f(x)在区间(-1,1)上的实根隔离算法实现

author: 史西文

1.程序说明

- 实现了 Interval 区间类，可以进行区间的加，减，乘运算
- 实现了基本的 Poly 多项式类，可以使用函数 f.eval (x) 计算 f (x) ，可以使用函数 f.eval (I) 计算 f (I)
- 实现了函数 root_isolation (f , I ,result) ,result 是一个区间 vector ， 此函数计算 f 在区间 I 中的实根隔离区间，并将这些区间添加到 result 中
- delta表示我们要求 f 的任意两个实根之间的距离都要大于delta (预设为1e-100)
- epsilon表示如果区间长度小于epsilon，就不再进行计算了(预设为1e-100)

2.实根隔离函数注解

实根隔离函数注解：（此函数在RootIsolation.hpp中，详见4.(2)）

```
//计算 f 在区间 I 中的实根隔离区间，并将这些区间添加到vector<Interval> result中
//delta表示我们要求f的任意两个实根之间的距离都要大于delta
//epsilon表示如果区间长度小于epsilon，就不再进行计算了
void root_isolation(const Poly<double> &f, Interval I, vector<Interval> &result,
double delta = 1e-100, double epsilon = 1e-100)
{
    //表示如果区间长度小于epsilon，就不再进行计算了
    if (I.getLength() < epsilon)
    {
        return;
    }
    //如果0不在f(I)中，则f在I中一定没有实根，故对于区间I不做任何处理
    if (if_0_In_(f, I) == false)
    {
        return;
    }
    //如果0在f(I)中，则f在I中可能有实根
    else
```

```
    {
        double a = I(0), b = I(1);
        if (f.eval(a) == 0)   //如果f(a)==0,则a是一个实根，在result中添加区间(a,a)，并令a+=delta
        {
            result.push_back(Interval(a, a));
            a += delta;
        }
        if (f.eval(b) == 0)   //如果f(b)==0,则b是一个实根，在result中添加区间(b,b)，并令b-=delta
        {
            result.push_back(Interval(b, b));
            b -= delta;
        }
        if ((f.eval(a) * f.eval(b) < 0) && (if_0_In_(f.diff(), Interval(a, b)) == false))
        {
   //如果f(a)f(b)<0且0不在区间f'(I)中，则f在区间(a,b)中有唯一的实根，在result中添加区间(a,b)
            result.push_back(Interval(a, b));
            return;
        }
        else
        {
   //否则的话，对区间(a,b)进行二分，并计算 f 在两个二分区间中的实根隔离区间，并将这些区间添加到result中
            root_isolation(f, Interval(a, (a + b) / 2), result);
            root_isolation(f, Interval((a + b) / 2, b), result);
            return;
        }
    }
}
```

3.计算结果

注：以下几个实根隔离的例子程序运行时间都在 10^{-4} ~ $10^{-5}s$ 左右。

(1) $f(x)=x-0.01$

```
f(x)=-0.01+1x^{1}
f(x)在区间(-1,1)中的实根隔离区间为：
[(-1 , 1)]
The run time is: 2e-06s
```

(2) $f(x)=(x-0.1)(x-0.2)(x-0.3)=x^3-0.6x^2+0.11x-0.006$

```
f(x)=-0.006+0.11x^{1}-0.6x^{2}+1x^{3}
f(x)在区间(-1,1)中的实根隔离区间为：
[(0.09375 , 0.109375),(0.1875 , 0.203125),(0.296875 , 0.3125)]
The run time is: 4.7e-05s
```

(3) $f(x)=(x-0.01)(x-0.02)(x-0.03)=x^3-0.06x^2+0.0011x-6*10^{-6}$

```
f(x)=-6e-06+0.0011x^{1}-0.06x^{2}+1x^{3}
f(x)在区间(-1,1)中的实根隔离区间为：
[(0.00976562 , 0.0117188),(0.0195312 , 0.0205078),(0.0292969 , 0.03125)]
The run time is: 6.1e-05s
```

(4) $f(x)=x(x-3)(x-0.9)(x+0.99)(x-0.000001)$

展开为 $f(x)=x^5-2.910001x^4-1.16099709x^3+2.673001161x^2-(2.673*10^{-6})*x$

```
f(x)=0-2.673e-06x^{1}+2.673x^{2}-1.161x^{3}-2.91x^{4}+1x^{5}
f(x)在区间(-1,1)中的实根隔离区间为：
[(0 , 0),(-1 , -0.75),(0 , 0),(9.53674e-07 , 1.90735e-06),(0.75 , 1)]
The run time is: 4.9e-05s
```

(5) $f=(x-0.1)(x-0.01)(x-0.001)(x-0.0001)$

f展开为：$x^4-0.1111x^3+0.0011211x^2-(1.111*10^{-6})x+1*10^{-10}$

```
f(x)=1e-10-1.111e-06x^{1}+0.0011211x^{2}-0.1111x^{3}+1x^{4}
f(x)在区间(-1,1)中的实根隔离区间为：
[(0 , 0.000488281),(0.000976562 , 0.00195312),(0.0078125 , 0.0117188),(0.09375 , 0.125)]
The run time is: 2.3e-05s
```

4.源代码

(1) main.cpp （主函数）

```
#include "include.hpp"
int main()
{
    //实验(1)
    //Poly<double> f(vector<double>{-0.01, 1});
    //实验(2)
    //Poly<double> f(vector<double>{-0.006, 0.11, -0.6, 1}); //表示f(x)= -0.006+
0.11x - 0.6x^2 + x^3
    //实验(3)x^3 - 0.06x^2 + 0.0011x - 6*10^{-6}
    Poly<double> f(vector<double>{-6e-6, 0.0011, -0.06, 1});
    //实验(4)
    //Poly<double> f(vector<double>{0,-2.673e-6,
2.673001161,-1.16099709,-2.910001, 1});
    //实验(5)
    //Poly<double> f(vector<double>{1e-10, -1.111e-6, 0.0011211, -0.1111, 1});
    vector<Interval> result;
    clock_t startTime, endTime;
    startTime = clock(); //计时开始
    root_isolation(f, Interval(-1, 1), result);
    endTime = clock(); //计时结束
    cout << "f(x)=" << f << endl;
    cout << "f(x)在区间(-1,1)中的实根隔离区间为: " << endl;
    cout << result << endl;
    cout << "The run time is: " << (double)(endTime - startTime) / CLOCKS_PER_SEC
<< "s" << endl;
    return 0;
}
```

(2) RootIsolation.hpp（实根隔离函数实现）

```
#pragma once
#include "Interval.hpp"
#include "Poly.hpp"
//判断0是否在f(I)中
bool if_0_In_(const Poly<double> &f, const Interval &I)
{
    return ifIn(0, f.eval(I));
}
//计算f在区间I中的实根隔离区间，并将这些区间添加到vector<Interval> result中
//delta表示两个实根之间的距离大于delta，epsilon表示如果区间长度小于epsilon，就不再进行计算了
void root_isolation(const Poly<double> &f, Interval I, vector<Interval> &result,
double delta = 1e-100, double epsilon = 1e-100)
{
    //表示如果区间长度小于epsilon，就不再进行计算了
    if (I.getLength() < epsilon)
    {
        return;
    }
    //如果0不在f(I)中，则f在I中一定没有实根，故对于区间I不做任何处理
    if (if_0_In_(f, I) == false)
    {
        return;
    }
    //如果0在f(I)中，则f在I中可能有实根
    else
    {
        double a = I(0), b = I(1);
        if (f.eval(a) == 0)
        {
            result.push_back(Interval(a, a));
            a += delta;
        }
        if (f.eval(b) == 0)
        {
            result.push_back(Interval(b, b));
            b -= delta;
        }
        if ((f.eval(a) * f.eval(b) < 0) && (if_0_In_(f.diff(), Interval(a, b)) ==
false))
        {
            result.push_back(Interval(a, b));
            return;
        }
        else
        {
            root_isolation(f, Interval(a, (a + b) / 2), result);
            root_isolation(f, Interval((a + b) / 2, b), result);
            return;
        }
    }
}
```

(3) Interval.hpp（区间类）

```
#pragma once
#include <iostream>
#include <string>
#include <vector>
using namespace std;
//double valued open interval
class Interval
{
private:
    double a, b;

public:
#pragma region 构造函数和输出函数
    //构造函数
    Interval(double m_a = 0, double m_b = 0) : a(m_a), b(m_b)
    {
        if (a > b)
        {
            throw invalid_argument("Interval() Wong:a must be less than b!");
        }
    }
    //输出函数
    friend ostream &operator<<(ostream &out, const Interval &I)
    {
        return out << "(" << I.a << " , " << I.b << ")";
    }
#pragma endregion

#pragma region 区间基本处理
    //判断一个数是否所在区间之中
    friend bool ifIn(double d, const Interval &I)
    {
        return I.a < d && d < I.b;
    }

    //返回区间的长度
    double getLength() const { return b - a; }
    //返回区间的左端点和右端点（可以修改）
    double &operator[](int i)
    {
        if (i == 0)
        {
            return a;
        }
        else
        {
            return b;
        }
    }
    //返回区间的左端点和右端点（不可以修改）
    double operator()(int i) const
    {
        if (i == 0)
        {
            return a;
```

```
        }
        else
        {
            return b;
        }
    }
#pragma endregion

#pragma region 区间算术

#pragma region +=
    //加等一个区间
    Interval &operator+=(const Interval &I)
    {
        double n1 = a + I.a;
        double n2 = a + I.b;
        double n3 = b + I.a;
        double n4 = b + I.b;
        a = min(n1, min(n2, min(n3, n4)));
        b = max(n1, max(n2, max(n3, n4)));
        return *this;
    }
    //加等一个常数
    Interval &operator+=(const double rhs)
    {
        double n1 = a + rhs;
        double n2 = b + rhs;
        a = min(n1, n2);
        b = max(n1, n2);
        return *this;
    }
#pragma endregion

#pragma region -=
    //减等一个区间
    Interval &operator-=(const Interval &I)
    {
        double n1 = a - I.a;
        double n2 = a - I.b;
        double n3 = b - I.a;
        double n4 = b - I.b;
        a = min(n1, min(n2, min(n3, n4)));
        b = max(n1, max(n2, max(n3, n4)));
        return *this;
    }
    //减等一个常数
    Interval &operator-=(const double rhs)
    {
        double n1 = a - rhs;
        double n2 = b - rhs;
        a = min(n1, n2);
        b = max(n1, n2);
        return *this;
    }

#pragma endregion

#pragma region *=
```

```
    //乘等一个区间
    Interval &operator*=(const Interval &I)
    {
        double n1 = a * I.a;
        double n2 = a * I.b;
        double n3 = b * I.a;
        double n4 = b * I.b;
        a = min(n1, min(n2, min(n3, n4)));
        b = max(n1, max(n2, max(n3, n4)));
        return *this;
    }
    //乘等一个常数
    Interval &operator*=(const double rhs)
    {
        double n1 = a * rhs;
        double n2 = b * rhs;
        a = min(n1, n2);
        b = max(n1, n2);
        return *this;
    }

#pragma endregion

#pragma endregion
};

#pragma region 区间取正和负
Interval operator+(const Interval &I)
{
    Interval re = I;
    return re;
}
Interval operator-(const Interval &I)
{
    Interval re;
    re[0] = -I(1);
    re[1] = -I(0);

    return re;
}
#pragma endregion

#pragma region 区间算术

#pragma region +
//两个区间相加
Interval operator+(const Interval &I, const Interval &J)
{
    Interval re = I;
    re += J;
    return re;
}
//一个区间和一个常数相加
Interval operator+(const Interval &I, double a)
{
    Interval re = I;
    re += a;
```

```
    return re;
}
//一个常数和一个区间相加
Interval operator+(double a, const Interval &J)
{
    Interval re = J;
    re += a;
    return re;
}
#pragma endregion

#pragma region -
//两个区间相减
Interval operator-(const Interval &I, const Interval &J)
{
    Interval re = I;
    re -= J;
    return re;
}
//一个区间减去一个常数
Interval operator-(const Interval &I, double a)
{
    Interval re = I;
    re -= a;
    return re;
}
//一个常数减去一个区间
Interval operator-(double a, const Interval &J)
{
    Interval re = -J;
    re += a;
    return re;
}
#pragma endregion

#pragma region *
//两个区间相乘
Interval operator*(const Interval &I, const Interval &J)
{
    Interval re = I;
    re *= J;
    return re;
}
//一个区间乘上一个常数
Interval operator*(const Interval &I, double a)
{
    Interval re = I;
    re *= a;
    return re;
}
//一个常数乘上一个区间
Interval operator*(double a, const Interval &J)
{
    Interval re = J;
    re *= a;
    return re;
}
#pragma endregion
```

```
#pragma endregion

#pragma region 区间相等判断
bool operator==(const Interval &I, const Interval &J)
{
    return I(0) == J(0) && I(1) == J(1);
}
bool operator!=(const Interval &I, const Interval &J)
{
    return ~(I == J);
}
#pragma endregion
```

(4) Poly.hpp（多项式类）

```
#pragma once

#include "includeSTL.hpp"
#include "Interval.hpp"
template <typename T> //T表示多项式的系数的数据类型
class Poly
{

private:
    vector<T> poly;
    int degree;

public:
    //构造函数（用一个vector初始化进行构造） test ok!
    Poly(const vector<T> &m_poly = vector<T>(1))
    {
        poly = m_poly;
        for (int i = poly.size() - 1; i >= 1; i--)
        {
            if (poly[i] == 0)
            {
                poly.pop_back();
            }
            else
            {
                break;
            }
        }
        degree = poly.size() - 1;
    }

#pragma region 返回i次项的系数（[] 可以修改和() 不可以修改），返回多项式的次数[test ok !]
    //返回x^i的系数（可以修改）
    T &operator[](int i)
    {
        return poly[i];
    }
    //返回x^i的系数（不可以修改）
    T operator()(int i) const
    {
        return poly[i];
```

```
    }
    //返回多项式的次数
    int getDegree() const
    {
        return degree;
    }
#pragma endregion

#pragma region 多项式加等test ok !, 减等test ok !, 乘等todo
    //多项式加等 test ok!
    Poly<T> &operator+=(const Poly &f)
    {
        if (degree >= f.getDegree()) //degree >= degree(f)
        {
            for (int i = 0; i <= f.getDegree(); i++)
                poly[i] += f(i);
        }
        else //degree < degree(f)
        {
            for (int i = 0; i <= degree; i++)
                poly[i] += f(i);
            for (int i = degree + 1; i <= f.getDegree(); i++)
                poly.push_back(f(i));
        }
        degree = poly.size() - 1;
        return *this;
    }
    //多项式减等 test ok!
    Poly<T> &operator-=(const Poly &f)
    {
        if (degree >= f.getDegree())
        {
            for (int i = 0; i <= f.getDegree(); i++)
                poly[i] -= f(i);
        }
        else //degree < degree(f)
        {
            for (int i = 0; i <= degree; i++)
                poly[i] -= f(i);
            for (int i = degree + 1; i <= f.getDegree(); i++)
                poly.push_back(-f(i));
        }
        degree = poly.size() - 1;
        return *this;
    }

#pragma endregion

#pragma region 计算f(x) test ok !, 计算f(I)
    //计算f(x) test ok!
    T eval(T x) const
    {
        T re = poly[degree];
        for (int i = degree; i >= 1; i--)
        {
            re *= x;
            re += poly[i - 1];
        }
```

```
        return re;
    }
    Interval eval(const Interval &I) const
    {
        Interval re(poly[degree], poly[degree]);
        for (int i = degree; i >= 1; i--)
        {
            re *= I;
            re += poly[i - 1];
        }
        return re;
    }

#pragma endregion

#pragma region 计算f的导数 test ok !
    // 计算f的导数 test ok!
    Poly<T> diff() const
    {
        if (poly.size() <= 1)
            return Poly();
        vector<T> a;
        for (int i = 0; i < degree; i++)
        {
            a.push_back((i + 1) * poly[i + 1]);
        }
        return Poly<T>(a);
    }

#pragma endregion
};
//输出多项式 test ok!
template <typename T>
ostream &operator<<(ostream &out, const Poly<T> f)
{
    out << f(0);
    for (int i = 1; i <= f.getDegree(); i++)
    {
        if (f(i) == 0)
        {
            continue;
        }
        else if (f(i) > 0)
        {
            out << "+" << f(i) << "x^{" << i << "}";
        }
        else
        {
            out << "-" << (-f(i)) << "x^{" << i << "}";
        }
    }
    return out;
}
```

(5) includeSTL.hpp

```cpp
#pragma once
#include <iostream>
#include <vector>
#include <string>
#include <ctime>
using namespace std;
template <typename T>
ostream &operator<<(ostream &out, const vector<T> &a)
{
    out << "[" << a[0];
    for (int i = 1; i < a.size(); i++)
    {
        out << "," << a[i];
    }
    out << "]";
    return out;
}
double myabs(double a)
{
    if (a > 0)
    {
        return a;
    }
    else
    {
        return -a;
    }
}
double min(double a, double b)
{
    return a > b ? b : a;
}
double max(double a, double b)
{
    return a > b ? a : b;
}
```

-- END --