The experimental report of Numerical Analysis

Jiandong Liu

School of Computer Science and Technology

Shandong University

Supervisor

Professor Baodong Liu

The Elite Class of 2017, 201700130011

December 10, 2019

Abstract

There are 8 chapters in this experimental report which contains several classic numerical algorithms and some important method of numerical analysis. The algorithms or numerical methods are described below.

- Error Theory
- Fixed-Point, Newton's and Secant Method.
- LU, LL^T, LDL^T factorization.
- The method of Interpolation.
- Different Numerical Difference Format.
- Euler's, Trapezoidal and Runge Kuta Method.
- The General Form of Discrete Least Square Rule.
- Power, Inverse Power and Symmetric Power Method.

Contents

Chapter 1

Error Theory

1.1 Experimental Requirements

1) Master the basic error theory including the source of error and the types of error.

2) Comprehend several error types such as model error, truncation error, rounding error and floating-point arithmetic rounding error.

3) Understand the measurements of errors, relative and absolute errors.

4) Comprehend the convergence of the iterative sequence and the convergence order of the error.

5) Master the concepts of error propagation, error accumulation, local error, and overall error.

1.2 Homework

1.2.1 Problem 1

Problem Description

Complete the following calculations:

$$\int_0^{\frac{1}{4}} e^{x^2} dx \approx \int_0^{\frac{1}{4}} (1 + x^2 + \frac{x^4}{2!} + \frac{x^6}{3!}) dx = \hat{p}$$

Figure out what type of error will occur in this calculation process and compare the result with the answer p which equals to 0.2553074606.

Knowledge Points Involved

There are four common types of error, model error, truncation error, rounding error and floating-point arithmetic rounding error.

Solution

The calculation process of the above integration formula is as follows.

$$\hat{p} = (x + \frac{x^3}{3} + \frac{x^5}{10} + \frac{x^7}{42})\Big|_0^{\frac{1}{4}} \approx 0.2553074428$$
$$\Delta_p = |p - \hat{p}| = 0.0000000178$$
$$\delta_p = \frac{|p - \hat{p}|}{|p|} \approx 0.0000000697 < 5 * 10^{-7}$$

Thus, $\hat{p}$ approximate p to 7 significant digits. Then we analysis the type of error involved in the calculation process above.

Firstly, we use the Taylor Theorem to transform this problem, so that there exists truncation error. What's more, the floating point is involved in the calculation process above. Therefore, there also exists rounding error and floating-point arithmetic rounding error.

1.2.2 Problem 2

Problem Description

Sometimes using triangles or algebraic identities to rearrange the terms in a function could avoid loss of precision. So the task is finding equivalent formulas for the following functions to avoid loss of precision.

$$(a)\ ln(x+1) - ln(x), x\ is\ large.$$
$$(b)\ \sqrt{x^2+1} - x, x\ is\ large.$$
$$(c)\ cos^2(x) - sin^2(x), x \approx \frac{\pi}{4}.$$
$$(d)\ \sqrt{\frac{1+cos(x)}{2}}, x \approx \pi.$$

Solution

This question requires us to merge the items in the function to avoid loss of precision and the solution is shown below.

$$(a)\ ln(x+1) - ln(x) = ln(\frac{x+1}{x}) = ln(1+\frac{1}{x})$$

$$(b)\ \sqrt{x^2+1} - x = \frac{(\sqrt{x^2+1}-x) * (\sqrt{x^2+1}+x)}{\sqrt{x^2+1}+x} = \frac{1}{\sqrt{x^2+1}+x}$$

$$(c)\ cos^2(x) - sin^2(x) = cos(2x)$$

$$(d)\ \sqrt{\frac{1+cos(x)}{2}} = \sqrt{\frac{1+2cos^2(\frac{x}{2})-1}{2}} = |cos(\frac{x}{2})|$$

1.2.3 Problem 3

Problem Description

Discuss the spread of error in the following calculation processes.

(a) The sum of three numbers:

$$p + q + r = (\hat{p} + \epsilon_p) + (\hat{q} + \epsilon_q) + (\hat{r} + \epsilon_r)$$

(b) The quotient of two numbers:

$$\frac{p}{q} = \frac{\hat{p} + \epsilon_p}{\hat{q} + \epsilon_q}$$

(c) The product of three numbers:

$$pqr = (\hat{p} + \epsilon_p)(\hat{q} + \epsilon_q)(\hat{r} + \epsilon_r)$$

Problem Analysis

The initial error is usually propagated through a series of calculations. For any numerical calculation, it is necessary to minimize the initial error mainly for the reason that small errors under the initial conditions have less impact on the final result in the respect of stable algorithm.

Therefore, in this problem, we are required to calculate the relative and absolute error and analyse the final result in the three numerical basis operations.

Theorem of Error Spread

Let ϵ denotes the initial error and $\epsilon(n)$ denotes the error after n steps. If $|\epsilon(n)| \approx n\epsilon$, the error growth is linear. If $|\epsilon(n)| \approx K^n\epsilon$, the error growth is exponential. If $K > 1$, $n \to \infty$, the error will be infinite, while $0 < K < 1$, the error will approximate to zero.

Solution

(a) The sum of three numbers:

$$\Delta = |\epsilon_p + \epsilon_q + \epsilon_r|$$
$$\delta = \frac{|\Delta|}{|p+q+r|} = |\frac{\epsilon_p + \epsilon_q + \epsilon_r}{p+q+r}|$$

According to the calculation result above, we can figure out the error spread is linear in addition.

(b) The quotient of two numbers:

$$\Delta = |\frac{\hat{p}+\epsilon_p}{\hat{q}+\epsilon_q} - \frac{\hat{p}}{\hat{q}}| = |\frac{\hat{q}\epsilon_p - \hat{p}\epsilon_q}{\hat{q}(\hat{q}+\epsilon_q)}|$$
$$\delta = \frac{|\Delta||q|}{|p|} = |\frac{\hat{q}\epsilon_p - \hat{p}\epsilon_q}{\hat{q}(\hat{p}+\epsilon_p)}|$$

According to the calculation result above, we can figure out the error spread is not linear or exponential in division.

(c) The product of three numbers:

$$\Delta = |pqr - \hat{p}\hat{q}\hat{r}| = |(\hat{p}+\epsilon_p)(\hat{q}+\epsilon_q)(\hat{r}+\epsilon_r) - \hat{p}\hat{q}\hat{r}|$$
$$\delta = \frac{|\Delta|}{|pqr|} = |1 - \frac{\hat{p}\hat{q}\hat{r}}{(\hat{p}+\epsilon_p)(\hat{q}+\epsilon_q)(\hat{r}+\epsilon_r)}|$$

According to the calculation result above, we can find the term $\hat{p}\hat{q}\hat{r}$ in Δ. Therefore, the error spread is exponential in multiplication.

1.2.4 Problem 4

Problem Description

According to the Taylor Theorem, the two functions below hold.

$$cos(h) = 1 - \frac{h^2}{2!} + \frac{h^4}{4!} + O(h^6)$$
$$sin(h) = h - \frac{h^3}{3!} + \frac{h^5}{5!} + O(h^7)$$

Determine the approximate order of their sum and product.

Related Theorem

There are functions $f(h)$ and $g(h)$. If there are constants C and c such that for any $h \leq c$,

$$|f(h)| \leq C|g(h)|$$

holds, then the function $f(h)$ is called a big Oh function of $g(h)$, expressed as $f(h) = O(g(h))$.

Solution

For addition,

$$\begin{aligned} cos(h) + sin(h) &= 1 - \frac{h^2}{2!} + \frac{h^4}{4!} + O(h^6) + h - \frac{h^3}{3!} + \frac{h^5}{5!} + O(h^7) \\ &= 1 + h - \frac{h^2}{2!} - \frac{h^3}{3!} + \frac{h^4}{4!} + \frac{h^5}{5!} + O(h^6) + O(h^7) \end{aligned}$$

holds. What's more, $O(h^6) + O(h^7) = O(h^6)$ holds with $|h| \leq 1$. Therefore, the formula above could be simplified to

$$cos(h) + sin(h) = 1 + h - \frac{h^2}{2!} - \frac{h^3}{3!} + \frac{h^4}{4!} + \frac{h^5}{5!} + O(h^6)$$

whose approximate order is $O(h^6)$.

For multiplication,

$$\begin{aligned} cos(h) * sin(h) &= (1 - \frac{h^2}{2!} + \frac{h^4}{4!} + O(h^6)) * (h - \frac{h^3}{3!} + \frac{h^5}{5!} + O(h^7)) \\ &= h - \frac{h^3}{2!} - \frac{h^3}{3!} + \frac{h^5}{4!} + \frac{h^5}{5!} + \frac{h^5}{2!3!} - \frac{h^7}{3!4!} - \frac{h^7}{2!5!} + \frac{h^9}{4!5!} + O(h^7) \end{aligned}$$

holds. What's more, $-\frac{h^7}{3!4!} - \frac{h^7}{2!5!} + \frac{h^9}{4!5!} + O(h^7) = O(h^7)$ holds with $|h| \leq 1$.

Therefore, the formula above could be simplified to

$$cos(h) * sin(h) = h - \frac{h^3}{2!} - \frac{h^3}{3!} + \frac{h^5}{4!} + \frac{h^5}{5!} + \frac{h^5}{2!3!} + O(h^7)$$

whose approximate order is $O(h^7)$.

1.2.5 Problem 5

Problem Description

Calculate the root of the equation $x^2 + (\alpha + \beta)x + 10^9 = 0$ with $\alpha = -10^9$ and $\beta = -1$. Discuss how to design calculation formats to effectively reduce errors and improve calculation accuracy.

Solution

With $\alpha = -10^9$ and $\beta = -1$, we could change the original equation to

$$x^2 + (\alpha + \beta)x + 10^9 = x^2 + (\alpha - 1)x - \alpha.$$

According to the Finding-Root formula in quadratic function with one variable,

$$\begin{aligned} x &= \frac{-(\alpha - 1) \pm \sqrt{(\alpha - 1)^2 + 4\alpha}}{2} \\ &= \frac{-(\alpha - 1) \pm \sqrt{(\alpha + 1)^2}}{2} \\ &= \frac{-(\alpha - 1) \pm (\alpha + 1)}{2} \end{aligned}$$

holds. Thus, the answer of this problem is $x_1 = 1, x_2 = -\alpha$.

1.2.6 Problem 6

Problem Description

Calculate the answer of x^31 and discuss how to design calculation formats to reduce the number of calculations.

Solution

The common method of calculating x^31 is computing 31 times in a loop. But with fast power method, we can change $O(31)$ to $O(log(31))$ with the calculation

format below.

$$x^{31} = x^{1+2+4+8+16} = x * x^2 + x^4 + x^8 + x^{16}$$

According to the format above, we could just use 4 steps to acquire the result of x^2, x^4, x^8, x^{16} and 4 steps to obtain the answer $x^{31} = x * x^2 + x^4 + x^8 + x^{16}$.

1.3 Experimental Assignments

1.3.1 Task 1

Problem Description

According to the formulas below, design algorithm and write code to find the answer of the equation $ax^2 + bx + c = 0$ in all situations including the situation of $|b| \approx \sqrt{b^2 - 4ac}$.

The formulas given for this problem are as follows.

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{1}$$

$$x_1 = \frac{-2c}{b + \sqrt{b^2 - 4ac}}, \quad x_2 = \frac{-2c}{b - \sqrt{b^2 - 4ac}} \tag{2}$$

And the equations that needed to be solved are as follows.

$$(a)\ x^2 - 1000.001x + 1 = 0$$

$$(b)\ x^2 - 10000.0001x + 1 = 0$$

$$(c)\ x^2 - 100000.00001x + 1 = 0$$

$$(d)\ x^2 - 1000000.000001x + 1 = 0$$

Algorithm Design

In (1), a can not be zero. Thus, if $a = 0$, the answer $x = \frac{-c}{b}$. What's more, in (2), c can not be zero. Thus, if $c = 0$, use (2) to calculate the answer.

In addition, in (2), $b + \sqrt{b^2 - 4ac}$ or $b - \sqrt{b^2 - 4ac}$ can not be zero. Therefore, if $b > 0$, use (2) to calculate x_1 and (1) to calculate x_2, otherwise use (1) to calculate x_1 and (2) to calculate x_2.

Pseudo Code

The equation is $ax^2 + bx + c = 0$ with the solution x_1 and x_2.

INPUT: Equation $ax^2 + bx + c$.

OUTPUT: Solution of the equation x_1 and x_2, or message of failure.

Step 1: If $a = 0$, $x_1 = x_2 = \frac{-c}{b}$. STOP.

Step 2: If $b^2 - 4ac < 0$, output "No solution". STOP.

Step 3: If $c = 0$

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}. \quad \text{STOP.}$$

Step 4: If $b > 0$

$$x_1 = \frac{-2c}{b + \sqrt{b^2 - 4ac}}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}. \quad \text{STOP.}$$

Step 5:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-2c}{b - \sqrt{b^2 - 4ac}}. \quad \text{STOP.}$$

Code

```python
import math


def Solution(a, b, c):
    x1 = x2 = 0
    tmp = b * b - 4 * a * c
    if a == 0:
        x1 = x2 = -c / b
    elif tmp < 0:
        print("No solution.\n")
        return
    else:
        tmp = math.sqrt(tmp)
        if c == 0:
            x1 = (-b + tmp) / (2 * a)
            x2 = (-b - tmp) / (2 * a)
        elif b > 0:
            x1 = -2 * c / (b + tmp)
            x2 = (-b - tmp) / (2 * a)
        else:
            x1 = (-b + tmp) / (2 * a)
            x2 = -2 * c / (b - tmp)
    print("x1:", x1, ", x2:", x2)
```

Solution

According the code written above, we can acquire answers of the four equation.

$$(a)\ x_1 = 1000.0, \quad x_2 = 0.001$$
$$(b)\ x_1 = 10000.0, \quad x_2 = 0.0001$$
$$(c)\ x_1 = 100000.0, \quad x_2 = 1e-05$$
$$(d)\ x_1 = 1000000.0, \quad x_2 = 1e-06$$

1.3.2 Task 2

Problem Description

Calculate the first ten numerical approximations to the following three difference equations with a small initial error in each case. If there is not a initial error, each case will generate the sequence $\{1/2^n\}_{n=1}^{\infty}$.

$$(a)\ r_0 = 0.994, \quad r_n = \frac{1}{2}r_{n-1}, \quad n = 1, 2, ...$$
$$(b)\ p_0 = 1, \quad p_1 = 0.497, \quad p_n = \frac{3}{2}p_{n-1} - \frac{1}{2}p_{n-2}, \quad n = 2, 3, ...$$
$$(c)\ q_0 = 1, \quad q_1 = 0.497, \quad q_n = \frac{5}{2}q_{n-1} - q_{n-2}, \quad n = 2, 3, ...$$

Solution

Let $\{x_n\} = \{1/2^n\}$, figure out the error spread with each sequence.

Table 1.1: Numerical Approximations of $\{x_n\}, \{r_n\}, \{p_n\}$ and $\{q_n\}$

n	x_n	r_n	p_n	q_n
0	1.0000000000	0.9940000000	1.0000000000	1.0000000000
1	0.5000000000	0.4970000000	0.4970000000	0.4970000000
2	0.2500000000	0.2485000000	0.2455000000	0.2425000000
3	0.1250000000	0.1242500000	0.1197500000	0.1092500000
4	0.0625000000	0.0621250000	0.0568750000	0.0306250000
5	0.0312500000	0.0310625000	0.0254375000	-0.0326875000
6	0.0156250000	0.0155312500	0.0097187500	-0.1123437500
7	0.0078125000	0.0077656250	0.0018593750	-0.2481718750
8	0.0039062500	0.0038828125	-0.0020703125	-0.5080859375
9	0.0019531250	0.0019414062	-0.0040351562	-1.0220429688
10	0.0009765625	0.0009707031	-0.0050175781	-2.0470214844

Table 1.2: Error Sequence of $\{x_n - r_n\}, \{x_n - p_n\}$ and $\{x_n - q_n\}$

n	$x_n - r_n$	$x_n - p_n$	$x_n - q_n$
0	0.0060000000	0.0000000000	0.0000000000
1	0.0030000000	0.0030000000	0.0030000000
2	0.0015000000	0.0045000000	0.0075000000
3	0.0007500000	0.0052500000	0.0157500000
4	0.0003750000	0.0056250000	0.0318750000
5	0.0001875000	0.0058125000	0.0639375000
6	0.0000937500	0.0059062500	0.1279687500
7	0.0000468750	0.0059531250	0.2559843750
8	0.0000234375	0.0059765625	0.5119921875
9	0.0000117188	0.0059882812	1.0239960938
10	0.0000058594	0.0059941406	2.0479980469

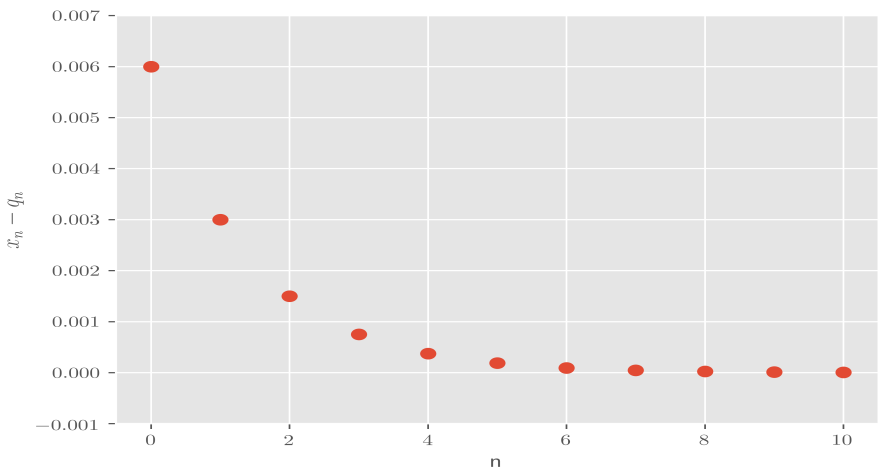

Figure 1.1 Sequence of $\{x_n - r_n\}$

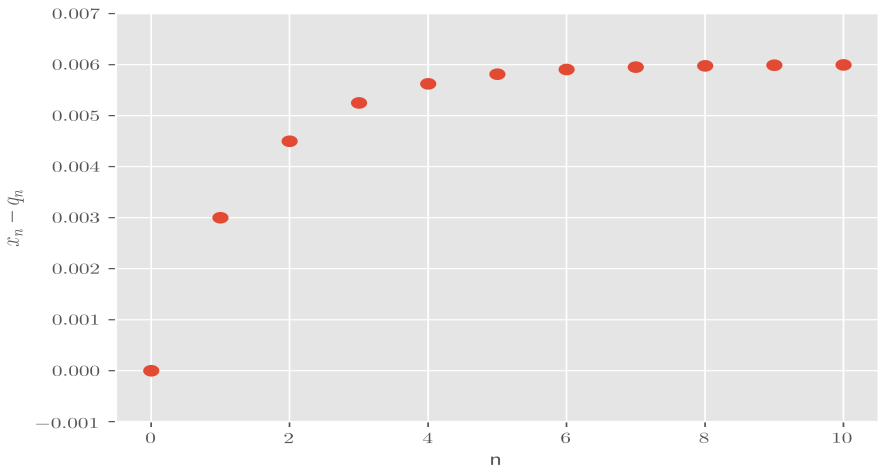

Figure 1.2 Sequence of $\{x_n - p_n\}$

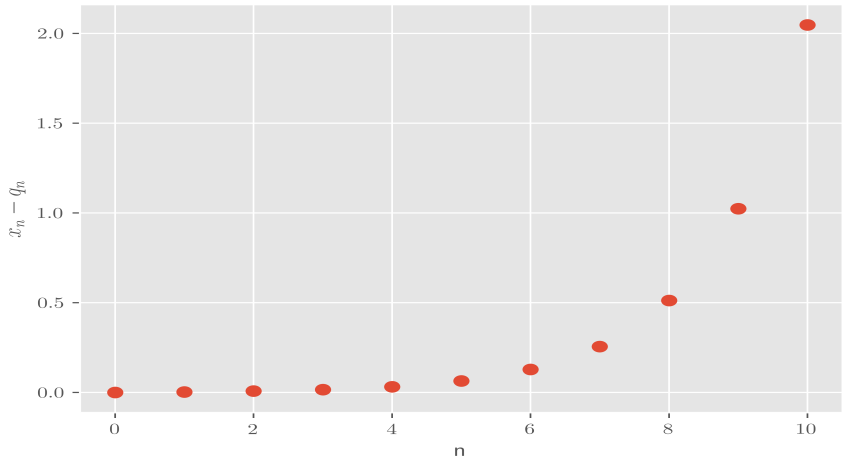

Figure 1.3 Sequence of $\{x_n - q_n\}$

According to the experimental result described above, we could find that $\{r_n\}$ and $\{p_n\}$ is stable while $\{q_n\}$ is not. Furthermore, though $\{p_n\}$ is stable, but with $\lim_{n\to\infty} p_n = 0$, the error dominate the value of $\{p_n\}$. Therefore, the term after p_3 have no significant digits.

Code

```
def Solution():
    x = [0] * 11
    r = [0] * 11
    p = [0] * 11
    q = [0] * 11
    x[1] = 0.5
    r[0] = 0.994
    r[1] = r[0] / 2
    x[0] = p[0] = q[0] = 1
    p[1] = q[1] = 0.497
    for i in range(2, 11):
        x[i] = x[i - 1] / 2.0
        r[i] = r[i - 1] / 2
        p[i] = 3 * p[i - 1] / 2 - p[i - 2] / 2
        q[i] = 5 * q[i - 1] / 2 - q[i - 2]
    for i in range(0, 11):
        print(i, '%.10f' % x[i], '%.10f' % r[i], '%.10f' % p[i], '%.10f' % q[i])
    for i in range(0, 11):
        print(i, '%.10f' % (x[i] - r[i]), '%.10f' % (x[i] - p[i]),
              '%.10f' % (x[i] - q[i]))
```

Chapter 2

Nonlinear Equations

2.1 Experimental Requirements

1) Master the basic concept including the root of equation, fixed point, iteration, convergence, convergence rate and the control of error.

2) Comprehend several basic algorithms and their convergence rate such as bisection method, fixed point method, Newton's method, secant method and false position method.

3) Understand the difference between these algorithms including the convergence rate and the choice of initial value.

2.2 Experimental Assignments

2.2.1 Task 1

Problem Description

Calculate the approximation value of the positive root of the equation $2x^2 + x - 15 = 0$ with $x^* = 2.5$ through the three calculation formats listed below.

(1) $x_{k+1} = 15 - x_k^2, k = 0, 1, 2, ..., initial\ value\ x_0 = 2.$

(2) $x_{k+1} = \dfrac{15}{2x_k + 1}, k = 0, 1, 2, ..., initial\ value\ x_0 = 2.$

(3) $x_{k+1} = x_k - \dfrac{2x_k^2 + x_k - 15}{4x_k + 1}, k = 0, 1, 2, ..., initial\ value\ x_0 = 2.$

Calculate the sequence $\{x_k\}$, draw a graph to observe the stability and convergence of the solution, and analyze the reasons.

Problem Analysis

From the problem description, we could figure out that calculate the sequence of x_k is compute the point of intersection between $f(x) = x$ and $f(x_k)$. What's more, we need to set a TOL so that when $|x_{k+1} - x_k| < TOL$, the iteration ends.

Besides, we should add a little error on the initial value to judge the stability of the equation.

Related Theorem

1) *Stable:* An algorithm is said to be stable imply that small changes in the initial data can produce correspondingly small changes in final results.

2) *Conditionally Stable:* Some algorithm are stable only for certain choices of initial data, this case are called conditionally stable.

3) *Fixed-point Theorem:* Meet the two conditions below, we could find that, for any number $p_0 \in [a, b]$, the sequence $\{p_n\}_0^\infty$ defined by $p_n = g(p_{n-1}), n \geq 1$ converges to the unique fixed point p in $[a, b]$.

- (Existence) If $g(x) \in C[a, b]$ and $g(x) \in [a, b]$ for all $x \in [a, b]$, then $g(x)$ has a fixed point in $[a, b]$. ($g(x) \in [a, b]$ means $max\{g(x)\} \leq b, min\{g(x)\} \geq a$)

- (Uniqueness) If, in addition, $g'(x)$ exists on (a, b), and a positive constant $k < 1$ exists with $|g'(x)| \leq k$, for all $x \in (a, b)$.

4) *Convergence Rate of Fixed-point Iteration:* Since $|p_n - p| \leq k^n * |p_0 - p| \leq k^n * |b - a|$, the Sequence $\{p_n\}_{n=0}^\infty$ converges to p with rate of convergence $O(k^n)$ with $k < 1$, that is

$$p_n = p + O(k^n), k < 1.$$

Pseudo Code

The equation is $x_{k+1} = f(x_k)$ with the initial value x_0.

INPUT: Equation $x_{k+1} = f(x_k)$; initial value x_0; TOL and maximal iteration times N.

OUTPUT: Sequence of $\{x_k\}, k = 0, 1, 2...$ and the message of convergence.

Step 1: Set x_0 equals to the initial value and $n = 1$.

Step 2: While $n \leq N$, do Steps $3 - 4$.

- Step 3: Set $x_n = f(x_{n-1})$.
- Step 4: If $|x_n - x_{n-1}| < TOL$, then output "Convergence". STOP.

Step 5: Output "The iteration times exceed the maximal number N". STOP.

Convergence

According to the pseudo code above, we could write program to calculate the sequence $\{x_k\}$ and the points of intersection between $f(x) = x$ and $f(x_k)$.

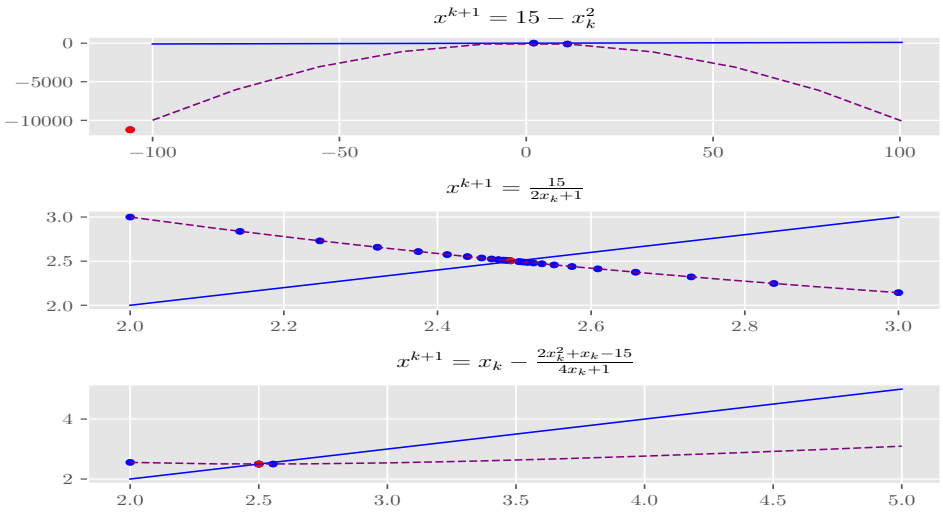

Figure 2.1 Points of Intersection between $f(x) = x$ and $f(x_k)$

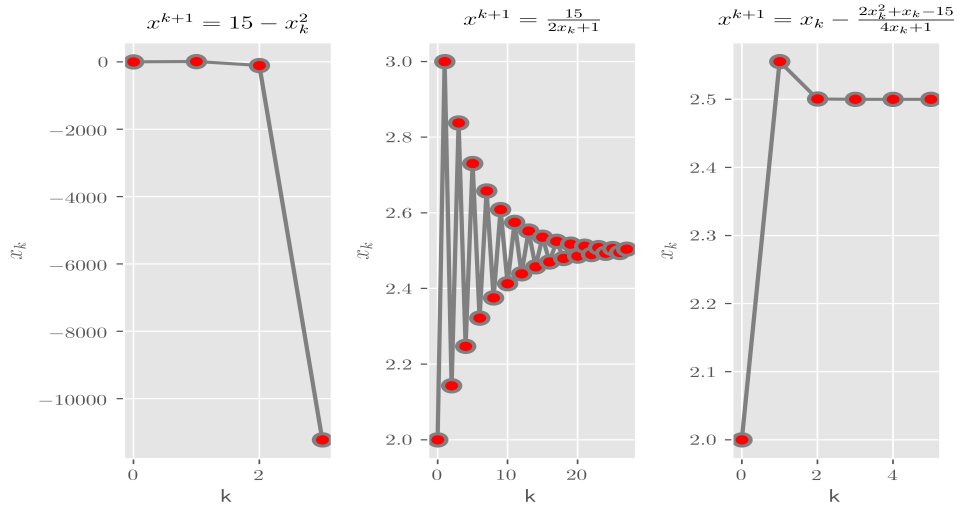

Figure 2.2 Sequence of $\{x_k\}$

Table 2.1: Sequence of $\{x_k\}$

k	Function 1	Function 2	Function 3
0	2.0000000000	2.0000000000	2.0000000000
1	11.000000000	3.0000000000	2.5555555556
2	-106.0000000	2.1428571429	2.5005500550
3	-11221.00000	2.8378378378	2.5000000550
4	-125910826.0	2.2469635628	2.5000000000
5	-∞	2.7302873987	2.5000000000
6	-∞	2.3217748375	2.5000000000
7	-∞	2.6579016513	2.5000000000
8	-∞	2.3749947998	2.5000000000
9	-∞	2.6087003707	2.5000000000
10	-∞	2.4125837506	2.5000000000
11	-∞	2.5750332495	2.5000000000
12	-∞	2.4389980177	2.5000000000
13	-∞	2.5518901186	2.5000000000
14	-∞	2.4574934577	2.5000000000
15	-∞	2.5359312226	2.5000000000
16	-∞	2.4704116958	2.5000000000
17	-∞	2.5249025280	2.5000000000
18	-∞	2.4794187351	2.5000000000
19	-∞	2.5172695303	2.5000000000
20	-∞	2.4856910941	2.5000000000
21	-∞	2.5119812344	2.5000000000
22	-∞	2.4900553544	2.5000000000
23	-∞	2.5083147671	2.5000000000
24	-∞	2.4930901786	2.5000000000
25	-∞	2.5057714778	2.5000000000
26	-∞	2.4951996702	2.5000000000
27	-∞	2.5040066860	2.5000000000

According to the Figure 2.1, 2.2 and Table 2.1, we could figure out that the first function is non-convergent while the second and third is convergent.

What's more, in the first function, after four times iterations, the error exceed 10^8. While in the third function, it reaches convergence only after three iterations. In addition, in the second function, the error reaches 10^{-2} after 27 iterations. Then, we will change the initial value of each function to observe the stability.

Stability

Set $\Delta = 0.05$ and change the initial value into $x_0 + \Delta$. Choose different Δ to change the initial value and observe the sequence $\{x_k\}$ of each function.

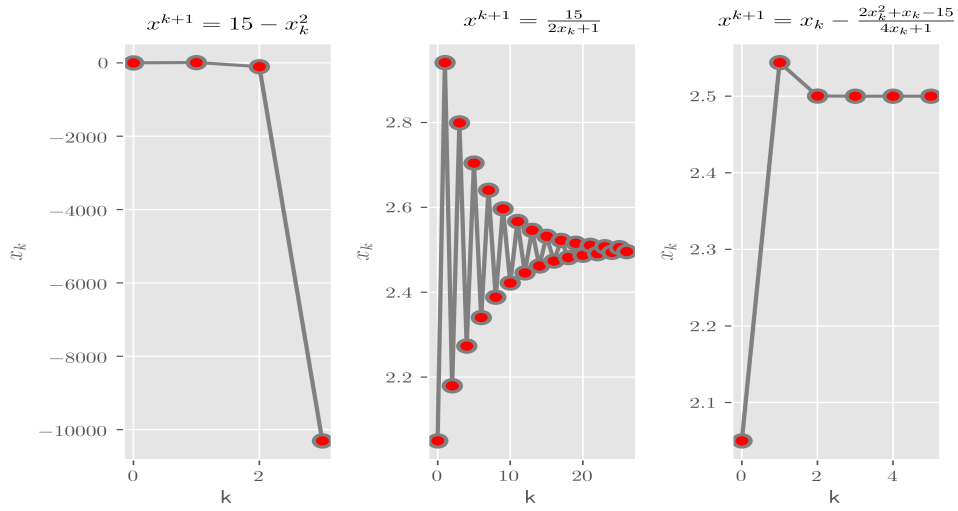

Figure 2.3 Sequence of $\{x_k\}$ with $\Delta = 0.05$

Table 2.2: Sequence of $\{x_k\}$ with $\Delta = 0.05$

k	Function 1	Function 2	Function 3
0	2.0500000000	2.0500000000	2.0500000000
1	10.797500000	2.9411764706	2.5440217391
2	-101.5860063	2.1794871795	2.5003467964
3	-10304.71667	2.7990430622	2.5000000219
4	-106187170.6	2.2733865120	2.5000000000
5	-∞	2.7042750686	2.5000000000
6	-∞	2.3406230237	2.5000000000
7	-∞	2.6402658633	2.5000000000
8	-∞	2.3883328121	2.5000000000
9	-∞	2.5966536711	2.5000000000
10	-∞	2.4219692599	2.5000000000
11	-∞	2.5667621159	2.5000000000
12	-∞	2.4455760560	2.5000000000
13	-∞	2.5461912568	2.5000000000
14	-∞	2.4620909745	2.5000000000
15	-∞	2.5319951563	2.5000000000
16	-∞	2.4736187274	2.5000000000
17	-∞	2.5221794344	2.5000000000
18	-∞	2.4816527816	2.5000000000
19	-∞	2.5153834297	2.5000000000
20	-∞	2.4872458760	2.5000000000
21	-∞	2.5106738151	2.5000000000
22	-∞	2.4911366892	2.5000000000
23	-∞	2.5074079787	2.5000000000
24	-∞	2.4938418908	2.5000000000
25	-∞	2.5051423133	2.5000000000
26	-∞	2.4957220717	2.5000000000
27	-∞	2.5035700310	2.5000000000

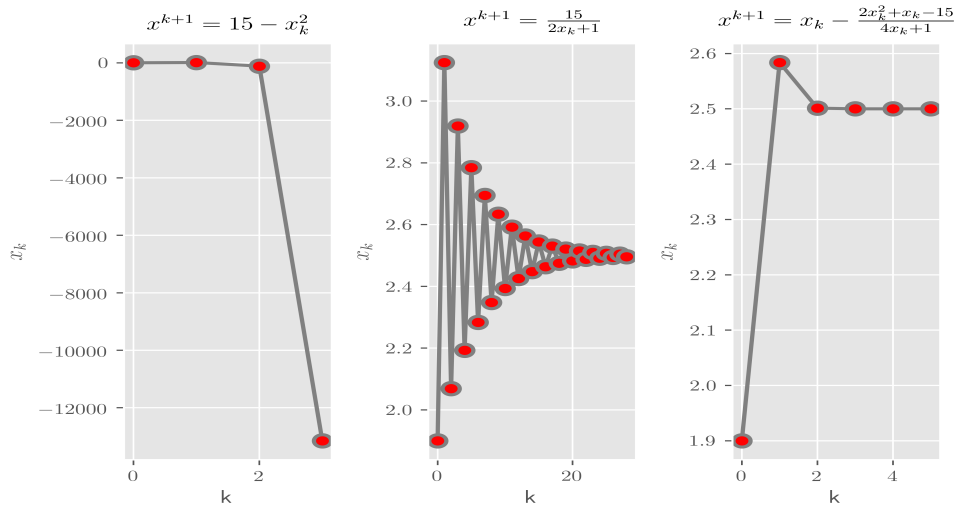

Figure 2.4 Sequence of $\{x_k\}$ with $\Delta = -0.1$

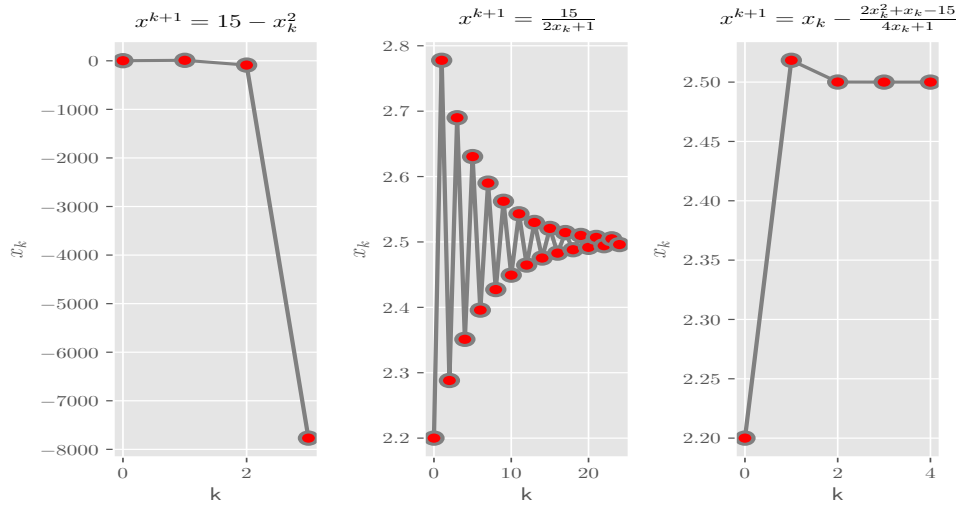

Figure 2.5 Sequence of $\{x_k\}$ with $\Delta = 0.2$

Compare the Figures about Δ and Table 2.2 with the original Figure 2.2 and Table 2.1, we could find that the second and third function both are conditionally stable.

Code

```
def solve1(x0):
    cnt = 2
    x1 = 15 - x0 * x0
    while abs(x1 - x0) < 200:
        x0 = x1
        x1 = 15 - x0 * x0
        cnt = cnt + 1
```

```
8
9
10  def solve2(x0):
11      cnt = 2
12      x1 = 15 / (2 * x0 + 1)
13      while abs(x1 - x0) > 1e-2:
14          x0 = x1
15          x1 = 15 / (2 * x0 + 1)
16          cnt = cnt + 1
17
18
19  def solve3(x0):
20      cnt = 2
21      x1 = x0 - (2 * x0 * x0 + x0 - 15) / (4 * x0 + 1)
22      while abs(x1 - x0) > 1e-8:
23          x0 = x1
24          x1 = x0 - (2 * x0 * x0 + x0 - 15) / (4 * x0 + 1)
25          cnt = cnt + 1
26
27
28  def main():
29      solve1(2)
30      solve2(2)
31      solve3(2)
```

2.2.2 Task 2

Problem Description

Prove that there is only one real root in (0,1) of the equation $2-3x-sin(x)=0$ and use bisection method to find the root whose error within 0.0005 and the number of iterations required.

Problem Analysis

Prove that there is only one real root of the equation $2-3x-sin(x)=0$ means we need to change the original equation into fixed-point format so that it is possible to use fixed-point theorem to prove the problem.

$$2-3x-sin(x)=0 \Longrightarrow x=\frac{2-sin(x)}{3}$$

Then we should apply the bisection method on this equation to find a root whose error is within 0.0005 and calculate the number of iterations required. Therefore, we need to introduce the fixed-point theorem and the bisection method.

Related Theorem

Fixed-point Theorem: Meet the two conditions below, we could find that, for any number $p_0 \in [a, b]$, the sequence $\{p_n\}_0^\infty$ defined by $p_n = g(p_{n-1}), n \geq 1$ converges to the unique fixed point p in $[a, b]$.

- (Existence) If $g(x) \in C[a, b]$ and $g(x) \in [a, b]$ for all $x \in [a, b]$, then $g(x)$ has a fixed point in $[a, b]$. ($g(x) \in [a, b]$ means $max\{g(x)\} \leq b, min\{g(x)\} \geq a$)
- (Uniqueness) If, in addition, $g'(x)$ exists on (a, b), and a positive constant $k < 1$ exists with $|g'(x)| \leq k$, for all $x \in (a, b)$.

Proof

According to the fixed-point theorem, we are supposed to calculate the $g(x) = \frac{2 - sin(x)}{3}$ and the $g'(x)$ in $(0, 1)$.

Firstly, $g(x) = \frac{2 - sin(x)}{3} \in C[0, 1]$, and for all $x \in [0, 1]$, $0 \leq sin(x) < 1$ so that $g(x) \in [\frac{1}{3}, \frac{2}{3}]$ which means $g(x) \in [0, 1]$ for $\forall x \in [0, 1]$. Consequently, $g(x)$ has a fixed point in $[0, 1]$. In addition, prove the uniqueness of the fixed point.

$$g(x) = \frac{2 - sin(x)}{3} \Longrightarrow g'(x) = \frac{-cos(x)}{3}$$

Because of $cos(x) \in [0, 1]$ for $\forall x \in [0, 1]$, $g'(x) \in [\frac{1}{3}, \frac{2}{3}]$. Therefore, there exists a positive constant $k < 1$ with $|g'(x)| \leq k$ for all $x \in [0, 1]$. Hence, there is only one fixed point in $[0, 1]$.

Pseudo Code of the Bisection Method

The goal is to find the real root of $x - 3x - sin(x) = 0$ in the interval $(0, 1)$.
INPUT: Endpoints $a = 0, b = 1$; tolerance $TOL = 0.0005$; Maximum number of iterations N.
OUTPUT: Approximate solution p or message of failure.
Step 1: Set $n = 1, FA = f(a)$.
Step 2: While $n \leq N$, do Steps 3 to 6.

- Step 3: Set $p = \frac{a + b}{2}$; and compute $FC = f(p)$.
- Step 4: If $FC = 0$ or $\frac{|b - a|}{2} < TOL$, then output p. (Procedure complete successfully) Stop.

- Step 5: If $FA * FC < 0$, then set $b = p$; else set $a = p$.
- Step 6: Set $n = n + 1$.

Step 7: OUTPUT "Method failed after N iterations." STOP.

Solution

Apply the Bisection method to the equation $2 - 3x - sin(x) = 0$ with the initial points $a = 0, b = 1$ and generate the sequence of $\{p_n\}$, $\{a_n\}$ and $\{b_n\}$.

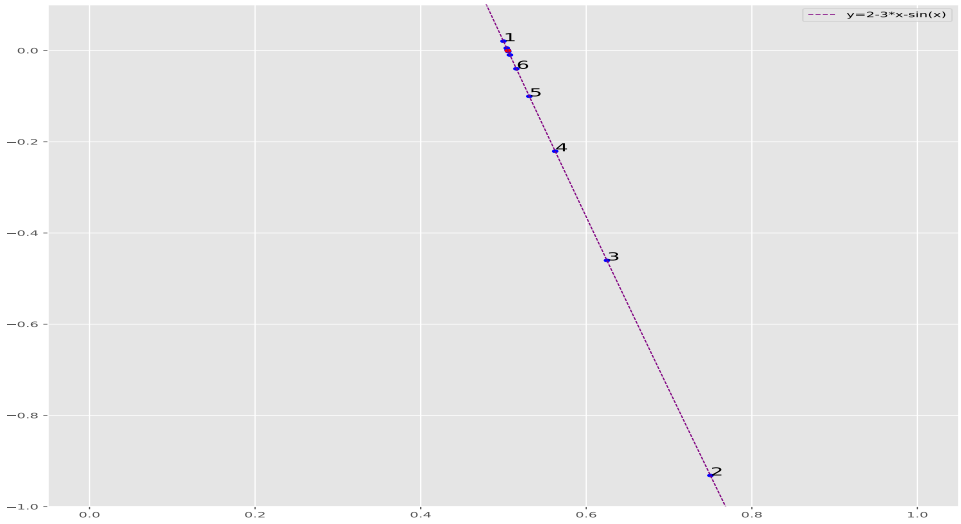

Figure 2.6 $\{p_n\}$ on the function $f(x) = 2 - 3x - sin(x)$

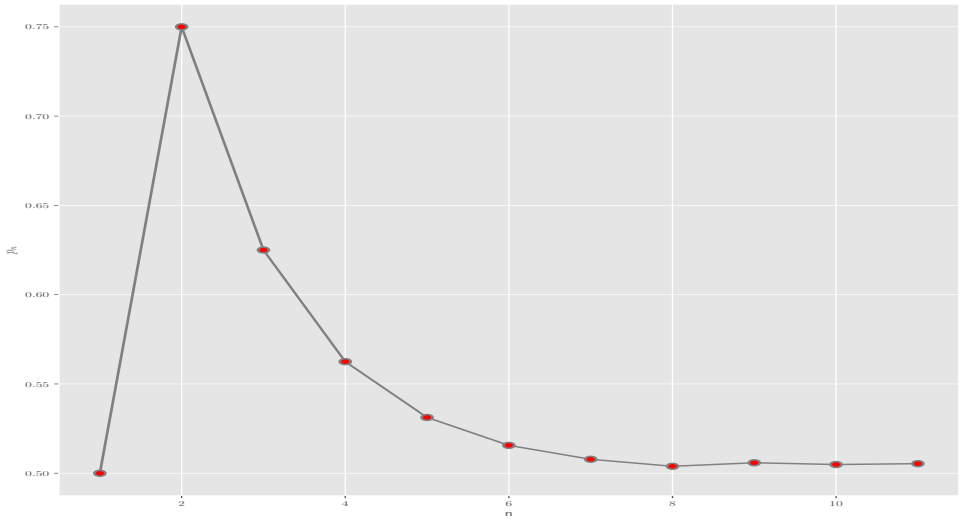

Figure 2.7 Sequence of $\{p_n\}$

Table 2.3: Result of Bisection Method

n	a_n	b_n	p_n	$2 - 3x_n - sin(x_n)$
1	0.5000000000	1.0000000000	0.5000000000	0.0205744614
2	0.5000000000	0.7500000000	0.7500000000	-0.9316387600
3	0.5000000000	0.6250000000	0.6250000000	-0.4600972729
4	0.5000000000	0.5625000000	0.5625000000	-0.2208026735
5	0.5000000000	0.5312500000	0.5312500000	-0.1003614548
6	0.5000000000	0.5156250000	0.5156250000	-0.0399536858
7	0.5000000000	0.5078125000	0.5078125000	-0.0097044518
8	0.5039062500	0.5078125000	0.5039062500	0.0054313210
9	0.5039062500	0.5058593750	0.5058593750	-0.0021374896
10	0.5048828125	0.5058593750	0.5048828125	0.0016466850
11	0.5048828125	0.5053710938	0.5053710938	-0.0002454600

According to the experimental result above, we could figure that the convergent point is 0.5053710938 and the number of iteration required is 11.

Code

```
import math


def func(x):
    return 2 - 3 * x - math.sin(x)


def solve(l, r):
    tot = 0
    while r - l > 0.0005:
        mid = (l + r) / 2
        tot = tot + 1
        if func(mid) > 0:
            l = mid
        else:
            r = mid
        print(tot, '%.10f' % l, '%.10f' % r, '%.10f' % mid, '%.10f' % func(mid))
    print("Total iteration times:", tot,
          ", Convergent point: (", r, ",", func(r), ")")


def main():
    solve(0, 1)
```

2.2.3 Task 3

Problem Description

Use Newton's method to solve the equation

$$\frac{1}{2} + \frac{1}{4}x^2 - xsin(x) - \frac{1}{2}cos(2x) = 0$$

with the initial value x_0 equals to $\frac{\pi}{2}, 5\pi, 10\pi$. And the accuracy is required less than 10^{-5}. Analysis the effect of initial value on result.

Introduction for Newton's Method

1) Suppose that $f \in C^2[a, b]$, and x^* is a solution of $f(x) = 0$.

2) Let $\hat{x} \in [a, b]$ be an approximation to x^* such that $f'(\hat{x}) \neq 0$ and $|\hat{x} - x^*|$ is "small".

Consider the first Taylor polynomial for $f(x)$ expanded about $\hat{x}$,

$$f(x) = f(\hat{x}) + (x - \hat{x})f'(\hat{x}) + \frac{(x - \hat{x})^2}{2}f''(\xi(x)).$$

where $\xi(x)$ lies between x and $\hat{x}$. Thus, consider $f(x^*) = 0$, and gives

$$0 = f(x^*) \approx f(\hat{x}) + (x^* - \hat{x})f'(\hat{x}).$$

Solution for finding x^* is

$$x^* \approx \hat{x} - \frac{f(\hat{x})}{f'(\hat{x})}.$$

Convergence Theorem for Newton's Method

1) $f \in C^2[a, b]$.

2) $p \in [a, b]$ is such that $f(p) = 0$ and $f'(p) \neq 0$.

Then there exists a $\delta > 0$ such that Newton's method generates a sequence $\{p_n\}_1^\infty$ converging to p for any initial approximation $p_0 \in [p - \delta, p + \delta]$.

Problem Analysis

According to the theorem described above, we could figure out that the initial value of Newton's Method should be close to the root, otherwise it may be non-convergent. What's more, the iterative equation is $x^* \approx \hat{x} - \frac{f(\hat{x})}{f'(\hat{x})}$ which means if the value of $f'(\hat{x}$ approximate to zero, a large loss of accuracy may occur.

Therefore, in this problem, we apply different initial value on the Newton's Iteration Method to analysis the effect on the result.

Pseudo Code

The function f is differentiable and p_0 is an initial approximation.

INPUT: Equation $f(x)$; Initial approximation x_0; Tolerance TOL; Maximum number of iteration N.

OUTPUT: Approximate solution x or message of failure.

Step 1: Set $n = 1$.

Step 2: While $n \leq N$, do Steps 3 to 5.

- Step 3: Set $x = x_0 - \frac{f(x_0)}{f'(x_0)}$.
- Step 4: If $|x - x_0| < TOL$, then output x; (Procedure complete successfully.) Stop!
- Step 5: Set $n = n + 1, x_0 = x$.

Step 6: OUTPUT "Method failed after N iterations." STOP!

Solution

$$f(x) = \frac{1}{2} + \frac{1}{4}x^2 - x * sin(x) - \frac{1}{2}cos(2x)$$
$$f'(x) = \frac{x}{2} - sin(x) - x * sin(x) + sin(2x)$$
$$x_{n+1} = g(x_n) = x_n - \frac{f(x_n)}{f'(x_n)}$$

According to the image of $f(x)$ and $f'(x)$, we could find that the root of equation is near zero and there are several zero points in $[0, 25]$ which may lead to the loss of accuracy. Then we use the Newton's method to calculate the answer.

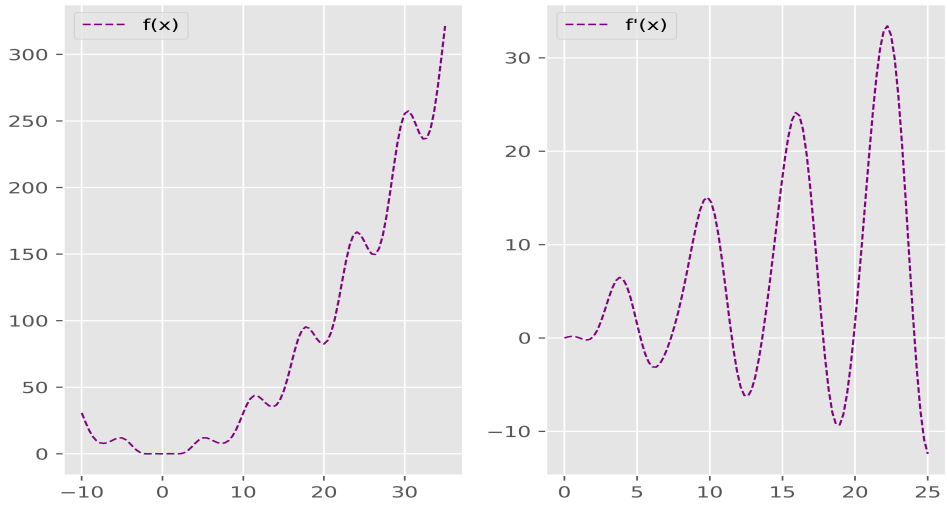

Figure 2.8 The Image of $f(x)$ and $'f(x)$

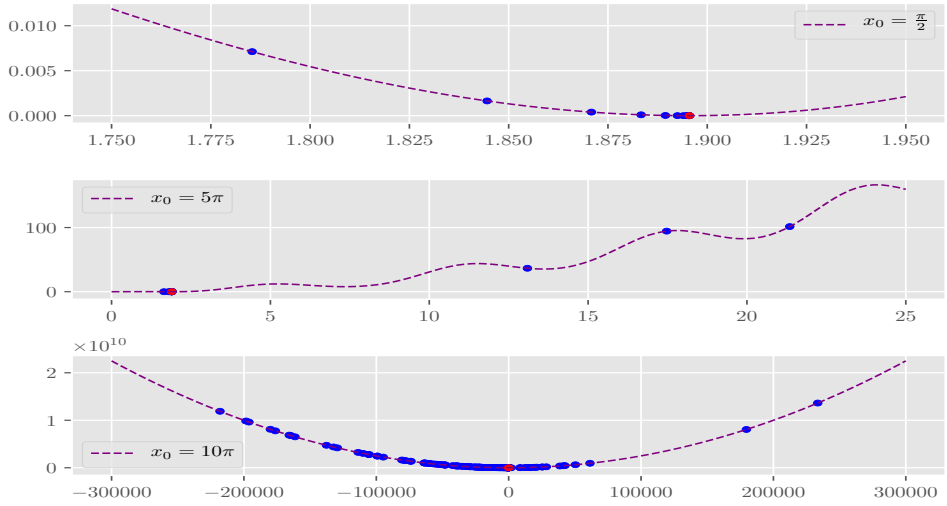

Figure 2.9 Result of $x_0 = \dfrac{\pi}{2}, 5\pi, 10\pi$

Table 2.4: Result of Newton's Method

	Iteration Times	x_n	$f(x_n)$
$x_0 = \frac{\pi}{2}$	15	1.895498	2.294109e-11
$x_0 = 5\pi$	19	1.895489	1.859912e-11
$x_0 = 10\pi$	154	-5.577213e-06	7.776335e-12

According to the result from the Table 2.4, it is obvious that the first and second sequence converge to the same point while the third is not. What's more, we could figure out the relation of the distance between the initial value and the root and the iteration times is that long-distance, more iteration times.

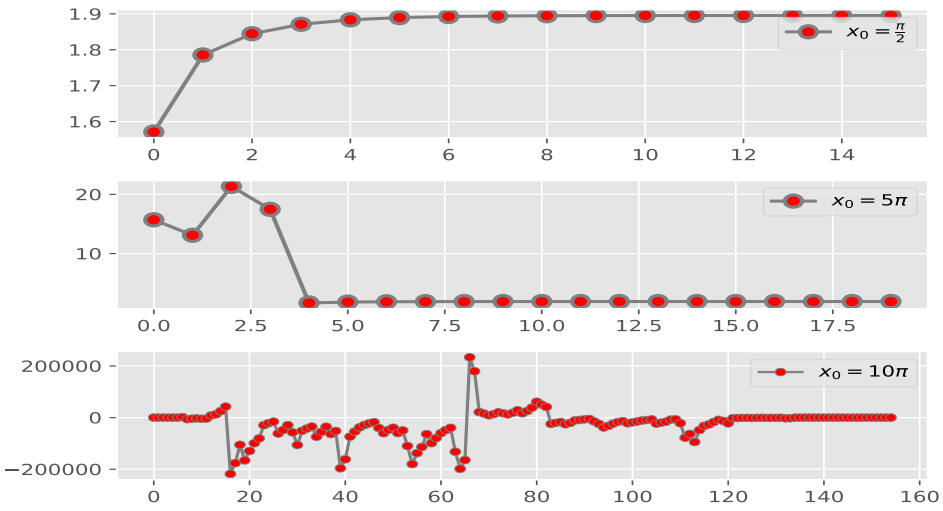

Figure 2.10 Sequence of $\{x_n\}$ when $x_0 = \frac{\pi}{2}, 5\pi, 10\pi$

Code

```python
import math


def func(x):
    return 0.5 + 0.25 * x * x - x * math.sin(x) - 0.5 * math.cos(2 * x)


def func1(x):
    return 0.5 * x - math.sin(x) - x * math.cos(x) + math.sin(2 * x)


def solve(x0, id):
    tot = 1
    x1 = x0 - func(x0) / func1(x0)

    while abs(x0 - x1) > 1e-5:
        tot = tot + 1
        x0 = x1
        x1 = x0 - func(x0) / func1(x0)
    print("Total iteration times:", tot, ", Convergent Point: (", x1, ",", func(x1), ")")


def main():
    solve(0.5 * math.pi, 0)
    solve(5 * math.pi, 1)
    solve(10 * math.pi, 2)
```

2.2.4 Task 4

Problem Description

There is a real root of equation

$$f(x) = 5x - e^x$$

in $(0, 1)$. Try to find it using bisection method, Newton's method, secant method and false position method.

Secant Method

For Newton's method, each iteration requires evaluation of both function $f(x_k)$ and its derivative $f'(x_k)$, which may be inconvenient or expensive. What's more, derivative is approximated by finite difference using two successive iterates. Therefore, iteration becomes

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

This method is known as secant method.

False Position Method

To find a solution of $f(x) = 0$ for a given continuous function f on the interval $[p_0, p_1]$, where $f(p_0)$ and $f(p_1)$ have opposite signs

$$f(p_0) * f(p_1) < 0.$$

The approximation p_2 is chosen in same manner as in Secant Method, as the x-intercept of the line joining $(p_0, f(p_0))$ and $(p_1, f(p_1))$. To decide the Secant Line to compute p_3, we need to check $f(p_2) * f(p_1)$ or $f(p_2) * f(p_0)$. If $f(p_2) * f(p_1)$ is negative, we choose p_3 as the x-intercept for the line joining $(p_1, f(p_1)$ and $p_2, f(p_2)$. In a similar manner, we can get a sequence $\{p_n\}_2^\infty$ which approximates to the root.

Pseudo Code of Secant Method

The equation is $x_{k+1} = g(x_k) = x_k - f(x_k) * \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$ with the initial value x_0, x_1.

INPUT: Initial approximation x_0, x_1; Tolerance TOL; Maximum number of iter-

ation N.

OUTPUT: Approximate solution x or message of failure.

Step 1: Set $n = 2, y_0 = f(x_0), y_1 = f(x_1)$.

Step 2: While $n \leq N$, do Steps 3 to 5.

- Step 3: Set $x = x_1 - y_1 * \dfrac{x_1 - x_0}{y_1 - y_0}$.(Compute x_i)
- Step 4: If $|x - x_1| < TOL$, then output x; (Procedure complete successfully.) Stop!
- Step 5: Set $n = n + 1, x_0 = x_1, x_1 = x, y_0 = y_1, y_1 = f(x)$.

Step 6: OUTPUT "Method failed after N iterations." STOP!

Pseudo Code of False Position Method

INPUT: Initial approximation x_0, x_1; Tolerance TOL; Maximum number of iteration N.

OUTPUT: Approximate solution x or message of failure.

Step 1: Set $n = 2, y_0 = f(x_0), y_1 = f(x_1)$.

Step 2: While $n \leq N$, do Steps 3 to 6.

- Step 3: Set $x = x_1 - y_1 * \dfrac{x_1 - x_0}{y_1 - y_0}$.(Compute x_i)
- Step 4: If $|x - x_1| < TOL$, then output x; (Procedure complete successfully.) Stop!
- Step 5: Set $n = n + 1, y = f(x)$.
- Step 6: If $y * y_1 < 0$, set $x_0 = x, y_0 = y$; else set $x_1 = x, y_1 = y$.

Step 6: OUTPUT "Method failed after N iterations." STOP!

Solution

In the four find-root methods above, we set the TOL to be 10^-5 and the maximal number of iteration to be 200. The results are shown below.

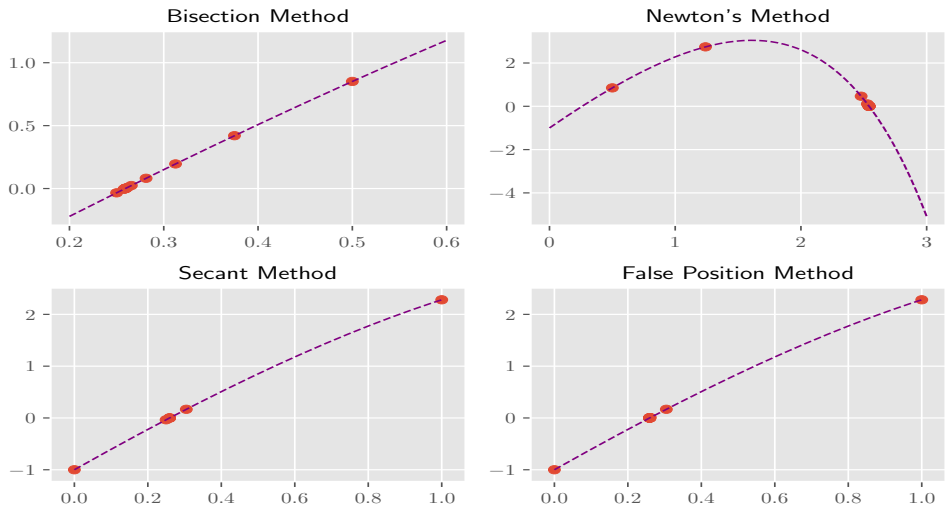

Figure 2.11 Results of the four Find-Root Methods

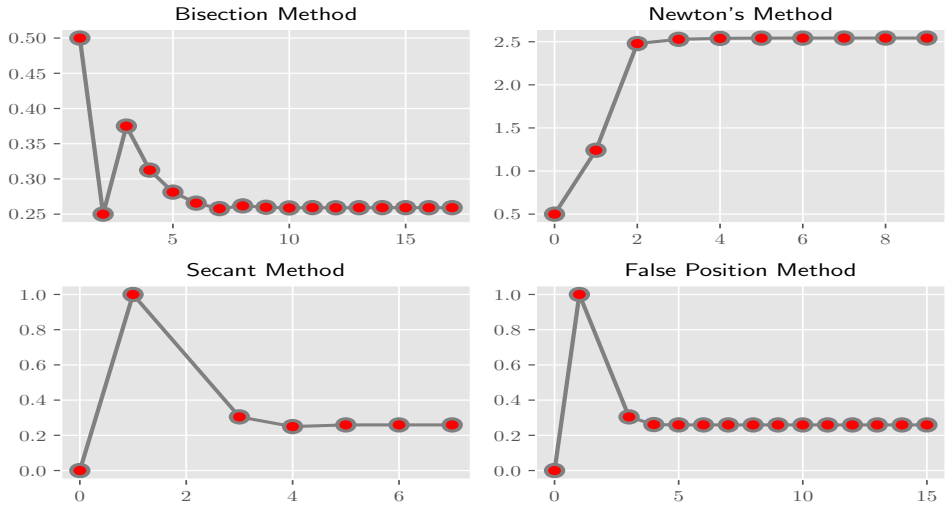

Figure 2.12 Sequence $\{x_n\}$ of the four Find-Root Methods

Table 2.5: Sequence of the four Find-Root Methods

n	Bisection Method	Newton's Method	Secant Method	False Position Method
0	0.5000	0.5000	0.0000	0.0000
1	0.2500	1.2411	1.0000	1.0000
2	0.3750	2.4790	0.3047	0.3047
3	0.3125	2.5283	0.2497	0.2610
4	0.2812	2.5392	0.2592	0.2592
5	0.2656	2.5418	0.2592	0.2592
6	0.2578	2.5424	0.2592	0.2592
7	0.2617	2.5426	0.2592	0.2592
8	0.2598	2.5426	0.2592	0.2592
9	0.2588	2.5426	0.2592	0.2592
10	0.2593	2.5426	0.2592	0.2592
11	0.2590	2.5426	0.2592	0.2592
12	0.2592	2.5426	0.2592	0.2592

According to the result shown above, we could find that the first, third and the fourth have the same convergence point. In addition, the third and fourth reaches the convergence point only after three times of iteration while the second needs seven times and the first uses twelve times.

Code

```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib
from matplotlib import style

style.use('ggplot')
matplotlib.rcParams['text.usetex'] = True
plt.figure(figsize=(10, 10), dpi=700)
f, ax = plt.subplots(2, 2)
style_list = ["g+-", "r--", "b.-", "yo-"]


def func(x):
    return 5 * x - np.exp(x)


def func2(x):
    return x - np.exp(x)


def solve1(l, r):
    # X, Y coordinate point
    X = []
    Y = []
    tot = 0

    # Function drawing
    x = np.linspace(0.2, 0.6, 1000)
    y = 5 * x - np.exp(x)
    ax[0][0].plot(x, y, color="#800080", linewidth=1.0, linestyle="--")

    # Calculation process
    while abs(r - l) > 1e-2:
        mid = (l + r) / 2
        tot = tot + 1
        if func(mid) > 0:
            r = mid
        else:
            l = mid
        X.append(mid)
```

```
41          Y.append(func(mid))
42          if tot >= 200:
43              break
44
45      ax[0][0].scatter(X, Y, s=30)
46      ax[0][0].set_title("Bisection Method", fontsize=12)
47
48
49  def solve2(x0):
50      # X, Y coordinate point
51      X = []
52      Y = []
53      tot = 1
54      X.append(x0)
55      Y.append(func(x0))
56      x1 = x0 - func(x0) / func2(x0)
57      X.append(x1)
58      Y.append(func(x1))
59
60      # Function drawing
61      x = np.linspace(0, 3, 1000)
62      y = 5 * x - np.exp(x)
63      ax[0][1].plot(x, y, color="#800080", linewidth=1.0, linestyle="--")
64
65      # Calculation process
66      while abs(x0 - x1) > 1e-2:
67          tot = tot + 1
68          x0 = x1
69          x1 = x0 - func(x0) / func2(x0)
70          X.append(x1)
71          Y.append(func(x1))
72          if tot >= 200:
73              break
74
75      ax[0][1].scatter(X, Y, s=30)
76      ax[0][1].set_title("Newton's Method", fontsize=12)
77
78
79  def solve3(x0, x1):
80      # X, Y coordinate point
81      X = []
82      Y = []
83      tot = 2
84      X.append(x0)
85      Y.append(func(x0))
86      X.append(x1)
87      Y.append(func(x1))
88
```

```
    # Function drawing
    xline = np.linspace(0, 1, 1000)
    yline = 5 * xline - np.exp(xline)
    ax[1][0].plot(xline, yline, color="#800080", linewidth=1.0, linestyle="--")

    # Calculation process
    while abs(x0 - x1) > 1e-2:
        tot = tot + 1
        x = x1 - func(x1) * (x1 - x0) / (func(x1) - func(x0))
        X.append(x)
        Y.append(func(x))
        x0 = x1
        x1 = x
        if tot >= 200:
            break

    ax[1][0].scatter(X, Y, s=30)
    ax[1][0].set_title("Secant Method", fontsize=12)


def solve4(x0, x1):
    # X, Y coordinate point
    X = []
    Y = []
    tot = 2
    X.append(x0)
    Y.append(func(x0))
    X.append(x1)
    Y.append(func(x1))

    # Function drawing
    xline = np.linspace(0, 1, 1000)
    yline = 5 * xline - np.exp(xline)
    ax[1][1].plot(xline, yline, color="#800080", linewidth=1.0, linestyle="--")

    # Calculation process
    while abs(x0 - x1) > 1e-2:
        tot = tot + 1
        x = x1 - func(x1) * (x1 - x0) / (func(x1) - func(x0))
        X.append(x)
        Y.append(func(x))
        if func(x) * func(x1) < 0:
            x0 = x
        else:
            x1 = x
        if tot >= 200:
            break

```

```
137     ax[1][1].scatter(X, Y, s=30)
138     ax[1][1].set_title("False Position Method", fontsize=12)
139 
140 
141 def main():
142     solve1(0, 1)
143     solve2(0.5)
144     solve3(0, 1)
145     solve4(0, 1)
146     plt.tight_layout()
147     plt.savefig('plot4-1.pdf', dpi=700)
148     plt.show()
149 
150 
151 if __name__ == '__main__':
152     main()
```

Chapter 3

Linear Systems

3.1 Experimental Requirements

1) Master the basic concept including the norm of vectors and matrices and special matrices such as strictly diagonally dominant matrix and positive definite symmetric matrix.

2) Comprehend several basic algorithms about matrix and their convergence and convergence rate.

   · Direct solution: LU factorization and LL^T, LDL^T factorization for symmetric matrices.

   · Iterative solution: Jacobi, Gauss-Seidel and SOR.

3) Understand the difference between these algorithms including their advantages and disadvantages and convergence rates.

3.2 Experimental Assignments

3.2.1 Task 1

Problem Description

Calculate the answer of the system of linear equations below.

$$4x - y + z = 7$$
$$4x - 8y + z = -21$$
$$-2x + y + 5z = 15$$

(1) Find the solution of this system of linear equations with LU factorization.

(2) Solve this system of linear equations with Jacobi and Gauss-Seidel methods, respectively.

Method of LU factorization

$$A = LU = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ l_{21} & 1 & 0 & \cdots & 0 \\ l_{31} & l_{32} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{pmatrix}$$
$$= \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix}$$

According to the equation above, we could calculate the answer of l_{ij} and u_{ij} for $i \in [1, n], j \in [1, n]$.

$$\begin{cases} u_{1j} = \dfrac{a_{1j}}{l_{11}}, j \in [1, n] \\ l_{j1} = \dfrac{a_{j1}}{u_{11}}, j \in [1, n] \\ l_{ii}u_{ii} = a_{ii} - \displaystyle\sum_{k=1}^{i-1} l_{ik}u_{ki}, i \in [2, n] \\ u_{ij} = \dfrac{1}{l_{ii}}[a_{ij} - \displaystyle\sum_{k=1}^{i-1} l_{ik}u_{kj}], j \in [i+1, n] \\ l_{ji} = \dfrac{1}{u_{ii}}[a_{ji} - \displaystyle\sum_{k=1}^{i-1} l_{jk}u_{ki}], j \in [i+1, n] \end{cases}$$

After LU factorization, $Ax = b$ could be changed into $LUx = b$. Therefore, we could denote $y = Ux$ so that it is easy to solve the $Ly = b$ with forward substitution method. Once y is determined, the same as the backward substitution method could be applied to solve linear system $Ux = y$ to determine the final answer x.

Solution of LU factorization

According to the algorithm above, we could factorize A into LU.

$$A = \begin{pmatrix} 4 & -1 & 1 \\ 4 & -8 & 1 \\ -2 & 1 & 5 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ -\frac{1}{2} & -\frac{1}{14} & 1 \end{pmatrix} \begin{pmatrix} 4 & -1 & 1 \\ 0 & -7 & 0 \\ 0 & 0 & \frac{11}{2} \end{pmatrix}$$

Then we denote $y = Ux$ so that it is easy to find the answer for the system $Ly = b$.

$$Ly = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ -\frac{1}{2} & -\frac{1}{14} & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = b = \begin{pmatrix} 7 \\ -21 \\ 15 \end{pmatrix}$$

After calculation, we acquire the vector y.

$$y = \begin{pmatrix} 7 \\ -28 \\ 16.5 \end{pmatrix}$$

At last, we need to solve the system equation $Ux = y$ which is shown below.

$$Ly = \begin{pmatrix} 4 & -1 & 1 \\ 0 & -7 & 0 \\ 0 & 0 & \frac{11}{2} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = y = \begin{pmatrix} 7 \\ -28 \\ 16.5 \end{pmatrix}$$

Therefore, we obtain the final result x.

$$x = \begin{pmatrix} 2 \\ 4 \\ 3 \end{pmatrix}$$

The answer of this system of linear equations is

$$\begin{cases} x = 2 \\ y = 4 \\ z = 3 \end{cases}$$

Substitute it back into the original equation and verify that the answer is correct.

Code of LU factorization

```
def solve(A, n):
    L = [[0] * n for i in range(n)]
    U = [[0] * n for i in range(n)]
    for i in range(0, n):
        L[i][i] = 1
    for j in range(0, n):
        U[0][j] = A[0][j] / L[0][0]
        L[j][0] = A[j][0] / U[0][0]
    for i in range(1, n):
        for j in range(i, n):
            U[i][j] = A[i][j]
            for k in range(0, i):
                U[i][j] -= L[i][k] * U[k][j]
            U[i][j] /= L[i][i]

            L[j][i] = A[j][i]
            for k in range(0, i):
                L[j][i] -= L[j][k] * U[k][i]
            L[j][i] /= U[i][i]
```

Method of Jacobi

Considering a iterative method to solve the the system of linear equations. Given $x^{(0)}$, generating each $x_i^{(k)}$ from components of $x^{(k-1)}$ for $k \geq 1$ by

$$x_i^{(k)} = \frac{\sum_{j=1, j\neq i}^{n} (-a_{ij} x_j^{(k-1)}) + b_i}{a_{ii}}, i = 1, 2, ..., n$$

The algorithm above is called Jacobi Iteration Algorithm. Next, we consider the convergence of this algorithm.

For any $x^{(0)} \in \mathbb{R}^n$, the sequence $\{x^{(k)}\}_{k=0}^{\infty}$ defined by

$$x^{(k)} = Tx^{(k-1)} + c, for\ each\ k \geq 1$$

converges to the unique solution of $x = Tx + c$ if and only if $\rho(T) < 1$.

In addition, if A is strictly diagonally dominant, then for any choice of $x^{(0)}$, both the Jacobi and Gauss-Seidel methods give sequences $\{x^{(k)}\}_{k=0}^{\infty}$ that converge to the unique solution of $Ax = b$.

Solution of Jacobi

According to the introduction for method of Jacobi, we need to set the initial vector for the algorithm above. Firstly, we choose $(1, 5, 2)^T$ while $TOL = 0.0001$. Then we could acquire the convergent sequence.

Table 3.1: Initial vector $x^{(0)} = (1, 5, 2)^T$ in Jacobi

k	x_k	y_k	z_k
0	1.0000000000	5.0000000000	2.0000000000
1	2.5000000000	3.3750000000	2.4000000000
2	1.9937500000	4.1750000000	3.3250000000
3	1.9625000000	4.0375000000	2.9625000000
4	2.0187500000	3.9765625000	2.9775000000
5	1.9997656250	4.0065625000	3.0121875000
6	1.9985937500	4.0014062500	2.9985937500
7	2.0007031250	3.9991210937	2.9991562500
8	1.9999912109	4.0002460938	3.0004570312
9	1.9999472656	4.0000527344	2.9999472656
10	2.0000263672	3.9999670410	2.9999683594
11	1.9999996704	4.0000092285	3.0000171387

In addition, we change the initial vector $x^{(0)} = (-1, 1000, 500)^T$ to observe the result of this algorithm.

According to the result, we could figure out that, in this linear system, different initial vector $x^{(0)}$ means different number of iteration. However, for any initial data, there exists the consistent convergent result mainly for the reason that A is strictly diagonally dominant.

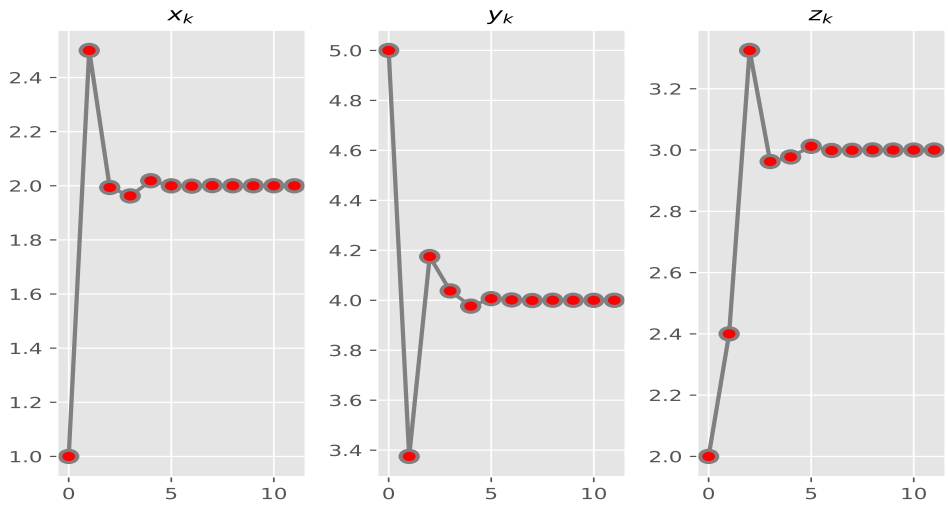

Figure 3.1 Initial vector $x^{(0)} = (1, 5, 2)^T$ in Jacobi

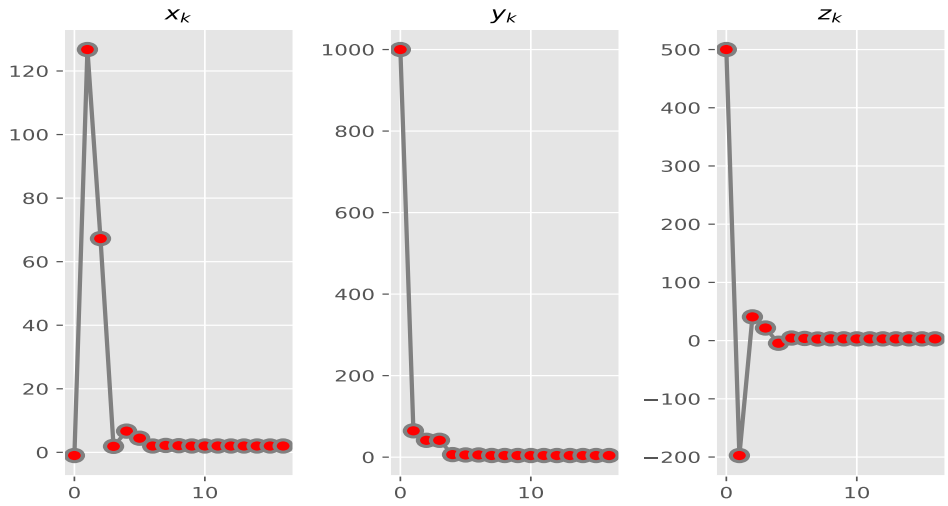

Figure 3.2 Initial vector $x^{(0)} = (-1, 1000, 500)^T$ in Jacobi

Code of Jacobi

```
1  import numpy as np
2
3
4  def NORM(A, B):
5      tmp = 0
6      for i in range(0, len(A)):
7          tmp += (A[i] - B[i]) * (A[i] - B[i])
8      return np.sqrt(tmp)
9
10
11 def solve(A, B, XO, n, N, TOL):
```

```
    X = [0] * n
    k = 1
    while k <= N:
        for i in range(0, n):
            tmp = 0
            for j in range(0, n):
                if j == i:
                    continue
                tmp -= X0[j] * A[i][j]
            X[i] = (tmp + B[i]) / A[i][i]
        if NORM(X, X0) < TOL:
            print("Succeed!")
            return X
        else:
            k = k + 1
            X0 = X[:]
    print("Fail!")
```

Table 3.2: Initial vector $x^{(0)} = (-1, 1000, 500)^T$ in Jacobi

k	x_k	y_k	z_k
0	-1.0000000000	1000.0000000000	500.0000000000
1	126.7500000000	64.6250000000	-197.4000000000
2	67.2562500000	41.3250000000	40.7750000000
3	1.8875000000	41.3500000000	21.6375000000
4	6.6781250000	6.2734375000	-4.5150000000
5	4.4471093750	5.3996875000	4.4165625000
6	1.9957812500	5.4006250000	3.6989062500
7	2.1754296875	4.0852539063	2.7181875000
8	2.0917666016	4.0524882812	3.0531210937
9	1.9998417969	4.0525234375	3.0262089844
10	2.0065786133	4.0031970215	2.9894320312
11	2.0034412476	4.0019683105	3.0019920410
12	1.9999940674	4.0019696289	3.0009828369
13	2.0002466980	4.0001198883	2.9996037012
14	2.0001290468	4.0000738116	3.0000747015
15	1.9999997775	4.0000738611	3.0000368564
16	2.0000092512	4.0000044958	2.9999851388

Method of Gauss-Seidel

Gauss-Seidel iterative technique is a improvement in Jacobi iterative technique. The modify entry point is the components of $x^{(k-1)}$ are used when compute $x_i^{(k)}$. Since, for

$$i > 1, x_1^{(k)}, x_2^{(k)}, ..., x_{i-1}^{(k)}$$

have already been computed and are likely to be better approximations to the actual solutions

$$x_1, x_2, ..., x_{i-1}$$

than

$$x_1^{(k-1)}, x_2^{(k-1)}, ..., x_{i-1}^{(k-1)}$$

Therefore, we change the Jacobi equation into

$$x_i^{(k)} = \frac{-\sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k-1)} + b_i}{a_{ii}}, i = 1, 2, ..., n$$

Solution of Gauss-Seidel

In the algorithm of Gauss-Seidel, we replace $x^{(k)}$ with $x^{(k-1)}$ to compute $x^{(k)}$. Therefore, we apply the same initial vector described above to this algorithm and compare the convergence and convergent rate.

According to the result, we could find that both algorithm reaches the correct convergent point while Gauss-Seidel possess the faster convergent rate.

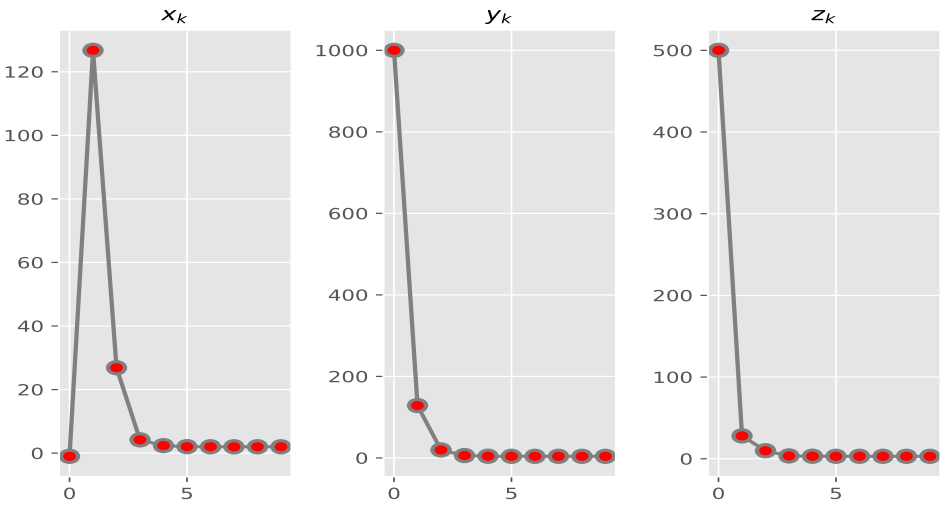

Figure 3.3 Initial vector $x^{(0)} = (-1, 1000, 500)^T$ in Gauss-Seidel

Table 3.3: Initial vector $x^{(0)} = (1, 5, 2)^T$ in Gauss-Seidel

k	x_k	y_k	z_k
0	1.0000000000	5.0000000000	2.0000000000
1	2.5000000000	4.1250000000	3.1750000000
2	1.9875000000	4.0156250000	2.9918750000
3	2.0059375000	4.0019531250	3.0019843750
4	1.9999921875	4.0002441406	2.9999480469
5	2.0000740234	4.0000305176	3.0000235059
6	2.0000017529	4.0000038147	2.9999999382

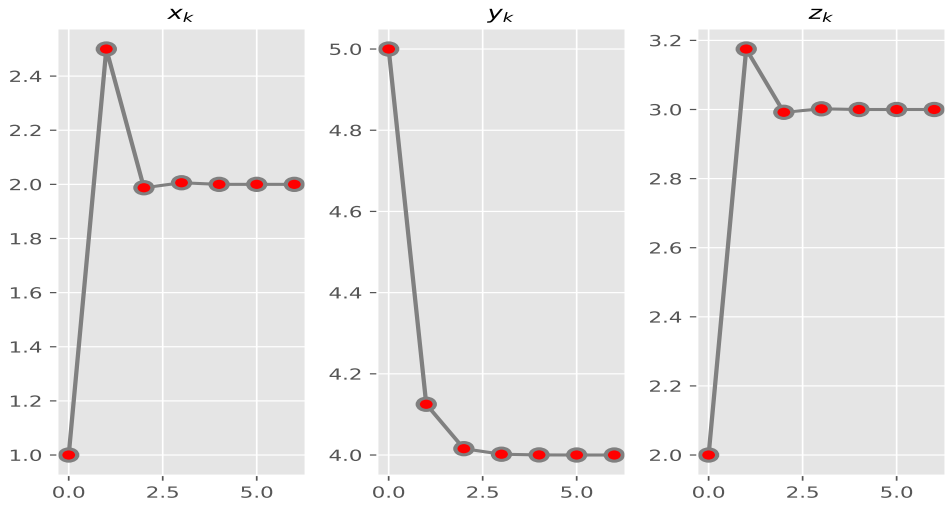

Figure 3.4 Initial vector $x^{(0)} = (1, 5, 2)^T$ in Gauss-Seidel

Table 3.4: Initial vector $x^{(0)} = (-1, 1000, 500)^T$ in Gauss-Seidel

k	x_k	y_k	z_k
0	-1.0000000000	1000.0000000000	500.0000000000
1	126.7500000000	128.5000000000	28.0000000000
2	26.8750000000	19.5625000000	9.8375000000
3	4.1812500000	5.9453125000	3.4834375000
4	2.3654687500	4.2431640625	3.0975546875
5	2.0364023437	4.0303955078	3.0084818359
6	2.0054784180	4.0037994385	3.0014314795
7	2.0005919897	4.0004749298	3.0001418099
8	2.0000832800	4.0000593662	3.0000214387
9	2.0000094819	4.0000074208	3.0000023086

Code of Gauss-Seidel

```
1 import numpy as np
2
```

```
3
4  def NORM(A, B):
5      tmp = 0
6      for i in range(0, len(A)):
7          tmp += (A[i] - B[i]) * (A[i] - B[i])
8      return np.sqrt(tmp)
9
10
11 def solve(A, B, X0, n, N, TOL):
12     X = [0] * n
13     k = 1
14     while k <= N:
15         for i in range(0, n):
16             tmp = 0
17             for j in range(0, i):
18                 tmp -= X[j] * A[i][j]
19             for j in range(i + 1, n):
20                 tmp -= X0[j] * A[i][j]
21             X[i] = (tmp + B[i]) / A[i][i]
22         if NORM(X, X0) < TOL:
23             print("Succeed!")
24             return
25         else:
26             k = k + 1
27             X0 = X[:]
28     print("Fail!")
```

Chapter 4

Interpolation and Polynomial Approximation

4.1 Experimental Requirements

1) Master the basic concept of interpolation and the existence and uniqueness conditions of interpolation polynomials.

2) Comprehend the construction method of Lagrange interpolation basis function, basic form of Lagrange interpolation polynomial of degree n and its error estimation.

3) Understand the Hermite interpolation and its error estimation.

4) Master the piecewise polynomial interpolation include Linear, quadratic and cubic interpolation and its error estimation.

5) Grasp the method of spline interpolation.

4.2 Homework

4.2.1 Problem 1

Problem Description

Derive the spline function based on the following boundary conditions.

1) Natural boundary

2) Clamped boundary

3) Periodic boundary

4) Force the third derivative of the first and second spline polynomials to be equal, and the third derivative of the cubic spline function of the second to last and the last to be equal.

Definition of Cubic Spline Interpolation

Given a function f defined on $[a, b]$ and a set of nodes

$$a = x_0 < x_1 < ... < x_n = b,$$

a cubic spline interpolation $S(x)$ for $f(x)$ is a function that satisfies the following conditions:

a. $S(x)$ are cubic polynomials composed of $S_j(x)$ defined in the subinterval $[x_j, x_{j+1}]$ with $j \in [0, n-1]$;

b. $S(x_j) = f(x_j)$ for each $j \in [0, n]$;

c. $S_{j+1}(x_{j+1}) = S_j(x_{j+1})$ for each $j \in [0, n-2]$;

d. $S'_{j+1}(x_{j+1}) = S'_j(x_{j+1})$ for each $j \in [0, n-2]$;

e. $S''_{j+1}(x_{j+1}) = S''_j(x_{j+1})$ for each $j \in [0, n-2]$;

f. a boundary condition defined by user.

Derivation of Cubic Spline Interpolation

Let the cubic polynomials

$$S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3$$

on the subinterval

$$[x_j, x_{j+1}], j = 0, 1, ..., n-1.$$

Then the left problem is to determine the coefficients

$$a_j, b_j, c_j, d_j, j = 0, 1, ..., n-1.$$

Clearly, by the condition (b), we could acquire

$$S_j(x_j) = a_j = f(x_j), j \in [0, n]$$

and apply condition (c) to obtain

$$a_{j+1} = S_{j+1}(x_{j+1}) = S_j(x_{j+1}) = a_j + b_j(x_{j+1} - x_j) + c_j(x_{j+1} - x_j)^2 + d_j(x_{j+1} - x_j)^3$$

for each $j \in [0, n-2]$. Then we let $h_j = x_{j+1} - x_j$ for each $j \in [0, n-1]$.

What's more, we could figure out

$$a_n = f(x_n), b_n = S'(x_n), c_n = S''(x_n)/2.$$

Therefore, after derivation, we acquire the equations below.

$$\begin{cases} b_{j+1} = b_j + 2c_j h_j + 3d_j h_j^2 \\ c_{j+1} = c_j + 3d_j h_j \\ b_j = \dfrac{1}{h_j}(a_{j+1} - a_j) - \dfrac{h_j}{3}(2c_j + c_{j+1}) \\ h_{j-1}c_{j-1} + 2(h_{j-1} + h_j)c_j + h_j c_{j+1} = \dfrac{3}{h_j}(a_{j+1} - a_j) - \dfrac{3}{h_{j-1}}(a_j - a_{j-1}) \\ j = 0, 1, ..., n-1 \end{cases}$$

According to the equations above, we could find that it is likely to solve the equations with two more formulas. With two more formulas, we could apply the algorithm of system of linear equations such as Gaussian elimination and LU factorization to solve the c_j and b_j, d_j for each $j \in [0, n-1]$. Furthermore, the two more formulas are provided by the boundary condition defined by user. Thus we will introduce the four boundary conditions provided by the problem to acquire the two more formulas.

Natural boundary

$$S''(x_0) = S''(x_n) = 0$$

Therefore, we could derive the equation into the two more formulas as follows.

$$\begin{cases} c_0 = 0 \\ c_n = 0 \end{cases}$$

Clamped boundary

$$S'(x_0) = f'(x_0), S'(x_n) = f'(x_n)$$

Consequently, the equation could be derived into the two more formulas described blow.

$$S'(x_0) = b_0 = f'(x_0), S'(x_n) = b_n = f'(x_n)$$

$$\begin{cases} 2h_0c_0 + h_0c_1 = \dfrac{3}{h_0}(a_1 - a_0) - 3f'(x_0) \\ h_{n-1}c_{n-1} + 2h_{n-1}c_n = 3f'(x_n) - \dfrac{3}{h_{n-1}}(a_n - a_{n-1}) \end{cases}$$

Periodic boundary

If $f(x)$ is a periodic function and $x_n - x_0$ is a period, $S(x)$ is required to be a periodic function which means

$$S(x_0) = S(x_n), S'(x_0) = S'(x_n), S''(x_0) = S''(x_n)$$

Thus, the equation is possible to be simplified to the two more formulas as follows.

$$b_0 = b_{n-1} + 2c_{n-1}h_{n-1} + 3d_{n-1}h_{n-1}^2$$

$$c_0 = c_{n-1} + 3d_{n-1}h_{n-1}$$

$$\begin{cases} c_0 - c_n = 0 \\ 2h_0c_0 + h_0c_1 + h_{n-1}c_{n-1} + 2h_{n-1}c_n = \dfrac{3(a_1 - a_0)}{h_0} - \dfrac{3(a_n - a_{n-1})}{h_{n-1}} \end{cases}$$

the Fourth Boundary Condition

$$S_0'(x) = S_1'(x), S_{n-2}'(x) = S_{n-1}'(x)$$

Hence, the equation is likely to be derived into the two more formulas described blow.

$$2c_0 + 6d_0(x - x_0) = 2c_1 + 6d_1(x - x_1)$$

$$2c_{n-2} + 6d_{n-2}(x - x_{n-2}) = 2c_{n-1} + 6d_{n-1}(x - x_{n-1})$$

$$\begin{cases} h_1c_0 - (h_0 + h_1)c_1 + h_0c_2 = 0 \\ h_{n-1}c_{n-2} - (h_{n-2} + h_{n-1})c_{n-1} + h_{n-2}c_n = 0 \end{cases}$$

4.3 Experimental Assignments

4.3.1 Task 1

Problem Description

Taking $y = sin(x)$ for example, generate 11 and 21 points of data in $[0, \pi]$. In terms of the four boundary conditions above, design algorithms and programs to calculate their spline interpolation, and analyze the differences between the four methods through graphs.

Related Theorem

If f is defined at $a = x_0 < x_1 < ... < x_n = b$, then f has a unique natural spline interpolant on the nodes $x_0, x_1, ..., x_n$; that is, a spline interpolant that satisfies the boundary conditions $S''(a) = 0$ and $S''(b) = 0$.

If f is defined at $a = x_0 < x_1 < ... < x_n = b$, and differentiable at a and b, then f has a unique clamped spline interpolant on the nodes $x_0, x_1, ..., x_n$; that is, a spline interpolant that satisfies the boundary conditions $S'(a) = f'(a)$ and $S'(b) = f'(b)$.

Solution

According to the theorem and the algorithm described above, we could program to acquire the result of this problem which is shown below.

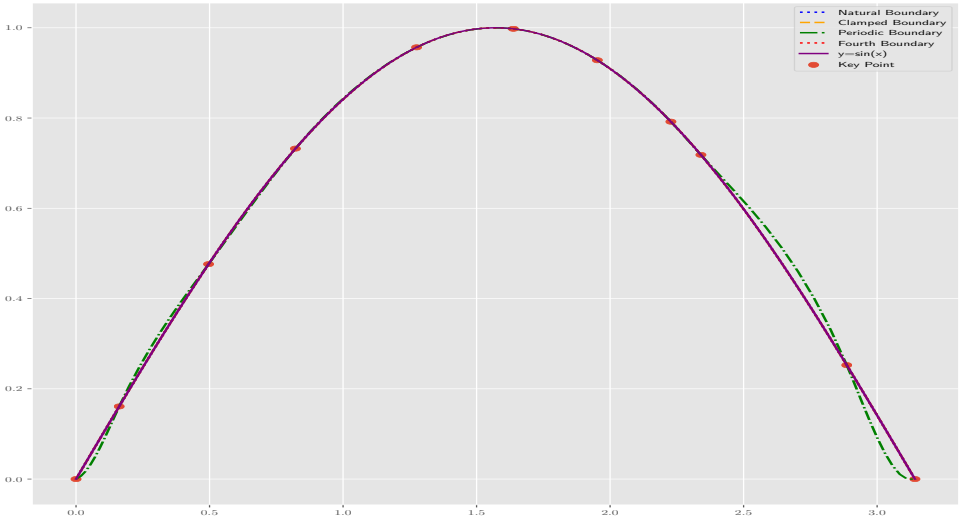

Figure 4.1 $n = 11$

According to Figure 4.1, we could find that the cubic spline interpolations of the four boundary conditions all fit well with the original function despite that there is a small error in the periodic boundary. What's more, with the respect of Figure 4.2, the spline interpolation with the periodic boundary exists huge error compared with the original while others are not which implies that the location of the initial points may have a great impact on the result of spline interpolation.

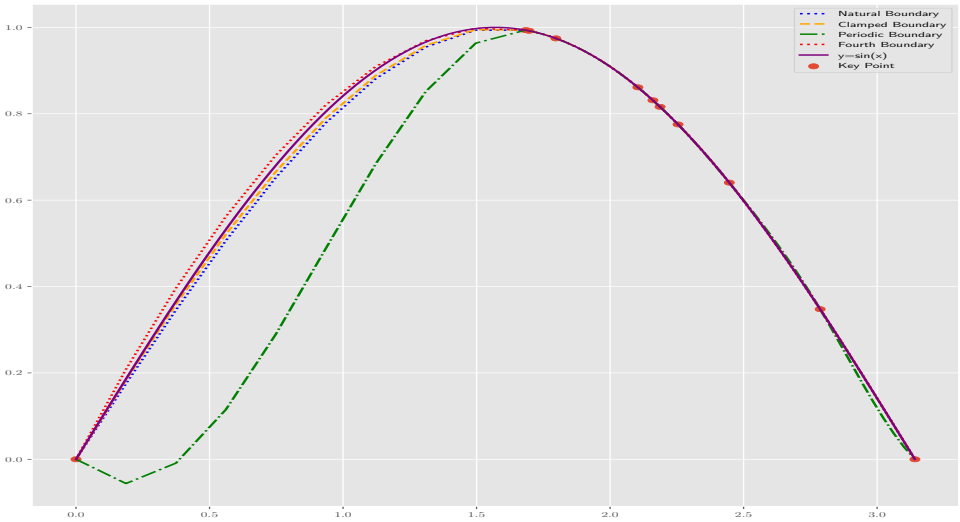

Figure 4.2 $n = 11$ with special initial points

However, with the larger number of initial points, after several tries, there is not huge error with the original which means the stable of spline interpolation is significantly improved. Therefore, we could conclude that the larger the initial data set, the more accurate the result of interpolation.

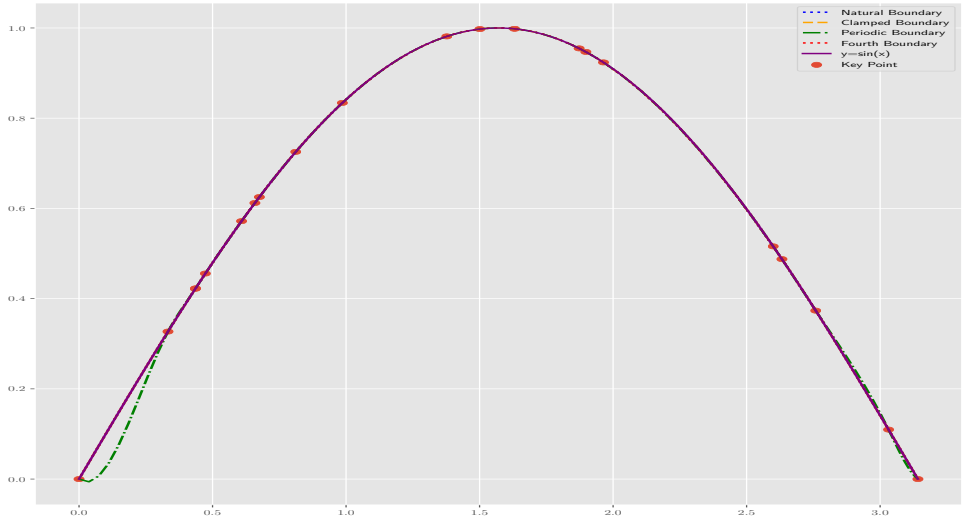

Figure 4.3 $n = 21$

Code

```
import numpy as np
import random
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import style
from copy import deepcopy as dco

style.use('ggplot')

import matplotlib

matplotlib.rcParams['text.usetex'] = True

plt.figure(figsize=(10, 10), dpi=70)
style_list = ["g+-", "r--", "b.-", "yo-"]

l, r, X, Y = 0, np.pi, [], []


def drawFunction(L, R, A, B, C, D, lineName, col, lwidth, style):
    x = np.linspace(L, R, 10)
    y = A + B * (x - L) + C * (x - L) * (x - L) + D * (x - L) * (x - L) * (x - L)
    if L == l:
        plt.plot(x, y, color=col, linewidth=lwidth, linestyle=style, label=lineName)
    else:
        plt.plot(x, y, color=col, linewidth=lwidth, linestyle=style)

def Gaussian(A):
    n = len(A)
```

```
30
31      for i in range(0, n):
32          # Search for maximum in this column
33          maxEl = abs(A[i][i])
34          maxRow = i
35          for k in range(i + 1, n):
36              if abs(A[k][i]) > maxEl:
37                  maxEl = abs(A[k][i])
38                  maxRow = k
39
40          # Swap maximum row with current row
41          for k in range(i, n + 1):
42              A[maxRow][k], A[i][k] = A[i][k], A[maxRow][k]
43
44          # Make all rows below this one 0 in current column
45          for k in range(i + 1, n):
46              c = -A[k][i] / A[i][i]
47              for j in range(i, n + 1):
48                  if i == j:
49                      A[k][j] = 0
50                  else:
51                      A[k][j] += c * A[i][j]
52
53      # Solve equation Ax=b for an upper triangular matrix A
54      x = [0 for i in range(n)]
55      for i in range(n - 1, -1, -1):
56          x[i] = A[i][n] / A[i][i]
57          for k in range(i - 1, -1, -1):
58              A[k][n] -= A[k][i] * x[i]
59      return x
60
61
62  def func(x):
63      return np.sin(x)
64
65
66  def func2(x):
67      return np.cos(x)
68
69
70  def spline(A, B, C, D, H, G, n, lineName, col, lwidth, style):
71      C = Gaussian(G)
72      for i in range(0, n):
73          D.append((C[i + 1] - C[i]) / (3 * H[i]))
74          B.append((A[i + 1] - A[i]) / H[i] - H[i] * (2 * C[i] + C[i + 1]) / 3)
75      for i in range(0, n):
76          drawFunction(X[i], X[i + 1], A[i], B[i], C[i], D[i], lineName, col, lwidth, style)
77
```

```

def natural(G, A, H, n):
    G[n - 1][0] = 1
    G[n][n] = 1
    return G


def clamped(G, A, H, n):
    G[n - 1][0] = 2 * H[0]
    G[n - 1][1] = H[0]
    G[n - 1][n + 1] = 3 * (A[1] - A[0]) / H[0] - 3 * func2(X[0])

    G[n][n - 1] = H[n - 1]
    G[n][n] = 2 * H[n - 1]
    G[n][n + 1] = 3 * func2(X[n]) - 3 * (A[n] - A[n - 1]) / H[n - 1]
    return G


def periodic(G, A, H, n):
    G[n - 1][0] = 1
    G[n - 1][n] = -1

    G[n][0] = 2 * H[0]
    G[n][1] = H[0]
    G[n][n - 1] = H[n - 1]
    G[n][n] = 2 * H[n - 1]
    G[n][n + 1] = 3 * (A[1] - A[0]) / H[0] - 3 * (A[n] - A[n - 1]) / H[n - 1]
    return G


def fourth(G, A, H, n):
    G[n - 1][0] = H[1]
    G[n - 1][1] = -H[0] - H[1]
    G[n - 1][2] = H[0]

    G[n][n - 2] = H[n - 1]
    G[n][n - 1] = -H[n - 2] - H[n - 1]
    G[n][n] = H[n - 2]
    return G


def Spline(n):
    H, A, B, C, D = [], [], [], [], []
    for i in range(0, n):
        A.append(Y[i])
        H.append(X[i + 1] - X[i])
    A.append(Y[n])
    G = [[0] * (n + 2) for i in range(n + 1)]
```

```
126     for j in range(1, n):
127         G[j - 1][j - 1] = H[j - 1]
128         G[j - 1][j] = 2 * (H[j - 1] + H[j])
129         G[j - 1][j + 1] = H[j]
130         G[j - 1][n + 1] = 3 * (A[j + 1] - A[j]) / H[j] - 3 * (A[j] - A[j - 1]) / H[j - 1]
131     spline(A, dco(B), C, dco(D), H, natural(dco(G), A, H, n), n,
132                           "Natural Boundary", "blue", 2.0, "dotted")
133     spline(A, dco(B), C, dco(D), H, clamped(dco(G), A, H, n), n,
134                           "Clamped Boundary", "orange", 2.0, "--")
135     spline(A, dco(B), C, dco(D), H, periodic(dco(G), A, H, n), n,
136                           "Periodic Boundary", "green", 2.0, "-.")
137     spline(A, dco(B), C, dco(D), H, fourth(dco(G), A, H, n), n,
138                           "Fourth Boundary", "red", 2.0, ":")
139
140
141 def main():
142     n = 11
143     X.append(l)
144     X.append(r)
145     for i in range(0, n - 2):
146         X.append(random.uniform(l, r))
147     X.sort()
148     for i in range(0, n):
149         Y.append(func(X[i]))
150     # Draw Point
151     plt.scatter(X, Y, s=50, label="Key Point")
152     Spline(n - 1)
153
154     # draw the original
155     x = np.linspace(l, r, 100)
156     y = np.sin(x)
157     plt.plot(x, y, color="purple", linewidth=2.0, linestyle="-",label="y=sin(x)")
158     plt.legend(loc="upper right")
159
160     plt.tight_layout()
161     plt.savefig('plot3-4.pdf',dpi = 700)
162     plt.show()
163
164
165 if __name__ == "__main__":
166     main()
```

Chapter 5

Numerical Differentiation and Integration

5.1 Experimental Requirements

1) Master the function of Taylor expansion that construct the difference quotient approximate calculation format of the first and second derivatives including forward difference, backward difference and central difference and its error estimation.

2) Comprehend the method of constructing numerical difference format using Lagrange interpolation polynomial and its error estimation.

3) Understand and grasp:

   · The definition of numerical integration.

   · Basic process of constructing numerical integral calculation format using Lagrange polynomial approximation, common calculation formats including trapezoid, Simpson and medium rectangle, estimation of error and algebraic precision.

   · Composite trapezoid, Simpson and its estimation of error.

   · Successive semi-integral method based on error control.

5.2 Homework

5.2.1 Problem 1

Problem Description

Derive the composite trapezoidal formula and its error estimate, and derive the adaptive quadrature formula and its error estimate based on error control.

Derivation of Composite Trapezoidal Formula

According to the Trapezoidal Rule when $n = 1$, we could attain the formula as follows.

$$\int_{x_0}^{x_1} f(x)dx = \frac{h}{2}[f(x_0) + f(x_1)] - \frac{h^3}{12}f''(\xi)$$

Then let $f \in C^2[a,b], h = (b-a)/n$, and $x_j = a + jh$ for each $j = 0, 1, ..., n$. Therefore, we could derive the formula as follows.

$$\begin{aligned}\int_a^b f(x)dx &= \sum_{j=1}^{n} \int_{x_{j-1}}^{x_j} f(x)dx \\ &= \sum_{j=1}^{n} \{\frac{h}{2}[f(x_{j-1}) + f(x_j)] - \frac{h^3}{12}f''(\xi_j)\} \\ &= \frac{h}{2}[f(x_0) + 2\sum_{j=1}^{n-1} f(x_j) + f(x_n)] - \frac{h^3}{12}\sum_{j=1}^{n} f''(\xi_j)\end{aligned}$$

where $\xi_j \in (x_{j-1}, x_j)$.

The error associated with this approximation is

$$E(f) = -\frac{h^3}{12}\sum_{j=1}^{n} f''(\xi_j)$$

where $\xi_j \in (x_{j-1}, x_j)$. Because $f \in C^2[a,b]$, according to the Extreme Value Theorem,

$$\min_{x\in[a,b]} f''(x) \le f''(\xi_j) \le \max_{x\in[a,b]} f''(x)$$

holds. Thus we could derive the formula continually as follows.

$$n \min_{x\in[a,b]} f''(x) \le \sum_{j=1}^{n} f''(\xi_j) \le n \max_{x\in[a,b]} f''(x)$$

$$\min_{x\in[a,b]} f''(x) \leq \frac{1}{n}\sum_{j=1}^{n} f''(\xi_j) \leq \max_{x\in[a,b]} f''(x)$$

According to the Intermediate Value Theorem and the formula derived above, there is a $\mu \in (a,b)$ such that

$$f''(\mu) = \frac{1}{n}\sum_{j=1}^{n} f''(\xi_j).$$

Hence,

$$E(f) = -\frac{h^3}{12}\sum_{j=1}^{n} f''(\xi_j) = -\frac{h^3}{12} * \frac{1}{n} f''(\mu) = -\frac{b-a}{12}h^2 f''(\mu)$$

holds. Consequently, the result of Composite Trapezoidal Formula describes below.

$$\int_a^b f(x)dx = \frac{h}{2}[f(a) + 2\sum_{j=1}^{n-1} f(x_j) + f(b)] - \frac{b-a}{12}h^2 f''(\mu), \mu \in (a,b)$$

Derivation of Adaptive Quadrature Formula

Take $n = 2$ and $n = 4$ for example, let $n = 2$, thus $h_1 = h = \frac{b-a}{2}$ and the approximation formula describes below.

$$\int_a^b f(x)dx = S(a,b) - \frac{h^5}{90}f^4(u), u \in (a,b)$$

$$S(a,b) = \frac{h}{3}[f(a) + 4f(a+h) + f(b)]$$

In addition, let $n = 4$, thus $h_2 = \frac{b-a}{4} = \frac{h}{2}$ and the approximation formula demonstrates below.

$$\begin{aligned}\int_a^b f(x)dx &= \frac{h}{6}[f(a) + 2f(a+h) + 4f(a+\frac{h}{2}) + 4f(a+\frac{3h}{2} + f(b))] - (\frac{h}{2})^4\frac{b-a}{180}f^{(4)}(\hat{\mu}) \\ &= S(a, \frac{a+b}{2}) + S(\frac{a+b}{2}, b) - \frac{1}{16}(\frac{h^5}{90})f^{(4)}(\hat{\mu})\end{aligned}$$

The error estimation is derived by assuming that $\mu \approx \hat{\mu}$ or, more precisely, $f^{(4)}(\mu) \approx f^{(4)}(\hat{\mu})$. Furthermore, the correctness of this technique depends on the accuracy of this assumption. Consequently, if it is accurate, then the equation

below holds.

$$S(a, \frac{a+b}{2}) + S(a, \frac{a+b}{2}) - \frac{1}{16}(\frac{h^5}{90})f^{(4)}(\mu) \approx S(a,b) - \frac{h^5}{90}f^{(4)}(\mu)$$

$$\frac{h^5}{90}f^{(4)}(\mu) \approx \frac{16}{15}[S(a,b) - S(a, \frac{a+b}{2}) - S(\frac{a+b}{2}, b)]$$

.

Simplify the formula continually, we could acquire the formula below.

$$|\int_a^b f(x)dx - S(a, \frac{a+b}{2}) - S(\frac{a+b}{2}, b)| \approx \frac{1}{15}|S(a,b) - S(a, \frac{a+b}{2}) - S(\frac{a+b}{2}, b)|$$

The formula above means that the formula in $n = 4$ approximates $\int_a^b f(x)dx$ about 15 times better than the value $S(a,b)$. Thus, if

$$|S(a,b) - S(a, \frac{a+b}{2}) - S(\frac{a+b}{2}, b)| < 15\epsilon,$$

we are likely to acquire the result

$$|\int_a^b f(x)dx - S(a, \frac{a+b}{2}) - S(\frac{a+b}{2}, b)| < \epsilon,$$

and the formula $S(a, \frac{a+b}{2}) + S(\frac{a+b}{2}, b)$ is assumed to a sufficiently accurate approximation to $\int_a^b f(x)dx$.

5.2.2 Problem 2

Problem Description

Let $h = (b-a)/3, x_0 = a, x_1 = a + h, x_2 = b$. Find the degree of precision of the quadrature formula.

$$\int_a^b f(x)dx = \frac{9}{4}hf(x_1) + \frac{3}{4}hf(x_2)$$

Related Theorem

The degree of accuracy, or precision, of a quadrature formula is the largest positive integer n such that the formula is exact for x_k, for each $k = 0, 1, ..., n$.

Solution

For $n = 1$, $f(x) = x^1$, then

$$\int_a^b f(x)dx = \int_a^b xdx = \frac{x^2}{2}\Big|_a^b = \frac{b^2 - a^2}{2}.$$

And

$$\frac{9}{4}hf(x_1) + \frac{3}{4}hf(x_2) = \frac{9}{4}h(a+h) + \frac{3}{4}hb = \frac{b^2 - a^2}{2}.$$

Therefore,

$$\int_a^b f(x)dx = \frac{9}{4}hf(x_1) + \frac{3}{4}hf(x_2)$$

for $n = 1$.

For $n = 2$, $f(x) = x^2$, then

$$\int_a^b f(x)dx = \int_a^b xdx = \frac{x^3}{3}\Big|_a^b = \frac{b^3 - a^3}{3}.$$

And

$$\frac{9}{4}hf(x_1) + \frac{3}{4}hf(x_2) = \frac{9}{4}h(a+h)^2 + \frac{3}{4}hb^2 = \frac{b^3 - a^3}{3}.$$

Therefore,

$$\int_a^b f(x)dx = \frac{9}{4}hf(x_1) + \frac{3}{4}hf(x_2)$$

for $n = 2$.

For $n = 3$, $f(x) = x^3$, then

$$\int_a^b f(x)dx = \int_a^b xdx = \frac{x^4}{4}\Big|_a^b = \frac{b^4 - a^4}{4}.$$

And

$$\frac{9}{4}hf(x_1) + \frac{3}{4}hf(x_2) = \frac{9}{4}h(a+h)^3 + \frac{3}{4}hb^3 \neq \frac{b^4 - a^4}{4}.$$

Therefore,

$$\int_a^b f(x)dx \neq \frac{9}{4}hf(x_1) + \frac{3}{4}hf(x_2)$$

for $n = 3$.

Consequently, the degree of precision of this quadrature formula is 2.

5.3 Experimental Assignments

5.3.1 Task 1

Problem Description

1) Write a program for a composite trapezoidal formula and Simpson formula.

2) Let $h = 0.01$ and calculate definite integral

$$I(f) = \frac{1}{\sqrt{2\pi}} \int_0^1 exp - \frac{x^2}{2} dx$$

using compound trapezoid and Simpson formula. Then compare the result with the exact solution, and explain the advantages and disadvantages of the two formats.

3) If the calculation precision is 10^{-4}, calculate the value of h and n.

Theorem of Composite Trapezoidal Rule

Let $f \in C^2[a,b], h = \frac{b-a}{n}$, and $x_j = a + jh$ for each $j = 0, 1, ..., n$.

There exists a $\mu \in (a,b)$ for which the Composite Trapezoidal Rule for n subintervals can be written with its term as

$$\int_a^b f(x)dx = \frac{h}{2}[f(a) + 2\sum_{j=1}^{n-1} f(x_j) + f(b)] - \frac{b-a}{12}h^2 f''(u).$$

Theorem of Composite Simpson's Rule

Let $f \in C^4[a,b], n$ be even. $h = \frac{b-a}{n}$, and $x_j = a + jh$ for each $j = 0, 1, .., n$.

There exists a $\mu \in (a,b)$ for which the Composite Simpson's Rule for n subintervals can be written with its error term as

$$\int_a^b f(x)dx = \frac{h}{3}[f(a) + 2\sum_{j=1}^{\frac{n}{2}-1} f(x_{2j}) + 4\sum_{j=1}^{\frac{n}{2}} f(x_{2j-1}) + f(b)] - \frac{b-a}{180}h^4 f^{(4)}(\mu).$$

Solution

$$I(f) = \frac{erf(\frac{1}{\sqrt{2}})}{2}$$

where

$$erf(z) \equiv \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt.$$

Therefore, we have no idea to simplify the equation $I(f)$ so that we use program to acquire the exact solution $ans = 0.3413447460685429$.

We choose several h for the integration method and obtain a table about the result of these two methods and the error with the exact solution. In the Table 7.1, we could figure out that the less value of h is, the more accurate result will be. What's more, in this integration equation, Composite Trapezoidal method behave better than Simpson's. When h equals to 0.05, the Δ of T reaches the error less than 10^{-4}, while the Δ of S reaches the error less than 10^{-4} until $h = 0.001$.

Table 5.1: Composite Trapezoidal and Simpson's Method

h	n	*ans* of T	Δ of T	*ans* of S	Δ of S
0.001	1000	0.3413447259	0.0000000202	0.3406089847	0.0007357614
0.002	500	0.3413446654	0.0000000807	0.3398708668	0.0014738793
0.003	333	0.3411021099	0.0002426361	0.3381499337	0.0031948124
0.004	250	0.3413444234	0.0000003226	0.3383875854	0.0029571606
0.005	200	0.3413442420	0.0000005041	0.3376424342	0.0037023119
0.01	100	0.3413427296	0.0000020164	0.3338818266	0.0074629194
0.02	50	0.3413366803	0.0000080658	0.3261890722	0.0151556739
0.03	33	0.3388584996	0.0024862465	0.3073440611	0.0340006850
0.04	25	0.3413124816	0.0000322645	0.2988677631	0.0424769830
0.05	20	0.3412943313	0.0000504148	0.3018099833	0.0395347627

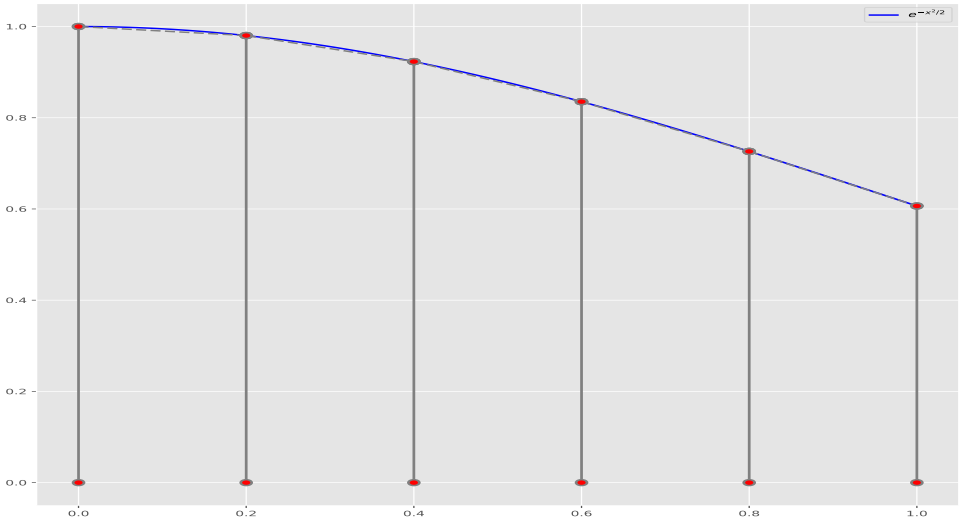

Figure 5.1 Integration Method with $n = 5$

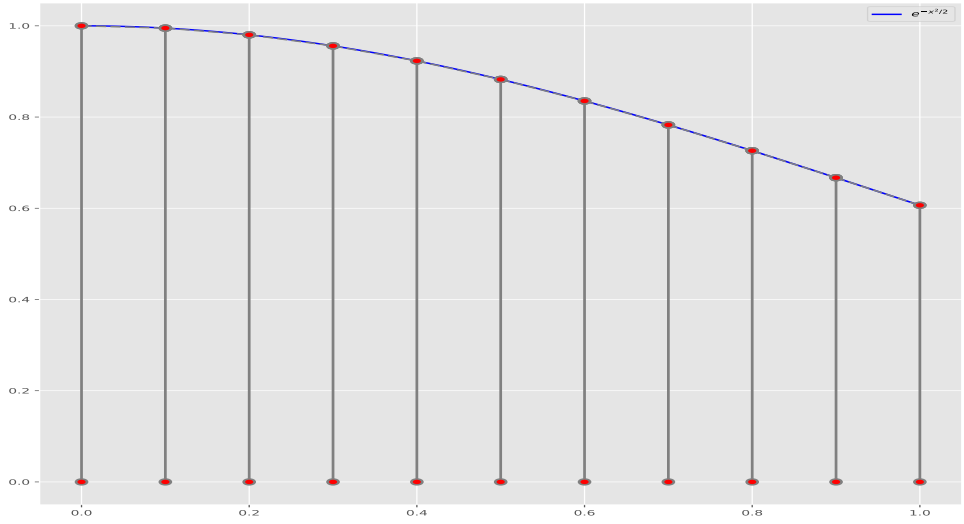

Figure 5.2 Integration Method with $n = 10$

Code

```python
import numpy as np


def func(x):
    return np.exp(-x * x / 2)


def trapezoidal(a, b, h):
    n, ans = (b - a) / h, func(a) + func(b)
    for i in range(1, int(n)):
        ans += 2 * func(a + i * h)
    ans *= h / 2
    ans /= np.sqrt(2 * np.pi)
    print("trapezoidal's ans:", ans)


def simpson(a, b, h):
    n, ans = (b - a) / h, func(a) + func(b)
    for i in range(1, int(n / 2) - 1):
        ans += 2 * func(a + 2 * i * h) + 4 * func(a + (2 * i - 1) * h)
    ans += 4 * func(a + (2 * (n - 1) - 1) * h)
    ans *= h / 3
    ans /= np.sqrt(2 * np.pi)
    print("simpson's ans:", ans)
```

Chapter 6

Ordinary Differential Equations

6.1 Experimental Requirements

1) Master the basic process or steps to obtain the numerical solution of the differential equation, and the existence and uniqueness of the solution.

2) Grasp three common methods for discretizing differential equations into numerical calculation formats: Taylor method, difference quotient method and integral method.

3) Comprehend the basic definitions of system error, truncation error, local error, and overall error.

4) Master common Euler methods (explicit, implicit), trapezoidal format, estimated correction format and Runge Kuta fourth-order format.

5) Understand the construction method of numerical calculation format of higher-order ordinary differential equation.

6) Realize the multi-step format construction process.

6.2 Experimental Assignments

6.2.1 Task 1

Problem Description

Apply the Euler explicit format, trapezoidal estimation correction format and 4-level Runge Kuta format to find the solution of the equation below.

$$y' = 1 + y^2, y(0) = 0$$

In addition, analyze the convergence of the three formats compared with the analytical solution.

The Uniqueness and Existence of ODE

Suppose that

$$D = (t, y)|a \leq t \leq b, -\infty < y < \infty$$

ans that $f(t, y)$ is continuous on D.

If f satisfies a Lipschitz condition on D in the variable y, then the initial-value problem

$$\begin{cases} y'(t) = f(t, y), a < t \leq b \\ y(a) = \alpha \end{cases}$$

has a unique solution $y(t)$ for $a \leq t \leq b$.

Euler's Method

Euler's Method apply the forward difference equation and the Taylor's Theorem to simplify the equations in its solution. In addition, it is a common method to design the numerical calculation format of differential equations by using Taylor expansion whose characteristic is that it could simultaneously obtain its truncation error estimate while designing the algorithm format.

$$\begin{cases} y'(t) = f(t, y), y(a) = \alpha \\ y(t_1) = y(t_0) + f(t_0, y(t_0))(t_1 - t_0) + \dfrac{y''(\xi_1)(t_1 - t_0)^2}{2} \end{cases}$$

Pseudo Code of Euler's Method

INPUT: Endpoints a, b; Integer N; Initial condition α.

OUTPUT: Approximation w to y at $N + 1$ points of t.

Step 1: Set $h = \dfrac{b - a}{N}; t = a; w = a$; Output t, w.

Step 2: For $i = 1, 2, ..., N$, do Steps $3 - 4$.

- Step 3: Set $w = w + h * f(t, w), t = a + ih$.
- Step 4: Output (t, w).

Step 5: Stop.

Modified Euler's Method

According to the equation below, we could modify the Euler's Method.

$$\begin{aligned} y(t_{j+1} - y(t_j)) &= \int_{t_j}^{t_{j+1}} dy = \int_{t_j}^{t_{j+1}} f(t,y)dt \\ &\approx \frac{h}{2}[f(t_j, y(t_j)) + f(t_{j+1}, y(t_{j+1}))] \end{aligned}$$

Therefore, in this part, we could use the Trapezoidal formula to modify the original Euler's Method which means change the original equation

$$y_{i+1} = y_i + f(t_i, y(t_i))$$

into Trapezoidal Estimation Correction format

$$y_{i+1} = y_i + \frac{h}{2}[f(t_i, y_i) + f(t_{i+1}, y_i + hf(t_i, y_i))]$$

that may more accurate than the original.

Runge-Kutta Method of Order Four

The Runge-Kutta Method is a new method with higher-order error, without repeated computing the higher-order derivatives of function $f(t,y)$, its general form is

$$y_{n+1} = y_n + \sum_{i=1}^{N} c_i K_i$$

Therefore, in the Runger-Kutta Method of Order Four, the equations

$$\begin{cases} y(0) = \alpha \\ y(t_{i+1}) = y(t_i) + \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4) \end{cases}$$

holds, where

$$\begin{cases} K_1 = hf(t_i, y_i) \\ K_2 = hf(t_i + \frac{h}{2} + y_i + \frac{1}{2}K_1) \\ K_3 = hf(t_i + \frac{h}{2}, y_i + \frac{1}{2}K_2) \\ K_4 = hf(t_{i+1}, y_i + K_3) \end{cases}$$

for each $i = 1, 2, ..., N-1$.

Solution

The Ordinary Differential Equation described in problem is a common equation which possess a analytical solution $y = tan(x + C)$. In this problem, $C = 0$.

We set the initial interval to be $[0, 1.5]$ and choose $n = 5, 10, 20, 50$ to observe the convergence of the Euler's Method. According to the result of Euler's Method, we could figure out that larger n is, more accurate the result will be.

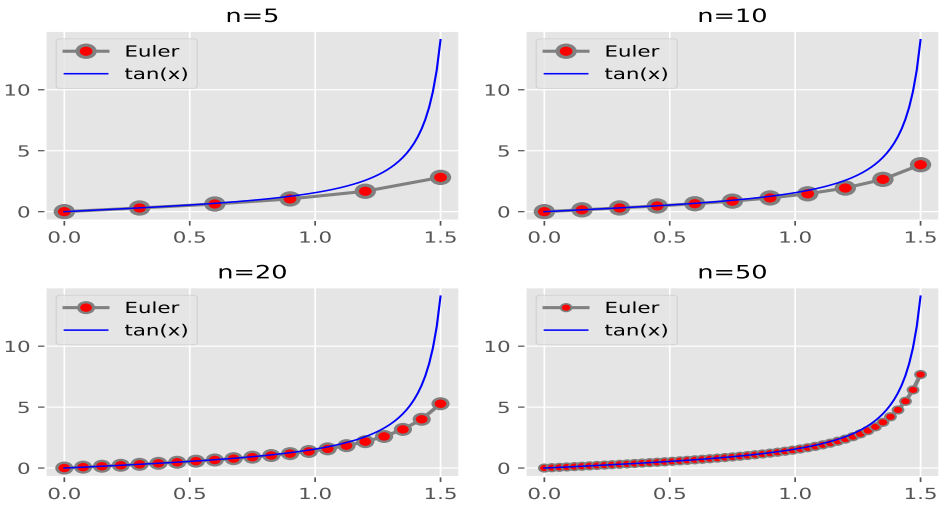

Figure 6.1 Euler's Method

We continually run the program of Modified Euler's Method to observe the change of result. According to the Figure 6.2, we could obviously find that the Modified Euler's Method possess a more accurate approximation to the analytical solution than the original.

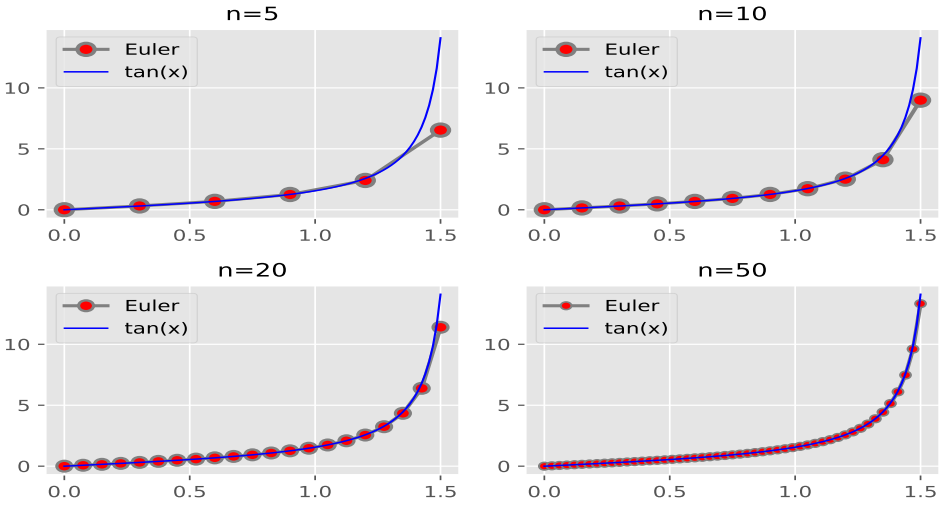

Figure 6.2 Modified Euler's Method

Besides, we apply the Runge-Kutta Method of Order Four to this equation and obtain a more accurate approximation result compared with the other two

methods. According to the Figure 6.3, it is apparent that the result of Runge-Kutta Method of Order Four is more approximate to the analytical solution even when $n = 5$.

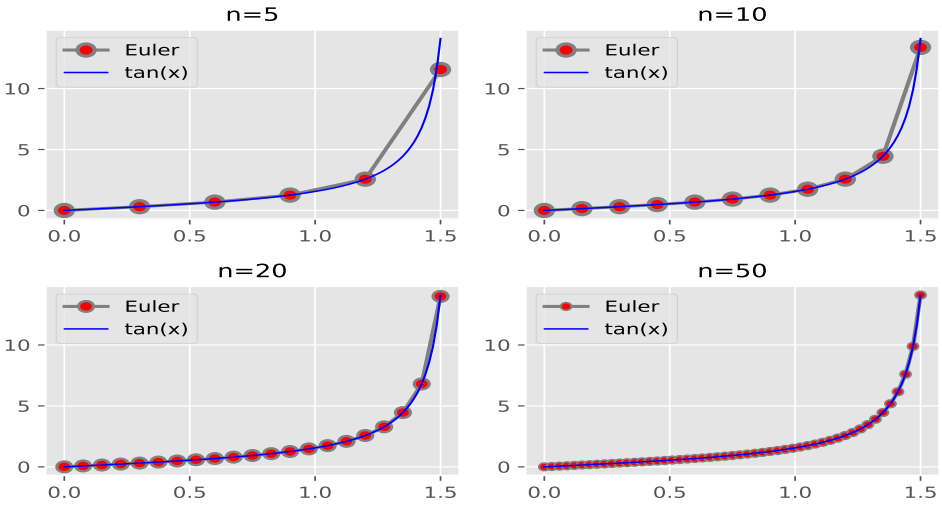

Figure 6.3 Runge-Kutta Method of Order Four

Furthermore, we take $n = 5$ for example to compare the convergence between the three methods. In the Table 7.1, E denotes Euler's Method, ME denotes Modified Euler's Method and R denotes the Runge-Kutta Method of Order Four. According to the Table 7.1, it is obvious that the Runge-Kutta Method of Order Four behaves better than the other methods and the Modified Euler's Method displays better than the Euler's Method in this problem. What's more, according to the Figure 6.4, it is apparent that the gap of effects between the three methods starts to appear after the third point, and the gap reached the maximum when it comes to the 5th point.

Table 6.1: The Difference of Convergence Between the Three Methods

t_i	y_i of E	Δ of E	y_i of ME	Δ of ME	y_i of R	Δ of R
0.3	0.300000	0.009336	0.313500	0.004163	0.309316	0.000019
0.6	0.627000	0.057136	0.690256	0.006119	0.684109	0.000026
0.9	1.044938	0.215219	1.254343	0.005814	1.260051	0.000107
1.2	1.672507	0.899643	2.406269	0.165882	2.566845	0.005306
1.5	2.811692	11.28972	6.536238	7.565181	11.56446	2.536958

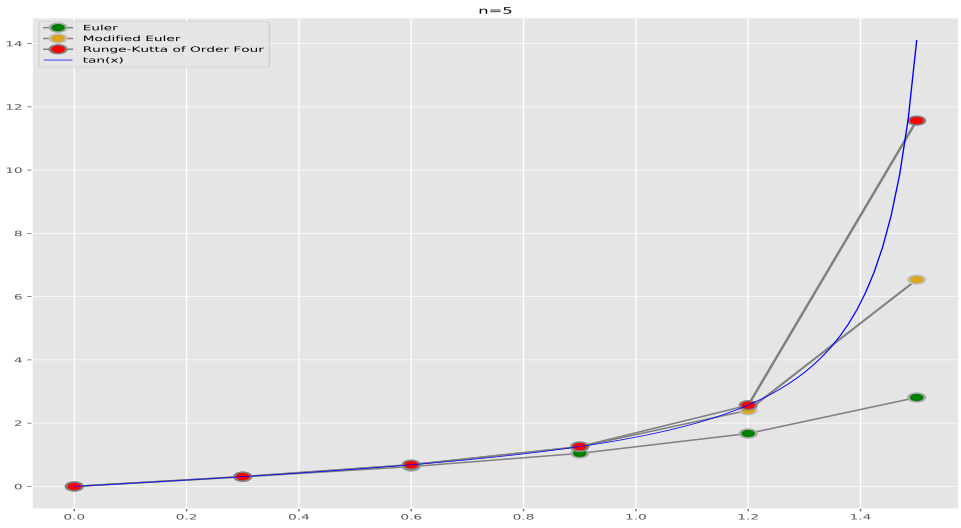

Figure 6.4 Difference Between Three Methods

Code

```
def func(y):
    return 1 + y * y


def Euler(a, b, n, y0):
    X, Y = [], []
    X.append(0)
    Y.append(y0)
    h = (b - a) / n
    for i in range(1, n + 1):
        Y.append(Y[i - 1] + h * func(Y[i - 1]))
        X.append(a + i * h)
        print(X[i], Y[i])


def Modifiedeuler(a, b, n, y0):
    X, Y = [], []
    X.append(0)
    Y.append(y0)
    h = (b - a) / n
    for i in range(1, n + 1):
        tmp = Y[i - 1] + h * func(Y[i - 1])
        Y.append(Y[i - 1] + h * (func(Y[i - 1]) + func(tmp)) / 2)
        X.append(a + i * h)
        print(X[i], Y[i])


def RungeKutta(a, b, n, y0):
    X, Y = [], []
```

```
30      X.append(0)
31      Y.append(y0)
32      h = (b - a) / n
33      for i in range(1, n + 1):
34          K1 = h * func(Y[i - 1])
35          K2 = h * func(Y[i - 1] + K1 / 2)
36          K3 = h * func(Y[i - 1] + K2 / 2)
37          K4 = h * func(Y[i - 1] + K3)
38          Y.append(Y[i - 1] + (K1 + 2 * K2 + 2 * K3 + K4) / 6)
39          X.append(a + i * h)
40          print(X[i], Y[i])
41
42
43  def main():
44      l, r = 0, 1.5
45      Euler(l, r, 5, 0)
46      Modifiedeuler(l, r, 5, 0)
47      RungeKutta(l, r, 5, 0)
48
49
50  if __name__ == "__main__":
51      main()
```

6.2.2 Task 2

Problem Description

Runge-Kutta 4th Order Method for solving classical van der Pol Differential Equations descriptive of oscillators.

$$\begin{cases} \dfrac{d^2y}{dt^2} - \mu(1-y^2)\dfrac{dy}{dt} + y = 0, \\ y(0) = 1, y'(0) = 0. \end{cases}$$

Set $u = 0.01, 0.1, 1$, respectively, and compare the calculation results by plotting.

Solution of Higher Order Differential Equation

Generally, the solution of higher-order differential equations always could be reduced to the first-order equations. Take the second order ordinary differential equation for example.

$$\begin{cases} y''(x) = f(x, y, y'), x_0 \le x \le x_n \\ y(x_0) = y_0, y'(x_0) = z_0 \end{cases}$$

Let $z = y'$, the second order ordinary differential equation could be derived to

$$\begin{cases} y'(x) = z, x_0 \leq x \leq x_n \\ z'(x) = f(x, y, z) \\ y(x_0) = y_0, z(x_0) = z_0 \end{cases}$$

That is, we could convert second-order ordinary differential equations into first-order ordinary differential equations for calculation.

Solution

According to the solution of Higher Order Differential Equation, we could simplify the original problem into

$$\begin{cases} y' = z \\ z' = \mu(1 - y * y)z - y \\ y(0) = 1, z(0) = 0 \end{cases}$$

Thus, we could let $\mu = 0.01, 0.05, 0.1, 1$ to plot the answers as follows.

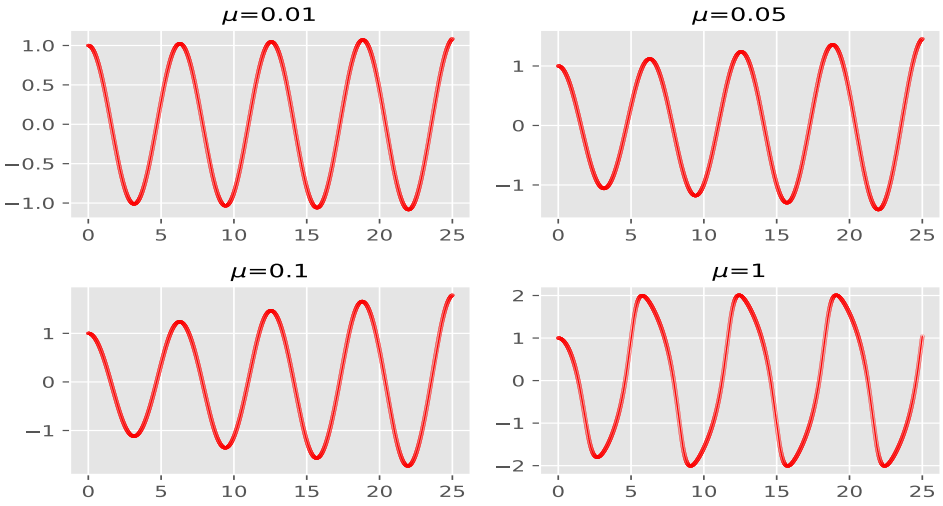

Figure 6.5 $n = 2500$ in $[0, 30]$

Furthermore, we could continually plot the four results together to make a more intuitive comparison. According to the Figure 6.6, we could conclude that the larger value of u is, the sharper the image is.

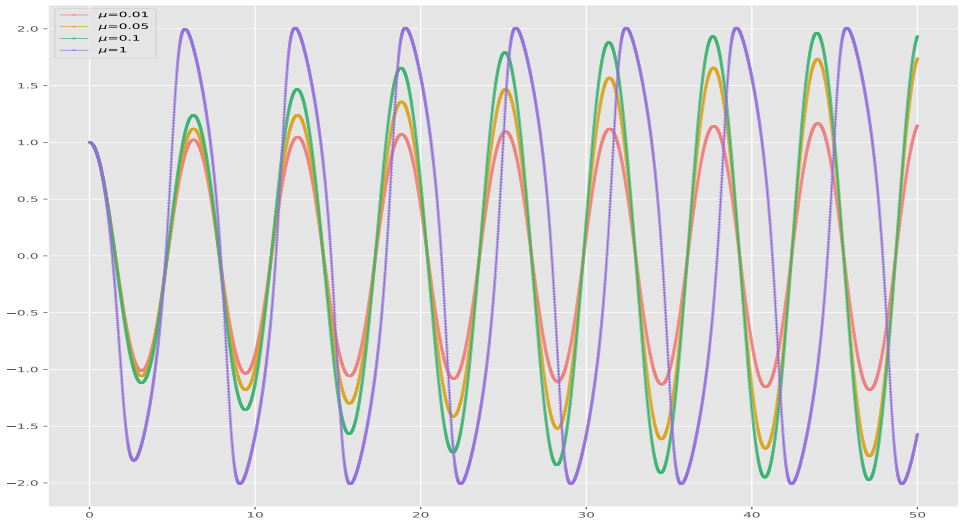

Figure 6.6 $n = 5000$ in $[0, 50]$ with the four values of μ

Code

```
import math
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import style

style.use('ggplot')
plt.figure(figsize=(10, 10), dpi=80)
style_list = ["g+-", "r--", "b.-", "yo-"]


def fz(u, y, z):
    return u * (1 - y * y) * z - y


def fy(z):
    return z


def RungeKutta(a, b, n, y0, z0, u, color, top):
    X, Y, Z = [], [], []
    X.append(0)
    Y.append(y0)
    Z.append(z0)
    h = (b - a) / n
    for i in range(1, n + 1):
        K11 = h * fy(Z[i - 1])
        K21 = h * fz(u, Y[i - 1], Z[i - 1])

        K12 = h * fy(Z[i - 1] + K21 / 2)
```

```
30          K22 = h * fz(u, Y[i - 1] + K11 / 2, Z[i - 1] + K21 / 2)
31
32          K13 = h * fy(Z[i - 1] + K22 / 2)
33          K23 = h * fz(u, Y[i - 1] + K12 / 2, Z[i - 1] + K22 / 2)
34
35          K14 = h * fy(Z[i - 1] + K23)
36          K24 = h * fz(u, Y[i - 1] + K13, Z[i - 1] + K23)
37
38          Y.append(Y[i - 1] + (K11 + 2 * K12 + 2 * K13 + K14) / 6)
39          Z.append(Z[i - 1] + (K21 + 2 * K22 + 2 * K23 + K24) / 6)
40          X.append(a + i * h)
41
42      plt.plot(X, Y, '-p', color=color,
43               marker='o',
44               markersize=0.5, linewidth=0.5,
45               markerfacecolor=color,
46               markeredgecolor=color,
47               label=r"$\mu$=" + top,
48               markeredgewidth=2)
49
50
51  def main():
52      l, r = 0, 50
53      RungeKutta(l, r, 5000, 1, 0, 0.01, "lightcoral", "0.01")
54      RungeKutta(l, r, 5000, 1, 0, 0.05, "goldenrod", "0.05")
55      RungeKutta(l, r, 5000, 1, 0, 0.1, "mediumseagreen", "0.1")
56      RungeKutta(l, r, 5000, 1, 0, 1, "mediumpurple", "1")
57
58      plt.legend(loc="best")
59      plt.tight_layout()
60      plt.savefig('plot5-6.pdf', dpi=700)
61      plt.show()
62
63
64  if __name__ == "__main__":
65      main()
```

Chapter 7

Approximation Theory

7.1 Experimental Requirements

1) Master the general form of line fitting.

2) Grasp the application of regular equations in line fitting.

3) Comprehend the general form of Discrete Least Square Rule.

4) Understand the Orthogonal Polynomials and Least Square Approximation.

7.2 Experimental Assignments

7.2.1 Task 1

Problem Description

According to the observed data shown below, find a quadratic polynomial to fit it, try to display its least squares fit model, and give its system of regular equations and solution.

x	-2	-1	0	1	2
$f(x)$	0	1	2	1	0

The General Form of Discrete Least Square Rule

The general problem of approximating a set of data

$$\{(x_i, y_i)|i = 1, 2, ..., m\},$$

with an algebraic polynomial

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$$

of degree $n < m - 1$.

The General Form of Discrete Least Square Rule

$$\min_{a_0,a_1,...,a_n} E = \sum_{i=1}^{m}(y_i - p_n(x_i))^2 \quad = \sum_{i=1}^{m}(y_i - \sum_{k=0}^{n} a_k x_i^k)^2.$$

To find the suitable parameters $a_0, a_1, ..., a_n$, such that E gets to be minimized. Let

$$0 = \frac{\partial E}{\partial a_j} = -2\sum_{i=1}^{m} y_i x_i^j + 2\sum_{k=0}^{n} a_k \sum_{i=1}^{m} x_i^{j+k}.$$

for each $j = 0, 1, ..., n$.

Then we could acquire $n+1$ normal equations in the $n+1$ unknown parameters a_j with $j = 0, 1, ..., n$.

$$\sum_{k=0}^{n} a_k \sum_{i=1}^{m} x_i^{j+k} = \sum_{i=1}^{m} y_i x_i^j,$$

for each $j = 0, 1, ..., n$. In order to simplify the equations, denote

$$R = \begin{pmatrix} x_1^n & x_1^{n-1} & \cdots & x_1 & 1 \\ x_2^n & x_2^{n-1} & \cdots & x_2 & 1 \\ x_3^n & x_3^{n-1} & \cdots & x_3 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_m^n & x_m^{n-1} & \cdots & x_m & 1 \end{pmatrix}, a = \begin{pmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_1 \\ a_0 \end{pmatrix}, y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{m-1} \\ y_m \end{pmatrix}.$$

Therefore, the normal equations above could be written as

$$R^T R a = R^T y.$$

Solution

According to the General Form of Discrete Least Square Rule, we could obtain vector x and y form observed data as follows.

$$x = \begin{pmatrix} -2 \\ -1 \\ 0 \\ 1 \\ 2 \end{pmatrix}, y = \begin{pmatrix} 0 \\ 1 \\ 2 \\ 1 \\ 0 \end{pmatrix}, R = \begin{pmatrix} 4 & -2 & 1 \\ 1 & -1 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 4 & 2 & 1 \end{pmatrix}, a = \begin{pmatrix} a_2 \\ a_1 \\ a_0 \end{pmatrix}, R^T R a = R^T y$$

After programming, we could acquire the answer of approximate Polynomial

of order two.

$$a = \begin{pmatrix} -0.428571 \\ 0.000000 \\ 1.657143 \end{pmatrix}, P_2(x) = a_2x^2 + a_1x^1 + a_0$$

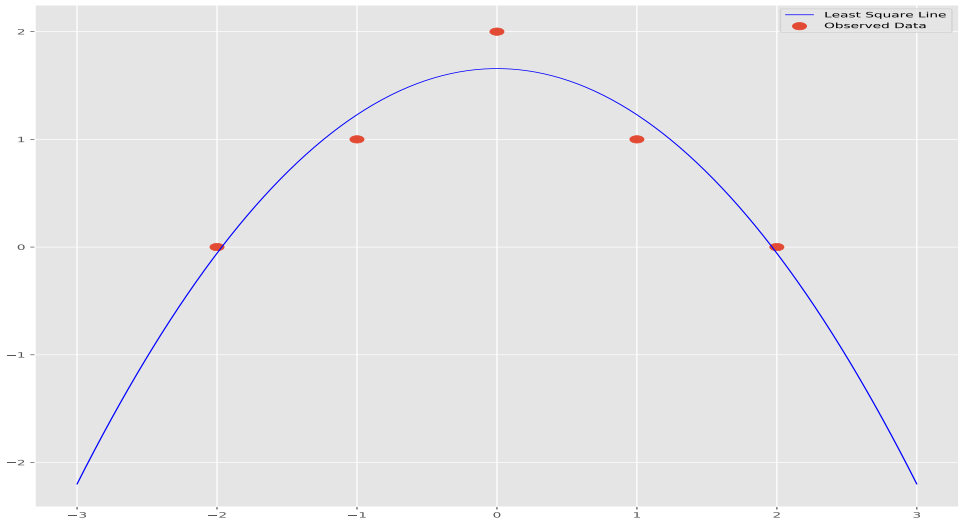

Figure 7.1 Approximate Polynomial of Order Two

Due to the *size* of observed data is equal to five, we could continually acquire the approximate Polynomials of order three and four as follows. According to the Figure 7.2, we could find that $P_2(x)$ is same as $P_3(x)$ mainly for the reason that $a_3 = 0$ for $P_3(x)$. What's more, the approximation polynomial of order four fits best.

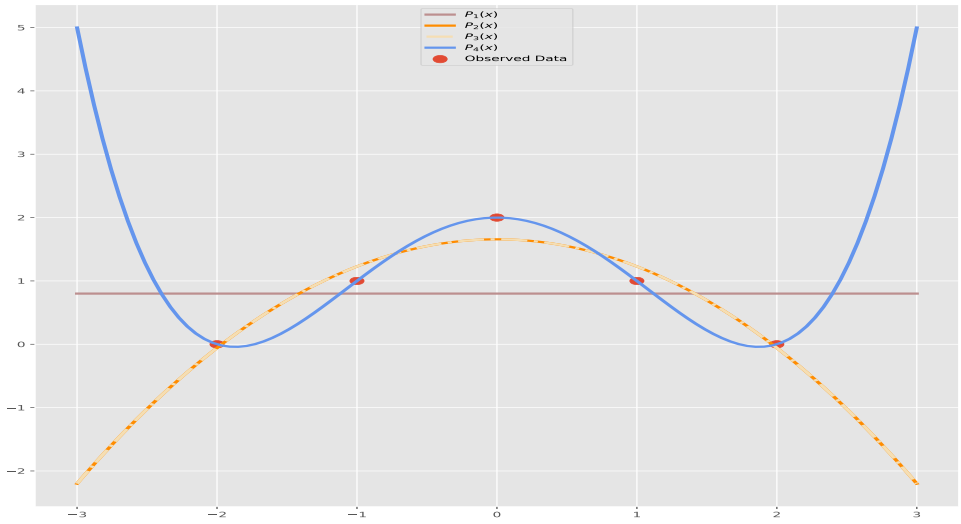

Figure 7.2 Approximate Polynomial of Order One, Two, Three and Four

Table 7.1: Vector a of the Approximation Polynomial of One, Two, Three and Four

i	a_i for $P_1(x)$	a_i for $P_2(x)$	a_i for $P_3(x)$	a_i for $P_4(x)$
a_4	0	0	0	0.166667
a_3	0	0	0	0
a_2	0	-0.428571	-0.428571	-1.166667
a_1	0	0	0	0
a_0	0.8	1.657143	1.657143	1.999999

Code

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import style

style.use('ggplot')
plt.figure(figsize=(10, 10), dpi=80)
style_list = ["g+-", "r--", "b.-", "yo-"]


def matrixT(A):
    n = len(A)
    m = len(A[0])
    B = [[0] * n for i in range(m)]
    for i in range(0, m):
        for j in range(0, n):
            B[i][j] = A[j][i]
    return B


def matrixMul(A, B):
    n = len(A)
    m = len(B[0])
    L = len(B)
    C = [[0] * m for i in range(n)]
    for i in range(0, n):
        for j in range(0, m):
            for k in range(0, L):
                C[i][j] += A[i][k] * B[k][j]
    return C


def Gaussian(A):
    n = len(A)

    for i in range(0, n):
```

```
36          # Search for maximum in this column
37          maxEl = abs(A[i][i])
38          maxRow = i
39          for k in range(i + 1, n):
40              if abs(A[k][i]) > maxEl:
41                  maxEl = abs(A[k][i])
42                  maxRow = k
43
44          # Swap maximum row with current row
45          for k in range(i, n + 1):
46              A[maxRow][k], A[i][k] = A[i][k], A[maxRow][k]
47
48          # Make all rows below this one 0 in current column
49          for k in range(i + 1, n):
50              c = -A[k][i] / A[i][i]
51              for j in range(i, n + 1):
52                  if i == j:
53                      A[k][j] = 0
54                  else:
55                      A[k][j] += c * A[i][j]
56
57      # Solve equation Ax=b for an upper triangular matrix A
58      x = [0 for i in range(n)]
59      for i in range(n - 1, -1, -1):
60          x[i] = A[i][n] / A[i][i]
61          for k in range(i - 1, -1, -1):
62              A[k][n] -= A[k][i] * x[i]
63      return x
64
65
66  def leastSquare(X, Y, n):
67      m = len(X)
68      R = [[0] * (n + 1) for i in range(m)]
69      for i in range(0, m):
70          R[i][n] = 1
71          for j in range(n - 1, -1, -1):
72              R[i][j] = R[i][j + 1] * X[i]
73      A = matrixMul(matrixT(R), R)
74      b = matrixMul(matrixT(R), Y)
75      for i in range(0, n + 1):
76          A[i].append(b[i][0])
77      return Gaussian(A)
78
79
80  def main():
81      n = 2
82      X = [-2, -1, 0, 1, 2]
83      Y = [[0], [1], [2], [1], [0]]
```

```
84
85      ans = leastSquare(X, Y, n)
86      X2 = np.linspace(-3, 3, 100)
87      Y2 = ans[0] * X2 * X2 + ans[1] * X2 + ans[2]
88      plt.plot(X2, Y2, color="blue", linewidth=3.0, linestyle="-", label=r"$P_2(x)$")
89
90      Y1 = [0, 1, 2, 1, 0]
91      plt.scatter(X, Y1, s=100, label="Observed Data")
92      plt.legend(loc="best")
93      plt.tight_layout()
94      plt.show()
95
96
97  if __name__ == "__main__":
98      main()
```

Chapter 8

Eigenvalues and Eigenvectors

8.1 Experimental Requirements

1) Master the basic concepts including norm of vector, norm of matrix, convergence of vector sequences and convergence matrix.

2) Grasp the fundamental notion including spectral radius and disk theorem.

3) Comprehend the similarities and differences of power method, inverse power method, and symmetric power method.

4) Understand the related theory of Householder transformation, Givens rotation transformation and QR decomposition of matrix.

8.2 Experimental Assignments

8.2.1 Task 1

Problem Description

Apply the power method, symmetric power method, and inverse power method on the symmetrical matrix A which possesses three different eigenvalues,

$$\lambda_1 = 6, \lambda_2 = 3, \lambda_3 = 1$$

to find the maximum eigenvalue and eigenvector, respectively.

$$A = \begin{pmatrix} 4 & -1 & 1 \\ -1 & 3 & -2 \\ 1 & -2 & 3 \end{pmatrix}$$

In addition, the initial vector $x^{(0)}$ could be $(1\ 1\ 1)^T$.

Power Method

Assume that the $n * n$ matrix A has n eigenvalues

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq |\lambda_n|$$

with an associated collection of linearly independent eigenvectors

$$\{v^{(1)}, v^{(2)}, v^{(3)}, ..., v^{(n)}\}.$$

If x is any vector in $\mathbb{R}^n$, then constants $\beta_1, \beta_2, ..., \beta_n$ exist with

$$x = \beta_1 v^{(1)} + \beta_2 v^{(2)} + ... + \beta_n v^{(n)} = \sum_{j=1}^{n} \beta_j v^{(j)}.$$

Multiplying both sides of this equation by $A, A^2, ..., A^k, ...$, we obtain

$$\begin{aligned}
Ax &= \sum_{j=1}^{n} \beta_j A v^{(j)} = \sum_{j=1}^{n} \beta_j \lambda_j v^{(j)} \\
A^2 x &= \sum_{j=1}^{n} \beta_j \lambda_j A v^{(j)} = \sum_{j=1}^{n} \beta_j \lambda_j^2 v^{(j)} \\
&\vdots \\
A^k x &= \sum_{j=1}^{n} \beta_j \lambda_j^k v^{(j)} = \lambda_1^k \sum_{j=1}^{n} \beta_j (\frac{\lambda_j}{\lambda_1})^k v^{(j)} \\
&= \lambda_1^k (\beta_1 v^{(1)} + \sum_{j=2}^{n} \beta_j (\frac{\lambda_j}{\lambda_1})^k v^{(j)})
\end{aligned}$$

Since $|\lambda_1| > |\lambda_j|$ for all $j = 2, 3, ..., n$, we have

$$\begin{aligned}
\lim_{k\to\infty} (\frac{\lambda_j}{\lambda_1})^k &= 0, \\
\lim_{k\to\infty} A^k x &= \lim_{k\to\infty} \lambda_1^k \beta_1 v^{(1)}
\end{aligned}$$

Therefore, we acquire the equation to proceed to find λ_1 and an associated eigenvector. However, it may converges to zero if $\lambda < 1$ and diverges if $\lambda_1 > 1$. Thus, we need to scale the powers of $A^k x$ in an appropriate manner to ensure that the limit of the equation if finite and nonzero.

Inverse Power Method

The Inverse Power method is a modification of the Power method that gives

faster convergence. It is used to determine the eigenvalue of A that is closest to a specified number q. Assume that the matrix A has eigenvalues

$$\lambda_1, ..., \lambda_n$$

with linearly indepedent eigenvectors

$$v^{(1)}, v^{(2)}, ..., v^{(n)}.$$

Suppose that $q \neq \lambda_i$ for $i = 1, 2, 3..., n$. We could easily find that the eigenvalues of the matrix $(A - qI)^{-1}$ are

$$\frac{1}{\lambda_1 - q}, \frac{1}{\lambda_2 - q}, , ..., \frac{1}{\lambda_n - q}$$

with eigenvectors

$$v^{(1)}, v^{(2)}, ..., v^{(n)}.$$

Applying the Power method to $(A - qI)^{-1}$ gives

$$y^{(m)} = (A - qI)^{-1} x^{(m-1)}$$

Let

$$\mu^(m) = y^{(m)}_{p_{m-1}} = \frac{y^{(m)}_{p_m - 1}}{x^{(m-1)}_{p_m - 1}} = \frac{\sum_{j=1}^{n} \beta_j \frac{1}{(\lambda_j - q)^m} v^{(j)}_{p_{m-1}}}{\sum_{j=1}^{n} \beta_j \frac{1}{(\lambda_j - q)^{m-1}} v^{(j)}_{p_{m-1}}}$$

Let p_m represents the smallest integer for which

$$|y^{(m)}_{p_m}| = ||y^{(m)}||$$

Define

$$x^{(m)} = \frac{y^{(m)}}{y^{(m)}_{p_m}}$$

The sequence $\{\mu^{(m)}\}$ converges to

$$\frac{1}{|\lambda_k - q|} = \max_{1 \leq j \leq n} \frac{1}{\lambda_i - q}$$

and λ_k is the eigenvalue of A closest to q.

Solution

According to the introduction of Power Method described above, we could

acquire the maximum eigenvalue and eigenvector by setting the $TOL = 10^{-8}$ and the maximal number of iteration is 50.

Table 8.1: The Result of Power Method with $TOL = 10^{-8}, N = 50$

Eigenvalue	Eigenvector	Iterative Times
5.99999998	$[1.0, -0.99999999, 0.99999999]^T$	29

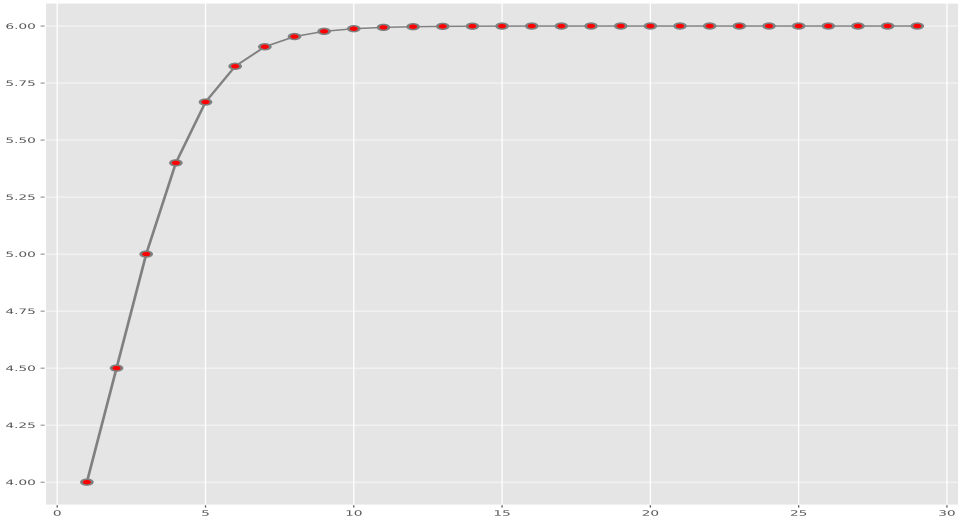

Figure 8.1 Convergence of Power Method

In terms of the application of Symmetric Power Method, we could obtain the maximum eigenvalue and eigenvector by setting the $TOL = 10^{-8}$ and the maximal number of iteration is 50.

Table 8.2: The Result of Symmetric Power Method with $TOL = 10^{-8}, N = 50$

Eigenvalue	Eigenvector	Iterative Times
5.99999999	$[0.57735027, -0.57735027, 0.57735027]^T$	28

According to the introduction of Inverse Power Method described above, we could abtain the maximum eigenvalue and eigenvector by setting the $TOL = 10^{-8}$ and the maximal number of iteration is 50.

Table 8.3: The Result of Inverse Power Method with $TOL = 10^{-8}, N = 50$

Eigenvalue	Eigenvector	Iterative Times
5.99999999	$[-0.50000000, -0.50000000, -0.50000000]^T$	23

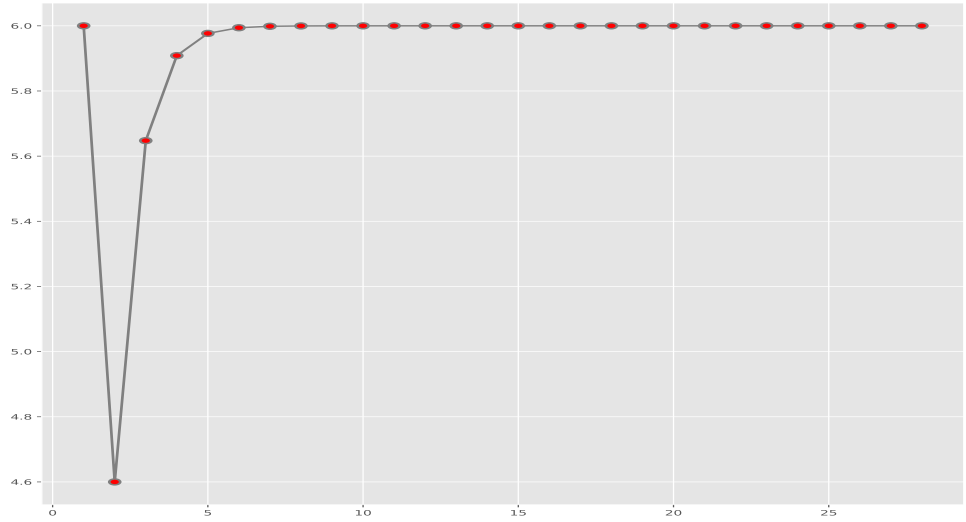

Figure 8.2 Convergence of Symmetric Power Method

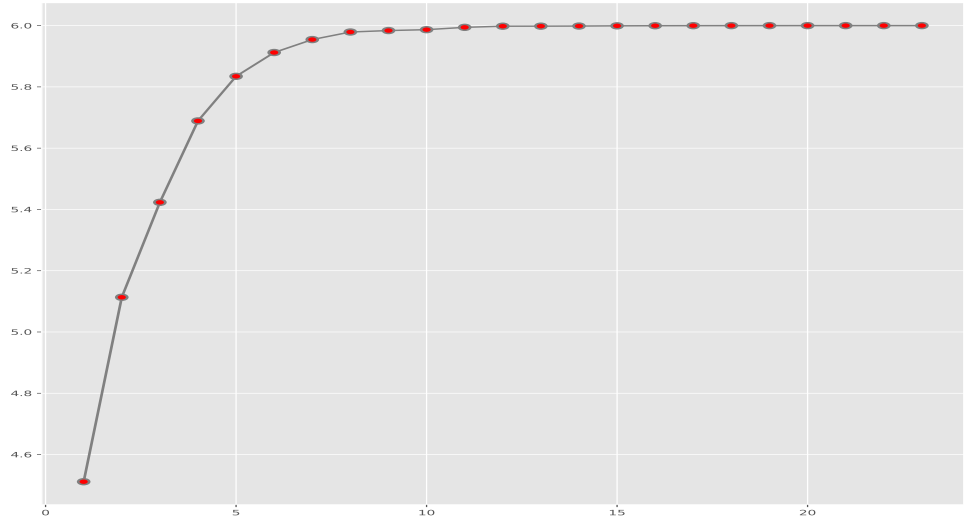

Figure 8.3 Convergence of Symmetric Power Method

Code of Power Method

```
def matrixMul(A, B):
    n = len(A)
    m = len(B[0])
    L = len(B)
    C = [[0] * m for i in range(n)]
    for i in range(0, n):
        for j in range(0, m):
            for k in range(0, L):
                C[i][j] += A[i][k] * B[k][j]
    return C


```

```
13 def norm(A):
14     tmp = 0
15     for i in range(0, len(A)):
16         tmp = max(tmp, A[i][0])
17     return tmp
18 
19 
20 def power(A, X, n, TOL):
21     base, value = [], []
22     for k in range(0, n):
23         y = matrixMul(A, X)
24         u = norm(y)
25         if u == 0:
26             print("Eigenvector", end=" ")
27             for i in range(0, len(X)):
28                 print(X[i][0], end=" \n"[0 == len(X) - 1])
29             print("A has the eigenvalue 0, select a new vector x and restart")
30         err = 0
31         for i in range(0, len(X)):
32             err = max(err, X[i][0] - (y[i][0] / u))
33         base.append(k)
34         value.append(u)
35         if err < TOL:
36             print("Eigenvalue", u)
37             print("Eigenvector", end=" ")
38             for i in range(0, len(X)):
39                 print(X[i][0], end=" \n"[0 == len(X) - 1])
40             break
41         for i in range(0, len(X)):
42             X[i][0] = y[i][0] / u
```

Code of Symmetric Power Method

```
1  import numpy as np
2  
3  def matrixT(A):
4      n = len(A)
5      m = len(A[0])
6      B = [[0] * n for i in range(m)]
7      for i in range(0, m):
8          for j in range(0, n):
9              B[i][j] = A[j][i]
10     return B
11 
12 
13 def matrixMul(A, B):
14     n = len(A)
```

```
    m = len(B[0])
    L = len(B)
    C = [[0] * m for i in range(n)]
    for i in range(0, n):
        for j in range(0, m):
            for k in range(0, L):
                C[i][j] += A[i][k] * B[k][j]
    return C


def norm2(A):
    tmp = 0
    for i in range(0, len(A)):
        tmp += A[i][0] * A[i][0]
    return np.sqrt(tmp)


def sympower(A, X, n, TOL):
    base, value = [], []
    for k in range(0, n):
        y = matrixMul(A, X)
        T1 = matrixMul(matrixT(X), y)
        u = T1[0][0]
        yp = norm2(y)
        if yp == 0:
            print("Eigenvector", end=" ")
            for i in range(0, len(X)):
                print(X[i][0], end=" \n"[0 == len(X) - 1])
            print("A has the eigenvalue 0, select a new vector x and restart")
        err = 0
        for i in range(0, len(A)):
            err += (X[i][0] - y[i][0] / yp) * (X[i][0] - y[i][0] / yp)
        err = np.sqrt(err)
        base.append(k + 1)
        value.append(u)
        for i in range(0, len(A)):
            X[i][0] = y[i][0] / yp
        if err < TOL:
            print("Eigenvalue", u, "times", k)
            print("Eigenvector", end=" ")
            for i in range(0, len(X)):
                print(X[i][0], end=" \n"[0 == len(X) - 1])
            break
```

Code of Inverse Power Method

```
def matrixT(A):
    n = len(A)
```

```
    m = len(A[0])
    B = [[0] * n for i in range(m)]
    for i in range(0, m):
        for j in range(0, n):
            B[i][j] = A[j][i]
    return B


def matrixMul(A, B):
    n = len(A)
    m = len(B[0])
    L = len(B)
    C = [[0] * m for i in range(n)]
    for i in range(0, n):
        for j in range(0, m):
            for k in range(0, L):
                C[i][j] += A[i][k] * B[k][j]
    return C


def norm(A):
    tmp = 0
    for i in range(0, len(A)):
        tmp = max(tmp, A[i][0])
    return tmp


def Gaussian(A):
    n = len(A)

    for i in range(0, n):
        # Search for maximum in this column
        maxEl = abs(A[i][i])
        maxRow = i
        for k in range(i + 1, n):
            if abs(A[k][i]) > maxEl:
                maxEl = abs(A[k][i])
                maxRow = k

        # Swap maximum row with current row
        for k in range(i, n + 1):
            A[maxRow][k], A[i][k] = A[i][k], A[maxRow][k]

        # Make all rows below this one 0 in current column
        for k in range(i + 1, n):
            c = -A[k][i] / A[i][i]
            for j in range(i, n + 1):
                if i == j:
```

```
                    A[k][j] = 0
                else:
                    A[k][j] += c * A[i][j]

    # Solve equation Ax=b for an upper triangular matrix A
    x = [0 for i in range(n)]
    for i in range(n - 1, -1, -1):
        x[i] = A[i][n] / A[i][i]
        for k in range(i - 1, -1, -1):
            A[k][n] -= A[k][i] * x[i]
    return x


def inversePower(A, X, n, TOL):
    T1 = matrixMul(matrixMul(matrixT(X), A), X)
    T2 = matrixMul(matrixT(X), X)
    q = T1[0][0] / T2[0][0]
    for k in range(0, n):
        B = [[0] * (len(A[0]) + 1) for i in range(len(A))]
        for i in range(0, len(A)):
            for j in range(0, len(A[0])):
                B[i][j] = A[i][j]
            B[i][i] -= q
            B[i][len(A[0])] = X[i][0]
        z = Gaussian(B)
        y = [[0] * 1 for i in range(len(z))]
        for i in range(0, len(z)):
            y[i][0] = z[i]
        u = norm(y)
        err = 0
        for i in range(0, len(X)):
            err = max(err, X[i][0] - (y[i][0] / u))
        for i in range(0, len(X)):
            X[i][0] = y[i][0] / u
        if err < TOL:
            u = 1 / u + q
            print("Eigenvalue", u, "times", k + 1)
            print("Eigenvector", end=" ")
            for i in range(0, len(X)):
                print(X[i][0], end=" \n"[0 == len(X) - 1])
            break
```

8.2.2 Task 2

Problem Description

Write the H_1, H_2, H_3 in the Household Transformation example in the PPT,

and verify the example results.

Household transformation

Householder transformation has form

$$H = I - 2 * \frac{vv^T}{v^T v}$$

Household QR Factorization

To compute QR factorization of A, use Householder transformation to annihilate subdiagonal entries of each successive column. Each Householder transformation is applied to entire matrix, but does not affect prior columns, so zeros are preserved.

In applying Householder transformation H to arbitrary vector u,

$$Hu = (I - 2\frac{vv^T}{v^T v})\mu = \mu - 2\frac{v^T \mu}{v^T v}v$$

which is much cheaper than general matrix-vector multiplication and requires only vector v, not full matrix H.

Process just described produces factorization

$$H_n ... H_1 A = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

where R is $n * n$ and upper triangular. If $Q = H_1 ... H_n$, then $A = Q \begin{bmatrix} R \\ 0 \end{bmatrix}$. To preserve solution of linear least squares problem, right-hand side b is transformed by same sequence of Householder transformations. Then solve triangular least squares problem

$$\begin{bmatrix} R \\ 0 \end{bmatrix} x \approx Q^T b$$

For solving linear least squares problem, product Q of Householder transformations need not be formed explicitly. R can be stored in upper triangle of array initially containing A. Householder vectors v can be stored in lower triangular portion of A.

Solution

In the context of example of Householder QR Factorization in PPT, we could

acquire the information below.

For polynomial data-fitting example given previously, with

$$A = \begin{bmatrix} 1 & -1 & 1.0 \\ 1 & -0.5 & 0.25 \\ 1 & 0.0 & 0.0 \\ 1 & 0.5 & 0.25 \\ 1 & 1.0 & 1.0 \end{bmatrix}, b = \begin{bmatrix} 1.0 \\ 0.5 \\ 0.0 \\ 0.5 \\ 2.0 \end{bmatrix}$$

Householder vector v_i for annihilating subdiagonal entries of first column of A is

$$v_1 = \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{bmatrix} - \begin{bmatrix} -2.236 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 3.236 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{bmatrix}$$

According to the equation of householder transformation, we could acquire the matrix H_1 as follows.

$$H = I - 2 * \frac{vv^T}{v^T v}$$

$$H_1 = \begin{bmatrix} -0.44720 & -0.44722 & -0.44722 & -0.44722 & -0.44722 \\ -0.44722 & 0.86180 & -0.13820 & -0.13820 & -0.13820 \\ -0.44722 & -0.13820 & 0.86180 & -0.13820 & -0.13820 \\ -0.44722 & -0.13820 & -0.13820 & 0.86180 & -0.13820 \\ -0.44722 & -0.13820 & -0.13820 & -0.13820 & 0.86180 \end{bmatrix},$$

Thus, we could obtain the answer in PPT as follows.

$$H_1 A = \begin{bmatrix} -2.23607 & -0.00002 & -1.11802 \\ -0.00002 & -0.19098 & -0.40452 \\ -0.00002 & 0.30902 & -0.65452 \\ -0.00002 & 0.80902 & -0.40452 \\ -0.00002 & 1.30902 & 0.34548 \end{bmatrix}, H_1 b = \begin{bmatrix} -1.78885 \\ -0.36182 \\ -0.86182 \\ -0.36182 \\ 1.13818 \end{bmatrix}.$$

Moreover, we continually verif the results in PPT.

$$v_2 = \begin{bmatrix} 0 \\ -1.772 \\ 0.309 \\ 0.809 \\ 1.309 \end{bmatrix}, H_2 = \begin{bmatrix} 1.00000 & 0.00000 & 0.00000 & 0.00000 & 0.00000 \\ 0.00000 & -0.12074 & 0.19543 & 0.51167 & 0.82790 \\ 0.00000 & 0.19543 & 0.96592 & -0.08922 & -0.14437 \\ 0.00000 & 0.51167 & -0.08922 & 0.76640 & -0.37798 \\ 0.00000 & 0.82790 & -0.14437 & -0.37798 & 0.38842 \end{bmatrix}$$

In addition, we could acquire the matrix H_2H_1A and H_2H_1b as follows.

$$H_2H_1A = \begin{bmatrix} -2.23607 & -0.00002 & -1.11802 \\ -0.00003 & 1.58114 & -0.00003 \\ -0.00002 & -0.00000 & -0.72505 \\ -0.00002 & -0.00004 & -0.58919 \\ -0.00001 & -0.00007 & 0.04668 \end{bmatrix}, H_2H_1b = \begin{bmatrix} -1.78885 \\ 0.63243 \\ -1.03520 \\ -0.81574 \\ 0.40372 \end{bmatrix}.$$

Furthermore, we are likely to calculate the answer of H_3 and verify the result in PPT.

$$v_3 = \begin{bmatrix} 0 \\ 0 \\ -1.660 \\ -0.589 \\ 0.047 \end{bmatrix}, H_3 = \begin{bmatrix} 1.00000 & 0.00000 & 0.00000 & 0.00000 & 0.00000 \\ 0.00000 & 1.00000 & 0.00000 & 0.00000 & 0.00000 \\ 0.00000 & 0.00000 & -0.77510 & -0.62984 & 0.05026 \\ 0.00000 & 0.00000 & -0.62984 & 0.77652 & 0.01783 \\ 0.00000 & 0.00000 & 0.05026 & 0.01783 & 0.99858 \end{bmatrix}$$

In addition, we could acquire the matrix H_2H_1A and H_2H_1b as follows.

$$H_3H_2H_1A = \begin{bmatrix} -2.23607 & -0.00002 & -1.11802 \\ -0.00003 & 1.58114 & -0.00003 \\ 0.00003 & 0.00002 & 0.93543 \\ -0.00000 & -0.00003 & -0.00002 \\ -0.00002 & -0.00007 & -0.00033 \end{bmatrix}, H_3H_2H_1b = \begin{bmatrix} -1.78885 \\ 0.63243 \\ 1.33645 \\ 0.02577 \\ 0.33657 \end{bmatrix}.$$

So far, we have obtained H_1, H_2, H_3 and verify the results in PPT which is correct.

Code of Householder Transformation

```
1 def matrixMul(A, B):
```

```
    n = len(A)
    m = len(B[0])
    L = len(B)
    C = [[0] * m for i in range(n)]
    for i in range(0, n):
        for j in range(0, m):
            for k in range(0, L):
                C[i][j] += A[i][k] * B[k][j]
    return C

def Householder(v):
    p1 = matrixMul(v, matrixT(v))
    p2 = matrixMul(matrixT(v), v)
    H = [[0] * 5 for i in range(5)]
    for i in range(0, 5):
        for j in range(0, 5):
            H[i][j] = I[i][j] - 2 * p1[i][j] / p2[0][0]
    return H
```

References

(1) Richard L. Burden, J. D. F., *Numerical Analysis, ninth edition*, 9th ed.; Brooks/Cole: America, 2011.

(2) D.Fink., J. H. M. K., *LaTeX: Numerical Methods Using MATLAB Fourth Edition*, 2nd ed.; Publishing House of Electronics Industry: China, 2002.