Chap3 Interpolation and Polynomial Approximation

Xu Feng, imxufeng@outlook.com, 201900090026

April 3, 2022

1 Introduction

As Weierstrass approximation theorem, any function f can be approximated by polynomials. In this work, I review several polynomial interpolation methods illustrated in the book [1] with C++ implementations, including Lagrange, Neville, Newton and Hermite's method, followed with analysis and comparison of different methods. Detailed interpolation and experiment codes is shown in the appendix.

2 Implementations of polynomial class

Operations of add and multiply of polynomials are needed in the implementation of interpolation methods, but there is no existing polynomial class in Standard Template Library of C++. So the first thing to do is to form a polynomial class which includes overloaded operators of add and multiply.

Implementations of polynomial class is in the file "poly.h". The constructor of the poly class can be made of a vector of double denoting the coefficients of the polynomial, which is the basis of following overloaded operators.

With $f(x) = \sum_{i=0}^{n} a_i x^i$ and $g(x) = \sum_{i=0}^{m} b_i x^i$, we have:

$$f(x) + g(x) = \sum_{i=0}^{max(n,m)} (a_i + b_i) x^i,$$

and:

$$f(x)g(x) = \sum_{k=0}^{n+m} (\sum_{i=0}^{k} a_i b_{k-i}) x^k.$$

It is possible that after the operations like add or multiply, several coefficient of the vector will be zero, especially the highest term of the polynomial in which case the degree of the polynomial should be updated. For example, take $f(x) = 1 + 2x + 3x^2$ with degree of two and $g(x) = 1+x-3x^2$ with degree of two, but $f(x)+g(x) = 2+3x$ with degree 3 which is less than two. If not checked, bugs will be raised in the following implementation process.

With above analysis, I implement the poly class with C++, detailed codes can be found in Appendix B.

3 Different methods and comparison

The polynomial methods can be categoried into two classes, one is the n-th degree polynomial result given n+1 data point and corresponding value of the function, and the other is (2n+1)-th degree polynomial result with extra knowledge of the derivative values of the given data point. Implementation codes can be found in Appendix C.

3.1 Polynomial interpolation of degree n

3.1.1 Similarity and difference

Lagrange, Neville and Newton's divided difference methods give the same result as these three methods all assume that the expected form the polynomial is the one with degree n, which gives the unique solution given the same data.

Lagrange method gives a explicit formula of the polynomial given data points, while Neville and Newton are both iterated algorithm which give the same result as Lagrange but calculate in a different way.

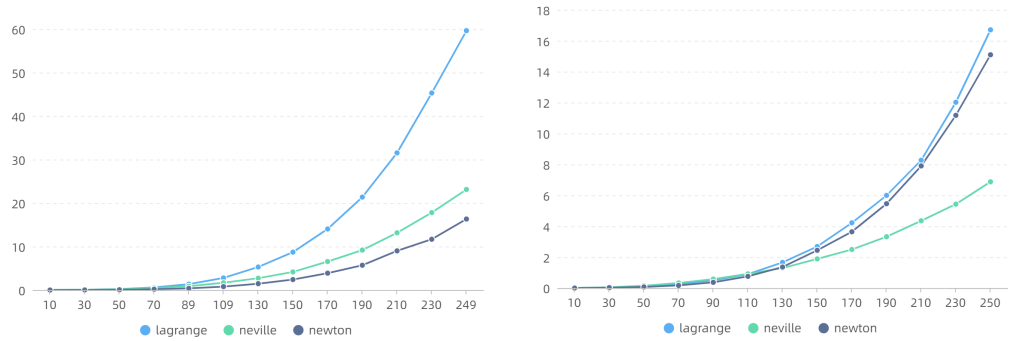
3.2 Prediction with data in the book

We use the data of 3c on page 110 the book [1], which gives that the point is $0.1, 0.2, 0.3, 0.4$, and the function value is 0.62049958, -0.28398668, 0.00660095, 0.24842440. The task is to predict f(0.25).

The result is the interpolation polynomial $p(x) = (3.9639) + (-49.7747)x^1 + (184.138)x^2 + (-207.306)x^3$ and $p(0.25) = -0.210337$ gives an approximation of $f(0.25)$.

3.2.1 Experiments of efficiency

The efficiency of the three algorithms is highly related to the property of given data. The following figure is a comparison of running time among these three algorithms. The experiment data is based on serval points with the corresponding function values randomly generated, and we takes the average running time of 100 repeats. We take the data points between 0 and 1, and also check points range from 0 to maximum of interger. It is found that different experiments get different results. The x label is the number of points and the y label is running time for 10 repeats.



(a) Running time of data points ranging from 0 to 1 (repeat 10 times)

(b) Running time of data points with larger range (repeat 10 times)

In the former experiment, the points are all between 0 and 1,the nature of explicit formula brings Lagrange polynomial interpolation too much things to calculate, and as a result, it may take much more time than the other two. As the data points are all between 0 and 1, Neville and Newton's method shows great superiority in the performance of calculating.

But when the range of points become larger, things become different. It is shown in the right figure that when the number of points become larger, Newton's method gets worse, while Neville's method perfomrs better.

3.3 Polynomial interpolation of degree 2n+1

With the knowledge of derivative values of given data point, we can get the result of polynomial of degree 2n+1 by the algorithm of Hermite. Hermite's algorithm can also implemented in a divided difference way.

We use the data of 1d on page 139 the book [1], which gives that the point is $0.1, 0.2, 0.3, 0.4$, and the function value is -0.62049958, 0.28398668, 0.00660095,

0.24842440. Also the derivative is known as 3.58502082, 3.14033271, 2.66668043, 2.16529366, respectively.

The result is the interpolation polynomial of degree n is $p_0(x) = (-4.40794) + (57.9594)x^1 + (-229.202)x^2 + (283.514)x^3$, and the (2n+1)-th degree polynomial is $p_1(x) = (-21.447) + (855.96)x^1 + (-14329.1)x^2 + (125096)x^3 + (-610571)x^4 + (1.67552e+06)x^5 + (-2.41389e+06)x^6 + (1.41993e+06)x^7$. The approximation of $f(0.25)$ is $p_0(0.25) = 0.18671$ and $p_1(0.25) = 0.136794$.

3.4 Comparison between degree n method and degree 2n+1 method

What can be explicitly seen in Taylor's theorem is that higher degree polynomial may have higher posssibility to approximate the function.

Take $f(x) = e^x$ as an example. We give $x = 1, 2, 3, 4$ and $f(x) = exp(1), exp(2), exp(3), exp(4)$ respectively for these algorithms and give $f'(x) = exp(1), exp(2), exp(3), exp(4)$ as an extra knowledge for method Hermite.

We get the result that: the first three algorithms give $p_0(x) = (-7.71721) + (17.9147)x^1 + (-9.77757)x^2 + (2.2984)x^3$, while Hermite gives that $p_1(x) = (0.856415) + (1.58543)x^1 + (-0.489549)x^2 + (1.06998)x^3 + (-0.443636)x^4 + (0.163895)x^5 + (-0.0268424)x^6 + (0.00259051)x^7$.

We test on value 6, and get that $p_0(6) = 244.233$ and $p_1(6) = 396.173$, while the ground truth is that $exp(6) = 403.429$. We also test on value 2.5, which is between the minimum number 1 and the maximum number of 4, and find that $p_0(x) = 11.8722$ and $p_1(x) = 12.1824$, while the ground truth is $exp(6) = 12.1825$.

It is obvious from the above example that in most times Hermite interpolation method gives higher-degree polynomial, which reduces the error and gets better interpolation performance.

4 Conclusion

Different polynomial interpolation methods have their own superiority faced with different situations. Usually

References

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

A Codes: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.21)
project (cm3)

set(CMAKE_CXX_STANDARD 14)

add_executable(cm3 poly.h main.cpp test3.h interpolation.h)
```

B Codes: Poly.h

```
#ifndef CM3_POLY_H
#define CM3_POLY_H

#include <vector>
#include <cmath>
using std::vector, std::cin, std::cout, std::endl, std::pow;

class DivByZeroException : public std::exception{
public:
    [[nodiscard]] const char* what() const noexcept override {
        return "DivByZero";
    }
};

class Poly {
public:
    Poly(){
        this->deg = 0;
        this->coefs = vector<double>{1.};
    }
    Poly(double m){
        this->deg = 0;
        this->coefs = vector<double>{m};
    }
    Poly(vector<double> coefs){
        this->deg = coefs.size()-1;
```

```
27          this->coefs = coefs;
28      }
29      int getDeg() const{
30          return this->coefs.size()-1;
31      }
32      double getCoef(int i) const{
33          return this->coefs[i];
34      }
35      vector<double> getCoefs() const{
36          return this->coefs;
37      }
38      void print() const{
39          cout << "(" << this->coefs[0] << ")";
40          for(int i=1; i<this->coefs.size(); i++){
41              cout << "+(" << this->coefs[i] << ")x^" << i;
42          }
43          cout << endl;
44      }
45      double getValue(double x){
46          double res = 0.;
47          for(int i=0; i<=getDeg(); i++){
48              res += getCoef(i) * std::pow(x, i);
49          }
50          return res;
51      }
52  private:
53      int deg = -1;
54      vector<double> coefs;
55  };
56
57
58  Poly operator+(Poly const& p0, Poly const& p1){
59      vector<double> res;
60      double tmp;
61      int deg;
62      for(int i=0; i<std::max(p0.getDeg(), p1.getDeg())+1; i++) {
63          tmp = 0.;
64          if (i <= p0.getDeg()) tmp += p0.getCoef(i);
```

```
65            if (i <= p1.getDeg()) tmp += p1.getCoef(i);
66            res.push_back(tmp);
67            if (std::abs(tmp)>=1e-10) {deg = i;}
68        }
69        for(int i=deg+1; i<std::max(p0.getDeg(), p1.getDeg())+1; i++) {
70            res.pop_back();
71        }
72        return {res};
73    }
74
75    Poly operator*(Poly const& p0, Poly const& p1){
76        vector<double> res;
77        double tmp = 0.;
78        int deg;
79        for(int i=0; i<=p0.getDeg()+p1.getDeg(); i++){
80            tmp = 0.;
81            for(int j=0; j<=i; j++){
82                if (j<=p0.getDeg() && i-j<=p1.getDeg()) {
83                    tmp += p0.getCoef(j) * p1.getCoef(i - j);
84                }
85            }
86            if (std::abs(tmp)>=1e-10) {deg = i;}
87            res.push_back(tmp);
88        }
89        for(int i=deg+1; i<std::max(p0.getDeg(), p1.getDeg())+1; i++) {
90            res.pop_back();
91        }
92        return {res};
93    }
94
95    Poly operator*(Poly const& p0, double m) {
96        vector<double> res;
97        for(int i=0; i<=p0.getDeg(); i++){
98            res.push_back(p0.getCoef(i)*m);
99        }
100       return {res};
101   }
102
```

```
103
104 Poly operator/(Poly const& p0, double m){
105     if (std::abs(m)<1e-20) throw DivByZeroException();
106     vector<double> res;
107     for(int i=0; i<=p0.getDeg(); i++){
108         res.push_back(p0.getCoef(i)/m);
109     }
110     return {res};
111 }
112
113
114 Poly operator-(Poly& p0, double m){
115     vector<double> res = p0.getCoefs();
116     res[0] -= m;
117     return {res};
118 }
119
120
121 Poly operator+(Poly& p0, double m){
122     vector<double> res = p0.getCoefs();
123     res[0] += m;
124     return {res};
125 }
126
127 #endif //CM3_POLY_H
```

C Codes: interpolation.h

```
1
2 #ifndef CM3_INTERPOLATION_H
3 #define CM3_INTERPOLATION_H
4
5 #include "poly.h"
6 #include <vector>
7 #include <iostream>
8 using std::vector, std::cout, std::cin, std::endl;
9
10
```

```
class LengthMatch : std :: exception {
public :
    [[ nodiscard ]] const char * what () const noexcept override {
        return "Length does not match!";
    }
};

class Interpolation {
public :
    static Poly lagrange( vector<double> xList, vector<double> yList){
        try{
            if (xList . size () != yList . size ()) throw LengthMatch();
            int n = xList . size ()-1;
            Poly p = vector<double> {0.};
            for (int k=0; k<=n; k++){
                Poly lnk = vector<double> {1.};
                for (int i=0; i<=n; i++){
                    if (i != k){
                        Poly x = vector<double> {-xList[i], 1.};
                        Poly tmp = x / (xList [k] - xList [i]);
                        lnk = lnk * tmp;
                    }
                }
                Poly tmp = lnk * yList [k];
                p = p + tmp;
            }
            return p;
        }
        catch (LengthMatch& lengthMatch){
            cout << lengthMatch.what() << endl;
        }
    }

    static Poly neville (vector<double> xList, vector<double> yList){
        try{
            const int n = xList . size ()-1;
            Poly Q[n+1][n+1];
            for( int i=0; i<=n; i++){
```

```
49                  Q[i][0] = Poly(vector<double>{yList[i]});
50              }
51
52              for(int i=1; i<=n; i++){
53                  for(int j=1; j<=i; j++){
54                      Q[i][j] = (Poly(vector<double>{-xList[i-j], 1.}) * Q[i][j
                            -1]
55                              + Poly(vector<double>{-xList[i], 1.}) * Q[i-1][j-1]
                                    * (-1.))
56                                  /(xList[i]-xList[i-j]) ;
57                  }
58              }
59              return Q[n][n];
60          }
61          catch (LengthMatch& lengthMatch){
62              cout << lengthMatch.what() << endl;
63          }
64      }
65
66      static Poly newton(vector<double> xList, vector<double> yList){
67          try{
68              const int n = xList.size()-1;
69              double F[n+1][n+1];
70
71              for(int i=0; i<=n; i++){
72                  F[i][0] = yList[i];
73              }
74
75              for(int i=1; i<=n; i++){
76                  for(int j=1; j<=i; j++){
77                      F[i][j] = (F[i][j-1]-F[i-1][j-1])/(xList[i]-xList[i-j]);
78                  }
79              }
80
81              Poly p = Poly(F[0][0]) ;
82
83              Poly tmp;
84              for(int i=1; i<=n; i++){
```

```
85                  tmp = Poly(1.);
86                  for(int j=0; j<=i-1; j++){
87                      tmp = tmp * Poly(vector<double>{-xList[j], 1.});
88                  }
89                  tmp = tmp * F[i][i];
90                  p = p + tmp;
91              }
92              return p;
93          }
94          catch (LengthMatch& lengthMatch){
95              cout << lengthMatch.what() << endl;
96          }
97      }
98
99      static Poly hermite(vector<double> xList, vector<double> yList, vector<
            double> deList){
100         try {
101             const int n = xList.size()-1;
102             double z[2*n+2];
103             double Q[2*n+2][2*n+2];
104
105             for (int i = 0; i <= n; i++) {
106                 z[2*i] = xList[i];
107                 z[2*i+1] = xList[i];
108                 Q[2*i][0] = yList[i];
109                 Q[2*i+1][0] = yList[i];
110                 Q[2*i+1][1] = deList[i];
111
112                 if(i != 0){
113                     Q[2*i][1] = (Q[2*i][0]-Q[2*i-1][0])/(z[2*i]-z[2*i-1]);
114                 }
115             }
116
117             for(int i=2; i<=2*n+1; i++){
118                 for(int j=2; j<=i; j++){
119                     Q[i][j] = (Q[i][j-1]-Q[i-1][j-1])/(z[i]-z[i-j]);
120                 }
121             }
```

```
122
123             Poly h = Poly(Q[0][0]);
124             Poly tmp = Poly(1.);
125             for(int i=1; i<=n; i++){
126                 tmp = tmp * Poly(vector<double>{-xList[i-1], 1.});
127                 h = h + tmp * Q[2*i-1][2*i-1];
128                 tmp = tmp * Poly(vector<double>{-xList[i-1], 1.});
129                 h = h + tmp * Q[2*i][2*i];
130             }
131             tmp = tmp * Poly(vector<double>{-xList[n], 1.});
132             h = h + tmp * Q[2*n+1][2*n+1];
133             return h;
134
135         }
136         catch (LengthMatch& lengthMatch){
137             cout << lengthMatch.what() << endl;
138         }
139     }
140
141 };
142
143 #endif //CM3_INTERPOLATION_H
```

D Codes: experiments and plots

```
1  #include <iostream>
2  #include <vector>
3  #include <ctime>
4  #include "test3.h"
5  #include "interpolation.h"
6  #include <functional>
7  #include <algorithm>
8  using std::vector, std::cout, std::cin, std::exp, std::sort;
9
10 template<typename F>
11 double recordTime(F f, int repeat = 1){
12     std::clock_t t;
13     t = std::clock();
```

```
14      for (int repeat_ = 0; repeat_ < repeat; repeat_++){
15          auto&& res = f();
16      }
17      t = std::clock() - t;
18      cout<<(double)(t)/CLOCKS_PER_SEC<<",";
19      return (double)(t)/CLOCKS_PER_SEC;
20  }
21
22  static double f(double x) { return exp(x); }
23  static double f1(double x) { return exp(x); }
24
25  int main() {
26
27  //    // generate random data to test efficiency
28  //    for(int len=10; len<=500; len+=20){
29  //        cout << len << ",";
30  //        vector<double> xList4;
31  //        vector<double> yList4;
32  //        for (int i=0; i<len; i++){
33  //            xList4.push_back(((double) rand() / (RAND_MAX)));
34  //            sort(xList4.begin(), xList4.end());
35  //            xList4.erase(unique(xList4.begin(), xList4.end()), xList4.end());
36  //        }
37  //
38  //        cout << xList4.size() << ",";
39  //
40  //        for (int i=0; i<xList4.size(); i++){
41  //            yList4.push_back(((double) rand() / (RAND_MAX)));
42  //        }
43  //        int repeat = 10;
44  //        recordTime([xList4, yList4]{return Interpolation::lagrange(xList4,
        yList4);}, repeat);
45  //        recordTime([xList4, yList4]{return Interpolation::neville(xList4,
        yList4);}, repeat);
46  //        recordTime([xList4, yList4]{return Interpolation::newton(xList4,
        yList4);}, repeat);
47  //
48  //        cout << endl;
```

```
49 //     }
50
51
52 //     vector<double> xList = {1., 2., 3., 4.};
53 //     vector<double> yList = {1., 4., 9., 16.};
54 //     vector<double> deList = {2., 4., 6., 8.};
55
56 //     vector<double> xList = {1., 2., 3., 4.,};
57 //     vector<double> yList = {f(1.), f(2.), f(3.), f(4.)};
58 //     vector<double> deList = {f1(1.), f1(2.), f1(3.), f1(4.)};
59
60     // p119 3c
61 //     vector<double> xList = {0.1, 0.2, 0.3, 0.4};
62 //     vector<double> yList = {0.62049958, -0.28398668, 0.00660095,
       0.24842440};
63
64     // p139 1d
65     vector<double> xList = {0.1, 0.2, 0.3, 0.4};
66     vector<double> yList = {-0.62049958, 0.28398668, 0.00660095, 0.24842440};
67     vector<double> deList = {3.58502082, 3.14033271, 2.66668043, 2.16529366};
68
69     Poly p0 = Interpolation::lagrange(xList, yList);
70     Poly p1 = Interpolation::neville(xList, yList);
71     Poly p2 = Interpolation::newton(xList, yList);
72     Poly p3 = Interpolation::hermite(xList, yList, deList);
73
74     p0.print();
75     p1.print();
76     p2.print();
77     p3.print();
78
79 //     cout << p0.getValue(6) << endl;
80 //     cout << p0.getValue(2.5) << endl;
81 //     cout << p3.getValue(6) << endl;
82 //     cout << p3.getValue(2.5) << endl;
83 //     cout << f(6) << endl;
84 //     cout << f(2.5) << endl;
85
```

```
86      cout << p0.getValue(0.25) << endl;
87      cout << p3.getValue(0.25) << endl;
88
89
90
91
92  }
```