mathalgorithm 参考文档

version 0.7 by 李想

开发时所用编译器及标准：gcc 10.1.1, C++14

目录

1 数值计算库 computational

头文件和源文件

- computational.h
- computational.cpp

命名空间 malg

简介 数值计算库用于大整数计算、多项式计算、矩阵计算、求解线性方程组、一元方程求根、插值、曲线拟合与函数逼近、数值微积分以及常微分方程求解。

使用时请包含如下头文件：

```
#include "computational.h"
```

1.1 大整数 BigInt

构造函数原型

```
BigInt (string str);
BigInt (int num);
```

参数

- str - 数字（以字符串表示）
- num - 数字（以整型表示）

示例 作下列计算：

$$p = 23948576235 \times 823761298$$

$$q = 2^{100}$$

并将结果输出到屏幕上。代码如下：

```
#include<iostream>
#include "computational.h"
int main()
{
    malg::BigInt a("23948576235");
    malg::BigInt b("823761298");
    malg::BigInt c("2");
    std::cout<<"p="<<a*b<<std::endl;
    std::cout<<"q="<<(c^100)<<std::endl;
}
```

可能的输出：

```
p=19727910244595553030
q=1267650600228229401496703205376
```

复杂度

(1) 大整数加减法使用朴素的按位加减法实现，复杂度 $O(n)$，其中 n 是数字的位数。

(2) 大整数乘法使用快速数论变换（Schönhage-Strassen 算法）实现，复杂度 $O(n \log n \log\log n)$，其中 n 是数字的位数。

(3) 大整数乘方使用快速幂实现，乘法的次数为 $O(\log m)$，其中 m 是幂次。

1.2 多项式 Polynomial

构造函数原型

```
template <class T>
Polynomial<T> (const vector<T>& r);
template <class T>
Polynomial<T> (const vector<T>& r,int deg);
```

参数

- r - 系数向量（按升幂排序）
- deg - 多项式次数

示例 定义两个整型多项式

$$f = 1 + x + x^2, \quad g = 5 - 3x + x^3$$

计算它们的乘积 $h = fg$ 以及 $h(5)$ 的值，并将结果输出在屏幕上。代码如下：

```
#include <iostream>
#include "computational.h"
int main()
{
    malg::Polynomial<int> f({1,1,1},2);
    malg::Polynomial<int> g({5,-3,0,1},3);
    malg::Polynomial<int> h=f*g;
    std::cout<<"h(x)="<<h<<std::endl;
    std::cout<<"h(5)="<<h(5)<<std::endl;
}
```

可能的输出：

```
h(x)=5+2x+2x^2-2x^3+1x^4+1x^5
h(5)=3565
```

复杂度

(1) 多项式乘法采用分治法实现，时间复杂度 $O(n^{1.58})$，其中 n 是多项式次数。但对于特殊的类型，采用了如下优化：

- 浮点型和复数浮点型多项式乘法使用快速傅里叶变换实现，复杂度 $O(n \log n)$
- 无符号整型多项式乘法使用快速数论变换实现，复杂度 $O(n \log n \log \log n)$

(2) 多项式求值采用秦九韶算法实现，时间复杂度 $O(n)$，其中 n 是多项式次数。

1.3 线性方程组求解 LinearSolve

函数原型

```
template <class T>
Matrix<T> LinearSolve(const Matrix<T>& A, const Matrix<T>& b);
template <class T>
Matrix<T> LinearSolve(const Matrix<T>& A, const Matrix<T>& b,string str);
```

参数

- A - 系数矩阵
- b - 常数项向量
- str - 求解方法

示例 求解线性方程组 $\mathbf{Ax} = \mathbf{b}$，其中

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & -1 \\ 1 & 2 & -2 \\ -2 & 1 & 1 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

代码如下：

```cpp
#include <iostream>
#include "computational.h"
int main()
{
    malg::Matrix<double> A({{1,1,-1},{1,2,-2},{-2,1,1}},3,3);
    malg::Matrix<double> b({{1}, {0}, {1}}, 3, 1);
    std::cout << malg::LinearSolve(A, b);
}
```

可能的输出：

```
[2]
[2]
[3]
```

方法 参数中的 str 用于选择求解线性方程组的方法。

- “LUP” - LUP 分解法（默认）
- “Gauss” - 高斯列主元消去法
- “LU” - LU 分解法
- “Thomas” - 追赶法

- “Cholesky” - 平方根法
- “Jacobi” - 雅克比迭代法
- “Gauss-Seidel” - 高斯-赛德尔迭代法
- “SOR” - 超松弛迭代法

复杂度

(1) 默认的 LUP 分解法具有 $O(n^3)$ 的时间复杂度，其中 n 是矩阵阶数。除此之外，高斯列主元消去法、LU 分解法、平方根法都具有与其相同的时间复杂度。注意，LU 分解法只适用于求解所有顺序主子式都大于 0 的矩阵，平方根法只适用于求解对称正定矩阵。

(2) 追赶法只适用于求解三对角矩阵，在严格对角占有的条件下具有数值稳定性。时间复杂度是 $O(n)$。

(3) 雅克比迭代法、高斯-赛德尔迭代法、超松弛迭代法适用于求解大规模矩阵，其中的迭代次数已经固定了上界（默认 20），因此其时间复杂度是 $O(n^2)$。注意，只有当迭代矩阵的谱半径小于 1 时，迭代法才收敛。

注意 如果矩阵的类型声明为 Matrix<int>：

```
Matrix<int> A({{1,1,-1},{1,2,-2},{-2,1,1}},3,3);
Matrix<int> b({{1}, {0}, {1}}, 3, 1);
malg::LinearSolve(A, b);
```

则可能不会返回预期的结果。因为在求解线性方程组时需要用到除法运算，而整型 int 变量的除法只会返回整数部分，导致结果失真。

1.4 插值 Interpolation

函数原型

```
template <class T>
T Interpolation(const Matrix<T> &base, T data);
template <class T>
T Interpolation(const Matrix<T> &base, T data, string str);
template <class T>
T Interpolation(const Matrix<T> &base, const Matrix<T> &T);
template <class T>
T Interpolation(const Matrix<T> &base, const Matrix<T> &T, string str);
```

参数

- base - 已知的数据矩阵
- data - 待插值的数据（数或向量）

示例 现在有一张数据表格，其中？处的数据未知：

x	0.4	0.5	0.6	0.7	0.8	0.9
y	-0.916	-0.693	?	?	-0.223	-0.105

请利用表格中已知的数据，利用插值补全未知数据。代码如下：

```
#include <iostream>
#include "computational.h"
int main()
{
    malg::Matrix<double> base(
        {{0.4,-0.916},{0.5,-0.693},{0.8,-0.223},{0.9,-0.105}});
    malg::Matrix<double> data({{0.6},{0.7}},2,1);
    malg::Matrix<double> result=malg::Interpolation(base,data);
    std::cout<<result;
}
```

可能的输出：

```
[-0.536333]
[-0.379667]
```

方法 参数中的 str 用于选择插值的方法。

- linear - 分段线性插值（已知数据只含函数值时默认）

- piecewise Hermite - 分段埃尔米特插值（已知数据含导数值时默认）
- Lagrange - 拉格朗日插值
- Newton - 牛顿插值
- Hermite - 埃尔米特插值
- spline - 样条插值

注意

(1) 当待插值的数据是一个向量的时候，已知数据矩阵 base 的 x 向量和待插值的数据向量 data 必须按递增排序。这样要求是为了尽可能提高程序运行的速度。

(2) 拉格朗日插值和牛顿插值的效果是完全一样的，但内部实现方式不同。牛顿插值显著快于拉格朗日插值。建议使用牛顿插值，弃用拉格朗日插值。

1.5 定积分 Integrate

函数原型

```
template <class T>
T Integrate(std::function<T(T)> f, T a,T b);
template <class T>
T Integrate(std::function<T(T)> f, T a,T b,string str);
template <class T>
T Integrate(Polynomial<T> f, T a,T b);
```

参数

- f - 被积函数
- a - 积分区间 $[a,b]$ 的左端点
- b - 积分区间 $[a,b]$ 的右端点
- str - 求解方法

示例 计算定积分

$$q=\int_0^1 \frac{4}{1+x^2}\mathrm{d}x$$

结果以 10 位有效数字输出。代码如下：

```cpp
#include <iostream>
#include "computational.h"
int main()
{
    auto f = [](double x) { return 4.0 / (1 + x * x); };
    auto q = malg::Integrate<double>(f, 0, 1);
    N(q, 10);
}
```

可能的输出：

```
3.141592654
```

方法 参数中的 str 用于选择求解定积分的方法。

- “Romberg” - 龙贝格算法（默认）
- “trapz” - 梯形公式
- “Simpson” - 辛普森公式
- “Simpson3/8” - 辛普森 3/8 公式

- “Milne” - 米尔尼公式
- “Newton-Cotes” - 牛顿-柯特斯型求积公式
- “Gauss-Legendre” - 高斯-勒让德求积公式
- “compound trapz” - 复化梯形公式
  （类似地，在名称前面加 “compound” 就可表示复化的方法还有辛普森公式、辛普森 3/8 公式和米尔尼公式）

1.6 求导数 D

函数原型

```
template <class T>
T D(std::function<T(T)> f, T x);
template <class T>
T D(std::function<T(T)> f, T x, string str);
template <class T>
std::function<T(T)> D(std::function<T(T)> f);
template <class T>
std::function<T(T)> D(int n, std::function<T(T)> f);
```

参数

- f - 函数
- x - 求导处
- n - 阶数
- str - 方法

示例 设

$$f(x) = x^2 e^x$$

求 $f(x)$ 在 $x = 0$ 处的二阶导数 $f''(0)$，并把结果输出到屏幕上。代码如下：

```
#include <iostream>
#include "computational.h"
int main()
{
    auto f=[](double x){ return x*x*exp(x); };
    auto g=malg::D<double>(2,f);
    std::cout<<g(0)<<std::endl;
}
```

可能的输出：

```
2
```

方法 参数中的 str 用于选择求导数的方法。

- “center” - 中心差商法（默认）
- “forward” - 向前差商法
- “backward” - 向后差商法

复杂度 中心差商法、向前差商法和向后差商法的复杂度均为 $O(cn)$，其中 n 是求导的阶数，c 是计算一次函数值的时间。

精度

(1) 中心差商法具有二阶精度。

(2) 向前差商法和向后差商法具有一阶精度。

注意

(1) 在函数的不可导点处求导，将会返回难以预料的结果。理论上来说，向前差商法将返回右导数，向后差商法将返回左导数，而中心差商法返回二者的平均值。当左导数不等于右导数时，函数在该点不可导，此时用三种方法所得到的结果很可能各不相同。在实际中由于数值运算难以避免的截断误差和舍入误差，在不可导点处求导的结果是无法预料的。

(2) 当求导阶数很高时，误差将变得显著。不建议作高阶导数的数值运算。