Chap2 Solutions of Equations in One Variable

Xu Feng, imxufeng@outlook.com, 201900090026

March 20, 2022

1 Introduction

In this paper, we implement several rootfinding methods demostrated in the chapter 2 of Numerical Analysis [1] with C++, followed with anlysis and comparison of different methods, including the impact of different initial values, different tolerances and sensitivity analysis.

2 Implementation Details and Codes

2.1 Preparations

We first define the class Result that denotes the return type as following rootfinding algorithms. Class "Result" records the final state (whether successful or not), the root and iteration records if successful and exception if failed.

```
class Result {
    public:
        int getIterNum() const{
            return this->iterNum;
        }
        Result(double p, vector<double> pVector, int iterNum, std::string methodName, int state
            =1){
            this->state = state;
            this->equationSolution = p;
            this->pVector = std::move(pVector);
            this->iterNum = iterNum;
            this->methodName = std::move(methodName);
            printResult();
        }
        Result(int state=0){
            this->state = 0;
            this->equationSolution = 0.;
            printResult();
        }
        void printResult(){
            switch (this->state) {
```

```
                case 1:
//                  cout << endl << endl;
//                  cout << "Method: " << this->methodName << endl;
//                  cout << "Success! Answer: " << this->equationSolution << "." << endl;
//                  cout << "The number of iterations is " << this->iterNum << "." << endl;
//                  for (double i : pVector) { cout << i << " "; }
                    break;
                case 0:
                    cout << endl << endl;
                    cout << "Fail!" << endl;
                default :
                    cout << "Error!" << endl;
            }
        }

    private:
        double equationSolution;
        int state = -1, iterNum = -1; // -1 denotes unreachable
        std::string methodName = "NULL";
        vector<double> pVector;
    };
```

We also defines some exceptions that may occur in the process.

```
class IterException : public exception{
    public:
        [[nodiscard]] const char * what() const noexcept override {
            return "Fail! Exceed the number of iterations.";
        }
    };

    class ComplexException : public exception{
    public:
        [[nodiscard]] const char * what() const noexcept override {
            return "Fail! May require complex arithmetic.";
        }
    };

    class ConditionException : public exception{
    public:
        [[nodiscard]] const char * what() const noexcept override {
            return "Fail! not meet the requirements.";
        }
    };
```

2.2 Bisection

```
static Result bisection(double a=1., double b=2.,
                        const function<double(double)>& f = Function::f){
    try{
        int i=0;
        vector<double> pVector;
        double fa = f(a), fp, p;

        // check condition
        double fb = f(b);
        if (fb * fa > 0 ) throw ConditionException();

        while(i<RootFinding().N){
            p = (a+b)/2;
            fp = f(p);
            pVector.push_back(p);
            if ((abs(fp)<RootFinding().eps) | (b-a < RootFinding().TOL))
                return {p, pVector, i, "bisection"};
            i++;
            if (fa * fp > RootFinding().eps) a = p;
            else b = p;
        }
        throw IterException();
    }
    catch (IterException& iterException){
        cout << iterException.what();
        return {0};
    }
    catch (ConditionException& conditionException){
        cout << conditionException.what();
        return {0};
    }
}
```

We choose the function $f(x) = 2ln(x) - x + 1$ which has two roots: 1 and a root between 3 and 4. Results are shown in the following gray box.

```
Method: bisection
Success! Answer: 3.51286.
The number of iterations is 18.
4 3.5 3.75 3.625 3.5625 3.53125 3.51562 3.50781 3.51172 3.51367 3.5127 3.51318 3.51294
    3.51282 3.51288 3.51285 3.51286 3.51286 3.51286
```

2.3 FixedPoint

```
static Result fixedPoint(double p0=3., function<double(double)> f = Function::f){
```

```
    try{
        vector<double> pVector;
        int i=0;
        double p;
        while(i<RootFinding().N){
            p = f(p0);
            pVector.push_back(p);
            if (abs(p-p0)<RootFinding().TOL){
                return {p, pVector, i, "fixedPoint", 1};
            }
            i++; p0 = p;
        }
        throw IterException();
    }
    catch(IterException& iterException){
        cout << iterException.what() << endl;
    }
    return {0};
}
```

For fixedPoint algorithm, we choose $x = 2ln(x) + 1$, which equals to calculate the root of the function $f(x) = 2ln(x) - x + 1$.

```
Method: fixedPoint
Success! Answer: 3.51285.
The number of iterations is 19.
3.19722 3.32457 3.40268 3.44913 3.47624 3.4919 3.50089 3.50604 3.50897 3.51065 3.5116
    3.51214 3.51245 3.51263 3.51273 3.
51279 3.51282 3.51284 3.51285 3.51285
```

2.4 Newton

```
static Result newton(double p0=1.5, const function<double(double)>& f = Function::f,
                     const function<double(double)>& f1 = Function::f1) {
    try{
        int i=0;
        double p;
        vector<double> pVector;
        while(i<RootFinding().N){
            p = p0 - f(p0)/f1(p0);
            pVector.push_back(p0);
            if (abs(p-p0)<RootFinding().TOL) return {p0, pVector, i, "newton"};
            i++, p0=p;
        }
        throw IterException();
    }
    catch (IterException& iterException){
        cout << iterException.what();
```

```
    }
    return {0};
}
```

```
    Method: newton
    Success! Answer: 3.51287.
    The number of iterations is 4.
    15 5.0955 3.71478 3.5195 3.51287
```

2.5 Secant

```
static Result secant(double p0=1, double p1=2, function<double(double)> f = Function::f){
        try{
            int i = 0;
            double p;
            double q0 = f(p0), q1 = f(p1);
            vector<double> pVector;
            while(i<RootFinding().N){
                p = p1 - q1*(p1-p0)/(q1-q0);
                pVector.push_back(p);
                if (abs(p-p1)<RootFinding().TOL) return {p, pVector, i, "secant"};
                i++, p0=p1, q0=q1, p1=p, q1=f(p);
            }
            throw IterException();
        }
        catch (IterException& iterException){
            cout << iterException.what() << endl;
        }
        return {0};
    }
```

```
    Method: secant
    Success! Answer: 3.51286.
    The number of iterations is 4.
    3.40318 3.49242 3.51331 3.51286 3.51286
```

2.6 FalsePosition

```
static Result falsePosition(double p0=1., double p1=2.,
                            function<double(double)> f = Function::f){
        try{
            int i = 0;
            double p;
            vector<double> pVector;
            double q0 = f(p0), q1 = f(p1), q;
            while(i<RootFinding().N){
                p = p1 - q1*(p1-p0)/(q1-q0);
```

```
                pVector.push_back(p);
                if (abs(p-p1)<RootFinding().TOL) return {p, pVector, i, "falsePosition"};
                i++, q=f(p);
                if (q*q1<0) {
                    p0 = p1;
                    q0 = q1;
                }
                p1 = p;
                q1 = q;
            }
            throw IterException();
        }
        catch (IterException& iterException){
            cout << iterException.what() << endl;
        }
        return {0};
    }
```

```
    Method: falsePosition
    Success! Answer: 3.51286.
    The number of iterations is 7.
    3.40318 3.49242 3.50916 3.51219 3.51274 3.51284 3.51286 3.51286
```

2.7 Steffensen

```
static Result steffensen(double p0=2.5, function<double(double)> f = Function::f){
        try{
            int i = 0;
            double p, p1, p2;
            vector<double> pVector;
            while(i<RootFinding().N){
                p1 = f(p0);
                p2 = f(p1);
                pVector.push_back(p);
                p = p0 - (p1-p0)*(p1-p0)/(p2-2*p1+p0);
                if (abs(p-p0)<RootFinding().TOL) return {p, pVector, i, "steffensen"};
                i++, p0=p;
            }
            throw IterException();
        }
        catch (IterException& iterException){
            cout << iterException.what() << endl;
        }
        return {0};
        }
```

```
    Method: steffensen
```

```
Success! Answer: 2.
The number of iterations is 7.
6.9513e-310 3.41176 2.89497 2.47376 2.17952 2.03428 2.00149 2
```

2.8 Horner

```
static Result horner(int n = 3, vector<double> coe = {2, 1, 3, 7},
                     double x0 = 2) {
    try{
        double y = coe[n], z=coe[n];
        for(int j = n-1; j>0; j--){
            y = x0 * y + coe[j];
            z = x0 * z + y;
        }
        y = x0 * y + coe[0];
        cout << y << " " << z << endl;
        return {1};
    }
    catch (IterException& iterException){
        cout << iterException.what() << endl;
    }
    return {0};
    }
```

Horner algorithm is intended to calculate the specified value of the given polynomial and its derivative with a given input value. So we take the polynomial $g(x) = 7x^3 + 3x^2 + x + 2$ whose derivative is $g'(x) = 21x^2 + 6x + 1$. Given input value 2, we have $g(2) = 72$ and $g'(2) = 97$. The c++ program gives us the exact value as expected.

2.9 Muller

```
static Result horner(int n = 3, vector<double> coe = {2, 1, 3, 7},
                     double x0 = 2) {
    try{
        double y = coe[n], z=coe[n];
        for(int j = n-1; j>0; j--){
            y = x0 * y + coe[j];
            z = x0 * z + y;
        }
        y = x0 * y + coe[0];
        cout << y << " " << z << endl;
        return {1};
    }
    catch (IterException& iterException){
        cout << iterException.what() << endl;
    }
    return {0};
    }
```

```
Method: muller
Success! Answer: 3.51286.
The number of iterations is 3.
3.50245 3.51268 3.51286 3.51286
```

3 Comparasion and Analysis

3.1 Comparasion of iteration number and running time

Given proper function and initial values, running results are shown above. Among root-finding algorithms, including bisection, newton, secant, falsePosition and muller, running time is highly related to the iteration number, as well as some other factors, such as the number of operations, especitially the operation of multiply.

To compare the running time of each algorithm, we repeat 10000 times for each algoritm as time for running once is too short to record and compare. We implement the "recordTime" function and set the input as each algorithm and their initial value. Codes are shown in the following gray box.

```
template<typename F>
double recordTime(F f, int repeat = 1){
    std::clock_t t;
    t = std::clock();
    for (int repeat_ = 0; repeat_ < repeat; repeat_++){
        auto&& res = f();
    }
    t = std::clock() - t;
    cout<<"Time: "<<(double)(t)/CLOCKS_PER_SEC<<"s"<<endl;
    return (double)(t)/CLOCKS_PER_SEC;
}
```

Table 1: Running time for each algorithm with **10000 repeats**

Algorithm	bisection	fixedpoint	newton	secant	falseposition	horner	muller
Running time	0.04	0.039	0.02	0.019	0.021	0.002	0.017

The time for calculating the same equation differs, as shown in the table above, muller takes the lead while newton and secant is nearly the same, but bisection and fixedpoint takes almost double time that newton does.

3.2 Impact of initial values

Each algorithm needs one or two initial value(s) as input parameter. Due to the iterative nature of the algorithms above, different initial values may have impact on the iteration number as well as running time. For instance, an expected initial value can even be the exact root of the equation, while an unexpected initial value may cause long running time.

For Bisection algorithm, an interval with 2 initial values are given and then we search for the root in the given interval by dividing the interval into half each iteration.

The power of 2 is huge ($2^{10} = 1,024, 2^{20} = 1,048,576$). As a result, when we divide the interval into half to search for the root, the iteration number depends on how many times the length of the interval would be reduced to the given TOL. Although Bisection algorithm is not effective enough, it is not sensitive to the initial values. In other words, even we give initial values that are far away from the actual root, Bisection can quickly reduce the length of the interval.

Newton and secant method perform much better when the initial value changes. Only several iteraitons are required when the initial value changes to a larger scale.

3.3 Impact of tolerance TOL

We initially set the TOL as 1e-5, and get Table 1. Following we will test different tolerance value TOL, including 1e-10, 1e-15.

Table 2: Impact of different tolerance value TOL

TOL	bisection	fixedpoint	newton	secant	falseposition	horner	muller
1e-5	0.04	0.039	0.02	0.019	0.021	0.002	0.017
1e-10	0.052	0.054	0.022	0.02	0.029	0.003	0.021
1e-15	0.056	0.091	0.026	0.022	0.037	0.003	0.023

Table 2 shows that the algorithm fixedpoint is highly sensitive to the TOL (As TOL becomes smaller, the running time ascends rapidly), the algorithm bisection, newton and falsePosition is related to the TOL, while secant and muller don't change much as the TOL reduces from 1e-5 to 1e-10.

3.4 Guildline for choice

According to the analysis above, each algorithm is suitable for a specified case, so a guildline is important when faced with different conditions.

Muller algorithm performs best in all of analysis, but it requires three values as input. But when not much attention is paid to the choice of initial values, newton and secant is more favorable.

Though a faster and robust algorithm is favored, bisection algorithm has its strength when we actually knows the approximate interval of the expected root. However, it's not suitable for the condition that the more than one root is included in the given interval.

4 Conclusion

Based on the implementations of several rootfinding algorithms, this paper compares different algorithms in the perspective of running time, iteration number and the impact of initial values, tolerance TOL and sensitivity analysis and finally comes to a basic guildline of choosing different algorithms when faced with different conditions.

References

[1] Burden, Richard L., J. Douglas Faires, and Annette M. Burden. Numerical analysis. Cengage learning, 2015.