数值计算报告

16340150
刘俊峰
软件工程教务 2 班

问题描述

1. 已知 sin(0.32)=0.314567, sin(0.34)=0.333487, sin(0.36)=0.352274， sin(0.38)=0.370920。请采用线性插值、二次插值、三次插值分别计算 sin(0.35)的值。
2. 请采用下述方法计算 115 的平方根，精确到小数点后六位。
   （1）二分法。选取求根区间为[10, 11]。
   （2）牛顿法。
   （3）简化牛顿法。
   （4）弦截法。
   绘出横坐标分别为计算时间、迭代步数时的收敛精度曲线。
3. 请采用递推最小二乘法求解超定线性方程组 Ax=b，其中 A 为 m x n 维的已知矩阵，b 为 m 维的已知向量，x 为 n 维的未知向量，其中 n=10，m=10000。A 与 b 中的元素服从独立同分 布的正态分布。绘出横坐标为迭代步数时的收敛精度曲线。
4. 请编写 1024 点快速傅里叶变换的算法。自行生成一段混杂若干不同频率正弦的信号，测试所编写的快速傅里叶变换算法。
5. 请采用复合梯形公式与复合辛普森公式，计算 sin(x)/x 在[0, 1]范围内的积分。采样 点数目为 5、9、17、33。
6. 请采用下述方法，求解常微分方程初值问题 y'=y-2x/y，y(0)=1，计算区间为[0, 1]， 步长为 0.1。
   （1）前向欧拉法。
   （2）后向欧拉法。
   （3）梯形方法。
   （4）改进欧拉方法。

算法设计

1. **插值计算 sin(0.35)**
   a. 线性插值

```python
# 一次插值
def linear_interpolation(coord, x):
    k = (coord[1]['y'] - coord[0]['y']) / (coord[1]['x'] - coord[0]['x'])
    return coord[0]['y'] + k * (x - coord[0]['x'])
```

如图。采用了线性插值公式的点斜式

斜率：k=(y1-y0)/(x1-x0), Lx = k(x-x0)+y0

b. 二次插值

```
# 二次插值
def quadratic_interpolation(coord, x):
    k1 = coord[0]['y'] * (x - coord[1]['x']) * (x - coord[2]['x']) / \
    (coord[0]['x'] - coord[1]['x']) / (coord[0]['x'] - coord[2]['x'])
    k2 = coord[1]['y'] * (x - coord[0]['x']) * (x - coord[2]['x']) / \
    (coord[1]['x'] - coord[0]['x']) / (coord[1]['x'] - coord[2]['x'])
    k3 = coord[2]['x'] * (x - coord[0]['x']) * (x - coord[1]['x']) / \
    (coord[2]['x'] - coord[0]['x']) / (coord[2]['x'] - coord[1]['x'])
    return k1 + k2 + k3
```

二次插值公式：

$$L_0(x)=\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}$$

$$L_1(x)=\frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}$$

$$L_2(x)=\frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}$$

$$P_2(x)=\sum_{k=0}^{2} y_k L_{1,k}(x)$$

代码中分别用 k1,k2,k3 表示 L0,L1,L2

c. 三次插值

书本上没有现成的三次插值的公式，采用的是 N 次的拉格朗日插值即：

$$L_{N,k}(x)=\frac{(x-x_1)(x-x_2)\ldots(x-x_{k-1})(x-x_{k+1})\ldots(x-x_n)}{(x_k-x_1)(x_k-x_2)\ldots(x_k-x_{k-1})(x_k-x_{k+1})\ldots(x-x_n)}$$

$$P_N(x)=\sum_{k=0}^{N} y_k L_{1,k}(x)$$

二次插值需要三个点，三次插值则需要四个点

```python
# 三次插值
def cubic_interpolation(coord, x):
    _sum = 0
    for i in range(4):
        temp = coord[i]['y']
        product = 1
        merchant = 1
        for j in range(4):
            if j != i:
                product *= x - coord[j]['x']
                merchant *= coord[i]['x'] - coord[j]['x']
        temp = temp * product / merchant
        _sum += temp
    return _sum
```

Merchant 是 Lx 的分母，product 是 Lx 的分子

2. 请采用下述方法计算 115 的平方根，精确到小数点后六位。

a. 二分法，区间为[10,11]

基本设计思路：

1） 找到区间[a,b]的中点 x=(a+b)/2

2） 计算 fx 的值

A．若 fx>0, b=x

B．若 fx<0, a=x

C．若 fx=0，返回 x

3） 判断 fx 是否有小于 E-6 精度的误差，小于返回 x；否则回到 1）

```python
# 二分法
def dichotomy(f, x1, x2):
    current = (x1 + x2) / 2
    while(True):
        if f(current) < 0:
            x1 = current
        elif f(current) > 0:
            x2 = current
        else:
            return current
        if equals(f(current), 0):
            break
        current = (x1 + x2) / 2
    return current
```

Equals 函数用于判断精度是否符合要求

```python
def equals(a, b):
    return Decimal(a).quantize(Decimal('0.000000')) ==\
    Decimal(b).quantize(Decimal('0.000000'))
```

b. 牛顿法

牛顿法是一种迭代算法，当结果精度满足时可以完成迭代

迭代公式：

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}$$

在设计算法的时候，需要自己输入 fx 的导函数 f_x

```python
'''
# 牛顿法
# @params {f} 原函数
# @params {f_} 原函数的导数
# @params {x} 初始值
'''
def newton(f, f_, x):
    while(True):
        x1 = x - f(x) / f_(x)
        if equals(x1, x):
            x = x1
            break
        else:
            x = x1
    return x
```

c. 简化牛顿法

和牛顿法类似，不过为了方便计算，将 f_x 的值固定为初始点的值

```python
'''
# 简化牛顿法
# @params {f} 原函数
# @params {f_x} 原函数在起始点的梯度
# @params {x} 起始点
'''
def simple_newton(f, f_x, x):
    while(True):
        x1 = x - f(x) / f_x
        if equals(x1, x):
            x = x1
            break
        else:
            x = x1
    return x
```

同样需要自己输入 f_x

d. 弦截法

可以看作是牛顿法的变种，用点斜式取代牛顿法中的 f_x，同样有简化计算的效果

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

```
'''
# 弦截法
# @params {f} 原函数
# @params {x0} 近似根1
# @params {x1} 近似根2
'''
def secant(f, x0, x1):
    while(True):
        x_new = x1 - f(x1) / (f(x1) - f(x0)) * (x1 - x0)
        if equals(x_new, x1):
            x1 = x_new
            break
        else:
            x0 = x1
            x1 = x_new
    return x1
```

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

由于 Ax=b 中，A 不是一个方阵，因此用消元法和迭代法都不能有效的解出它的结果。
利用公式：
ATAx=ATb，将原方程转化为可解的普通线性方程，再利用最小二乘法：

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

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

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

$$(k = 1,2,\cdots,n)$$

随机产生初始权值向量 x^0=rand(n, 1)，设 $P^0=a*I\in R^{n*n}$（a 是足够大的正数，一般取 a = 10^6~10^{10}），$I\in R^{n*n}$是单位矩阵)。

迭代解出 x 的值
根据题目要求，要先生成一个矩阵 A，向量 b，单位阵 I，以及初始化 P

```python
def generate():
    A = []
    b = []
    I = []
    p = []
    mu = 0
    sigma = 5
    s = np.random.normal(mu, sigma, 120000)
    for i in range(10000):
        A.append([])
        for j in range(10):
            A[-1].append(s[random.randint(0, 119999)])
        b.append([s[random.randint(0, 119999)]])
    for i in range(10):
        I.append([])
        p.append([])
        for j in range(10):
            if i == j:
                I[-1].append(1)
                p[-1].append(10000)
            else:
                I[-1].append(0)
                p[-1].append(0)
    return [A, b, I, p]
```

另外要算相对误差，就必须算出标准解

这里利用了 scipy 库的 solve 方法，解出 ATAx=ATb 的值

```python
x_s = solve(array(A.T * A), array(A.T * b))
```

然后设计最小二乘算法

```python
def least_square(A, b, I, p, x_s):
    x0 = []
    for i in range(10):
        x0.append([random.randint(1,10)])
    x0 = mat(x0)
    x = [x0]
    error = []
    steps = []
    for i in range(10000):
        q = p * A[i].T / (1 + A[i] * p * A[i].T)
        x.append(x[-1] + q * (b[i,0] - A[i] * x[-1]))
        p = (I - q * A[i]) * p
        error.append(norm(x[-1], x_s))
        if len(steps) > 0:
            steps.append(steps[-1] + 1)
        else:
            steps.append(1)
    print(error[1000])
    plt.figure()
    plt.plot(steps[:100], error[:100])
    plt.savefig('../assets/least_square.png')
```

X_s 是标准解，误差用 x-x_s 的二范数表示

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

算法设计如下：

228：1，它比一般 FFT 的计算量（pN 次乘法）也快一倍. ……为改进的

FFT 算法，下面给出这一算法的程序步骤：

步骤 1 给出数组 $A_1(N)$，$A_2(N)$ 及 $\omega(N/2)$.

步骤 2 将已知的记录复数数组 $\{x_k\}$ 输入到单元 $A_1(k)$ 中（k 从 0 到 $N-1$）.

步骤 3 计算 $\omega^m=\exp\left(-\mathrm{i}\frac{2\pi}{N}m\right)$ $\left(或\ \omega^m=\exp\left(\mathrm{i}\frac{2\pi}{N}m\right)\right)$ 存放在单元 $\omega(m)$ 中（m 从 0 到 $(N/2)-1$）.

步骤 4 q 循环从 1 到 p，若 q 为奇数做步骤 5，否则做步骤 6.

步骤 5 k 循环从 0 到 $2^{p-q}-1$，j 循环从 0 到 $2^{q-1}-1$，计算

$$A_2(k2^q+j)=A_1(k2^{q-1}+j)+A_1(k2^{q-1}+j+2^{p-1}),$$

$$A_2(k2^q+j+2^{q-1})=[A_1(k2^{q-1}+j)-A_1(k2^{q-1}+j+2^{p-1})]\omega(k2^{q-1}).$$

转步骤 7.

步骤 6 k 循环从 0 到 $2^{p-q}-1$，j 循环从 0 到 $2^{q-1}-1$，计算

$$A_1(k2^q+j)=A_2(k2^{q-1}+j)+A_2(k2^{q-1}+j+2^{p-1}),$$

$$A_1(k2^q+j+2^{q-1})=[A_2(k2^{q-1}+j)-A_2(k2^{q-1}+j+2^{p-1})]\omega(k2^{q-1}).$$

k,j 循环结束，做下一步.

步骤 7 若 $q=p$ 转步骤 8，否则 $q+1\rightarrow q$ 转步骤 4.

步骤 8 q 循环结束，若 $p=$ 偶数，将 $A_1(j)\rightarrow A_2(j)$，则 $c_j=A_2(j)(j=0,1,\cdots,N-1)$ 为所求.

例 13 设 $f(x)=x^4-3x^3+2x^2-\tan x(x-2)$. 给定数据 $\{x_j,f(x_j)\}_{j=0}^{7}$，$x_j=\frac{j}{\ }$ 确定

源码:

生成正弦波:

```
x = []
tx = []
w=[]
w[1] = rand(1);
w[2]= rand(1);
w[3] = rand(1);
w[4] = rand(1);
w[5] = rand(1);
count = 1;
for i = 1: 2^10
    t = i + 2* pi / 1024;
    x(count) = sin(w[1] * t) + sin(w[2] * t) +sin(w[3]* t) + sin(w[4] * t) + sin(w[5]
    tx(count) = t;
    count = count + 1;
end
```

用了 5 段正弦波形叠加

初始化 A1 和 w

```
for k = 0:N - 1
    A1(k + 1) = x(k + 1)
end
for m = 0:(N/2 - 1)
    w(m + 1) = exp(-1i * 2 * pi * m / N);
end
```

计算 A2

```
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^
                A2(k * 2^q + j + 2^(q-1) + 1) = [A1(k * 2^(q-1) + j + 1) - A1(k * 2^(
            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^
                A1(k * 2^q + j + 2^(q-1) + 1) = [A2(k * 2^(q-1) + j + 1) - A2(k * 2^(
            end
        end
    end
end
if mod(p, 2) == 0
    for j = 1:N
        A2(j) = A1(j);
    end
end
```

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

梯形公式:

$$T_n = \frac{b-a}{2n}[f(a)+2\sum_{i=0}^{n-1} f(x_i)+f(b)]$$

辛普森公式:

$$S_n = \frac{b-a}{6n}[f(a)+4\sum_{i=0}^{n-1} f(x_{i+\frac{1}{2}})+2\sum_{i=1}^{n-1} f(x_i)+f(b)]$$

其中

$$x_{k+1/2}=x_k+\frac{1}{2}h$$

```
'''
# 梯形法
# @params {f} 原函数
# @params {from_} 起点
# @params {to_} 终点
# @params {n} 份数
'''
def trapezoid(f, from_, to_, n):
    h = (to_ - from_) / n
    sum_ = 0
    for i in range(1, n):
        sum_ += f(from_ + i * h)
    return h/2 * (f(from_) + sum_ * 2 + f(to_))
```

```python
'''
# 辛普森
# @params {f} 原函数
# @params {from_} 起点
# @params {to_} 终点
# @params {n} 份数
'''
def simpson(f, from_, to_, n):
    h = (to_ - from_) / n
    sum_1 = 0
    sum_2 = 0
    for i in range(n):
        sum_1 += f(from_ + i * h + 0.5 * h)
    for i in range(1,n):
        sum_2 += f(from_ + i * h)
    return h / 6 * (f(from_) + 4 * sum_1 + sum_2 * 2 + f(to_))
```

测试:

```python
if __name__ == "__main__":
    print('梯形法: ')
    print('n=5: ', end=' ')
    print(trapezoid(f, 0, 1, 5))
    print('n=9: ', end=' ')
    print(trapezoid(f, 0, 1, 9))
    print('n=17: ', end=' ')
    print(trapezoid(f, 0, 1, 17))
    print('n=33: ', end=' ')
    print(trapezoid(f, 0, 1, 33))
    print('------\n辛普森: ')
    print('n=5: ', end=' ')
    print(simpson(f, 0, 1, 5))
    print('n=9: ', end=' ')
    print(simpson(f, 0, 1, 9))
    print('n=17: ', end=' ')
    print(simpson(f, 0, 1, 17))
    print('n=33: ', end=' ')
    print(simpson(f, 0, 1, 33))
    print('-----')
    print('标准解: ', end=' ')
    x = Symbol('x')
    print('%f' % integrate(sin(x)/x, (x, 0, 1)))
```

6. **请采用下述方法，求解常微分方程初值问题 y'=y-2x/y，y(0)=1，计算区间为 [0, 1]，步长为 0.1。**
   (1) 前向欧拉法。
   (2) 后向欧拉法。
   (3) 梯形方法。
   (4) 改进欧拉方法。

a. 前向欧拉法

公式:

$$y(x_{k+1}) \approx y(x_k) + hf(x_k, y(x_k))$$

算法设计:

```python
'''
# 前进欧拉
# @params {f} 微分函数
# @params {x0} 初始点
# @params {y0} 初始值
'''
def forward_euler(f, x0, y0):
    x = []
    y = []
    x.append(x0)
    y.append(y0)
    for i in range(10):
        x_new = x[-1] + 0.1
        y_new = y[-1] + 0.1 * f(x[-1], y[-1])
        x.append(x_new)
        y.append(y_new)
    return [x, y]
```

b. 后向欧拉法:

公式

$$y(x_{k+1}) \approx y(x_k) + hf(x_{k+1}, y(x_{k+1}))$$

$$y_{k+1} = y_k + hf(x_{k+1}, y_{k+1}) \quad k = 0,1,2....n-1$$

后向欧拉法是隐式迭代公式，要算出 yk+1 可以用这样的迭代方式:

$$\begin{cases} y_{n+1}^{0} = y_n + h\, f(x_n, y_n) \\ y_{n+1}^{(k+1)} = y_n + h\, f(x_{n+1}, y_{n+1}^{(k)}) \end{cases}$$

具体设计:

```python
'''
# 后退欧拉
# @params {f} 微分函数
# @params {x0} 初始点
# @params {y0} 初始值
'''
def backward_euler(f, x0, y0):
    x = [x0]
    y = [y0]
    for i in range(10):
        y_new = iteration(f, x[-1], y[-1])
        x_new = x[-1] + 0.1
        x.append(x_new)
        y.append(y_new)
    return [x, y]

def iteration(f, x0, y0):
    y_new = y0 + 0.1 * f(x0, y0)
    x_new = x0 + 0.1
    for i in range(50):
        y_new = y0 + 0.1 * f(x_new, y_new)
    return y_new
```

后面的 iteration 用来迭代 yk+1

c. 梯形方法

跟后向欧拉方法类似，是隐式迭代方法

公式

$$y(x_{k+1}) - y(x_k) = \int_{x_k}^{x_{k+1}} f(x, y(x))dx$$

即：

$$y_{k+1} \approx y_k + \frac{h}{2}[f(x_k, y_k) + f(x_{k+1}, y_{k+1})] \quad k = 0,1,2....n-1$$

递推公式为：

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

具体实现：

```python
def trapezoid(f, x0, y0):
    x = [x0]
    y = [y0]
    for i in range(10):
        y_new = y[-1] + 0.1 * f(x0, y0)
        x_new = x[-1] + 0.1
        for j in range(50):
            y_new = y[-1] + 0.05 * (f(x[-1], y[-1]) + f(x_new, y_new))
        x.append(x_new)
        y.append(y_new)
    return [x, y]
```

进行 50 次迭代逼近 yk+1

d. 改进欧拉

$$\begin{cases} y_p = y_k + hf(x_k, y_k) \\ y_c = y_k + hf(x_{k+1}, y_p) \\ y_{k+1} = \dfrac{1}{2}(y_p + y_c) \end{cases}$$

先用欧拉方法求出近似值，然后用梯形方法校正

不需要隐式迭代，比梯形方法的效率高

```python
'''
# 改进欧拉
# @params {f} 微分函数
# @params {x0} 初始点
# @params {y0} 初始值
'''
def improve_euler(f, x0, y0):
    x = [x0]
    y = [y0]
    for i in range(10):
        x_new = x[-1] + 0.1
        yp = y[-1] + 0.1 * f(x[-1], y[-1])
        yc = y[-1] + 0.1 * f(x_new, yp)
        y_new = 0.5 * (yp + yc)
        x.append(x_new)
        y.append(y_new)
    return [x, y]
```

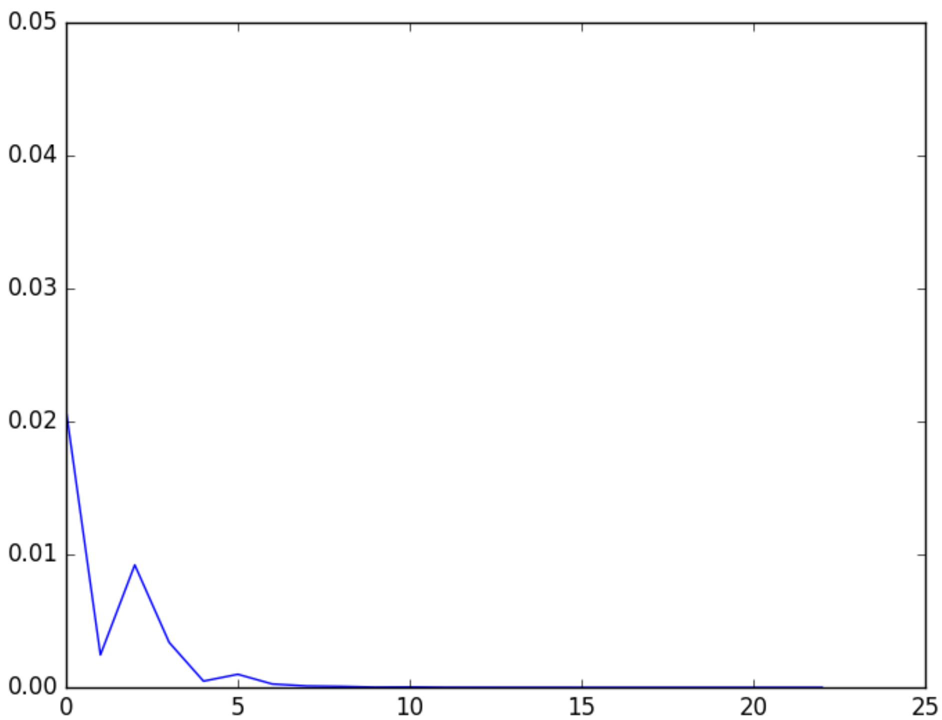
数值实验

1. 三种插值

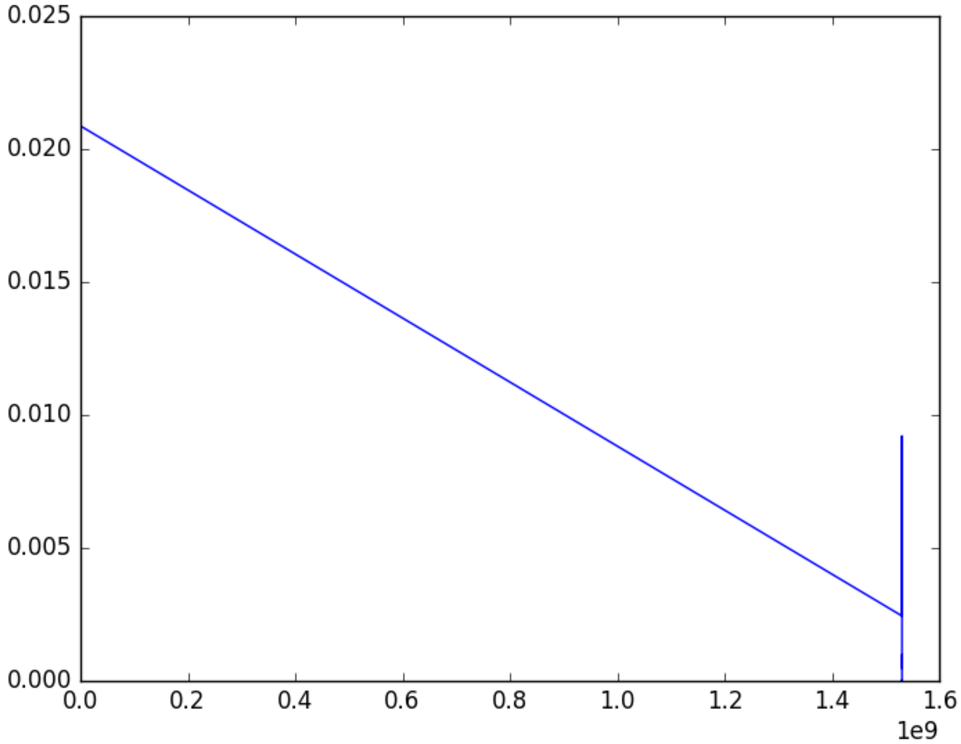
```
一次插值：  0.3428805
二次插值：  0.34289712499999997
三次插值：  0.34289762499999993
结果：  0.34289780745545134
```

2. 二分法

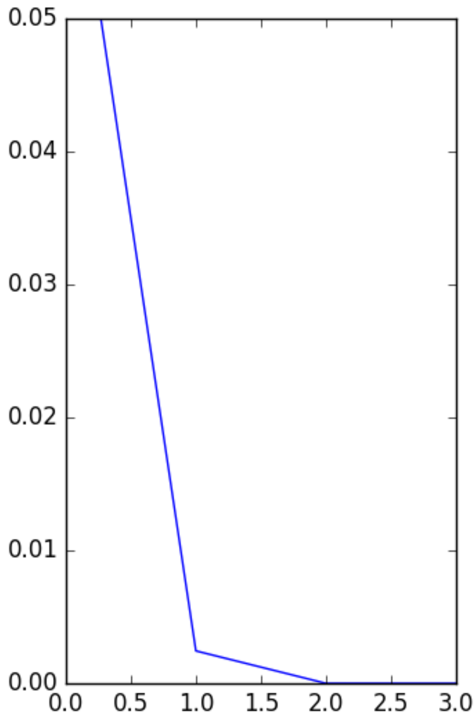
收敛速度较慢



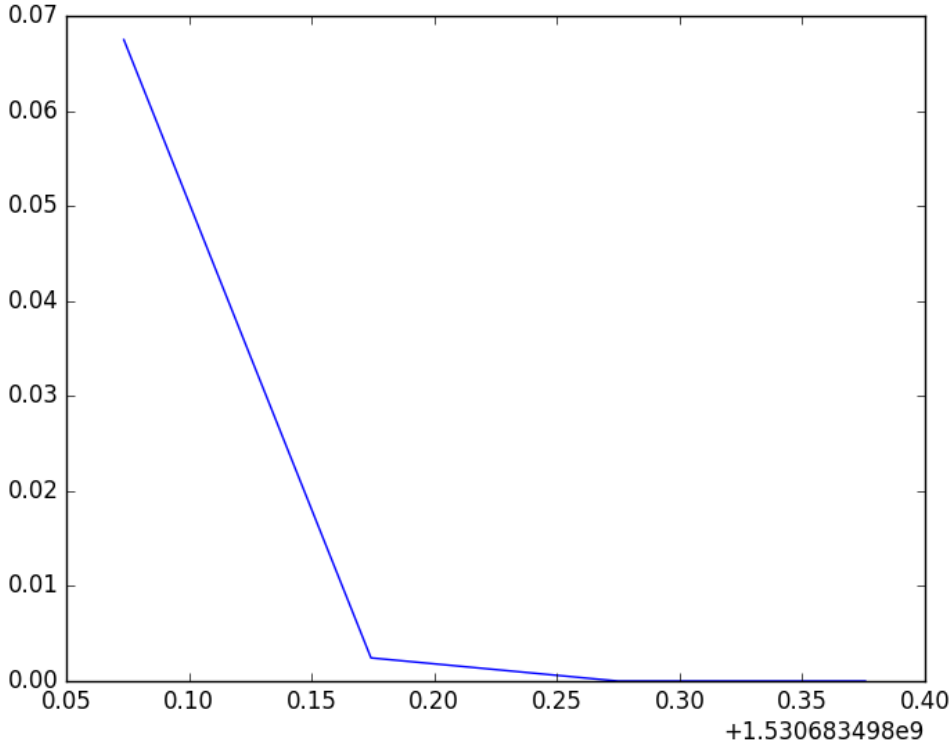
Time



3. 牛顿法
   收敛速度非常快

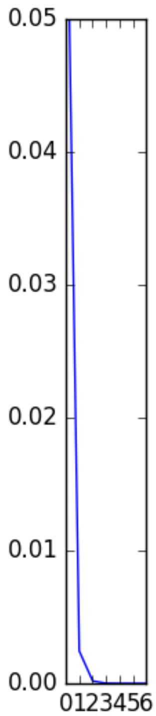


Time



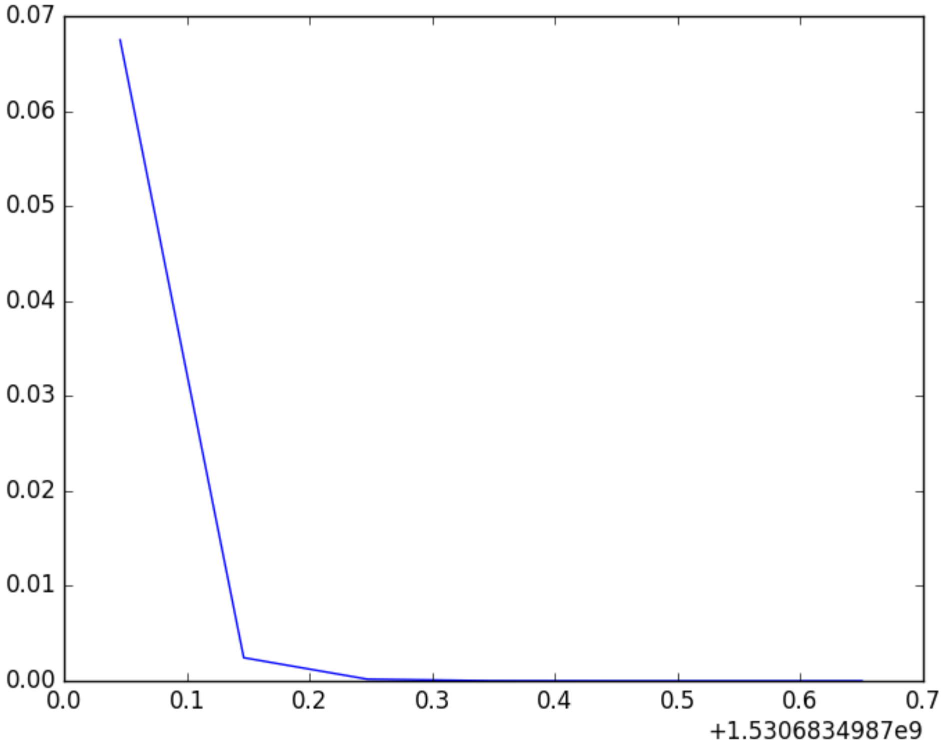
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
+1.530683498e9

4. 简单牛顿



0.05
0.04
0.03
0.02
0.01
0.00
0123456

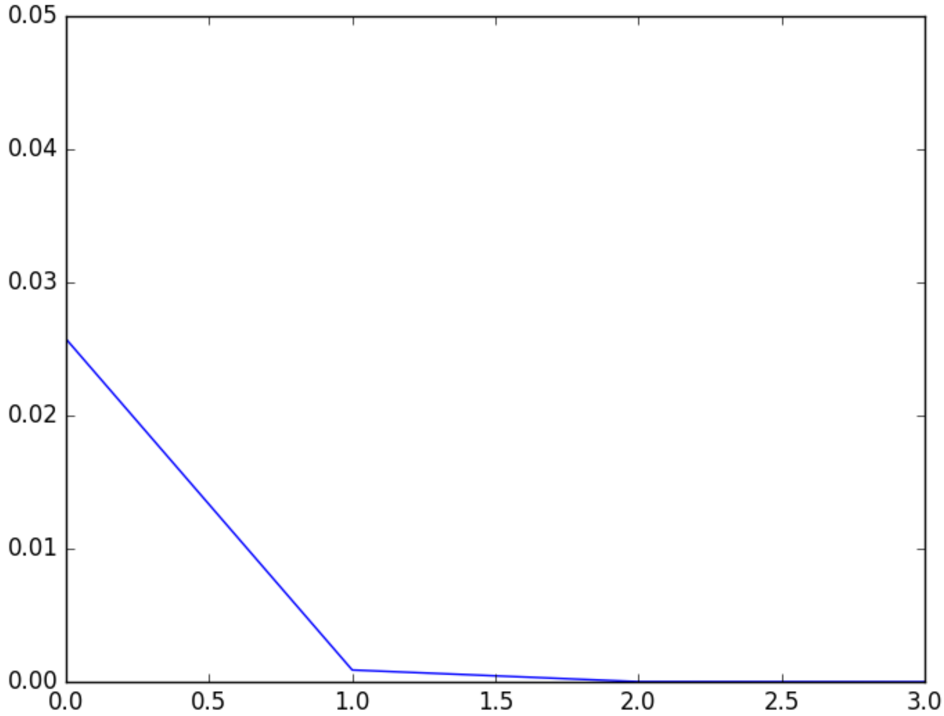
Time



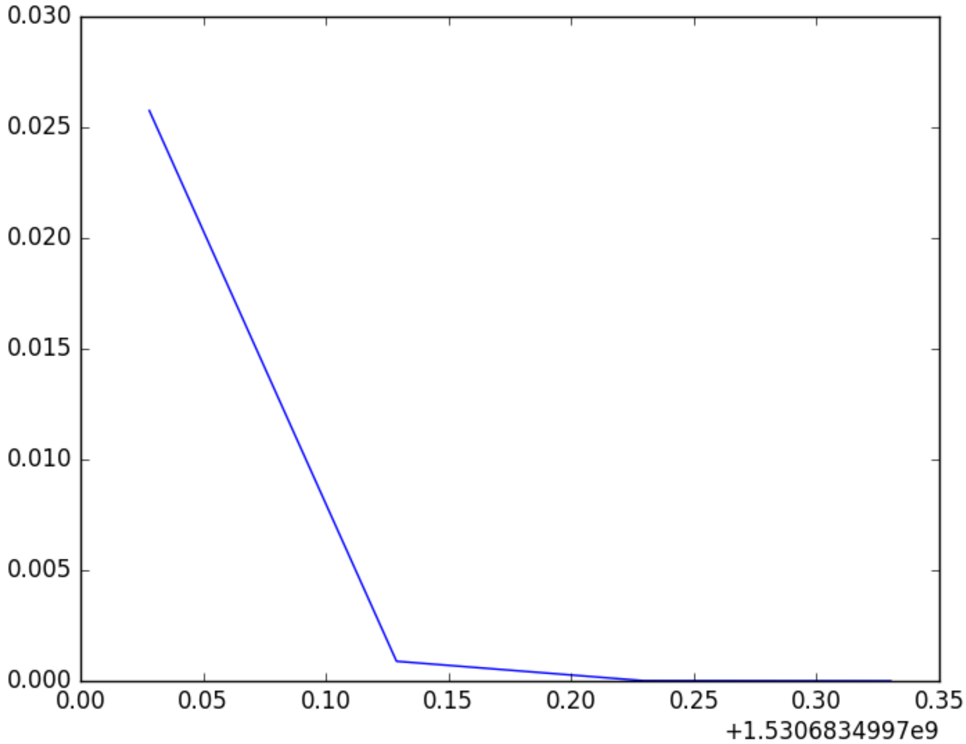
收敛速度比牛顿稍慢

5. 弦截法

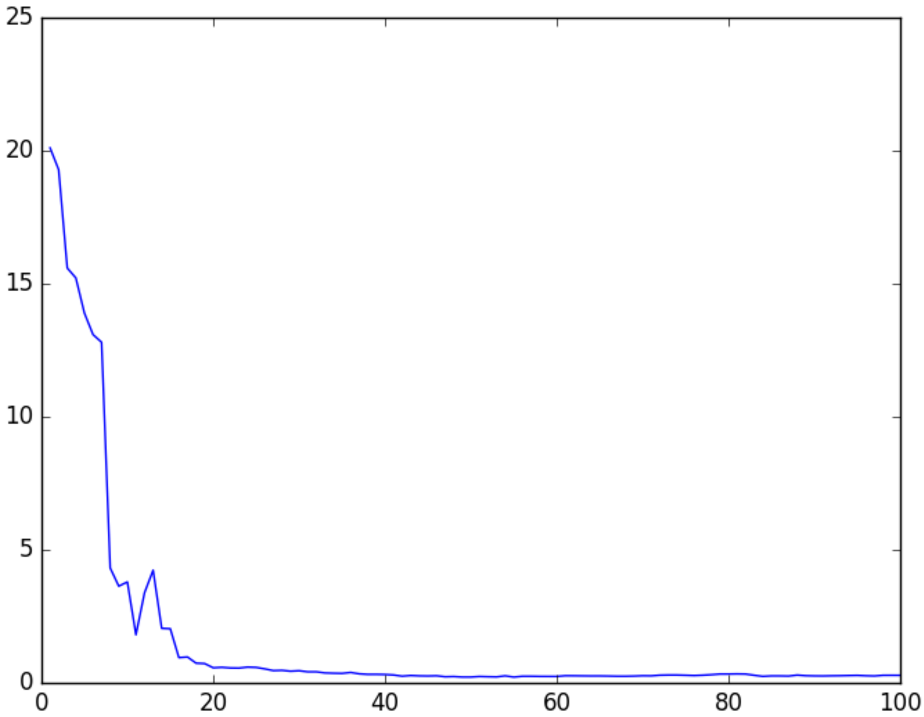
速度仅次于牛顿法



Time



6. RLS 解超定线性方程


25
20
15
10
5
0
0
20
40
60
80
100

7. 快速傅里叶变换

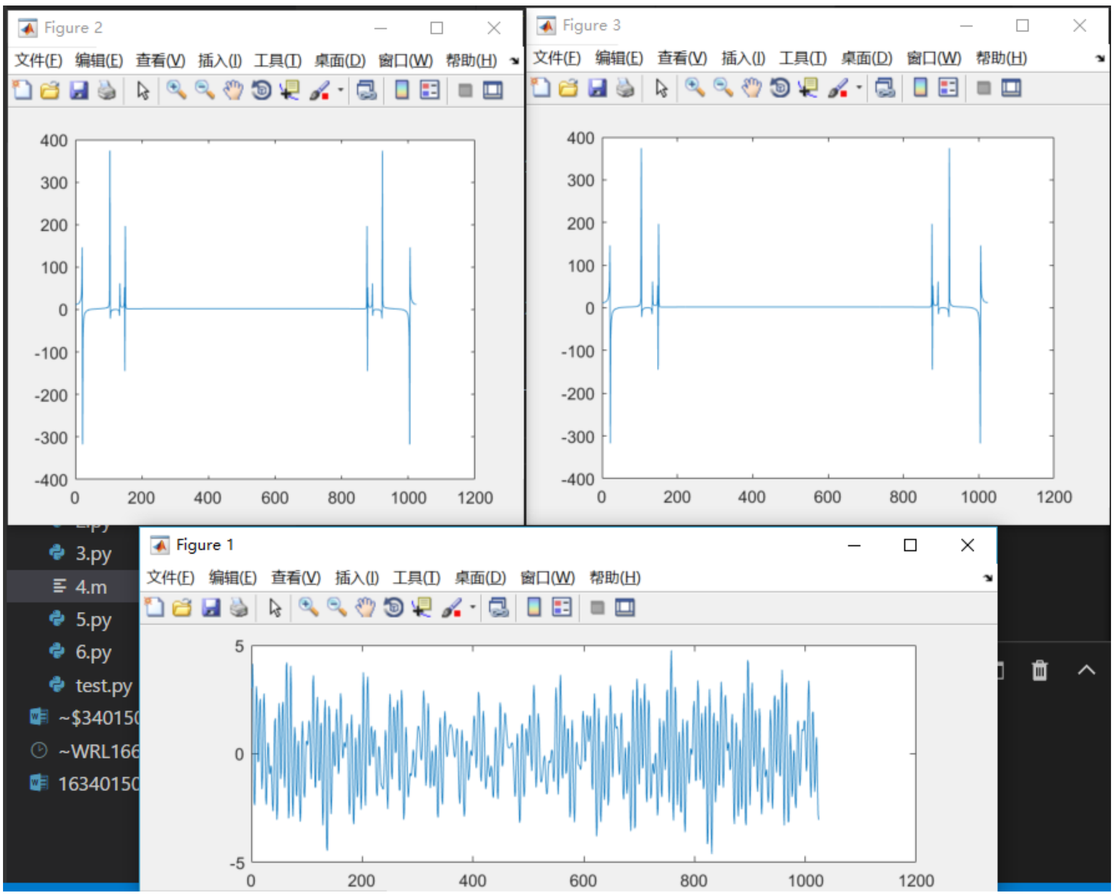


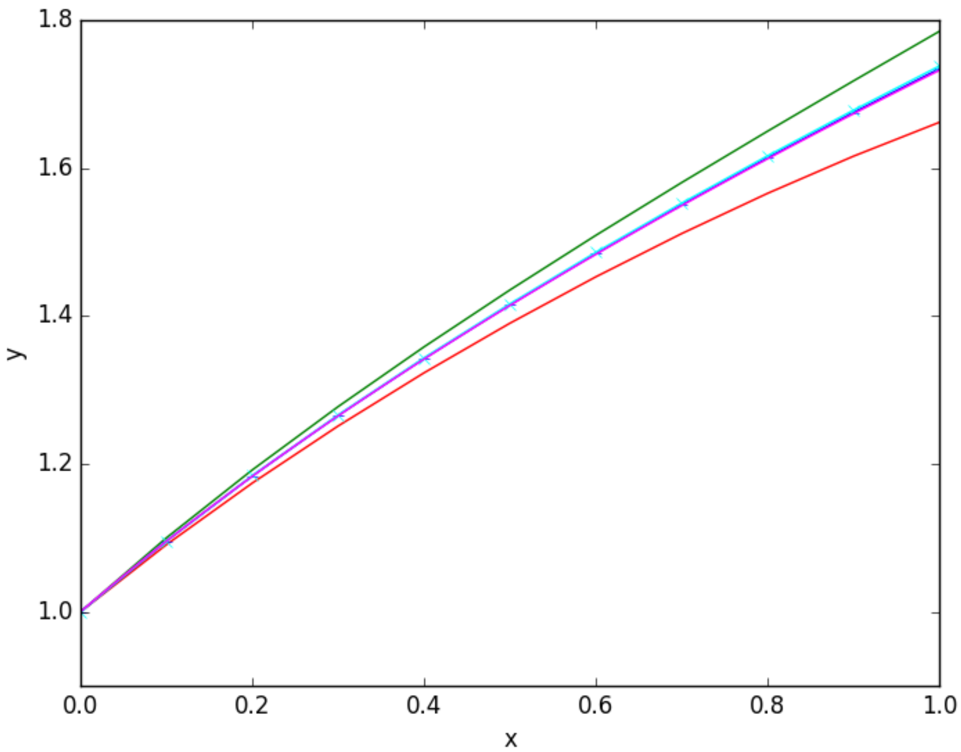
Figure 2
文件(F) 编辑(E) 查看(V) 插入(I) 工具(T) 桌面(D) 窗口(W) 帮助(H)
Figure 3
文件(F) 编辑(E) 查看(V) 插入(I) 工具(T) 桌面(D) 窗口(W) 帮助(H)
3.py
4.m
5.py
6.py
test.py
Figure 1
文件(F) 编辑(E) 查看(V) 插入(I) 工具(T) 桌面(D) 窗口(W) 帮助(H)

左上图为 fft，右上图为内置 fft，下图为正弦波

8. 梯形公式和辛普森公式

```
梯形法：
n=5:  0.9450787809534019
n=9:  0.945773188549752
n=17:  0.9459962252423758
n=33:  0.9460600238880433
------
辛普森：
n=5:  0.9460831688380728
n=9:  0.9460830797420532
n=17:  0.9460830711034892
n=33:  0.9460830704190363
-----
标准解：  0.946083
```

9. 解常微分方程



绿色是向前欧拉，红色是向后欧拉。蓝色线是梯形公式，浅蓝色线是改进欧拉，紫色线是标准解。

结果分析

1. 三种插值法的结果分析
   显然，三次插值的结果是离标准解最近的，二次插值次之。三种插值都离标准解较近，说明拉格朗日插值方法的正确

2. 四种迭代算法的误差分析
   二分法的迭代步数是四种方法里面最多的，进行到 20+步时才收敛到精确值附近
   牛顿法的收敛速度很快，两步之内收敛到标准解
   简化牛顿法的收敛速度比牛顿法稍微慢一点，但计算上简单，而且，只要 6 步就收敛
   弦截法速度是四种方法里第二快的算法，要 4 步收敛，而且计算上并不复杂（不需要求导），因此也是一种优秀的迭代方法。
   在时间图上，由于后三种方法收敛速度较快，在时间上几乎体现不出区别。(通过 sleep 函数提高耗时画图)
3. 由上图可得，最小二乘法的收敛速度并不是十分优秀，在 100+步之后仍然没有达到很高的精度。
4. 上图为测试结果，可以看出混杂若干不同频率正弦的信号在经过傅里叶变换后，可以明显看出 ck 的值。并且在比较内置 fft 算法后发现差别不大（误差在 E-15）
5. 梯形公式和辛普森公式。对于两个公式而言，取的点越多越逼近准确值。而比较两种算法可以发现，在取相同点的时候，辛普森算法更加接近真实值，比梯形公式要更加优秀
6. 常微分方程求解。由图可以发现，前向和后向欧拉方法的误差会较大。而改进欧拉方法和梯形方法与真实值的差别较小。