Interpolation

报告主要内容

> **Use Lagrange, Newton and Cubic Spline interpolation method to rebuild the function**
>
> $$f(x) = \frac{1}{1+x^2}, x \in [-5, 5]$$
>
> **based on points (N=15):**
>
> $$x_i = -5 + \frac{10}{N} i, i = 0, 1, \ldots N$$

主程序

原函数:

```
function f(x)
    implicit none
    real :: x,f
    f=1.0/(1.0+x*x)
end function f
```

将要用到的方法封装到插值模块中，在主函数里调用:

```
program main
    use interpolation ! 使用差值模块
    implicit none

    ! 差值的原函数
    interface
        function f(x)
            real :: x,f
```

```
        end function f
    end interface

    integer :: i,N=15
    real :: x0(16),y0(16),x(101),y(101),r(101)

    ! init x0 & y0
    x0=(/ (-5+10.0*i/N,i=0,N) /)
    y0=(/ (f(x0(i)),i=1,N+1) /)
    x=(/ (-5+10.0*i/100,i=0,100) /)
    y=0
    r=0

    print *
    call init(x0,y0)
    print *,'Set initail x='
    print '(f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5)',x0
    print *
    print *,'y='
    print '(f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5)',y0
    print *
    print *,'Lagrange interpolation'
    call lagrange_interpolation(x,y)
    call lagrange_interpolation_error_analysis(x,r)
    print '(5x,a9,5x,a9,5x,a9)','x','y','r'
    print '(5x,f9.5,5x,f9.5,5x,f9.5)',
(x(i),y(i),r(i),i=1,size(x),10)
    print *
    print *,'Write the data to file: lagrange_data.txt'
    open(101,file="lagrange_data.txt")
    write(101,'(f9.5,f9.5,f9.5)') (x(i),y(i),r(i),i=1,size(x))
    close(101)
    print *
    print *,'Newton interpolation'
    call newton_interpolation(x,y)
    print '(5x,a9,5x,a9)','x','y'
    print '(5x,f9.5,5x,f9.5)',(x(i),y(i),i=1,size(x),10)
    print *
    print *,'Write the data to file: newton_data.txt'
    print *
    print *,'Cubic spline interpolation'
    call cubic_spline_curve(x,y)
    print *
    print '(5x,a9,5x,a9)','x','y'
    print '(5x,f9.5,5x,f9.5)',(x(i),y(i),i=1,size(x),10)
    print *
    print *,'Write the data to file: cubic_data.txt'
    open(101,file="cubic_data.txt")
```

```
    write(101,'(f9.5,f9.5)') (x(i),y(i),i=1,size(x))
    close(101)
    print *

end program main
```

整个模块的代码如下:

```
module interpolation
    implicit none
    private
    public ::
init,lagrange_interpolation,newton_interpolation,lagrange_interp
olation_error_analysis,cubic_spline_curve

    real,allocatable :: x_(:),y_(:)
    integer :: N

    contains

    ! 初始化
    subroutine init(x,y)
        implicit none
        real :: x(:),y(:)
        N=size(x)
        allocate (x_(N),y_(N))
        x_=x
        y_=y
    end subroutine init

    ! 计算Li(x)的值
    function L(i,x,s,e)
        real :: L,x
        integer :: i,j,s,e
        L=1
        do j=s,e
            if (i /= j) L=L*(x-x_(j))/(x_(i)-x_(j))
        end do
    end function L

    function Ln(x,s,e)
        real :: Ln,x
        integer :: i,s,e
        Ln=0.0
        do i=s,e
            Ln=Ln+L(i,x,s,e)*y_(i)
```

```
        end do
    end function Ln

    ! 拉格朗日插值法
    subroutine lagrange_interpolation(x,y)
        implicit none
        real :: x(:),y(:)
        integer :: i
        y=(/ (Ln(x(i),1,N),i=1,size(x)) /)
    end subroutine lagrange_interpolation

    ! 误差分析
    subroutine lagrange_interpolation_error_analysis(x,r)
        implicit none
        real :: x(:),r(:),t
        integer :: m,i,j
        m=size(x)
        do i=1,m
            r(i)=(x(i)-x_(1))/(x_(1)-x_(N))*(Ln(x(i),1,N-1)-
Ln(x(i),2,N))
        end do
    end subroutine lagrange_interpolation_error_analysis

    ! Difference quotient
    recursive function f_(a,b) result(r)
        implicit none
        real :: r
        integer,intent(in) :: a,b

        select case (abs(b-a))
            case (0)
                r=y_(a)
            case (1)
                r=(y_(b)-y_(a))/(x_(b)-x_(a))
            case default
                r=(f_(a+1,b)-f_(a,b-1))/(x_(b)-x_(a))
        end select
    end function f_

    ! Nn(x)
    function Nn(x,a)
        implicit none
        real :: Nn,x,a(:),t
        integer :: i,j
        do i=1,N
            t=1.0
            do j=1,i-1
                t=t*(x-x_(j))
```

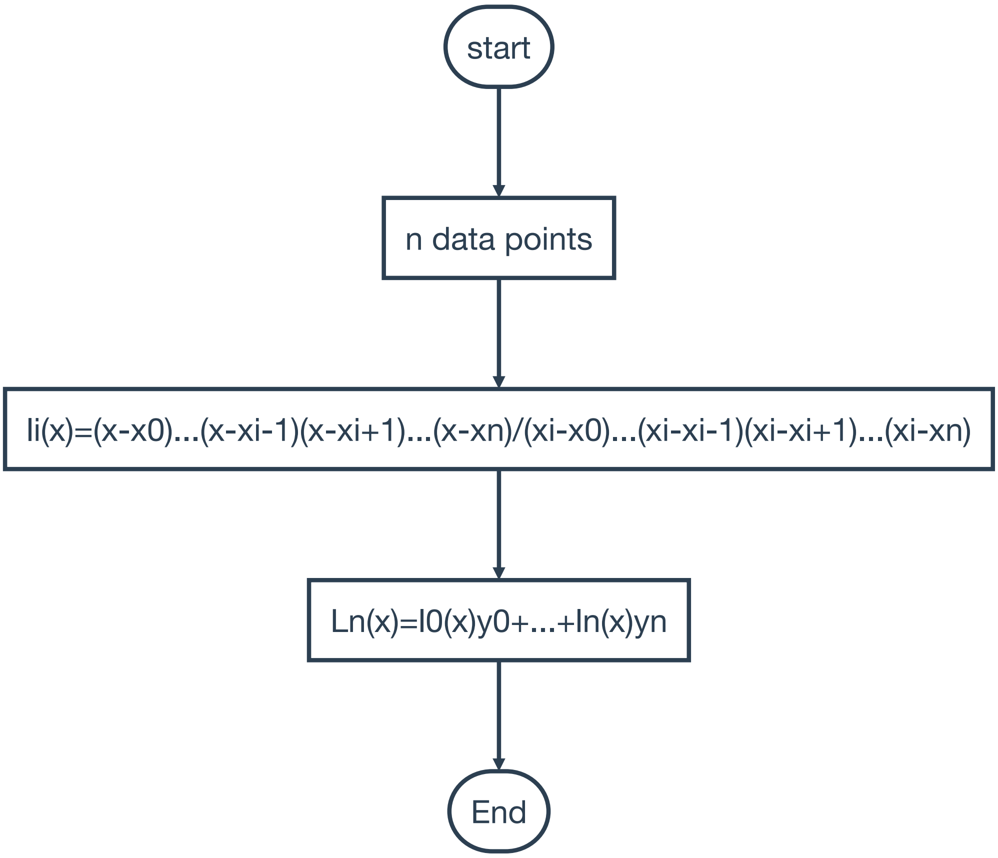
```
            end do
            Nn=Nn+a(i)*t
        end do
    end function

    ! 牛顿插值法
    subroutine newton_interpolation(x,y)
        implicit none
        real :: x(:),y(:),a(N)
        integer :: i
        a=(/ (f_(1,i),i=1,N) /)
        print *,'a='
        print '(f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5)',a
        print *

        open(101,file="newton_data.txt")
        do i=1,101
            ! There is a bug if you comment the following code,
I don't know why this would happen.
            write(101,'(f9.5,f9.5)') x(i),Nn(x(i),a)
            y(i)=Nn(x(i),a)
        end do
        close(101)

    end subroutine newton_interpolation

    ! 追赶法
    subroutine chasing(matrix,x)
        implicit none
        real :: matrix(:,:),x(:)
        real,allocatable :: u(:),q(:)
        integer :: i,m,n
        m=size(matrix(1,:))
        n=size(matrix(:,1))

        allocate(u(n-1),q(n))
        u(1)=matrix(1,2)/matrix(1,1)
        q(1)=matrix(1,m)/matrix(1,1)

        do i=2,n-1
            u(i)=matrix(i,i+1)/(matrix(i,i)-u(i-1)*matrix(i,i-
1))
        end do

        do i=2,n
            q(i)=(matrix(i,m)-q(i-1)*matrix(i,i-
1))/(matrix(i,i)-u(i-1)*matrix(i,i-1))
        end do
```

```
        x(n)=q(n)
        do i=n-1,1,-1
            x(i)=q(i)-u(i)*x(i+1)
        end do

        deallocate(u,q)
    end subroutine chasing

    ! Cubic Spline Curve
    subroutine cubic_spline_curve(x,y)
        implicit none
        real :: h(N-1),mu(N-2),lambda(N-2),d(N-2),M(N),mat(N-
2,N-1),x(:),y(:)
        integer :: i,j

        h=(/ (x_(i)-x_(i-1),i=2,N) /)
        mu=(/ (h(i)/(h(i)+h(i+1)),i=1,N-2) /)
        lambda=(/ (1-mu(i),i=1,N-2) /)
        d=(/ (6*f_(i-1,i+1),i=2,N-1) /)
        M=0

        mat=0
        do i=1,N-2
            if (i /= 1) mat(i,i-1)=mu(i)
            mat(i,i)=2
            if (i /= N-2) mat(i,i+1)=lambda(i)
            mat(i,N-1)=d(i)
        end do
        call chasing(mat,M(2:N-1))

        y=0
        do i=1,size(x)
            do j=2,size(x_)
                if ( x(i)<=x_(j) ) exit
            end do
            y(i)=M(j-1)*(x_(j)-x(i))/(6*h(j-1))
            y(i)=y(i)+M(j)*(x(i)-x_(j-1))/(6*h(j-1))
            y(i)=y(i)+(y_(j-1)-M(j-1)*h(j-1)*h(j-1)/6)*(x_(j)-
x(i))/h(j-1)
            y(i)=y(i)+(y_(j)-M(j)*h(j-1)*h(j-1)/6)*(x(i)-x_(j-
1))/h(j-1)
        end do

        ! 输出一些基本信息到屏幕上
        print *
        print *,'h='
        print '(f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5)',h
```

```
        print *
        print *,'mu='
        print '(f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5)',mu
        print *
        print *,'lambda='
        print '(f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5)',lambda
        print *
        print *,'d='
        print '(f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5)',d
        print *
        print *,'Solve the matrix to get M'
        print 
'(f6.2,f6.2,f6.2,f6.2,f6.2,f6.2,f6.2,f6.2,f6.2,f6.2,f6.2,f6.2,f6
.2,f6.2,f6.2)',(mat(i,:),i=1,N-2)
        print *
        print *,'M='
        print '(f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5)',M
        print *

    end subroutine cubic_spline_curve

end module interpolation
```

拉格朗日插值法

流程图:



原理:

$$l_i(x) = \frac{(x-x_0)...(x-x_{i-1})(x-x_{i+1})..(x-x_n)}{(x_i-x_0)...(x_i-x_{i-1})(x_i-x_{i+1})...(x_i-x_n)}$$

$$L_n(x) = \sum_{i=1}^{N} l_i(x) y_i$$

代码:

```fortran
    ! 计算Li(x)的值
    function L(i,x,s,e)
        real :: L,x
        integer :: i,j,s,e
        L=1
        do j=s,e
            if (i /= j) L=L*(x-x_(j))/(x_(i)-x_(j))
        end do
    end function L

    function Ln(x,s,e)
        real :: Ln,x
        integer :: i,s,e
        Ln=0.0
        do i=s,e
            Ln=Ln+L(i,x,s,e)*y_(i)
        end do
    end function Ln

    ! 拉格朗日插值法
    subroutine lagrange_interpolation(x,y)
        implicit none
        real :: x(:),y(:)
        integer :: i
        y=(/ (Ln(x(i),1,N),i=1,size(x)) /)
    end subroutine lagrange_interpolation
```

误差分析:

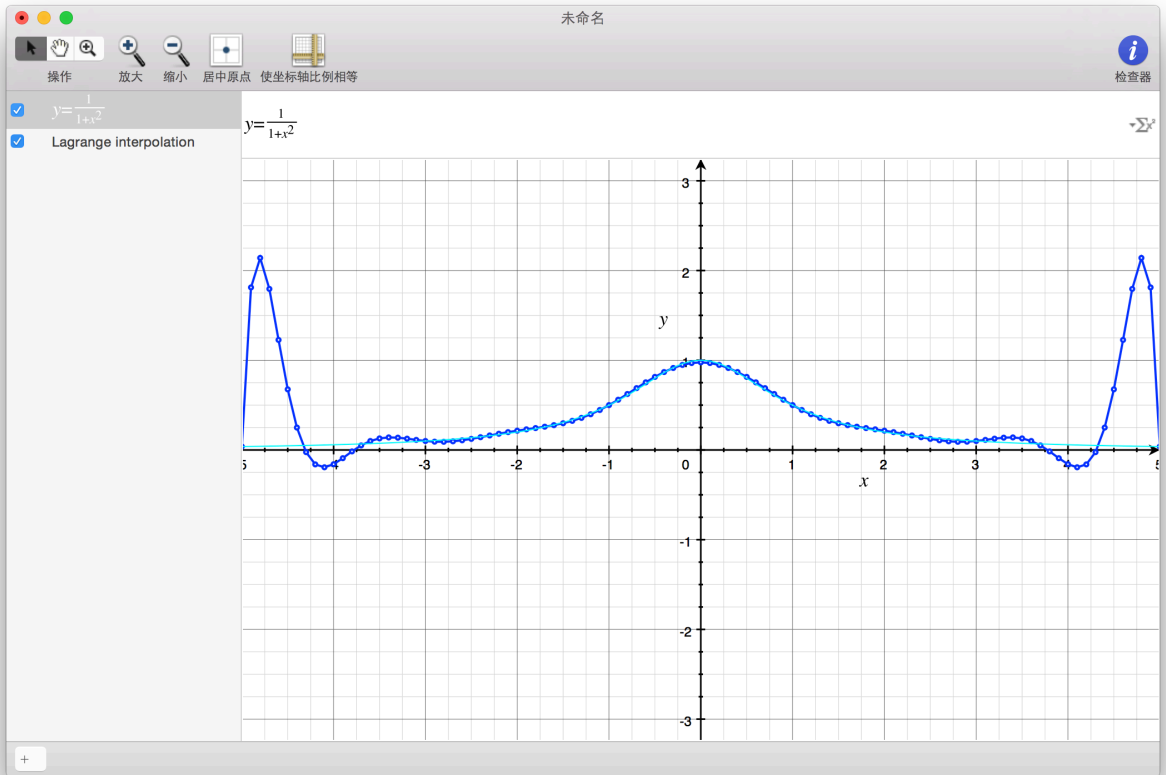
$$R_{n+1}(x) = \frac{x - x_0}{x_0 - x_{n+1}} (L_n(x) - L_n'(x))$$

```fortran
    ! 误差分析
    subroutine lagrange_interpolation_error_analysis(x,r)
        implicit none
        real :: x(:),r(:),t
        integer :: m,i,j
        m=size(x)
        do i=1,m
            r(i)=(x(i)-x_(1))/(x_(1)-x_(N))*(Ln(x(i),1,N-1)-Ln(x(i),2,N))
        end do
    end subroutine lagrange_interpolation_error_analysis
```

输出结果:

```
f — bash — 80×24
Set initail x=
-5.00000 -4.33333 -3.66667 -3.00000 -2.33333 -1.66667 -1.00000 -0.33333
 0.33333  1.00000  1.66667  2.33333  3.00000  3.66667  4.33333  5.00000

y=
 0.03846  0.05056  0.06923  0.10000  0.15517  0.26471  0.50000  0.90000
 0.90000  0.50000  0.26471  0.15517  0.10000  0.06923  0.05056  0.03846

Lagrange interpolation
            x             y             r
     -5.00000       0.03846      -0.00000
     -4.00000      -0.15894       0.00000
     -3.00000       0.10000      -0.00000
     -2.00000       0.21502      -0.00000
     -1.00000       0.50000      -0.00000
      0.00000       0.97625       0.00000
      1.00000       0.50000      -0.00000
      2.00000       0.21502      -0.00000
      3.00000       0.10000      -0.00000
      4.00000      -0.15894       0.00001
      5.00000       0.03846      -0.00047

Write the data to file: lagrange_data.txt
```

图例:

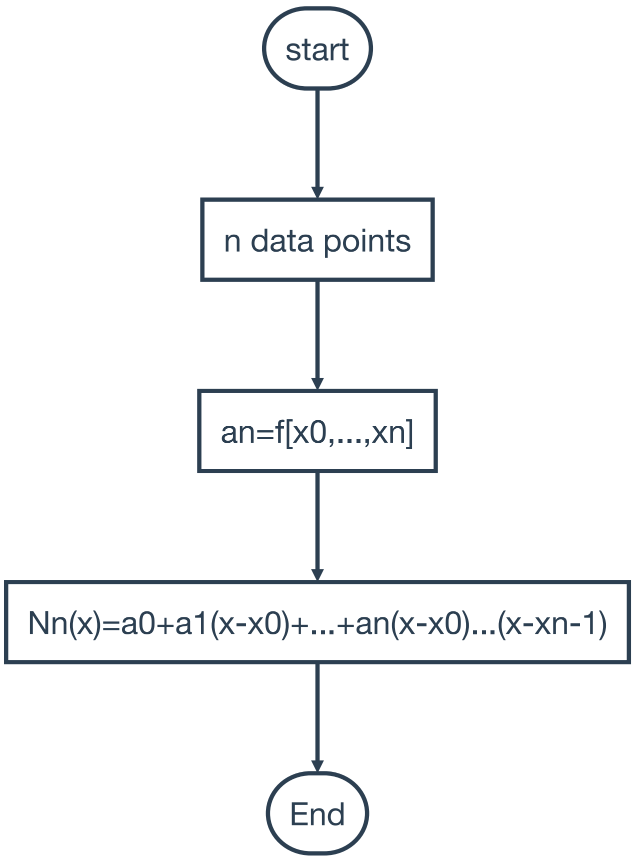


分析:

> 从图表中可以看出拉格朗日插值法算出的值与真实值大体上符合，但是在边界上存在较大的误差，这一点从图上看的很明显。
>
> 其原因主要是：当插值点比较多的时候，拉格朗日插值多项式的次数可能会很高，因此具有数值不稳定的特点，也就是说尽管在已知的几个点取到给定的数值，但在附近却会和“实际上”的值之间有很大的偏差。也就是所谓的龙格现象，解决的办法是分段用较低次数的插值多项式。

牛顿插值法

流程图：



原理：

$$a_n = f[x_0, \ldots, x_n]$$

$$N_n(x) = a_0 + a_1(x - x_0) + \ldots + a_n(x - x_0)\ldots(x - x_{n-1})$$

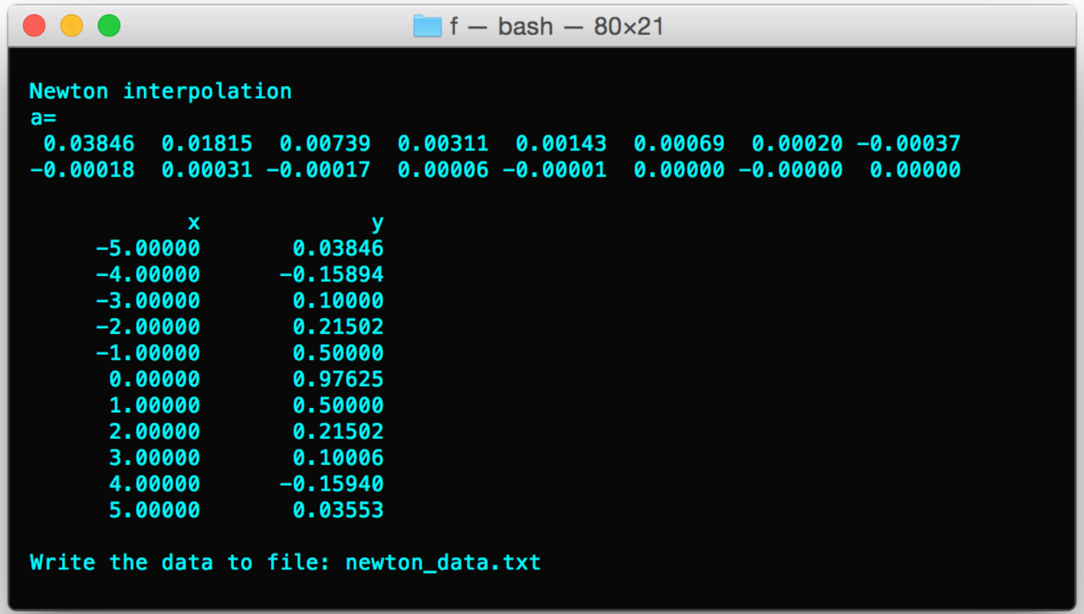
其中：$f[x_0, \ldots, x_k] = \dfrac{f[x_1, \ldots, x_k] - f[x_0, \ldots, x_{k-1}]}{x_k - x_0}$

代码：

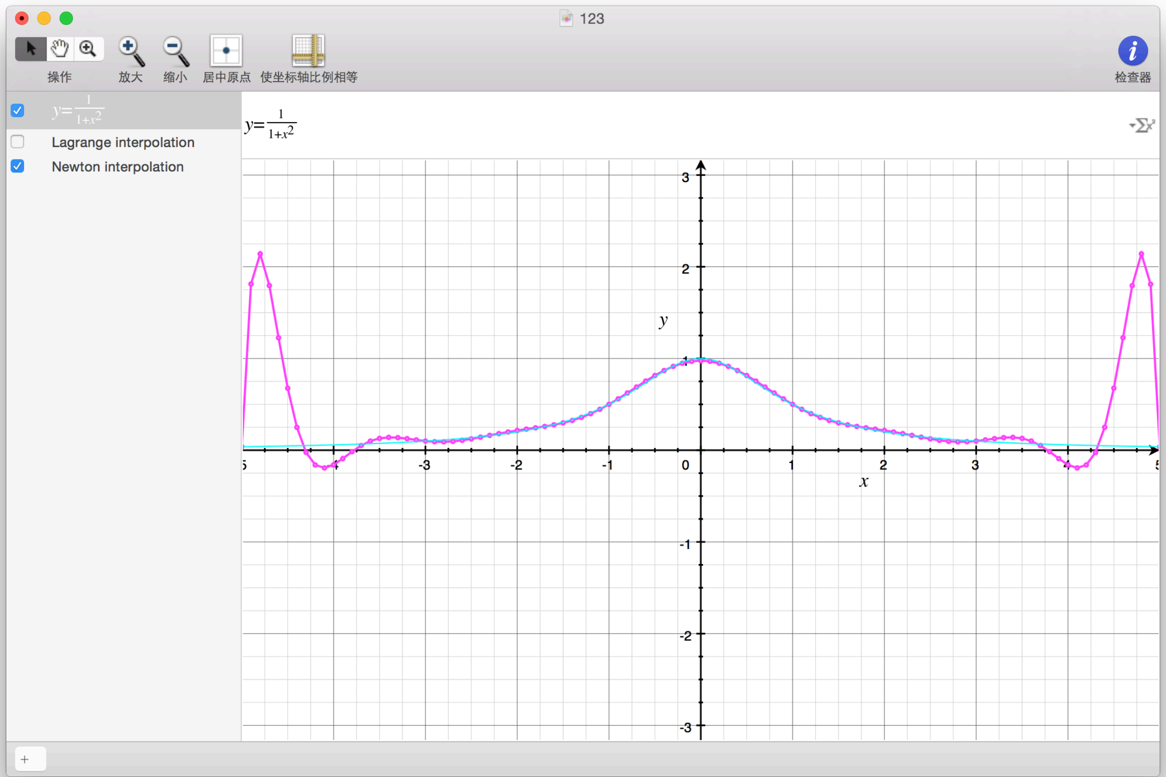
```
    ! Difference quotient
    recursive function f_(a,b) result(r)
        implicit none
        real :: r
        integer,intent(in) :: a,b

        select case (abs(b-a))
            case (0)
                r=y_(a)
            case (1)
                r=(y_(b)-y_(a))/(x_(b)-x_(a))
            case default
                r=(f_(a+1,b)-f_(a,b-1))/(x_(b)-x_(a))
        end select
    end function f_

    ! Nn(x)
    function Nn(x,a)
        implicit none
        real :: Nn,x,a(:),t
        integer :: i,j
        do i=1,N
            t=1.0
            do j=1,i-1
                t=t*(x-x_(j))
            end do
            Nn=Nn+a(i)*t
        end do
    end function

    ! 牛顿插值法
    subroutine newton_interpolation(x,y)
        implicit none
        real :: x(:),y(:),a(N)
        integer :: i
        a=(/ (f_(1,i),i=1,N) /)
        print *,'a='
        print '(f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5)',a
        print *

        open(101,file="newton_data.txt")
        do i=1,101
            ! There is a bug if you comment the following
code, I don't know why this would happen.
            write(101,'(f9.5,f9.5)') x(i),Nn(x(i),a)
            y(i)=Nn(x(i),a)
        end do
        close(101)
```

```
end subroutine newton_interpolation
```

输出结果:



图例:

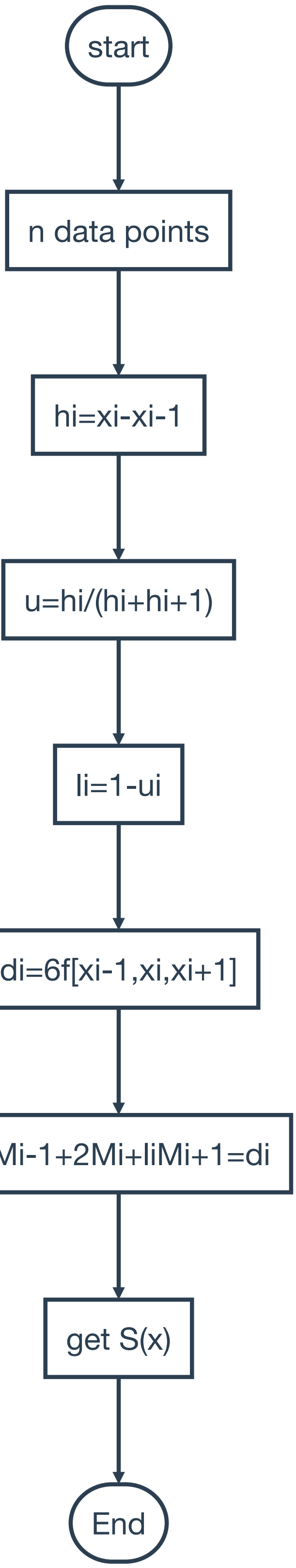


分析：

> 在拉格朗日插值法中，当插值节点增减时全部插值基函数均要随之变化，整个公式也将发生变化， 这在实际计算中是很不方便的，而牛顿插值法则克服这一缺点。
>
> 然而牛顿插值法得到的结果与拉格朗日插值法的结果类似，同样的在边界有较大的误差，而在中间拟合的较好。

三次曲线插值

流程图：



原理:

$$\begin{cases} S_i(x_i) = y_i & S_i(x_{i+1}) = y_{i+1} \\ S_i''(x_i) = M_i & S_i''(x_{i+1}) = M_{i+1} \end{cases} \quad x \in [x_i, x_{i+1}]$$

$$\mu_i M_{i-1} + 2M_i + \lambda_i M_{i+1} = d_i$$

其中

$$\mu_i = \frac{h_{i-1}}{h_i + h_{i-1}}$$

$$\lambda_i = \frac{h_i}{h_i + h_{i-1}}$$

$$d_i = 6f[x_{i-1}, x_i, x_{i+1}]$$

代码:

```
    ! Cubic Spline Curve
    subroutine cubic_spline_curve(x,y)
        implicit none
        real :: h(N-1),mu(N-2),lambda(N-2),d(N-2),M(N),mat(N-2,N-1),x(:),y(:)
        integer :: i,j

        h=(/ (x_(i)-x_(i-1),i=2,N) /)
        mu=(/ (h(i)/(h(i)+h(i+1)),i=1,N-2) /)
        lambda=(/ (1-mu(i),i=1,N-2) /)
        d=(/ (6*f_(i-1,i+1),i=2,N-1) /)
        M=0

        mat=0
        do i=1,N-2
            if (i /= 1) mat(i,i-1)=mu(i)
            mat(i,i)=2
            if (i /= N-2) mat(i,i+1)=lambda(i)
            mat(i,N-1)=d(i)
        end do
        call chasing(mat,M(2:N-1))

        y=0
        do i=1,size(x)
            do j=2,size(x_)
                if ( x(i)<=x_(j) ) exit
            end do
            y(i)=M(j-1)*(x_(j)-x(i))/(6*h(j-1))
            y(i)=y(i)+M(j)*(x(i)-x_(j-1))/(6*h(j-1))
            y(i)=y(i)+(y_(j-1)-M(j-1)*h(j-1)*h(j-1)/6)*(x_(j)-x(i))/h(j-1)
            y(i)=y(i)+(y_(j)-M(j)*h(j-1)*h(j-1)/6)*(x(i)-x_(j-1))/h(j-1)
        end do

        ! 输出一些基本信息到屏幕上
        print *
```

```
        print *,'h='
        print '(f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5)',h
        print *
        print *,'mu='
        print '(f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5)',mu
        print *
        print *,'lambda='
        print '(f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5)',lambda
        print *
        print *,'d='
        print '(f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5)',d
        print *
        print *,'Solve the matrix to get M'
        print
'(f6.2,f6.2,f6.2,f6.2,f6.2,f6.2,f6.2,f6.2,f6.2,f6.2,f6.2,f6.2,f6
.2,f6.2,f6.2)',(mat(i,:),i=1,N-2)
        print *
        print *,'M='
        print '(f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5,f9.5)',M
        print *

    end subroutine cubic_spline_curve
```

其中，用到了追赶法解对角占优矩阵求得M：

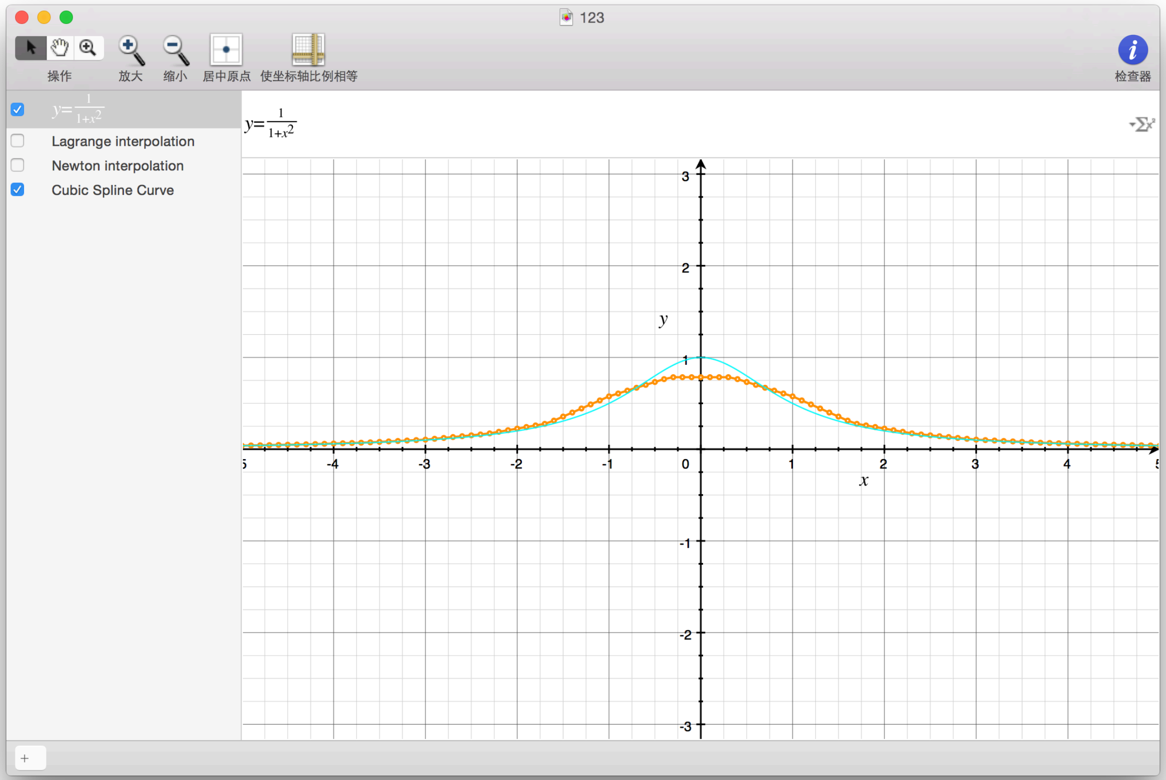
```fortran
    ! 追赶法
    subroutine chasing(matrix,x)
        implicit none
        real :: matrix(:,:),x(:)
        real,allocatable :: u(:),q(:)
        integer :: i,m,n
        m=size(matrix(1,:))
        n=size(matrix(:,1))

        allocate(u(n-1),q(n))
        u(1)=matrix(1,2)/matrix(1,1)
        q(1)=matrix(1,m)/matrix(1,1)

        do i=2,n-1
            u(i)=matrix(i,i+1)/(matrix(i,i)-u(i-1)*matrix(i,i-
1))
        end do

        do i=2,n
            q(i)=(matrix(i,m)-q(i-1)*matrix(i,i-
1))/(matrix(i,i)-u(i-1)*matrix(i,i-1))
        end do

        x(n)=q(n)
        do i=n-1,1,-1
            x(i)=q(i)-u(i)*x(i+1)
        end do

        deallocate(u,q)
    end subroutine chasing
```

输出结果:

f — bash — 90×52

```
Cubic spline interpolation

h=
 0.66667  0.66667  0.66667  0.66667  0.66667  0.66667  0.66667  0.66667
 0.66667  0.66667  0.66667  0.66667  0.66667  0.66667  0.66667

mu=
 0.50000  0.50000  0.50000  0.50000  0.50000  0.50000  0.50000
 0.50000  0.50000  0.50000  0.50000  0.50000  0.50000  0.50000

lambda=
 0.50000  0.50000  0.50000  0.50000  0.50000  0.50000  0.50000
 0.50000  0.50000  0.50000  0.50000  0.50000  0.50000  0.50000

d=
 0.04434  0.08168  0.16472  0.36694  0.84888  1.11177 -2.70000
-2.70000  1.11177  0.84888  0.36694  0.16472  0.08168  0.04434

Solve the matrix to get M
 2.00  0.50  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.04
 0.50  2.00  0.50  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.08
 0.00  0.50  2.00  0.50  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.16
 0.00  0.00  0.50  2.00  0.50  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.37
 0.00  0.00  0.00  0.50  2.00  0.50  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.85
 0.00  0.00  0.00  0.00  0.50  2.00  0.50  0.00  0.00  0.00  0.00  0.00  0.00  0.00  1.11
 0.00  0.00  0.00  0.00  0.00  0.50  2.00  0.50  0.00  0.00  0.00  0.00  0.00  0.00 -2.70
 0.00  0.00  0.00  0.00  0.00  0.00  0.50  2.00  0.50  0.00  0.00  0.00  0.00  0.00 -2.70
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.50  2.00  0.50  0.00  0.00  0.00  0.00  1.11
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.50  2.00  0.50  0.00  0.00  0.00  0.85
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.50  2.00  0.50  0.00  0.00  0.37
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.50  2.00  0.50  0.00  0.16
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.50  2.00  0.50  0.08
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.50  2.00  0.04

M=
 0.00000  0.01573  0.02576  0.04460  0.12528  0.18816  0.81983 -1.24397
-1.24397  0.81983  0.18816  0.12528  0.04460  0.02576  0.01573  0.00000


             x             y
     -5.00000        0.03846
     -4.00000        0.06182
     -3.00000        0.10413
     -2.00000        0.22445
     -1.00000        0.57591
      0.00000        0.78482
      1.00000        0.57591
      2.00000        0.22445
      3.00000        0.10413
      4.00000        0.06182
      5.00000        0.03846
```

图例：



分析：

> 由图可知三次样条插值法求得的结果与真实结果整体都十分接近，误差相对于拉格朗日插值法和牛顿插值法都要小很多。
>
> 并且，由于三次样条插值法使用低阶多项式样条，所以能实现较小的插值误差，这样就避免了使用高阶多项式所出现的龙格现象。