数值分析 - 第一次上机作业 - 实验报告

樊睿
强基数学 2001 班

September 22,2022

摘要

本文详细介绍了非线性方程的几种算法的实现，并用它们解决了一些实际问题。

1 仿函数类

为了使算法能够接受函数作为形参，我们在实现算法之前，先实现了抽象类“仿函数类” `Function`。所有仿函数均由这个抽象类派生。类中重载了 `()` 运算符（在语法上是强制类型转换运算符）作为纯虚函数，返回函数值。这样若定义了 `f` 作为某派生类的对象，则调用 `f(x)` 就返回了 x 的函数值。如果还需要支持函数求导，则在类中再定义一个成员函数 `d`。调用 `f.d(x)` 返回 $f'(x)$。

如要使用该类，需要包含该头文件。使用该类时需先定义一个 `Function` 类的派生类，再创建这个派生类的对象。必须重载 `()` 运算符且必须仅有一个类型和该函数类型相同的参数。求导成员函数可以不定义。若对导数未定义的函数求导，则默认使用导数的原始定义，即差商的极限（但因为我们不可能真的取到“无穷小量”，所以直接用原式定义求导的精度是非常差的，建议能自己定义的都自己定义）。

附源码。

头文件：

```
template <class type>
class Function{
public:
```

```
    virtual type operator ()(const type& x) const = 0;
    virtual type d(const type& x) const {throw 1;}
};
```

使用举例（定义 $f(x) = \sin(x)$）：

```
class Sin : public Function<double>{
public:
    virtual double operator ()(const double& x) const {
        return sin(x);
    }
    virtual double d(const double& x) const {
        return cos(x);
    }
} f;
```

2 方程求解器类

定义抽象类“方程求解器类”（`EquationSolver`），三个求解器（二分法、牛顿法、割线法）均由该抽象类派生。源码如下：

```
template <class type>
class EquationSolver{
protected:
    virtual type solve() = 0;
};
```

2.1 二分法

参考课本 Algo 1.9 实现。

输入 f, a, b （必须指定）和 M, δ（可以指定。若不指定，则默认 $M = 100, \delta = 10^{-6}$。必须保证 f 连续且 $f(a)f(b) \le 0$。输出 f 在 $[a, b]$ 上的一个近似零点 x^*，保证 $f(x^*) < \varepsilon$，或存在一个零点 α 满足 $|x^* - \alpha| < \delta$。若迭代次数超过 M，则抛出异常。

这里的 ε 是舍入误差，在 64 位系统下是 2^{-52}。但若 ε 过小，算法的效率将严重受到精度误差的影响，不便于分析。考虑到舍入误差的累积以及 C++ math.h 中 `sin`、`cos`、`exp` 等库函数精度误差，本项目中取 $\varepsilon = 10^{-12}$。

源码如下：

```
template <class type>
class EquationSolver{
protected:
    virtual type solve() = 0;
};

template <class type>
class Bisection : public EquationSolver <type> {
private:
    const Function<type> &f;
    type a, b, delta;
    int M;
public:
    Bisection(const Function<type> &f, const type &a, const type &b, const int &M =
        f(f), a(a), b(b), delta(delta), M(M) {}
    virtual type solve() {
        if (f(a) * f(b) > eps) throw "Invalid Interval!";
        type h = b - a, u = f(a), c, w, x = a;
        int k = 1;
        while (k <= M) {
            h /= 2, c = x + h, w = f(c);
            if (fabs(h) < delta || fabs(w) < eps) break;
            else if (w * u > 0) x = c;
            ++ k;
        }
        if (k > M) std::cout << "Time Limit Exceeded!" << std::endl;
        std::cerr << "Bisection : times = " << k << ", " << "delta = " << h << std:::
        return c;
    }
```

```
};
```

使用举例:

```
// f 是仿函数派生类的一个对象。
double r = Bisection(f, 0, 1).solve();
double r1 = Bisection(f, 0, 1, 20, 1e-3).solve();
```

2.2 牛顿法

参考课本 Algo 1.14 实现。

输入 f, x_0（必须指定）和 M（可以指定。若不指定，则默认 $M = 10$）。输出 f 在 x_0 附近的近似零点 x^*。保证 $f(x^*) < \varepsilon$。若迭代次数超过 M，则抛出异常。

另外注意：当 x_0 距离 f 的零点过远时，则不保证输出结果的正确性；f 必须定义导数，否则会抛出异常。

源码如下:

```
template <class type>
class Newton : public EquationSolver <type> {
private:
    const Function<type> &f;
    type x0;
    int M;
public:
    Newton(const Function<type> &f, const type &x0, const int& M = 10) :
        f(f), x0(x0), M(M){}
    virtual type solve() {
        type x = x0, u;
        int k = 1;
        while (k <= M) {
            u = f(x);
            if (fabs(u) < eps) break;
            x -= u / f.d(x);
            ++ k;
        }
```

```
        if (k > M) std::cout << "Time Limit Exceeded!" << std::endl;
        std::cerr << "Newton : times = " << k << std::endl;
        return x;
    }
};
```

使用举例:

```
// f 是仿函数派生类的一个对象。必须定义导函数。
double r = Newton(f, 0).solve();
double r1 = Newton(f, 0, 5, 1e-3).solve();
```

2.3 割线法

参考课本 Algo 1.19 实现。

输入 f, x_0, x_1（必须指定）和 M, δ（可以指定，若不指定则默认 $M = 30, \delta = 10^{-6}$）。输出 f 在 x_0 附近的近似零点 x^*。保证存在 $f(x^*) < \varepsilon$，或存在一个零点 α 满足 $|x^* - \alpha| < \delta$。若迭代次数超过 M，则抛出异常。

另外注意：当 x_0, x_1 距离 f 的零点过远时，则不保证输出结果的正确性。

源码如下:

```
template <class type>
class Secant : public EquationSolver <type> {
private:
    const Function<type> &f;
    type a, b, delta;
    int M;
public:
    Secant<type>(const Function<type> &f, const type &a, const type &b, const in
        f(f), a(a), b(b), delta(delta), M(M) {}
    virtual type solve() {
        type x0 = a, x1 = b, u = f(x1), v = f(x0), s;
        int k = 2;
        while (k <= M) {
            if (fabs(u) > fabs(v)) std::swap(x0, x1), std::swap(u, v);
```

```
                s = (x1 - x0) / (u - v);
                x0 = x1, v = u;
                x1 -= u * s, u = f(x1);
                if (fabs(x0 - x1) < delta || fabs(u) < eps) break;
                ++ k;
            }
            if (k > M) std::cout << "Time Limit Exceeded!" << std::endl;
            std::cerr << "Secant : times = " << k << ", " << "delta = " << fabs(x0 -
            return x1;
        }
    };
```

使用举例:

```
// f 是仿函数派生类的一个对象。
double r = Secant(&f, 0, 1).solve();
double r = Newton(&f, 0, 1, 5, 1e-3).solve();
```

3 问题求解

3.1 第二题，二分法的测试

首先定义函数:

```
class F1 : public Function <double> {
    virtual double operator () (const double& x) const {
        return 1.0 / x - tan(x);
    }
}f1;

class F2 : public Function <double> {
    virtual double operator () (const double& x) const {
        return 1.0 / x - pow(2, x);
    }
}f2;
```

```
class F3 : public Function <double> {
    virtual double operator () (const double& x) const {
        return pow(2, -x) + exp(x) + 2 * cos(x) - 6;
    }
}f3;

class F4 : public Function <double> {
    virtual double operator () (const double& x) const {
        return (((x + 4) * x + 3) * x + 5) / (((2 * x - 9) * x + 18) * x - 2);
    }
}f4;
```

然后按题意进行求解：

```
cout << "2(1)\n" << Bisection<double>(f1, 0.0, PI/2).solve() << endl;
cout << "2(2)\n" << Bisection<double>(f2, 0.0, 1.0).solve() << endl;
cout << "2(3)\n" << Bisection<double>(f3, 1.0, 3.0).solve() << endl;
cout << "2(4)\n" << Bisection<double>(f4, 0.0, 4.0).solve() << endl;
```

得到结论

```
2(1)
Bisection : times = 21, delta = 7.49014e-07
0.860333
2(2)
Bisection : times = 20, delta = 9.53674e-07
0.641185
2(3)
Bisection : times = 21, delta = 9.53674e-07
1.82938
2(4)
Bisection : times = 22, delta = 9.53674e-07
0.117877
```

第一行的 `times` 是二分法实际迭代的次数，第二行是求得的根。

验证,发现前三个解确实是对应方程的一个根,但第四个解 $x_0 = 0.117877$ 不是方程的近似根：事实上，$f(x_0) \to \infty$。原因在于 x_0 是分母 $2x^3 - 9x^2 +$

$18x-2$ 的近似零点。设分母精确的零点为 α，当 $x\to\alpha^-$ 时 $f(x)\to-\infty$，$x\to\alpha^+$ 时 $f(x)\to+\infty$。因此二分法会“误认为” α 是方程的一个根。从这里可以看出，二分法“$f(x)$ 在 $[a,b]$ 上连续”的条件是必要的。

3.2 第三题，牛顿法的测试

首先定义函数。由 $f(x)=x-\tan x$ 计算导数可得 $f'(x)=1-\dfrac{1}{\cos^2 x}$

```
class G : public Function <double> {
    virtual double operator () (const double& x) const {
        return x - tan(x);
    }
    virtual double d(const double& x) const {
        double t = cos(x);
        return 1 - 1.0 / (t * t);
    }
}g;
```

然后按题意进行求解：

```
cout << "3(1)\n" << Newton<double>(g, 4.5).solve() << endl;
cout << "3(2)\n" << Newton<double>(g, 7.7).solve() << endl;
```

得到结论：

```
3(1)
Newton : times = 5
4.49341
3(2)
Newton : times = 6
7.72525
```

比较迭代次数还可以看出，精度相同时，牛顿法的迭代次数明显比二分法少。

3.3 第四题，割线法的测试

首先定义函数：

```
class H1 : public Function <double> {
    virtual double operator () (const double& x) const {
        return sin(x / 2) - 1;
    }
}h1;

class H2 : public Function <double> {
    virtual double operator () (const double& x) const {
        return exp(x) - tan(x);
    }
}h2;

class H3 : public Function <double> {
    virtual double operator () (const double& x) const {
        return ((x - 12) * x + 3) * x + 1;
    }
}h3;
```

然后按题意进行求解:

```
cout << "4(1)\n" << Secant<double>(h1, 0.0, PI/2).solve() << endl;
cout << "4(2)\n" << Secant<double>(h2, 1.0, 1.4).solve() << endl;
cout << "4(3)\n" << Secant<double>(h3, 0.0, -0.5).solve() << endl;
```

得出结论:

```
4(1)
Secant : times = 29, delta = 1.26763e-06
3.14159
4(2)
Secant : times = 15, delta = 2.59851e-09
1.30633
4(3)
Secant : times = 8, delta = 2.83825e-09
-0.188685
```

可见割线法的效率并没有理论中那样高。输出中间结果时可以发现，由于函数 sin 的精度损失严重（仅能保留约 10 位有效数字），在 x_n 很接近函数零点时，收敛速度已远低于 1.618 阶，甚至在某些情况下会低于二分法的收敛速度。

3.4 第五题，量筒

按题意构建模型。只需求解方程

$$f(h) = L[\frac{1}{2}\pi r^2 - r^2 \arcsin\frac{h}{r} - h(r^2 - h^2)^{\frac{1}{2}}] - V = 0 \tag{1}$$

计算导数，可得

$$f'(x) = -2Lh(r^2 - h^2)^{\frac{1}{2}} \tag{2}$$

分别用三种算法求解该方程。

函数的定义：

```
class P : public Function <double> {
    virtual double operator () (const double& h) const {
        return L * (PI/2 * r * r - r * r * asin(h / r) - h * sqrt(r * r - h * h)) -
    }
    virtual double d(const double& h) const {
        return L * (-2 * sqrt(r * r - h * h));
    }
private:
    double L, r, V;
public:
    P(double L, double r, double V) : L(L), r(r), V(V) {}
};
```

将题目中数据代入，调用算法，求解：

```
cout << "5\n";
P p(10, 1, 12.4);
cout << Bisection<double>(p, 0, 1, 20, 0.001).solve() << endl;
cout << Newton<double>(p, 0.5).solve() << endl;
cout << Secant<double>(p, 0, 1, 20, 0.001).solve() << endl;
```

结论：

```
5
Bisection : times = 10, delta = 0.000976562
0.166992
Newton : times = 5
0.166166
Secant : times = 4, delta = 0.000474122
0.166164
```

因此（保留两位小数）$h = 0.17\text{ft}$。

3.5 第六题，汽车

按题意定义函数并计算导数：

```
class Q : public Function <double> {
    virtual double operator () (const double& a) const {
        double _a = a * PI / 180, s = sin(_a), c = cos(_a);
        return A * s * c + B * s * s - C * c - E * s;
    }
    virtual double d(const double& a) const {
        double _a = a * PI / 180;
        return (A * cos(2 * _a) + B * sin(2 * _a) + C * sin(_a) - E * cos(_a)) * (PI
    }
private:
    double A, B, C, E;
public:
    Q(const double& l, const double& h, const double& D, const double& b1) {
        double _b = b1 * PI / 180;
        A = l * sin(_b);
        B = l * cos(_b);
        C = (h + 0.5 * D) * sin(_b) - 0.5 * D * tan(_b);
        E = (h + 0.5 * D) * cos(_b) - 0.5 * D;
    }
};
```

将题目中数据代入，调用牛顿法和割线法求解，并令 $x0, x1$ 逐渐远离 $\alpha_0 = 33$。

```
cout << "6\n";
Q q(89, 49, 55, 11.5);
cout << Newton<double>(q, 33).solve() << endl;
q = Q(89, 49, 30, 11.5);
cout << Newton<double>(q, 33).solve() << endl;

cout << Secant<double>(q, 30, 45).solve() << endl;
cout << Secant<double>(q, 60, 90).solve() << endl;
cout << Secant<double>(q, 90, 180).solve() << endl;
cout << Secant<double>(q, 180, 360).solve() << endl;
```

结果：

```
6
Newton : times = 3
32.9722
Newton : times = 4
33.1689
Secant : times = 6, delta = 9.22356e-09
33.1689
Secant : times = 10, delta = 3.79062e-08
-11.5
Secant : times = 9, delta = 7.73916e-07
168.5
Secant : times = 9, delta = 1.4678e-07
168.5
```

前两问的答案见上述结果。对于第三问，当 $x0, x1$ 远离 33 时，割线法会收敛到其他的解，特别地，在 $x0, x1$ 取 $60, 90$ 时，割线法没有收敛到最近的解 33.1689，而是收敛到更远的解 -11.5。这是因为割线法的第一步已经越过了 33.1689 这个解。由此可见，当割线法的初始点和零点距离过远时，其收敛情况很复杂，无法保证收敛到最近解。