Programming Homework2

Shuang Hu

October 29, 2022

问题描述

设计并实现一个软件包用于求解插值问题，要求实现牛顿插值和 Hermite 插值算法，并利用你的软件包进行一些测试，进而解决一些问题:

1. 用等距节点的牛顿插值拟合曲线 $f(x) = \frac{1}{1+x^2}$，并观测龙格现象。
2. 用 Chebyshev 多项式零点作为插值节点，用牛顿插值拟合曲线 $f(x) = \frac{1}{1+25x^2}$，观察这种做法对龙格现象的改善。
3. 利用 Hermite 插值模拟小汽车的运动距离和行进速度，并判断其是否超速。
4. 利用 Newton 插值拟合冬蛾幼虫的种群增长，并判断其是否会灭绝 (die out)。

明确算法的契约是我们进行算法与程序设计的第一步（具体概念见课本 1.2 节），所有的具体设计都要以算法的契约作为根本依据。
对于本次需要讨论的插值问题，我们可以写出这样的契约:

- 输入 (input)：一系列插值节点 (x_i, y_i)，对于 Hermite 插值问题，还需要一些点处的导数值，如 $(x_i, y_i'), (x_i, y_i''), \cdots, (x_i, y_i^{(n)})$。
- 先决条件 (precondition)：插值节点需要满足函数的定义，即: $x_i \neq x_j (\forall i \neq j)$。
- 输出 (output)：多项式 $p(x)$。
- 后置条件 (postcondition)：$p(x_i) = y_i$, $p^{(n)}(x_i) = y_i^{(n)}$，不存在 $\deg q < \deg p$ 使得 $q(x)$ 满足上述条件。

用算法的契约指导类的设计

在明确问题之后，具体程序和数据结构的设计都要为算法的契约服务。下面我们对上一页所述的契约进行分析，进而明确我们的设计需要哪些类，以及各类需要满足哪些要求。

- 存放输入变量:class InterpCondition
- 存放程序输出:class Polynomial
- 插值基类:class Interpolation
- 实现 Newton 插值:class Newton:public Interpolation
- 实现 Hermite 插值:class Hermite:public Interpolation

这是一个大体的框架。下面我们需要充分考虑每个类的具体需求，进而给出每个类的成员变量/成员函数的设计。

class InterpCondition

这个类用于存储插值问题的所有输入条件，且对 Newton 和 Hermite 插值问题均适用。

- **成员变量**:
  1. int n: **表示插值条件总个数。**
  2. std::vector<double> interpNodes: **表示插值节点的横坐标。**
  3. std::vector<std::vector<double> > interpValues: **表示与横坐标对应的函数值和各阶导数值。**
- **成员函数**:
  1. Interpolation(const std::vector<double>& interpNodes, const std::vector<double>& interpValues);
public **成员函数**，构造函数，记录当前插值问题的所有插值信息。
  2. void addNodes(std::vector<double> x,std::vector<std::vector<double> > values);
public **成员函数**，添加一些插值节点。
  3. void removeNodes(std::vector<double> x);
public **成员函数**，删除一些插值节点。
  4. std::vector<double> getNodes();
public **成员函数**，获取插值节点的横坐标。
  5. std::vector<std::vector<double> > getInterpValues;
public **成员函数**，获取插值节点的纵坐标。

这个类用于存放多项式，且要求实现多项式的各项运算，包括加减法，数乘，乘法，取值等等。

- 成员变量
  1. std::vector<double> Coefficients: 存放多项式的系数。
  2. std::vector<int> Exponents: 存放多项式的次数。
- 成员函数和友元函数:
  1. Polynomial()=default;
     默认构造函数, 什么也不做。
  2. Polynomial(double a); **类型转换构造函数**, 构造单项式 $p(x) = a$，有执行类型转换的效果。
  3. Polynomial(double a,double b); **构造函数**，构造单项式 $p(x) = ax + b$。
  4. Polynomial(std::vector<double> coe,std::vector<int> exp); **构造函数**，初始化一个一般的多项式。
  5. friend ostream& operator«(ostream& os,Polynomial& p); **友元函数**，输出多项式的所有相关信息。

6. friend Polynomial operator+(const Polynomial& p1,const Polynomial& p2);
   友元函数，实现多项式加法。
7. Polynomial operator-();
   public **成员函数**，实现多项式取负。
8. friend Polynomial operator-(const Polynomial& p1,const Polynomial& p2);
   友元函数，实现两个多项式相减。
9. friend Polynomial operator*(const Polynomial& p1,const Polynomial& p2);
   友元函数，实现两个多项式相乘。
10. double operator()(double x);
    public **成员函数**，实现多项式的求值，即求 p(x) 的值。
11. void output(std::string filename);
    public **成员函数**，将多项式的相关信息输出到目标文件中。
12. void draw(std::string filename);
    public **成员函数**，利用目前的多项式给出一个作图脚本，以便可视化插值多项式。

class Interpolation

这个类用于记录插值问题所需要的信息，包括插值点和差商表，并给出解决插值问题的统一接口。

- 成员变量:
  1. InterpCondition ic:protected **成员变量**, 记录插值条件。
  2. std::vector<std::vector<double> > diffTable:protected **成员变量**, 记录差商表。
- 成员函数:
  1. Interpolation(const InterpCondition& ic);
     构造函数, 记录所有的插值条件。
  2. void generateTable() = 0;
     public **成员函数**, 纯虚函数，用于存放差商表。
  3. Polynomial solve() = 0;
     public **成员函数**, 纯虚函数，用于插值问题的最终实现。

这两个 class 都由基类 Interpolation 派生而来。因此在这两个类内，都仅需给出纯虚函数 void generateTable() 和 Polynomial solve() 的 override 即可。

明确类与类之间的关系 (UML 图)。

设计文档

- 如上所述是一个非常简易的设计文档。
- 我们建议，在正式开始编程之前，应当先从问题整体出发，先写出这样的一份设计文档。这样做至少有如下好处：
  1. ”大处着眼”：在撰写设计文档时，你可以暂时摆脱各种繁杂细节的困扰，把主要精力放在问题的理解和模块的划分上。这有助于你从整体上思考问题，最大程度避免了编码过程中的瞻前顾后。
  2. ”小处着手”：在给出一个完整的设计后，编程实现就是相对容易了。此时的编程只需要考虑多个”小型模块”的实现，每个模块的编写都不会耗费太多精力，最后进行适当的连接即可。这种工作方式对合作开发也非常合适。
  3. 易于调试：在有了一个完整的设计后，我们可以分别调试每个子模块，最大程度上避免了”找不到错误”的尴尬。
  4. 易于接手：你认为你的合作者更愿意读文档还是读源代码？
- 设计文档是需要打磨的，很难一步到位。所以，在完成初版设计之后，请再多读几遍，确认设计的合理性之后，再开始编程。

B 题和 C 题，只要作图正确，就给全对了。
D 题 (b) 问很多人被扣了分，因为利用 $x > 13$ 时的”最大速度”来判定超速是不合理的。根据多项式的性质，在 $x \to \infty$ 时，$p(x) \to \infty$，我们可以想象在 $x > 13$ 后，这个多项式的取值一定会增大到一个非常夸张的数值，这一定是不符合实际的。导致这个现象的原因并非一些同学所写的 “Runge 现象”，而纯粹是因为我们缺少 $x > 13$ 时的速度信息。
E 题和 D 题差不多道理，但 E 题的图线更为离谱: 某一种幼虫居然会“起死回生”！这又一次提醒我们，插值多项式拟合有时候并不一定如我们想的那么万能。

Thanks!
Questions?