数值计算实验报告

1. 使用插值法完成sin(0.35)的计算

1.1 问题描述

> 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。

1.2 算法分析

1.2.1 线性插值

1.2.1.1 线性插值法定义

线性插值是一种较为简单的插值方法，其插值函数为一次多项式。

线性插值在各插值节点上插值的误差为 0 。

设函数 f(x) 在两点上x_0,x_1上的值分别为y_0,y_1,求多项式

$$y = \varphi_1(x) = a_0 + a_1 x$$

使满足

$$\varphi_1(x_0) = y_0, \varphi_1(x_1) = y_1$$

由解析几何可知

$$y = \varphi_1(x) = y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0)$$

称$\frac{f(x_j) - f(x_i)}{x_j - x_i}$为 $f(x)$在 x_i , x_j 处的一阶均差，记以$f(x_i，x_j)$。于是，得

$$\varphi_1(x) = f(x_0) + f(x_0, x_1)(x - x_0)$$

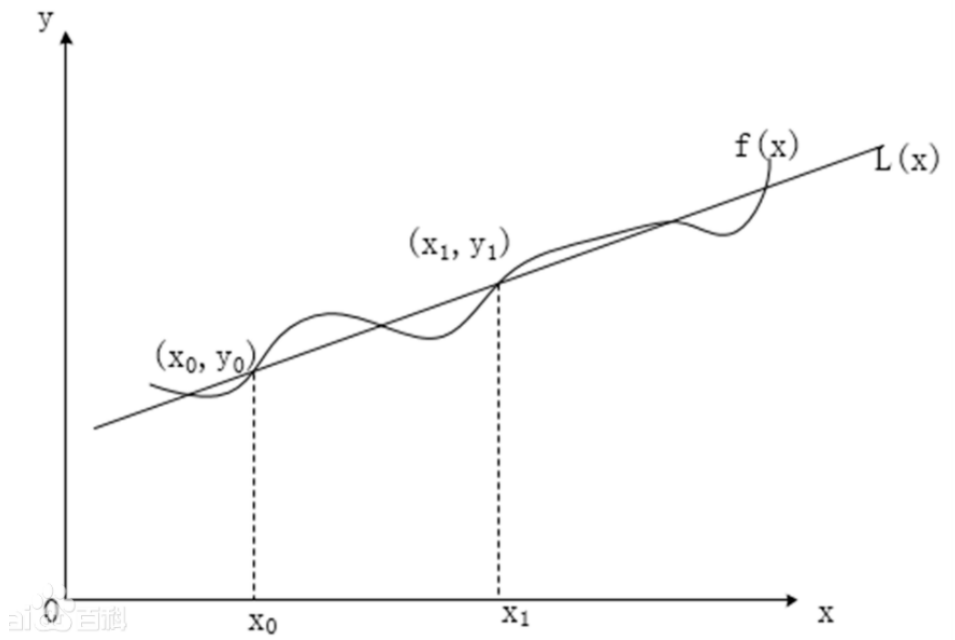
如果按照 y_0 , y_1整理，则

$$\varphi_1(x)=\frac{x-x_1}{x_0-x_1}y_0+\frac{x-x_0}{x_1-x_0}y_1$$

以上插值多项式为一次多项式，这种插值称为线性插值。

1.2.1.2 线性插值法几何意义

> 线性插值的几何意义如下图所示，即为利用过点 (x_0,y_0) , (x_1,y_1) 的直线 $L(x)$ 来近似原函数$f(x)$



1.2.2 多项式插值法（拉格朗日插值法）

1.2.2.1 多项式插值法定义

在平面上有 (x_0,y_0) , (x_1,y_1) , ... , (x_{n-1},y_{n-1}) 共n个点，现作一条函数$f(x)$使其图像经过其n个点。

设集合D_n是关于点(x,y)的角标的集合，$D_n=\{0,1,.....,n-1\}$,作n个多项式$p_j(x)$, j $\in D_n$ 。对于任意的 $k \in D_n$ 都有 $p_k(x)$,B_k = $\{i|i\neq k,i\in D_n\}$

使得

$$p_k(x)=\prod_{i\in B_k}\frac{x-x_i}{x_k-x_i}$$

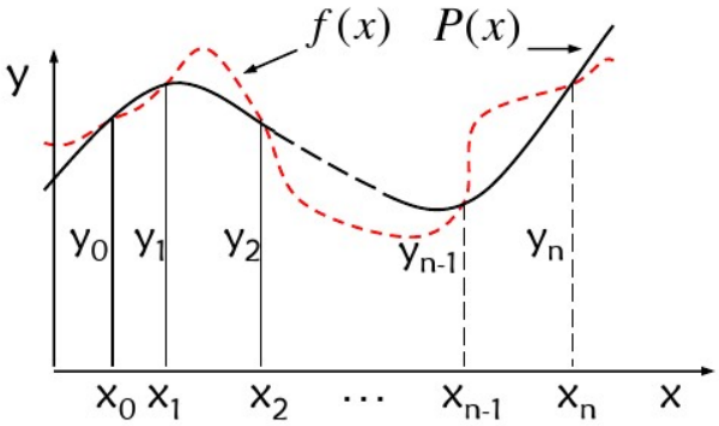
这里$p_k(x)$是 n - 1 次多项式，且满足$\forall m\in Ik,p_k(x_m)=0$并且$p_k(x_k)$ = 1 。

最后可得

$$L_n(x)=\sum_{j=0}^{n-1}y_jp_j(x)$$

1.2.2.2 多项式插值法几何意义

多项式插值的几何意义如下图所示，即为利用过点 (x_0,y_0) , (x_1,y_1) ..... (x_{n-1},y_{n-1})的直线 $P(x)$ 来近似原函数 $f(x)$



1.3 程序的运行、结果、分析

1.3.1 线性插值

1.3.1.1 Matlab代码

```
function output = myFun()
    % 由于0.35位于0.34与0.36之间所以这里使用sin(0.34)与sin(0.36)的值进行线性插值计算
    sin_34 = 0.333487 ;
    sin_36 = 0.352274 ;
    sin_32 = 0.314567 ;
    sin_38 = 0.370920 ;
    y_0 = sin_34 ;
    y_1 = sin_36 ;
    x_1 = 0.36 ;
    x_0 = 0.34 ;
    %线性插值计算
    sin_35 = y_0 + ( y_1 - y_0 ) / (x_1 - x_0 ) * (0.35 - x_0 ) ;
    %真实计算值
    fprintf('\nThe computing value is %5.15f\n',sin_35);
    %实际值
    fprintf('\nThe true value is %5.15f\n',sin(0.35));
    %误差
    fprintf('\nThe error is %5.15f\n',abs(sin(0.35) - sin_35));
end
```

1.3.1.2 运行结果（线性插值与Matlab内库sin函数所求数值对比）

```
>> linear_interpolation

The computing value is 0.342880500000000

The true value is 0.342897807455451

The error is 0.000017307455451
```

> 注：
>
> 1. 第一行代表线性插值函数计算出来的值
> 2. 第二行代表Matlab内库sin函数的运算结果

> 3. 第三行代表计算值与真实值之间的误差

1.3.1.3 分析

通过真实值与计算值之间的对比我们发现，在使用线性插值求值时候，我们可以把误差值控制在10^{-5}这个数量级上，但是误差相对来说还是较大，但是总的来说线性插值还是一个相对比较简单且比较好的可以用于逼近真实值的算法。

1.3.2 二次插值

1.3.2.1 Matlab代码

```
  % 由于0.35位于0.34与0.36之间且接近0.33所以这里使用sin(0.33),sin(0.34)与sin(0.36)的值进行二次插值计算
    sin_35 = y_0 * ((x - x_1) * (x - x_2) / ((x_0 - x_1)*( x_0 - x_2 )) ) +
    ( y_1 * (x - x_0) * (x - x_2) / ((x_1 - x_0)*( x_1 - x_2)) ) +
    ( y_2 * (x - x_0) * (x - x_1) / ((x_2 - x_0)*( x_2 - x_1 )) ) ;
  % 与线性插值相同的代码不再显示
```

1.3.2.2 运行结果（二次插值与Matlab内库sin函数所求数值对比）

```
>> quadratic_interpolation

The computing value is 0.342897125000000

The true value is 0.342897807455451

The error is 0.000000682455451
```

> 注：
>
> 1. 第一行代表二次插值函数计算出来的值
> 2. 第二行代表Matlab内库sin函数的运算结果
> 3. 第三行代表计算值与真实值之间的误差

1.3.2.3 分析

通过真实值与计算值之间的对比我们发现，在使用二次插值求值时候，我们已经可以把误差值控制在10^{-7}这个数量级上，这个误差相对线性插值算法已经获得了10^{2}的缩小，二次插值已经可以较好的达成我们所要求的标准，同时二次插值比一次插值的计算量有所提升，程序在运行时候所花费的时间也会有所提升，但是求解的精确度会大大提升。

1.3.3 三次插值

1.3.3.1 Matlab代码

```
%在进行三次插值计算时候，所有给定的点都需要用上
    sin_35 =
    y_0 * ( ( x - x_1 ) * ( x - x_2 ) * ( x - x_3 ) / ( ( x_0 - x_1 ) * ( x_0 - x_2 ) * ( x_0 -
x_3 ) ) ) +
    y_1 * ( x - x_0 ) * ( x - x_2 ) * ( x - x_3 ) / ( ( x_1 - x_0 ) * ( x_1 - x_2 ) * ( x_1 -
x_3 ) )  +
    y_2 * ( x - x_0 ) * ( x - x_1 ) * ( x - x_3 ) / ( ( x_2 - x_0 ) * ( x_2 - x_1 ) * ( x_2 -
x_3 ) ) +
    y_3 * ( x - x_0 ) * ( x - x_1 ) * ( x - x_2 ) / ( ( x_3 - x_2 ) * ( x_3 - x_1 ) * ( x_3 -
x_0 ) ) ;
```

1.3.3.2 运行结果（三次插值与Matlab内库sin函数所求数值对比）

```
>> Cubic_interpolation

The computing value is 0.342897625000000

The true value is 0.342897807455451

The error is 0.000000182455451
```

> 注:
>
> 1. 第一行代表三次插值函数计算出来的值
> 2. 第二行代表Matlab内库sin函数的运算结果
> 3. 第三行代表计算值与真实值之间的误差

1.3.3.3 分析

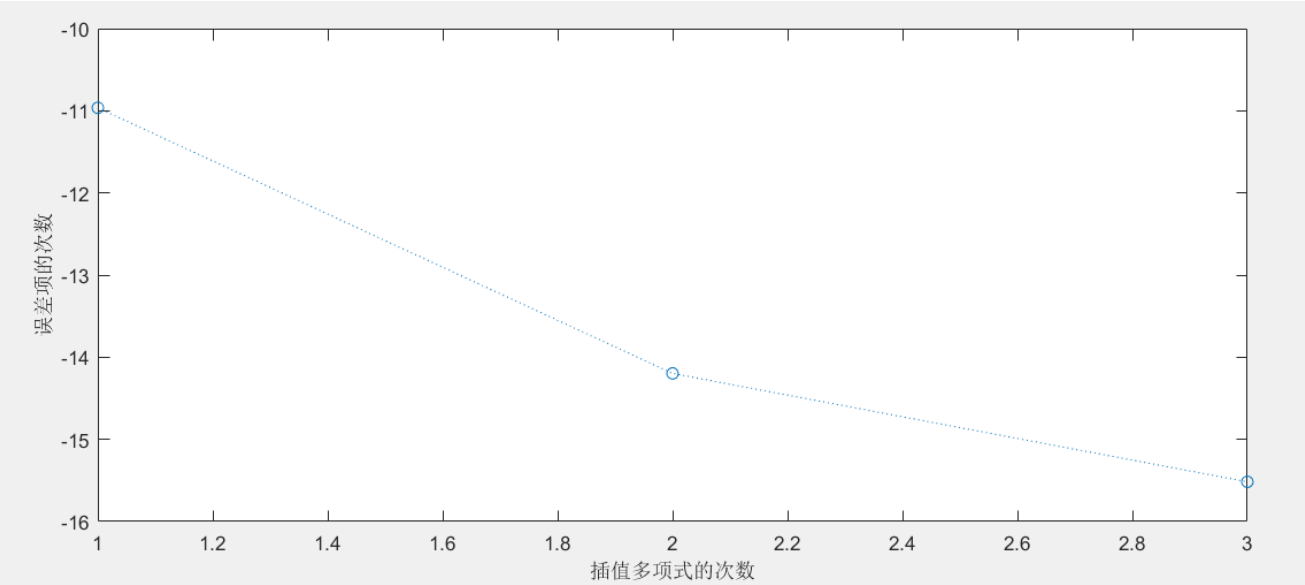
对比实验结果，我们发现三次插值计算的误差是停留在10^{-7}与二次插值相比并没有太大数量级的缩小，但是三次插值程序的计算量已经是变得非常的庞大，同时从课本上的例子中我们也可以看出，从二次插值过后，随着插值多项式的项数提升，精度的提升速度会逐渐缩小。

1.4 线性插值，一次差值，二次插值误差数量级的变动对比

1.4.1 Matlab代码

```
error = zeros(1,3) ;
error(1,1) = linear_interpolation ;
error(1,2) = quadratic_interpolation ;
error(1,3) = Cubic_interpolation ;
%使用以e为底的自然对数放大误差，便于计算
plot(1:1:3,log(error),'o:');
xlabel('插值多项式的次数');
ylabel('误差项的次数')
```

1.4.2 运行结果



1.4.3 分析

通过图示我们也就很明显的可以看出，插值计算结果次数与误差之间的变化关系，从一次插值到二次插值，误差的数量级有明显提升，但是从二次到三次误差数量级的变化就相对变得平和了许多。

---

2. 计算$\sqrt{115}$的平方根

2.1 问题描述

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。

2.2 算法分析

2.2.1 二分法

给定精确度 ξ ,用二分法求函数 $f(x)$ 零点近似值的步骤如下:

1. 确定区间 $[a,b]$,验证 $f(a)\cdot f(b)<0$,给定精确度 ξ .
2. 求区间 (a,b) 的中点 c .
   (1) 若 $f(c)=0$, 则c就是函数的零点;
   (2) 若 $f(a)\cdot f(c)<0$,则令 $b=c$;
   (3) 若 $f(c)\cdot f(b)<0$,则令 $a=c$.

(4) 判断（b - a）是否达到精确度 ξ :即若$|a-b|<\xi$, 则得到零点近似值a(或 b),否则重复2-4.

2.2.2 牛顿法

给定精确度 ξ ,用牛顿法求函数$f(x)$零点近似值的步骤如下:

1. 设 r 是$f(x)=0$的根，
2. 选取 x_0作为 r 的初始近似值
3. 过点$(x_0, f(x_0))$ 做曲线 $y=f(x)$的切线 L , L 的方程为$y=f(x_0)+f'(x_0)(x-x_0)$，求出L与x轴交点的横坐标 $x_1=x_0-\dfrac{f(x_0)}{f'(x_0)}$，称$x_1$为 r 的一次近似值。
4. 过点$(x_1, f(x_1))$做曲线 $y=f(x)$的切线，并求该切线与 x轴交点的横坐标 $x_2=x_1-\dfrac{f(x_1)}{f'(x_1)}$，称 x_2为r的二次近似值。
5. 判断x_n-x_{n-1}是否达到精度，如果达到，退出程序，否则重复$x_{n-1}=x_n$以上过程

2.2.3 简化牛顿法

简化牛顿法与牛顿法大体相同，唯一不同之处就是在进行迭代时候$f'(x)$的值不使用当前迭代步骤下的$f'(x_k)$而使用初始值$f'(x_0)$，这样就可以简化计算量。

2.2.4 弦截法

给定精确度 δ ,用弦截法求函数$f(x)$零点近似值的步骤如下:

1. 记第k 次弦截法下的近似根为 x_k ，如果 $f(x_k)=0$ ，那么 x_k 就是方程的根，迭代结束，且x_k的迭代公式如下:

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

2. 如果$f(x_k)\neq 0$ ，根据 $f(x_k)$的正负号按新求出照如下规则操作：如果 $f(x_k)<0$ ，则将x_k记为a （将x_k赋值给a ），反之将x_k记为b ，可以得到新的缩小了的含根区间 $[a,b]$
3. 判断精度要求是否得到满足，如果 $b-a<\delta$（δ 为给定的精度要求），迭代结束。
4. 如果精度要求没得到满足，那么在新的含根区间 $[a,b]$ 上继续用弦截法求出$k+1$ 次近似解

2.3 程序的运行、结果、分析

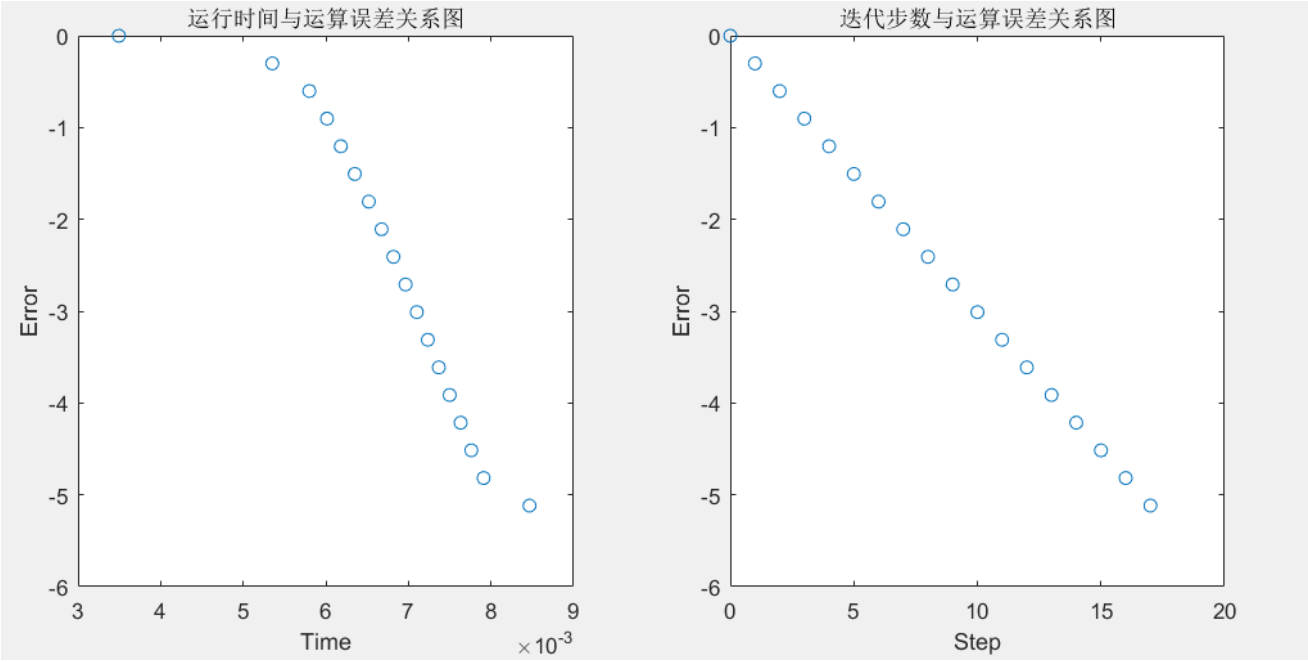
2.3.1 二分法

2.3.1.1 Matlab代码

```
function output = myFun()
    x_a = 10 ; %左区间
    x_b = 11 ; %右区间
    time = zeros(1,20) ;%记录时间
    error = zeros(1,20) ;%记录误差
    step = zeros(1,20) ;%记录步数
    for k = 1 : 1 : 20
        step(1,k) = k - 1 ;
    end
```

```
    for  i = 1:1:20
        tic
        x_b-x_a
        i
        if( x_b - x_a < 10e-6)
            error(1,i) = x_b - x_a ;
            time(1,i) = time(1,i-1) + toc ;
            break ;
        end
        error(1,i) = x_b - x_a ;
        x_c = ( x_a + x_b ) / 2 ;
        if( ( x_c * x_c - 115 ) * ( x_a * x_a - 115 ) < 0 )
            x_b = x_c ;
        end
        if( ( x_c * x_c - 115 ) * ( x_a * x_a - 115 ) > 0 )
            x_a = x_c ;
        end
        if( ( x_c * x_c - 115 ) * ( x_a * x_a - 115 ) == 0 )
            break
        end
        if ( i == 1 )
            time(1,i) = toc ;
        end
        if ( i > 1 )
            time(1,i) = time(1,i-1) + toc ;
        end
    end
    subplot(1,2,1);
    plot(time,log10(error),'o') ;
    title('运行时间与运算误差关系图');
    xlabel('Time');
    ylabel('Error');
    subplot(1,2,2);
    plot(step,log10(error),'o') ;
    title('迭代步数与运算误差关系图');
    xlabel('Step');
    ylabel('Error');
end
```

2.3.1.2 运行结果



> 注：
>
> 1. 图一：迭代时间与迭代误差的关系图
>
> 2. 图二：迭代步数与迭代误差的关系图
>
> 3. 迭代时间与迭代误差图中第一个点与第二个点之间横坐标差距过大是因为`在刚启动tic,toc进行时间统计时候会对进行到该点的程序进行一个整体的时间记录，所以也就导致第一个点与第二个点之间的时间间隔较大，而由于后面进行迭代过程的时间记录仅是for循环内部进行的，所以时间间隔也就较小。`

2.3.1.3 分析

1. 从两幅图关系图上，我们可以发现，二分法即使在最小误差限度为10^{-6}级别约束条件下，其求方程解的收敛速度仍然是比较缓慢的，这是由于我们进行二分法求值时操作次数是$log_2(N)$级别的（注：N为收敛精度），当我们的收敛精度在10^{-1}次方上进行变化时候，求值的操作次数也会随之进行常数级别（$O(1)$）的增加；

2. 另外，当我们的求值区间内如果`没有所要的根的时候`，我们的求值迭代步数虽然会在达到收敛精度时候停下来，但是我们并不能得到理想的求值结果，这也是二分法一个比较致命的缺陷。

3. 当我们已知求值区间以及求值精度时候，二分法仍是我们首选的一个求值的方法。

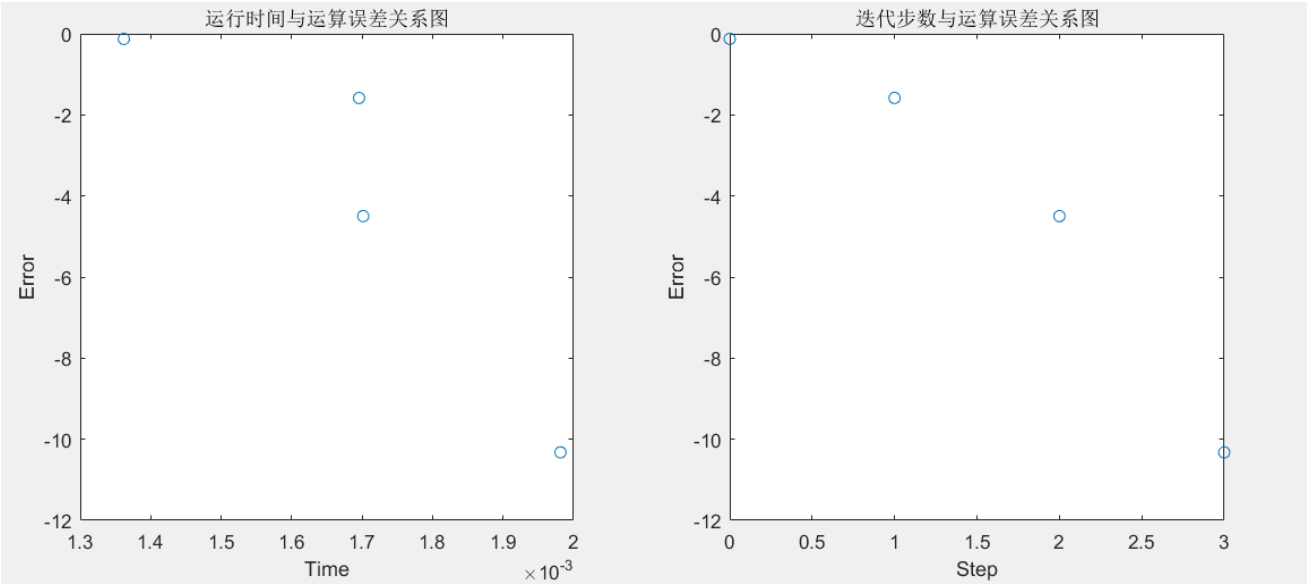
2.3.2 牛顿法

2.3.2.1 Matlab代码

```
function output = myFun(start_number)
    x = start_number ;
    x_k = ( x + 115 / x )/2 ;
    time = zeros(1,10) ;%记录时间
    error = zeros(1,10) ;%记录误差
    step = zeros(1,10);%记录步数
    for i = 1:1:10
        step(1,i) = i - 1 ;
    end
    for i = 1:1:10
        tic
```

```
        if( abs( x_k - x ) < 10e-6 )
            time(1,i) = time(1,i-1) + toc ;
            error(1,i) = abs(x_k - x) ;
            break ;
        end
        error(1,i) = abs(x_k - x) ;
        x = x_k ;
        x_k = ( x + 115 / x ) /2 ;
        if( i == 1)
            time(1,i) = toc ;
        end
        if(i > 1)
            time(1,i) = time(1,i-1) + toc ;
        end
    end
    subplot(1,2,1);
    plot(time,log10(error),'o') ;
    title('运行时间与运算误差关系图')
    xlabel('Time');
    ylabel('Error');
    subplot(1,2,2);
    plot(step,log10(error),'o') ;
    title('迭代步数与运算误差关系图')
    xlabel('Step');
    ylabel('Error');
end
```

2.3.2.2 运行结果

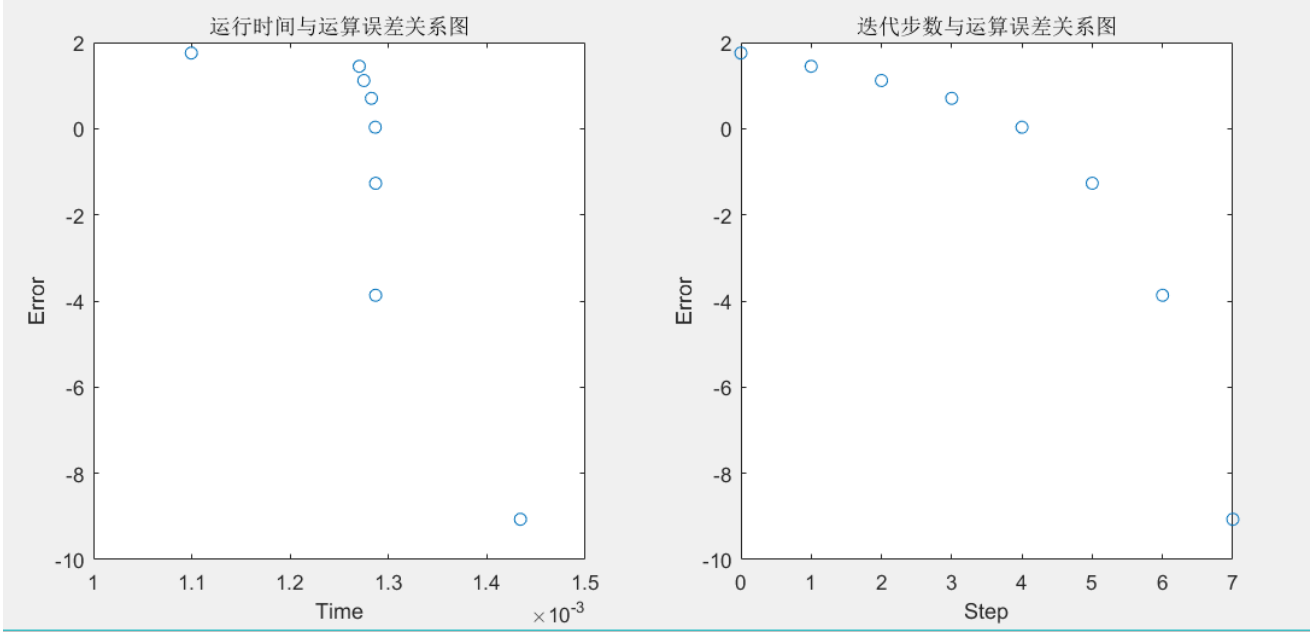


> 注：
>
> 1. 图一：迭代时间与迭代误差的关系图
> 2. 图二：迭代步数与迭代误差的关系图

> 3. 迭代时间与迭代误差图中第一个点与第二个点之间横坐标差距过大是因为`在刚启动tic,toc进行时间统计时候会对进行到该点的程序进行一个整体的时间记录，所以也就导致第一个点与第二个点之间的时间间隔较大，而由于后面进行迭代过程的时间记录仅是for循环内部进行的，所以时间间隔也就较小。`

2.3.2.3 分析

从牛顿法的形式 $x_{k+1} = x_k - f(x_k)/f'(x_k)$上我们可以看出，牛顿法是通过求出`过迭代点的直线与横轴的交点`来逼近真实值；同时牛顿法的收敛速度是依赖于初始点的选择（如图为当`起始点为1`时的收敛曲线），所以通过上面与该图的对比我们可以知道，当我们选择初始点较好时，也就能相对快的达到目的，但是当起始点选择的优越性较低时候，我们的迭代步数与迭代时间会增加。



同时，我们从2.2.2.2收敛图上可以看出，牛顿法仅使用 3 步就达到了我们的目标，所以牛顿法的收敛速度相比二分法有了极大的提高，同时收敛时间也相对于二分法提升了`100`倍，牛顿法也是我们求值较为常用且效率较高的算法。

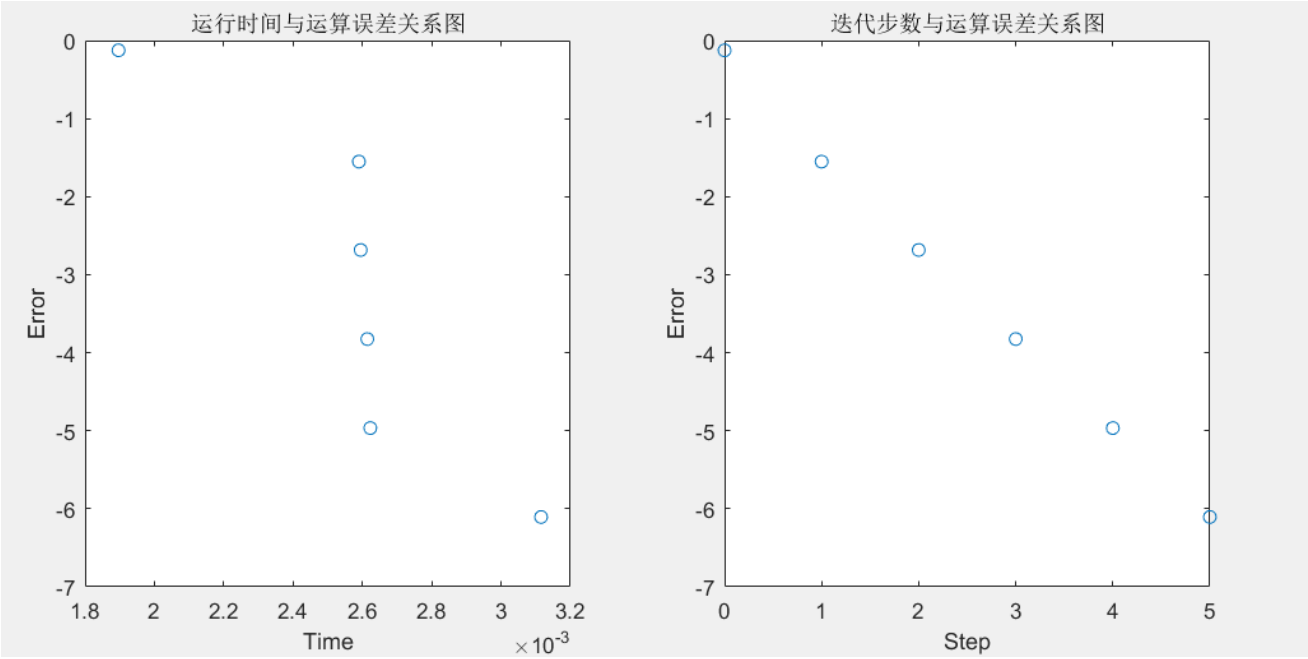
2.3.3 简化牛顿法

2.3.3.1 Matlab 代码

```
function output = myFun(start_number)
    x = start_number ;
    x_k = x - ( x * x - 115 ) / (2 * start_number) ;
    time = zeros(1,10) ;%记录时间
    error = zeros(1,10) ;%记录误差
    step = zeros(1,10);%记录步数
    for i = 1:1:10
        step(1,i) = i - 1 ;
    end
    for i = 1:1:10
        tic
        if( abs( x_k - x ) < 10e-6 )
            error(1,i) = abs(x_k - x) ;
            time(1,i) = time(1,i-1) + toc ;
            break ;
        end
```

```
        error(1,i) = abs(x_k - x) ;
        x = x_k ;
        x_k = x - ( x * x - 115 ) / (2 * start_number) ;
        if( i == 1)
            time(1,i) = toc ;
        end
        if(i > 1)
            time(1,i) = time(1,i-1) + toc ;
        end
    end
    subplot(1,2,1);
    plot(time,log10(error),'o') ;
    title('运行时间与运算误差关系图')
    xlabel('Time');
    ylabel('Error');
    subplot(1,2,2);
    plot(step,log10(error),'o') ;
    title('迭代步数与运算误差关系图')
    xlabel('Step');
    ylabel('Error');
end
```

2.3.3.2 运行结果

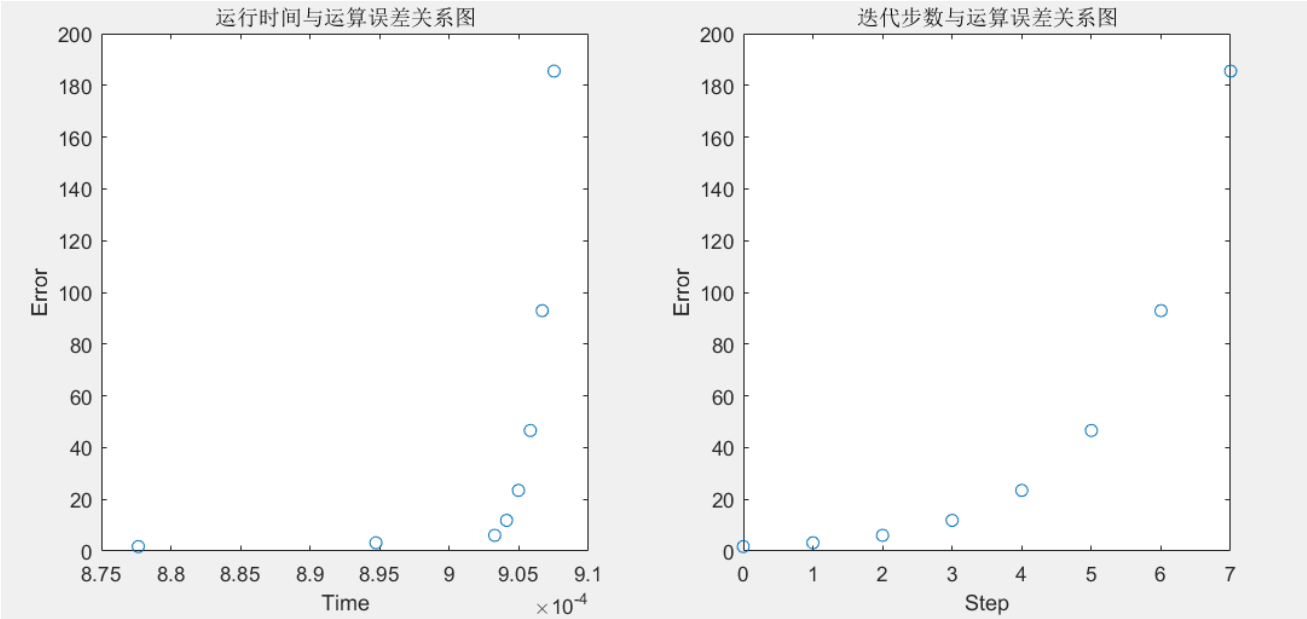


> 注：
>
> 1. 图一：迭代时间与迭代误差的关系图
> 2. 图二：迭代步数与迭代误差的关系图
> 3. 迭代时间与迭代误差图中第一个点与第二个点之间横坐标差距过大是因为`在刚启动tic,toc进行时间统计时候会对进行到该点的程序进行一个整体的时间记录，所以也就导致第一个点与第二个点之间的时间间隔较大，而由于后面进行迭代过程的时间记录仅是for循环内部进行的，所以时间间隔也就较小。`

2.3.3.3 分析

简化牛顿法相对牛顿法来讲是使用$f'(x_0)$代替的值代替$f'x_k)$的值，这样的改动之后我们可以简化我们在迭代过程中的计算量，但是与牛顿法求解的关系图对比，简化牛顿法在求解的迭代的步数以及时间上上并没有很明显的优化（这里我觉得可能是因为我们所要求解的方程比较简单，优化的效果不明显）。

在经历这样的改动之后，我们在进行求值时候就会更加依赖初始值，例如当我们本题选择起始值$x_0 = 1$时，以上的关系图就会变为以下情况：



我们从这个图上可以看出，当我们选择一个较差的初始值时候，迭代过程中求值精度反而不再收敛，所以这就显示出了简化牛顿法对迭代初始值的高依赖性，当我们选择一个离所求的值差距较大的初始值时候，我们反而得不到自己想要的结果。

所以我们使用简化牛顿法的时候一定要认真考虑初始值的选择。

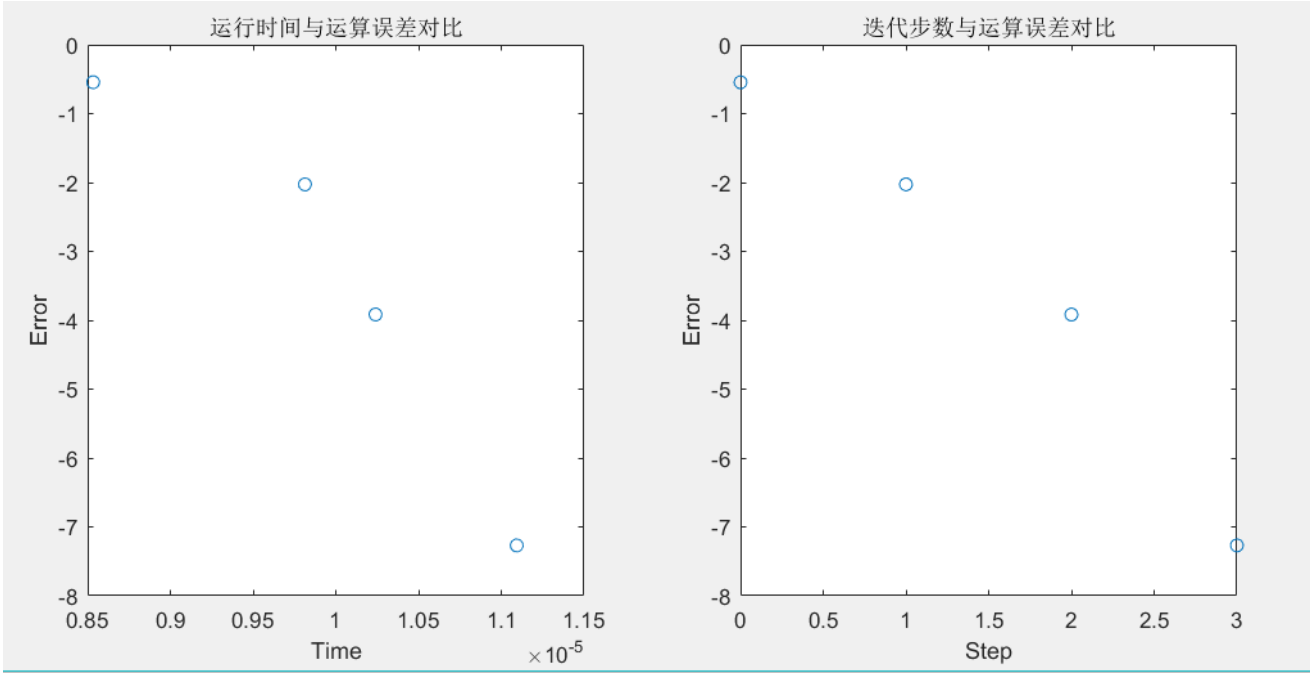
2.3.4 弦截法

2.3.4.1 Matlab代码

```
function output = myFun(first_number,second_number)
    x_1 = first_number ;
    x_2 = second_number ;
    x_k = x_2 - ( x_2 * x_2 - 115 ) /  (x_2 * x_2 - x_1 * x_1) * (x_2 - x_1) ;
    time = zeros(1,10) ;%记录时间
    error = zeros(1,10) ;%记录误差
    step = zeros(1,10);%记录步数
    for i = 1:1:10
        step(1,i) = i - 1 ;
    end
    for i = 1:1:10
        tic
        if( abs( x_k - x_2 ) < 10e-6 )
            error(1,i) = abs(x_k - x_2) ;
            time(1,i) = time(1,i-1) + toc ;
            break ;
        end
        error(1,i) = abs(x_k - x_2) ;
        x_1 = x_2 ;
```

```
        x_2 = x_k ;
        x_k = x_2 - ( x_2 * x_2 - 115 ) / (x_2 * x_2 - x_1 * x_1) * (x_2 - x_1)  ;
        if( i == 1)
            time(1,i) = toc ;
        end
        if(i > 1)
            time(1,i) = time(1,i-1) + toc ;
        end
    end
    subplot(1,2,1);
    plot(time,log10(error),'o') ;
    title('运行时间与运算误差对比')
    xlabel('Time');
    ylabel('Error');
    subplot(1,2,2);
    plot(step,log10(error),'o') ;
    title('迭代步数与运算误差对比')
    xlabel('Step');
    ylabel('Error');
end
```

2.3.4.2 运行结果



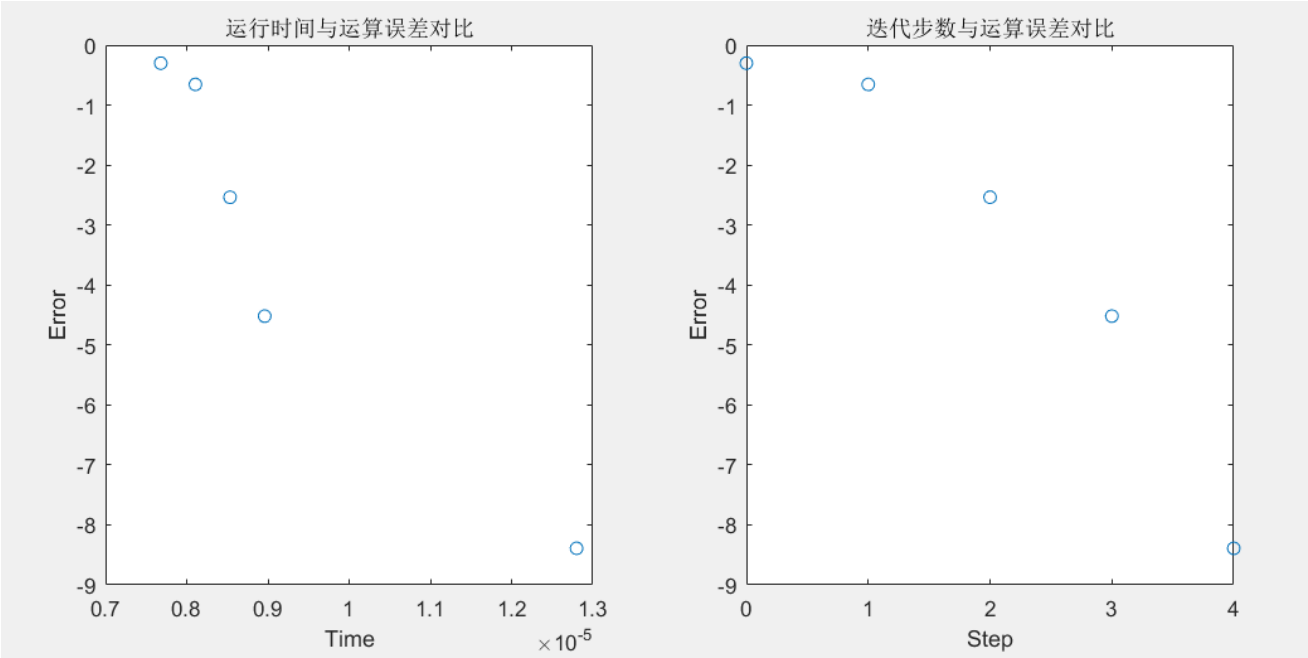
> 注：
>
> 1. 图一：迭代时间与迭代误差的关系图
> 2. 图二：迭代步数与迭代误差的关系图
> 3. 迭代时间与迭代误差图中第一个点与第二个点之间横坐标差距过大是因为`在刚启动tic,toc进行时间统计时候会对进行到该点的程序进行一个整体的时间记录，所以也就导致第一个点与第二个点之间的时间间隔较大，而由于后面进行迭代过程的时间记录仅是for循环内部进行的，所以时间间隔也就较小。`

2.3.4.3 分析

弦截法是牛顿法中$f'(x_k)$改为$f(x_k)-f(x_{k-1})/(x_k-x_{k-1})$的一个改良形式，这里弦截法由于是使用先前两个点来求第三个点的值，所以对初始值有了更高的要求。

从弦截法的表达式形式上我们可以发现，弦截法也是对求值区间有所依赖的，同时当表达式的次数较高的时候，弦截法的运算量是要比牛顿法要小的，所以在更高次数的表达式的求解上，弦截法的运行时间是会有所降低的。

为了进一步探究弦截法对求值初始值的依赖，我取了初始值为[1,11],进行迭代，得到如下图示：



从图上，我们可以看出即使是求值区间扩大了10倍，算法运行的时间也还只是提高了$1.5*10^{-6}$左右，迭代步数也只是提升了 1 步，同时求值的结果也是能够满足准确度与精度的需求，所以弦截法应该是更值得信赖且稳定性更高的算法，但是由于弦截法需要给定两个初始值，这也就一定程度上限定了弦截法的使用。

3. 递推最小二乘法求方程的根

3.1 问题描述

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 mxn 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。

3.2 算法分析

设线性方程组 $Ax=b$, 则有

$$(1) \qquad A(k,:)x=b_k,(k=1,2,.....,n)$$

其中

$$(2) \qquad A(k,:)=[a_{k_1},a_{k_2},.....,a_{k_n}],x=[x_1,x_2,.....,x_n]^T$$

设

$$(3) \qquad f(k) = A(k,:)x$$

下面采用基于递推最小二乘法(RLS)的神经网络算法来训练权值向量x，以获得线性方程组(1)的解x。

由式(3)可知，若以$f(k)$为神经网络输出，以b_k为神经网络训练样本，以x为神经网络权值向量，$A(k,:) = [a_{k1}, a_{k_2}, \ldots\ldots, a_{k_n}]$为神经网络输入向量，采用RLS算法训练神经网络权值向量x，算法如下：

Step1： 神经网络输出：

$$f(k) = A(k,:)x$$

Step2： 误差函数：

$$e(k) = b_k - f(k)$$

Step3： 性能指标：

$$J = 1/2 * (e^2(1) + e^2(2) + \ldots\ldots + e^2(k))$$

Step4： 使$J = min$的权值向量x，极为所求的神经网络权值向量x,这是一个多变量线性优化问题，为此，由:

$$\partial J / \partial x = 0$$

可得最小二乘法（RLS）：

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

$$Q^k = P^k * A^T(k,:)/(1 + A(k,:)P^k A^T(k,:))$$

$$P^{k+1} = [I - Q^k A(k,:)]P^k (k = 1, 2, \ldots\ldots n)$$

随机产生初始权值向量$x^0 = rand(n,1)$,设 $P^0 = \alpha I \in R^{n*n}$($\alpha$是足够大的正数),通过对样本数据训练，就可以获得神经网络权值向量x,即线性方程组(1)的解

3.3 程序的运行、结果、分析

3.3.1 Matlab程序设计

```
function output = myFun()
    %生成n*n的随机矩阵
    A = normrnd(10,1,10000,10) ;
    %生成n*1的随机矩矩阵
    b = normrnd(10,1,10000,1) ;
    %求解答案与后面的算法进行对比
    x_1 = A\(b) ;
    %定义临时变量用求算法的解
    x_2 = zeros(10 ,1) ;
    x_3 = zeros(10 ,1) ;
    %设定误差
    error = zeros(10000,1) ;
    step = zeros(10000,1) ;
    for i = 1 : 1 : 10000
```

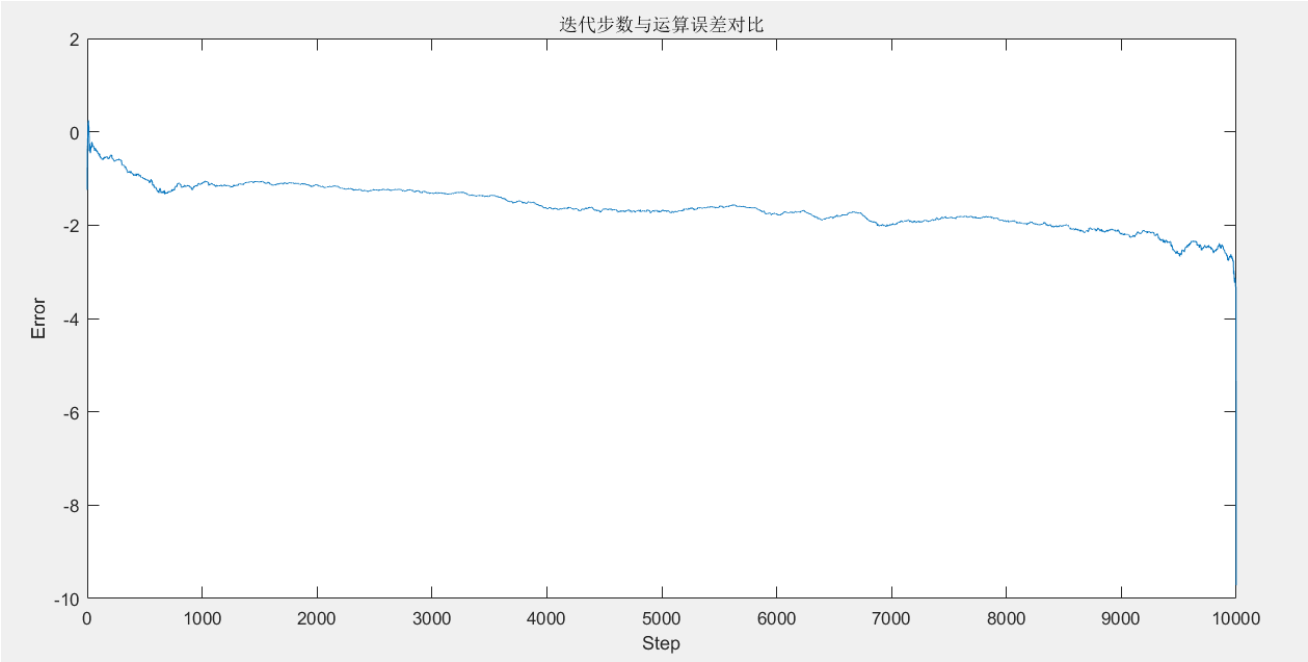
```
        step(i,1) = i - 1 ;
    end
    %给定初始p0
    p_0 = 10000 *  eye(10);
    for k = 1 : 1 : 10000
        Q_0 = ( p_0 * (A(k,:))' ) / (1 + A(k,:) * p_0 * (A(k,:))' ) ;
        p_1 = (eye(10) - Q_0*A(k,:))*p_0 ;
        p_0 = p_1 ;
        x_3 = x_2 + Q_0*(b(k,1) - A(k,:)*x_2) ;
        x_2 = x_3 ;
        error(k,1) = norm(x_3 - x_1) ;
    end
    %显示最终结果验证运算正确性
    x_3
    x_1
    plot(step,log10(error)) ;
    title('迭代步数与运算误差对比')
    xlabel('Step');
    ylabel('Error');
end
```

3.3.2 运行结果

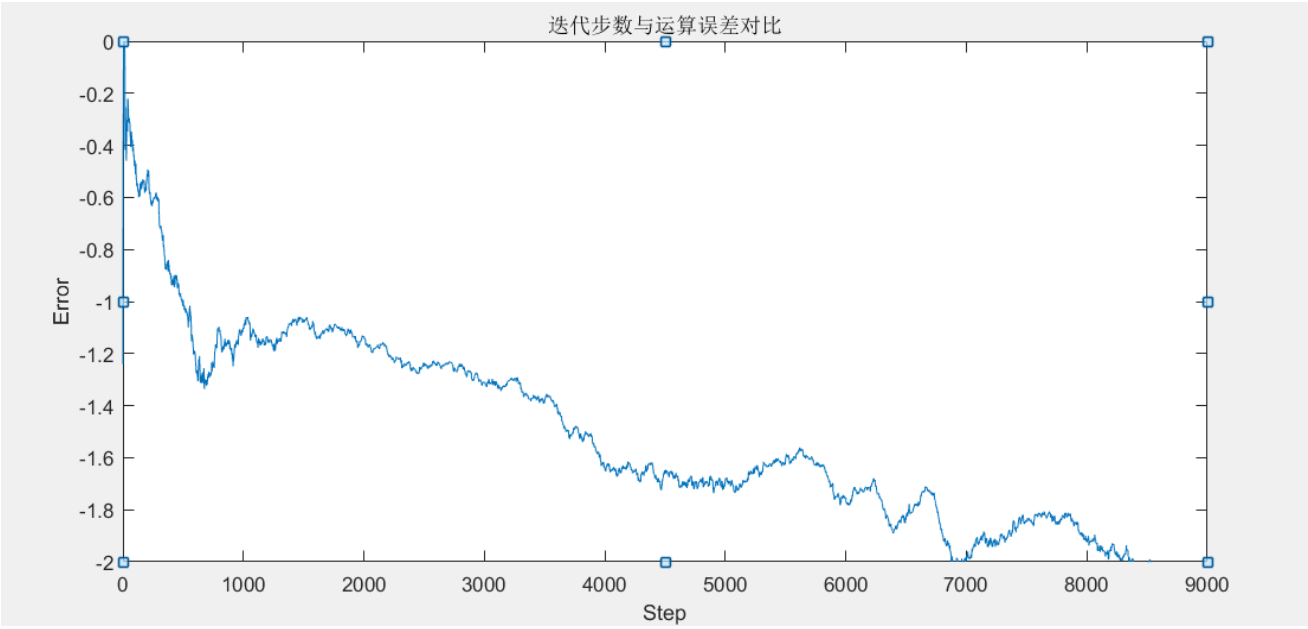
3.3.2.1 A/b结果与最小二乘法求值结果对比

```
x_3 =

   0.097457599099616
   0.105305987941284
   0.106688409988056
   0.106683955678094
   0.097188812304712
   0.104732130137861
   0.101631115094077
   0.095676265136066
   0.086737473436631
   0.098396893437039


x_1 =

   0.097457599072483
   0.105305987994213
   0.106688410055916
   0.106683955750224
   0.097188812276065
   0.104732130185536
   0.101631115112618
   0.095676265090226
   0.086737473302205
   0.098396893424267
```

3.3.2.2 迭代步数变动收敛曲线



3.3.2.3 y位于$[-2, 0]$区间的迭代变动收敛曲线



> 注:
>
> 1. `3.3.2.1图例` 的x_1代表A/b也即方程的真实解集，x_3代表最小二乘的递归解集
> 2. `3.3.2.2图例` 代表收敛精度与迭代步数的整体变动关系
> 3. `3.3.2.3图例` 代表迭代步数与收敛精度图示截取 $0 <= y <= 2$范围

3.3.3 分析

递推最小二乘法针对超定方程一类的问题进行求解。

在使用递推最小二乘法之前，我们首先接触了一般形式的最小二乘法，同时也接触了一个基于最小二乘法演变出来的批处理算法，批处理算法基本形式是:

$$a = (\Phi^T \Phi)^{-1} \Phi^T y$$

（注：这里的a也即是超定方程A的解）

批处理算法的结果是我们对给定矩阵A以及待求向量a在进行极小化噪声像时对$y-\Phi a$求偏倒数后再求梯度为0的结果。

当我们对待求的样本进行扩大时候，比如对矩阵A增加一列，那么这时候我们就用到递推最小二乘法。

从`3.2`推导的递推最小二乘法的形式上我们可以发现，递推最小二乘法在第k步递推时候，使用给定矩阵第k行的值进行变化量Δx的迭代，最后再利用$k-1$步的x值加上变化量Δx完成第k步的x值的更新，所以我们通过使用递推最小二乘法可以对待求解区域随时进行扩展并且可以将0个待测样本推广到 k待测样本。

同时，从运算复杂量上看，递推最小二乘法通过循环充分的使用了待求矩阵A的每一行，而不是类似于雅可比迭代与高斯赛德尔迭代法直接将整个矩阵带入求值进行迭代，这样也就大大简化了计算量，同时在运算时间上也是进行了优化。

但是从收敛精度图形上我们也可清楚的发现，递推最小二乘法的收敛速度是十分缓慢且十分不稳定的，这里我个人分析如下：

1. 产生的误差应该是与迭代过程中向量$A(k,:)$有关，迭代过程中在进行向量的乘积运算时候，难免容易丢失部分的浮点数；
2. 收敛速度缓慢可能是因为我们一次只改变了待求值的某一行，而非对待求值整体进行改变，所以待求值整体的变化较为缓慢；

   另外，我们进行的`10000`次的迭代过程，收敛精度会忽上忽下变化并不会保持一个相对稳定的收敛趋势，求值的精度最终也仅仅能维持到10^{-10}数量级上，这就相比传统的迭代法求方程的根稍显不足，但是我们在日常的应用当中，10^{-10}数量级的误差已经可以达到了我们允许的最大误差限度。

4. 1024点快速傅立叶变换

4.1 问题描述

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。

4.2 算法分析

4.2.1 基础概念

定义：若 W_n^k ，则称W为1的复根，也称作单位复根。单位复根必有n个，它们是

$$\sum_{k=0}^{n-1} e^{2\pi i\frac{k}{n}}$$

为了方便我们定义：

$$W_n^k = e^{2\pi i\frac{k}{n}}$$

它的性质为以下这些：

- $W_n^n = 1$,根据欧拉公式 $e^{i\pi}+1=0$, 当W_n^k中$k=n$时,

$$W_n^n = e^{2\pi i} = (e^{\pi i})^2 = 1$$

- 相消引理

$$W_{dn}^{dk} = e^{2\pi i\frac{k}{n}} = W_n^k$$

- 折半引理

$$（W_n^k）^2 = e^{2\pi i\frac{2k}{n}} = e^{2\pi i\frac{k}{\frac{n}{2}}} = W_{\frac{n}{2}}^k$$

$$W_n^{k+\frac{n}{2}} = e^{2\pi i\frac{2k+n}{2n}} = e^{(2\pi i\frac{k}{n})} * e^{\pi i} = -W_n^k$$

$$(W_n^{k+\frac{n}{2}})^2 = (-W_n^k)^2 = W_{\frac{n}{2}}^k$$

- 欧拉展开

$$W_n^k = e^{2\pi i\frac{k}{n}} = \cos(2\pi\frac{k}{n}) + i * \sin(2\pi\frac{k}{n})$$

4.2.2 FFT推导

DFT公式：

$$y_k = \Sigma_{j=0}^{n-1} x_j (e^{-2\pi i\frac{k}{n}})^j$$

首先让我们把整个公式拆掉：所有数 = 奇数 + 偶数（$j = 2r + 2r + 1$）

$$y_k = \sum_{r=0}^{\frac{n}{2}-1} x_{2r} (e^{-2\pi i\frac{2r}{n}})^k + \Sigma_{r=0}^{\frac{n}{2}-1} x_{2r+1} (e^{-2\pi i\frac{2r+1}{n}})^k$$

再把把奇数项拆开：记$x_0[r] = x[2r], x_1[r] = x[2r+1]$，可得：

$$y_k = \sum_{j=0}^{\frac{n}{2}-1} x0_r((e^{-2\pi i\frac{r}{n}})^2)^k + e^{(-2\pi i\frac{r}{n})} * \sum_{j=0}^{\frac{n}{2}-1} x1_r((e^{-2\pi i\frac{r}{n}})^2)^k$$

所以抽象出来，即为

$$FFT(x)[k] = DFT(x0)[k] + W_n^r * DFT(x1)[k]$$

所以可以根据折半引理可以把FFT写成如下形式：

$$FFT(x)[k]=DFT((W_n^r)^{2k})+W_n^r*DFT((W_n^r)^{2k}) \qquad (r\le\frac{n}{2}-1)$$

$$FFT(x)[k]=DFT((W_{\frac{n}{2}}^r)^{k})+W_n^r*DFT((W_{\frac{n}{2}}^r)^{k}) \qquad (r>\frac{n}{2}-1)$$

4.3 程序的运行、结果、分析

4.3.1 Matlab程序

4.3.1.1 自定义FFT函数

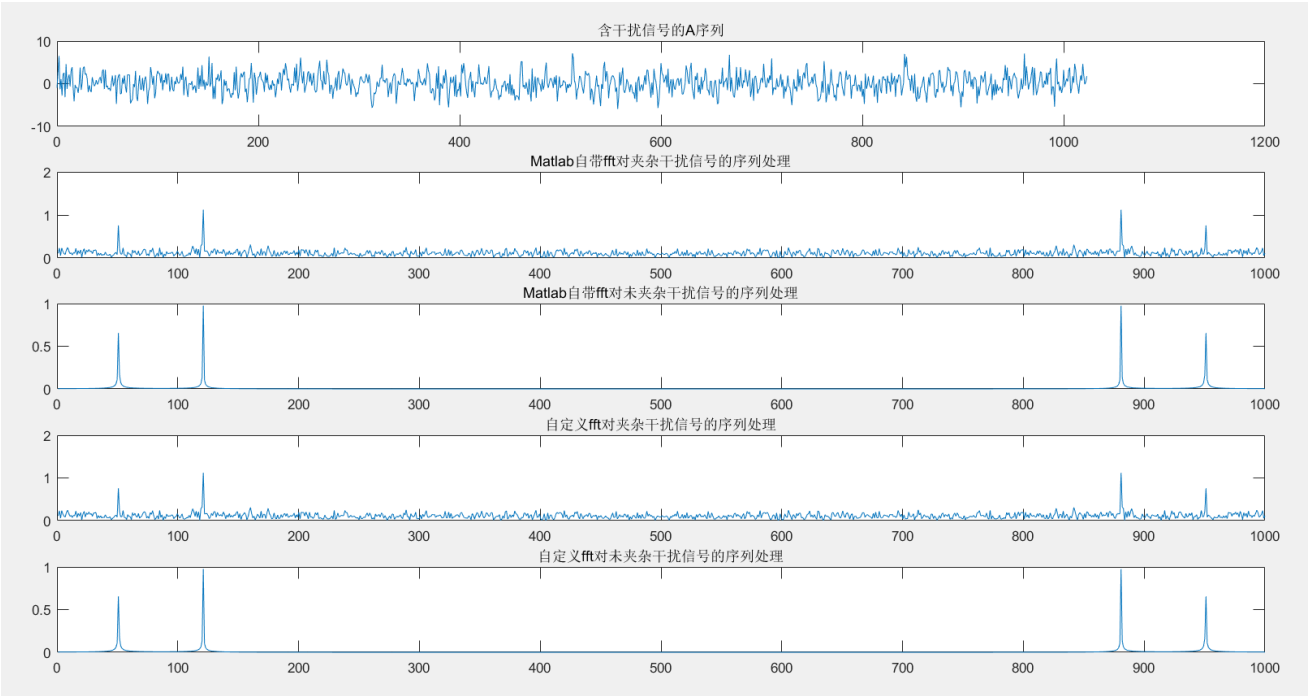
```
%传入正弦波序列A与递归层数M
function [A] = myfft(A,M)
    N=2^M;  % M 表示层数
    LH=N/2;
    J=LH;
    N1=N-2;
    %倒序程序
    A = bitrevorder(A) ;
    W = exp(-j*2*pi/N) ;
    for L=1:1:M
        B=2^(L-1);
        K = N / (2 ^ L) ;
        for k = 0 : 1 : K -1
            for J = 0 : 1 : B -1
                % 求出迭代系数
                p = J * 2 ^ (M - L) ;
                %  定位迭代序列到某一项
                q = A(k*2^L+J+1) ;
                % 前项
                A(k*2^L+J+1) = q + W ^ p * A(k*2^L+J+B+1) ;
                % 后项
                A(k*2^L+J+B+1) = q - W ^ p * A(k*2^L+J+B+1) ;
            end
        end
    end
    B=(0);
    for i=1:2^M
        B(i)=A(i);
    end
    A=B;
end
```

4.3.1.2 测试函数

```
Fs = 1000 ;
Len = 1024 ;
T = 1 / Fs ;
t = (0:Len-1) * T ;
```

```
%原序列
S =  0.7 * sin (2 * pi * 50 * t ) +  sin (2 * pi * 120 * t ) ;
%夹杂干扰信号序列
A = 0.7 * sin (2 * pi * 50 * t ) +  sin (2 * pi * 120 * t ) + 2*randn(size(t));
%限定长度
N = 1024 ;
L=length(A(:)); %输入序列长度
b=0;

%原序列
subplot(5,1,1);
plot(0:1:Len - 1,A);
title('含干扰信号的A序列');

%MATLAB自带函数fft对干扰信号求1024点FFT
subplot(5,1,2);
X1=fft(A,N);
P2 = abs(X1/Len) ;
P1 = P2(1:Len) ;
P1(2:end-1) = 2 * P1(2:end-1);
f = Fs * (1:(Len)) / L ;
plot(f,P1);
title('Matlab自带fft对夹杂干扰信号的序列处理');

%MATLAB自带函数fft对非干扰信号求1024点FFT
subplot(5,1,3);
X1=fft(S,N);
P2 = abs(X1/Len) ;
P1 = P2(1:Len) ;
P1(2:end-1) = 2 * P1(2:end-1);
f = Fs * (1:Len) / L ;
plot(f,P1);
title('Matlab自带fft对未夹杂干扰信号的序列处理');

%MATLAB自带函数fft对非干扰信号求1024点FFT
subplot(5,1,4);
X1=myfft(A,10);
P2 = abs(X1/Len) ;
P1 = P2(1:Len) ;
P1(2:end-1) = 2 * P1(2:end-1);
f = Fs * (1:Len) / L ;
plot(f,P1);
title('自定义fft对夹杂干扰信号的序列处理');

%MATLAB自带函数fft对非干扰信号求1024点FFT
subplot(5,1,5);
X1=myfft(S,10);
P2 = abs(X1/Len) ;
P1 = P2(1:Len) ;
P1(2:end-1) = 2 * P1(2:end-1);
f = Fs * (1:Len) / L ;
plot(f,P1);
title('自定义fft对未夹杂干扰信号的序列处理');
```

4.3.2 运行结果



> 注：
>
> 1. 图一为随机生成的夹杂干扰的正弦波序列信号
> 2. 图二为Matlab库中FFT函数对夹杂干扰的正弦波序列的处理结果
> 3. 图三为Matlab库中FFT函数对未夹杂干扰的正弦波序列的处理结果
> 4. 图四为自定义FFT函数对夹杂干扰的正弦波序列的处理结果
> 5. 图五为自定义FFT函数对未夹杂干扰的正弦波序列的处理结果

4.3.3 分析

快速傅立叶算法是离散傅立叶变换的快速算法，可以将一个信号变换到频域。有些信号在时域上是很难看出什么特征的，但是如果变换到频域之后，就很容易看出特征了。这就是很多信号分析采用FFT变换的原因。另外，FFT可以将一个信号的频谱提取出来。

采样得到数字信号，就可以做FFT变换。N个采样点，经过FFT之后，就可以得到N个点的FFT结果。为了方便进行FFT运算，通常N取2的整数次方,整个算法的运算量如下：

$$Nlog2N$$

为什么采样点N的个数要选用2的整数次方呢？这是因为，在FFT中，利用W_n的周期性和对称性，把一个N项序列（设N=2k,k为正整数），分为两个N/2项的子序列，每个N/2点DFT变换需要（N/2）2次运算，再用N次运算把两个N/2点的DFT变换组合成一个N点的DFT变换。这样变换以后，总的运算次数就变成
$N+2*(N/2)^2=N+N^2/2$。也就是说，FFT提高了运算速度，但是，也对参与运算的样本序列作出了限制，即要求样本数为2^N;$1024=2^{10}$满足 FFT运算要求,1000点则不满足2的整数次方这个限制条件，若采用1000点，FFT算法会在其后补零，自动补足1024点，但是，这样，被分析的样本就变了，结果误差较大。

快速傅立叶变换大大简化了在计算机中进行DFT的过程，简单来说，如果原来计算DFT的复杂度是$N*N$次运算（N代表输入采样点的数量），进行FFT的运算复杂度是$N*lg10(N)$，因此，计算一个1,000采样点的DFT，使用FFT算法只需要计算3,000次，而常规的DFT算法需要计算1,000,000次！

下面解释一下我作图的原理，这里之所以这么设置作图函数，是参考了Matlab内置内置函数FFT的使用过程，随后通过查阅资料，明白了其中的道理.

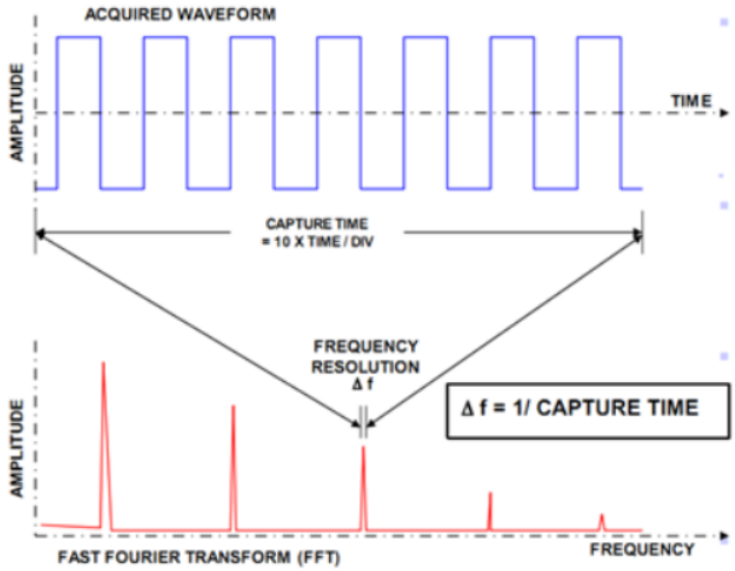
作图代码:

```
Fs = 1000 ;
Len = 1024 ;
T = 1 / Fs ;
t = (0:Len-1) * T ;
%原序列
S =  0.7 * sin (2 * pi * 50 * t ) +  sin (2 * pi * 120 * t ) ;
%夹杂干扰信号序列
A = 0.7 * sin (2 * pi * 50 * t ) +  sin (2 * pi * 120 * t ) + 2*randn(size(t));
%限定长度
N = 1024 ;
L=length(A(:)); %输入序列长度
b=0;

%举其中一个例子做说明

%MATLAB自带函数fft对干扰信号求1024点FFT
subplot(5,1,2);
X1=fft(A,N);
P2 = abs(X1/Len) ;
P1 = P2(1:Len) ;
P1(2:end-1) = 2 * P1(2:end-1);
f = Fs * (1:(Len)) / L ;
plot(f,P1);
title('Matlab自带fft对夹杂干扰信号的序列处理');
```

假设采样频率为Fs，信号频率F，采样点数为N。那么FFT之后结果就是一个为N点的复数。每一个点就对应着一个频率点。这个点的模值，就是该频率值下的幅度特性。那么这个点的幅度的特性我们可以进行下面的推导得出跟原始信号的幅度的关系。

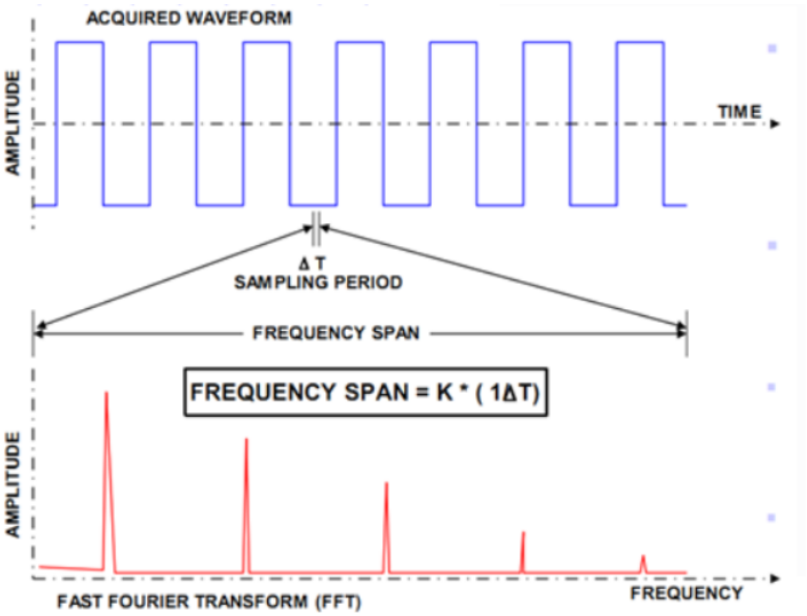
假设原始信号的峰值为A，那么FFT的结果的每个点（除了第一个点直流分量之外）的模值就是A的N/2倍。而第一个点就是直流分量，它的模值就是直流分量的N倍。而每个点的相位呢，就是在该频率下的信号的相位。第一个点表示直流分量（即0Hz），而最后一个点N的再下一个点（实际上这个点是不存在的，这里是假设的第N+1个点，也可以看做是将第一个点分做两半分，另一半移到最后）则表示采样频率Fs，这中间被N-1个点平均分成N等份，每个点的频率依次增加。例如某点n所表示的频率为：$Fn = (n-1) * Fs/N$。由上面的公式可以看出，Fn所能分辨到频率为为Fs/N，如果采样频率Fs为1024Hz，采样点数为1024点，则可以分辨到1Hz。1024Hz的采样率采样1024点，刚好是1秒，也就是说，采样1秒时间的信号并做FFT，则结果可以分析精确到1Hz，如果采样2秒时间的信号并做FFT，则结果可以分析精确到0.5Hz。如果要提高频率分辨率，则必须增加采样点数，也即采样时间。频率分辨率和采样时间是倒数关系。

下面图示可以更好的显示这种对应关系:



变换之后的频谱的宽度（Frequency Span）与原始信号也存在一定的对应关系。根据Nyquist采样定理，FFT之后的频谱宽度（Frequency Span）最大只能是原始信号采样率的1/2，如果原始信号采样率是4GS/s，那么FFT之后的频宽最多只能是2GHz。时域信号采样周期（Sample Period）的倒数，即采样率（Sample Rate）乘上一个固定的系数即是变换之后频谱的宽度，即 Frequency Span = $K*（1/\Delta T）$，其中ΔT为采样周期，K值取决于我们在进行FFT之前是否对原始信号进行降采样（抽点），因为这样可以降低FFT的运算量。

如下图的对应关系：



所以通过以上分析，我们就可以清楚的得出Matlab内置库中如此作图的原因，我们使用FFT算法就是在DFT简化了计算量的基础上完成了对正弦波以及余弦波的系数的求解。

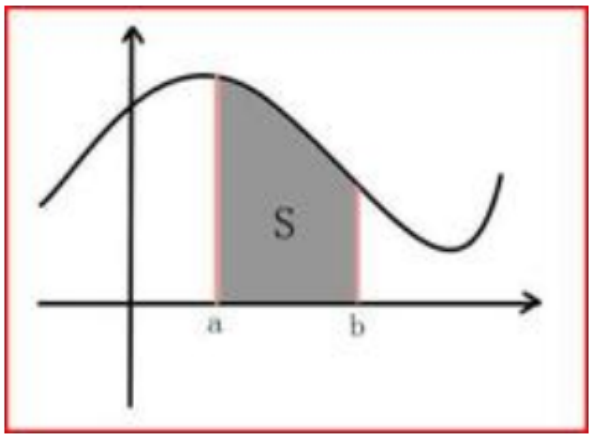
5. 复合梯形公式、复合辛普森公式求积分

5.1 问题描述

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。

5.2 算法分析

5.2.1 复合梯形公式



假设被积函数为f(x)，积分区间为$[a,b]$，把区间[a,b]等分成n个小区间，各个区间的长度为$step$，即$step=(b-a)/n$，称之为“步长”。根据定积分的定义及几何意义，定积分就是求函数$f(x)$在区间$[a,b]$中图线下包围的面积。将积分区间n等分，各子区间的面积近似等于梯形的面积，面积的计算运用梯形公式求解，再累加各区间的面积，所得的和近似等于被积函数的积分值n越大，所得结果越精确。以上就是利用复合梯形公式实现定积分的计算的算法思想。

计算公式如下：

$$T_n=\frac{step}{2}\left[f(a)+2\sum_{i=1}^{n-1}f(x_i)+f(b)\right]$$

复合梯形公式计算积分的误差:

$$R_n(f)=((b-a)/3)*(h/2)^2*f''(\xi),\xi\in[a,b]$$

5.2.2 复合辛普森公式

将区间$[a,b]$划分为 n 等分，在每个子区间$[x_k,x_{k+1}]$上使用辛普森公式，若记$x_{k+1/2}=x_k+1/2*h$，$h=(a-b)/n$,则可得复合辛普森求积分公式：

$$\int_a^b f(x)dx\approx\frac{h}{6}\int(f(a)+4\times\sum_{k=0}^{n-1}f(x_{\frac{k+1}{2}})+2\times\sum_{k=1}^{n-1}f(x_k)+f(b))+R_n(x)$$

复合辛普森公式求积分的误差：

$$R_n(f)=I-S_n=-\frac{h}{180}(\frac{h}{2})^4\sum_{k=0}^{n-1}f^{(4)}(\eta_k),\eta_k\in(x_k,x_{k+1}).$$

5.3 程序的运行、结果、分析

5.3.1 Matlab程序

5.3.1.1 复合梯形公式求积分

```
function output = myFun(n)
    x_1 = 0 ;
    x_2 = 1 ;
    h = ( x_2 - x_1 ) / n ;
    Sum = 0 ;
    for k = 1 : 1 : n - 1
        x_k = x_1 + h * k ;
        Sum = Sum + 2 * ( sin(x_k) / x_k ) ;
    end
    %由于sin(x) / x -> 无穷 所以使用洛必达法则求得改点极限为 1
    integration =  ( h / 2 )  * ( 1 + Sum + ( sin( x_2 ) / x_2 ) ) ;
    %下面求使用复合梯形公式的误差
    %误差公式为 Rn(f) = -(b-a)/12 * h ^ 2 * f(二阶导数)(ξk)
    %由课本P108知 sin(x) / x 的k阶导数可近似为 1/(k+1)
    %所以误差可以理解为 Rn(g) = -(b-a)/12 * h ^ 2 * ( 1 / (1+k) )
    error = - ( x_2 - x_1 ) / 12 * h ^ 2 * ( 1 / (1 + 2) ) ;
    %显示积分结果
    integration
    %显示结果
    error = abs(error)
end
```

5.3.1.2 复合辛普森公式求积分

```
function output = myFun(n)
    x_1 = 0 ;
    x_2 = 1 ;
    h = ( x_2 - x_1 ) / n ;
    Sum_1 = 0 ;
    Sum_2 = 0 ;
    x_k = x_1 + h / 2 ;
    %计算第一个求和
    for k = 0 : 1 : n - 1
        Sum_1 = Sum_1 + 4 * ( sin(x_k) / x_k ) ;
        x_k = x_k + h ;
    end
    x_k2 = x_1 ;
    %计算第二个求和
    for k = 1 : 1 : n - 1
        x_k2 = x_1 + h * k ;
        Sum_2 = Sum_2 + 2 * ( sin(x_k2) / x_k2 ) ;
    end
    %由于sin(x) / x -> 无穷 所以使用洛必达法则求得改点极限为 1
    integration =  ( h / 6 )  * ( 1 + Sum_1 + Sum_2  + ( sin( x_2 ) / x_2 ) ) ;
    %下面求使用复合辛普森公式的误差
    %误差公式为 Rn(f) = -(b-a)/180 * ( h / 2 ) ^ 4 * f(四阶导数)(ξk)
    %由课本P108知 sin(x) / x 的k阶导数可近似为 1/(k+1)
    %所以误差可以理解为 Rn(g) = -(b-a)/180 * ( h / 2 ) ^ 4 * ( 1 / (1+k) )
    error = - ( x_2 - x_1 ) / 180 * ( h / 2 ) ^ 4 * ( 1 / (1 + 4) ) ;
    %显示积分结果
```

```
    integration
    %显示结果
    error = abs(error)
end
```

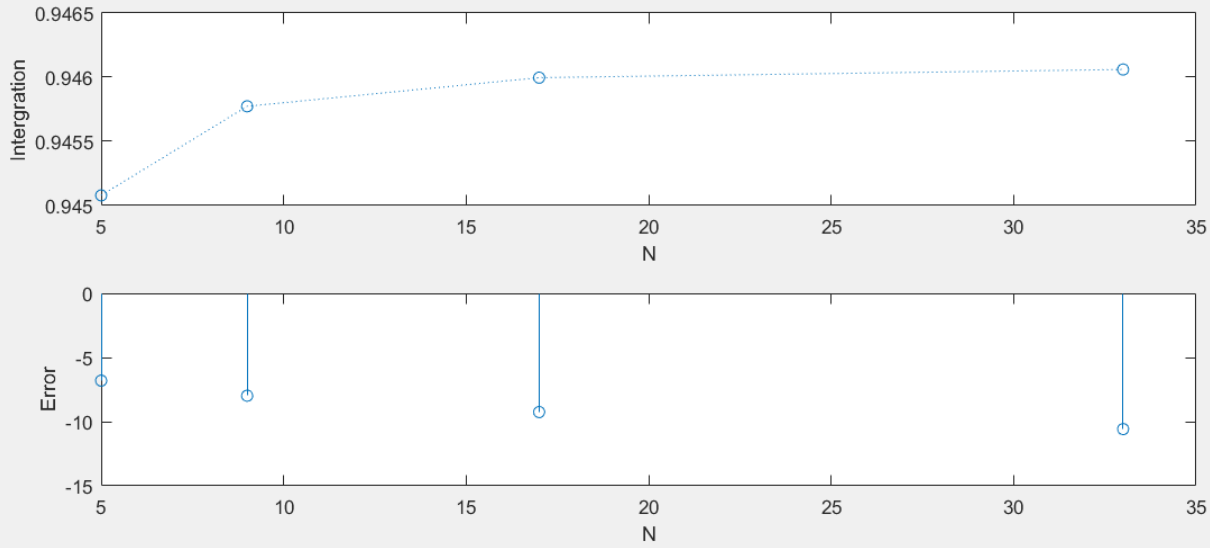
5.3.2 运行结果

5.3.2.1 复合梯形公式求积分

5.3.2.1.1 积分值与误差值的数值结果

```
intergration =

   0.945078780953402   0.945773188549752   0.945996225242376   0.946060023888043


error =

   0.001111111111111   0.000342935528121   0.000096116878124   0.000025507601265
```

5.3.2.1.2 积分值与误差值的图示对比（误差取自然对数）

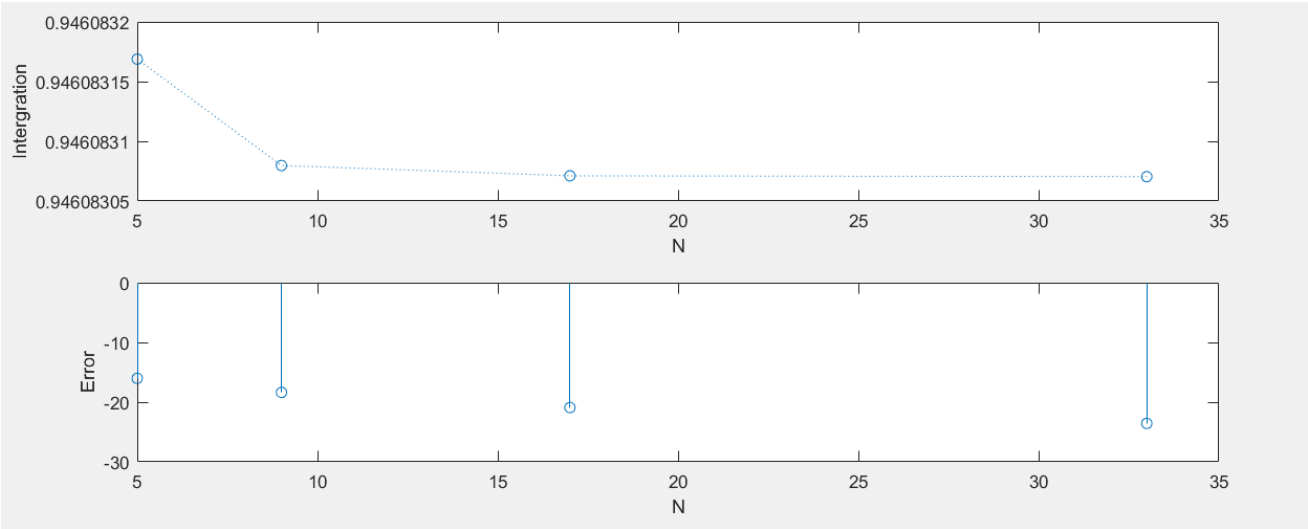


5.3.2.2 复合辛普森公式求积分

5.3.2.2.1 积分值与误差值的数值结果

```
intergration =

   0.946083168838073   0.946083079742053   0.946083071103489   0.946083070419036


error =

   1.0e-06 *

   0.111111111111111   0.010584429880269   0.000831460883424   0.000058557395007
```

5.3.2.2.2 积分值与误差值的图示对比（误差取自然对数）



> 注：
>
> 对比图示的第一行: N = 5,9,17,33 的积分值图
>
> 对比图示的第二行: N = 5,9,17,33 的误差值图（对误差取对数）

5.3.3 分析

1. 从复合梯形公式求值结果的变动情况来看，我们求得的积分值是随着划分区间N的个数的增加而增加的，我觉得这应该是与我们所要求的函数是一个凹函数有关。

   在题目的凹函数的限制下，我们在使用梯形公式求积分时候，很容易把$f(x_k)$与$f(x_{k+1})$之间连线的上半部分给遗漏掉，这也就导致我们所求的积分值会比真实的积分值要小一些；当我们扩大可划分区间的个数N的时候由于$f(x_k)$与$f(x_{k+1})$与待求函数之间的距离越来越小，我们的计算值也就会越发接近真实值。

   同时也可以从收敛的误差曲线上看到，随着划分区间N的数目成2倍的数量级上的扩大，误差也会进行10^{-1}次方级别上的缩小。

2. 从复合辛普森公式求值结果的变动情况来看，我们求得的积分值是随着划分区间N的个数的增加而减小进而逼近真实的积分值。

复合辛普森公式的求值准确度是要明显快于复合梯形公式的，从初次计算的积分值上我们就可以发现，即使是当N = 3 的时候，复合辛普森公式的求值已经基本接近于真实值，在往后的迭代中，即使是同样的收敛速度，复合辛普森的求值结果已经是精度要高于复合梯形公式了，这样我们就可以大大减少计算的过程以及计算量。

从收敛速度上看，复合辛普森公式求积分的收敛速度也是要快于复合梯形公式的，比如 N 从7变动到15 ， 收敛精度提升了10^2级别。

3.

整体上来说，复合梯形公式求积分的形式较为简单，复合辛普森公式求积分略显复杂但是计算精度与计算收敛速度是要快一点的，我们可以根据实际情况择优选择。

6. 微分方程初值类问题

6.1 需求分析

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。

6.2 算法分析

6.2.1 前向欧拉方法

若已知初值y_0,使用 x_n,y_n,$f(x_n,y_n)$以及步长h推导 y_{n+1},推导公式如下：

$$y_{n+1}=y_n+f(x_n,y_n)$$

6.2.2 后向欧拉方法

若已知初值y_0,使用y_n,$f(x_{n+1},y_{n+1})$以及步长h的值来求y_{n+1},推导公式如下:

$$y_{n+1}=y_n+f(x_{n+1},y_{n+1})$$

6.2.3 梯形方法

若已知初值y_0,使用y_n,$f(x_n,y_n)$,$f(x_{n+1},y_{n+1})$以及步长$\frac{h}{2}$的值求y_{n+1},推导公式如下：

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

6.2.4 改进欧拉方法

在改进的欧拉方法，我们先用欧拉公式求得一个初步的近似值 $\overline{y}_{n+1}$,称之为预测值，预测值 $\overline{y}_{n+1}$的精度可能很差，再用梯形公式将其矫正一次得y_{n+1}，这个结果成为校正值。

根据这个思想已知y_0,步长h可以把改进欧拉公式表示为平均化形式

$$y_p = y_n + h * f(x_n, y_n)$$

$$y_c = y_n + h * f(x_{n+1}, y_p)$$

$$y_{n+1} = \frac{1}{2}(y_p + y_c)$$

6.3 程序的运行、结果、分析

6.3.1 Matlab 代码

6.3.1.1 前项欧拉方法

```
function output = myFun(input)
    h = 0.1 ;
    n = 10 ;
    y_0 = 1 ;
    y_n_1 = y_0 + h * ( y_0 ) ;
    y_n = y_n_1 ;
    y_true = zeros(1,10) ;
    y_compute = zeros(1,10) ;
    y_compute(1,1) = y_n_1 ;
    y_true(1,1) = sqrt(1+2*0.1) ;
    for k = 1 : 1 : n - 1
        y_n_1 = y_n +  h * (y_n - 2 * 0.1 * k / y_n ) ;
        y_n = y_n_1 ;
        y_compute(1,k + 1) = y_n_1 ;
        y_true(1,k + 1) = sqrt(1+2*0.1*(k+1)) ;
    end
    plot(1:1:10,y_true,'o:b') ;
    hold on ;
    plot(1:1:10,y_compute,'o:r') ;
    legend('True','Compute');
    xlabel('N');
    ylabel('Value');
    title('真实值与计算值的对比');
end
```

6.3.1.2 后项欧拉方法

```
function output = myFun(input)

    h = 0.1 ;
    n = 10 ;
    y_0 = 1 ;
    y_n_1 = ( y_0 + sqrt(y_0 * y_0 - 4 * (1 - h) * 2 * h * 0.1) ) / (2 * (1 - h)) ;
    y_n = y_n_1 ;
    y_true = zeros(1,10) ;
    y_compute = zeros(1,10) ;
```

```
    y_compute(1,1) = y_n_1 ;
    y_true(1,1) = sqrt(1+2*0.1) ;
    for k = 1 : 1 : n - 1
        y_n_1 = ( y_n + sqrt(y_n * y_n - 4 * (1 - h) * 2 * h * (0.1*(k+1))) ) / (2 * (1 - h));
        y_n = y_n_1 ;
        y_compute(1,k + 1) = y_n_1 ;
        y_true(1,k + 1) = sqrt(1+2*0.1*(k+1)) ;
    end
    y_compute
    y_true
    plot(1:1:10,y_true,'o:b') ;
    hold on ;
    plot(1:1:10,y_compute,'o:r') ;
    legend('True','Compute');
    xlabel('N');
    ylabel('Value');
    title('真实值与计算值的对比');
end
```

6.3.1.3 梯形方法

```
function output = myFun(input)

    h = 0.1 ;
    n = 10 ;
    y_0 = 1 ;
    y_n_1 =  ( ( ( h / 2 * y_0 + y_0 ) + sqrt ( ( h / 2 * y_0 + y_0 ) ^ 2 - 4 * (1 - h / 2) * h
* (0.1) ) ) / (2 - h));
    y_n = y_n_1 ;
    y_true = zeros(1,10) ;
    y_compute = zeros(1,10) ;
    y_compute(1,1) = y_n_1 ;
    y_true(1,1) = sqrt(1+2*0.1) ;
    for k = 1 : 1 : n - 1
        %这里使用一元二次方程求解y_n+1
        y_n_1 =  ( ( ( h / 2 * y_n + y_n - h * (0.1 * k ) / y_n) + sqrt ( ( h / 2 * y_n + y_n -
h * (0.1 * k) / y_n) ^ 2 - 4 * (1 - h / 2) * h * (0.1 * (k+1)) ) ) / (2 - h));
        y_n = y_n_1 ;
        y_compute(1,k + 1) = y_n_1 ;
        y_true(1,k + 1) = sqrt(1+2*0.1*(k+1)) ;
    end
    %对比计算值与真实值
    y_compute
    y_true
    plot(1:1:10,y_true,'o:b') ;
    hold on ;
    plot(1:1:10,y_compute,'o:r') ;
    legend('True','Compute');
    xlabel('N');
    ylabel('Value');
    title('真实值与计算值的对比');
end
```

6.3.1.4 改进欧拉方法

```
function output = myFun(input)

    h = 0.1 ;
    n = 10 ;
    y_0 = 1 ;
    y_true = zeros(1,10) ;
    y_compute = zeros(1,10) ;
    y_true(1,1) = sqrt(1+2*0.1);

    y_p = y_0 + h * (y_0) ;
    y_c = y_0 + h * (y_p - 2 * 0.1 / y_p) ;
    y_p_1 = (y_p + y_c) / 2 ;
    y_n = y_p_1 ;
    y_compute(1,1) = y_n ;
    for k = 1 : 1 : n - 1
        y_p = y_n + h * (y_n - 2 * 0.1 * k / y_n ) ;
        y_c = y_n + h * (y_p - 2 * 0.1 * ( k + 1 ) / y_p) ;
        y_p_1 = (y_p + y_c) / 2 ;
        y_n = y_p_1 ;

        y_compute(1,k + 1) = y_p_1 ;
        y_true(1,k + 1) = sqrt(1+2*0.1*(k+1)) ;
    end
    y_compute
    y_true
    plot(1:1:10,y_true,'o:b') ;
    hold on ;
    plot(1:1:10,y_compute,'o:r') ;
    legend('True','Compute');
    xlabel('N');
    ylabel('Value');
    title('真实值与计算值的对比');
end
```

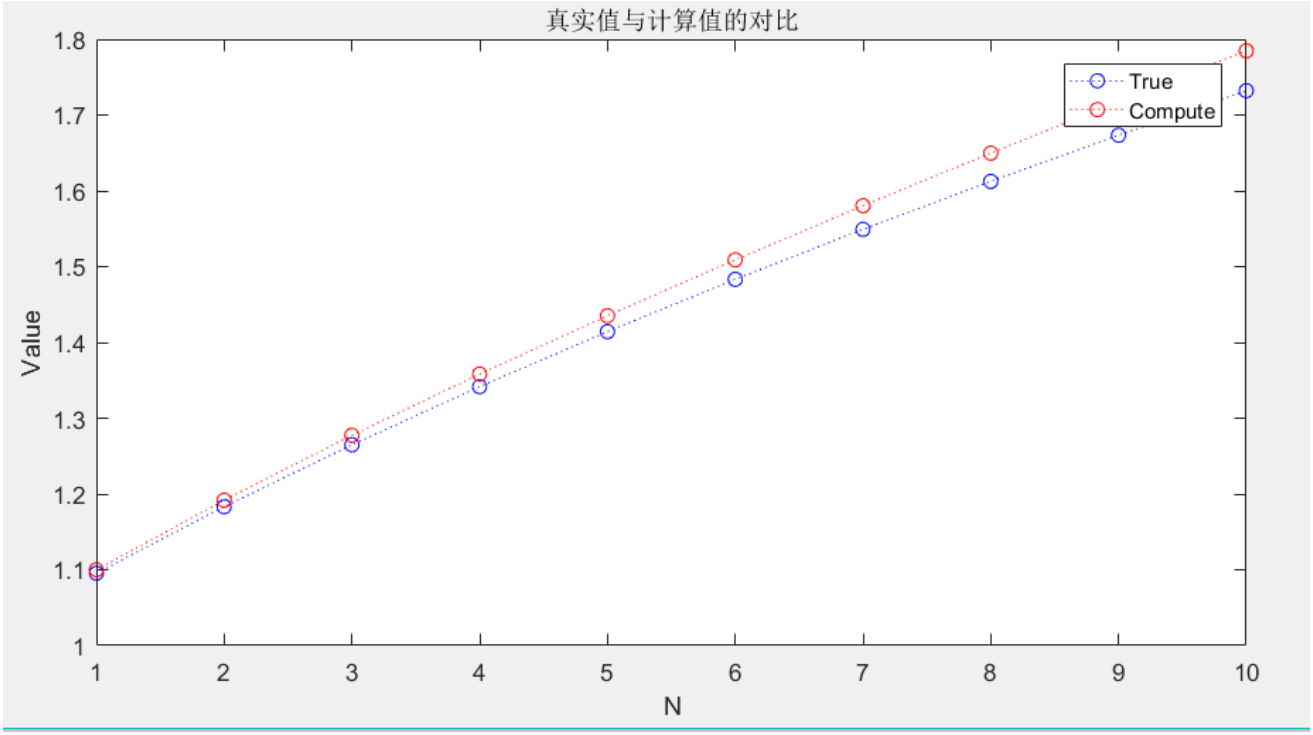
6.3.2 运行结果

6.3.2.1 前项欧拉方法

6.3.2.1.1 迭代值与真实值数值对比

```
y_true =

   1.095445115010332   1.183215956619923   1.264911064067352   1.341640786499874   1.414213562373095   1.483239697419133   1.549193338482967   1.612451549659710   1.673320053068151   1.732050807568877

y_compute =

   1.100000000000000   1.191818181818182   1.277437833714722   1.358212599560289   1.435132918657796   1.508966253566332   1.580338237655217   1.649783431047711   1.717779347860087   1.784770832497982
```

6.3.2.1.2 迭代值与真实值图例对比



注：

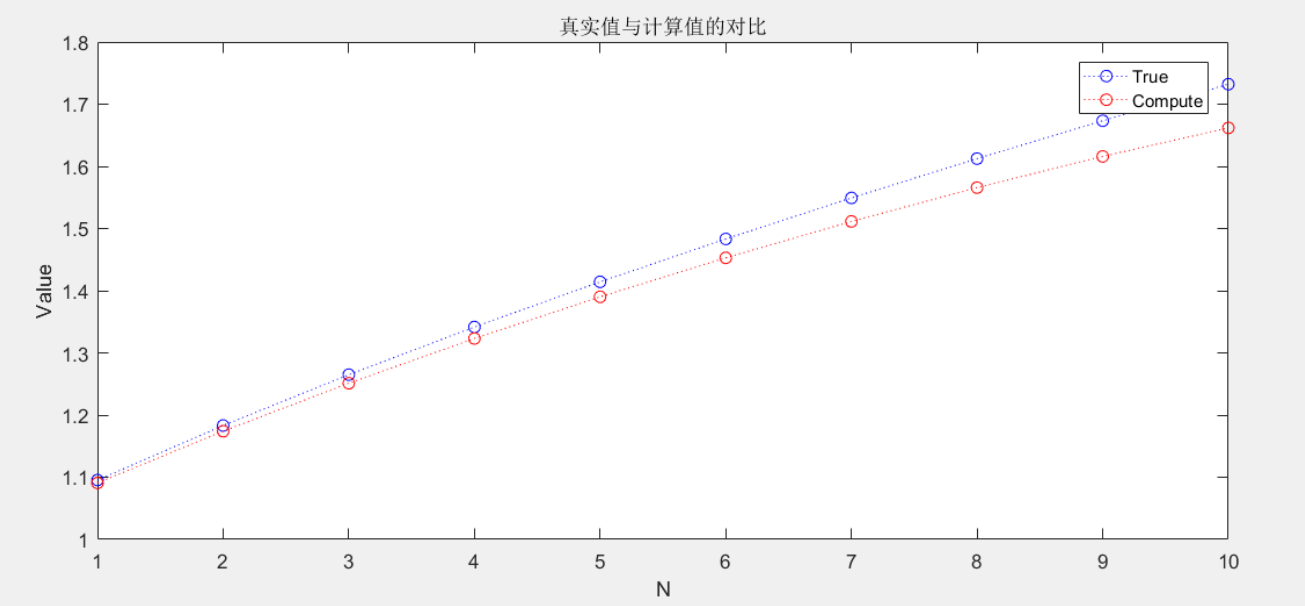
1. 横坐标代表迭代步数
2. 纵坐标代表当前步数下求得的值
3. 红色的线代表使用书上$y=\sqrt{(1+2x)}$的值，也即准确值$y(x_n)$
4. 蓝色的线代表前项欧拉方法求得的值

6.3.2.2 后项欧拉方法

6.3.2.2.1 迭代值与真实值数值对比

```
y_compute =

   1.090737536835213   1.174075761293480   1.251248506796506   1.323093497752086   1.390178074627193   1.452869923324636   1.511376837164648   1.565767235451984   1.615977254482524   1.661807042621093

y_true =

   1.095445115010332   1.183215956619923   1.264911064067352   1.341640786499874   1.414213562373095   1.483239697419133   1.549193338482967   1.612451549659710   1.673320053068151   1.732050807568877
```

6.3.2.2.2 迭代值与真实值图例对比



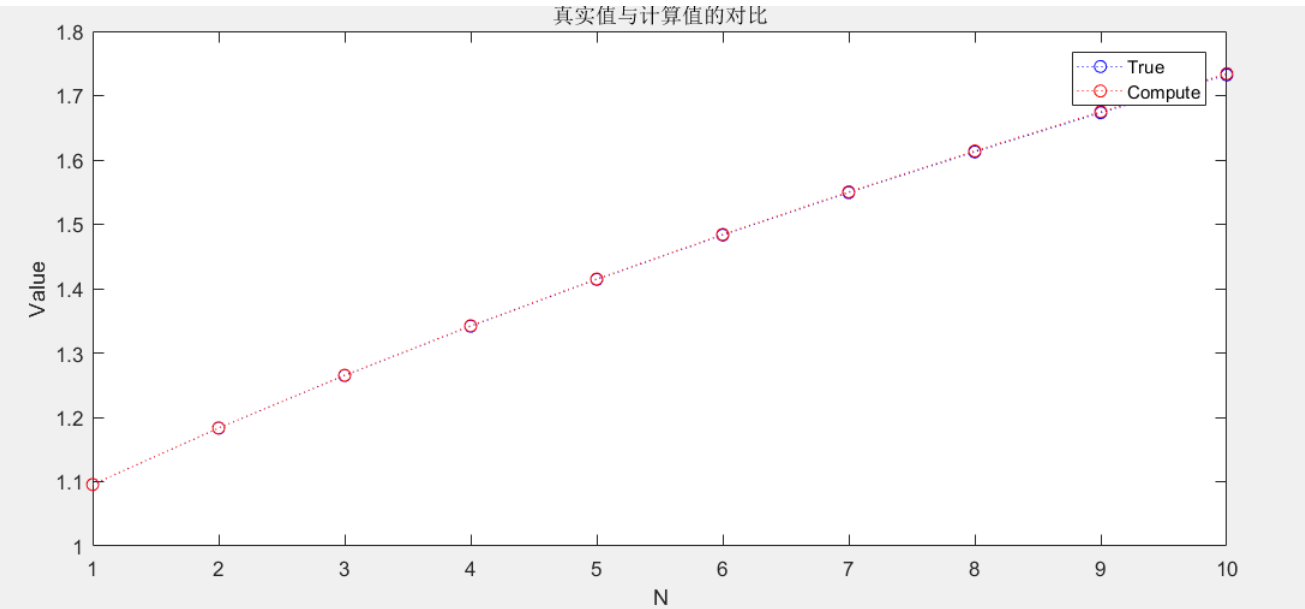
> 注：
>
> 1. 横坐标代表迭代步数
> 2. 纵坐标代表当前步数下求得的值
> 3. 红色的线代表使用书上$y=\sqrt{(1+2x)}$的值，也即准确值$y(x_n)$
> 4. 蓝色的线代表前项欧拉方法求得的值

6.3.2.3 梯形方法

6.3.2.3.1 迭代值与真实值数值对比

```
y_compute =

   1.095655838313732   1.183593669162530   1.265440529010879   1.342322417136668   1.415058105112867   1.484266055534500   1.550427908099594   1.613928403848637   1.675081692031514   1.734149362127398

y_true =

   1.095445115010332   1.183215956619923   1.264911064067352   1.341640786499874   1.414213562373095   1.483239697419133   1.549193338482967   1.612451549659710   1.673320053068151   1.732050807568877
```

6.3.2.3.2 迭代值与真实值图例对比



> 注：

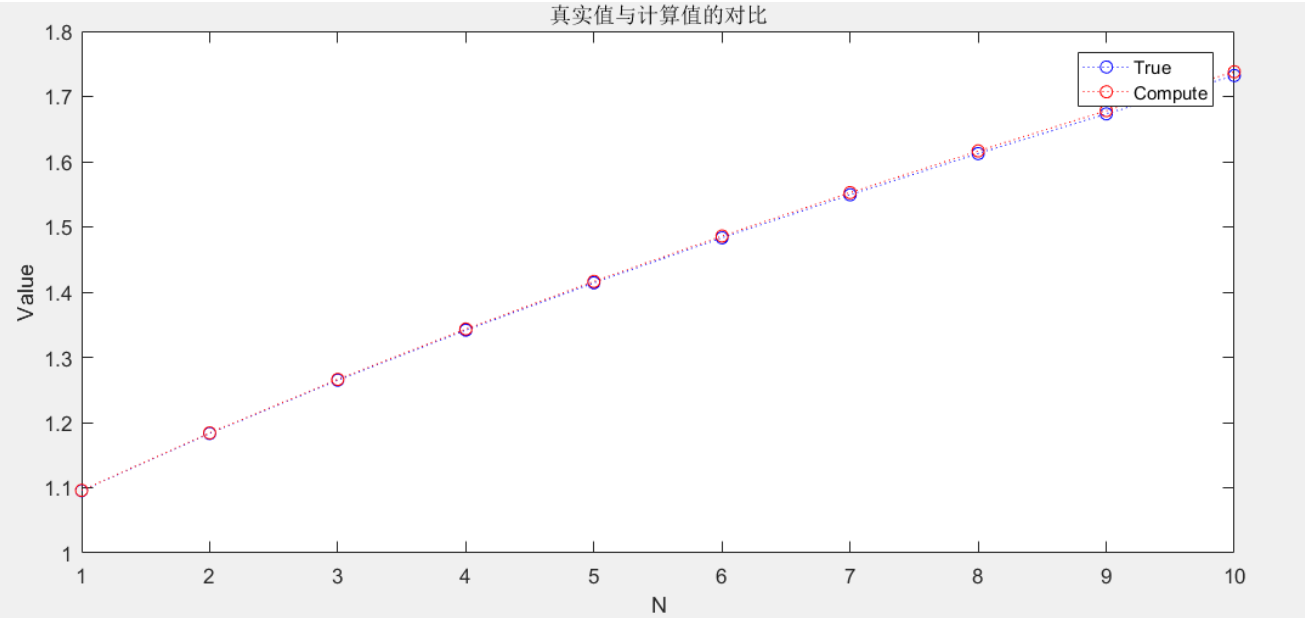
> 1. 横坐标代表迭代步数
> 2. 纵坐标代表当前步数下求得的值
> 3. 红色的线代表使用书上$y=\sqrt{(1+2x)}$的值，也即准确值$y(x_n)$
> 4. 蓝色的线代表前项欧拉方法求得的值

6.3.2.4 改进欧拉方法

6.3.2.4.1 迭代值与真实值数值对比

```
y_compute =

   1.095909090909091   1.184096569242997   1.266201360875776   1.343360151483999   1.416401928536909   1.485955602415669   1.552514091326146   1.616474782752058   1.678166363675186   1.737867401035414

y_true =

   1.095445115010332   1.183215956619923   1.264911064067352   1.341640786499874   1.414213562373095   1.483239697419133   1.549193338482967   1.612451549659710   1.673320053068151   1.732050807568877
```

6.3.2.4.2 迭代值与真实值图例对比



> 注：
>
> 1. 横坐标代表迭代步数
> 2. 纵坐标代表当前步数下求得的值
> 3. 红色的线代表使用书上$y=\sqrt{(1+2x)}$的值，也即准确值$y(x_n)$
> 4. 蓝色的线代表前项欧拉方法求得的值

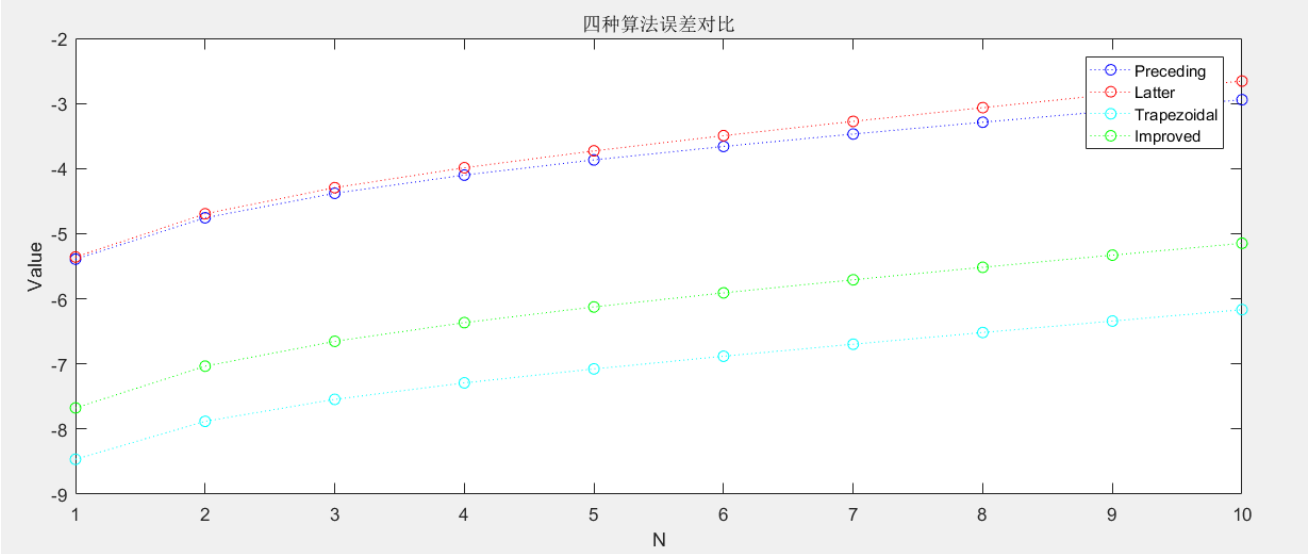
6.3.3 分析

从`6.3.2`真实值与迭代值的对比图例上我们可以看到，前项后项欧拉方法的求值结果距离真实值较远,同时二者的收敛速度也是较为缓慢，并且随着迭代步数的上升，这两个方法的求值结果会离准确值得差距越来越大，这就证明了这两个算法并不是十分稳定的算法，所以我们在进行微分方程初值类问题计算时候，尽量要避免使用这种误差大且不稳定的算法。

但是从计算量上我们可以发现，这两个算法是计算量最少的，同时也是形式最为简单的，这就方便了我们理解与操作，但是真正应用与数值计算，我们应选用求值精度更高的。

再来分析梯形方法与改进欧拉方法，从对比图例上我们看到，这两个方法的迭代求值结果已经近似等于准确值$y(x_n)$,并且在计算的过程中，计算的迭代值与真实值的误差也保持在一个相对稳定的收敛范围内，所以这两个算法应该是适合我们使用进行计算的算法；但是使用梯形公式繁琐一点，就本题而言，我们并不知道y_{n+1}的值，并且这个值同时出现在迭代求值公式的两边，这就需要我们使用二次方程反解出y_{n+1},这也就增加了我们的计算量，反观改进欧拉方法，直接套公式求解即可，已知变量全在求值公式右边，未知变量全在等号左边，这也就方便了我们的计算。

为了直观的显示这四种算法计算值与真是值的差距，我进一步进行了作图：



所以从此图上我们可以明显看出，梯形方法的初始误差是最小的，后项后拉方法的初始误差是最大的，但是就误差的变化趋势来说，这四个方法误差的变动速度基本是一致的，这是因为我们在计算误差时候，是使用$|y_{true} - y_{compute}|$计算的；除了使用减法操作，我们对误差值的统计还可以使用公式$\frac{h^2}{2}y''(\xi)$,由于我们的目标函数没有发生变化，所以我们误差值变动速度是基本一致的，但是由于选择的第一个初始点y_k的不同，这就导致我们的初值有大有小。

所以不管是从计算精度，还是从计算复杂度上来说，我们在计算微分方程初值类问题时候，应当优先选择梯形方法与改进欧拉方法。