数值计算实验2

数据科学与计算机学院

梁育诚

学号 16340133

班级 教务二班

内容1

已知 sin(0.32)=0.314567，sin(0.34)=0.333487，sin(0.36)=0.352274，sin(0.38)=0.370920。请采用线性插值、二次插值、三次插值分别计算 sin(0.35)的值。

问题描述

本题涉及到了插值的问题。所谓插值，就是根据已经得到的一些离散的点，通过某种插值方式，求得一个**较为理想**的插值函数，从而求出特定位置的值。这题要求分别使用**线性插值**、**二次插值**、**三次插值**三种插值方式去求得sin(0.35)的值。这三种插值方式可由**拉格朗日插值多项式**统一表达，因此我们只需要写好一个函数，通过输入不同的插值点即可。

算法设计

假设已知n+1个节点，这样就可以构造n次多项式$L_n(x)$，其中$L_n(x)$满足

$$L_n(x_j) = y_i, j = 0, 1, \dots, n.$$

也就是说，插值多项式在已知节点上的值必须是和已知值相等的。
然后我们就需要构造一个完整的插值多项式$L_n(x)$。构造$L_n(x_j)$的一个思路是使用$y_0, y_1, \dots, yn$这些点的线性组合，我们就需要定义n次插值的基函数$l_j(x)(j = 0, 1, \dots, n)$，它满足

$$l_j(x_k) = \begin{cases} 1, k = j, \\ 0, k \neq j \end{cases}$$

通过推导，这里直接给出n次插值基函数的形式：

$$l_k(x) = \frac{(x - x_0)\dots(x - x_{k-1})(x - x_{k+1})\dots(x - x_n)}{(x_k - x_0)\dots(x_k - x_{k-1})(x_k - x_{k+1})\dots(x_k - x_n)}, k = 0, 1, \dots, n$$

故我们可以得出完整的**拉格朗日插值多项式**

$$L_n(x_j) = \sum_{k=0}^{n} y_k l_k(x_j) = y_j, j = 0, 1, \dots, n$$

根据上面得出的拉格朗日插值多项式，我们就可以代入已知的点，根据多项式求出$sin(0.35)$的值了。

数值实验

根据上述推导出来的公式，编写matlab代码：首先计算出所有的基函数，然后使用基函数构成关于y_k的线性组合得出插值多项式，最后代入指定的点得出结果。
线性插值代入两个点，二次插值代入三个点，三次插值代入四个点，然后使用matlab系统自带的函数计算出结果，观察不同次数的插值结果的误差。
matlab代码如下：

```
% 拉格朗日插值
function [y0] = Lagrange(x,y,x0)
format long;
% 给定插值点(x,y)，求在x=x0处的值y0
y0 = 0;
n = length(x);
l = ones(1,n); % 基函数集合
for k = 1:n
    for j = 1:n
        if j ~= k
            l(k) = l(k)*(x0-x(j))/(x(k) - x(j)); % 计算插值基函数
        end
    end
end
% 计算插值多项式
for i = 1:n
    y0 = y0 + y(i)*l(i);
end
```

结果分析

测试代码运行结果如下：

```
>> test_Lagrange
标准答案：
    0.3429

线性插值：
   0.342880500000000

二次插值：
   0.342897125000000

三次插值：
   0.342897625000000
```

由上述结果可以看出，插值点越多，计算的结果误差越小，这也符合我们的一般逻辑——提供越多的点，插值的结果越精确。

内容2

请采用下述方法计算 115 的平方根，精确到小数点后六位。

（1）二分法。选取求根区间为[10, 11]。
（2）牛顿法。
（3）简化牛顿法。
（4）弦截法。
绘出横坐标分别为计算时间、迭代步数时的收敛精度曲线。

问题描述

　　本题涉及到的是非线性方程的求根方法。非线性问题一般不存在直接求解的方法，大多数都是使用**迭代法**来求解，通过确定误差范围，来求得一个**满意的近似解**。题目要求用到的方法大致可以分为两类：一是通过缩小有根区间而得到近似解的方法，如二分法；二是不动点迭代法，如牛顿法、简化牛顿法和弦截法。
　　本题中求解115的平方根可以转化为非线性方程$f(x) = x^2 - 115$的求根问题，下面我将用四种方法来求解。

算法设计

1.二分法

　　二分法的实现思想是通过迭代的方法来缩小有根区间，最终这个区间必收敛到一点x^*，这点就是我们要求的根。在我们实际求解的过程中，没有必要求出这一点x^*，我们只需要确定一个误差范围，让这个有根区间的长度小于这个误差即可。
　　二分法的核心是如何确定有根区间，首先给出一个较大的有根区间，然后通过不断地二分，通过比对端点值与中点值得正负，来判断根所在的区间。下面给出算法步骤：
① 准备：计算$f(x)$在有根区间$[a, b]$端点处的值$f(a)$，$f(b)$。
② 二分：计算$f(x)$在区间中点$\frac{a+b}{2}$处的值$f(\frac{a+b}{2})$
③ 判断：若$f(\frac{a+b}{2})$ = 0，则$\frac{a+b}{2}$就是该非线性方程的根，计算结束，否则检验；若$f(\frac{a+b}{2})f(a)$ < 0，则以$\frac{a+b}{2}$代替b成为区间上界，否则以$\frac{a+b}{2}$代替a成为区间下界。
重复执行步骤②和步骤③，直至计算结束或区间长度小于规定的误差，此时**中点**$\frac{a+b}{2}$**为所求的近似根。**

2.牛顿法

　　牛顿法的核心思想是将非线性问题转化为线性问题处理。对于非线性方程$f(x) = 0$，假设已知有近似根x_k（假定$f'(x_k) \neq 0$）,将函数$f(x)$在点x_k进行**泰勒展开**，有

$$f(x) \approx f(x_k) + f'(x_k)(x - x_k)\,,$$

于是方程$f(x) = 0$可以近似地表示为

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

上面这个是一个线性方程，记其根为x_{k+1}，则求得x_{k+1}的计算公式为：

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}\,,$$

以上的迭代方法就称为**牛顿法**。

3.简化牛顿法

简化牛顿法是牛顿法的变种，原因在于，牛顿法中每一步都需要计算$f(x_k)$和$f'(x_k)$，这是需要很大计算量的。除此之外，牛顿法中的初始近似x_0只在精确解x^*附近才能保证收敛。简化牛顿法就是为了解决这个问题的。首先给出简化牛顿法的迭代公式：

$$x_{k+1} = x_k - Cf(x_k),$$

则迭代函数为：$\phi(x) = x - Cf(x)$.

若$|\phi'(x)| = |1 - Cf'(x)| < 1$，即取$0 < Cf'(x) < 2$在根附近成立，则该迭代法局部收敛。

同时取$C = \frac{1}{f'(x_0)}$，这样就只需要在第一步计算$f'(x_0)$，大大减少了计算量。其几何意义是用斜率为$f'(x_0)$的平行弦与x轴交点作为x^*的近似解。

4.弦截法

弦截法也是牛顿法的一个变种，同样也是为了避免计算$f'(x_k)$。这里采用的方法是使用已求的函数值$f(x_k), f(x_{k-1}), \ldots$来回避导数值$f'(x_k)$的计算，这种方法是建立在插值原理的基础上的。

设x_k, x_{k-1}是$f(x) = 0$的近似根，我们利用$f(x_k), f(x_{k-1})$构造一次插值多项式$p_1(x)$，并用$p_1(x) = 0$的根作为$f(x) = 0$的新的近似根x_{k+1}，根据一次插值公式：

$$p_1(x) = f(x_k) + \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}(x - x_k),$$

因此有

$$x_{k+1} = x_k - \frac{f(x_k)}{f(x_k) - f(x_{k-1})}(x_k - x_{k-1}),$$

以上就是弦截法的迭代公式了。与牛顿法对比，不难看出，弦截法使用$\frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$代替了牛顿法中的导数$f'(x_k)$。

弦截法的几何意义是，使用曲线$y = f(x)$上的两点x_k, x_{k-1}的弦线与x轴的交点作为作为x^*的近似解。弦截法与牛顿法都是线性化方法，但是两者有很大不同。牛顿法在计算x_{k+1}的时候只用到了上一步的结果x_k，而弦截法在求x_{k+1}时要用到前两部的计算结果x_k，x_{k-1}，因此在使用弦截法的时候要首先给出两个值x_0，x_1。

数值实验

1.二分法

根据上述分析的二分法算法过程，编写matlab代码。

```
% 二分法
function [y, count, error, time] = dichotomy(a, b)
format long;
count = []; %输出迭代次数的数组
error = []; %输出误差的数组
time = []; % 输出迭代时间的数组
err = 0; % 误差变量
k = 0; % 迭代次数变量
e = 0.000001; % 精度控制

x = (a + b) / 2;
f1 = myFun(a);
f2 = myFun(b);
fx = myFun(x);
y = a;

if (fx == 0)
    y = x;
end

tic
while (b-a) > (2*e)
    fx = myFun(x);
    % 得出精确解
    if (fx == 0)
      y = x;
      break;
    elseif (f1 * fx < 0)
        b = x;
        f2 = fx;
    else
        a = x;
        f1 = fx;
    end
    y_old = y;
    y = x;
    k = k + 1;
```

```
    count(k) = k;
    x = (a + b) / 2;
    err = abs(y-y_old)/y;
    error(k) = err;
end
toc

% 均分时间间隔
temp = toc / k;
for i=1:k
    time(i) = i*temp;
end

end

% 求解函数
function [y] = myFun(x)
y = x*x - 115;
end
```

2.牛顿法

根据上述分析得出的牛顿法迭代公式，编写matlab代码。

```
% 牛顿法
function [y, count, error, time] = Newton(x, e)
    % y为最终结果
    % x为开始迭代的初始坐标
    % e为迭代精度

    del_x = 0.0000001; % 用于求函数导数值的极小量
    count = [];% 输出迭代次数的数组
    time = []; % 输出迭代时间的数组
    error = [];% 输出误差的数组
    k = 0; % 迭代次数变量
    err = 0;% 误差变量
    y = x;
    x = y + 1000; % 保证迭代能开始
    n = 50; % n为最大迭代次数
    tic
    while 1
        if (abs(y-x) <= e)
            disp('满足迭代精度');
            break;
        elseif (k > n)
            disp('迭代次数过多，迭代结束');
            break;
        else
            x = y;
            if((myFun(x+del_x) - myFun(x)) == 0)
                disp('导数为0');
                break;
            else
                y_deriv = (myFun(x+del_x) - myFun(x)) / del_x; %x点的导数值
                y = x - myFun(x) / y_deriv; % 牛顿迭代
                k = k + 1; %迭代次数加1
                count(k) = k;
                err = abs(y-x) / y;
                error(k) = err;
            end
        end
    end
    toc

    % 均分时间间隔
    temp = toc / k;
    for i=1:k
        time(i) = i*temp;
    end

    disp('牛顿迭代结束');
end


function [y] = myFun(x)
y = x*x - 115;
end
```

3.简化牛顿法

简化牛顿法可以先在一开始就计算出$f'(x_0)$，随后在循环过程中直接使用就可以了。

```
function [y, count, error, time] = Newton1(x0, e)
    % y为最终结果
    % x为开始迭代的初始坐标
    % e为迭代精度
    k = 0; % 迭代次数变量
    err = 0;% 误差变量
    count = [];% 输出迭代次数的数组
    time = []; % 输出迭代时间的数组
    error = [];% 输出误差的数组
    n = 50; % n为最大迭代次数
    del_x = 0.0000001; % 用于求函数导数值的极小量
    y_deriv = (myFun(x0+del_x) - myFun(x0)) / del_x; % x0点的导数值
    y = x0;
    x0 = y + 1000; % 保证迭代能开始

tic
while 1
    if (abs(y-x0) <= e)
        disp('满足迭代精度');
        break;
    elseif (k > n)
        disp('迭代次数超界');
        break;
    else
        x0 = y;
        if((myFun(x0+del_x) - myFun(x0)) == 0)
                disp('导数为0');
                break;
        else
            y = x0 - myFun(x0) / y_deriv; % 简化牛顿法
            k = k + 1; % 迭代次数加1
            count(k) = k;
            err = abs(y-x0)/y;
            error(k) = err;
        end
    end
end
toc

    % 均分时间间隔
    temp = toc / k;
    for i=1:k
        time(i) = i*temp;
    end

    disp('简化牛顿迭代结束');
end

function [y] = myFun(x)
y = x*x - 115;
end
```

4.弦截法

根据上面推导的迭代函数，编写代码。

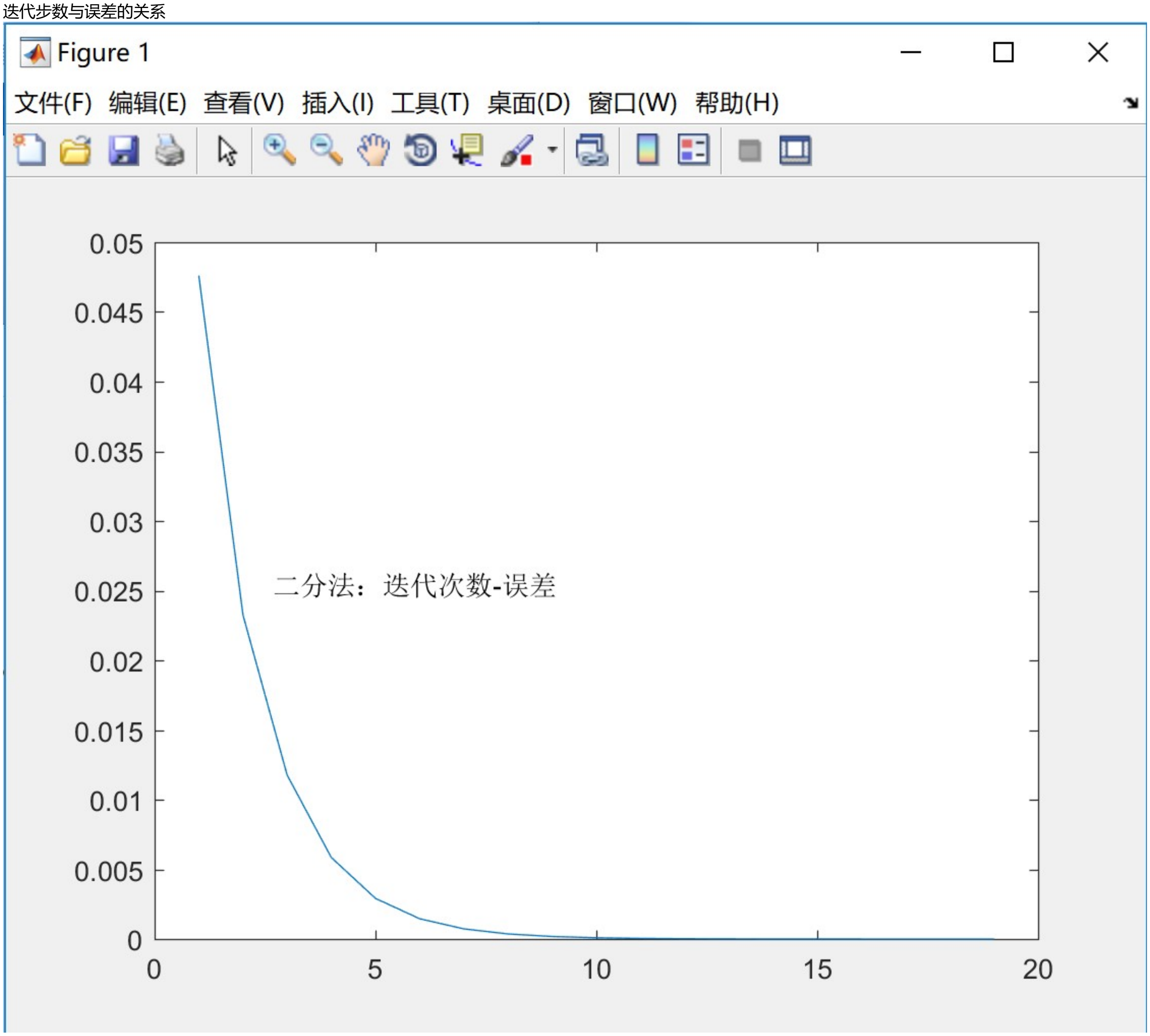
```
function [y, count, error, time] = Secant(x0, x1, e)
    % y为最终结果
    % x为开始迭代的初始坐标
    % e为迭代精度
    n = 50; % n为最大迭代次数
    k = 0;% 迭代次数变量
    err = 0;% 误差变量
    count = [];%输出迭代次数的数组
    time = []; % 输出迭代时间的数组
    error = [];% 输出误差的数组

    tic
    while 1
        y = x1 - myFun(x1) * (x1 - x0) / (myFun(x1) - myFun(x0));
        err = abs(y - x1) / y;
        if (abs(y - x1) <= e)
            disp('满足迭代精度');
            break;
        elseif (k > n)
            disp('迭代次数超界');
            break;
        else
            x0 = x1;
            x1 = y;
            k = k + 1;
            count(k) = k;
```

```
            error(k) = err;
        end
    end
    toc

    % 均分时间间隔
    temp = toc / k;
    for i=1:k
        time(i) = i*temp;
    end

    disp('弦截法迭代结束');
end

function [y] = myFun(x)
y = x*x - 115;
end
```

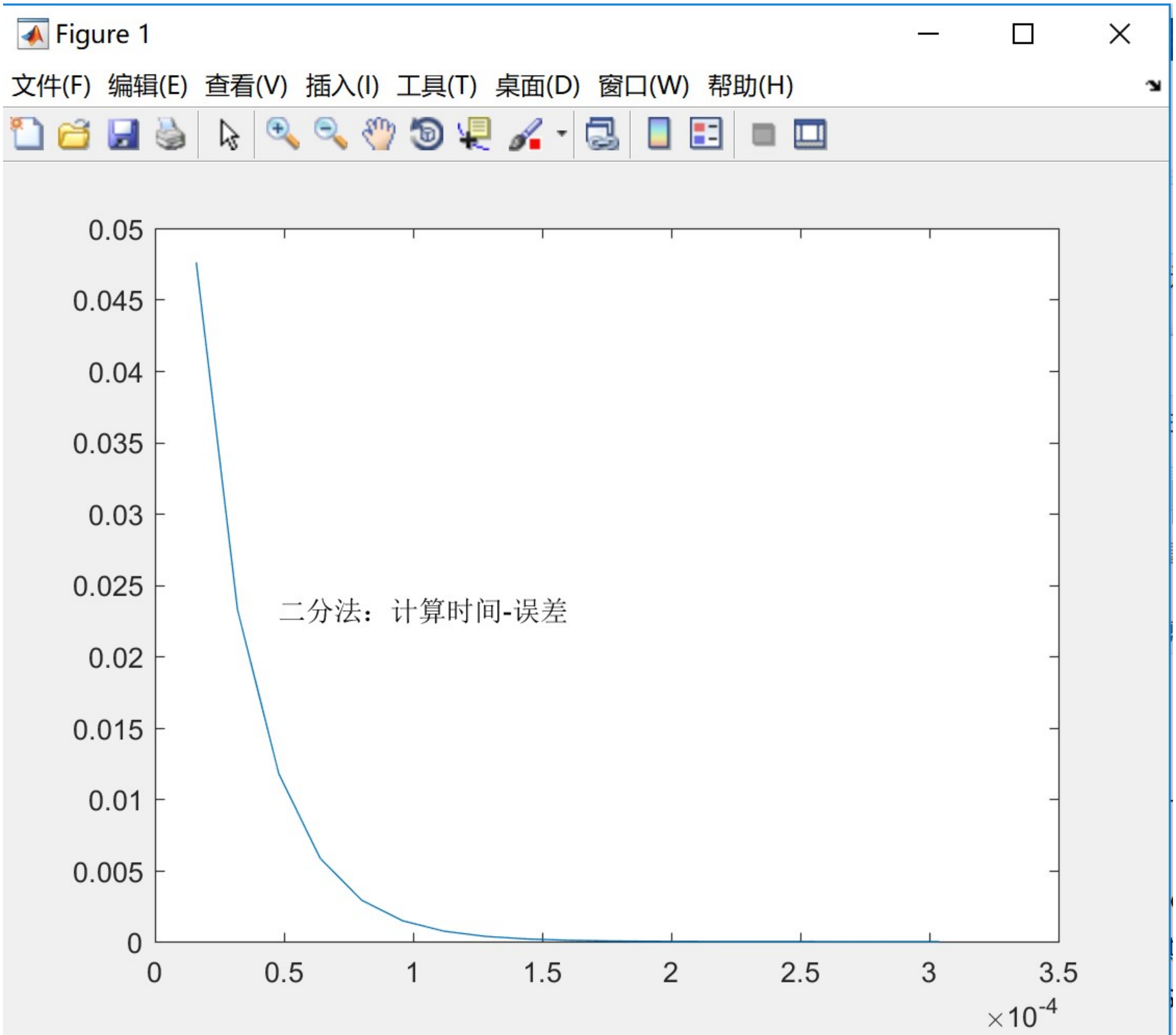
结果分析

1.二分法

运行二分法测试代码，每步都计算误差，从而得出**迭代步数-误差**和**计算时间-误差**两条曲线。结果如图：
迭代步数与误差的关系



计算时间与误差的关系



分析：由此可知，随着迭代次数的增加或计算时间的增加，误差逐渐收敛到0。从图一可以看出，大概需要18步就可以求得**近似解**（精确到小数点后6位）

2.牛顿法

运行代码后，得出结果如下：
迭代步数与误差的关系

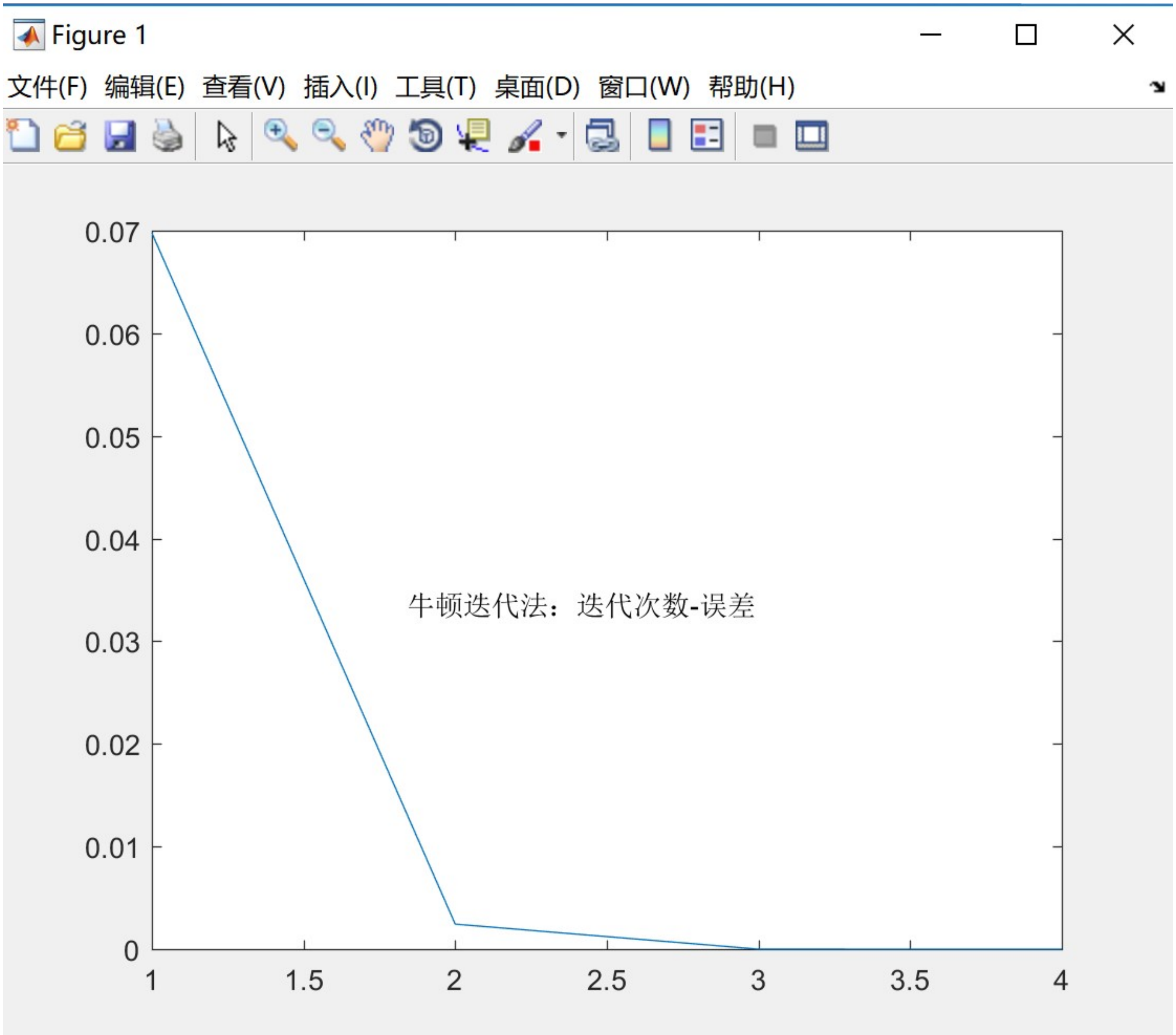
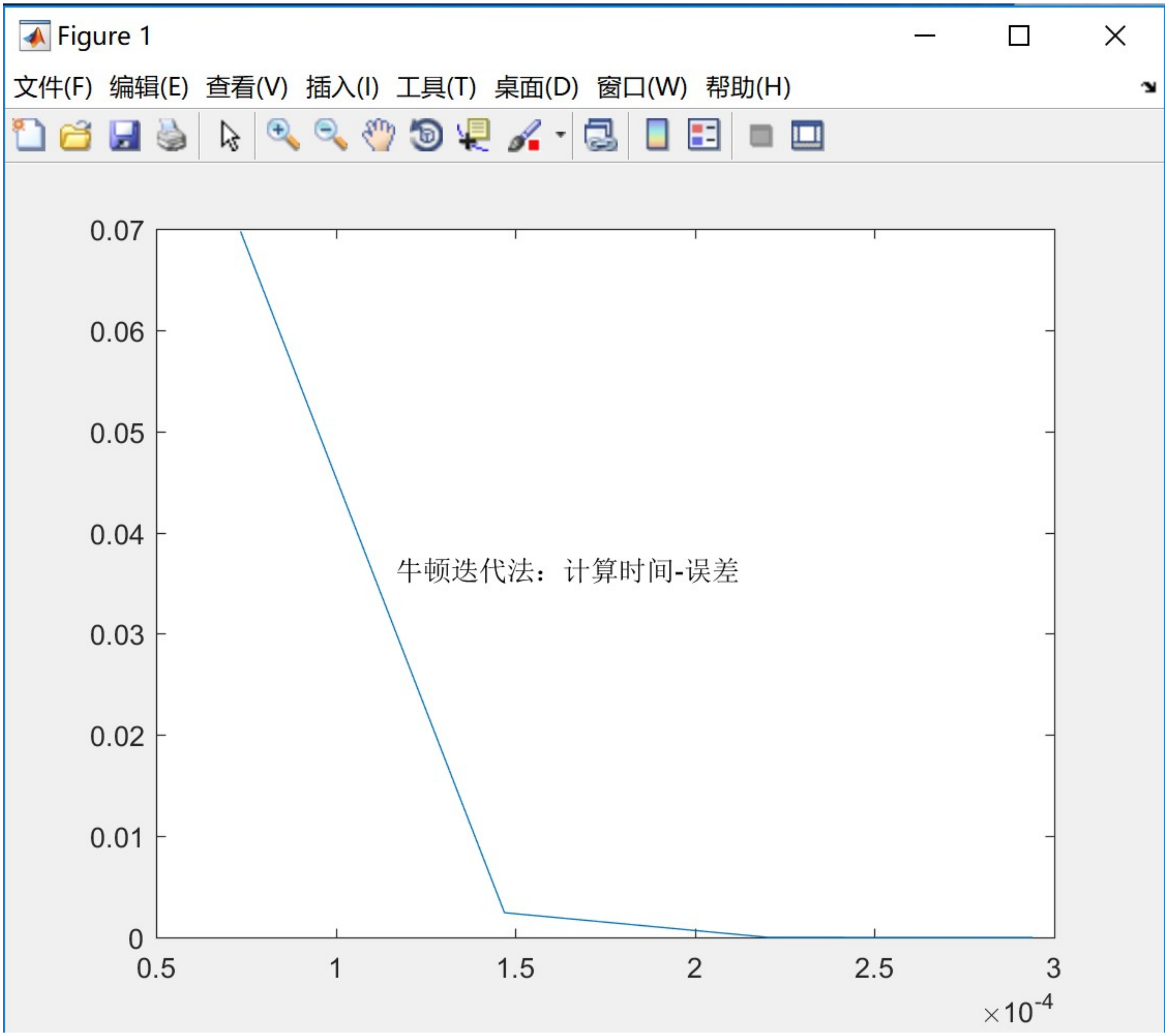


Figure 1
文件(F) 编辑(E) 查看(V) 插入(I) 工具(T) 桌面(D) 窗口(W) 帮助(H)
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
1
1.5
2
2.5
3
3.5
4
牛顿迭代法：迭代次数-误差

计算时间与误差的关系



分析：从图形看出，牛顿法的收敛速度很快，4步就已经收敛了。时间也比**二分法**稍快一些。可以看出**牛顿法**的收敛速度是很快的，符合理论推导。

3.简化牛顿法

运行代码，得出的结果如下：
迭代步数与误差的关系

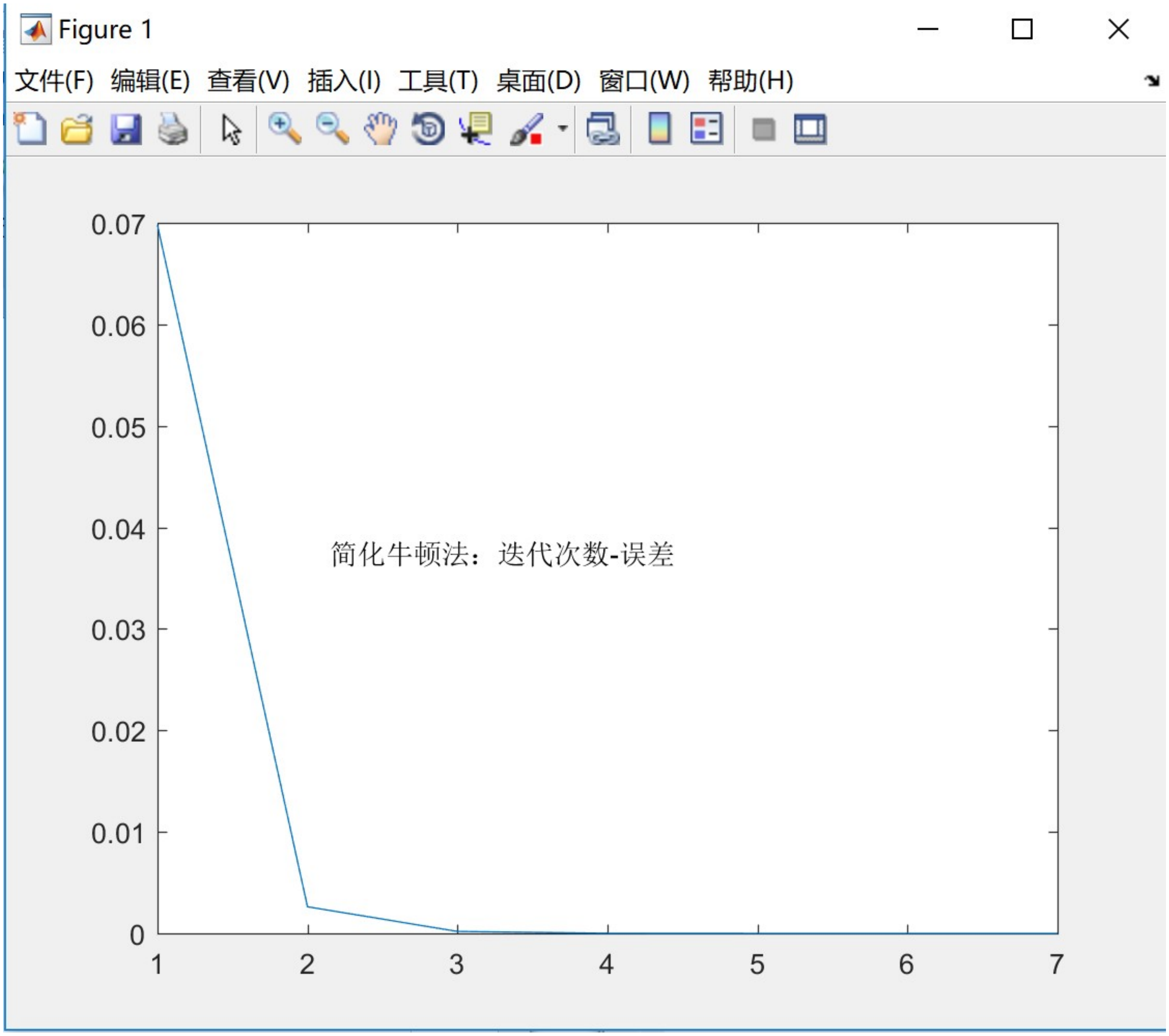
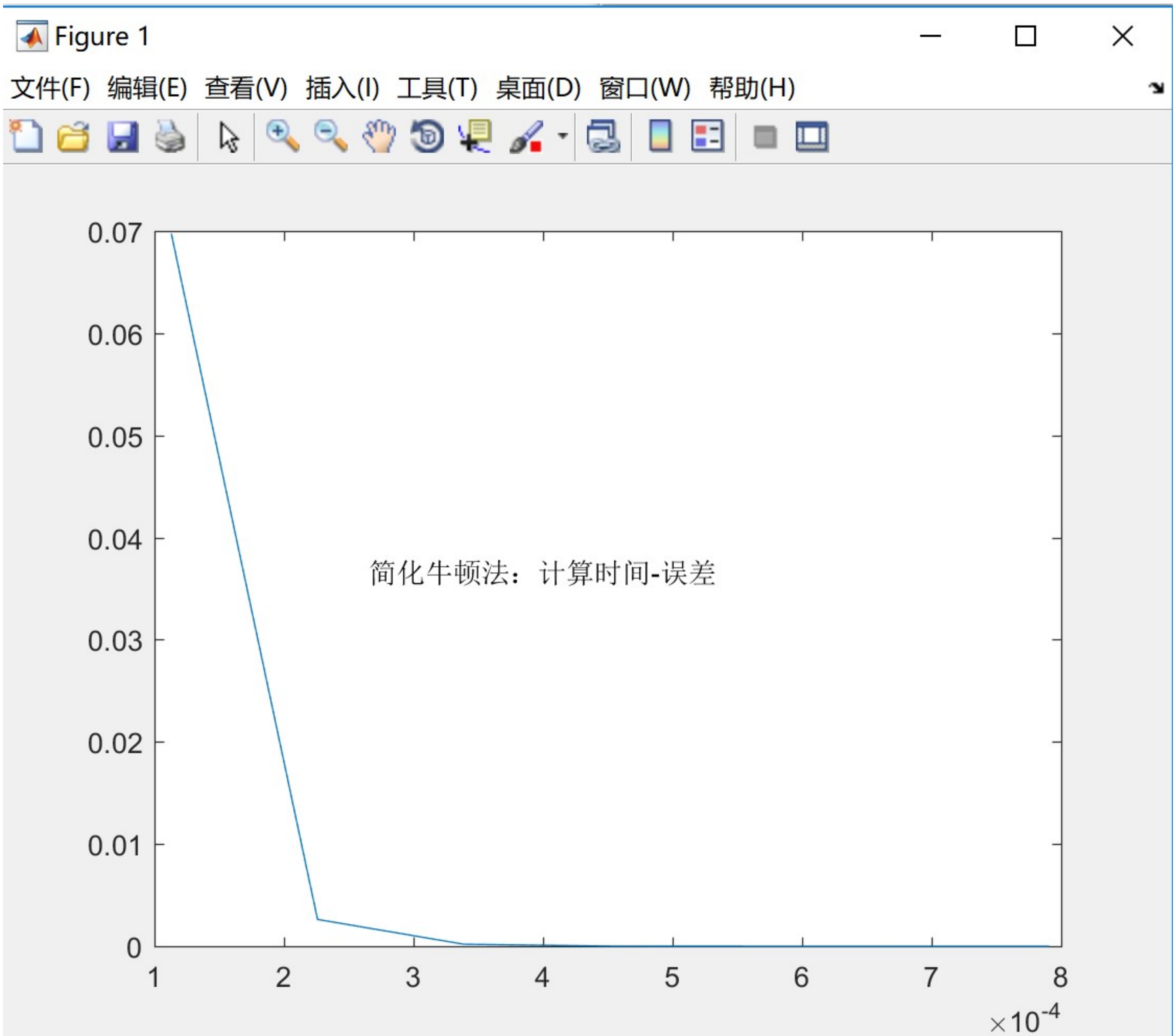


Figure 1
文件(F) 编辑(E) 查看(V) 插入(I) 工具(T) 桌面(D) 窗口(W) 帮助(H)
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
1
2
3
4
5
6
7
简化牛顿法：迭代次数-误差

计算时间与误差的关系



分析：从结果分析，简化牛顿法的迭代次数要比牛顿法多，这说明简化牛顿法的收敛速度是不如牛顿法的。它的特点是减少了运算量，但是**收敛速度是线性**的。

4.弦截法

运行代码，有如下结果：
迭代步数与误差的关系

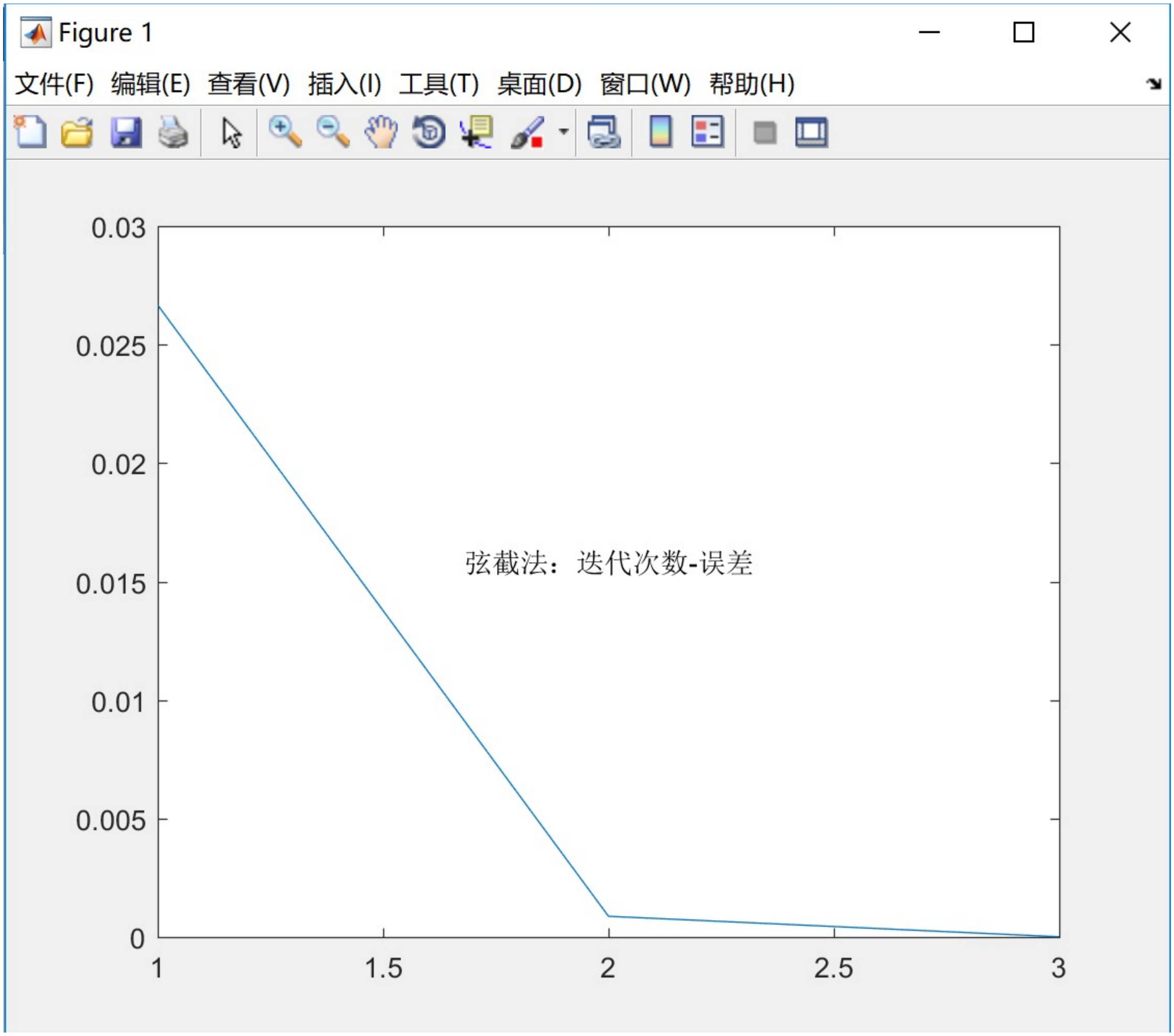
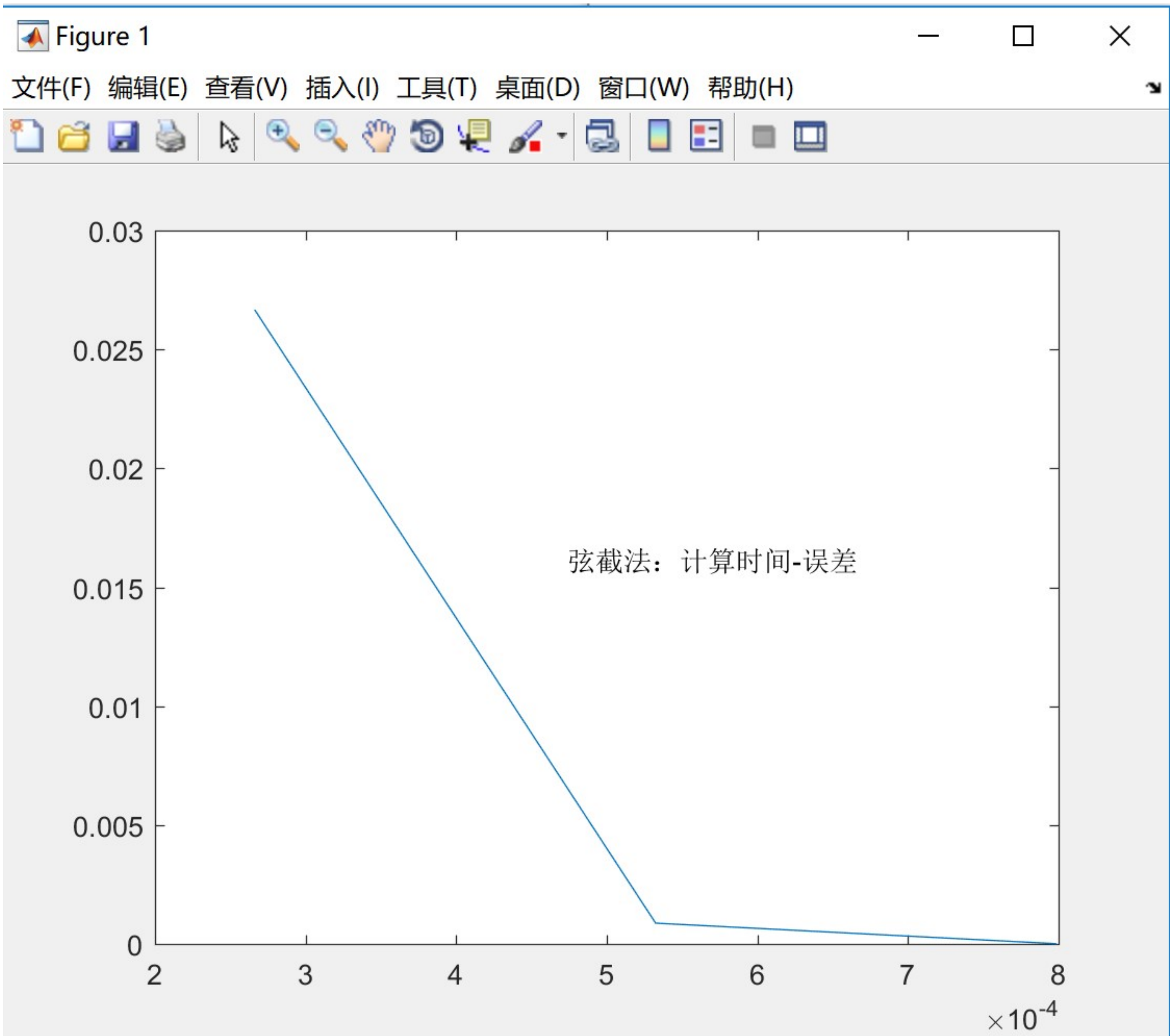


Figure 1
文件(F) 编辑(E) 查看(V) 插入(I) 工具(T) 桌面(D) 窗口(W) 帮助(H)
0.03
0.025
0.02
0.015
0.01
0.005
0
1
1.5
2
2.5
3
弦截法：迭代次数-误差

计算时间与误差的关系



分析：由结果分析可知，弦截法的迭代次数和计算时间都是最少的。从理论上讲，牛顿法在精确解x^*附近是**平方收敛**的，而弦截法是**超线性收敛的**，收敛速度约为1.618，但是在这里，由于我选取的前两个点是$x=10$和$x=11$，非常接近精确解，所以迭代步数会比牛顿法要少。如果换成其他更加复杂和庞大的数据，牛顿法的收敛速度会比弦截法快。

内容3

请采用递推最小二乘法求解超定线性方程组 Ax=b，其中 A 为 m x n 维的已知矩阵，b 为m 维的已知向量，x 为 n 维的未知向量，其中n=10，m=10000。A 与 b 中的元素服从独立同分布的正态分布。绘出横坐标为迭代步数时的收敛精度曲线。

问题描述

对于给定的超定线性方程组$Ax=b$，直接通过矩阵求解是困难的。这里就用到了逼近的思想。假设上述方程存在一个噪声e，则可以写成$b-Ax=e$，接下来要做的就是求解出能够**极小化**这个噪声的x。本题要求使用到了递推最小二乘法。

算法设计

递推最小二乘法的思路是基于最小二乘法，采用递推的形式来求得最终的x。首先来分析x的形式：

$$x^* = \min_x ||b-Ax||^2 ,$$

对上述等式右边求一阶梯度：

$$2A^T(Ax-b)=0$$
$$x=(A^TA)^{-1}A^Tb$$

得到了x的表示形式后，就可以根据每个$b(m)$来推导$x(m)$。

它的递推公式为：

$$K(m)=\frac{P(m-1)\phi_m}{1+\phi_m^T P(m-1)\phi_m}$$

$$P(m)=(I-K(m)\phi_m^T)P(m-1)$$

$$\hat{x}(m)=\hat{x}(m-1)+K(m)(b_m-\phi_m^T\hat{x}(m-1))$$

，其中ϕ_m^T为矩阵A的第m行。

每一步迭代，我们都通过在旧的估计值上，加上经过预测误差的真正测量值来得到新的估计值，最后得到较为准确的结果x。

数值实验

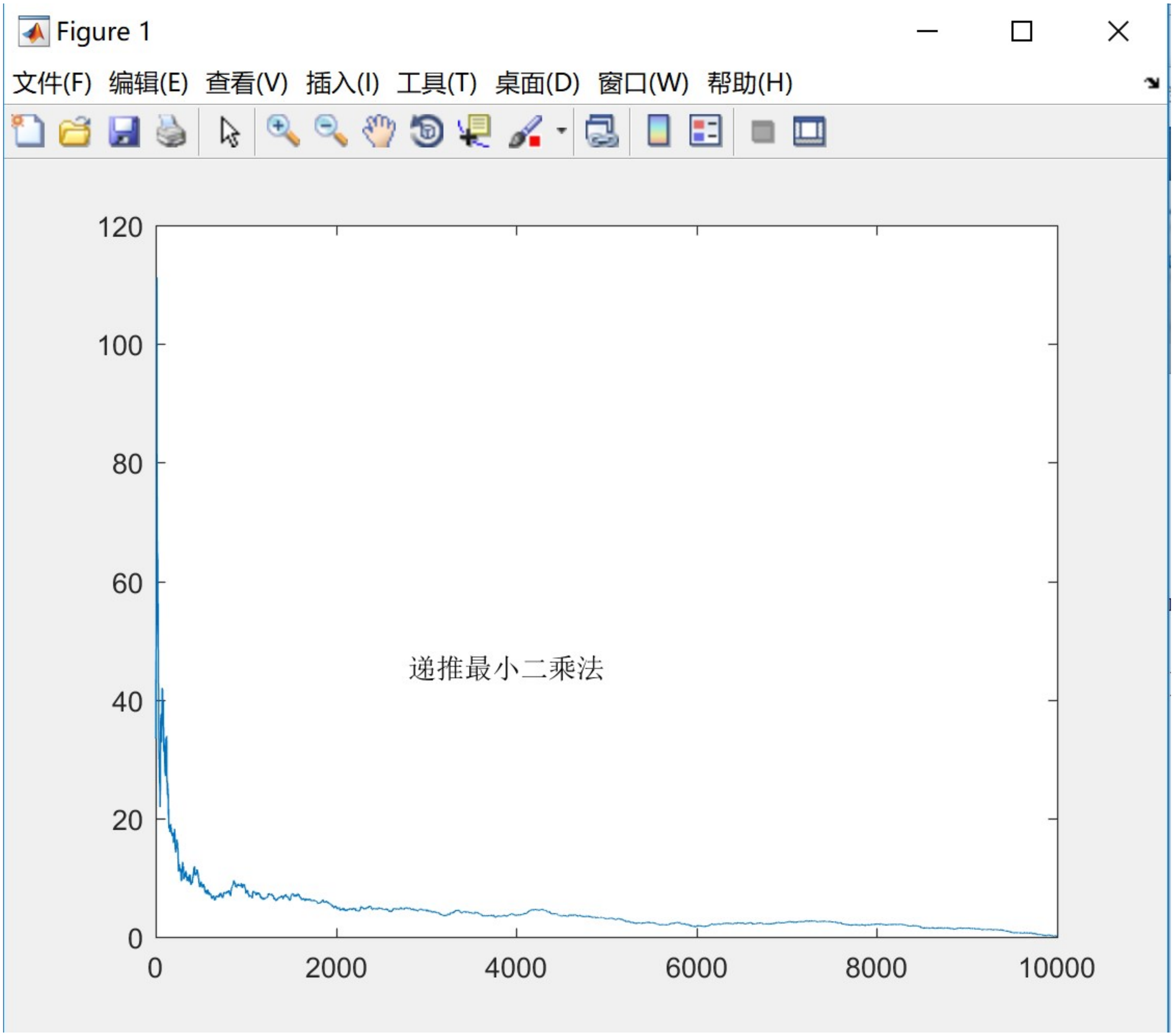
根据上述推导的公式，编写matlab代码，特别需要注意ϕ_m^T是$A(m,:)$，而ϕ_m是$A(m,:)'$。

```
% 递推最小二乘法
function [result, error, count] = LeastSquares(A,b)
    m = 10000;
    n = 10;
    count = [];
    standard = lsqnonneg(A,b); % 标准答案
    p = 1000*eye(n); % p为n*n的单位矩阵
    x = zeros(n,1); % x为n*1的向量
    k = zeros(n,1); % k为n*1的向量;

    for i=1:m
        k = (p*A(i,:)') / (1 + A(i,:)*p*A(i,:)');
        p = (eye(n) - k*A(i,:)) * p;
        x = x + k * (b(i,1) - A(i,:)*x);
        error(i) = norm((x - standard),2);
        count(i) = i;
    end
    result = x;
end
```

结果分析

运行代码，结果如下：

迭代步数-收敛精度曲线



分析：从图中可以看出，递推最小二乘法最终也是可以求得一个精确解的。递推最小二乘法最大的特点是在线识别，需要存储的信息较少。但是如果给出的数据较少或者初值点比较粗糙，递推最小二乘法的精度会有所降低。

内容4

请编写 1024 点快速傅里叶变换的算法。自行生成一段混杂若干不同频率正弦的信号，测试所编写的快速傅里叶变换算法。

问题描述

在日常生活的很多测试中，我们得到的数据都是由不同频率不同振幅的波形叠加起来的数据。在计算的时候需要逼近这种波形，选取具有周期性的三角函数作为基函数是合适的。对于计算傅里叶逼近系数问题，都可以统一地归结为：

$$c_j=\sum_{k=0}^{N-1}x_k\omega_N^{kj},j=0,1,\ldots,N-1$$

，这就是**N点DFT**
其中$x_{k_0}^{N-1}$为已知的输入数据点（采样点），$c_{j_0}^{N-1}$为输出数据。

算法设计

为了减少乘法的次数，这里充分利用了三角函数的周期性。对于$\omega_N^{jk}(j,k=0,1,\ldots,N-1)$而言，最多有N个不同的值。特别地，有

$$\omega_N^0=\omega_N^N=1,\omega_N^{N/2}=-1$$

因此当$N=2^p$时，ω_N^{jk}只有$n/2$个不同的值，所以可以利用这个性质，将求和的式子分为两部分：

$$c_j = \sum_{k=0}^{N/2-1} x_k \omega_N^{jk} + \sum_{k=0}^{N/2-1} x_{N/2+k}\omega_N^{j(N/2+k)} = \sum_{k=0}^{N/2-1}[x_k + (-1)^j x_{N/2+k}]\omega_N^{jk}$$

分别奇数项和偶数项进行考察，得到：

$$c_{2j} = \sum_{k=0}^{N/2-1}(x_k + x_{N/2+k})\omega_{N/2}^{jk},$$

$$c_{2j+1} = \sum_{k=0}^{N/2-1}(x_k - x_{N/2+k})\omega_N^{k}\omega_{N/2}^{jk}$$

这样对每个点反复进行二分就可以得到FFT算法了。

实际计算的时候，为了减少运算量，可以将k, j用二进制表示，则

$$c_j = c(j_2 j_1 j_0), x_k = x(k_2 k_1 k_0)\ ,$$

引入如下记号简化算式：

$$A_0(k_2k_1k_9) = x(k_2k_1k_0),$$

$$A_1(k_1k_0j_0) = \sum_{k_2=0}^{1} A_0(k_2k_1k_0)\omega^{j_0(k_2k_1k_0)},$$

$$A_2(k_0k_1j_0) = \sum_{k_1=0}^{1} A_1(k_1k_0j_0)\omega^{j_1(k_1k_00)},$$

$$A_3(j_2j_1j_0) = \sum_{k_0=0}^{1} A_2(k_0j_1j_0)\omega^{j_2(k_000)},$$

如此类推，从$A_0(k) = x_k$一直计算到$A_p(j) = c_j$，就可以得出所有的A，这就是要求的系数。

数值实验

根据迭代公式

$$\begin{cases} A_q(k2^q + j) = A_{q-1}(k2^{q-1} + j) + A_{q-1}(k2^{q-1} + j + 2^{p-1}) \\ A_q(k2^q + j + 2^{q-1}) = [A_{q-1}(k2^{q-1} + j) - A_{q-1}(k2^{q-1} + j + 2^{p-1})]\omega^{k2^{q-1}} \end{cases}$$

编写matlab代码。

```
function [c] = FFT(A)
    N = 1024; % 采样点数量
    p = 10;
    W = exp(i*2*pi/N);

    for q=1:1:p
        T = A; % A(q-1)
        for k=0:1:2^(p-q)-1
            for j=0:1:2^(q-1)-1
                index1 = k*2^(q-1);
                temp1 = T(k*2^(q-1) + j + 1); % 运算前的A
                temp2 = T(k*2^(q-1) + j + 2^(p-1) + 1);
                A(k*2^q + j + 1) = temp1 + temp2;
                A(k*2^q + j + 2^(q-1) + 1) = (temp1 - temp2) * (W^index1);
            end
        end
    end
    c = A;
end
```

在编写代码的过程中，特别需要注意要使用一个临时变量储存$A(q-1)$，因为在j, k循环的过程中，会不断更新$A(p)$，而$A(p)$的计算是基于$A(p-1)$的。

结果分析

运行代码后，得出如果结果：

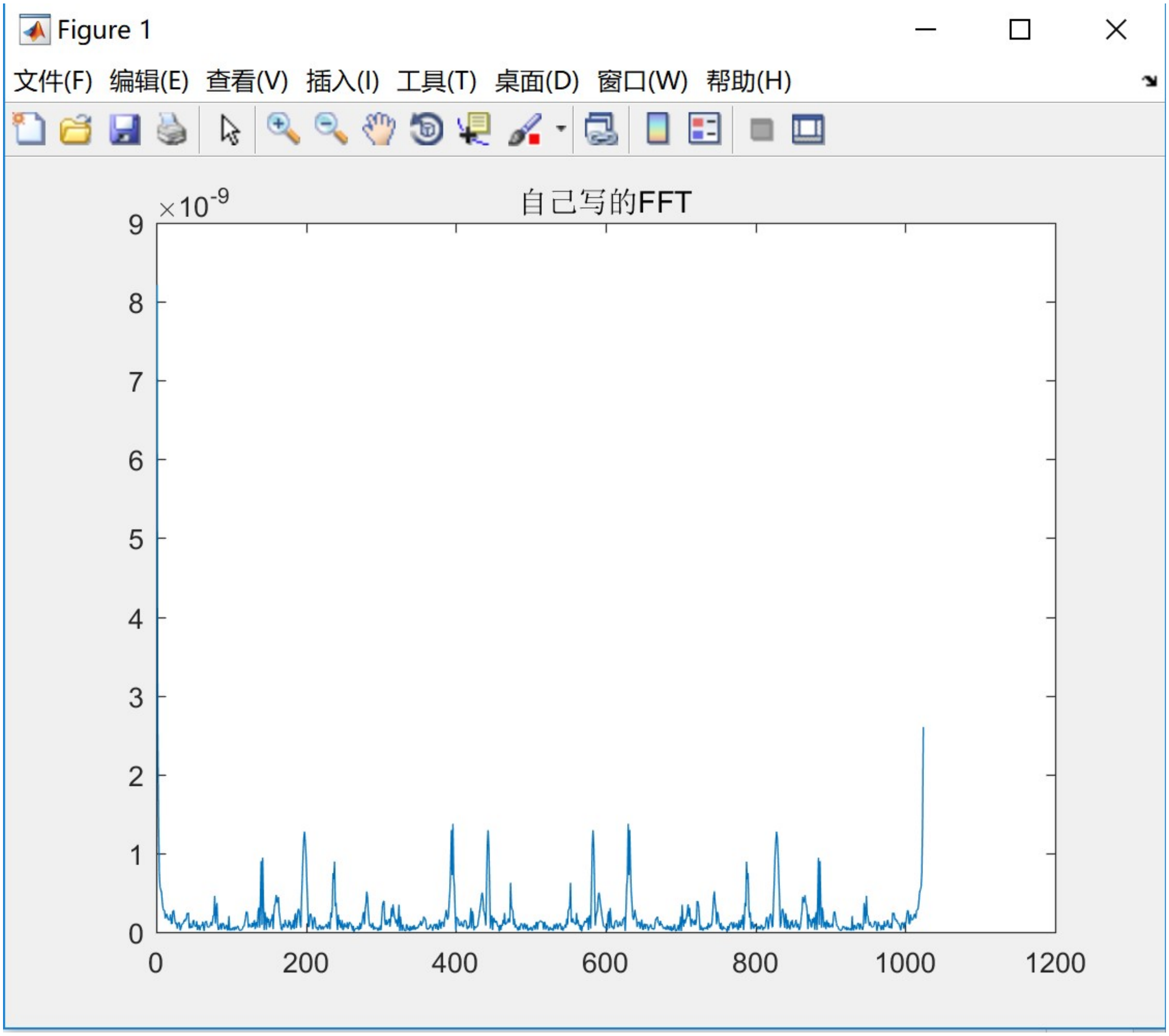
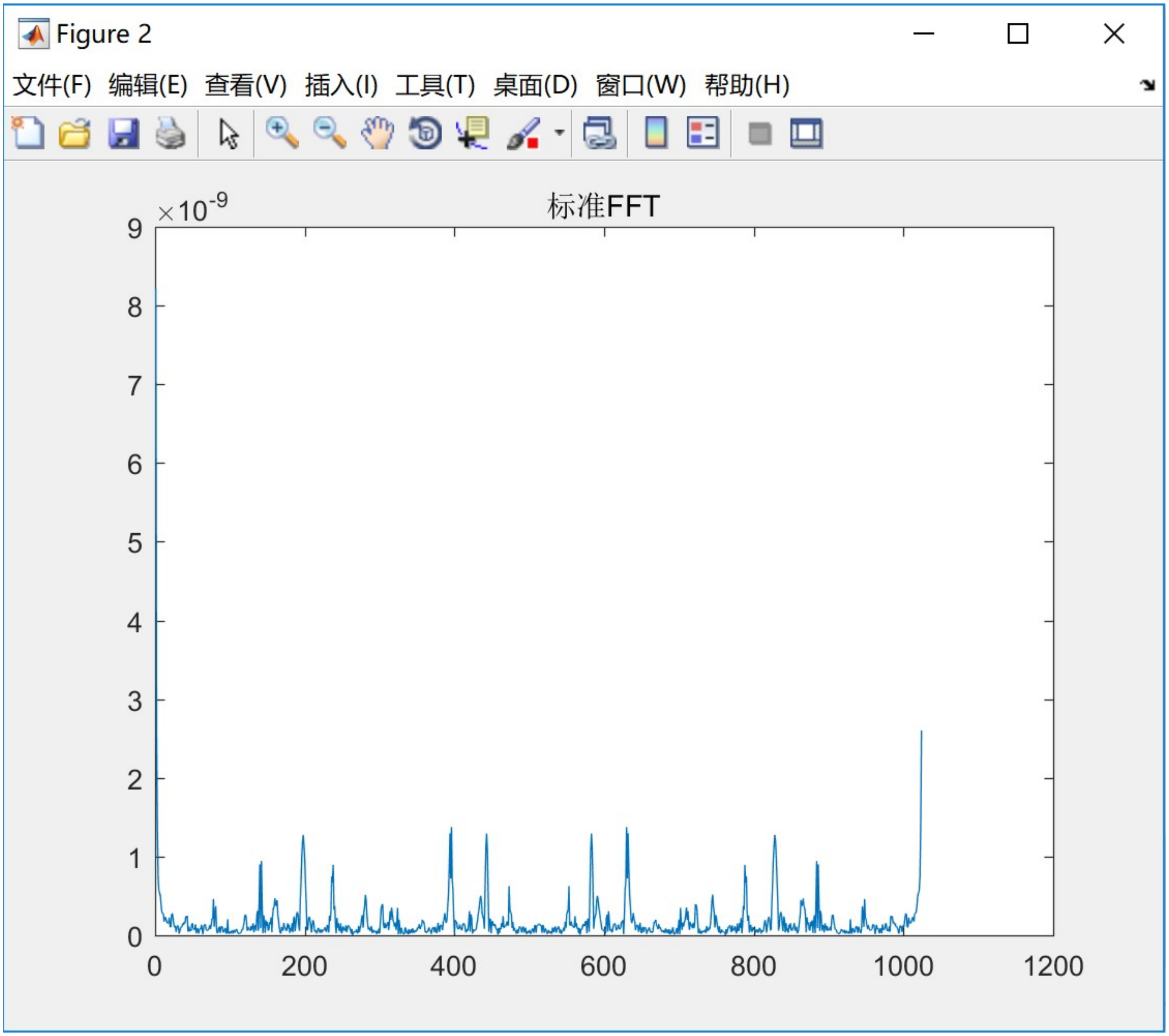


Figure 1
文件(F) 编辑(E) 查看(V) 插入(I) 工具(T) 桌面(D) 窗口(W) 帮助(H)
自己写的FFT
×10-9
9
8
7
6
5
4
3
2
1
0
0
200
400
600
800
1000
1200



分析：上面的图是经过自己写的FFT算法得出的系数，下面的图是使用matlab自带的FFT方法求出的系数，对比两幅图基本一致，可以基本判断FFT算法实现正确。

内容5

请采用复合梯形公式与复合辛普森公式，计算 sin(x)/x 在[0, 1]范围内的积分。采样点数目为 5、9、17、33。

问题描述

本题要解决的是一个数值积分问题，对于一些难于求积的函数，使用**牛顿-莱布尼茨**公式显然是不科学的。因此对于这类问题，我们可以从**积分中值定理**出发，使用矩形或梯形的面积去近似积分值。

算法设计

这道题要求用到**复合梯形公式**和**复合辛普森公式**，统称为**复合求积法。**这种通过把积分区间细分成若干个子区间（通常是等分），再在每个子区间上使用低阶求积公式，从而提高了计算精度。

1.复合梯形公式

该公式核心思想是对于细分后的每一个子区间，使用梯形公式求积。假设将区间$[a, b]$等分为n个子区间，分点$x_k = a + kh$, $h = \frac{b-a}{n}$, $k = 0, 1, \ldots, n$，公式的原理就是对于每一块小梯形而言，上底为$f(x_k)$，下底为$f(x_{k+1})$，高为h。

复合梯形公式如下：

$$I = \int_a^b f(x)dx = \sum_{k=0}^{n-1} \int_{x_k}^{x_{k+1}} f(x)dx = \frac{h}{2} \sum_{k=0}^{n-1} [f(x_k) + f(x_{k+1})] \text{ ,}$$

而在计算机编程时，通常使用的是下面这个形式：

$$T_n = \frac{h}{2}\sum_{k=0}^{n-1}[f(x_k)+f(x_{k+1})] = \frac{h}{2}\left[f(a)+2\sum_{k=1}^{n-1}f(x_k)+f(b)\right]$$

2.复合辛普森公式

复合辛普森公式的思路和复合梯形公式的思路一致，差别在于它的低阶求积公式使用了辛普森公式。辛普森公式是经过加权改造的梯形公式，集中在中间的点的权值更高，因此相比起简单的梯形公式，辛普森公式拥有更高的数值精度。

将区间&[a,b]&分为&n&等分，在每个字区间$[x_k, x_{k+1}]$上采用辛普森公式，记$x_{k+1/2} = x_k + \frac{1}{2}h$，得：

$$I = \int_a^b f(x)dx = \sum_{k=0}^{n-1}\int_{x_k}^{x_{k+1}} f(x)dx$$

$$= \frac{h}{6}\sum_{k=0}^{n-1}[f(x_k)+4f(x_{k+1/2})+f(x_{k+1})]$$

在计算机编程时，通常写成下列形式：

$$S_n = \frac{h}{6}\left[f(a)+4\sum_{k=0}^{n-1}f(x_{k+1/2})+2\sum_{k=1}^{n-1}f(x_k)+f(b)\right]$$

数值实验

1.复合梯形公式

根据上述推到的复合梯形公式，编写matlab代码。

```
% 复合梯形公式
function [result] = CompositeTrapezoid(a, b, n)
    if (b < a)
        c = b;
        b = a;
        a =c;
    end
    h = (b - a) / n; % 计算步长
    result = myFun(a) + myFun(b);
    for k = 1:n-1
        x = a + k * h;
        result = result + 2 * myFun(x);
    end
    result = (h / 2) * result;
end

function [y] = myFun(x)
    if (x == 0)
        y = 1;
    else
        y = sin(x) / x;
    end
end
```

2.复合辛普森公式

根据上述推导的复合辛普森公式，编写matlab代码。

```
% 复合辛普森公式
function [result] = CompositeSimpson(a, b, n)
    if (b < a)
        c = b;
        b = a;
        a = c;
    end
    h = (b - a) / n; % 计算步长
    result = myFun(a) + myFun(b);
    for k = 1:n-1
        x = a + k * h;
        result = result + 2 * myFun(x);
    end
    for k = 1:n
        x = a + (k-1)*h + 1/2 * h;
        result = result + 4 * myFun(x);
    end
    result = result * (h / 6);
end

function [y] = myFun(x)
    if (x == 0)
```

```
        y = 1;
    else
        y = sin(x) / x;
    end
end
```

结果分析

采样点的数目分别为5，9，17，33，分别运行测试代码，得出结果如下图：

1. 复合梯形公式

```
>> test_CompositeTrapezoid
n = 5:
   0.945078780953402

n = 9:
   0.945773188549752

n = 17
   0.945996225242376

n = 33
   0.946060023888043
```

2. 复合辛普森公式

```
>> test_CompositeSimpson
n = 5:
   0.946083168838073

n = 9:
   0.946083079742053

n = 17
   0.946083071103489

n = 33
   0.946083070419036
```

分析：对于同一种方法来讲，采样点越多，积分值越准确；对比两种方法来说，复合辛普森公式更加准确，原因是它对中间的值分配更高的权重，使得数值积分更加准确。

内容6

请采用下述方法，求解常微分方程初值问题 y’ =y-2x/y，y(0)=1，计算区间为[0, 1]，步长为 0.1。

（1）前向欧拉法。
（2）后向欧拉法。
（3）梯形方法。
（4）改进欧拉方法。

问题描述

本题涉及到的是常微分方程初值问题的求解方法。求解常微分方程可以分为两类解法：（1）解析方法；（2）**数值解法。**前者仅限于对特殊的方程进行求解，而后

者可以应用于一般方程的求解。

常微分方程初值问题的一般形式是：

$$y' = f(x,y), x \in [x_0, b],$$
$$y(x_0) = y_0 \text{ ,}$$

而我们需要做的就是求解出$y = y(x)$。

算法设计

1.前向欧拉法

在xy平面上，微分方程$f(x,y) = y'$的解$y = y(x)$称作它的**积分曲线**。从几何角度出发，从初始点$P_0(x_0, y_0)$出发，沿该点切线方向$f(x_0, y_0)$推进到$x = x_1$上的一点P_1，然后再从$P1$点推进到$x = x_2$上的一点P_2，一直推进到P_n。

对于相邻的两个点P_n和P_{n+1}，有如下关系：

$$\frac{y_{n+1} - y_n}{x_{n+1} - x_n} = f(x_n, y_n) \text{ ,}$$

通过变换得到：

$$y_{n+1} = y_n + hf(x_n, y_n) \text{ ,}$$

这个方法就称为**欧拉方法**（前向欧拉方法）。实际上这是对常微分方程中的导数用**均差**来近似。若初值y_0已知，则可以通过递推算出y_n

2.后向欧拉法

如果对微分方程从x_0到x_{n+1}积分，可以得到：

$$y(x_{n+1}) = y(x_n) + \int_{x_n}^{x_{n+1}} f(t, y(t))dt \text{ ,}$$

对右端的积分使用**右矩形公式**近似，得到：

$$y_{n+1} = y_n + hf(x_{n+1}, y_{n+1}) \text{ ,}$$

这种方法称为**后退的欧拉法**（后向欧拉法）。由于等式右边含有y_{n+1}，所以后向欧拉法是**隐式的。**

隐式方法计算的时候，要使用迭代法，逐步显式化。首先使用欧拉公式提供初值，再进行迭代。

$$y_{n+1}^{(0)} = y_n + hf(x_n, y_n)$$
$$y_{n+1}^{(k+1)} = y_n + hf(x_{n+1}, y_{n+1}^{(1)})$$

3.梯形方法

在欧拉方法的基础上，对积分右端近似的方法进行改进，使用**梯形求积公式**近似，可以得到精度更高的计算公式，主要形式为：

$$y_{n+1} = y_n + \frac{h}{2}\left[f(x_n, y_n) + f(x_{n+1}, y_{n+1})\right] \text{ ,}$$

这种方法被称为**梯形方法**，由于等式右端同样含有y_{n+1}，因此这种方法也是**隐式的。**求解的方法同后向欧拉类似，由欧拉方法给出初值，再逐步显式化。

$$y_{n+1}^{(0)} = y_n + hf(x_n, y_n)$$
$$y_{n+1}^{(k+1)} = y_n + \frac{h}{2}\left[f(x_n, y_n) + f(x_{n+1}, y_{n+1}^{(k)})\right], k = 0, 1, 2, \ldots$$

4.改进欧拉方法

梯形方法给我们提供了一种获得更高精度的算法，但是其计算量是庞大的，因为每步迭代都需要重新计算$f(x,y)$的值。这里给出了一种思路：先使用欧拉方法求得一个**初步的近似值**$\overline{y}_{n+1}$\，称为**预测值**，但是这个预测值的精度可能会很差，因此需要校正。这里使用**梯形公式**对其进行校正，得到一个**校正值**y_{n+1}。这种**预测-校正系统**称为**改进的欧拉公式**：

$$\begin{cases} \text{预测}\overline{y}_{n+1} = y_n + hf(x_n, y_n) \text{ ,} \\ \text{校正}y_{n+1} = y_n + \dfrac{h}{2}\left[f(x_n, y_n) + f(x_{n+1}, \overline{y}_{n+1})\right] \end{cases}$$

数值实验

1.前向欧拉法

根据上述推导的公式，编写matlab代码：

```
% 前向欧拉方法
function [x,y] = Euler(a, b, y0, h)
    x(1) = a;
    y(1) = y0;
```

```
    n = (b - a) / h;
    for i = 1:n
        x(i+1) = x(i) + h;
        y(i+1) = y(i) + h * myFun(x(i),y(i));
    end
end

function f = myFun(x, y)
    f = y - 2 * x / y;
end
```

2.后向欧拉法

根据上述推导的公式，编写matlab代码：

```
% 后向欧拉方法
function [x,y] = EulerBackward(a, b, y0, h)
    n = (b - a) / h;
    x = zeros(1, n);
    x(1) = a;
    y(1) = y0;

    for i = 1:n
        x(i+1) = x(i) + h;
        yt = y(i) + h * myFun(x(i), y(i)); % 使用欧拉公式给出迭代初值
        finished = 0; % 初始化
        while ~finished
            y(i+1) = y(i) + h * myFun(x(i+1), yt);
            finished = (abs(y(i+1) - yt) < 0.000001);
            yt = y(i+1);
        end
    end
end

function f = myFun(x, y)
    f = y - 2 * x / y;
end
```

3.梯形方法

根据上述推导的公式，编写matlab代码：

```
% 梯形方法
function [x,y] = Trapezoid(a, b, y0, h)
    n = (b - a) / h;
    x = zeros(1, n);
    y(1) = y0;
    x(1) = a;

    for i = 1:n
        x(i+1) = x(i) + h;
        yt = y(i) + h * myFun(x(i), y(i)); % 使用欧拉方法提供迭代初值
        finished = 0; % 初始化
        while ~finished
            y(i+1) = y(i) + (h/2) * (myFun(x(i), y(i)) + myFun(x(i+1), yt));
            finished = (abs(y(i+1) - yt) < 0.000001);
            yt = y(i+1);
        end
    end
end

function f = myFun(x, y)
    f = y - 2 * x / y;
end
```

4.改进欧拉方法

根据上述推导的公式，编写matlab代码：

```
% 改进欧拉方法
function [x,y] = EulerImproved(a, b, y0, h)
    n = (b - a) / h;
    x = zeros(1, n);
    y(1) = y0;
    x(1) = a;

    for i = 1:n
        x(i+1) = x(i) + h;
        yt = y(i) + h * myFun(x(i), y(i)); % 预测
        y(i+1) = y(i) + (h/2) * (myFun(x(i), y(i)) + myFun(x(i+1), yt)); % 校正
    end
```

```
end

function f = myFun(x, y)
    f = y - 2 * x / y;
end
```

结果分析

运行上述各方法的代码后，得出结果如下：
前向欧拉方法结果：

```
>> test_Euler
前向欧拉方法
x = 0.100000 ,y = 1.100000
x = 0.200000 ,y = 1.191818
x = 0.300000 ,y = 1.277438
x = 0.400000 ,y = 1.358213
x = 0.500000 ,y = 1.435133
x = 0.600000 ,y = 1.508966
x = 0.700000 ,y = 1.580338
x = 0.800000 ,y = 1.649783
x = 0.900000 ,y = 1.717779
x = 1.000000 ,y = 1.784771
```

后向欧拉方法结果：

```
>> test_EulerBackward
后向欧拉方法
x = 0.100000,y = 1.090738
x = 0.200000,y = 1.174076
x = 0.300000,y = 1.251249
x = 0.400000,y = 1.323094
x = 0.500000,y = 1.390178
x = 0.600000,y = 1.452870
x = 0.700000,y = 1.511377
x = 0.800000,y = 1.565768
x = 0.900000,y = 1.615978
x = 1.000000,y = 1.661808
```

梯形方法结果：

梯形方法
x = 0.100000 ,y = 1.095656
x = 0.200000 ,y = 1.183594
x = 0.300000 ,y = 1.265441
x = 0.400000 ,y = 1.342323
x = 0.500000 ,y = 1.415058
x = 0.600000 ,y = 1.484267
x = 0.700000 ,y = 1.550429
x = 0.800000 ,y = 1.613929
x = 0.900000 ,y = 1.675083
x = 1.000000 ,y = 1.734151

改进欧拉方法结果：

```
>> test_EulerImproved
改进欧拉方法
x = 0.100000,  y = 1.095909
x = 0.200000,  y = 1.184097
x = 0.300000,  y = 1.266201
x = 0.400000,  y = 1.343360
x = 0.500000,  y = 1.416402
x = 0.600000,  y = 1.485956
x = 0.700000,  y = 1.552514
x = 0.800000,  y = 1.616475
x = 0.900000,  y = 1.678166
x = 1.000000,  y = 1.737867
```

由上述几种方法可以看出，前向欧拉方法和后向欧拉方法的误差是比较大的，改进的欧拉方法精度比欧拉法要好，而在四种方法中，梯形方法拥有更高的精度。但是从计算复杂度而言，欧拉法是比较简单的。考虑到数值稳定性的问题，有时可以使用后向欧拉方法来保证数值的稳定性。综合计算复杂度和精度而言，个人认为改进欧拉方法是一个比较好的折中方案。