数值分析实验报告

（实验二）

姓名：陈明亮

学号：16340023

班级号：06

上课时间： 周一 9-10 节，

周四 9-10 节

【一】问题重述

问题 1：结合插值方法以及其相关算法，采用线性插值，二次插值和三次插值，分别求解在条件：sin(0.32)=0.314567, sin(0.34)=0.333487, sin(0.36)=0.352274, sin(0.38)=0.370920 的情况下，计算 sin(0.35)的值。

问题 2：熟悉并掌握非线性方程组的各种数值解法，运用二分法，牛顿法，简化牛顿法，以及弦截法分别求解 115 的平方根，精确到小数点后 6 位，并绘制出横坐标分别为计算时间，或迭代步数时的收敛精度曲线。

问题 3：学习最小二乘法的曲线拟合知识，并采用递推最小二乘法，求解不存在精确解的超定方程组 Ax=b 的近似解，给出横坐标为迭代步数时的近似解收敛精度曲线，此处采用前后两步迭代解的差值范数作为收敛精度。此处要求矩阵 A 为 m x n 维的已知矩阵，b 为 m 维的已知向量，同时 A 与 b 中的元素服从独立同分布的正态分布，x 为 n 维的未知向量，是我们需要求解的值。

问题 4：了解并掌握离散傅里叶变换 DFT 和快速傅里叶变换 FFT 的算法步骤，明白此两种算法将信号从时域变换到频域的过程和原理，以及快速傅里叶变换的优势，低复杂度(O(NlogN))。编写测试点数为 1024 点的快速傅里叶变换的算法，自行生成一段掺杂不同频率正弦的信号，测试其频域变换结果并绘图，与 Matlab 自带 fft 函数的求解值比较。

问题 5：编写两种数值积分求解算法程序：复合梯形公式和复合辛普森公式，规

定求解区间为[0, 1]，采样点数目分别为 5,9,17,33，即将区间分为 4,8,16,32 等分，分别运用两种数值积分方法求解函数 sinx/x 在[0,1]范围内的积分值。

问题 6：掌握常微分方程初值问题的四种基本解法：前向欧拉方法，后向欧拉方法，梯形法和改进的欧拉方法。固定计算区间为[0,1]，步长为 0.1，给定常微分方程初值条件为 $y'=y-2x/y$，y(0)=1，运用上述四种解法求解每一步的函数值，分别已知原函数 $y=(1+2x)^{1/2}$ 在每一步的函数值比较，先绘制两者的所得函数值曲线，再查看每一种解法，在每一步时关于精确解的差值，反映计算精度，得出结论。

【二】算法设计

一、线性插值，二次插值以及三次插值方法设计

1. 简单线性插值法的 Matlab 实现和截断误差计算

所谓的线性插值实际上是将插值多项式定义为一次线性多项式，此处我们可以结合所求的横坐标点 x=0.35，线性插值只需要两点，构造过该两点的直线即可完成插值。

我们此处采用距离 0.35 最近的两点 x=0.34 和 x=0.36，根据线性插值的点斜式形式 $L_1(x)=y_k+\frac{y_{k+1}-y_k}{x_{k+1}-x_k}(x-x_k)$，我们分别代入两点的 x 和 y 值，以及 x=0.35 的求解点横坐标，可得计算过程：

```
%% 输出：Lin - 线性插值结果，Qua - 二次插值结果，Tri - 三次插值结果%%
x = [0.32; 0.34; 0.36; 0.38];
y = [0.314567; 0.333487; 0.352274; 0.370920];
R = zeros(3, 1);
x0 = 0.35;

%% 线性插值 %%
Lin = y(2) + (y(3) - y(2)) * (x0 - x(2)) / (x(3) - x(2));
%% 计算线性插值截断误差 %%
R(1) = 0.5 * y(3) * abs((x0-x(2))*(x0-x(3)));
```

此处也计算了线性插值的截断误差，运用公式：

$R_1(x)=\frac{M_2}{2}\left|(x-x_0)(x-x_1)\right|$，记录截断误差值便于之后的误差值比较图的绘制。

2. 二次插值和三次插值的 Matlab 程序实现及截断误差计算

二次插值和三次插值则是在线性插值的基础上，拓展了所需要的插值所需点数，使得构建出的函数图像更加精确，运用到的点函数值更加多，插值基函数的数量也随着增多，插值所得函数则更加复杂，随着插值基函数数目的增加，最终的求解结果也更加精确。

首先讨论二次插值的计算方法。二次插值需要的插值节点数目为三个，对应于多项式的构建形式为二次函数，最高项阶数为 2 阶，所以称为二次插值，需要构建的插值基函数数目也为 3 个，最终的插值多项式为：

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

此处我们选择的三个插值点为 0.34，0.36，0.38，并将对应的函数值带入上式，编写的 Matlab 程序如下：

```
%% 二次插值 %%
Qua = 0;
```

```
for i=2:4
Temp1 = 1;
Temp2 = 1;
for j=2:4
if j ~= i
Temp1 = Temp1 * (x0 - x(j));
Temp2 = Temp2 * (x(i) - x(j));
end
end
Qua = Qua + y(i) * Temp1 / Temp2;
end
%% 计算二次插值截断误差 %%
R(2) = sqrt(1 - y(2)*y(2)) / 6 * abs((x0-x(2))*(x0-x(3))*(x0-x(4)));
```

对应的，我们根据 n 次插值的截断误差公式，计算二次插值的截断误差存储向量 R。同样地，接下来我们讨论三次插值的计算过程，其实际上则是将插值基函数拓展到 4 个，采取的插值节点为 4 个，即题目所提供的 4 个节点全部用上，计算三次插值多项式 L_3 的解。

实际上，在数学层面上三次插值的公式比二次插值复杂很多，基本上每一个插值基函数都需要添加上新加的一个点，但在编程方面，结合我此处的循环求积再进行除法构造基函数的写法，只需要在二次插值循环的基础上增加一个新点值即可，具体步骤如下：

```
%% 三次插值 %%
Tri = 0;
for i=1:4
Temp1 = 1;
Temp2 = 1;
for j=1:4
if j ~= i
Temp1 = Temp1 * (x0 - x(j));
Temp2 = Temp2 * (x(i) - x(j));
end
end
Tri = Tri + y(i) * Temp1 / Temp2;
end
%% 计算三次插值截断误差 %%
R(3) = y(4) / 24 * abs((x0-x(1))*(x0-x(2))*(x0-x(3))*(x0-x(4)));
```

二、二分法、牛顿法、简化牛顿法和弦截法的算法设计

1. 二分法的 Matlab 程序实现

二分法求解问题 2 的关键在于，整数 115 的平方根实际上是方程：$x^2-115=0$ 的正数解。那么根据题目为二分法选取的求根区间[10,11]，我们在该区间内实施二分手续，取中点 $x_{T/2}=(a+b)/2$，并且检测中点横坐标所得值与精确值之间的误差，一步一步进行迭代，若达到了所需的精度范围内，那我们就承认该解是充分近似的，此时的中点横坐标记为近似解。

在实际的 Matlab 代码编写中，为了更好地使用多种方法进行求解，我们将二分法（以及其余的解法）独立设为函数文件，同时为了更好地绘制图像，我们将计算模式分为：时间模式和迭代步数模式。以下是迭代步数（总共进行了 1000 步）模式的二分法核心代码：

```
for i = 1:1:1000
precision(step+1) = abs((x(1)+x(2))/2 - result);
if(precision(step+1) <= maxError && ~conFlag)
con = step;
conFlag = true;
end
if (((x(1)+x(2))/2)^2 - 115)*(x(1)^2-115) < 0
x(2) = (x(1)+x(2))/2;
else
x(1) = (x(1)+x(2))/2;
end
step = step + 1;
end
```

可以看到，每一步我们都将所得区间的中点坐标与精确值之间的差距存储在每一步对应的 precision 单元中，当检测到误差值小于 maxError 时（此处选取 10^{-11}），我们则判断该近似值为精度范围内的近似根，该方法所得的近似解成功收敛到精度范围内。同时，二分法的重点还在于每一步之后的区间选取，若有

$f(\frac{a+b}{2})f(a)<0$，中点函数值和起点函数值符号相反，那么我们则选取$\frac{a+b}{2}$作为新区间的右界，因为$f(x)=0$的解必定在 a 和$\frac{a+b}{2}$中，反之则用$\frac{a+b}{2}$替换 a，继续进行迭代步骤。

2. 牛顿法的 Matlab 程序实现

牛顿法求解非线性方程的思想是：通过将非线性方程逐步逼近，归结成线性方程来进行求解。而在我们解决实际问题时，牛顿法经常采用函数的导数进行某点上的函数值逼近，通过$f(x)\approx f(x_k)+f'(x_k)(x-x_k)$这一近似根求解方式，结合非线性方程常规形式$f(x)=0$，得到牛顿法的迭代计算公式：

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

在实际的 Matlab 程序编写中，为简化计算，本人首先将迭代公式的右侧结合题目所给函数$x^2-115=0$化简，实际上运用的是以下公式：

$$x_{k+1}=\frac{1}{2}(x_k+\frac{115}{x_k})$$

具体的函数实现如下（迭代步数模式，总迭代步长为 1000 步）：

```
for i = 1:1:1000
temp = (x0 + 115/x0)/2;
precision(step+1) = abs(temp - result);
if(precision(step+1) <= maxError && ~conFlag)
con = step;
conFlag = true;
end
step = step + 1;
x0 = temp;
end
```

为减少空间复杂度，此处的每步迭代结果都只存储在一个单元 x0，在异步迭

代完成时更新其值。同样地，与之前的二分法程序类似，在每一步迭代内我们保存精度，监控收敛到规定精度的最少步数。

3. 简化牛顿法的 Matlab 程序实现

为了克服牛顿法每一迭代都要重新计算某一点导数值，计算量大的缺点，简化牛顿法产生了。该方法首先使用常数 C 替代每一步计算公式中的 $\frac{1}{f'(x_k)}$，实际上是采用了初始点 x0 的导数值的倒数作为 C，减少了计算量，通用的迭代公式为：

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_0)}$$

在实际的 Matlab 程序实现中，结合此题所给函数，我们运用的真实迭代公式是：

$$x_{k+1} = x_k - \frac{x_k^2 - 115}{20}$$

具体的实现步骤如下：

```
for i=1:1000
temp = x0 - (x0*x0-115)/20;
precision(step+1) = abs(result - temp);
if(precision(step+1) <= maxError && ~conFlag)
con = step;
conFlag = true;
end
x0 = temp;
step = step + 1;
end
```

同样地，我们代入简化牛顿法迭代公式计算每一步所得解，进行收敛精度的记录和判断，最终输出。

4. 弦截法的 Matlab 程序实现

弦截法采用线性插值的方法，回避了牛顿法中，每一步迭代都必须要计算某一点导数值的缺点，改成使用前两步迭代所得的点和函数值构造一次插值多项式，这样导出的迭代公式如下：

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

弦截法实际上是将前面两步迭代所得到的点的斜率作为导数值的替代品代入公式，实际上求得的解，往往是弦 P 与横坐标轴的交点，这也是为什么该方法被称为弦截法。结合题目所给目标函数，在实际程序中我们采用的迭代公式为：

$$x_{k+1} = x_k - \frac{(x_k^2 - 115)(x_k - x_{k-1})}{(x_k^2 - 115)(x_{k-1}^2 - 115)}$$

```
for i=1:1000
temp = x1 - ((x1*x1-115)*(x1-x0)) / ((x1*x1-115) - (x0*x0-115));
precision(step+1) = abs(result - temp);
if(precision(step+1) <= maxError && ~conFlag)
con = step;
conFlag = true;
break;
end
x0 = x1;
x1 = temp;
step = step + 1;
end
```

三、递推最小二乘法求解超定方程组的算法设计

1. 递推最小二乘法的 Matlab 算法实现

递推最小二乘法常用于曲线拟合方面，而此处使用该方法解决超定方程组，实际上我们可以得知超定方程组没有精确解，所以采用最小二乘法来逐步收敛精确到其近似解。

递推最小二乘的核心在于，线性方程组 Ax=b 可以推出：$A(k,:)x = b_k$，即矩阵 A 的每一行与未知向量 x 的乘积必等于向量 b 的第 k 个元素，k=1,2，，，n，其中 n 为 b 向量维度。那么对于超定方程组的近似求解过程，我们看做是误差函数：$e(k) = b_k - f(k)$ 基于性能指标 J 的优化求解问题。结合神经网络解决多变量优化问题，此处我们选取性能指标 $J = \frac{1}{2}\sum_{k=1}^{n} e^2(k)$，使 $\frac{\partial J}{\partial \mathrm{x}} = 0$，不断地通过迭代去逼近该条件，那我们就能够由此得出递推最小二乘 RLS 的计算公式：

$$x_{k+1} = x_k + Q^k[b_k - A(k,:)x_k]$$

其中此处的 Q 矩阵和 P 矩阵是为了简化算式，提出的中间变量。为了设置初始条件，我们首先将矩阵 P 定义为 $P = \partial I$，其中 I 为单位矩阵（n x n），对于该题为 10 x 10，初始的向量 x 则随机产生，维度也是 n，此处为 10。以下是两个中间矩阵 Q 和 P 的运算公式：

$$Q^{\mathrm{k}} = \frac{P^k A^T(k,:)}{1 + A(k,:)P^k A^T(k,:)}$$

$$P^{\mathrm{k+1}} = [I - Q^k A(k,:)]P^k$$

通过一步一步的迭代，我们逐渐获得逼近近似解，并且在每一步迭代中与上次迭代结果之间的差距不断缩小的解，在小于一定的误差范围内时，我们就认为该近似解 x 是收敛到精度范围内的，以下是上述算法在 Matlab 中的实现：

```
for step=1:1000
temp = x0;
Q0 = P0*(A(step,:))' / (param+A(step,:)*P0*(A(step,:))');
P0 = (I - Q0*A(step,:))*P0/param;
x0 = x0 + Q0*(b(step) - A(step,:)*x0);
precision(step) = norm(x0 - temp);
if precision(step) <= maxError && ~conFlag
```

```
con = step;
conFlag = true;
end
end
```

此处为简化空间复杂度，我们均采用一个单元来存放中间变量，同时每一步迭代都记录其与上一步结果之间的差值范数，迭代总步数为 1000 步，此处采用的误差精度 maxError 设置在 10^{-5}，符合条件时就把该处的迭代步数 step 传出，变量 param 被称为是遗忘因子，此处全部设置为 1，I 为单位矩阵。

2. 超定线性方程组矩阵 A，b 的生成

此处的 A，b 对应于维度 10000 x 10，和 10000 x 1，两矩阵内部的元素都需要符合独立同分布的正态分布，此处采用 Matlab 自带的 randn 函数，生成对应的问题条件矩阵。

```
%% 随机生成 10000X10 的，元素服从独立同分布的正态分布的矩阵 A，以及维度为 10000 的向量 b %%
function [A, b] = Generate()
A = randn(10000, 10);
b = randn(10000, 1);
end
```

四、1024 点快速傅里叶变换的算法设计

1. N 点的快速傅里叶变换 FFT 算法实现

对于所需要分析的样本数据具有周期性的情况，我们经常采用三角函数作为逼近的基函数，如正弦函数或余弦函数，其他的线性函数都是无效的。此时，傅里叶变换就可以通过三角函数逼近给定数据的最小二乘和插值等计算过程，关于离散点的傅里叶变换我们称为 DFT，离散傅里叶变换。后来根据运算数据的周期性和对称性，算法上得到了极大地改进，称为 FFT，快速傅里叶变换。

快速傅里叶变换是建立在 DFT 的基础上的，首先运用傅里叶变换公式，将

一系列的时域上的点通过变换映射到频域上，但 FFT 不断的在每一步运用二分手续，结合对称性大大简化了运算的过程。为了更好地解决问题 4，我们此处采用改进的快速傅里叶变换算法，不仅仅大大减少了每一步的存储单元数量，更是节省了计算量，核心算式如下：

$$A_q(k2^q + j) = A_{q-1}(k2^{q-1} + j) + A_{q-1}(k2^{q-1} + j + 2^{p-1})$$

$$A_q(k2^q + j + 2^{q-1}) = [A_{q-1}(k2^{q-1} + j) - A_{q-1}(k2^{q-1} + j + 2^{p-1})] * w^{k2^{q-1}}$$

结合具体的改进方法，以下是 Matlab 程序实现图，此处选择 N=1024：

```
%% 算法实现 %%
%% 初始化 A1 和 w 向量 %%
for i0=1:N
A1(i0) = signals(i0);
end
for j=1:N/2
w(j) = exp(-1i*2*pi*(j-1)/N);
end

%% 循环二分 %%
for q=1:p
if mod(q, 2) == 1
    for k=0:2^(p-q)-1
        for j=0:2^(q-1)-1
            A2(k*2^q+j+1) = A1(k*2^(q-1)+j+1) + A1(k*2^(q-1)+j+2^(p-1)+1);
            A2(k*2^q+j+2^(q-1)+1) = (A1(k*2^(q-1)+j+1) -
                                A1(k*2^(q-1)+j+2^(p-1)+1))*w(k*2^(q-1)+1);
        end
    end
else
    for k=0:2^(p-q)-1
        for j=0:2^(q-1)-1
            A1(k*2^q+j+1) = A2(k*2^(q-1)+j+1) + A2(k*2^(q-1)+j+2^(p-1)+1);
            A1(k*2^q+j+2^(q-1)+1) = (A2(k*2^(q-1)+j+1) -
                                A2(k*2^(q-1)+j+2^(p-1)+1))*w(k*2^(q-1)+1);
        end
    end
end
if q==p
    break;
```

```
end
end
if mod(p, 2)==0
A2(1:N) = A1(1:N);
end
```

此处的 signals 变量即为生成的原始信号，采样点数目依旧是 N=1024，根据改进的 FFT 算法，我们此处定义两个初始向量 A1，A2，并将所得到的原始信号首先存储到 A1 向量内部，其次则是权值 w 的初始化，此处我们选择 exp 函数，接收区间长度为 2pi 内部的 1024 个点的虚数坐标。接下来则是核心的循环迭代部分，基于从 1 到 p（p 为迭代点数基于 2 的对数值，此处为 10）的 q 的奇偶性，我们分别进行 A1->A2，或者 A2->A1 的内部循环更新（基于原始的 FFT 迭代公式）。最终若 p 为偶数，则输出 A1 为变换结果，反之则为 A2.

2. 混杂不同频率正弦的信号生成

此处我们将采样点数设置为 1024，采样频率设为 1s，信号基函数为两个不同频率的正弦函数，输出生成的信号，以及对应的采样频率序列。

```
%% 生成测试快速傅里叶变换的不同频率正弦信号 %%
function [signals, fre] = Generate1()
N0 = 1024; %% 正弦信号长度为 1024
Fs = 1; %% 采样频率
Dt = 1/Fs;
time = [0:N0-1]*Dt; %% 时间序列
x0 = sin(2*pi*0.24*[0:1023]) + sin(2*pi*0.26*[0:1023]);
signals = [x0, zeros(1, N0-1024)]; %%输出正弦信号
f0 = 1/(Dt*N0);
fre = [0:ceil((N0-1)/2)]*f0; %% 频率序列

plot(time, signals, 'r-');
end
```

五、复合梯形公式和复合辛普森公式的算法设计

1. 复合梯形公式的实现

复合梯形公式通过将积分区间分为多个子区间，再根据所提供的积分函数，分别在每个子区间上运用梯形公式近似求解对应区间上的积分值，最终再结合梯形面积公式，整体地将每个子积分面积集合起来，得到最终的近似解，所采用的公式如下：

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

可以看到该公式与梯形面积公式，和原始的梯形求积分公式十分相像，基本上是复合使用，提高精度的版本。而实际上，我们在 Matlab 实现程序中，结合本题的原函数 $f(x) = \frac{\sin x}{x}$，经常会遇到区间初始端点函数值为 Infinite 的情况，所以我们需要将每个子区间的端点函数值为 0 的情况剔除，为那些为 0 的点加上一个极小值，然后进行复合梯形公式的代入，求解。

```
n = num-1; %% 区间划分为 n 等份
h = (b-a) / n; %% 每一份的宽度
x = a + (0:n)*h; %% 每个采样点坐标
for i=1:n+1
if x(i) == 0
     x(i) = 10e-10; %% 处理函数值为 0 的坐标点
end
end
T1 = sin(x(1)) / x(1) + sin(x(n+1)) / x(n+1);
T2 = 0;

%% 复合梯形算法 %%
for k=1:n-1
T2 = T2 + sin(x(k+1)) / x(k+1);
end
T = h*(T1 + 2*T2) / 2;
```

此处我们将求解函数值之和分为两部分 T1 和 T2，分别是原始区间的端点函

数值，以及子区间的求解积分值，运用公式即可得出复合梯形法的解，过程十分简单。

2. 复合辛普森公式的实现

辛普森公式与梯形公式的区别就在于，辛普森公式采用的插值方法为二次插值，即插值基函数有三个，柯斯特系数为 1/6, 2/3, 1/6，精度相比于梯形公式更加精确。而复合辛普森公式就是将所求区间分为若干个子区间，在他们上面使用辛普森公式求解积分，然后再根据其公式：

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

将所求得的子区间积分通过系数组合，即可得到复合辛普森公式的求解。在实际的 Matlab 程序实现中，我们同样采用初始化剔除全 0 函数值，同时分块计算子区间对于数值积分的方法，最终运用复合辛普森公式，得到所求。

```
n = num-1; %% 区间划分为 n 等份
h = (b-a) / n; %% 每一份的宽度
x = a + (0:n)*h; %% 每个采样点坐标
for i=1:n+1
if x(i) == 0
     x(i) = 10e-10; %% 处理函数值为 0 的坐标点
end
end
S1 = sin(x(1)) / x(1) + sin(x(n+1)) / x(n+1);
S2 = 0;
S3 = 0;

%% 复合辛普森算法 %%
for k=0:n-1
S3 = S3 + sin(x(k+1)+h/2) / (x(k+1)+h/2);
if k > 0
     S2 = S2 + sin(x(k+1)) / x(k+1);
end
end
```

```
S = h*(S1 + 2*S2 + 4*S3) / 6;
```

六、前向欧拉法，后向欧拉法，梯形方法和改进欧拉方法的算法设计

1. 前向欧拉法的 Matlab 程序实现

前向欧拉方法运用每一步的导数值，即常微分问题所提供的一阶导数公式，结合当前迭代的点坐标和函数值，结合步长，计算推出下一点的函数值，通用的计算迭代公式为：

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

其中有函数 f(x)为 y 关于 x 的一阶导数表达式，h 为计算的求解区间每一步的步长。结合本题所给的条件，我们有 $f(x) = y - \frac{2x}{y}$，那么，在 Matlab 实现程序中，我们实际上运用的计算公式是：

$$y_{k+1} = y_k + h(y_k - \frac{2x_k}{y_k})$$

在 Matlab 实现中，我们将每个数值方法定义为函数，输入为区间的左右端点横坐标，以及步长 h，初值 y0，然后根据步长构建横坐标向量 x，其近似函数值存储在向量 y 中，包含初值 y0，下面给出核心代码：

```
%% 前向欧拉方法 - Matlab 函数 %%
function [result] = ForwardEuler(a, b, h, y0)
x = a + (0:h:b);
y = zeros(b/h+1, 1);
y(1) = y0;
%% 进行前向循环求解 %%
for i=2:b/h+1
y(i) = y(i-1) + h * (y(i-1) - (2*x(i-1)/y(i-1)));
end
result = y;
End
```

2. 后向欧拉法的 Matlab 程序实现

后向欧拉方法与常规的欧拉法最大的不同在于，此处我们不再利用左矩阵公式，即上一步迭代作为取值点传入导数表达式 f(x)中，而是直接利用未知点(x_{k+1}, y_{k+1})，实际上是隐式公式，来计算每一步迭代的数值。

考虑到后退的欧拉法实际上很需要考验隐式方程的求解，除非我们很明显地给出显式一阶导数表达式。结合此题目中 $y^{'} = f(x) = y - \frac{2x}{y}$，我们直接代入后退法的一般公式：$y_{k+1} = y_k + hf(x_{k+1}, y_{k+1})$，我们直接得出，问题 6 的后退欧拉法的迭代公式：

$$y_{k+1}^2 - y_k * y_{k+1} + 2hx_k = 0$$

我们可以很清楚地看到，该迭代公式实际上是一个一元二次方程，我们需要做的就是解方程得出 y_{k+1} 的解，就是该迭代公式的解。在实际的 Matlab 程序中，我们使用 root 函数求解一元二次方程，只需要传入对应项的系数即可，就可以返回该方程的解。

```
%% 后向欧拉方法 - Matlab 函数 %%
function [result] = BackwardEuler(a, b, h, y0)
x = a + (0:h:b);
y = zeros(b/h+1, 1);
y(1) = y0;
%% 进行后向循环求解 %%
for i=2:b/h+1
temp = roots([(1-h), -y(i-1), 2*h*x(i)]);
y(i) = temp(1); %% yn+1 就是方程(1-h)y^2 - yn*y + 2*h*xn+1 = 0 的解
end
result = y;
end
```

3. 梯形方法的 Matlab 程序实现

为了得到比欧拉方法更加精确的解，我们引入梯形方法，实际上是借鉴了梯

形求积公式，将上一步的已知点代入一阶导数表达式 f(x)，和这一步的未知点一并代入并求和，根据梯形公式最终得出解，一般情况的迭代公式为：

$$y_{k+1} = y_k + \frac{h}{2}[f(x_k, y_k) + f(x_{k+1}, y_{k+1})]$$

实际上与后退欧拉法十分相同的是，该数值方法也是隐式公式，不能够直接代入坐标点进行求解，但可以通过解方程进行迭代求解。对应于本题已经给出的原始一阶导数，我们将问题 6 梯形方法的迭代公式归结如下：

$$(1-\frac{h}{2})*y_k*y_{k+1}^2+[hx_k-(1+\frac{h}{2})*y_k^2]*y_{k+1}+hx_{k+1}*y_k=0$$

Matlab 实现代码如下：

```
%% 梯形法 - Matlab 函数 %%
function [result] = Echelon(a, b, h, y0)
x = a + (0:h:b);
y = zeros(b/h+1, 1);
y(1) = y0;
%% 进行梯形方法求解，将隐式函数转换为方程求解 %%
for i=2:b/h+1
temp = roots([((1-h/2)*y(i-1)), (x(i-1)*h - (1+h/2)*y(i-1)^2), h*x(i)*y(i-1)]);
y(i) = temp(1);
end
result = y;
end
```

4. 改进欧拉方法的 Matlab 程序实现

改进的欧拉方法通过对梯形方法迭代求解未知量的次数控制，有效地解决了计算量大，迭代时间长的缺点，同时利用预测值和校正值更加好地优化了精度，下面是改进欧拉方法的平均化方程组形式：

$$y_p = y_k + hf(x_k, y_k)$$
$$y_c = y_k + hf(x_{k+1}, y_p)$$
$$y_{k+1} = \frac{1}{2}(y_p + y_c)$$

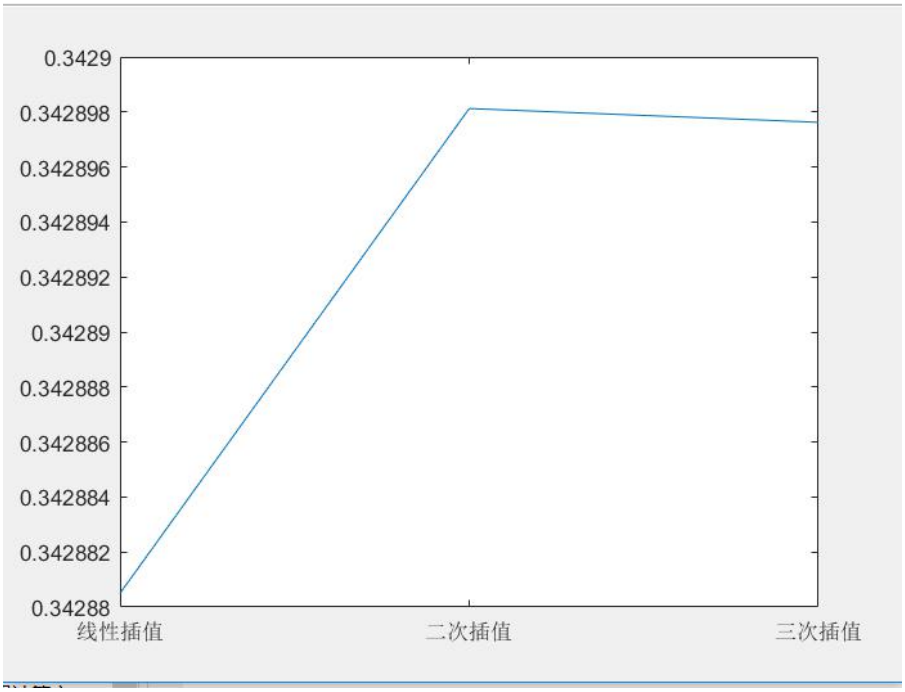
以下是改进欧拉方法的 Matlab 程序：

```
%% 改进欧拉方法 - Matlab 函数 %%
function [result] = ImprovedEuler(a,b,h,y0)
x = a + (0:h:b);
y = zeros(b/h+1, 1);
y(1) = y0;
%% 进行改进欧拉方法求解 %%
for i=2:b/h+1
yp = y(i-1) + h * (y(i-1) - (2*x(i-1)/y(i-1)));
yc = y(i-1) + h * (yp - (2*x(i)/yp));
y(i) = (yp+yc)/2;
end
result = y;
end
```

【三】数值实验与结果分析

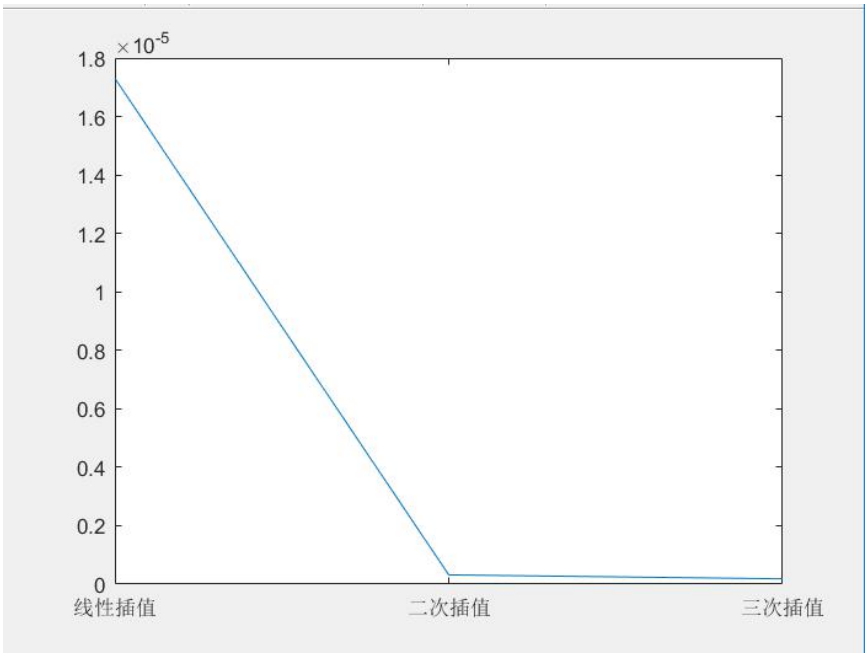
一、对问题一的数值实验与分析

我们分别运用线性插值，二次插值和三次插值方法，计算 sin(0.35)的值，所得的结果图像如下：



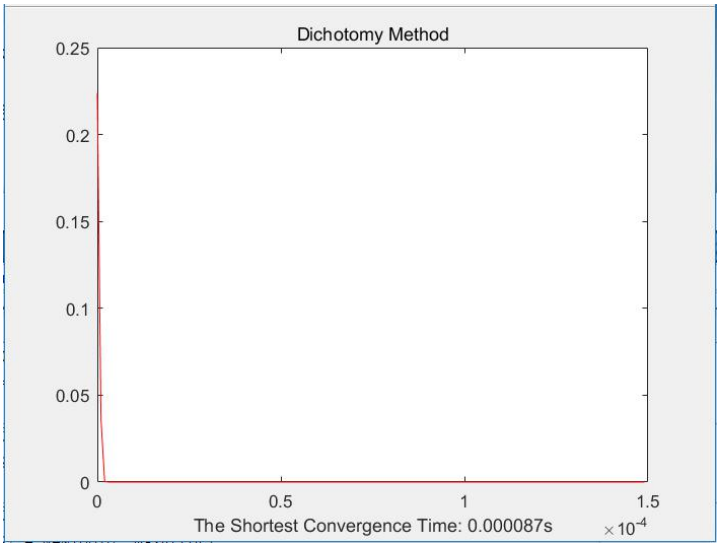

```
>> format long
>> [Lin;Qua;Tri]

ans =

   0.342880500000000
   0.342898125000000
   0.342897625000000
```

可得对于求解 sin(0.35)的函数值，线性插值所得解为 0.342881，二次插值所得结果为 0.342898，三次插值所得解为 0.342898。（四舍五入，取小数点后 6 位数），下图为三种插值方法所得目标解与真正的精确解之间的误差，可以看到线性插值解与精确解之间的误差是最大的，达到了 $1.8*10^{-5}$，而二次插值和三次插值则接近于 $0.01*10^{-5}$，整体误差逐渐缩小。

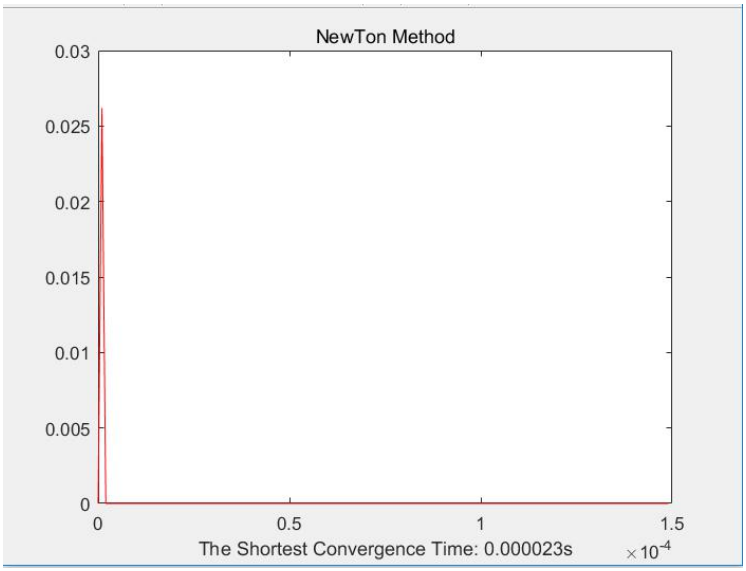


二、对问题二四种求解非线性方程的数值实验与分析

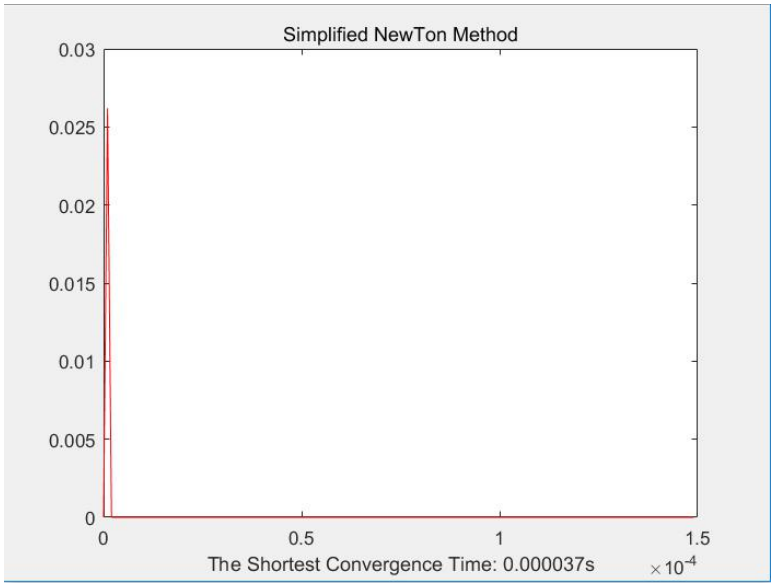
对于问题二四种方法的数值实验，由题可得，我们首先规定非线性方程：$x^2-115=0$，并作为目标函数输入到每个方法函数中，首先我们将时间作为横坐标值，纵坐标则是每一步方法求解与精确解之间的误差精度，下面给出实验图片：



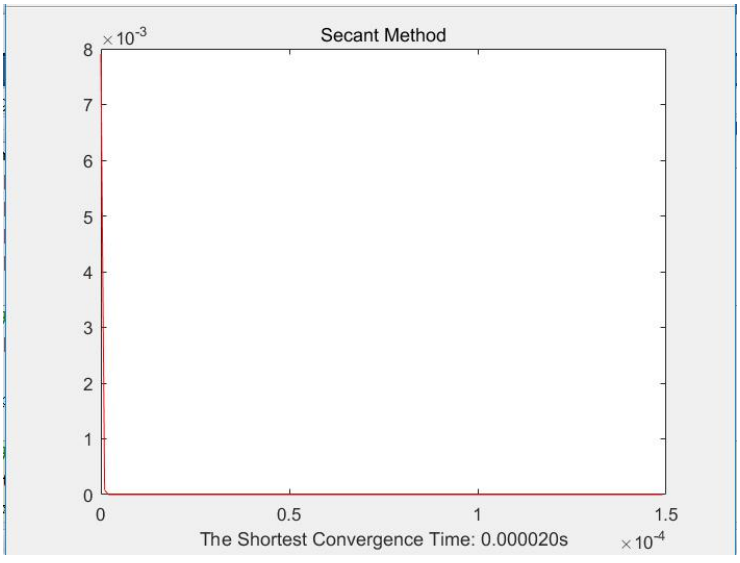
二分法的收敛曲线，可得最短收敛精度时间为 $0.87*10^{-4}s$



牛顿法的收敛曲线，可得最短收敛精度时间为 $0.23*10^{-4}s$

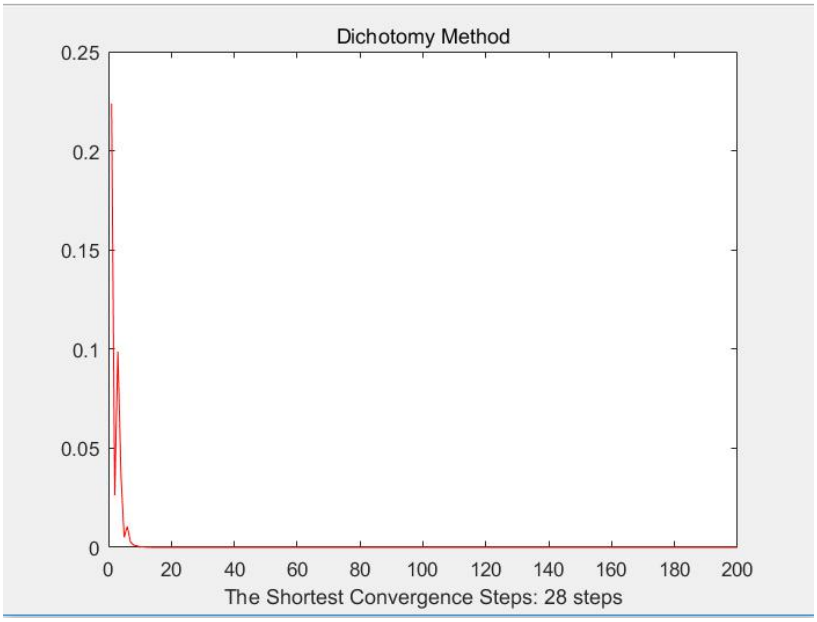


简化牛顿法的收敛曲线，可得最短收敛精度时间为 $0.37*10^{-4}s$

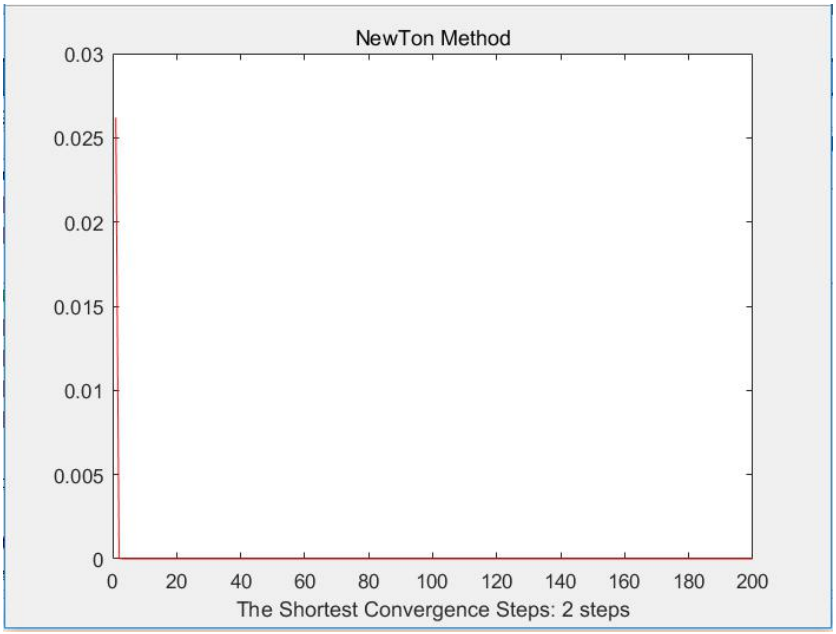


弦截法的收敛曲线，可得最短收敛精度时间为 $0.20*10^{-4}$s

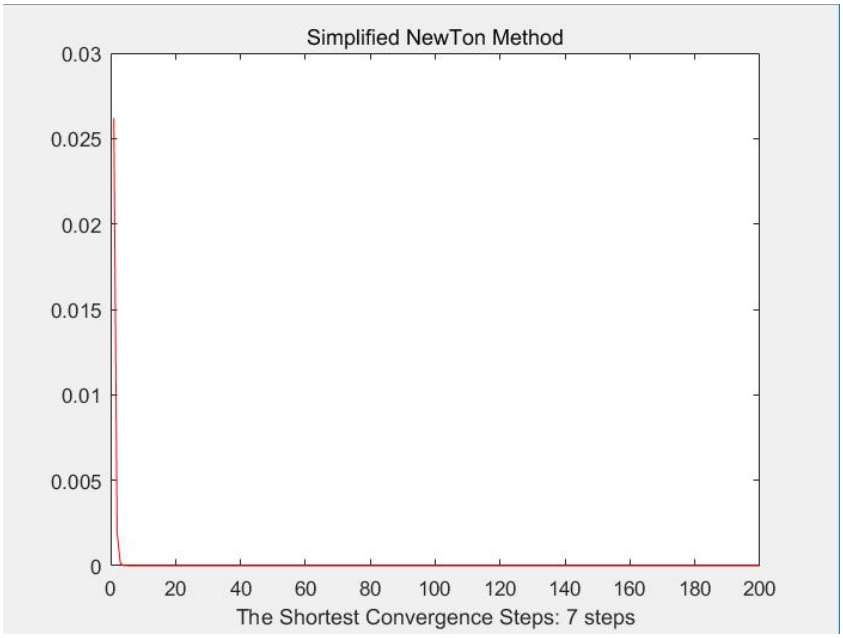
分析：根据这四种方法的先后顺序，我们已知实际上收敛速度是二分法最差，其余三种实际上是差不多的，都是相对比较好的算法（x 的运算初值皆设为接近的 10）。此处我们的收敛精度范围设置在了 10^{-10}，因此收敛结论是相对可靠的，因为精度范围十分严格。下面给出横坐标为迭代步数时的情况：



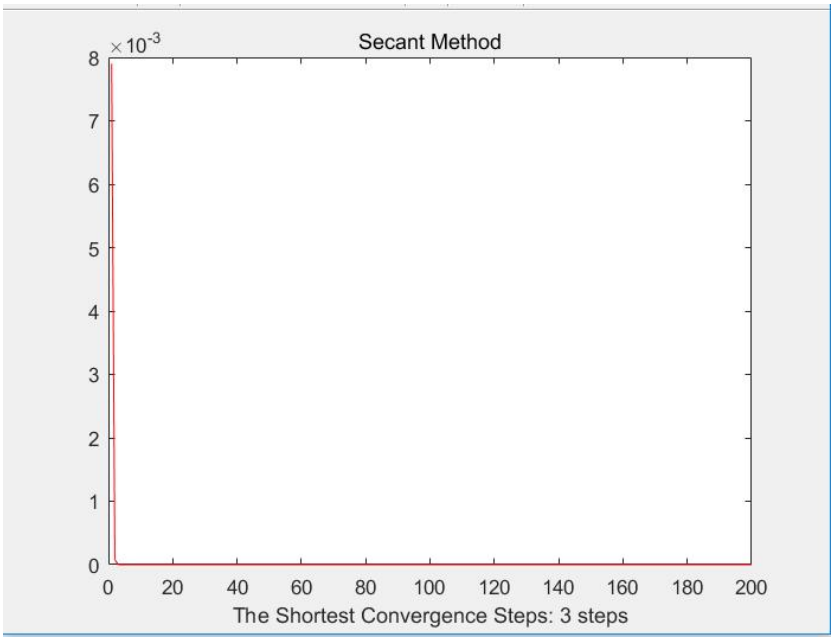
二分法的收敛曲线，可得最短收敛步数为 28 步



牛顿法的收敛曲线，可得最短收敛步数为 2 步

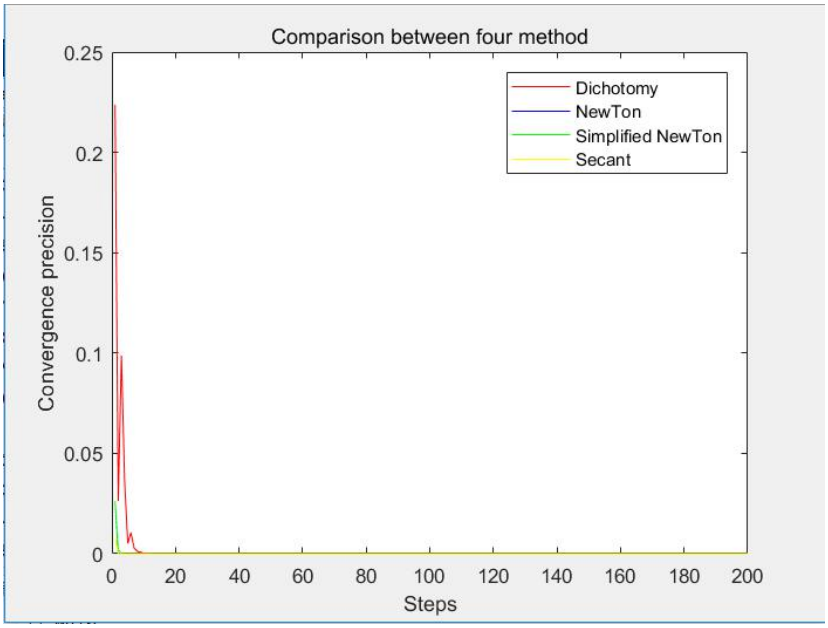


简化牛顿法的收敛曲线，可得最短收敛步数为 7 步

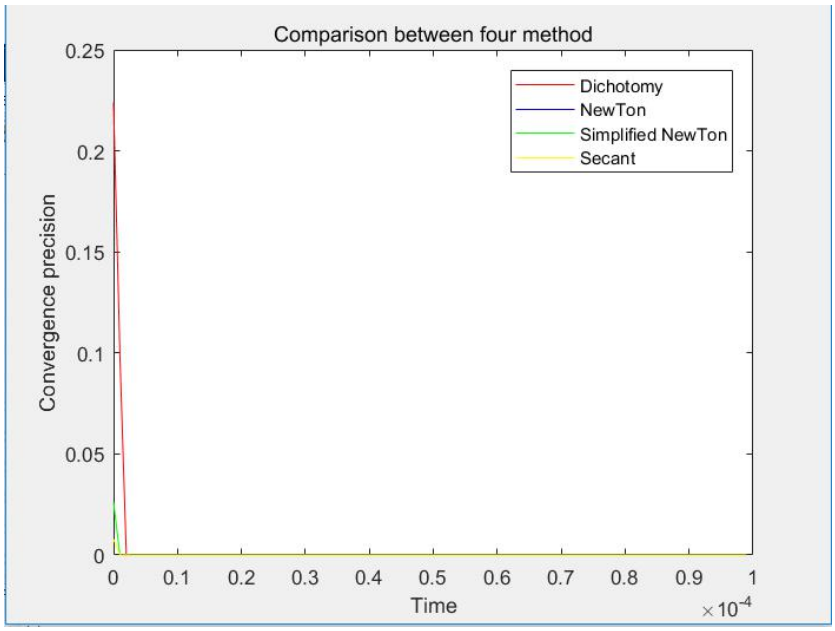


弦截法的收敛曲线，可得最短收敛步数为 3 步

可以看到，横坐标为迭代步数时，二分法的性能也相对其他三种较弱，需要迭代到 28 步才能够收敛。而此处我们对其他三种方法的初值皆设置为 10，所以牛顿法可以很快地，只需要两步即可收敛到精度范围内。简化牛顿法需要更多的迭代步数，这与我们的预期相同，弦截法的收敛速度和牛顿法是不相上下的。以下是四种方法的比较实验图。



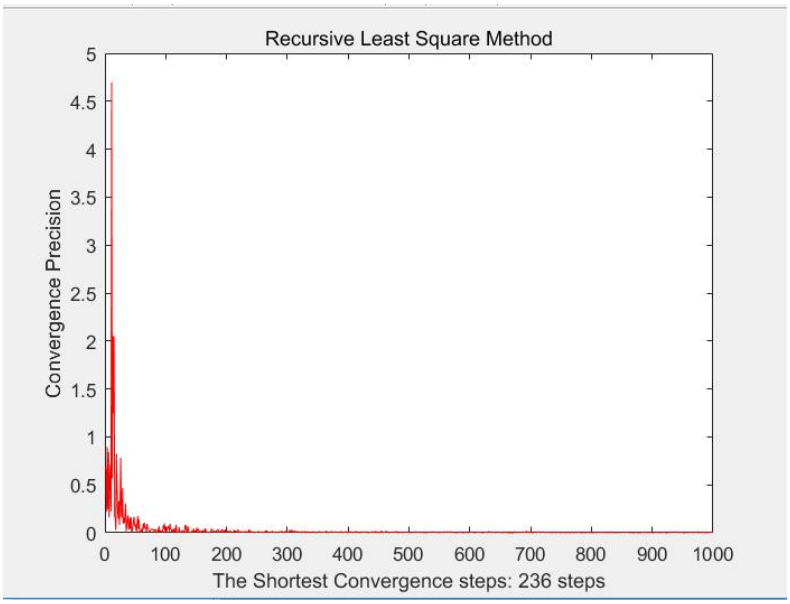
基于迭代步数的收敛比较曲线



基于迭代时间的收敛比较曲线

三、对问题三递推最小二乘法的数值实验与分析

通过对递推最小二乘算法的实现，我们随机地生成符合条件的 A，b 矩阵，下面给出横坐标为迭代步数时，算法每一步迭代与上一步迭代值差范数作为纵坐标的收敛精度曲线：



此处我们设置的收敛精度为 10^{-5}，实际上作为超定方程组近似解的逼近，递推最小二乘法给到的精度不会十分地好，此处我们可以看到在 236 步时就已经是收敛到精度范围内了，整体趋势收敛，除了初始时的较多抖动，实际上是因为对初值的随机的逐步收敛需要时间，所以不免在开始时出现该情况。

四、对问题四 1024 点快速傅里叶变换的数值实验与分析

对于快速傅里叶变换的问题，我们首先要生成初始的时域输入正弦信号。结合算法设计的部分，我们此处采用两个正弦信号作为基信号，所生成的原始信号如下图 4.1，可以看到原始信号是相对对称和具有周期性的。同时，我们将原始信号输入到编写好的 1024 点 FFT 算法中，得到实验图 4.2。为了更好地说明结果的正确性，我们与 Matlab 内部自带的 fft 函数输出值作对比，图 4.3，发现两图完全相同，从而证明了 1024 点 FFT 算法编写的正确性。

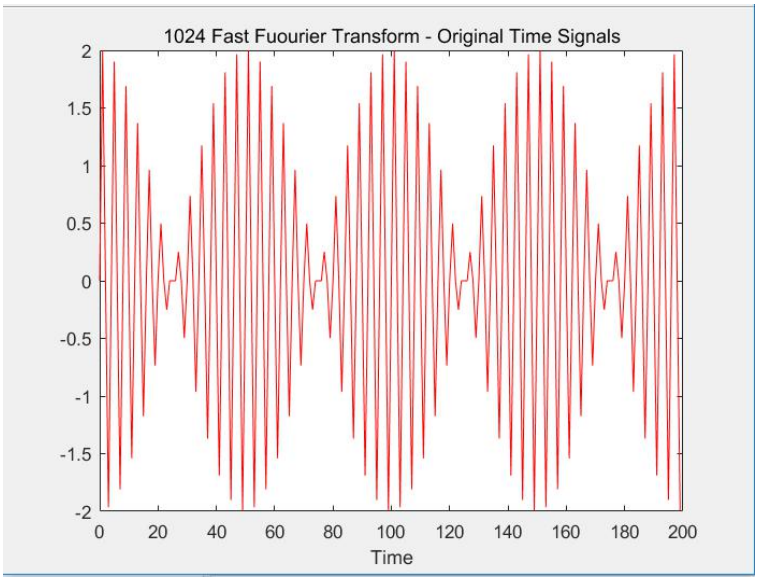


图 4.1 原始时域信号

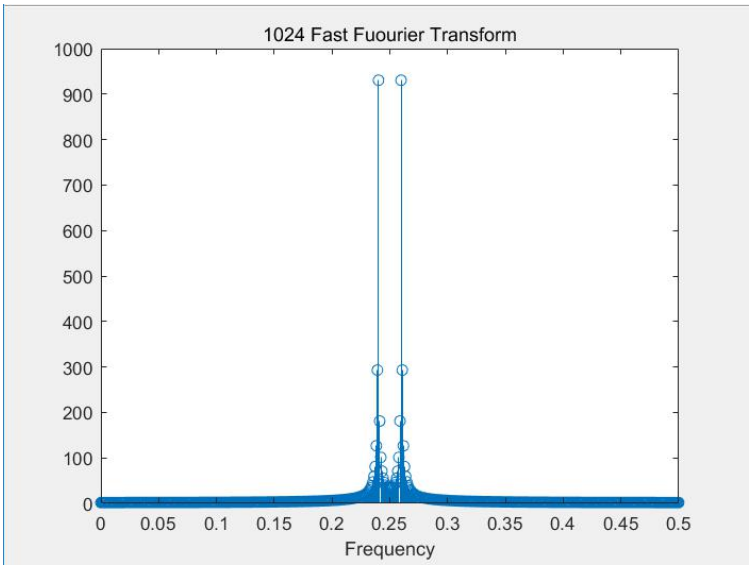


图 4.2 自己编写的 1024 点 FFT 产生的频域信号

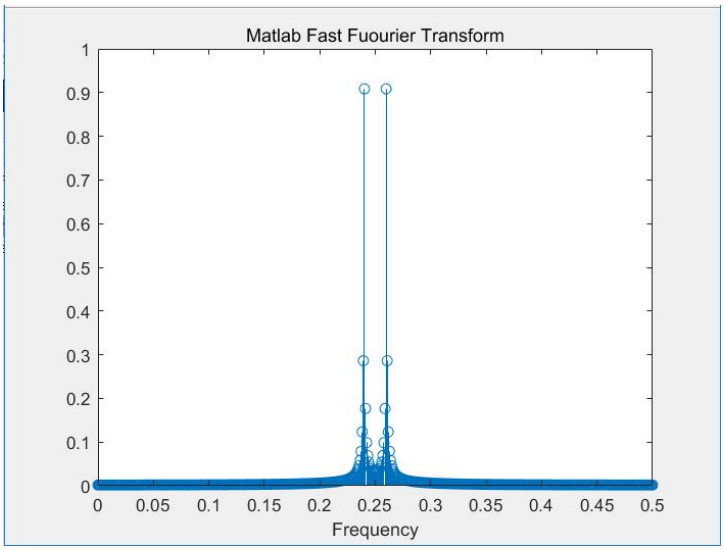
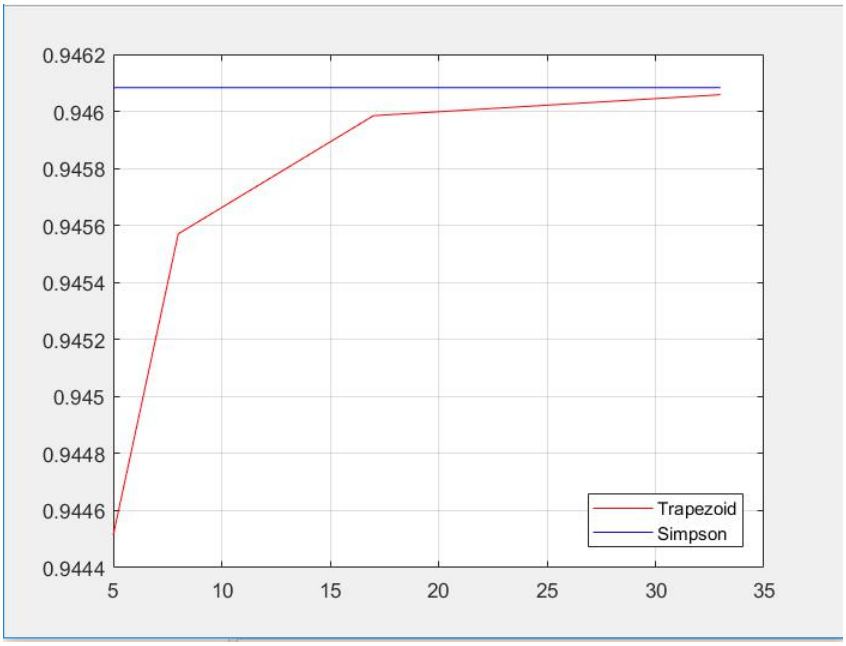


图 4.3 Matlab 的 FFT 函数输出结果

分析：对于 FFT 算法输出的频域信号，我们可以清楚地看到其对称性，和两个不同的峰值，实际上正好对应着两个正弦基函数生成的原始信号，完成了从时域到频域的信号变换，问题 4 成功解决。

五、对问题五复合梯形公式和复合辛普森公式的数值实验与分析

对于两种数值积分求解方法，首先我们进行的是关于采样点数目变化，两者的求解积分值的变化实验，以下给出实验图：



实际上我们从算法设计部分就可以得知，复合辛普森公式的计算精度要比复合梯形公式优秀，这也是为何复合辛普森公式在一开始采样点数为 5 个时就很好地，稳定地固定在精确解 $I = 0.9460831$ 附近。考虑到问题 5 更多地是在询问两种计算方法在不同采样点数目时的计算结果，以下给出运算结果图：

```
>> Tn

Tn =

   0.944513521665390
   0.945570776256246
   0.945985029934386
   0.946058560962768
```

复合梯形公式积分计算结果

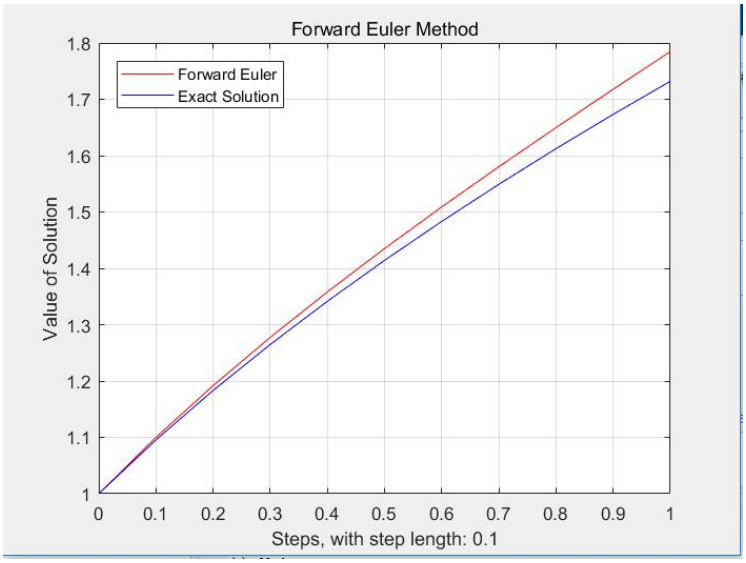
```
>> Sn

Sn =

   0.946083310881538
   0.946083095987137
   0.946083071305128
   0.946083070425720
```

复合辛普森公式积分计算结果

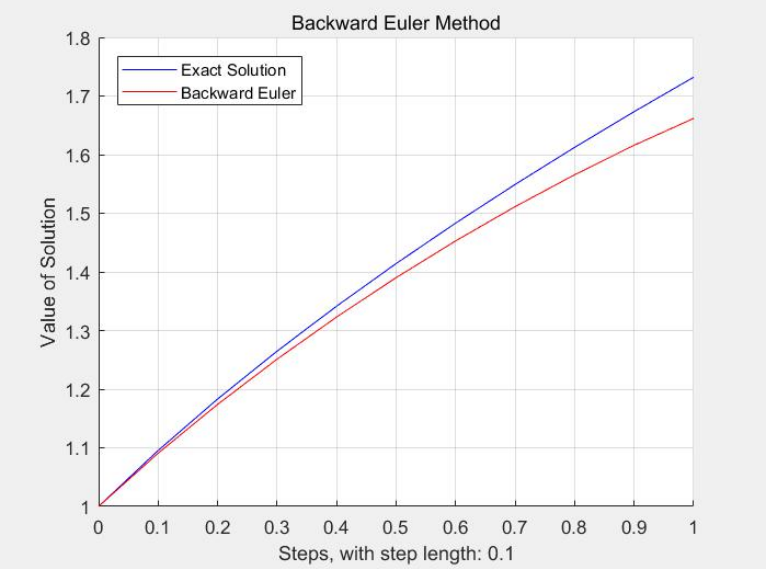
分析：上述运算结果对应的采样点数目分别为：5,9,17,33，可以看到复合梯形公式的计算结果是逐步逼近精确解，但收敛速度慢，比不上一开始就十分接近精确解的复合辛普森公式。

六、对问题六四种求解常微分方程初值问题的数值实验与分析

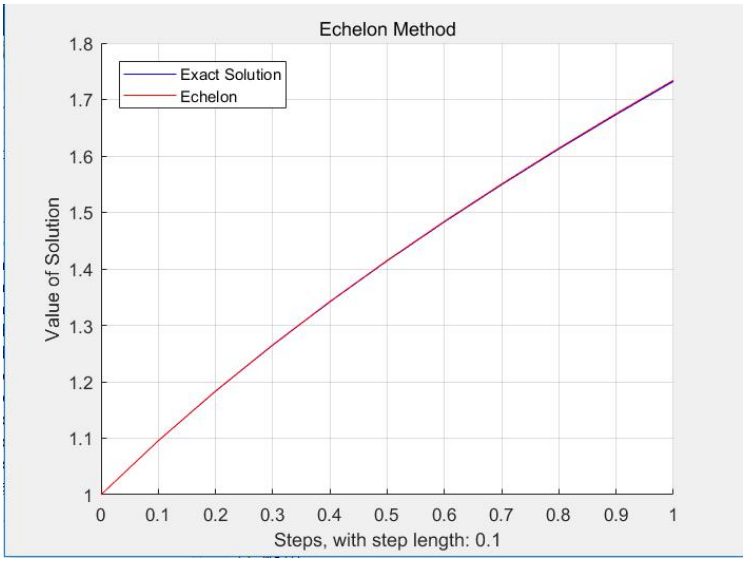
关于问题 6 的常微分方程初值问题，我们首先结合题目所给条件，输入到编写好的四种方法函数中，首先给出四种方法在横坐标为每一步长时，纵坐标为所得数值时，求解曲线跟已知的精确解之间的比较实验图：



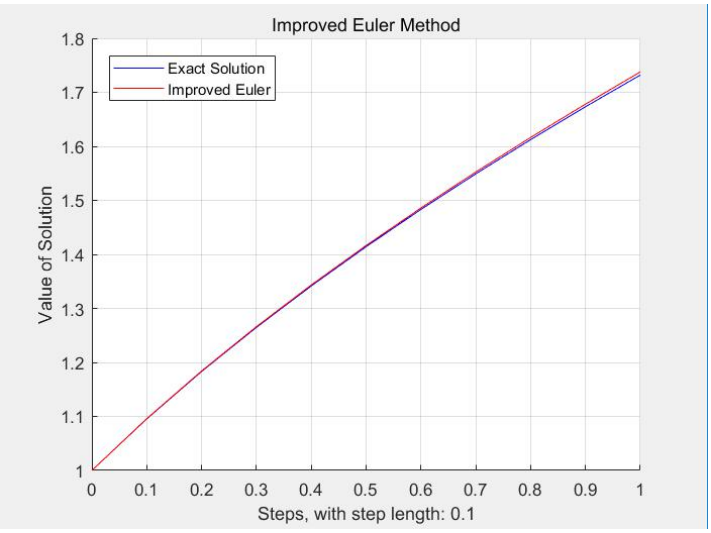
前向欧拉法每一步解与精确解比较图



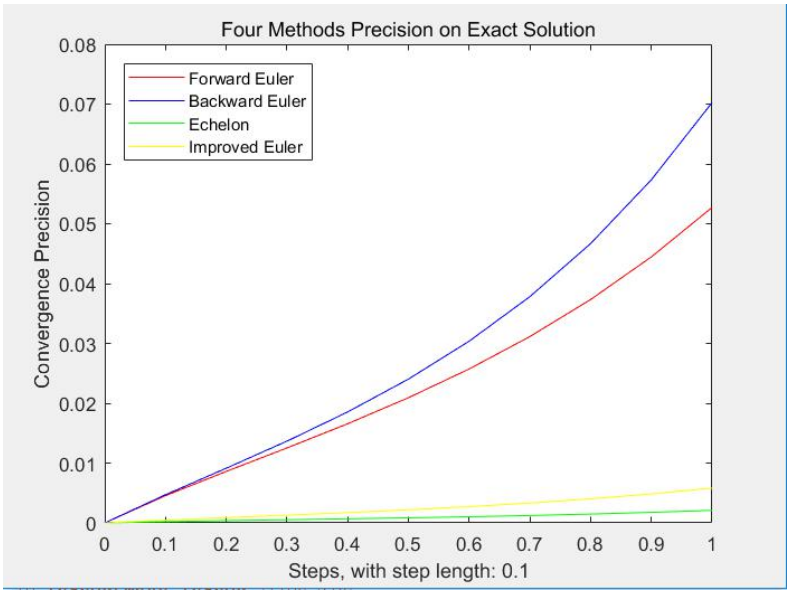
后向欧拉法每一步解与精确解比较图



梯形法每一步解与精确解比较图



改进欧拉法每一步解与精确解比较图



四种方法与精确解之间的误差曲线，横坐标为每一步

分析：我们可以由上述图曲线看到，前向欧拉法和后向欧拉法作为传统的欧拉解常微分方程解法，实际上收敛到精确解的速度和精度都是比较差的。而改进的梯形法和改进欧拉法基本上可以在每一步都跟精确解曲线相互重合，收敛速度快精度好。再结合最后的精度比较图，前/后向欧拉法的误差实际上还是挺高，梯形法和改进欧拉法则具有很小的误差，横坐标为每一步执行的 x 值。

考虑到问题 6 更多地是在询问常微分方程的解，最后给出四种方法的运行结果图，从上到下的结果分别是每一步的求解方程所得值，左右顺序与题目相同：

```
>> yf

yf =

    1.0000
    1.1000
    1.1918
    1.2774
    1.3582
    1.4351
    1.5090
    1.5803
    1.6498
    1.7178
    1.7848
```

```
>> yb

yb =

    1.0000
    1.0907
    1.1741
    1.2512
    1.3231
    1.3902
    1.4529
    1.5114
    1.5658
    1.6160
    1.6618
```

```
>> ye

ye =

    1.0000
    1.0957
    1.1836
    1.2654
    1.3423
    1.4151
    1.4843
    1.5504
    1.6139
    1.6751
    1.7341
```

```
>> yi

yi =

    1.0000
    1.0959
    1.1841
    1.2662
    1.3434
    1.4164
    1.4860
    1.5525
    1.6165
    1.6782
    1.7379
```