《数值计算方法》实验

实验报告

题目	Matlab 下实现一些算法
姓名	廖蕾
学号	16340135
班级	16 级软件工程教务 2 班

一．实验环境：

在 win10 操作系统下，使用 Matlab R2017a 完成的。

二．实验内容与完成情况：

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。

1) 线性插值：

输入的量：

选取的两点 x1, x2，与其对应的 y1, y2，还有要计算的 x3;

输出的量：

X3 对应的 y3 的值。

算法描述：

使用拉格朗日插值中的线性插值的点斜式，构造一个式子，计算函数的值。

原理是：$L_1(x) = y_k + \dfrac{y_{k+1} - y_k}{x_{k+1} - x_k}(x - x_k)$

以下是完整的代码实现：

```
function[result] = linearInset(x1, y1, x2, y2, x3)
    result = y2 + (y2 - y1) / (x2 - x1) * (x3 - x1);
```

数值实验：

在命令行中运行得到结果：

Sin(0.35) = 0.3617。

```
>> [result] = linearInset(x1, y1, x2, y2, x3)

result =

    0.3617

fx >> 
```

2) 二次插值

输入的量：
选取的两点 x1, x2, x3， 与其对应的 y1, y2, y3， 还有要计算的 x4;

输出的量：
X4 对应的 y4 的值。

算法描述：
使用拉格朗日插值中的线性插值的点斜式，构造一个式子，计算函数的值。
原理是：

$$L_2(x) = y_{k-1}\frac{(x-x_k)(x-x_{k+1})}{(x_{k-1}-x_k)(x_{k-1}-x_{k+1})} + y_k\frac{(x-x_{k-1})(x-x_{k+1})}{(x_k-x_{k-1})(x_k-x_{k+1})} + y_{k+1}\frac{(x-x_{k-1})(x-x_k)}{(x_{k+1}-x_{k-1})(x_{k+1}-x_k)}$$

以下是完整的代码实现：
```
function[result] = quadraticInset(x1, y1, x2, y2, x3, y3, x4)
    temp1 = (x4 - x2)*(x4 - x3)/(x1 - x2)/(x1 - x3);
    temp2 = (x4 - x1)*(x4 - x3)/(x2 - x1)/(x2 - x3);
    temp3 = (x4 - x1)*(x4 - x2)/(x3 - x1)/(x3 - x2);
    result = y1 * temp1 + y2 * temp2 + y3 * temp3;
```

数值实验：
在命令行中运行得到结果：
Sin(0.35) = 0.3429。

```
>> [result] = quadraticInset(x1, y1, x2, y2, x3, y3, x4);
>> result

result =

    0.3429

>>
```

3) 三次插值
输入的量：
选取的两点 x1, x2, x3, x4， 与其对应的 y1, y2, y3, y4， 还有要计算的 x5;

输出的量：
x5 对应的 y5 的值。

算法描述：
使用拉格朗日插值中的线性插值的点斜式，构造一个式子，计算函数的值。
原理是：

$$L_3(x) = y_{k\text{-}1}\frac{(x-x_k)(x-x_{k+1})(x-x_{k+2})}{(x_{k-1}-x_k)(x_{k-1}-x_{k+1})(x_{k\text{-}1}-x_{k+2})} + y_k\frac{(x-x_{k-1})(x-x_{k+1})(x-x_{k+2})}{(x_k-x_{k-1})(x_k-x_{k+1})(x_k-x_{k+2})}$$
$$+\, y_{\text{k+1}}\frac{(x-x_{k-1})(x-x_k)(x-x_{k+2})}{(x_{k+1}-x_{k-1})(x_{k+1}-x_k)(x_{k+1}-x_{k+2})} + y_{\text{k+2}}\frac{(x-x_{k-1})(x-x_k)(x-x_{k+1})}{(x_{k+2}-x_{k-1})(x_{k+2}-x_k)(x_{k+2}-x_{k+1})}$$

以下是完整的代码实现：

```
function[result] = threeInset(x1, y1, x2, y2, x3, y3, x4, y4, x5)
    temp1 = (x5 - x2)*(x5 - x3)*(x5 - x4)/(x1 - x2)/(x1 - x3)/(x1 - x4);
    temp2 = (x5 - x1)*(x5 - x3)*(x5 - x4)/(x2 - x1)/(x2 - x3)/(x2 - x4);
    temp3 = (x5 - x1)*(x5 - x2)*(x5 - x4)/(x3 - x1)/(x3 - x2)/(x3 - x4);
    temp4 = (x5 - x1)*(x5 - x2)*(x5 - x3)/(x4 - x1)/(x4 - x2)/(x4 - x3);
    result = y1 * temp1 + y2 * temp2 + y3 * temp3 + y4 * temp4;
```

数值实验：

在命令行中运行得到结果：

Sin(0.35) = 0.3429。

```
>> [result] = threeInset(x1, y1, x2, y2, x3, y3, x4, y4, x5)

result =

    0.3429

>>
```

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。

1) 二分法：

输入的量：求根区间范围 a 与 b；

输出的量：计算结果 result；

算法描述：

考察有根区间[a, b]，取中点 x0 = (a+b)/2 将他分为两半，假设中电 x0 不是 f(x)的零点，然后进行根的搜索，将区间减小一半，当区间长度小于 5*10^(-7)时，停止减小区间，将此时区间中点作为需要求的零点。设此时的函数为： $f(x)=x^2-115$

具体代码实现如下：

```
    while (x2 - x1) > prec
```

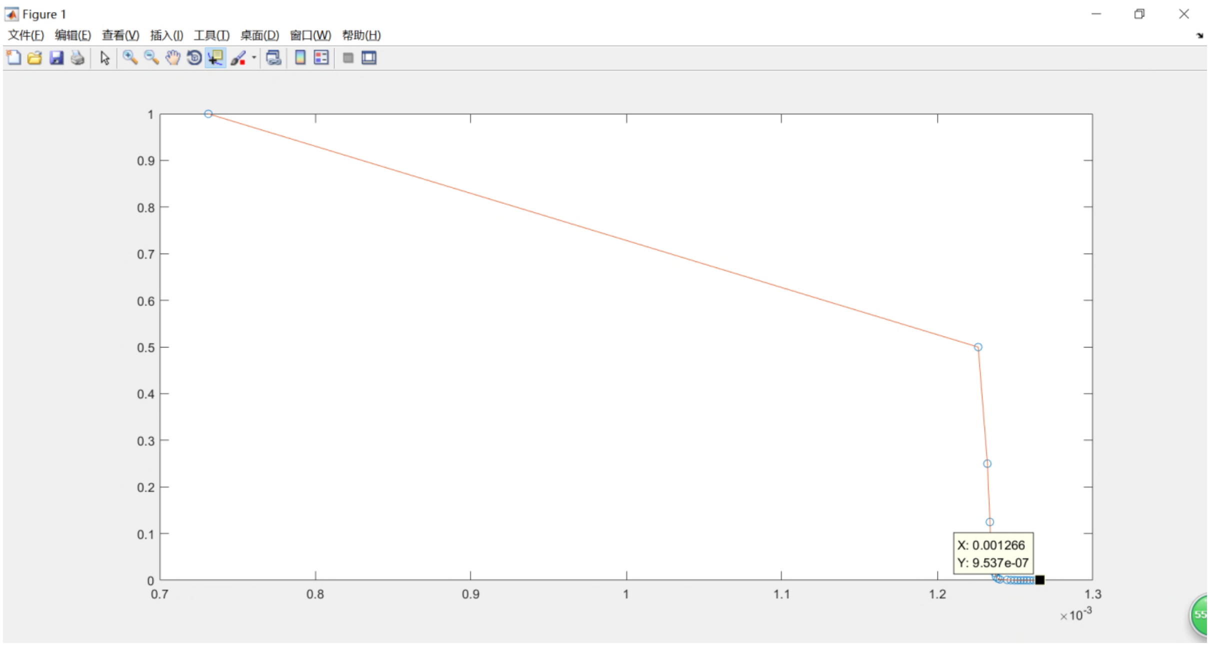
```
            temp = (x2 + x1) / 2;
            if (temp * temp - 115) > 0
            x2 = temp;
        else
            x1 = temp;
        end
    end
    result = (x1 + x2) / 2;
```

数值实验：

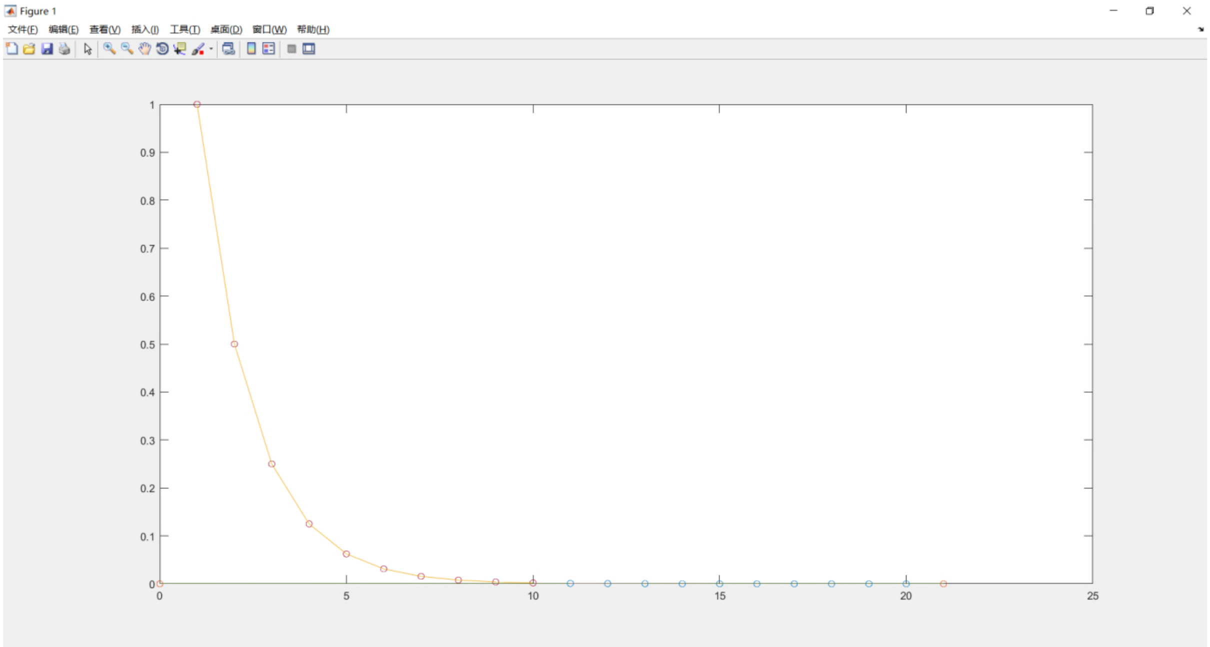
解出结果：result = 10.723805189132690

```
>> [result] = dichotomy(a, b);
>> result

result =

  10.723805189132690

>>
```

横坐标为计算时间时的收敛精度曲线如下：



横坐标为迭代步数时的收敛精度曲线如下：



2) 牛顿法

输入的量：给定的初值 x0；

输出的量：计算结果 result 以及迭代次数 count；

算法描述：

在这里的迭代中，原方程为 $f(x)=x^2-115$，所以使用牛顿法的迭代方程为：

$x_{k+1}=\dfrac{x_k}{2}+\dfrac{115}{2x_k}$。这里我取的初值为 1。

具体实现代码如下：

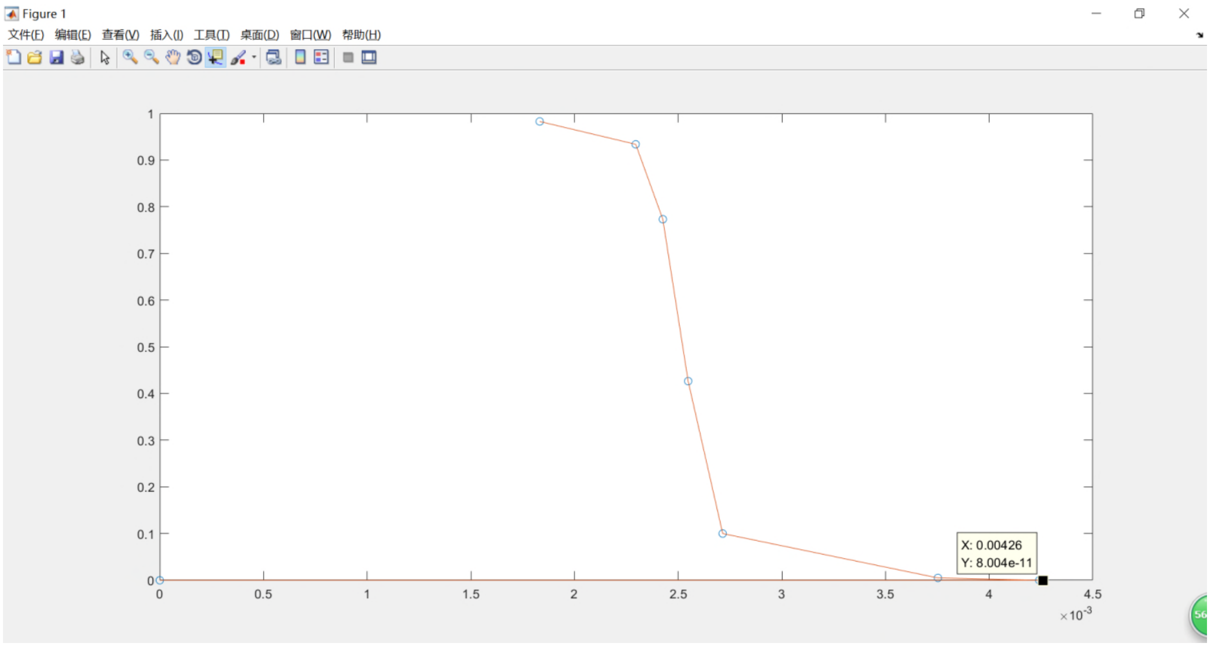
```
while eps > prec
      x1 = x0/2 + 115/(2 * x0);
      if abs(x1) < 1
            eps = abs(x1 - x0);
      else
            eps = abs(x1 - x0) / abs(x1);
      end
      x0 = x1;
      i = i + 1;
end
num = i;
result = x1;
```

数值实验

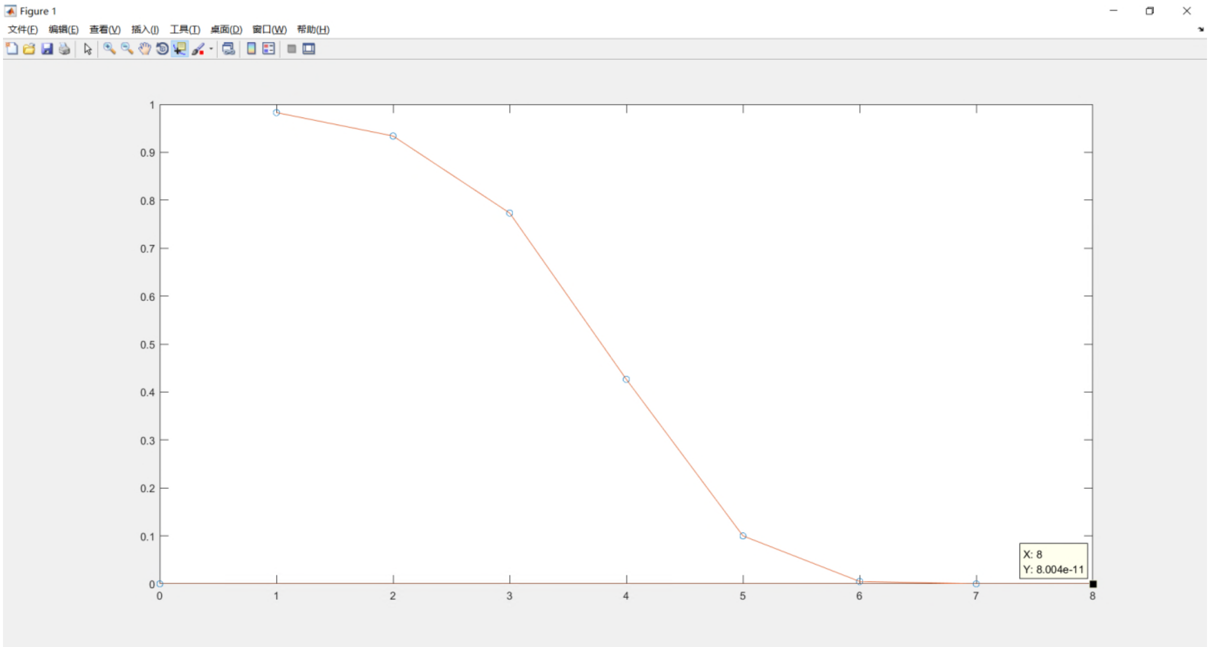
解出结果：result = 10.723805294763608

```
>> [result, count] = newton(x0);
>> result

result =

  10.723805294763608

>> count

count =

     9

>>
```

横坐标为计算时间时的收敛精度曲线如下：



横坐标为迭代步数时的收敛精度曲线如下：



3) 简化牛顿法：

输入的量：所取的初值 x0；

输出的量：计算结果 result 以及迭代次数 count；

算法描述：

本算法与牛顿法不同的地方在于，x 每次迭代时的方程为：

$x_{k+1} = x_k - Cf(x_k), C = \dfrac{1}{f'(x_0)}$。由于简化牛顿法主要时简化了牛顿法繁琐的计算过程，没能很好的解决如果初始值不在所要的结果附近时的收敛性，所以这里取初值 x0=20 计算。

具体实现代码如下：

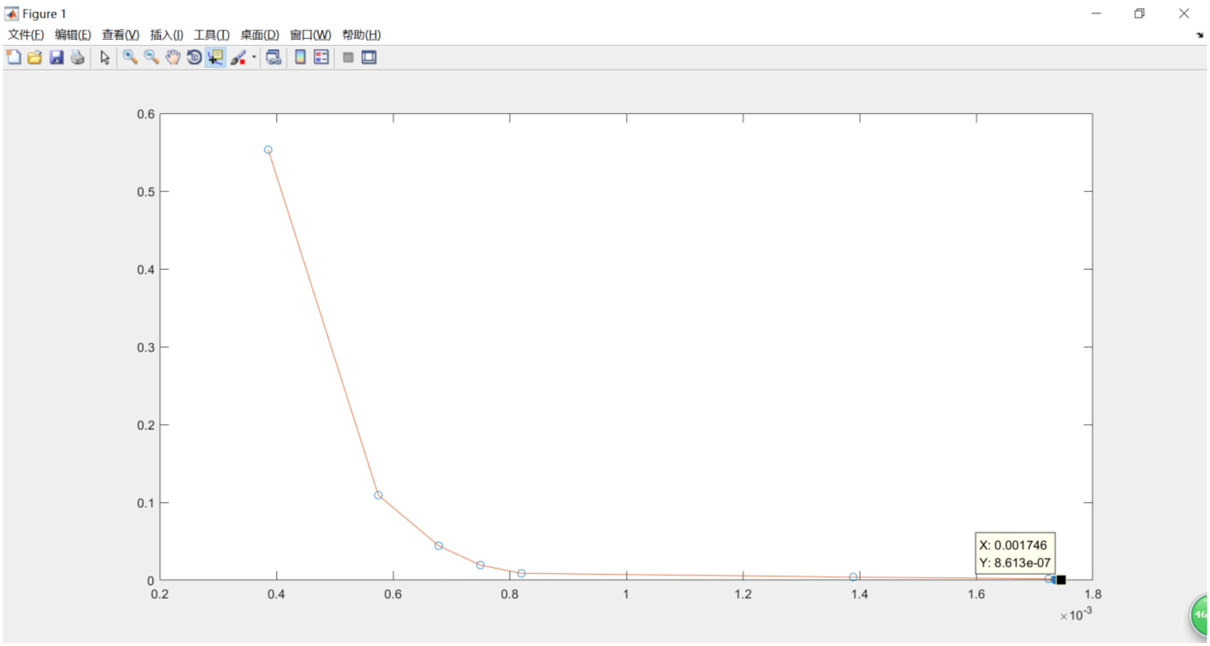
```
while eps > prec
    x1 = x0 - C * x0 * x0 + C * 115;
    if abs(x1) < 1
        eps = abs(x1 - x0);
    else
        eps = abs(x1 - x0) / abs(x1);
    end
    i = i + 1;
    x0 = x1;
end
num = i;
result = x1;
```

数值实验：

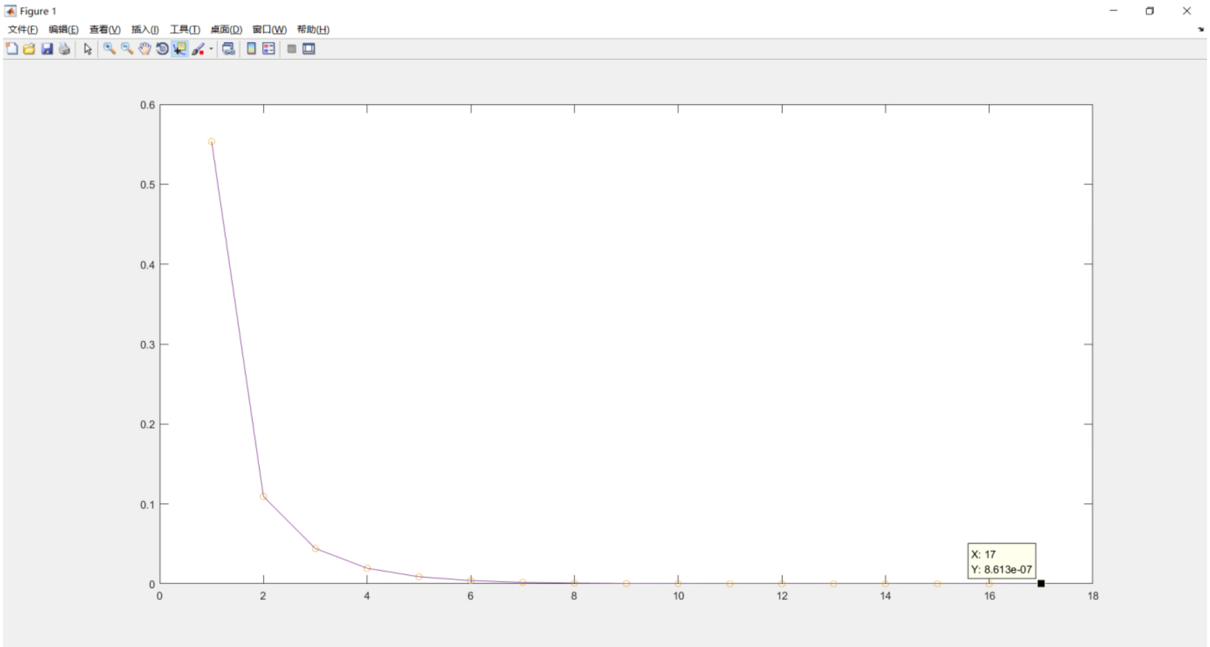
解出结果：result = 10.723813284151365；

```
>> [result, count] = newtonDec(x0);
>> result

result =

  10.723813284151365

>> count

count =

    18

fx >> |
```

横坐标为计算时间时的收敛精度曲线如下：



横坐标为迭代步数时的收敛精度曲线如下：



4) 弦截法

输入的量：初值 x0 与 x1；

输出的量：计算结果 result 以及迭代步数 count；

算法描述：

弦截法的时在去线上取两点连成弦，然后通过延长这条弦与横坐标交点为下一次迭代的点。迭代方程如下：

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

我这里选取的初值为 x0=15，x1=20。

具体实现代码如下：

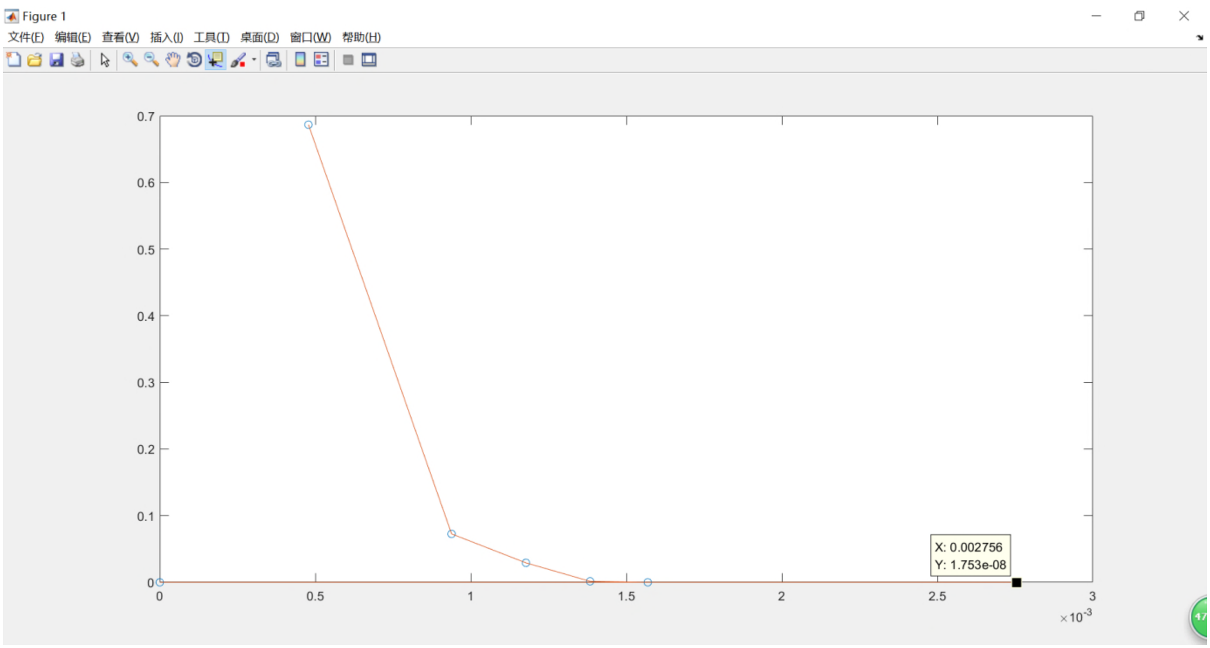
```
while eps > prec
	x2 = x1-(x1*x1 - 115)/(x1*x1 - x0*x0)*(x1-x0);
	if abs(x2) < 1
		eps = abs(x2 - x1);
	else
		eps = abs(x2 - x1) / abs(x2);
	end
	i = i + 1;
	x0 = x1;
	x1 = x2;
end
result = x2;
num = i;
```

数值实验：

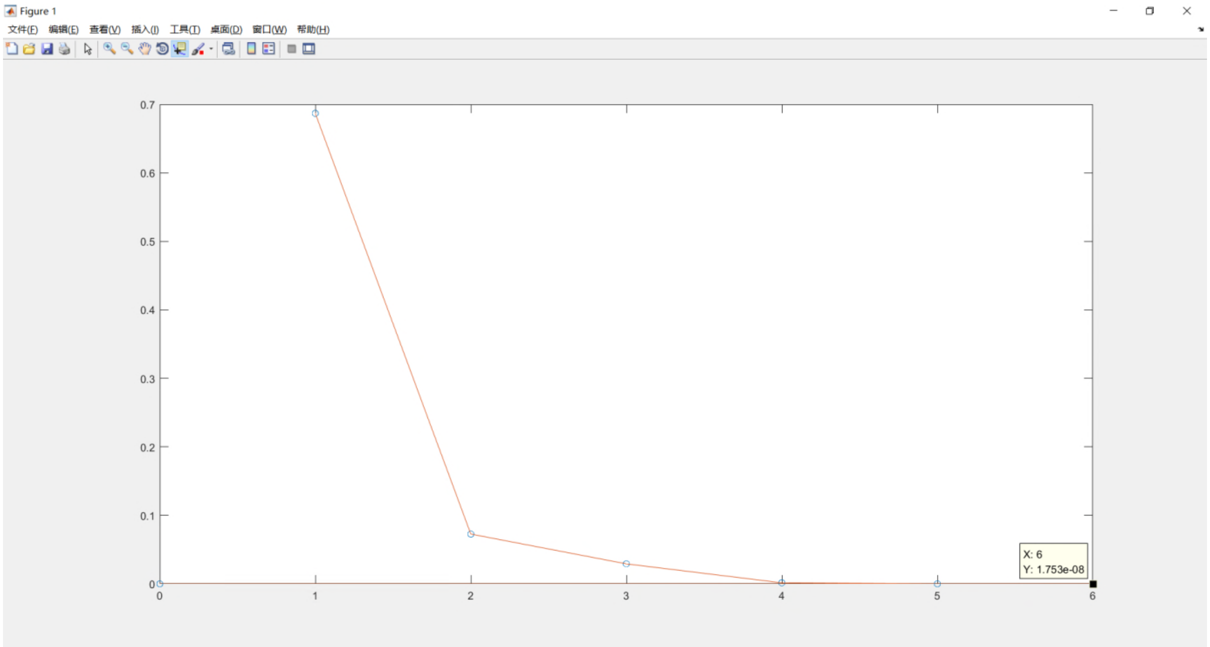
在这个初值下，计算得到结果为：result = 10.723805294765775；

```
>> [result, count]=stringCut(x0, x1)

result =

  10.723805294765775


count =

     7

fx >>
```

横坐标为计算时间时的收敛精度曲线如下：



横坐标为迭代步数时的收敛精度曲线如下：



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×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。

输入的量：方程组中的 A 与 b；

输出的量：解得的答案 x；

算法描述：根据递推的最小二乘法可知，当用最小二乘法算出维度比较小的超定方程组的解，之后可以用递推的思想快速的计算出高维超定方程组的最小二乘解。此时算法可以分为以下几步：

a) 选取低纬度的 A(m)以及 b(m);

b) 根据公式：$X(m) = \left(A^T(m)A(m)\right)^{-1} A^T(m)b(m)$可以计算得到 X 的一个最小二乘解。

记：$G(m) = A^T(m)A(m)$，$M(m+1) = 1 + a^T(m+1)G^{-1}(m)a(m+1)$

则递推的最小二乘法公式如下：

$$\begin{cases} x(m+1) = x(m) + G^{-1}(m)a(m+1)M^{-1}(m+1)[b(m+1) - a^T(m+1)x(m)] \\ G^{-1}(m+1) = G^{-1}(m) - G^{-1}(m)a(m+1)M^{-1}(m+1)a^T(m+1)G^{-1}(m) \end{cases}$$

具体代码实现如下：

现在 matlab 命令行中输入：

```
A = normrnd(0, 1, 10, 10000);
b = normrnd(0, 1, 10000, 1);
```

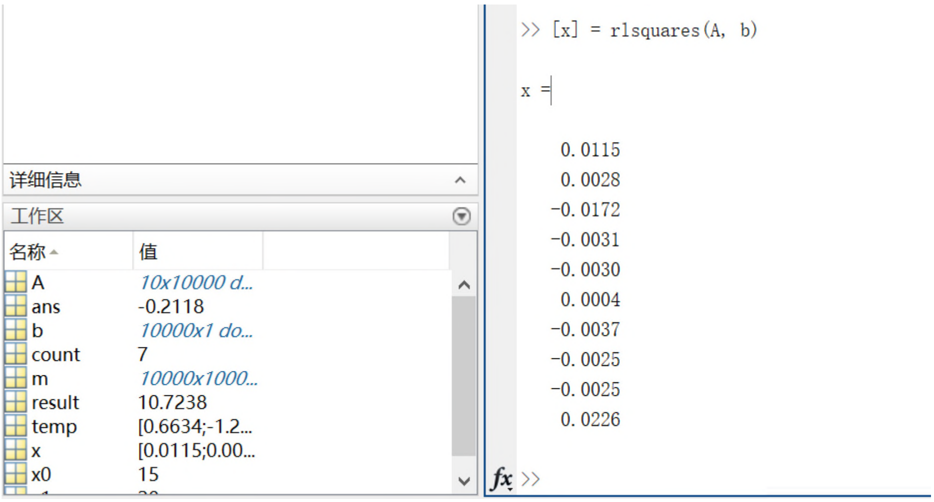
然后实现函数如下：

```
function[x] = rlsquares(A, b)
    temp = A(:, 1:10);
    tempB = b(1:10, 1);
    Gm = inv(temp'*temp);
    x = Gm*temp'*tempB;
```

```
    for i = 11:10000
        a = A(:, i);
        m = 1 + a'*Gm*a;
        x = x + Gm*a*inv(m)*(b(i)-a'*x);
        Gm = Gm-Gm*a*inv(m)*a'*Gm;
    end
end
```

数值实验：

在我本地生成的 A、b 下，计算得到结果：



运行时间如下：

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。

输入的量：无输入的量；

输出的量：处理前的信号 y，以及处理后的信号 F；

算法描述：

根据离散傅里叶变换，对于一个 x 信号来说有：

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N}nk} = \sum_{n=0}^{N-1} x_n W_N^{nk}$$

由周期性，我们有如下公式：

$$W_N^{n(N+k)} = e^{-i\times 2\pi n(k+N)/N} = e^{-i\times 2\pi nk/N} \times e^{-i\times 2\pi n} = e^{-i\times 2\pi nk/N} = W_N^{nk}$$

由于共轭性：

$$W_N^{n(N+k)} = e^{-i\times 2\pi n(k+\frac{N}{2})/N} = e^{-i\times 2\pi nk/N} \times e^{-i\times \pi n} = -W_N^{nk}$$

还有这个公式的规模性（等比例性）：

$$W_{N/m}^{n(\frac{k}{m})} = e^{-i\times 2\pi n(\frac{k}{m})/(\frac{N}{m})} = W_N^{nk}$$

当这里的 N 能被 2 整除时：

将x_n分为偶数序列X_{1n}和X_{2n}，则有：

$$F(k) = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} \times e^{-i\times\frac{2\pi 2nk}{N}} + \sum_{n=0}^{\frac{N}{2}-1} x_{2n-1} \times e^{-i\times\frac{2\pi(2n+1)k}{N}} = F1(k) + W_N^k \times F2(k)$$

所以根据共轭性有：

$$F\left(k + \frac{N}{2}\right) = F1 - W_N^k \times F2$$

这样就能实现递归了。

生成一段不同频率的正弦波信号，实现代码如下：

```
Fs = 100;      %频率
T = 1/Fs;      %时间
L = 1024;    %信号长度
t = (0:L-1)*T;
x = 5 + 7*sin(2*pi*15*t - 30*pi/180) + 3*sin(2*pi*40*t - 90*pi/180);
y = x + randn(size(t));          %添加噪声
```

算法实现部分：

在主函数中：

```
for k = 1 : N / 2
    [F(k), F(k+N/2)] = my_fft_ele(y, N, k);
end
```

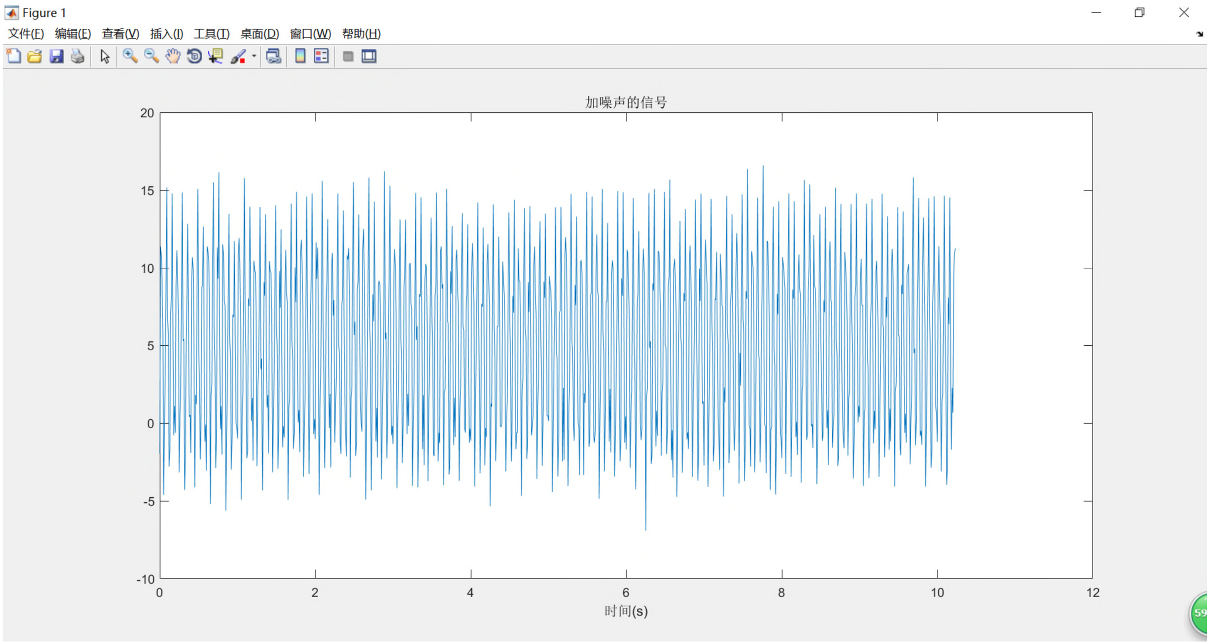
在递归函数中：

```
function [Fk, Fkn] = my_fft_ele(x, N, k)
    if N==1
        Fk=x;
        Fkn=x;
        return;
    else
        x1=x(1:2:N-1);%ÆæÊý
        x2=x(2:2:N);%Å¼Êý
```

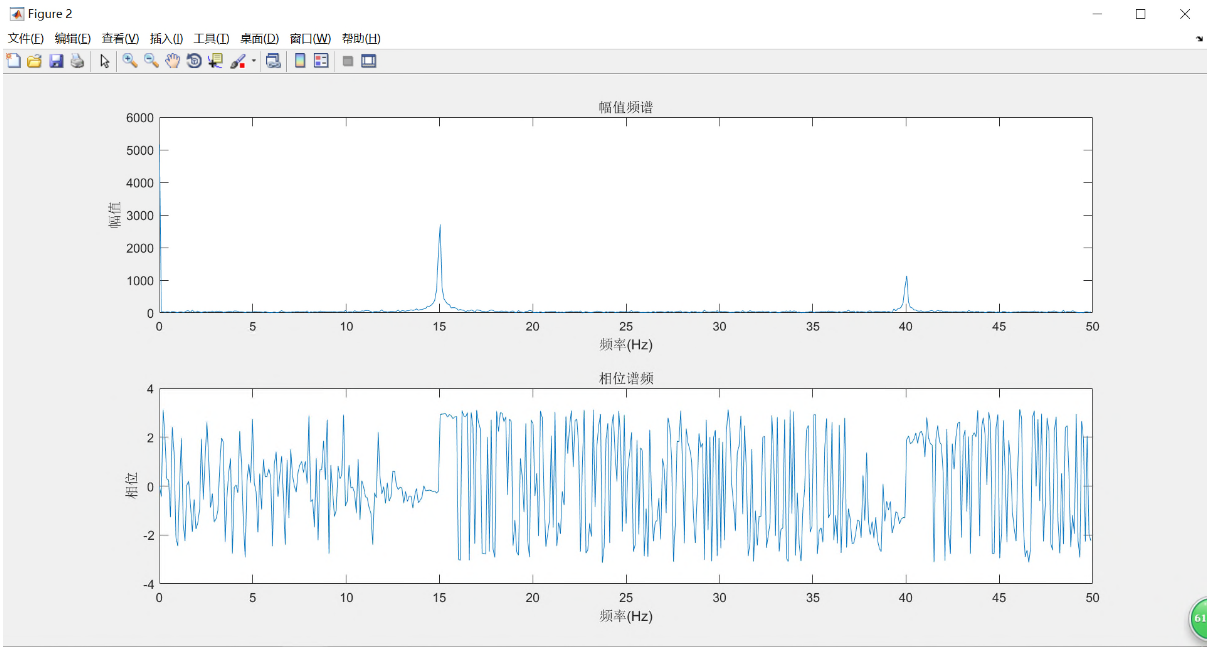
```
        F1=my_fft_ele(x1,N/2,k);
        F2=my_fft_ele(x2,N/2,k);
        Wkn=exp(-i*2*pi*(k-1)/N);
        Fk=F1+Wkn*F2;
        Fkn=F1-Wkn*F2;
    end
end
```

数值实验：

生成的原始信号波形如下：



快速傅里叶变换之后的波形如下：



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。

1) 复合梯形公式：

输入的量：区间范围 a 和 b，以及采样点数目 n；

输出的量：计算结果 result；

算法描述：

复合梯形公式时在梯形公式的基础上改变的，公式可以表示如下：

将区间[a, b]划分为 n 等分，分点 $x_k = a + kh$，$h = \dfrac{b-a}{n}$，$k = 0,1,\cdots,n$，

记：$T_n = \dfrac{h}{2}\sum_{k=0}^{n-1}[f(x_k) + f(x_{k+!})] = \dfrac{h}{2}[f(a) + 2\sum_{k=1}^{n-1} f(x_k) + f(b)]$

具体代码实现：

```
function[result] = comptra(a, b, n)
    h = (b - a)/n;
    sum = 0;
    for k = 1 : n-1
        sum = sum + sin(a+k.*h) / (a+k.*h);
    end
    result = (1 + 2*sum + sin(b)/b)*h/2;
end
```

数值实验：

当 n=5 时：result = 0.945078780953402；

当 n=9 时：result = 0.945773188549752；

当 n=17 时：result = 0.945996225242376；

当 n=33 时：result = 0.946060023888043；

```
>> n=[5, 9, 17, 33];
>> [result] = comptra(a, b, n(1))

result =

   0.945078780953402

>> [result] = comptra(a, b, n(2))

result =

   0.945773188549752

>> [result] = comptra(a, b, n(3))

result =

   0.945996225242376

>> [result] = comptra(a, b, n(4))

result =

   0.946060023888043

fx >>
```

2) 复合辛普森公式：

输入的量：区间范围 a 和 b，以及采样点数目 n；

输出的量：计算结果 result；

算法描述：

将区间[a, b]分为 n 等份，在每个子区间$[x_k, x_{k+1}]$上采用辛普森公式，若记$x_{k+1/2} = x_k + \frac{1}{2}h$，则得：

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

具体代码实现：

```
function[result] = compsinp(a, b, n)
    h = (b - a)/n;
    sum1 = 0;
    sum2 = 0;
    for i = 0 : n-1
        temp1 = a+(i+1/2).*h;
        sum1 = sum1 + sin(temp1)/temp1;
```

```
    end
    for j = 1 : n-1
        temp2 = a+j.*h;
        sum2 = sum2 + sin(temp2)/temp2;
    end
    result = h/6 * (1 + 4*sum1 + 2*sum2 + sin(b)/b);
end
```

数值实验：

当 n=5 时：result = 0.946083168838073；

当 n=9 时：result = 0.946083079742053；

当 n=17 时：result = 0.946083071103489；

当 n=33 时：result = 0.946083070419036；

```
>> [result] = compsinp(a, b, n(1))

result =

   0.946083168838073

>> [result] = compsinp(a, b, n(2))

result =

   0.946083079742053

>> [result] = compsinp(a, b, n(3))

result =

   0.946083071103489

>> [result] = compsinp(a, b, n(4))

result =

   0.946083070419036

>>
```

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。

1) 前向欧拉法：

输入的量：区间范围 a 与 b，步长 h

输出的量：得到的一组 x 的向量和一组 y 向量。

算法描述：
在 xy 平面上，微分方程的解 y=y(x)称作它的积分曲线，积分曲线上一点(x, y)的切线斜率等于函数 f(x, y)的值。如果按函数 f(x, y)在 xy 平面上建立一个方向场，那么，积分曲线上每一点的切线方向均与方向场在该点的方向相一致。
所以呢，相邻两点坐标之间有关系如下：

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

具体代码实现：
```
function[x, y] = forwardEular(a, b, h)
    x = a:h:b;
    y0 = 1;
    y(1) = y0;
    for n = 1 : length(x)-1
        y(n+1) = y(n) + h * (y(n) - 2 * x(n)/y(n));
    end
end
```

数值实验：
解得结果如下：

xn	yn	xn	Yn
0.1	1.100000000000000	0.6	1.508966253566332
0.2	1.191818181818182	0.7	1.580338237655217
0.3	1.277437833714722	0.8	1.649783431047711
0.4	1.358212599560289	0.9	1.717779347860087
0.5	1.435132918657796	1.0	1.784770832497982

```
>> [x, y] = forwardEular(a, b, h)

x =

  1 至 7 列

                   0   0.100000000000000   0.200000000000000   0.300000000000000   0.400000000000000   0.500000000000000   0.600000000000000

  8 至 11 列

   0.700000000000000   0.800000000000000   0.900000000000000   1.000000000000000


y =

  1 至 7 列

   1.000000000000000   1.100000000000000   1.191818181818182   1.277437833714722   1.358212599560289   1.435132918657796   1.508966253566332

  8 至 11 列

   1.580338237655217   1.649783431047711   1.717779347860087   1.784770832497982
```

2) 后向欧拉法

输入的量：区间范围 a 与 b，步长 h

输出的量：得到的一组 x 的向量和一组 y 向量。

算法描述：

在前向欧拉法的基础上修改迭代的公式如下：

$$y_{k+1} = y_k + hf(x_k, y_k);$$
$$y_{k+1} = y_k + hf(x_{k+1}, y_{k+1})$$

具体的代码实现如下：

```
function[x, y] = backEuler(a, b, h)
    x = a:h:b;
    x0 = 0;
    y0 = 1;
    y(1) = y0;
    for n = 1 : length(x)-1
        y(n+1) = y(n) + h * (y(n) - 2 * x(n)/y(n));
        y(n+1) = y(n) + h * (y(n+1) - 2* x(n+1)/y(n+1));
    end
end
```

数值实验：

计算结果如下：

xn	yn	xn	Yn
0.1	1.091818181818182	0.6	1.460937319600308
0.2	1.176264940057727	0.7	1.521616739611564
0.3	1.254630113352294	0.8	1.578641399417121
0.4	1.327809263590539	0.9	1.632075184847358
0.5	1.396432121354096	1.0	1.681879743353214

```
>> [x, y] = backEuler(a, b, h)

x =

  1 至 7 列

                   0   0.100000000000000   0.200000000000000   0.300000000000000   0.400000000000000   0.500000000000000   0.600000000000000

  8 至 11 列

   0.700000000000000   0.800000000000000   0.900000000000000   1.000000000000000


y =

  1 至 7 列

   1.000000000000000   1.091818181818182   1.176264940057727   1.254630113352294   1.327809263590539   1.396432121354096   1.460937319600308

  8 至 11 列

   1.521616739611564   1.578641399417121   1.632075184847358   1.681879743353214

fx >>
```

3) 梯形方法：

输入的量：区间范围 a 与 b，步长 h

输出的量：得到的一组 x 的向量和一组 y 向量。

算法描述：

在欧拉法的基础上修改迭代的公式如下：

$$\begin{cases} y_{n+1}^{0} = y_n + hf(x_n, y_n) \\ y_{n+1}^{(k+1)} = y_n + \frac{h}{2}[f(x_n, y_n) + f(x_{n+1}, y_{n+1}^{(k)})], k = 0,1,2,\cdots \end{cases}$$

具体代码实现如下：

```
function[x, y] = traprl(a, b, h)
    x = a : h : b;
    y0 = 1;
    y(1) = y0;
    for n = 1 : length(x)-1
        y(n+1) = y(n) + h * (y(n) - 2 * x(n)/y(n));
        y(n+1) = y(n) + h / 2 * (y(n) - 2 * x(n)/y(n)) + h / 2 * (y(n+1) - 2 * x(n+1)/y(n+1));
    end
end
```

数值实验：

计算结果如下：

xn	yn	xn	Yn
0.1	1. 095909090909091	0.6	1. 485955602415668
0.2	1. 184096569242997	0.7	1. 552514091326145
0.3	1. 266201360875776	0.8	1. 616474782752057
0.4	1. 343360151483998	0.9	1. 678166363675185
0.5	1. 416401928536909	1.0	1. 737867401035412

```
>> [x, y] = trapr1(a, b, h)

x =

  1 至 6 列

                    0   0.100000000000000   0.200000000000000   0.300000000000000   0.400000000000000   0.500000000000000

  7 至 11 列

   0.600000000000000   0.700000000000000   0.800000000000000   0.900000000000000   1.000000000000000


y =

  1 至 6 列

   1.000000000000000   1.095909090909091   1.184096569242997   1.266201360875776   1.343360151483998   1.416401928536909

  7 至 11 列

   1.485955602415668   1.552514091326145   1.616474782752057   1.678166363675185   1.737867401035412

fx >>
```

4) 改进欧拉方法

输入的量：区间范围 a 与 b，步长 h

输出的量：得到的一组 x 的向量和一组 y 向量。

算法描述：

在欧拉法的基础上修改迭代的公式如下：

$$\begin{cases} y_p = y_n + hf(x_n, y_n), \\ y_c = y_n + hf(x_{n+1}, y_p), \\ y_{n+1} = \dfrac{1}{2}(y_p + y_c) \end{cases}$$

具体代码实现如下：

```
function[x, y] = improveEular(a, b, h)
    x = a:h:b;
    y0 = 1;
    y(1) = y0;
    for n = 1 : length(x)-1
        yp = y(n) + h * (y(n) - 2 * x(n)/y(n));
        yc = y(n) + h * (yp - 2 * x(n)/yp);
        y(n+1) = 1 / 2 * (yp + yc);
    end
end
```

数值实验：

计算结果如下：

xn	yn	xn	Yn
0.1	1. 105000000000000	0.6	1. 554851395854456
0.2	1. 202718823930517	0.7	1. 638924385009096
0.3	1. 295205321919605	0.8	1. 723270004884592
0.4	1. 383958948579526	0.9	1. 808771008838299
0.5	1. 470169637654031	1.0	1. 896342971973737

```
>> [x, y] = improveEular(a, b, h)

x =

  1 至 6 列

                   0   0.100000000000000   0.200000000000000   0.300000000000000   0.400000000000000   0.500000000000000

  7 至 11 列

   0.600000000000000   0.700000000000000   0.800000000000000   0.900000000000000   1.000000000000000


y =

  1 至 6 列

   1.000000000000000   1.105000000000000   1.202718823930517   1.295205321919605   1.383958948579526   1.470169637654031

  7 至 11 列

   1.554851395854456   1.638924385009096   1.723270004884592   1.808771008838299   1.896342971973737

>>
```

5) 上面四种方法放在一起得到结果：