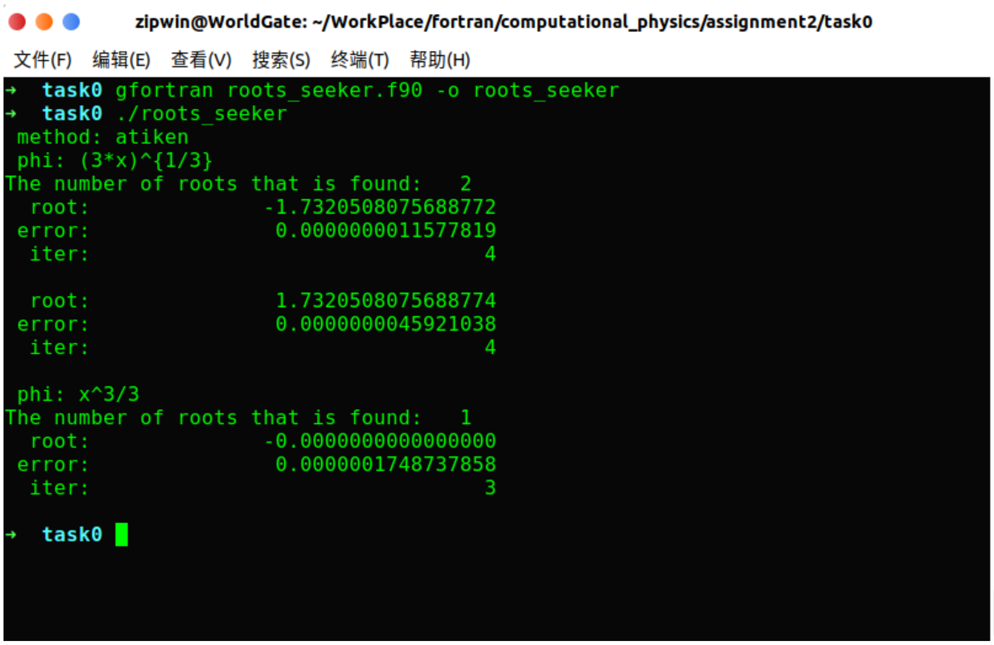

图 6: Atiken 法

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．

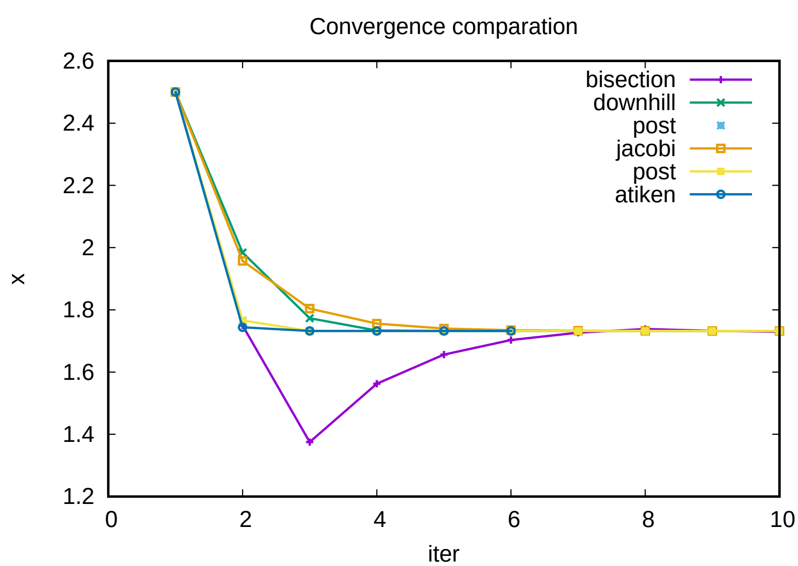

图 7: 各方法收敛速度的比较

附录

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

Listing 10: **roots_seeker.f90**

```
module utils
    implicit none
    integer, parameter :: EVAL_NUM = 1000

    type result
        real(8) :: value
        real(8) :: error
    end type

    contains
    real(8) function numerical_derivative(func, x) result(dfdx)
    !--------------------------------------------------------------------------------
    ! This function computes the numerical derivative of func at x.
    ! It returns the results in real(8).
    !
    ! Argments:
    !          func: a real(8) function
    !          x: a real(8) number
    !--------------------------------------------------------------------------------
        implicit none
        real(8), external :: func
        real(8), intent(in) :: x

        real(8) :: h
        h = 1e-5

        dfdx = (func(x+h) - func(x-h)) / (2*h)

        return
    end function

    subroutine convergence_check(phi, a, b, stat)
    !--------------------------------------------------------------------------------
    ! This subroutine check whether the iteration x = phi(x) converges in [a, b].
    ! It return the results in stat. If converges, stat is true; if not, false.
    !
    ! Argments:
    !          phi: a real(8) function
```

```
    !         a: real(8), the left boundary of the interval
    !         b: real(8), the right boundary of the interval
    !         stat: logical, save the state of convergence
    !-----------------------------------------------------------------------------
        implicit none
        real(8), external :: phi
        real(8), intent(in) :: a, b
        logical, intent(out) :: stat

        real(8), dimension(EVAL_NUM) :: xs, phis, dphis
        real(8) :: eval_step
        integer :: i

        eval_step = abs(a - b) / EVAL_NUM

        do i = 1, EVAL_NUM
            xs(i) = a + eval_step * i
            phis(i) = phi(xs(i))
            dphis(i) = numerical_derivative(phi, xs(i))
        end do

        stat = .false.
        if (maxval(phis) <= maxval((/a, b/)) .and. minval(phis) >= minval((/a, b/))) then
            if (maxval(abs(dphis)) < 1) then
                stat = .true.
            end if
        end if

        return
    end subroutine

end module


module roots_seeker
    use utils

    implicit none
    integer, parameter :: MAX_ITER_NUM = 1000000
    integer, parameter :: MAX_ROOTS_NUM = 100

    contains
    !-----------------------------------------------------------------------------
    ! This following subroutines finds only one root in a given interval or initial point
```

```fortran
    !
    ! Argments:
    !         f: real(8), the original function
    !         phi: a real(8) function (not need for some methods)
    !         left_init: the lower limit of the interval
    !         right_init: the greater limit of the interval
    !         x_init: real(8), the initial point
    !         epsilon1: real(8), the expected error between x_n and x_{n-1}
    !         epsilon2: real(8), the expected error between f(x_n) and 0
    !         file_name: a string, the iteration history will be save into a file named as file_name
    !         root: save the found root
    !         error: save the error
    !         iter: save the number of times of iteration
    !         stat: if converges, true; else, false
    !--------------------------------------------------------------------------------
    subroutine bisection(f, left_init, right_init, epsilon1, epsilon2, file_name, root, iter, stat)
        implicit none
        real(8), external :: f
        real(8), intent(in) :: left_init, right_init, epsilon1, epsilon2
        character(50), intent(in) :: file_name
        type(result), intent(out) :: root
        integer, intent(out) :: iter
        logical, intent(out) :: stat

        real(8), dimension(MAX_ITER_NUM) :: history
        real(8) :: left, middle, right, lv, mv, rv
        integer :: i
        logical :: stop_condition

        left = left_init
        right = right_init
        middle = (left + right) / 2
        i = 1
        history(i) = middle
        stat = .true.

        stop_condition = .false.
        do while (.not. stop_condition)
            lv = f(left)
            mv = f(middle)
            rv = f(right)

```

```
            if (abs(lv - 0.0d0) <= epsilon2) then
                right = left + 2 * epsilon1
            else if (abs(rv - 0.0d0) <= epsilon2) then
                left = right - 2 * epsilon1
            end if

            if (lv * rv < 0) then
                if (lv * mv < 0) then
                    right = middle
                else if (mv * rv < 0) then
                    left = middle
                end if
            else
                stat = .false.
                exit
            end if

            middle = (left + right) / 2
            i = i + 1
            history(i) = middle

            stop_condition = ((abs(right - left) <= epsilon1) &
                             .and. (abs(f(middle)) <= epsilon2)) &
                             .or. (i >= max_iter_num)

        end do

        root%value = middle
        root%error = right - left
        iter = i

        if (i == MAX_ITER_NUM) then
            stat = .false.
        end if

        open(file=file_name, unit=10, action="write")
        do i = 1, iter
            write (10, "(i8, a4, f8.6)") i, " ", history(i)
        end do
        close(unit=10)

        return
    end subroutine

```

```
    subroutine jacobi(f, phi, x_init, epsilon1, epsilon2, file_name, root, iter, stat)
        implicit none
        real(8), external :: phi, f
        real(8), intent(in) :: x_init, epsilon1, epsilon2
        character(50), intent(in) :: file_name
        type(result), intent(out) :: root
        integer, intent(out) :: iter
        logical, intent(out) :: stat

        real(8), dimension(MAX_ITER_NUM) :: history
        real(8) :: x
        integer :: i
        logical :: stop_condition

        x = x_init
        i = 1
        history(i) = x

        stop_condition = .false.
        do while (.not. stop_condition)
            x = phi(x)
            i = i + 1
            history(i) = x
            stop_condition = ((x - history(i-1) <= epsilon1)  &
                                .and. (abs(f(x)) <= epsilon2)) &
                                .or. (i >= max_iter_num)
        end do

        iter = i
        root%value = history(i)
        root%error = abs(history(i) - history(i-1))

        if (i == max_iter_num) then
            stat = .false.
        else
            stat = .true.
        end if

        open(file=file_name, unit=10, action="write")
        do i = 1, iter
            write (10, "(i8, a4, f8.6)") i, " ", history(i)
        end do
        close(unit=10)

```

```fortran
        return
    end subroutine

    subroutine newton_downhill(f, x_init, epsilon1, epsilon2, file_name, root, iter, stat)
        implicit none
        real(8), external :: f
        real(8), intent(in) :: x_init, epsilon1, epsilon2
        character(50), intent(in) :: file_name
        type(result), intent(out) :: root
        integer, intent(out) :: iter
        logical, intent(out) :: stat

        real(8), dimension(MAX_ITER_NUM) :: history
        real(8) :: x, tmp, lambda
        integer :: i
        logical :: stop_condition

        x = x_init
        i = 1
        history(i) = x

        stop_condition = .false.
        do while (.not. stop_condition)
            lambda = 1.0d0 ! reset the value of lambda
            do while (.true.)
                tmp = x - lambda * f(x) / numerical_derivative(f, x)
                if (abs(f(tmp)) < abs(f(x))) then
                    exit
                else
                    lambda = lambda / 2
                end if
            end do
            x = tmp
            i = i + 1
            history(i) = x
            stop_condition = ((x - history(i-1) <= epsilon1)  &
                             .and. (abs(f(x)) <= epsilon2)) &
                             .or. (i >= max_iter_num)
        end do

        iter = i
        root%value = history(i)
        root%error = abs(history(i) - history(i-1))

```

```fortran
        if (i == max_iter_num) then
            stat = .false.
        else
            stat = .true.
        end if

        open(file=file_name, unit=10, action="write")
        do i = 1, iter
            write (10, "(i8, a4, f8.6)") i, " ", history(i)
        end do
        close(unit=10)

        return
    end subroutine

    subroutine post_accelerating(f, phi, x_init, epsilon1, epsilon2, file_name, root, iter, stat)
        implicit none
        real(8), external :: f, phi
        real(8), intent(in) :: x_init, epsilon1, epsilon2
        character(50), intent(in) :: file_name
        type(result), intent(out) :: root
        integer, intent(out) :: iter
        logical, intent(out) :: stat

        real(8), dimension(MAX_ITER_NUM) :: history
        real(8) :: x, L
        integer :: i
        logical :: stop_condition

        x = x_init
        i = 1
        history(i) = x

        stop_condition = .false.
        do while (.not. stop_condition)
            L = numerical_derivative(phi, x)
            x = (phi(x) - L * x) / (1.0d0 - L)
            i = i + 1
            history(i) = x
            stop_condition = ((x - history(i-1) <= epsilon1)  &
                            .and. (abs(f(x)) <= epsilon2)) &
                            .or. (i >= max_iter_num)
        end do
```

```

        iter = i
        root%value = history(i)
        root%error = abs(history(i) - history(i-1))

        if (i == max_iter_num) then
            stat = .false.
        else
            stat = .true.
        end if

        open(file=file_name, unit=10, action="write")
        do i = 1, iter
            write (10, "(i8, a4, f8.6)") i, " ", history(i)
        end do
        close(unit=10)

        return
    end subroutine

    subroutine atiken(f, phi, x_init, epsilon1, epsilon2, file_name, root, iter, stat)
        implicit none
        real(8), external :: f, phi
        real(8), intent(in) :: x_init, epsilon1, epsilon2
        character(50), intent(in) :: file_name
        type(result), intent(out) :: root
        integer, intent(out) :: iter
        logical, intent(out) :: stat

        real(8), dimension(MAX_ITER_NUM) :: history
        real(8) :: x
        integer :: i
        logical :: stop_condition

        x = x_init
        i = 1
        history(i) = x

        stop_condition = .false.
        do while (.not. stop_condition)
            x = phi(phi(x)) - (phi(phi(x)) - phi(x))**2 / (phi(phi(x)) - 2*phi(x) + x)
            i = i + 1
            history(i) = x
            stop_condition = ((x - history(i-1) <= epsilon1)  &
```

```fortran
                         .and. (abs(f(x)) <= epsilon2)) &
                         .or. (i >= max_iter_num)
        end do

        iter = i
        root%value = history(i)
        root%error = abs(history(i) - history(i-1))

        if (i == max_iter_num) then
            stat = .false.
        else
            stat = .true.
        end if

        open(file=file_name, unit=10, action="write")
        do i = 1, iter
            write (10, "(i8, a4, f8.6)") i, " ", history(i)
        end do
        close(unit=10)

        return
    end subroutine

    subroutine multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1,
        epsilon2, file_name, iters, f, phi)
    !------------------------------------------------------------------------------
    ! This subroutine search all the possible roots in [left, right].
    ! It divides the interval into several adjoint small intervals, whose length is step_size
        .
    !
    ! Argments:
    !         seeker: the roots solver to use, including bisection, Jacobi, Newton-downhill,
        post acceleration, Aitken
    !         f: real(8), the original function
    !         phi: a real(8) function
    !         left: real(8), the left boundary of the interval
    !         right: real(8), the right boundary of the interval
    !         step_size: real(8), the small interval length
    !         epsilon1: real(8), the expected error between x_n and x_{n-1}
    !         epsilon2: real(8), the expected error between f(x_n) and 0
    !         file_name: a string, the iteration history will be save into a file named as
        file_name
    !         root_num: the number of roots that has been found
    !         roots: save the found roots
```

```fortran
    !         errors: save the errors
    !         iters: save the numbers of times of iteration
    !-----------------------------------------------------------------------------
        implicit none
        real(8), external :: f
        real(8), external, optional :: phi
        character(20), intent(in) :: seeker
        real(8), intent(in) :: left, right, step_size, epsilon1, epsilon2
        character(50), intent(in) :: file_name
        integer, intent(out) :: root_num

        integer, parameter :: MAX_ROOTS_NUM = 100
        type(result), dimension(MAX_ROOTS_NUM) :: roots
        integer, dimension(MAX_ROOTS_NUM) :: iters
        type(result) :: root
        real(8) :: a, b, x_init
        integer :: iter, i, idx
        logical :: can_converge, is_converge, is_found

        if (trim(seeker) == "bisection") then
            ! Bisection
            i = 1
            a = left
            b = left + step_size
            do while (b <= right)
                call bisection(f, a, b, epsilon1, epsilon2, file_name, root, iter, is_converge)
                if (is_converge) then
                    is_found = .false.
                    do idx = 1, i-1
                        if (abs(root%value - roots(idx)%value) < 2*epsilon1) then
                            is_found = .true.
                            exit
                        end if
                    end do
                    if (.not. is_found) then
                        roots(i) = root
                        iters(i) = iter
                        i = i + 1
                    else if (root%error <= roots(idx)%error) then
                        roots(idx) = root
                        iters(idx) = iter
                    end if
                end if
                a = a + step_size
```

```
            b = b + step_size
        end do
        root_num = i - 1
    end if

    if (trim(seeker) == "downhill") then
        ! Newton downhill
        i = 1
        a = left
        b = left + step_size
        do while (b <= right)
            call random_seed()
            call random_number(x_init)
            x_init = a + x_init * (a - b)
            call newton_downhill(f, x_init, epsilon1, epsilon2, file_name, root, iter, is_converge)
            if (is_converge) then
                is_found = .false.
                do idx = 1, i-1
                    if (abs(root%value - roots(idx)%value) < 2*epsilon1) then
                        is_found = .true.
                        exit
                    end if
                end do
                if (.not. is_found) then
                    roots(i) = root
                    iters(i) = iter
                    i = i + 1
                else if (root%error <= roots(idx)%error) then
                    roots(idx) = root
                    iters(idx) = iter
                end if
            end if
            a = a + step_size
            b = b + step_size
        end do
        root_num = i - 1
    end if

    if (trim(seeker) == "jacobi" .or. trim(seeker) == "post" .or. trim(seeker) == "atiken" ) then
        i = 1
        a = left
        b = left + step_size
```

```
            do while (b <= right)
                call convergence_check(phi, a, b, can_converge)
                if (can_converge) then
                    call random_seed()
                    call random_number(x_init)
                    x_init = a + x_init * (a - b)
                    if (trim(seeker) == "jacobi") then
                        call jacobi(f, phi, x_init, epsilon1, epsilon2, file_name, root, iter, is_converge)
                    else if (trim(seeker) == "post") then
                        call post_accelerating(f, phi, x_init, epsilon1, epsilon2, file_name, root, iter, is_converge)
                    else if (trim(seeker) == "atiken") then
                        call atiken(f, phi, x_init, epsilon1, epsilon2, file_name, root, iter, is_converge)
                    end if
                    if (is_converge) then
                        is_found = .false.
                        do idx = 1, i-1
                            if (abs(root%value - roots(idx)%value) < 2*epsilon1) then
                                is_found = .true.
                                exit
                            end if
                        end do
                        if (.not. is_found) then
                            roots(i) = root
                            iters(i) = iter
                            i = i + 1
                        else if (root%error <= roots(idx)%error) then
                            roots(idx) = root
                            iters(idx) = iter
                        end if
                    end if
                end if
                a = a + step_size
                b = b + step_size
            end do
            root_num = i - 1
        end if

        return
    end subroutine

end module
```

```fortran


program main
    use utils
    use roots_seeker

    implicit none
    real(8), external :: f, phi1, phi2
    real(8), parameter :: step_size = 0.1d0, epsilon1 = 1e-5, epsilon2 = 1e-5
    type(result), dimension(MAX_ROOTS_NUM) :: roots
    integer, dimension(MAX_ROOTS_NUM) :: iters
    real(8) :: left, right
    character(50) :: file_name = "history.dat"
    character(20) :: seeker = "post"
    integer :: iter, root_num, i
    logical :: stat

    left = -10.0d0
    right = 10.0d0

    print *, "method: "//trim(seeker)
    print *, "phi: (3*x)^{1/3}"
    call multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1, epsilon2,
        file_name, iters, f, phi1)
    print "(a35, i3)", "The number of roots that is found: ", root_num
    do i = 1, root_num
        print "(a8, f32.16)", "root: ", roots(i)%value
        print "(a8, f32.16)", "error: ", roots(i)%error
        print "(a8, i32)", "iter: ", iters(i)
        print *, " "
    end do

    print *, "phi: x^3/3"
    call multi_seeker(root_num, roots, seeker, left, right, step_size, epsilon1, epsilon2,
        file_name, iters, f, phi2)
    print "(a35, i3)", "The number of roots that is found: ", root_num
    do i = 1, root_num
        print "(a8, f32.16)", "root: ", roots(i)%value
        print "(a8, f32.16)", "error: ", roots(i)%error
        print "(a8, i32)", "iter: ", iters(i)
        print *, " "
    end do

end program
```

```fortran

!------------------------------------------------------------------------------
! This is the function for iteration x = phi(x)
! To find different roots, different phi should be used
! The form of function phi is chosen manually
!
! Argments:
!          x: a real(8) number
!------------------------------------------------------------------------------
real(8) function f(x)
    implicit none
    real(8), intent(in) :: x

    f = sign(abs(x)**3/3, x) - x

    return
end function

real(8) function phi1(x) result(phi)
    implicit none
    real(8), intent(in) :: x

    phi = sign(abs(3*x)**(1.0d0/3.0d0), x)

    return
end function

real(8) function phi2(x) result(phi)
    implicit none
    real(8), intent(in) :: x

    phi = x**3 / 3

    return
end function
```