数值分析编程作业1报告

> 学号：3210101440；姓名：胡子豪

Requirements

本次编程作业的主要目标如下：

- 利用`C++`构造抽象基类`EquationSolver`，并由此实现三个派生类，用于二分法、牛顿迭代法和割线法求解方程的根；
- 针对剩余的$B\ to\ F$题，利用上述实现的类进行相关的求解与分析。

Problem-solving ideas

本次编程作业的实现主要分为两个方面：类及其成员函数的构建与相应题目的代码构建。具体来说是先实现类的构建，再将此应用于已撰写好的题目代码之上。

- 类的构建：
  1. 创建`Solve.h`与`Solve.cpp`，将类的声明与实现分开；
  2. 利用`virtual`关键字构造抽象基类的成员函数`solve()`，再对三个派生类的成员函数进行重构；
  3. 按照二分法、牛顿迭代法和割线法实现具体的函数代码。
- 解决具体问题：根据题目要求撰写代码，调用相关函数进行问题求解并输出有效信息。

Code analysis

核心代码分析如下：

- 构造一个用于储存参数的类`Init`用于储存`solve()`函数用到的参数：

  ```cpp
  class Init
  {
  public:
      double x0, x1;
      function<double(double)> func, dfunc;
      Init(double left, double right, function<double(double)> func);
      Init(double x0, function<double(double)> func, function<double(double)> dfunc);
  };
  ```

  在使用时，`Init* init;`的目的在于使函数接口统一化，便于实现抽象类的构造（实质为纯虚函数的构造）。
- 派生类的`solve()`函数（以二分法为例）：

  ```cpp
  vector<double> solve(Init* init);
  ```

  1. 函数返回类型选用`vector<double>`是考虑到有多个返回值；
  2. 传入的参数是类指针`init`，其包括`solve()`函数中所用到的参数；
  3. 三种方法的具体实现按照相应方法的思路即可。
- `lambda`函数块：

```cpp
auto func1 = [](double x) { return 1.0/x - tan(x); };
```

1. 这个 `lambda` 函数没有名称，而是将其赋值给变量 `func1`。`func1` 接受一个参数 `x`。
2. 此时 `func1` 可作为参数首先传入进入储存参数的类中，再通过这个类的实例对象传入 `solve()` 函数中。

Test results and Analysis

- `B` 题：

```
PS D:\Code\NAcode\project1> make run
g++ -c B.cpp -o B.o
g++ -c Solve.cpp -o Solve.o
g++ B.o Solve.o -o project1
./project1
---------------------B
Question1:
root: 0.860336, width of interval: 1.19842e-05, maximum iterative count: 16
---------
Question2:
root: 0.641182, width of interval: 7.62939e-06, maximum iterative count: 16
---------
Question3:
root: 1.82938, width of interval: 7.62939e-06, maximum iterative count: 17
---------
Question4:
Error for preconditions: sgn(f(a)) == sgn(f(b))
root: 0, width of interval: 0, maximum iterative count: 0
---------
PS D:\Code\NAcode\project1>
```

- `C` 题：

```
PS D:\Code\NAcode> make run
g++ -c C.cpp -o C.o
g++ C.o Solve.o -o project1
./project1
---------------------C
x0 = 4.5:
root: 4.49341, maximum iterative count: 2
---------
x0 = 7.7:
root: 7.72525, maximum iterative count: 4
---------
PS D:\Code\NAcode>
```

- `D` 题：

```
PS D:\Code\NAcode> make run
g++ -c D.cpp -o D.o
g++ D.o Solve.o -o project1
./project1
---------------------D
Question1:
x_{n}: 3.13427, x_{n-1}: 3.12974, maximum iterative count: 11
---------
Question2:
x_{n}: 1.30633, x_{n-1}: 1.30632, maximum iterative count: 8
---------
Question3:
x_{n}: -0.188685, x_{n-1}: -0.188683, maximum iterative count: 6
---------
PS D:\Code\NAcode>
```

- `E` 题：

```
g++ -c E.cpp -o E.o
g++ -c Solve.cpp -o Solve.o
g++ E.o Solve.o -o project1
./project1
----------------------E
Bisection:
root: 0.17, width of interval: 7.6e-06, maximum iterative count: 16
---------
Newton, x0 = 0.15:
root: 0.17, maximum iterative count: 3
---------
Secant:
x_{n}: 0.17, x_{n-1}: 0.17, maximum iterative count: 3
---------
PS D:\Code\NAcode\project1>
```

- `F` 题：

（c）选取相距33°较近的点：

```
PS D:\Code\NAcode> make run
g++ -c F.cpp -o F.o
g++ -c Solve.cpp -o Solve.o
g++ F.o Solve.o -o project1
./project1
----------------------F
(a)Newton
alpha = 32.9724
---------
(b)Newton, x0 = 0.575959
alpha = 33.1689
---------
(c)Secant, x0 = 0.575959, x1 = 1.0472
x_{n}: 33.1689, x_{n-1}: 33.1675
---------
PS D:\Code\NAcode>
```

（c）选取相距33°较远的点：

```
PS D:\Code\NAcode> make run
g++ -c F.cpp -o F.o
g++ F.o Solve.o -o project1
./project1
----------------------F
(a)Newton
alpha = 32.9724
---------
(b)Newton, x0 = 0.575959
alpha = 33.1689
---------
(c)Secant, x0 = 0.575959, x1 = 104.72
x_{n}: -326.831, x_{n-1}: -326.834
---------
PS D:\Code\NAcode>
```

可以看出，使用割线法时，当选取的另一个初始值与33°相差甚远时，得到的结果与理论结果相差较远。

原因在于：如果初始取值远离理论根，割线法会产生发散，即算法不会收敛到根，而是在无限迭代中趋向无穷大或无穷小。这会导致计算失败或错误的根，本题即是产生错误的根。