计算物理第四次作业

白博臣 2022141220036

Problem 1：

代码：

```
from sympy import symbols, Eq, solve
import numpy as np

# 定义符号变量
x = symbols('x')
# 给定 b 的值
b = eval(input('请给定 b 的值：'))
# 定义方程
equation = Eq(x ** 2 - b * x + 1, 0)

# 求解方程得到符号解
symbolic_solution = solve(equation, x)[0]

# 使用公式法求解得到数值解
r = np.sqrt(b ** 2 - 4)
x_1 = (b + r) / 2
x_2 = (b - r) / 2
numerical_solution = x_2

# 计算误差
absolute_error = abs(symbolic_solution.evalf() - numerical_solution)
percent_error = abs(absolute_error / symbolic_solution.evalf() * 100)

print("符号解：", symbolic_solution.evalf())
print("数值解：", numerical_solution)
print("绝对误差：", absolute_error)
print("误差百分比：", percent_error, "%")
print('-------------------------------分割线----------------------------------')

# 改进方法后
x_2_new = 2 / (b + r)
numerical_solution = x_2_new
absolute_error = abs(symbolic_solution.evalf() - numerical_solution)
percent_error = abs(absolute_error / symbolic_solution.evalf() * 100)

```

```
print("符号解: ", symbolic_solution.evalf())
print("数值解: ", numerical_solution)
print("绝对误差: ", absolute_error)
print("误差百分比: ", percent_error, "%")
```

计算值：

```
请给定b的值：100
符号解：  0.0100010002000500
数值解：  0.010001000200048793
绝对误差：  1.22124532708767e-15
误差百分比：  1.22112319034007e-11 %
------------------------------分割线--------------------------------
符号解：  0.0100010002000500
数值解：  0.010001000200050014
绝对误差：  0
误差百分比：  0 %
```

增大 b 的值：

```
请给定b的值：1000
符号解：  0.00100000100000200
数值解：  0.0010000010000226212
绝对误差：  2.06210916398053e-14
误差百分比：  2.06210710186930e-9 %
------------------------------分割线--------------------------------
符号解：  0.00100000100000200
数值解：  0.001000001000002
绝对误差：  0
误差百分比：  0 %
```

增大 b 的值：

```
请给定b的值：10000
符号解：  0.000100000001000000
数值解：  0.00010000000111176632
绝对误差：  1.11766309098330e-13
误差百分比：  1.11766307980666e-7 %
------------------------------分割线--------------------------------
符号解：  0.000100000001000000
数值解：  0.00010000000100000001
绝对误差：  0
误差百分比：  0 %
```

产生误差的原因分析：

对于方程$x^2 - b + 1 = 0$，使用求根公式：$x = b \pm \sqrt{b^2 - 4}$后计算得到解，其中在$b \pm \Delta$计算时会产生 unit-roundoff 误差。一种有效的改进措施便是改求根公式为$x_2 = \frac{b-r}{2}\frac{b+r}{b+r} = \frac{2}{b+r}$，如此则能避免$b - r$的计算造成较大的舍入误差。

Problem 2

由于[0.7,1.3]的范围有些大，对现象的观察效果不明显，故缩小了一部分范围改为[0.998,1.002]。

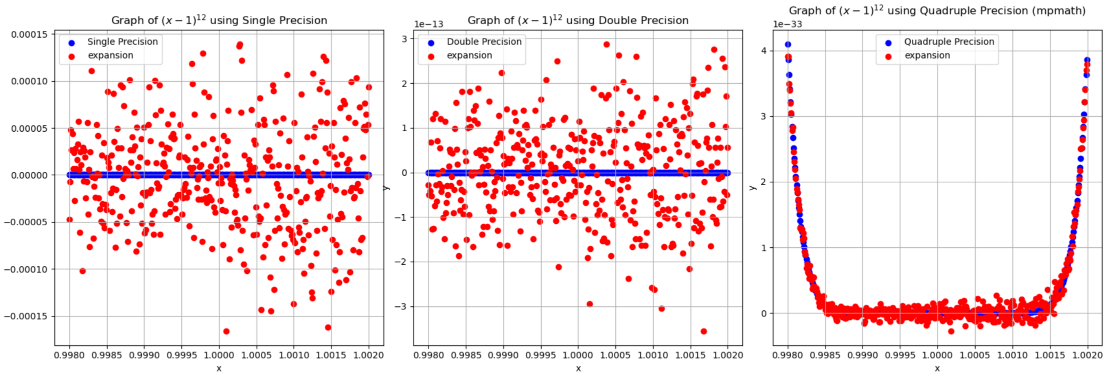
代码：

```
import matplotlib.pyplot as plt
import numpy as np
from mpmath import mp

# 设置mpmath库的精度为四倍精度
mp.dps = 36


# 定义函数
def func(x):
    return (x - 1) ** 12


def func2(x):
    return x ** 12 - 12 * x ** 11 + 66 * x ** 10 - 220 * x ** 9 + 495 * x ** 8 - 792 * x ** 7 + 924 * x ** 6 - 792 * x ** 5 + 495 * x ** 4 - 220 * x ** 3 + 66 * x ** 2 - 12 * x + 1


def func3(x):
    return (((((((((((
                             x - 12) * x + 66) * x - 220) * x + 495) * x - 792) * x + 924) * x - 792) * x + 495) * x - 220) * x + 66) * x - 12) * x + 1


# 定义 x 范围
x_single = np.linspace(0.998, 1.002, 400, dtype=np.float32)
x_double = np.linspace(0.998, 1.002, 400, dtype=np.float64)
step = mp.mpf('0.00001')
x_quad = [mp.mpf(val) for val in mp.arange('0.998', '1.002', step)]

# 计算函数值
y_single = func(x_single)
y_double = func(x_double)
y_quad = [func(x) for x in x_quad]

# 计算展开式函数图
```

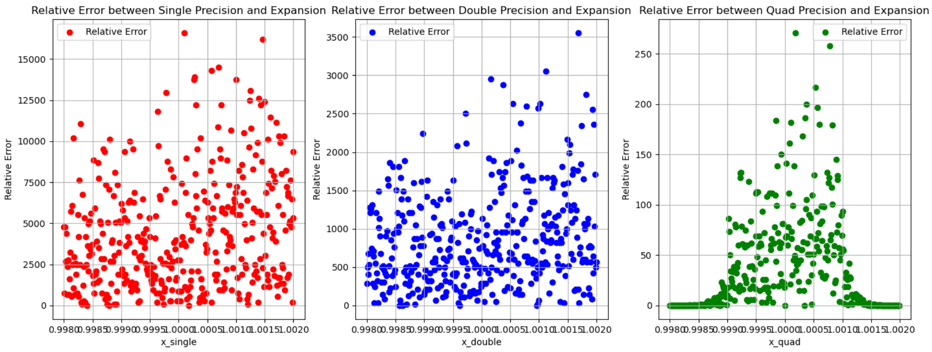
```python
y_single2 = func2(x_single)
y_double2 = func2(x_double)
y_quad2 = [func2(x) for x in x_quad]
y_single3 = func3(x_single)
y_double3 = func3(x_double)
y_quad3 = [func3(x) for x in x_quad]

# 绘制图像
plt.figure(figsize=(24, 6))

plt.subplot(1, 3, 1)
plt.scatter(x_single, y_single, label='Single Precision', color='blue')
plt.scatter(x_single, y_single2, label='expansion', color='red')
plt.scatter(x_single, y_single3, label='Horner', color='green')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.title('Graph of $(x-1)^{12}$ using Single Precision')

plt.subplot(1, 3, 2)
plt.scatter(x_double, y_double, label='Double Precision', color='blue')
plt.scatter(x_double, y_double2, label='expansion', color='red')
plt.scatter(x_single, y_double3, label='Horner', color='green')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.title('Graph of $(x-1)^{12}$ using Double Precision')

plt.subplot(1, 3, 3)
plt.scatter(x_quad, y_quad, label='Quadruple Precision', color='blue')
plt.scatter(x_quad, y_quad2, label='expansion', color='red')
plt.scatter(x_single, y_quad3, label='Horner', color='green')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.title('Graph of $(x-1)^{12}$ using Quadruple Precision (mpmath)')

plt.tight_layout()
plt.show()

# 设置一个很小的数，用于替代零
```

```python
epsilon_single = 1e-8
epsilon_double = 1e-16
epsilon_quad = 1 * 10 ** (-36)

# 计算相对误差，避免除以零
relative_error_single = np.abs(y_single - y_single2) / (np.abs(y_single) + epsilon_single)
relative_error_double = np.abs(y_double - y_double2) / (np.abs(y_double) + epsilon_double)
y_quad = np.array(y_quad)
y_quad2 = np.array(y_quad2)
relative_error_quad = np.abs(y_quad - y_quad2) / (np.abs(y_quad) + epsilon_quad)

# 绘制相对误差图表
plt.figure(figsize=(24, 6))
plt.subplot(1, 3, 1)
plt.scatter(x_single, relative_error_single, label='Relative Error', color='red')
plt.xlabel('x_single')
plt.ylabel('Relative Error')
plt.legend()
plt.grid(True)
plt.title('Relative Error between Single Precision and Expansion')

plt.subplot(1, 3, 2)
plt.scatter(x_double, relative_error_double, label='Relative Error', color='blue')
plt.xlabel('x_double')
plt.ylabel('Relative Error')
plt.legend()
plt.grid(True)
plt.title('Relative Error between Double Precision and Expansion')

plt.subplot(1, 3, 3)
plt.scatter(x_quad, relative_error_quad, label='Relative Error', color='green')
plt.xlabel('x_quad')
plt.ylabel('Relative Error')
plt.legend()
plt.grid(True)
plt.title('Relative Error between Quad Precision and Expansion')

plt.show()
```

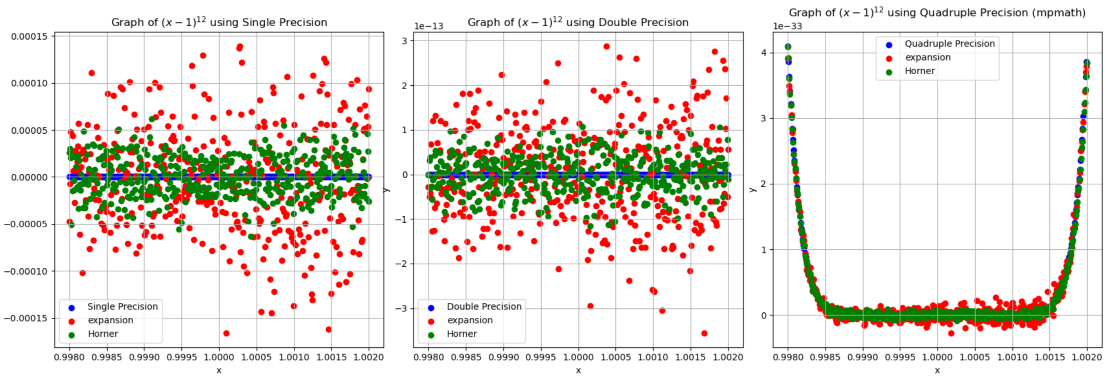
输出后的结果：



以及相对误差：



引入 Horner's Method 后：



引入 Horner's Method 后，避免了多重的乘方运算，一定程度上缩小了误差，同时我们发现随着精度的增加，其精确性也会显著地增加。

Problem 3：

```
import numpy as np

unit_roundoff_half = np.finfo(np.float16).eps
unit_roundoff_single = np.finfo(np.float32).eps
unit_roundoff_double = np.finfo(np.float64).eps
```

```python

half_min = np.finfo(np.float16).tiny

print("半精度数据类型的最小精度:", half_min)

print("Half Precision Unit Roundoff:", unit_roundoff_half)
print("Single Precision Unit Roundoff:", unit_roundoff_single)
print("Double Precision Unit Roundoff:", unit_roundoff_double)
```

输出结果：

```
半精度数据类型的最小精度: 6.104e-05
Half Precision Unit Roundoff: 0.000977
Single Precision Unit Roundoff: 1.1920929e-07
Double Precision Unit Roundoff: 2.220446049250313e-16
```

Problem 4：

通过循环迭代算法，将a_n的各项值写出，代码如下：

```python
import numpy as np


def cal(n):
    a_0 = 2
    for i in range(3, n + 1):
        a_n = 2 ** (i - 1 - 1 / 2) * np.sqrt(1 - np.sqrt(1 - 4 ** (1 - (i - 1)) * a_0 ** 2))
        a_0 = a_n
    return a_0


for i in range(2, 31):
    print('n=', i, cal(i))
```

输出结果如下：

```
n= 2 2
n= 3 2.8284271247461903
n= 4 3.0614674589207187
n= 5 3.121445152258053
n= 6 3.136548490545941
n= 7 3.140331156954739
n= 8 3.141277250932757
n= 9 3.1415138011441455
n= 10 3.141572940367827
n= 11 3.141587725279961
n= 12 3.141591421504635
n= 13 3.141592345611077
n= 14 3.141592576545005
n= 15 3.1415926334632487
n= 16 3.1415926548075896
n= 17 3.1415926832667105
n= 18 3.1415927591577
n= 19 3.1415929109396727
n= 20 3.141594125195191
n= 21 3.1415965537048196
n= 22 3.1415965537048196
n= 23 3.1416742650217575
n= 24 3.141829681889201
n= 25 3.142451272494134
n= 26 3.142451272494134
n= 27 3.1622776601683795
n= 28 3.1622776601683795
n= 29 3.4641016151377544
n= 30 4.000000000000001
n= 31 8.000000000000002
n= 32 16.000000000000004
n= 33 32.00000000000001
n= 34 64.00000000000001
n= 35 128.00000000000003
```

发现在第 30 项时发生改变，我们观察给出的迭代关系：

$$Z_{n+1} = 2^{n-\frac{1}{2}}\sqrt{1-\sqrt{1-4^{1-n}z_n^2}},$$

可以观察到达到一定次数的乘方和开方之后会有较大的误差,再进行外层的运算，最后导致只有后面的 2 的乘方（对于图中对应的就是$n \geq 30$时），如果想要提高精确度的话可以采用 decimal 库提高精度。