

《数值计算实验》

实验报告

题目	插值与拟合、不同方法求解微积分以及快速FFT法
姓名	刘硕
学号	16340154
班级	软件工程二班

实验一

一. 实验环境和工具

在 Win10 环境下利用Matlab-R2014b 完成实验。

二. 问题描述

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

三. 实验内容

插值法又称“内插法”，是利用函数 $f(x)$ 在某区间中已知的若干点的函数值，作出适当的特定函数，在区间的其他点上用这特定函数的值作为函数 $f(x)$ 的近似值，这种方法称为插值法。如果这特定函数是多项式，就称它为插值多项式。

插值方法主要分为以下几种：Lagrange 插值将待求的n次多项式插值函数 $P_n(x)$ 改写成另一种表示方式，再利用插值条件确定其中的待定函数，从而求出插值多项式；Newton 插值提出另一种构造插值多项式的方法，将待求的 n 次插值多项式 $P_n(x)$ 改写为具有承袭性的形式，然后利用插值条件确定 $P_n(x)$ 的待定系数，以求出所要的插值函数；Hermite 插值利用 Lagrange 插值函数的构造方法，先设定函数形式，再利用插值条件求出插值函数；分段插值将被插值函数 $f(x)$ 的插值节点由小到大排序，然后每对相邻的两个节点为端点的区间上用 m 次多项式去近似 $f(x)$ 。

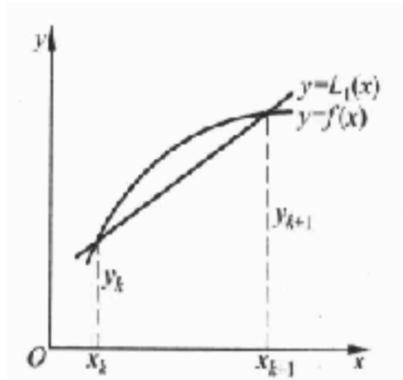
Matlab 提供了 **interp1(x, v, vp)** 函数来处理一维数据的插值，另外拟合函数 **polyfit** 也可以用于这类问题的求解。

1. 线性插值法

线性插值是利用连接两个已知量的直线来确定一个未知量的方法。

1.1. 算法设计

已知二维直角坐标系中的两点 $A(x_0, y_0)$ 与 $B(x_1, y_1)$ ，要得到 $[x_0, x_1]$ 区间内某一位置 x 在直线上的值。根据图中所示，我们得到直线方程：



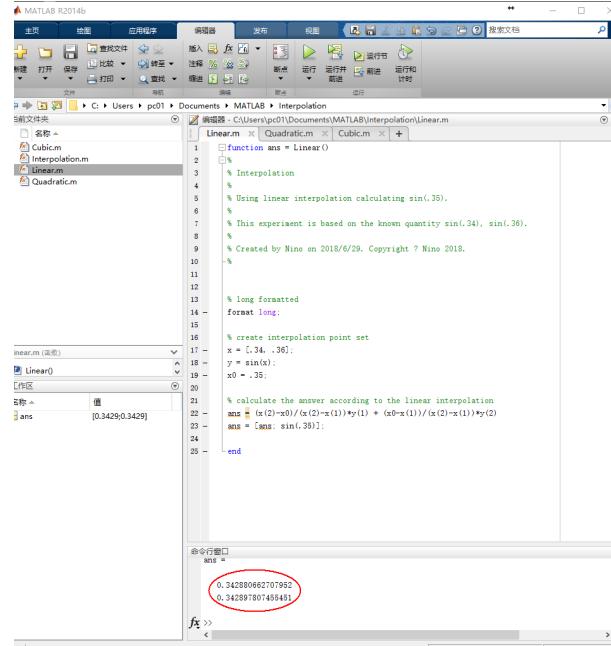
$$L_1(x) = y_k + \frac{y_{k+1} - y_k}{x_{k+1} - x_k}(x - x_k)$$

1.2. 数值实验

这里，为了使用线性插值法计算 $\sin(0.35)$ 的值，设定 $y(k)$ 和 $y(k+1)$ 分别是 $\sin(0.34)$ 和 $\sin(0.36)$ 。

1.3. 结果分析

计算的结果如图所示，看出计算结果比真实结果约差了 1.7×10^{-5} 。



2. 抛物线插值法

抛物线插值法是通过利用连接三个已知量的抛物线来确定一个未知量的方法。

2.1. 算法设计

已知二维直角坐标系中的点 A(x_0, y_0)，B(x_1, y_1)，C(x_2, y_2) 要得到 $[x_0, x_2]$ 区间内某一位置 x 在直线上的值。得到直线方程：

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

2.2. 数值实验

这里，为了使用线性插值法计算 $\sin(0.35)$ 的值，设定 $y_{(k-1)}$ ， $y_{(k)}$ 和 $y_{(k+1)}$ 分别是 $\sin(0.32)$ ， $\sin(0.34)$ ， $\sin(0.36)$ 。

2.3. 结果分析

计算的结果如图所示。看出计算结果比真实结果约差了 4.7×10^{-6} ，比线性插值更准确。

```

% Function ans = Quadratic()
%
% Interpolation
%
% Using quadratic interpolation calculating sin(.35).
%
% This experiment is based on the known quantity sin(.32), sin(.34), sin(.36).
%
% Created by Nino on 2018/6/19. Copyright ? Nino 2018.
%
% long formatted
format long;
%
% create interpolation point set
x = [.32, .34, .36];
y = sin(x);
x0 = .35;
%
% calculate the answer according to the quadratic interpolation
ans = (x0-x(2)) * (x0-x(3)) / (x(1)-x(2)) / (x(1)-x(3)) * y(1);
ans = ans + (x0-x(1)) * (x0-x(3)) / (x(2)-x(1)) / (x(2)-x(3)) * y(2);
ans = ans + (x0-x(1)) * (x0-x(2)) / (x(3)-x(1)) / (x(3)-x(2)) * y(3);
ans = [ans sin(.35)];
end

```

命令行窗口显示结果：

```

ans =
0.34297338608783

```

3. 三次拉格朗日插值法

抛物线插值法是通过利用连接四个已知量的三次线来确定一个未知量的方法。

3.1. 算法设计

已知二维直角坐标系中的点 $A(x_0, y_0)$, $B(x_1, y_1)$, $C(x_2, y_2)$, $D(x_3, y_3)$ 要得到 $[x_0, x_3]$ 区间内某一位置 x 在直线上的值。类比上述过程，构造拉格朗日插值多项式。得到直线方程：

$$l_k(x) = \frac{(x - x_0)(x - x_1)\cdots(x - x_{k-1})(x - x_{k+1})\cdots(x - x_n)}{(x_k - x_0)\cdots(x_k - x_{k-1})(x_k - x_{k+1})\cdots(x_k - x_n)}, \quad k = 0, 1, \dots, n.$$

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

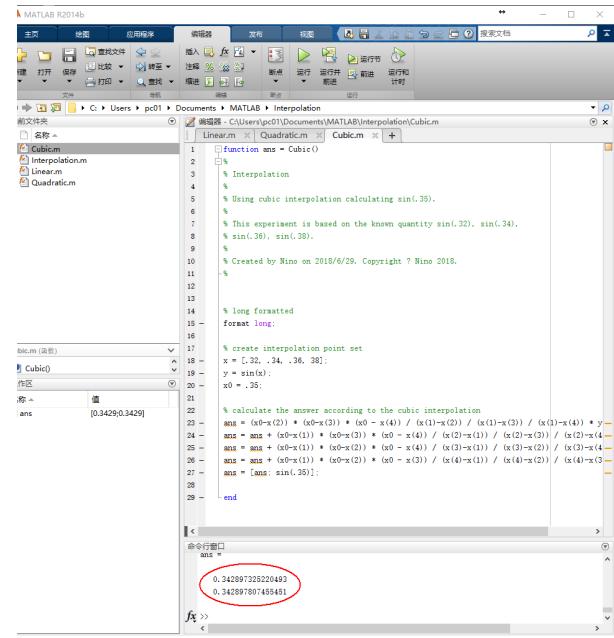
3.2. 数值实验

这里，为了使用线性插值法计算 $\sin(0.35)$ 的值，设定 $y_{(k-1)}$, $y_{(k)}$, $y_{(k+1)}$ 和 $y_{(k+2)}$ 分别是 $\sin(0.32)$, $\sin(0.34)$ 和 $\sin(0.36)$ 和 $\sin(0.38)$ 。

3.3. 结果分析

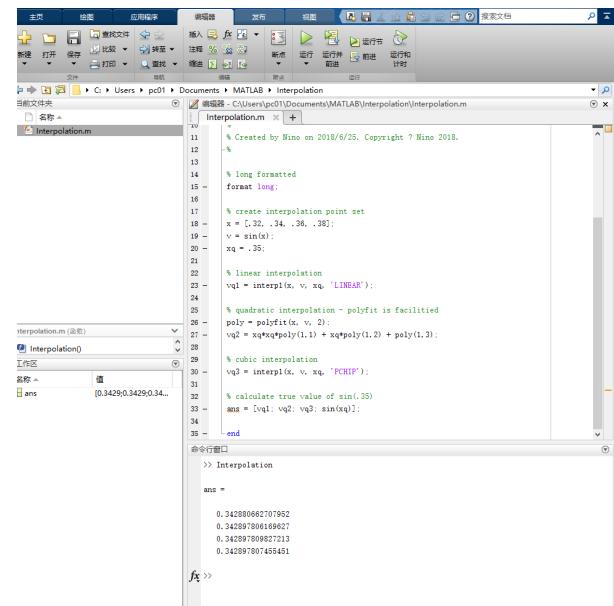
计算的结果如图所示。看出计算结果比真实结果约差了 4.8×10^{-6} ，结果竟然不如二次插值法准

确，可见不一定次数越多，插值法的结果就越准确。



四. 实验心得

在这次实验中，我主要先是使用了Matlab自带的函数进行计算，得到了较好的结果。如图是通过Matlab自带函数得到的结果。



可是后来发现，做这个实验的初衷是自行实现以下插值算法。并且 Matlab 函数的插值运算有如下几个问题：

首先，`interp1` 函数的 `method` 参数并没有给出二次插值法（抛物线插值法）。只能通过 Matlab 的 `polyfit` 函数二次拟合多项式。然后通过带入数值 0.35 得到目标。但是拟合与插值在定义方面是不同的，所以这样做的精确性有待考量。

另外，`interp1` 函数的 `method` 不同的参数，对于三次插值法有不同的插值方法，比如三次样条差值和三次拉格朗日插值。

拉格朗日插值法较为简单，重点是推出其插值公式；另外对插值和拟合的理解也是未来深度学习的基础。

实验二

一. 实验环境和工具

在 Windows 10 环境下利用 Matlab-R2014b 完成实验。

二. 问题描述

分别用二分法(选取求根区间为[10, 11])、牛顿法、简化牛顿法、弦截法，计算 115 的平方根，精确到小数点后六位。绘出迭代步数的收敛精度曲线和各种方法的运行时间。

三. 实验内容

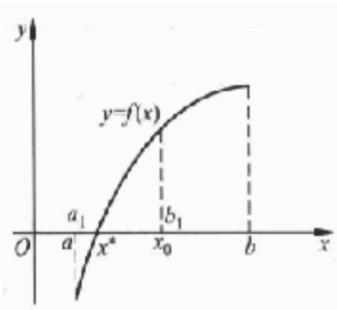
求一个数的平方根可以化成非线性方程根的求解问题。求 a 的平方根 x ，可以把问题转化成求方程 $f(x) = x^2 - a$ 的解。求解非线性方程有很多种方法，比如二分法、牛顿法、弦截法等等。这里以二分法（选取求根区间为[10, 11]）、牛顿法、简化牛顿法、弦截法为例，求解115的平方根。

1. 二分法

二分法是解决非线性方程的最基本方法。核心思想是对有根区间进行不断分割。

1.1. 算法设计

考察有根区间 $[a, b]$, 取中点 x_0 把区间分成两半, 假如 x_0 不是函数的零点, 那么检查 $f(x_0)$ 的符号, $f(x_0)$ 跟 $f(a)$ 、 $f(b)$ 谁的符号不同, 就靠近那一侧。

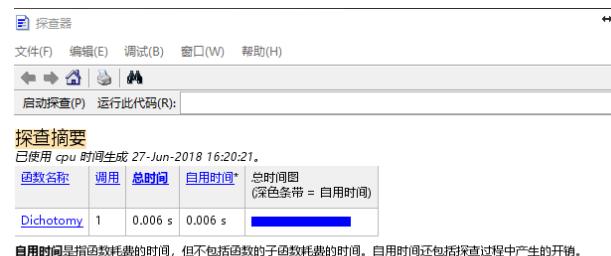
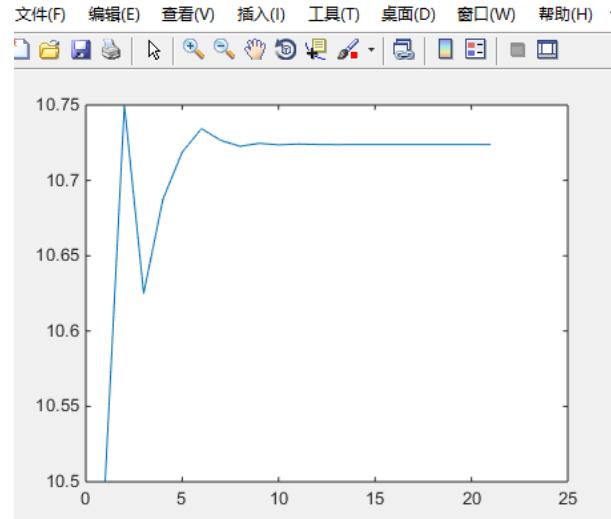


1.2. 数值实验

为了求 $\sqrt{115}$ 的平方根, 给定求根区间 10 到 11 , 不断对区间进行二分, 直到足够靠近零点。可以通过迭代或者递归来实现二分法求根。这里采用的是迭代法, 迭代初始条件是给定的求根区间, 之后根据二分区间中值的函数值, 确定新的求根区间, 求根下限或者求根上限随之改变。迭代的终止条件是如果某一点的函数值小于一个小量 eps , 那么迭代停止。

1.3. 结果分析

```
% Dichotomy.m
% initialize the iteration
% iter + 1;
ans = [];
% dichotomy method
while iter < MAX
    mid = (left + right)/2;
    iter = iter+1;
    ans = [ans, mid];
    val = mid*mid - a;
    % if near enough
    if abs(val) < eps
        ans = [ans, mid];
        return;
    end
    % split
    if val > 0
        right = mid;
    elseif val < 0;
        left = mid;
    end
end
% 10 to 20
10.7237548828125000 10.7238159179687500 10.723785400390625 10.7238006859179488
17 to 20
10.72380828874019 10.723804473876993 10.723806301205886 10.7238054275951270
21 to
10.7238054275951270
f1>
```



如图所示是二分法求未知数的平方根的结果，浅蓝色的 plot 图是迭代的收敛曲线，运行时间0.006秒。图中红色框内是每一次迭代的求根结果，可以发现一共用了21次迭代出了结果10.723805。

2. Newton 迭代法

Newton 法是一种线性化的思想，其基本思想是把非线性方程逐步归结为某种线性方程来求解。

2.1. 算法设计

设已知方程 $f(x)=0$ 有近似根 x_k (假定 $f'(x_k)\neq 0$)，将函数 $f(x)$ 在点 x_k 展开，有

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

于是方程 $f(x)=0$ 可近似地表示为

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

这是个线性方程，记其根为 x_{k+1} ，则 x_{k+1} 的计算公式为

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad k = 0, 1, \dots$$

2.2. 数值实验

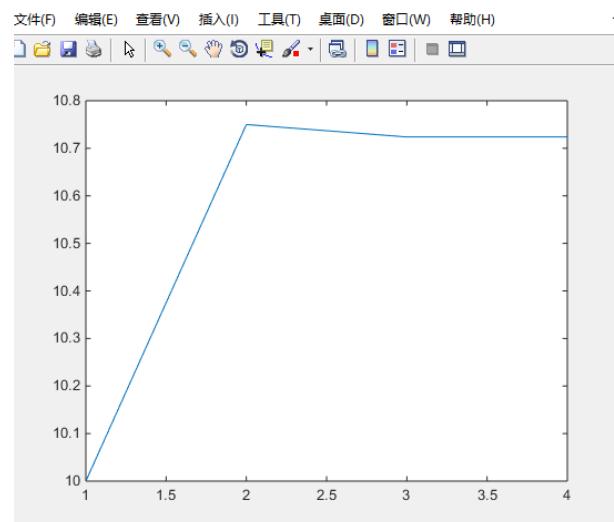
为了求 $\sqrt{115}$ 的平方根，不断对非线性函数进行线性化。可以通过迭代来实现牛顿法求根，迭代初始条件我选定的是 $x = 10$ ，之后不断根据 x 点出切线与 x 轴的交点迭代，确定新的结果。迭代的终止条件是如果某一点的函数值小于一个小量 eps ，那么迭代停止。

2.3. 结果分析

The screenshot shows the MATLAB interface. The current file is `Newton.m`, which contains the following code:

```
function [ans, iter] = Newton(a, x, eps)
% Square Root Calculation
% Using Newton method to calculate the square root of a particular number.
%
% Param a is the number to be rooted.
% Param x is the initial number of the root.
% Param eps is the error tolerance of the Newton method.
%
% Return ans is the root of target number during iterations.
% Return iter is the iteration number.
%
% Created by Nino on 2018/6/25. Copyright ? Nino 2018.
%
% max iterations
MAX = 100;
%
% long formatted
format long;
%
% argument adjustment
if nargin == 0
    a = 1;
end
for i = 1:MAX
    f = a - x^2;
    df = -2*x;
    x = x - f/df;
    if abs(f) < eps
        break;
    end
end
ans = x;
iter = i;
```

In the workspace, there is a variable `ans` with the value `10.750010.723...`. The command window shows the result of running the function: `>> Newton(115, 10)` followed by the output `ans = 10.00000000000000 10.75000000000000 10.723837209302324 10.723808294811097`.





如图是利用牛顿法求解未知数的平方根的结果，其收敛曲线，还有运行时间。可以发现，一共迭代了4次就找到根10.723805，比二分法迭代的次数少了很多。但是因为迭代过程中，二分法只需要求解 x_0 处的函数值，而牛顿法还要求解函数在迭代点处的导数，实质上相当于每一步迭代多了一些过程。不过牛顿法确实比较快，只需要0.001秒。

3. 简化 Newton 迭代法

上文的结果可以看出，虽然牛顿法比二分法迭代次数少了很多，但是这是一场迭代次数与每次迭代工作量的博弈。于是简化 Newton 应运而生，其核心思想是减少了每次迭代过程中函数的求解，用一个近似的常数进行替代。

3.1. 算法设计

简化牛顿法，也称平行弦法。其迭代公式为

$$x_{k+1} = x_k - C f(x_k), \quad C \neq 0, \quad k = 0, 1, \dots$$

也就是把牛顿法的 x_0 处导数的倒数用常数替代。这样替代可以减少每次求导，因为根附近的导数差别更低阶，所以可以做这样的近似处理。

3.2. 数值实验

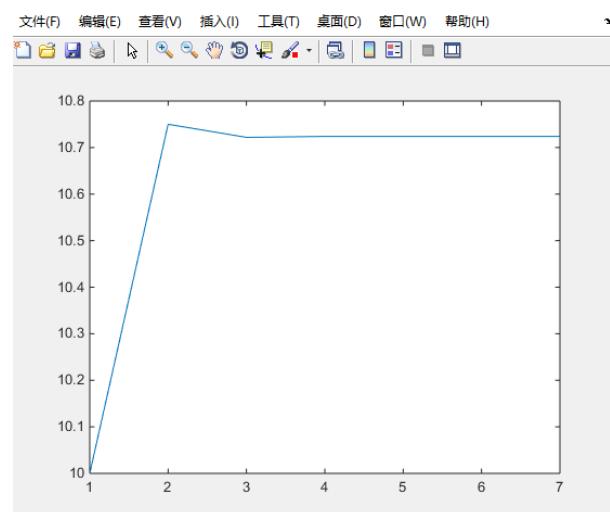
为了求115的平方根，不断对非线性函数进行线性化。可以通过迭代来实现简化牛顿法求根，迭代初始条件我选定的是 $x = 10$ 。同牛顿法，之后不断根据 x 点处切线的估值 ($C = 20$) 与 x 轴的交点迭代，确定新的结果。迭代的终止条件是如果某一点的函数值小于一个小量 eps ，那么迭代停止。

3.3. 结果分析

如图是利用简化牛顿法求解未知数的平方根的结果，其收敛曲线，还有运行时间。可以发现，一共迭代了7次找到根10.723805，比牛顿法迭代的次数多。是因为迭代过程中，牛顿法还要求解函数在迭代点处的导数。不过简化版牛顿法比牛顿法较慢，需要0.002秒。可见求导对运行造成的影响比迭代多次造成的影响要大。不过这个结论还需要在大量的计算中验证。

The screenshot shows the MATLAB interface with several windows open:

- Current File**: Shows the file `NewtonPro.m` selected. The code implements the Newton-Raphson method to calculate the square root of a number `a` starting from an initial guess `x` with a tolerance `eps`. It includes a comment about enhanced Newton's method.
- Variables**: Shows the variable `ans` with the value `10.750010721`.
- Command Window**: Displays the result of the calculation: `ans = 10.750010750010721`. The output is highlighted with a red oval.



The screenshot shows the WinDbg interface with the 'Search Summary' tab selected. Below it is a table with the following data:

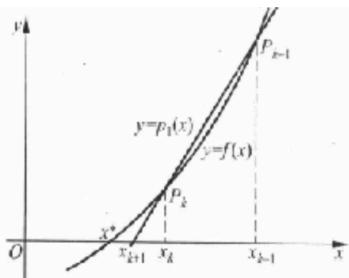
函数名称	调用	总时间	自用时间	总时间图 (深色条带 = 自用时间)
NewtonPro	1	0.002 s	0.002 s	

4. 弦截法

牛顿法利用曲线上单点切线与 x 轴的交点不断迭代求根。但这种迭代效率不是很高，当考虑到前后迭代数值差别很大的时候——切线决定的迭代更新变化不够大。如果用曲线双点进行迭代，当迭代点数值差别巨大时，效果较好。

4.1. 算法设计

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} 。



因此有迭代公式：

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

4.2. 数值实验

为了求 $\sqrt{115}$ 的平方根，不断对非线性函数进行线性化。可以通过迭代来实现弦截法求根，迭代初始条件我选定的是 $x_0 = 10, x = 11$ 。同牛顿法，之后不断根据 x 点处弦与 x 轴的交点迭代，确定新的结果。迭代的终止条件是如果某一点的函数值小于一个小量 eps ，那么迭代停止。

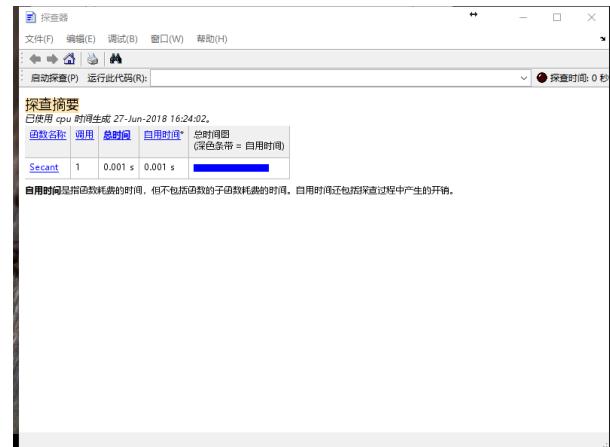
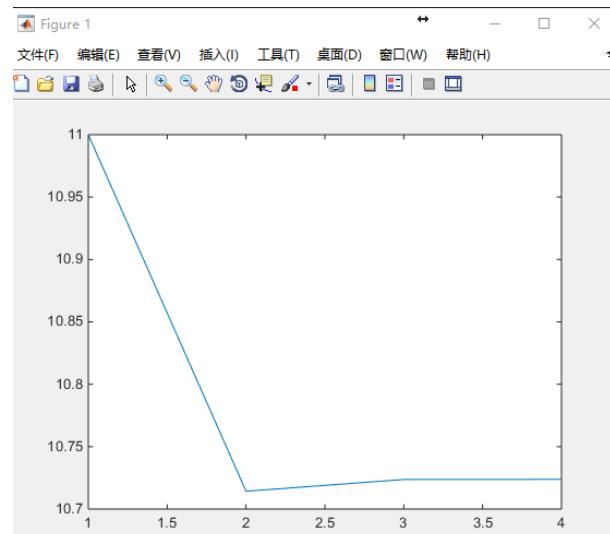
4.3. 结果分析

如图是利用弦截法求解未知数的平方根的结果，其收敛曲线，还有运行时间。可以发现，一共迭代了4次找到根 10.723805 ，我相信，在处理更大数的平方根的时候，弦截法会在性能上比牛顿法体现更多优越性。因为弦截法利用弦进行点更新，所以更新速度更快，效果更明显。需要0.001秒，完成迭代。

“迭代次数即效率”这句话在这个方法依然成立！

The screenshot shows the MATLAB interface with the following details:

- Editor:** Displays the `Secant.m` file content. The code implements the Secant method for calculating the square root of a number `a`. It takes initial values `x0` and `x`, and an error tolerance `eps`. The function returns the root `ans` and the iteration number `iter`.
- Command Window:** Shows the command `>> Secant(115)` and the output `ans = 10.714285714285714 10.723884210826318 10.723805348531346`. The last value is circled in red.
- Variable Browser:** Shows the variable `ans` with the value `10.714285714285714`.



四. 实验心得

实验把对未知量求根的问题转换成了一个非线性方程的求解问题。通过对各种方法的比对，得出了些结论。

我发现二分法求线性方程的根效率最低，因为它的迭代次数比其他方法都要多很多次。

除此之外，在未知数比较小的时候，比如这次实验所采用的115，没法看出弦截法和简化牛顿法对普通牛顿法性能上的提升。因为当未知数的数值比较小的时候，迭代对上一步的更新差距不大。如果迭代初始值离真实根差了很多，或者原未知量比较大，牛顿法的劣势可能暴露出来——迭代次数多、耗时长。

简化牛顿法对牛顿法的改进主要是体现在对导数的估计。这个做法的好处是，省略了每次的导数计算，而且在根附近迭代导数差别不大，相当于把求导过程 inline 化。然而这种优化是在牺牲准确性的基础上的，如果 C 找得不准，迭代次数很可能会显著提升。

弦截法较简化牛顿法更加准确，因为弦上的两点都在曲线上。可是迭代的初始条件变成了两个点。没次做的运算也要比简化牛顿法复杂。

这次实验的重点是理解这几种方法的几何意义，难点在于对各种方法性能的分析及它们分别适用的情景。

实验三

一. 实验环境和工具

在 Windows 10 环境下利用 Matlab-R2014b 完成实验。

二. 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 $m \times n$ 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 $n=10$, $m=100$ 。 (本身实验要求是设定一个 $100000 * 10$ 的超定方程, 但是考虑到运行时间过长、结果太过密集, 所以才用了 $100 * 10$ 的超定方程) 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。

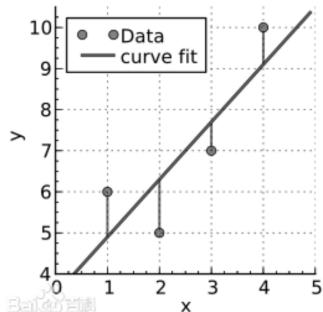
三. 实验内容

1. 最小二乘法求解超定方程

设线性方程组 $Ax=b$, 其中 A 为 $m \times n$ 维的已知矩阵, b 为 m 维的已知向量, x 为 n 维的未知向量。如果矩阵的行数大于矩阵的列数, 也就是 $m > n$, 那么称这个方程组为超定方程组。最小二乘法常常用于超定方程组的求解, 求得的解是最小二乘解。把每一行看作是一次迭代, 那么共可以迭代 m 次, 找到方程的最小二乘解。

1.1. 算法设计

最小二乘估计法, 又称最小平方法。是一种数学优化技术。它通过最小化误差的平方和寻找数据的最佳函数匹配。利用最小二乘估计法可以简便地求得未知的数据, 并使得这些求得的数据与实际数据之间误差的平方和为最小。



超定方程 $Ax=b$ 的最小二乘解 x^* 可以看作是 $f(x) = Ax - b$ 的零点逼近值。

微观来看, 如果矩阵的每一列分别计作 a_1, a_2, a_3, \dots 最小二乘解 x^* 的每一行计作 x_1, x_2, x_3, \dots 那么, 矩阵的最小二乘解 x^* 可以看成是 $a_1^T x_1 + a_2^T x_2 + \dots + a_n^T x_n - b$, 对每一行 i , 进行 `fminunc` 函数极小值运算 $\text{abs}(a_{1i}^T x_1 + a_{2i}^T x_2 + \dots + a_{ni}^T x_n - b_i)$, 得到相应的 x^* 的每一行。用第 i 行的计算结果作为第 $i+1$ 行的初始值, 不断迭代。最终可以得到该超定方程的总体最小二乘。

通过证明, 可以确定最小二乘解只有一个, 可以通过平方根法和 SOR 方法求得。此外, 通过对必要性和充分性的证明, 可以得到如下定理。

定理：如果 x^* 是超定方程的最小二乘解，那么其充要条件是 $A' * A * x = A' * b$ 。

它是关于 x_1, x_2, x_3, \dots 的方程组，被称为正规方程组或者法方程组。我们用上述定理获得最小二乘解，并用它来验证所求得的迭代最小二乘解的准确性。

1.1.2. 简单案例

为了验证方法的准确性，我们首先设置一个简单的超定方程： m 为 4； n 为 2；矩阵 $A = [2, 4; 3, -5; 1, 2; 2, 1]$ ；向量 $b = [11; 3; 6; 7]$ 。用定理求得的该超定方程的最小二乘解如图。

The screenshot shows the MATLAB R2014b interface with the following details:

- Current Window:** The "Proof.m" script is open in the editor.
- Script Content:**

```
function ans = Proof(A, b)
% Least Square Method
% Func func obtaining the least square solution.
% Param A is the matrix to be solved.
% Param b is the vector to be solved.
% Return ans is the ideal least square solution.
% Created by Nino on 2018/6/25. Copyright ? Nino 2018.
%
% long formatted
format long;

% arguments adjustment
if nargin == 0
    A = [2, 4; 3, -5; 1, 2; 2, 1];
    b = [11; 3; 6; 7];
end

% as proved. A'*A*x = A'*b
ans = (A'*A)\(A'*b);
```
- Command Window:** Shows the command `>> Proof` and the resulting output:

```
ans =
3.040293040293041
1.241758241758242
```

The last two digits of the second result are circled in red.
- Workspace:** Shows the variable `ans` with the value `[3.04029312418]`.

看出，对于该超定方程，其最小二乘解两项分别是 3.040293、1.241758。

MATLAB R2014b

编辑器 - C:\Users\pc01\Documents\MATLAB\LSM\LSMIter.m

```

1 % LSM Iteration Method
2 % Least Square Method
3 % Using LSM to solve overdetermined equation problem.
4 %
5 % Param A is the matrix to be solved.
6 %
7 % Param b is the vector to be solved.
8 %
9 %
10 % Return xiters is the approximate value of solution during iterations.
11 %
12 % Return erriters is the error during iterations.
13 %
14 % Created by Nino on 2013/6/25. Copyright ? Nino 2013.
15 %
16 %
17 %
18 % long formatted
19 % format long;
20 %
21 % global value
22 % global A;
23 % global b;
24 % global iter;
25 %
26 % argument adjustment
27 if margin == 0

```

命令行窗口

```

fminunc stopped because it cannot decrease the objective function
along the current search direction.

<stopping criteria details>

ans =

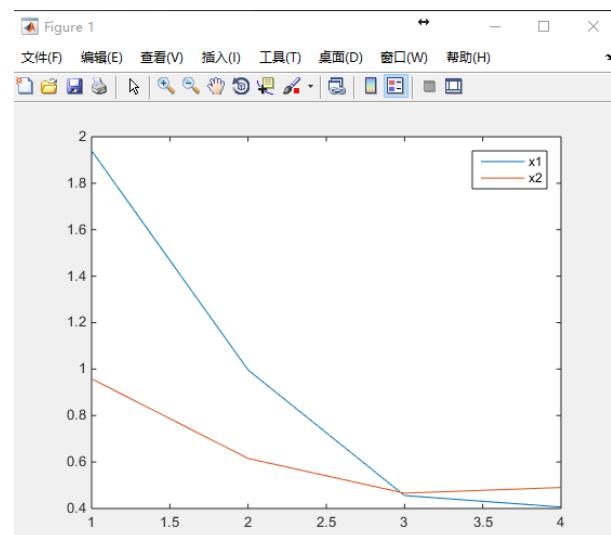
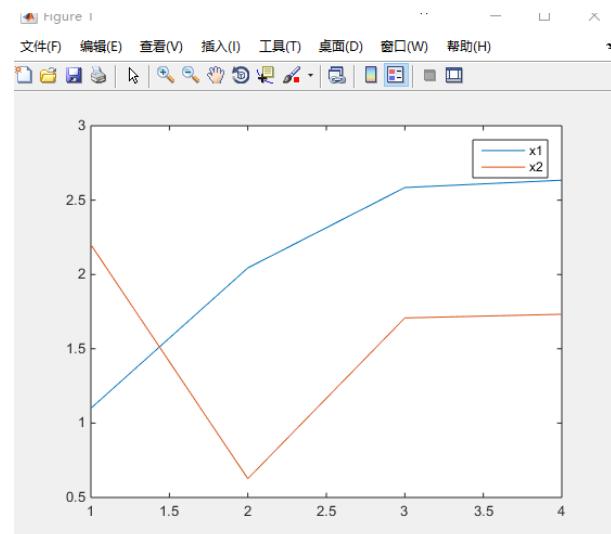
```

```

1.099999999999373 2.04417826425863 2.884705873198251 2.833882346290204
2.199999999999745 0.626470576599935 1.707647053778394 1.73235290049466

```

fx >



接下来用迭代最小二乘法求解，每次迭代得到的解、收敛曲线、误差曲线如图。发现得到的结果是 2.633882 和 1.732235，迭代的误差两项都逐渐递减。

但是收敛的结果好像不一样啊？

事实上，这个结果误差是因为迭代次数相对于所求向量太少以及初始值离真实值太远造成的。

因为要求出的最小二乘解有 2 项，但只是迭代了 4 次，相对来说迭代次数最少。除此之外，如果迭代的 x 的初始值更靠近真实值，那么 4 次迭代也足以让结果收敛。通过后来的实验——改变了 x 的初始值——最终得到了离目标值足够近的最小二乘解。

1.2. 数值实验

为了探索用迭代最小二乘法求超定方程组的最小二乘解的算法性能和适用条件。我们设置了一个 $100 * 10$ （本身实验要求是设定一个 $100000 * 10$ 的超定方程，但是考虑到运行时间过长、结果太过密集，所以才用了 $100 * 10$ 的超定方程）的超定方程。

1.2.1. 实验条件

更普遍的应用需要一些条件制约。

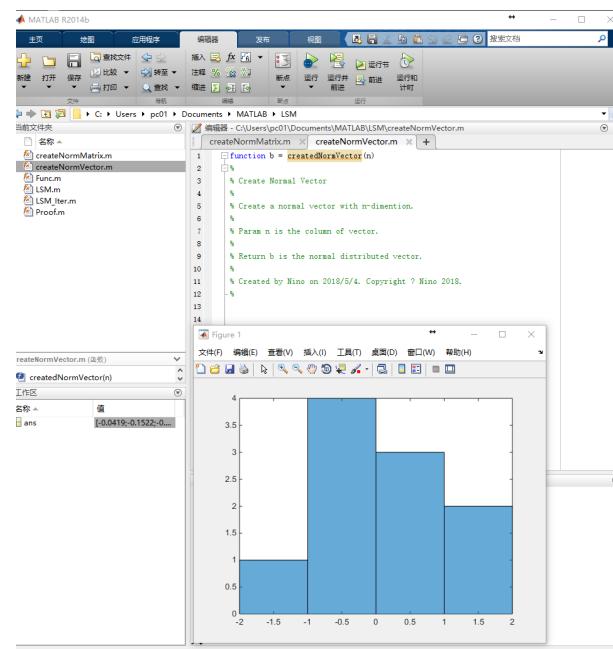
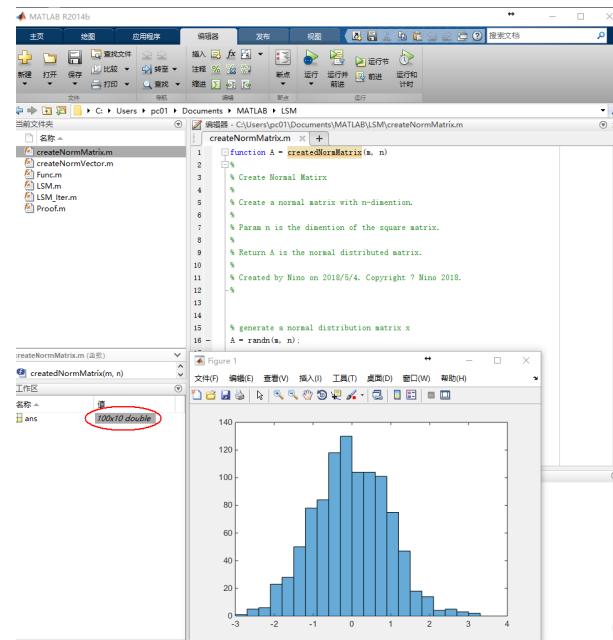
首先需要用 `randn` 函数创建独立同分布的正太分布的矩阵和向量。考虑到行数比列数足够多（尤其考虑到原计划求解的超定方程是 $100000 * 10$ 的，行数是列数的 $10e2$ ），迭代次数相较于迭代所求未知量足够大，所以初识迭代值不再另外设有额外的要求。

1.2.2. 附加步骤

用 `randn` 函数创建独立同分布的正太分布的矩阵和向量。用 `LSM` 函数对得到的 `LSMIter` 函数结果进行归纳。依然利用 `Proof` 函数进行结果比对。

1.3. 结果分析

创建独立同分布的正太分布的矩阵和向量的直方图如图所示，结果确实呈现正态分布。



运行 LSM 函数，求解超定方程的最小二乘解，最小二乘解每次的迭代结果、收敛曲线、迭代误差如图所示。

```

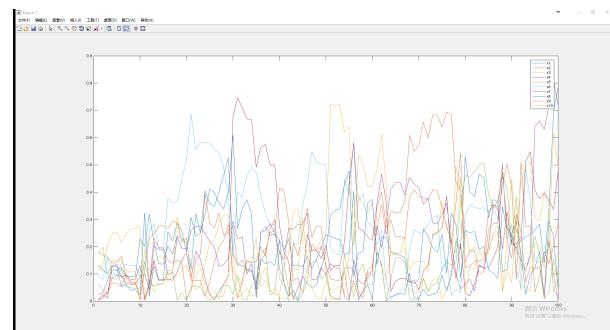
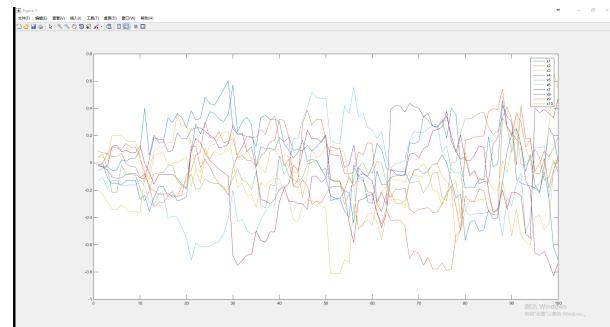
% global value
global A;
global b;
global iter;
% long formatted
format long;

% argument adjustment
if nargin == 0
    n = 100;
    n = 10;
end

% create matrix and vector
A = createNormMatrix(n, n);
b = createNormVector(n);

% get the xters, errters, and the answer of iteration
[Iterers, errters, ans1] = LSM_Iter(A, b);

```



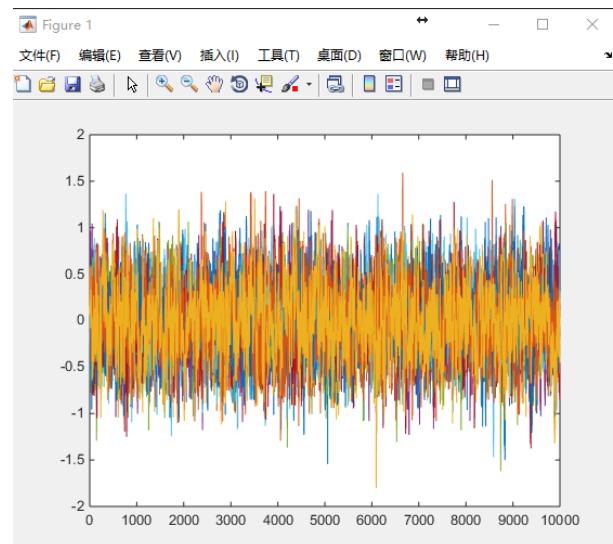
通过结果图，我们可以看出，迭代结果到了一定步骤之后并没有逐渐收敛，有些步骤甚至还会发散。迭代误差到了一定步数之后不一定持续变小，甚至还有可能增多，但是迭代的误差始终没有超过1。出现这种结果的原因分析是矩阵中的有些方程可能有较大误差，考虑到

迭代的更改作用，新的值可能在迭代之后靠近这个误差大的方程，从而整体上偏离了最小二乘解。

这种问题的改善方法有两种。

首先，如果行数是 100000 行，这种问题可能会有所改善。但是治标不治本，如果最后几个方程误差都很大，那么可能会让之前繁杂的运算前功尽弃。

对每个方程得到的结果加权。这样误差大的方程的错误更改影响相对小。采用平均加权已经显示出了良好的运行效果。如果利用 fminunc 函数返回的误差值 xval 进行更复杂的过滤型加权，排除误差大的量，每个方程所占权值不同。通过这种运算，结果会更准确。



最后，实现了 $100000 * 10$ 的独立同分布正态分布超定方程的最小二乘法运行示例。因为图像太密集，文件夹中有保存下来的 ans.mat 文件。发现方程的根中的元素，都最终收敛于0附近，满足预期。

四. 实验心得

这次实验思路比较复杂。重点是理解最小二乘法如何应用到超定方程的求解，如何进行迭代；难点在求最小值函数的使用。

实验的结果显示，误差并没有随着迭代逐渐减小，主要是有些方程可能有较大误差造成的，考虑到迭代的更改作用，新的值可能在迭代之后靠近这个误差大的方程，从而整体上偏离了最小二乘解。但是这种问题会随着超定方程迭代次数的增加逐渐改善。

最开始我是用的加权最小二乘法，效果要好于迭代最小二乘法。但在大量迭代的过程中这种算法复杂度更高（每一步迭代都要加权），而且随着迭代次数的增加，每一次的迭代结果可能会更相近，没有必要每一次再加权。**至此我也理解了为什么实验要求初定为 $100000 * 10$ 的超定方程——迭代次数足够使结果精确。**

通过对各种方法的比较，分析出问题出现的原因和每种方法的优劣是这次实验的趣味所在。

实验四

一. 实验环境和工具

在 Windows 10 环境下利用 Matlab-R2014b 完成实验。

二. 问题描述

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。

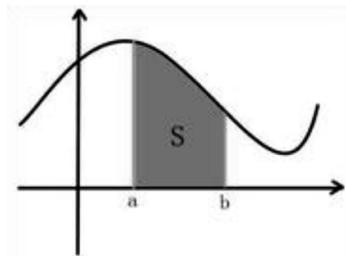
三. 实验内容

Newton Cotes 公式在更高阶的情况下具有不稳定性。为了提升求积精度，通常把积分区间等分化成若干子区间，在每个子区间上用低阶求积公式求积，称为复合求积法。

1. 复合梯形公式

复合梯形公式是把区间等分，看成小的梯形进行积分的。可以看作是小区间内的一阶 Newton Cotes 求积。

1.1. 算法设计



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 h 。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。

复合梯形公式是：

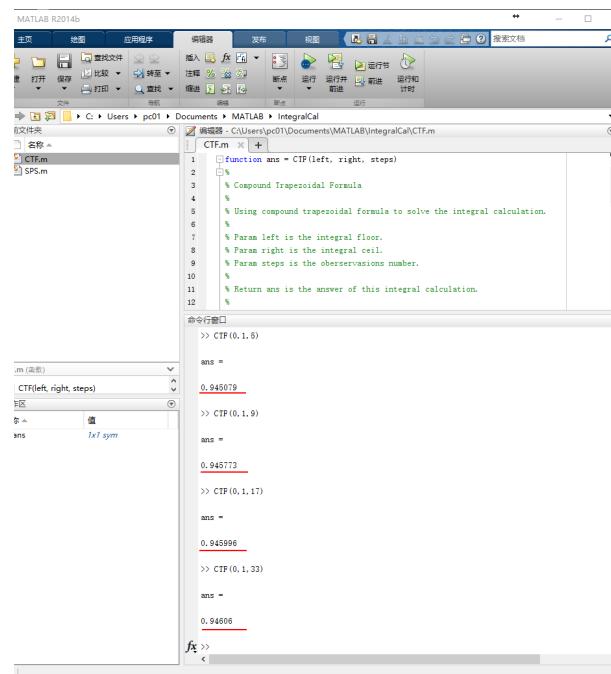
$$T_n = \frac{h}{2} \sum_{k=0}^{n-1} [f(x_k) + f(x_{k+1})] = \frac{h}{2} [f(a) + 2 \sum_{k=1}^{n-1} f(x_k) + f(b)]$$

1.2. 数值实验

为了求 $\sin(x)/x$ 在 $[0, 1]$ 范围内的积分，对区间进行 n 等分，求解各子区间的面积，并累加。

1.3. 结果分析

如图是利用复合梯形法求解积分的结果，输入参数分别是积分上下限和步长。发现随着取点数的增加，结果逐渐靠近真实值，可见区间划分越细，结果越准确。



2. 复合 Simpson 法

复合辛普森公式是把区间等分，可以看作是小区间内的二阶 Newton Cotes 求积。

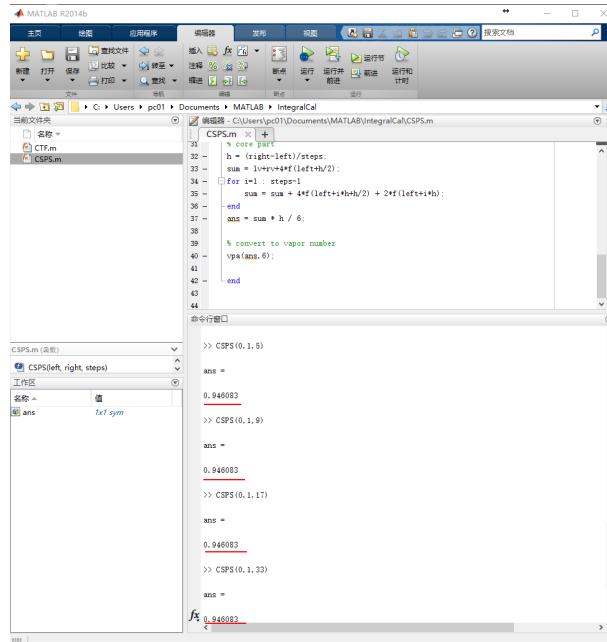
2.1. 算法设计

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 h 。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，各子区间进行辛普森积分，再累加各区间的积分，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。

2.2. 数值实验

为了求 $\sin(x)/x$ 在 $[0, 1]$ 范围内的积分，对区间进行 n 等分，求解各子区间的面积，并累加。

2.3. 结果分析



如图是利用复合辛普森法求解积分的结果，输入参数分别是积分上下限和步长。发现随着取点数的增加，结果逐渐靠近真实值，可见区间划分越细，结果越准确。

四. 实验心得

实验主要是应用复合法求解积分。

一开始的时候，我没有注意到要用复合 Simpson 法求解，于是先利用 Simpson 方法求解。不过歪打正着，对比了复合 Simpson 法与 Simpson 法之后，发现复合法的思想确实可以提升精度。

这次实验同时巩固了我对 Newton Cotes 公式的理解。之前一直没有注意到这个方法的局限性——高阶不稳定性，利用这次实验，我重新查看了牛顿科特斯方法。另外利用复合辛普森方法强化了我对辛普森法的理解。

这次实验的重点是掌握复合法的好处和如何利用，难点是理解牛顿科特斯法的不稳定性。

实验五

一. 实验环境和工具

在 Windows 10 环境下利用 Matlab-R2014b 完成实验。

二. 问题描述

分别采用前向欧拉法、后向欧拉法、梯形方法和改进欧拉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。

三. 实验内容

求解常微分方程虽然有很多解析方法, 但是解析方法只能用来求解一些特殊类型的方程, 实际问题中归结出来的常微分方程主要还是通过数值方法求解。

所谓数值解法, 就是寻求解 $y(x)$ 在一系列离散节点上的近似值, 相邻节点的间距称为步长。

1. 欧拉法

欧拉法是一种简单的数值求解常微分方程初值问题的方法。

1.1. 算法设计

从初始点出发, 依方向场在该点的方向推进到下一点, 最终得到一条折线。利用常微分方程 $y' = y - 2x/y$ 中该点处斜率与数值之间关系, 得到:

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

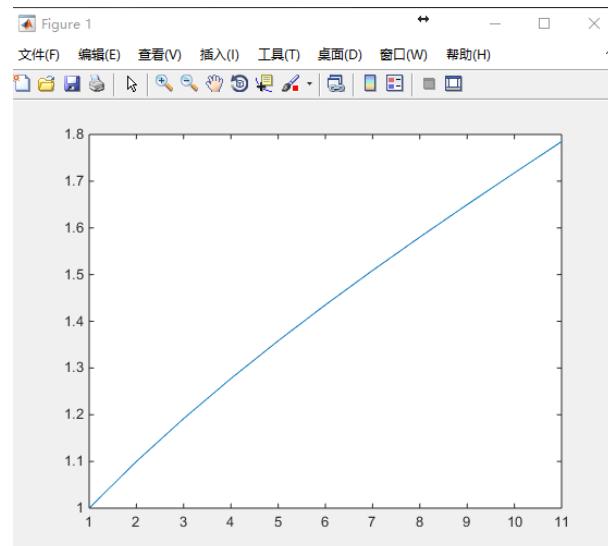
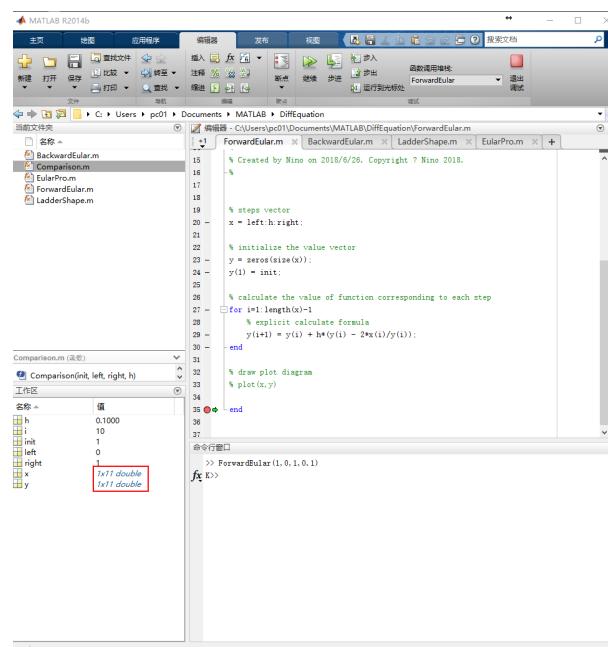
$$y_{n+1} = y_n + h f(x_n, y_n)$$

不断进行迭代, 得到离散节点的数值集合。

1.2. 数值实验

用欧拉法求解常微分方程 $y' = y - 2x/y$ 的解集。根据算法设计中得到的公式，不断进行迭代，最终归纳出求解区间的数值估计，并描绘成点阵图。

1.3. 结果分析



由图，得到了用欧拉法得到的常微分方程在给定区间间隔步长的解集。

2. 后退欧拉法

后退欧拉法不同于前向欧拉法，它是隐式的。需要先用欧拉法进行一次计算，算出迭代下一点的初始估计值。不断对该值进行迭代，直到收敛，能够推出下一个迭代的点。这样做的好处是每一步迭代更加精确，但是这种精确度提升是建立在每一次迭代增加运算量的基础上的，并且提升了算法的时间复杂度。

2.1. 算法设计

从初始点出发，依方向场在该点的方向推进到下一点，最终得到一条折线。利用常微分方程 $y' = y - 2x/y$ 中下一个迭代点斜率与数值之间关系，发现下一点的迭代斜率 $f(x_{n+1}, y_{n+1})$ 不能直接获得（**隐式**），所以需要先用前向欧拉法进行一次求值，然后不断迭代（**迭代的终止条件是迭代斜率收敛**）。这样的迭代结果更加精确。

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

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

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

2.2. 数值实验

用后退欧拉法求解常微分方程 $y' = y - 2x/y$ 的解集。根据算法设计中得到的公式，不断进行迭代，最终归纳出求解区间的数值估计，并描绘成点阵图。

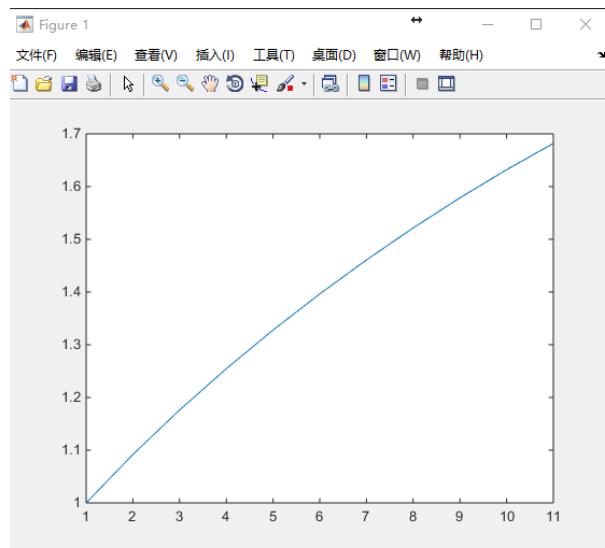
2.3. 结果分析

The screenshot shows the MATLAB R2014b interface. The current file is `BackwardEuler.m`, which contains the following code:

```
% Backward Euler method for y' = y - 2x/y
% Initialize
x = 0; y = 1;
h = 0.1;
n = 10;
for i=1:n
    % implicit calculate formula
    tmp = y + h*(y(i) - 2*x(i)/y(i));
    prev = tmp;
    for j=1:MAX
        y(j+1) = y(i) + h*(tmp - 2*x(i+1)/tmp);
        if abs(y(j+1) - prev) < eps
            disp(j); %display iteration number
            break;
        end
        prev = y(j+1);
    end
end
% draw plot diagram
```

The workspace window shows the following variables:

名称	值
eps	1.0000e-07
h	0.1000
i	10
int	1
j	2
left	0
MAX	100
prev	1.6819
right	1
tmp	1.6850
x	7x17 double
y	7x17 double



由图，得到了用后退欧拉法得到的常微分方程在给定区间间隔步长的解集。输出每一步迭代种收敛迭代的次数，法相2次就可以收敛。

3. 梯形法

需要先用欧拉法进行一次计算，算出迭代下一点的初始估计值。不断对该值进行迭代——迭代公式的不同是与后退欧拉法的主要区别，直到收敛，能够推出下一个迭代的点。和后退欧拉法一样，这样做的好处是每一步迭代更加精确，但是这种精确度提升是建立在每一次迭代增加运算量的基础上的，并且提升了算法的时间复杂度。梯形法可以看作是后退欧拉法的改进。

3.1. 算法设计

从初始点出发，依方向场在该点的方向推进到下一点，最终得到一条折线。利用常微分方程 $y' = y - 2x/y$ 中下一个迭代点斜率与数值之间关系，发现下一点的迭代斜率 $f(x_{n+1}, y_{n+1})$ 不能直接获得（**隐式**），所以需要先用前向欧拉法进行一次求值，然后不断迭代（**迭代的终止条件是迭代斜率收敛**，此处的迭代公式与后退欧拉法不同，**迭代斜率是该点和下一次迭代点的平均值**）。这样的迭代结果更加精确。

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

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

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

3.2. 数值实验

用梯形法求解常微分方程 $y' = y - 2x/y$ 的解集。根据算法设计中得到的公式，不断进行迭代，最终归纳出求解区间的数值估计，并描绘成点阵图。

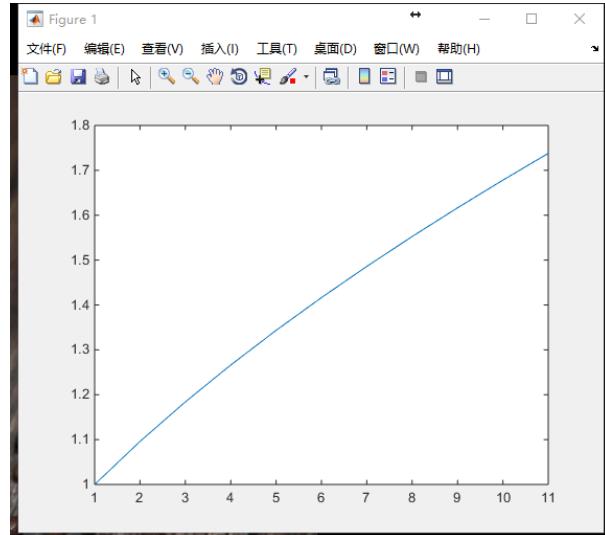
3.3. 结果分析

The screenshot shows the MATLAB R2014b interface. The code editor displays three files: ForwardEular.m, BackwardEular.m, and LadderShape.m. The LadderShape.m file contains the following code:

```
% calculate the value of function corresponding to each step
for i=1:length(x)-1
    % implicit calculate formula
    tmp = y(i) + h*(y(i) - 2*x(i)/y(i));
    prev = tmp;
    for i=1:h
        y(i+1) = y(i) + h/2*( (y(i)-2*x(i)/y(i)) + (tmp-2*x(i+1)/tmp) );
        if abs(y(i+1) - prev) < eps
            disp(i); % display the iteration number
            break;
        end
        prev = y(i+1);
    end
end
```

The command window shows the output of the LadderShape function:

```
>> LadderShape(1, 0.1, 0.1)
2
2
2
2
2
2
2
2
2
2
```



由图，得到了用梯形法得到的常微分方程在给定区间间隔步长的解集。输出每一步迭代种收敛迭代的次数，法相2次就可以收敛。

4. 改进欧拉法

改进欧拉法也是隐式单步迭代，只不过不同于之前的后退欧拉法和梯形法，改进欧拉法在迭代过程中不再需要对隐式变量利用迭代求解。这种方法最大的优点是，比起后退欧拉法和梯形法，它降低了时间复杂度，并且保证了精确度。

4.1. 算法设计

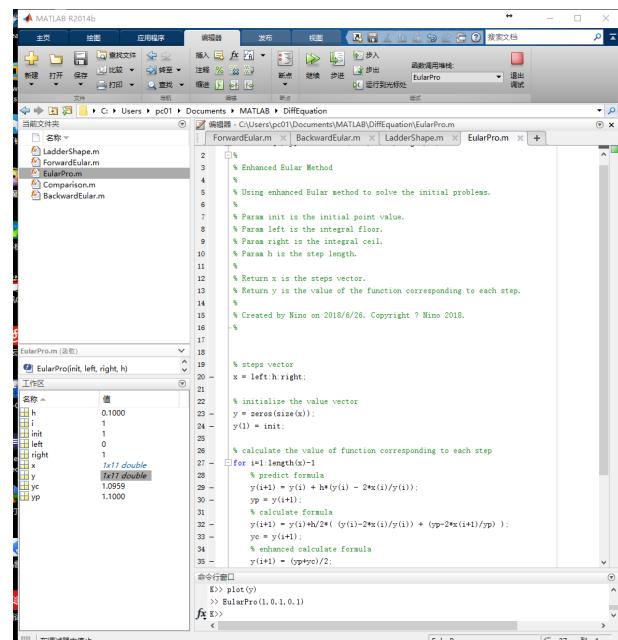
从初始点出发，依方向场在该点的方向推进到下一点，最终得到一条折线。利用常微分方程 $y' = y - 2x/y$ 中下一个迭代点斜率与数值之间关系，发现下一点的迭代斜率 $f(x_{n+1}, y_{n+1})$ 不能直接获得（**隐式**），所以需要先用前向欧拉法进行一次求值，得到估计值 $f(x_{n+1}, y_{n+1}) = y_p$ 。用估计值 y_p 带入隐式函数，得到计算值 y_c 。这样的迭代结果更加精确，而且省去了迭代收敛过程。

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

4.2. 数值实验

用改进欧拉法求解常微分方程 $y' = y - 2x/y$ 的解集。根据算法设计中得到的公式，不断进行迭代，最终归纳出求解区间的数值估计，并描绘成点阵图。

4.3. 结果分析



The screenshot shows the MATLAB R2014b interface with the Enhanced Euler Method code in the Editor window. The code implements the improved Euler method to solve an initial value problem. It includes comments explaining parameters like step length h , initial point x_0 , and interval boundaries $left$ and $right$. The code initializes a value vector y and iterates through steps to calculate function values at each point, using both forward and backward Euler formulas to estimate the next value y_{i+1} .

```
% Enhanced Euler Method
% Using enhanced Euler method to solve the initial problems.
%
% Param init is the initial point value.
% Param left is the integral floor.
% Param right is the integral ceiling.
% Param h is the step length.
%
% Return x is the steps vector.
% Return y is the value of the function corresponding to each step.
%
% Created by Nino on 2018/6/26. Copyright ? Nino 2018.

% steps vector
x = left:h:right;

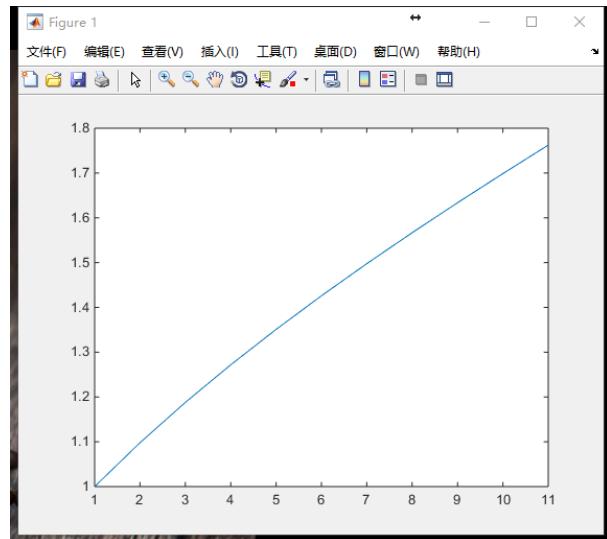
% initialize the value vector
y = zeros(size(x));
y(1) = init;

% calculate the value of function corresponding to each step
for i=1:length(x)-1
    % predict formula
    y(i+1) = y(i) + h*(y(i) - 2*x(i)/y(i));
    yp = y(i+1);

    % calculate formula
    y(i+1) = y(i) + h/2*( (y(i)-2*x(i)/y(i)) + (yp-2*x(i+1)/yp) );
    yc = y(i+1);

    % enhanced calculate formula
    y(i+1) = (yp+yc)/2;
end

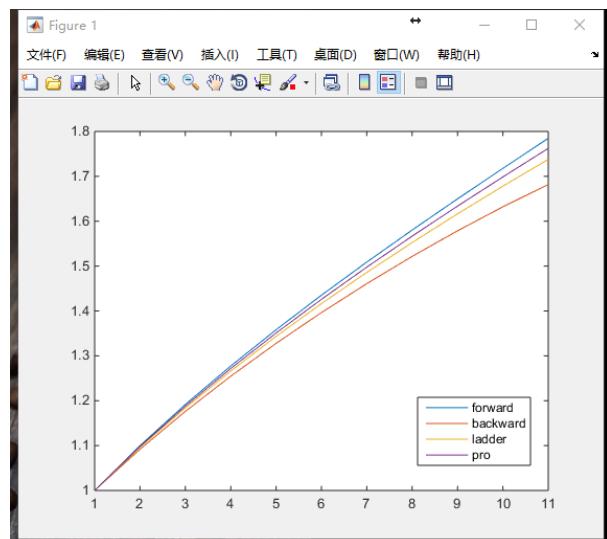
% plot
plot(y)
>> EulerPro(1, 0.1, 0.1)
```



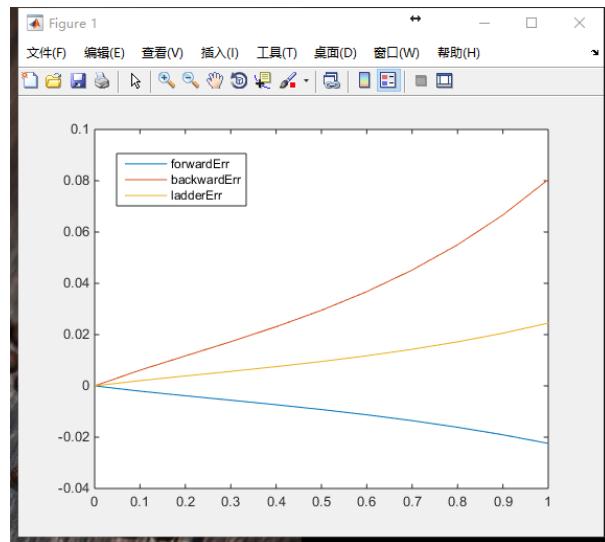
由图，得到了用改进欧拉法法得到的常微分方程在给定区间间隔步长的解集。

5. 对比

为了对比几种方法求得结果，绘制几种方法的解集曲线，以及前三种方法相对于改进欧拉法的绝对误差，结果如图。



通过这张图可以看出，四条曲线都是逐步上升并靠近，最终都达到了1.7附近。为了得到更直观的差值分析，绘制欧拉法、后退欧拉法、梯形法相对于改进欧拉法的误差曲线。



可见四种方法得到的结果差距都不是很大，后退欧拉法没有体现出每一步迭代精细化的优势，梯形法和欧拉改进法靠得最近，这两种方法得出的结果应该相对来说准确。

四. 实验心得

这次实验利用了欧拉法和相关改进措施求解常微分方程的初值问题。实验最开始我没有对后退欧拉法和梯形法进行收敛迭代。不过发现进行了收敛迭代之后的效果差距不是太大，因为收敛迭代只有两次。此外，每种改进方法对前一种的提升都有助于理解欧拉法。我认为，对不同改进优劣的思考更加有助于我们理解数值计算这门学科。

本来这次实验是没有第五步对比这个步骤的，参考了网上的博客，发现把几种方法联系到一起确实有很直观的理解。

本次实验的重点是理解欧拉法求解常微分方程的初值问题迭代的几何意义以及各种方法的优势劣势难点在于理解后几种方法迭代的过程，以及对结果的分析。

之后我还计划实现一下龙格库塔算法，对比它们的性能以及结果。

实验六

一. 实验环境和工具

在 Windows 10 环境下利用 Matlab-R2014b 完成实验。

二. 问题描述

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。

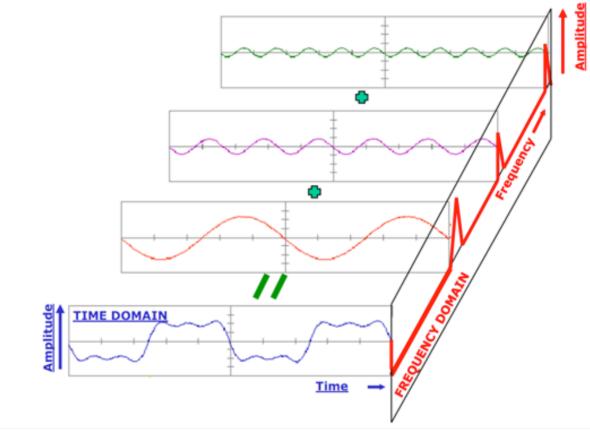
三. 实验内容

1. 快速傅立叶变换

傅立叶变换是一种分析信号的方法，它可分析信号的成分，也可用这些成分合成信号。许多波形可作为信号的成分，比如正弦波、方波、锯齿波等，傅立叶变换用正弦波作为信号的成分。

傅里叶变换属于谐波分析；傅里叶变换的逆变换容易求出，而且形式与正变换非常类似；正弦基函数是微分运算的本征函数，从而使得线性微分方程的求解可以转化为常系数的代数方程的求解。在线性时不变的物理系统内，频率是个不变的性质，从而系统对于复杂激励的响应可以通过组合其对不同频率正弦信号的响应来获取；卷积定理指出：傅里叶变换可以化复杂的卷积运算为简单的乘积运算，从而提供了计算卷积的一种简单手段；离散形式的傅立叶变换可以利用数字计算机快速地算出（其算法称为快速傅里叶变换算法）。

FFT 的基本思想是把原始的 N 点序列，依次分解成一系列的短序列。充分利用 DFT 计算式中指数因子所具有的对称性质和周期性质，进而求出这些短序列相应的 DFT 并进行适当组合，达到删除重复计算，减少乘法运算和简化结构的目的。此后，在这思想基础上又开发了高基和分裂基等快速算法。



Matlab 中提供了 **fft** 函数进行快速傅立叶变换，输入离散的采集信号数据，就可以得出相应的快速傅立叶变换的复数序列结果。

1.1. 算法设计

由于快速傅立叶变换的公式较为复杂，此处还未能自行手动实现 FFT 算法，暂且用 Matlab 的 API **fft** 方法来进行测试。

根据采样定理，**fft** 能分辨的最高频率为采样频率的一半（即Nyquist频率），函数 **fft** 返回值是以 Nyquist 频率为轴对称的，Y 的前一半与后一半是复数共轭关系。

作 FFT 分析时，幅值大小与输入点数有关，要得到真实的幅值大小，只要将变换后的结果乘以 2 除以 N 即可（但此时零频—直流分量—的幅值为实际值的 2 倍）。对此的解释是：Y 除以 N 得到双边谱，再乘以 2 得到单边谱（零频在双边谱中本没有被一分为二，而转化为单边谱过程中所有幅值均乘以 2，所以零频被放大了）。

若分析数据时长为 T，则分析结果的基频就是 $f_0=1/T$ ，分析结果的频率序列为 $[0:N-1]*f_0$ 。

使用 N 点 FFT 时，不应使 N 大于 y 向量的长度，否则将导致频谱失真。

最后用 **plot** 图画出原始信号的波形，用 **stem** 画出输出的复数序列。幅值显著增高的点对应的值除以频率就是快速傅立叶变换的余弦参数。

1.2. 数值实验

要求给出快速傅立叶变换的采样点数——原始信号长度、采样频率以及快速傅立叶变换余弦参数序列。首先绘制原始信号。之后进行快速傅立叶变换处理，得到了复数序列的 **stem** 图后，幅值突出点与余弦参数序列进行比对。分别设置不同的点数 1024 和 2048；为了方便比对，频率都设置为 1Hz；余弦参数序列的值在 [0,0.5] 之间。

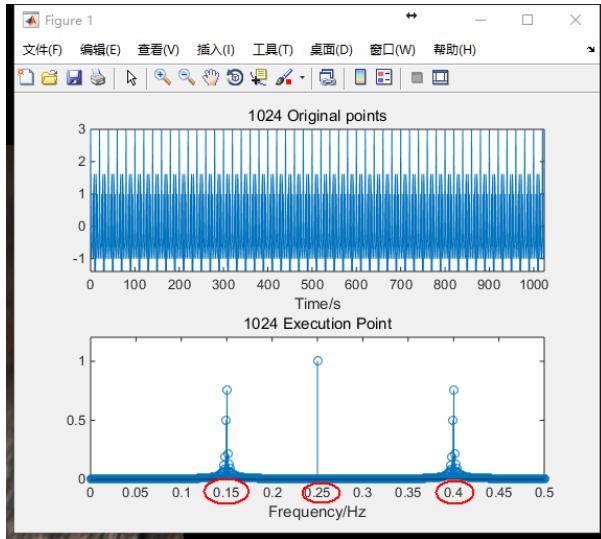
1.3. 结果分析

1.3.1. 当 N = 1024 时：

```
%>> FFT(1024, 1, [18, 28, 4])
f<>
c<>
```

```
%>> FFT(1024, 1, [18, 28, 4])
f<>
c<>
```

如图，首先设置 1024 点的 FFT 算法，输入的余弦参数为 0.15、0.25、0.4。



得到的原始信号序列的 plot 图和复数序列 stem 图如图所示。如图，进行快速傅立叶变换处理，得到了复数序列的 stem 图后，幅值突出点与余弦参数序列进行比对。可见，幅值显著增高的点对应的值除以频率就是快速傅立叶变换的余弦参数。

1.3.2. 当 $N = 2048$ 时：

如图，首先设置 2048 点的 FFT 算法，输入的余弦参数为 0.1、0.15、0.23、0.29、0.36、0.47。

The screenshot shows the MATLAB R2014b interface. The workspace contains the following code:

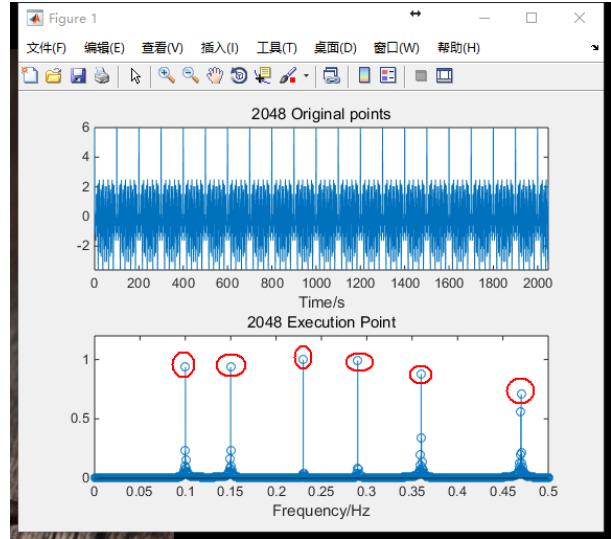
```

% Create coefficient sequence
c = [0.1 0.15 0.23 0.29 0.36 0.47];
% Sampling interval for x-axis
dt = 1/2048;
% Time sequence for x-axis
t = 0:N-1*dt;
% Original signal for y-axis
y = 0;
for i=1:size(c)
    y = y + cos(2*pi*c(i)*t);
end
% Construct diagram
subplot(2,1,1);
plot(t,y);
axis([0 N min(y) max(y)]);
xlabel('Time/s'), title('2048 Original points');

```

The command window shows the command `fft(2048,1,[1,15,23,29,36,47])` being entered.

得到的原始信号序列的 plot 图和复数序列 stem 图如图所示。如图，进行快速傅立叶变换处理，得到了复数序列的 stem 图后，幅值突出点与余弦参数序列进行比对。可见，幅值显著增高的点对应的值除以频率就是快速傅立叶变换的余弦参数。



四. 实验心得

这次实验思路比较复杂。重点是理解快速傅立叶变换的几何意义和实现方法。难点有很多：第一是掌握 FFT 较 DFT 的提升；第二是快速傅立叶变换的使用；第三就是推导 FFT 方法的过程中有很多困难的知识点。

我这次调用了 Matlab 的 fft 函数 API 实现了快速傅立叶变换的算法。如果有机会，我将来还会自己实现 FFT 这个算法，并完善实验步骤和实验结果分析。另外，在用 2048 点进行 FFT 检测的时候，发现系数超过 0.5 之后结果成倍增加的原因——与频率有关以及制图过程中对幅值的理解，对我理解 FFT 算法中参数对结果的影响很有帮助。

此外，我在做这个实验中联想到了 **DNN** —— 深度神经网络的有关知识。之后我还会对这两者的应用场景和算法进行分析，看看能不能应进行一下 fusion（或者他们本来就可能有关系）。

还需要好好学习数值计算啊！

还需要好好学习深度学习啊！

还有好多坑要填啊！