计算物理学第一次作业报告

陈跃元

物理学院 1600011328

1 双精度浮点数和其 IEEE754 编码的互相转化

1.1 问题描述

实现任意十进制数于其规范化的二进制双精度浮点数编码间的相互转换. 具体而言, 输入一个双精度浮点数, 需要给出它按照 IEEE754 标准的编码; 输入一个 64 位, 由'0'和'1'组成的字符串, 将它按照 IEEE754 标准转化为双精度浮点数.

1.2 算法

1.2.1 双精度浮点数转编码

有两种可行的算法:

1. 按定义转换
   - 确定符号位 (若 $x == 0$ 直接返回全零),exponent 初始值设为 1023.
   - (确定 exponent) 判断 $x \geqslant 1$?
     - Yes
       不断重复 $x/=2, \quad \text{exponent}++$, 直到 $x<2$.
     - No
       不断重复 $x*=2, \quad \text{exponent}--$, 直到 $x>=1$.
   - $x--$
   - (小数的二进制转换) 从 $i=12$ 开始, 不断重复 $x*=2, \quad s[i]=x\%1, \quad i++$, 直至将 s 的小数部分 52 位全部填满.
   - 返回 s.

   这一算法比起将 x 分解为整数部分和小数部分分别转换的效率要更高 (同为 $O(\lg n)$ 级别, 但常数比分开转换要小), 且在大数情形下, 不会出现分解出的整数部分过大造成的 overflow.

2. 利用强制指针类型转换
   这一算法某种程度上是一种投机取巧的办法. 在 `C++`中, `double` 类型和 `long long` 类型都是由 8 个字节来存储的, 且 `double` 类型在内存中的存储方式就是按照 IEEE 754 标准执行的, 故可以通过将指向 x 的 `double` 类指针强制转化为 `long long` 类指针, 随后将这一 `long long` 按照整数的二进制转换来得到编码.

1.2.2 编码转双精度浮点数

这时的算法相当于将上面两个算法倒过来.

1. 按定义转换

    - (确定小数) 将 s 的后 52 位按照二进制小数的规则转化为 x.
    - $x++$
    - (确定 exponent) 将 exponent 部分计算出来. 判断:
        - exponent $== 0$
          返回 $x = 0$.
        - exponent $>= 1023$
          不断重复 $x* = 2, \quad \text{exponent} --$, 直到 exponent $= 1023$.
        - $0 <$ exponent < 1023
          不断重复 $x/ = 2, \quad \text{exponent} ++$, 直到 exponent $= 1023$.
    - 若符号位为 1, $x* = -1$.
    - 返回 x.

2. 利用强制指针类型转换
   将 s 按照整数的二进制转化规则转化为 `long long` 类型的 x, 并将指向 x 的 `long long` 指针强制类型转换为 `double` 指针, 返回指针所指的双精度浮点数.

源代码见附录A.1.1, 或 `src/Problem_1/main.cpp`.

1.3 结果

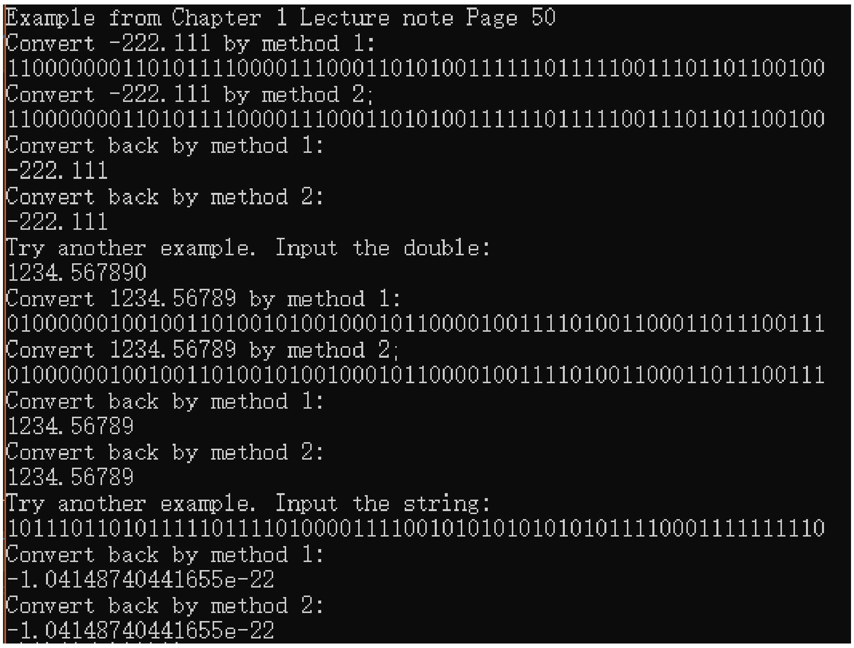

图 1: 第一题的结果

运行如图1所示. 两种方法得到的结果相同.

2 带权的高斯积分

2.1 问题描述

构造下列三个高斯积分的节点和权重.

$$\int_0^1 \sqrt{x} f(x) dx \approx A_0 f\left(x_0\right)+A_1 f\left(x_1\right) \tag{1}$$

$$\int_0^1 \sqrt{x} f(x) dx \approx A_0 f\left(x_0\right)+A_1 f\left(x_1\right)+A_2 f\left(x_2\right) \tag{2}$$

$$\int_{-1}^1\left(1+x^2\right) f(x) dx \approx A_0 f\left(x_0\right)+A_1 f\left(x_1\right) \tag{3}$$

2.2 推导和结果

对 (1), 对于节点 x_i, 应满足对所有 $0 \leqslant k \leqslant n$,

$$\int_0^1 \sqrt{x}\left[\prod_{i=0}^n\left(x-x_i\right)\right] P_k(x) dx=0 \tag{4}$$

即

$$\left\{\begin{array}{l}\frac{1}{7}-\frac{x_0}{5}-\frac{x_1}{5}+\frac{x_0 x_1}{3}=0 \\ \frac{1}{9}-\frac{x_0}{7}-\frac{x_1}{7}+\frac{x_0 x_1}{5}=0\end{array}\right. \tag{5}$$

解得 $x_0=\frac{1}{63}(35-2 \sqrt{70}) \approx 0.289949, x_1=\frac{1}{63}(35+2 \sqrt{70}) \approx 0.821162$. 带入原式, 为使所有阶数 $\leqslant 1$ 的多项式积分为准确值, 应有

$$\left\{\begin{aligned} A_0+A_1 & =\frac{2}{3} \\ A_0 x_0+A_1 x_1 & =\frac{2}{5}\end{aligned}\right. \tag{6}$$

解得 $A_0 \approx 0.277556, A_1 \approx 0.389111$.

对 (2), 同样有 (4) 的要求, 即

$$\left\{\begin{aligned} -\frac{2}{3} a b c+\frac{2}{5}(a b+b c+a c)-\frac{2}{7}(a+b+c)+\frac{2}{9} & =0 \\ -\frac{2}{5} a b c+\frac{2}{7}(a b+b c+a c)-\frac{2}{9}(a+b+c)+\frac{2}{11} & =0 \\ -\frac{2}{21} a b c+\frac{2}{15}(a b+b c+c a)-\frac{10}{77}(a+b+c)+\frac{14}{117} & =0\end{aligned}\right. \tag{7}$$

解得 $x_0=\frac{1}{63}(35-2 \sqrt{70}) \approx 0.289949, x_1=\frac{1}{63}(35+2 \sqrt{70}) \approx 0.821162$. 带入原式, 为使所有阶数 $\leqslant 1$ 的多项式积分为准确值, 应有

$$\left\{\begin{aligned} A_0+A_1 & =\frac{2}{3} \\ A_0 x_0+A_1 x_1 & =\frac{2}{5}\end{aligned}\right. \tag{8}$$

解得 $A_0 \approx 0.277556, A_1 \approx 0.389111$.

3 衰变

3.1 问题描述

两类核 A 和 B 发生衰变, 初始时刻 $N_A = N_B = 1$, 满足

$$\frac{dN_A}{dt} = -\frac{N_A}{\tau_A} \tag{9}$$

$$\frac{dN_B}{dt} = \frac{N_A}{\tau_A} - \frac{N_B}{\tau_B} \tag{10}$$

编写程序求解耦合微分方程组, 并和解析解比较.

3.2 算法

解析解:

$$N_A = \exp(-\frac{t}{\tau_A}), \quad N_B = \frac{\tau_A \exp(-\frac{t}{\tau_B}) + \tau_B(\exp(-\frac{t}{\tau_A}) - 2\exp(-\frac{t}{\tau_B}))}{\tau_A - \tau_B} \tag{11}$$

笔者考虑了下面四种不同的算法 (伪代码中将 N_A 和 N_B 分别记作 `a`, `b`):

1. Naive iteration

```
a[n] = a[n-1] - delta * a[n-1] / tau_a
b[n] = b[n-1] + delta * (a[n-1] / tau_a - b[n-1] / tau_b)
```

   其中 `n` 是迭代步数, `delta` 是时间的步长, `tau_a` 和 `tau_b` 的意义不言自明, 下同.

2. Naive iteration, 但对 N_B 迭代时, N_A 取前后两次的 N_A 的平均值.

```
a[n] = a[n-1] - delta * a[n-1] / tau_a
b[n] = b[n-1] + delta * ((a[n] + a[n-1]) / (2 * tau_a) - b[n-1] / tau_b)
```

3. 对 N_A 用 RK-4, 对 N_B 用 Naive iteration, 但 N_A 的值取两次迭代的平均值.

```
k[0] = - delta * a[n-1] / tau_a
k[1] = - delta * (a[n-1] + k[0] / 2) / tau_a
k[2] = - delta * (a[n-1] + k[1] / 2) / tau_a
k[3] = - delta * (a[n-1] + k[2]) / tau_a
a[n] = a[n-1] + (k[0] + 2 * k[1] + 2 * k[2] + k[3]) / 6
b[n] = b[n-1] + delta * ((a[n-1] + a[n]) / (2 * tau_a) - b[n-1] / tau_b)
```

4. 对 N_A 用 RK-4, 然后再对 N_B 用 RK-4.

```
k[0] = -delta * a[n-1] / tau_a
k[1] = -delta * (a[n-1] + k[0] / 2) / tau_a
k[2] = -delta * (a[n-1] + k[1] / 2) / tau_a
k[3] = -delta * (a[n-1] + k[2]) / tau_a
a[n] = a[n-1] + (k[0] + 2 * k[1] + 2 * k[2] + k[3]) / 6

k[0] = delta * (a[n-1] / tau_a - b[n-1] / tau_b);
k[1] = delta * ((a[n-1] + a[n]) / (2 * tau_a) - (b[n-1] + k[0] / 2) / tau_b)
// Explicit time dependent in N_A, linear intrapolate
k[2] = delta * ((a[n-1] + a[n]) / (2 * tau_a) - (b[n-1] + k[1] / 2) / tau_b)
// Explicit time dependent in N_A, linear intrapolate
k[3] = delta * (a[n] / tau_a - (b[n-1] + k[2]) / tau_b)
b[n] = b[n-1] + (k[0] + 2 * k[1] + 2 * k[2] + k[3]) / 6
```

需要注意一点, 在得到 `a[n]`以后, 代入计算 N_B 时, RK-4 要求 `k[1]` 和 `k[2]` 中取 `t[n] + delta / 2` 时的 N_A, 但这一时刻并不在迭代的时刻中, 考虑到 `delta`较小, 期间 N_A 的变化可以近似看做线性的, 故通过线性插值得到 `t[n] + delta / 2` 时的 N_A.

源代码见附录A.1.2, 或 `src/Problem_3/main.cpp`.

3.3 结果

3.3.1 几种方法的比较

笔者首先比较了不同算法在 $\tau_A = 1\,\mathrm{s}, \tau_B = 10\,\mathrm{s}, \Delta t = 0.2\,\mathrm{s}$ 的情况下的误差, 以找到其中误差最小的方法. 图2、3、4分别给出了各个方法的计算结果、相对误差和绝对误差. 表1中列出了各种方法的最大相对误差. 很显然, 第四种算法的误差相比于剩下三种方法的误差要小得多, 故之后的分析都基于第四种方法.

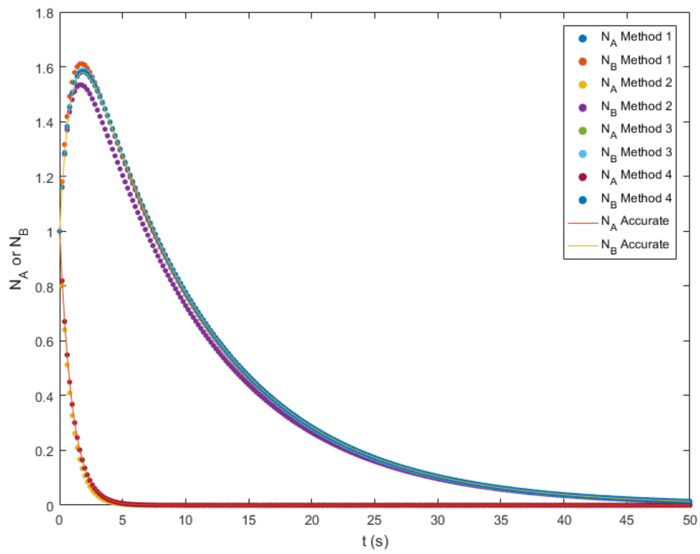

图 2: 不同方法在 $\tau_A = 1\,\mathrm{s}, \tau_B = 10\,\mathrm{s}, \Delta t = 0.2\,\mathrm{s}$ 的情况下的结果

表 1: 不同方法在 $\tau_A = 1\,\mathrm{s}, \tau_B = 10\,\mathrm{s}, \Delta t = 0.2\,\mathrm{s}$ 的情况下的最大相对误差 (绝对值)

	Method 1	Method 2	Method 3	Method 4
N_A	99.69%	99.69%	0.10%	0.10%
N_B	4.95%	9.94%	4.18%	0.22%

3.4 不同的 τ_B 下的行为

如图5所示, N_A 的行为与 τ_A 和 τ_B 的大小关系无关, 而当 $\tau_B > \tau_A$ 时,N_B 短期行为将会出现增加, $\tau_B \leqslant \tau_A$ 时,N_B 将一直减小.N_B 的长期行为成指数衰减.

3.5 步长对误差的影响

表2给出了 $t = 50\,\mathrm{s}$ 内不同步长 Δt 下的最大相对误差 (绝对值). 由于算法本身的误差已经很小, 步长变化造成的影响很小, 但依然可以看出, 步长越小, 误差越小.

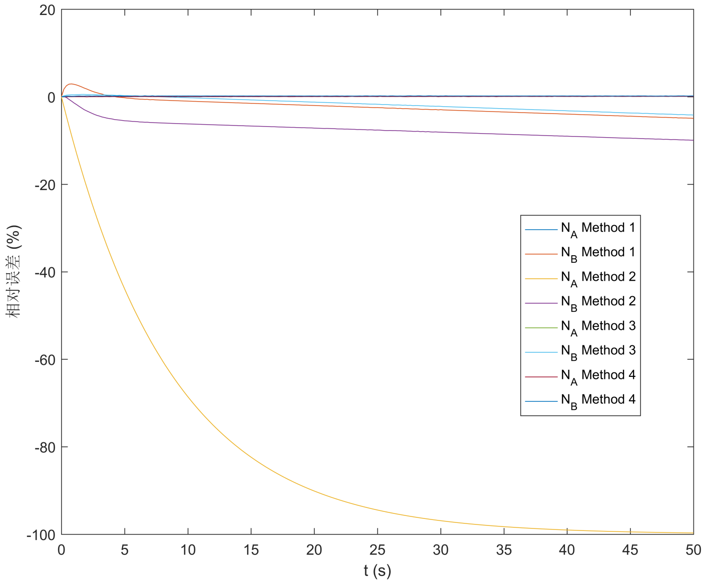

图 3: 不同方法在 $\tau_A = 1\,\mathrm{s}, \tau_B = 10\,\mathrm{s}, \Delta t = 0.2\,\mathrm{s}$ 的情况下的相对误差

表 2: $t = 50\,\mathrm{s}$ 内不同步长 Δt 下的最大相对误差 (绝对值)

Δt	$0.2\,\mathrm{s}$	$0.1\,\mathrm{s}$	$0.05\,\mathrm{s}$
N_A	0.103%	0.048%	0.048%
N_B	0.219%	0.089%	0.057%

4 双精度浮点数和其 IEEE754 编码的互相转化

4.1 问题描述

4.2 算法

4.3 结果

4.4 源代码

5 双精度浮点数和其 IEEE754 编码的互相转化

5.1 问题描述

5.2 算法

5.3 结果

5.4 源代码

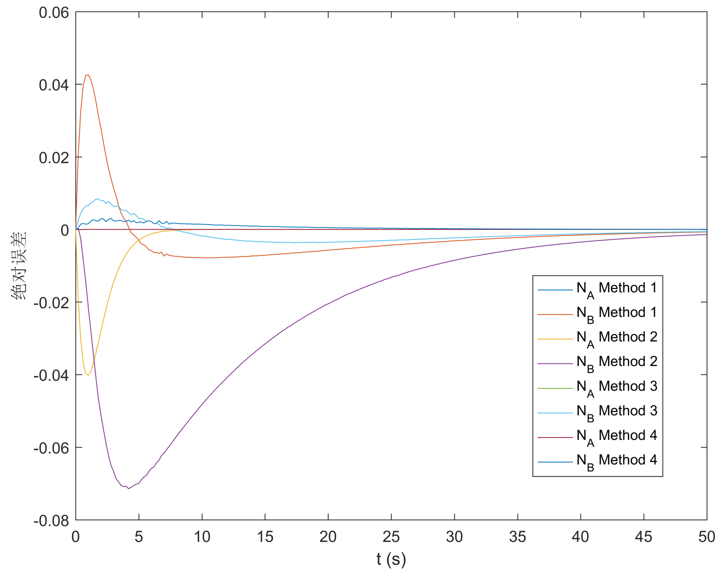

图 4: 不同方法在 $\tau_A = 1\,\mathrm{s}, \tau_B = 10\,\mathrm{s}, \Delta t = 0.2\,\mathrm{s}$ 的情况下的绝对误差

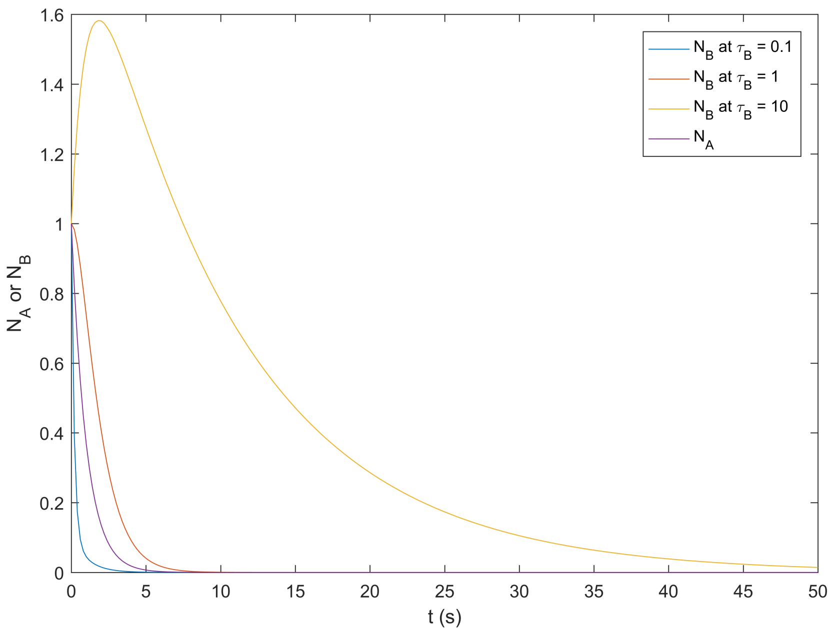

图 5: 不同 τ_B 下的结果

A 附录

A.1 源代码

A.1.1 第一题

```cpp
#include <iostream>
#include <string>
#include <iomanip>

using namespace std;

/*
================================================================================
Program Description
--------------------------------------------------------------------------------
This program contains the following functions:

1. char * double_2_ieee_method_1:
    This method converts a double to a c string of its IEEE 754 format, by using
    the definition of double-precision floating point format of IEEE 754 2008
    standard.

2. double ieee_2_double_method_1:
    This method converts a c string of IEEE 754 format to a double, by using the
    definition of double-precision floating point format of IEEE 754 2008
    standard.

3. char * double_2_ieee_method_2:
    This method converts a double to a c string of its IEEE 754 format, by
    explicitly converting the double to a long long, reading the converted
    number bit by bit, and writing these bits into the string.

4. double ieee_2_double_method_2:
    This method converts a c string of IEEE 754 format to a double, by
    converting the string to a long long and then explicitly convert the long
    long to a double.

The other methods are for presentation of the results.

Detailed explanation of the algorithm and the results can be found in the
report.
================================================================================
*/

// Convert by definition (IEEE 754)
// ** Subnormal numbers are NOT considered **
char * double_2_ieee_method_1(double x) {
    char * result = new char[64];
    int exponent = 1023;

    memset(result, '0', 64);

    cout << "Convert " << x << " by method 1:" << endl;

    // Special case: x=0
    // Default: x=positive zero
    if (x == 0) {
        return result;
    }

    // Determine sign bit
    if (x < 0) {
        result[0] = '1';
        x = -x;
    }

    // Determine exponent
    if (x >= 2) {
        while (x >= 2) {
            x /= 2;
            exponent++;
```

```
        }
    }
    else {
        if (x < 1) {
            while (x < 1) {
                x *= 2;
                exponent--;
            }
        }
    }
    for (int i = 11; exponent > 0; i--, exponent >>= 1) {
        result[i] = char('0' + (exponent & 1));
    }

    // Determine significant digits
    // Since subnormal numbers are NOT considered, at this moment, x must lies in [1,2).
    x--;
    for (int i = 12; x > 0 && i < 64; i++) {
        x *= 2;
        result[i] = x >= 1 ? '1' : '0';
        if (x >= 1) {
            x--;
        }
    }
    return result;
}

// Convert by definition (IEEE 754)
// ** Subnormal numbers are NOT considered **
double ieee_2_double_method_1(char * s) {
    double result = 1;
    double temp = 1;
    int exponent = 0;

    cout << "Convert back by method 1:" << endl;

    // Determine significant digits
    for (int i = 12;i<64;i++) {
        temp /= 2;
        result += temp * (int(s[i]) - int('0'));
    }

    // Determine exponent
    for (int i = 11; i > 0; i--) {
        exponent += (int(s[i]) - int('0')) * (1 << (11 - i));
    }
    if (exponent == 0) {
        return 0;
    }
    exponent -= 1023;
    if (exponent >= 0) {
        for (; exponent > 0; exponent--) {
            result *= 2;
        }
    }
    else {
        for (; exponent < 0; exponent++) {
            result /= 2;
        }
    }

    // Determine sign
    if (s[0] == '1') {
        result = -result;
    }

    return result;
}

// Convert by explicit type conversion
char * double_2_ieee_method_2(double x) {
    char * result = new char[64];
    long long * byte = (long long *)(&x);   // sizeof(long long)=sizeof(double)=8
```

```
    memset(result, '0', 64);

    cout << "Convert " << x << " by method 2;" << endl;

    // Fill result from the back 'bit' by 'bit'
    for (int i = 63; i >= 0; i--) {
        result[i] = (char)(int('0') + ((*byte) & 1));
        *byte >>= 1;
    }

    return result;
}

// Convert by explicit type conversion
double ieee_2_double_method_2(char * s) {
    long long result = 0;

    cout << "Convert back by method 2:" << endl;

    // Convert the string to long long
    for (int i = 0; i < 64; i++) {
        result <<= 1;
        result += (int)(s[i]) - (int)'0';
    }

    // Converting the long long to a double
    return *((double *)(&result));
}

// Method for output of cstrings
inline void stringout(char * s, int n) {
    for (int i = 0; i < n; i++) {
        cout << s[i];
    }
    cout << endl;
}

// Method for showcasing the examples
inline void showcase(double x) {
    char * s1, *s2;
    stringout(s1 = double_2_ieee_method_1(x), 64);
    stringout(s2 = double_2_ieee_method_2(x), 64);
    cout << setprecision(15) << ieee_2_double_method_1(s1) << endl;
    cout << setprecision(15) << ieee_2_double_method_2(s2) << endl;
}

int main() {
    double x = 0;
    char * s = new char[64];
    // Example from Chapter 1 Lecture note Page 50
    cout << "Example from Chapter 1 Lecture note Page 50" << endl;
    showcase(-222.111);

    // Other examples
    cout << "Try another example. Input the double:" << endl;
    cin >> x;
    showcase(x);

    cout << "Try another example. Input the string:" << endl;
    cin >> s;
    cout << setprecision(15) << ieee_2_double_method_1(s) << endl;
    cout << setprecision(15) << ieee_2_double_method_2(s) << endl;
    system("pause");
    return 0;
}
```

A.1.2 第三题

```
#include <iostream>
#include <iomanip>
#include <fstream>

using namespace std;
```

```
double a, b, tau_a, tau_b, t, delta;

double a_prev;
double b_prev;
ofstream out("result.txt");

void solve_core() {
// Method 1
// Naive iteration
/*
    a -= delta * a_prev / tau_a;
    b += delta * (a_prev / tau_a - b_prev / tau_b);
*/
// Method 2
// Naive iteration for a, a small correction for b
/*
    a -= delta * a_prev / tau_a;
    b += delta * ((a_prev + a) / (2 * tau_a) - b_prev / tau_b);
*/
// Method 3
// RK4 for a, no correction for b
/*
    double k[4];
    k[0] = - delta * a_prev / tau_a;
    k[1] = - delta * (a_prev + k[0] / 2) / tau_a;
    k[2] = - delta * (a_prev + k[1] / 2) / tau_a;
    k[3] = - delta * (a_prev + k[2]) / tau_a;
    a += (k[0] + 2 * k[1] + 2 * k[2] + k[3]) / 6;
    b += delta * ((a_prev + a) / (2 * tau_a) - b_prev / tau_b);
*/
// Method 4
// RK4 for a, after that, RK4 for b

    double k[4];
    k[0] = -delta * a_prev / tau_a;
    k[1] = -delta * (a_prev + k[0] / 2) / tau_a;
    k[2] = -delta * (a_prev + k[1] / 2) / tau_a;
    k[3] = -delta * (a_prev + k[2]) / tau_a;
    a += (k[0] + 2 * k[1] + 2 * k[2] + k[3]) / 6;

    k[0] = delta * (a_prev / tau_a - b_prev / tau_b);
    k[1] = delta * ((a_prev + a) / (2 * tau_a) - (b_prev + k[0] / 2) / tau_b); // Explicit time dependent in N_A, linear intrapolate
    k[2] = delta * ((a_prev + a) / (2 * tau_a) - (b_prev + k[1] / 2) / tau_b); // Explicit time dependent in N_A, linear intrapolate
    k[3] = delta * (a / tau_a - (b_prev + k[2]) / tau_b);
    b += (k[0] + 2 * k[1] + 2 * k[2] + k[3]) / 6;

}

void solver() {
    double t_now = 0;
    int n = 0;
    cin >> t;
    cin >> delta;
    out << "a = " << setw(16) << setprecision(4) << a << endl;
    out << "b = " << setw(16) << setprecision(4) << b << endl;
    out << "tau_a = " << setw(16) << setprecision(4) << tau_a << endl;
    out << "tau_b = " << setw(16) << setprecision(4) << tau_b << endl;
    out << "t = " << setw(16) << setprecision(4) << t << endl;
    out << "delta = " << setw(16) << setprecision(4) << delta << endl;
    a_prev = a;
    b_prev = b;
    while (t_now <= t) {
        n++;
        out << setw(16) << setprecision(4) << t_now << setw(16) << setprecision(4) << a << setw(16) << setprecision(4) << b << endl;
        solve_core();
        t_now += delta;
        a_prev = a;
        b_prev = b;
    }
```

```
    out << setw(16) << setprecision(6) << t_now << setw(16) << setprecision(4) << a << setw(16) << setprecision(4) << b << endl;
    a = ((t - t_now) / delta + 1) * (a - a_prev) + a;
    b = ((t - t_now) / delta + 1) * (b - b_prev) + b;
}

int main() {
    a = b = 1;
    cin >> tau_a >> tau_b;
    solver();
    cout << setw(16) << setprecision(4) << a << setw(16) << setprecision(4) << b << endl;
    getchar();
    getchar();
    return 0;
}
```