

# 实验报告

## 一、实验目的

已指一组实验数据：

$i$	1	2	3	4	5	6	7	8	9
$x_i$	1	3	4	5	6	7	8	9	10
$y_i$	10	5	4	2	1	1	2	3	4

试用最小二乘法求它的二次多项式、三次多项式拟合曲线，并分别求出最低点的位置。要求：需要在 MATLAB 中绘制原始的 9 个点，二次多项式曲线，以及三次多项式曲线。

## 二、实验方法（要求用自己的语言描述算法）

### 1) 最小二乘法的概念

在实际的工程应用中，由于各个统计点产生的数值偏差，使用插值多项式的方式来构造函数曲线会保留各个数值点的测量误差，最小二乘法曲线拟合在于构造参数组使解是最接近曲线的。

我们定义  $n$  维函数组  $\{f_0, f_1, \dots, f_n\}$ ，一系列的数据点为  $(x_i, y_i) (1 \leq i \leq m)$ ，需要需要在

函数组  $\Phi = \text{span}(f_0, f_1, \dots, f_n)$ ，找到一个函数  $S^*(x) = \sum_{i=0}^n a_i f_i$ ，使得

$\|\delta_i\|^2 = \sum_{i=1}^m (y(x_i) - y_i)^2$  最小。经过一系列计算后，我们定义  $f_r = (f_r(x_1), f_r(x_2), \dots, f_r(x_m))$ ，

$y = (y_1, y_2, \dots, y_m)$ 。定义点乘运算符  $a * b = \sum_{i=1}^n a_i b_i$ 。则下列方程组的解即为需要的参数值

$a = (a_0, a_1, \dots, a_n)$ 。

$$\begin{bmatrix} f_0 * f_0 & f_0 * f_1 & \dots & f_0 * f_n \\ f_1 * f_0 & f_1 * f_1 & \dots & \dots \\ \dots & \dots & \dots & \dots \\ f_n * f_0 & \dots & \dots & f_n * f_n \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ \dots \\ a_n \end{bmatrix} = \begin{bmatrix} f_0 * y \\ f_1 * y \\ \dots \\ f_n * y \end{bmatrix}$$

而解方程组的算法在实验二列主元高斯消元法中已经实现，故在此实验中将直接调用高斯消元法以求得所需要的解。

因此，在此代码中关键在于解列主元高斯消元法中所需要的增广矩阵的构造上。只要解决了这个问题，问题就可以解决了。

### 2) 编写代码的思路

我们需要定义一个函数集  $\mu(x, i) = f_i(x)$ ，对于  $n$  次多项式，构造  $(n+1) * (n+2)$  的增广矩

阵，并对各个元素进行填充。

求曲线的最低点的方法一般通过数值的解法（即通过求导用迭代法求出零点，带回原式求出最低点）。但在这个例子中，二次，三次曲线的最低点都可以使用解析法求得，因此不再使用数值求法。

### 3) 伪代码

```
# points:=输入的点数据, points.x(i)表示xi, points.y(i)表示yi
# n:=问题规模
# u:=函数组 u(x,i)=ui(x)
def curve_get_para(points, n, u):
    d = matrix(n+1, n+2) # 构造矩阵
    for i in 0..n :
        for j in 0..n+1 :
            double v = 0 # v 表示每个格子的值
            if j != n+1 : # 对应fi * fj
                for m = 0..points.size-1 :
                    v = v + u(points.x(m),i)*u(points.x(m),j)
            else : # 对应fi * y
                for m = 0..point.size-1 :
                    v = v + u(points.x(m),i) * points.y(m)
            d[i, j] = v
    # 构造完成后解增广矩阵
    matrix a = d.solve()
    return a

# 此函数表示通过参数a和函数组u构造表达式的方法，略。
def curve_expression(a, u):
```

## 三、实验代码

见附录

### matlab 脚本

```
clear
clc

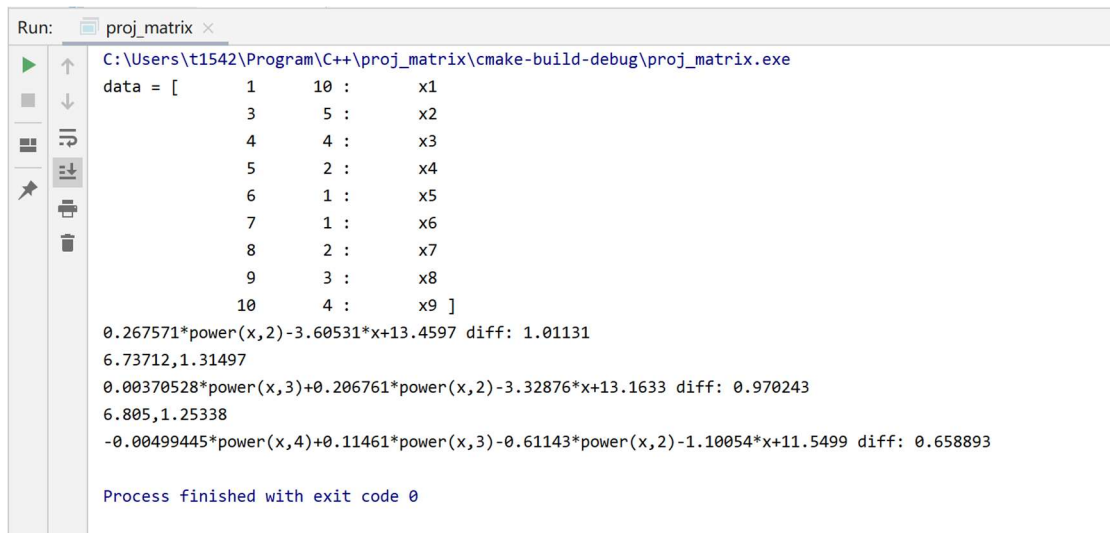
% points
figure()
x1 = [1 3 4 5 6 7 8 9 10];
y1 = [10 5 4 2 1 1 2 3 4];
plot(x1,y1,'red*')
% curve1
x = 1:0.01:10;
f = 0.267571*power(x,2)-3.60531*x+13.4597;
hold on
```

```

plot(x,f,'green-')
% min point1
px = 6.73712;
py = 1.31497;
hold on
plot(px,py,'green*')
% curve2
f = 0.206761*power(x,3)-3.32876*power(x,2)+13.1633*x+0.00370528;
hold on
plot(x,f,'blue-')
% min point2
px = 8.11939;
py = -1.89322;
hold on
plot(px,py,'blue*')
% curve3
f = -0.00499445*power(x,4)+0.11461*power(x,3)-0.61143*power(x,2)-
1.10054*x+11.5499;
hold on
plot(x,f,'black-.')

```

#### 四、实验结果及其讨论



Run: proj\_matrix x

C:\Users\t1542\Program\C++\proj\_matrix\cmake-build-debug\proj\_matrix.exe

```

data = [      1      10 :      x1
          3       5 :      x2
          4       4 :      x3
          5       2 :      x4
          6       1 :      x5
          7       1 :      x6
          8       2 :      x7
          9       3 :      x8
         10       4 :      x9 ]

0.267571*power(x,2)-3.60531*x+13.4597 diff: 1.01131
6.73712,1.31497
0.00370528*power(x,3)+0.206761*power(x,2)-3.32876*x+13.1633 diff: 0.970243
6.805,1.25338
-0.00499445*power(x,4)+0.11461*power(x,3)-0.61143*power(x,2)-1.10054*x+11.5499 diff: 0.658893

Process finished with exit code 0

```

图 1 程序执行结果

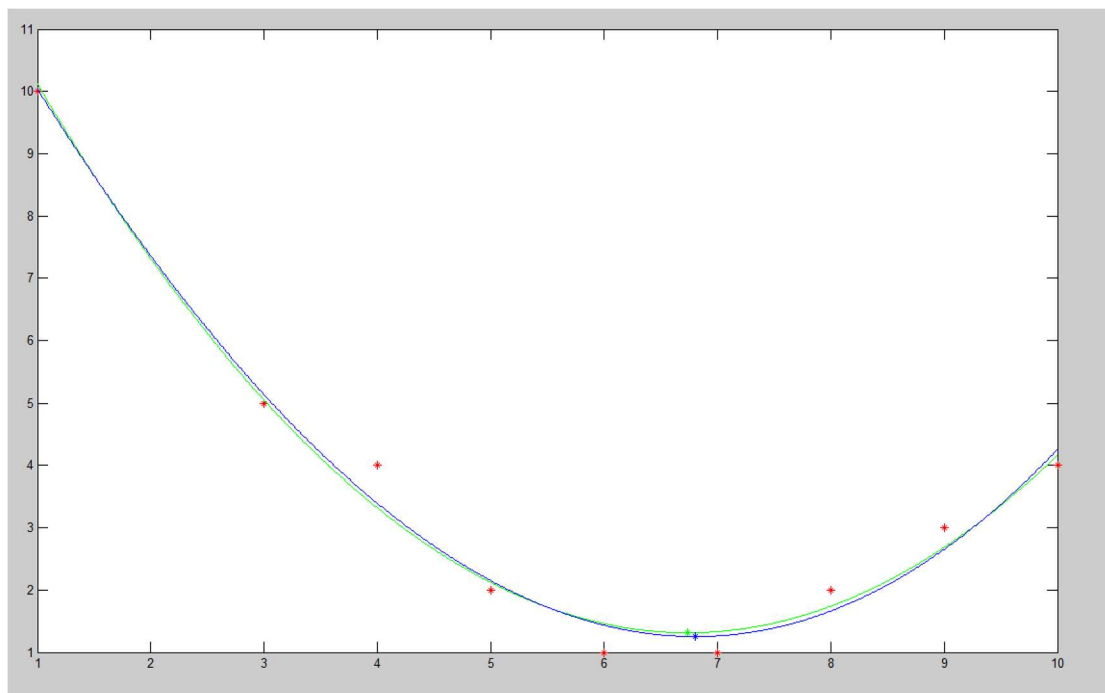
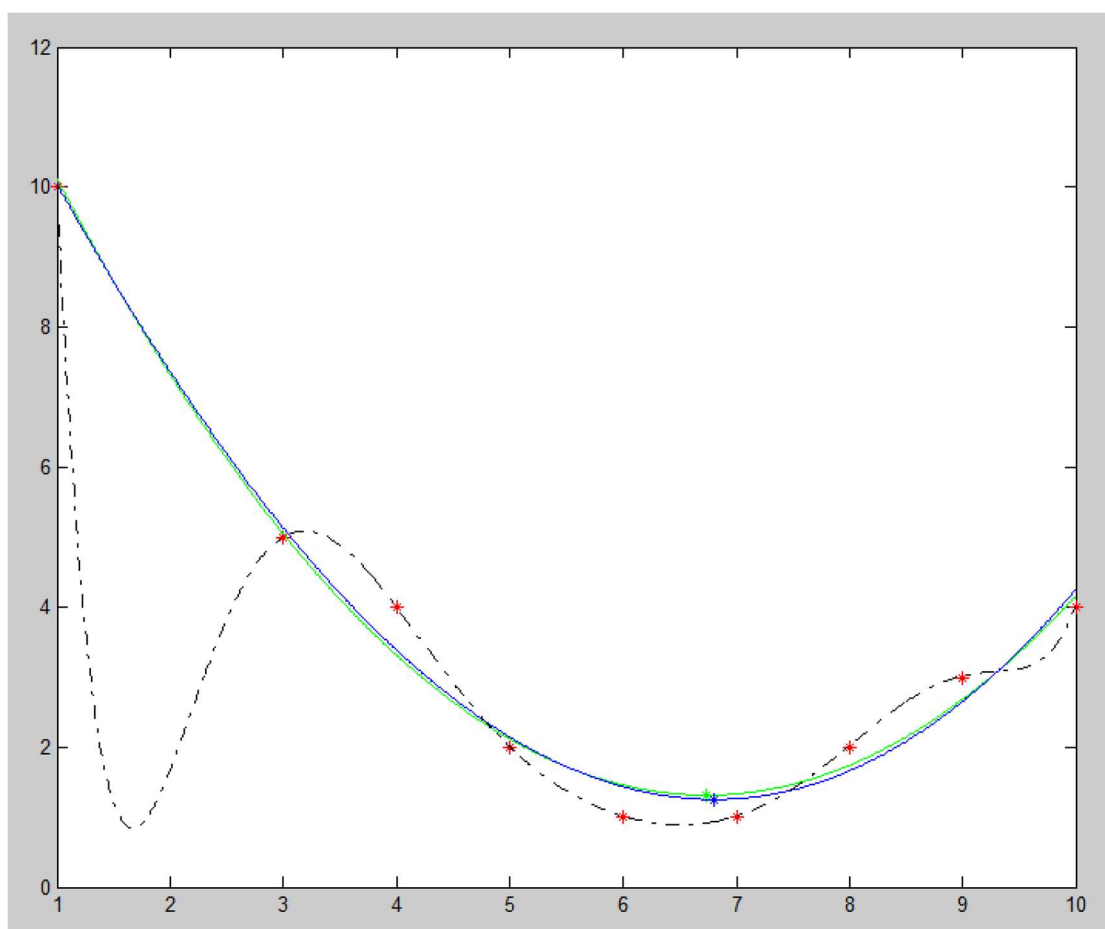


图 2 最小二乘法拟合曲线。



图三 最小二乘法拟合曲线（黑色为 8 次拟合曲线）

根据 C++ 代码的运行结果，两条拟合曲线分别为：（其中  $diff$  表示残差的平方和）

$$y = 0.267571x^2 - 3.60531x + 13.4597, diff = 1.01131$$

$$y = 0.00370528x^3 + 0.203761x^2 - 3.32876x + 13.1633, \text{diff} = 0.920743$$

在上面的图中，绿色线是二次函数的拟合曲线，蓝色线是三次函数的拟合曲线。观察发现三次函数的三次项系数很接近 0，所以两条曲线十分接近。且肉眼观察可以发现函数基本上是一条抛物线，**所以选择二次函数拟合比较合适**。在使用最小二乘法的方法中，一般第一步便是观察曲线的大致走向，确定函数类型，并进行计算。

根据进一步的实验可以知道，在每个点都有误差的前提下，并不是拟合的次数越高越接近实际的情况。**在高次拟合曲线中，容易导致测量误差被放大。**

## 五、总结

通过此实验，对最小二乘法拟合曲线的构造和高斯消元法有了更加深入的了解。通过实验，发现最小二乘法拟合曲线并不是次数越高越好，需要按照曲线的走势选取比较合适的拟合曲线。

## 六、附录（代码）

main.cpp

```
#include <iostream>
#include "matrix.h"
#include "points.h"
#include "liner_generator.h"

using namespace std;
int main() {
    liner_generator g;
    points p(matrix({
        {1, 10},
        {3, 5},
        {4, 4},
        {5, 2},
        {6, 1},
        {7, 1},
        {8, 2},
        {9, 3},
        {10, 4}
    }, {}), &g);
    p.data().print_with_title("data");
    double diff;
    matrix r = p.min_2_multi(2, diff);
    cout << r.to_expression() << " diff: " << diff << endl;
    auto p1 = r.min_point(1, 10);
    cout << p1.first << "," << p1.second << endl;
    matrix v = p.min_2_multi(3, diff);
    cout << v.to_expression() << " diff: " << diff << endl;
```

```

    auto p2 = v.min_point(1, 10);
    cout << p2.first << "," << p2.second << endl;
    matrix v2 = p.min_2_multi(4, diff);
    cout << v2.to_expression() << " diff: " << diff << endl;
}

```

matrix.h matrix.cpp (仅部分)

```

#pragma once
#include <iostream>
#include <iomanip>
#include <string>
#include <initializer_list>
#include <sstream>
#include <cmath>

using namespace std;

class matrix {
public:
    matrix(initializer_list<initializer_list<double>> data){
        get_row_column(data, _row_count, _column_count);
        init_data(data, fills, _row_count, _column_count);
    }
    matrix(initializer_list<initializer_list<double>> data,
initializer_list<string> marks): matrix(data){
        int i = 0;
        _marks = new pair<int, string>[_row_count];
        for(const auto& p: marks){
            if(i < _row_count)
                _marks[i++] = pair<int, string>(i, p);
        }
        for (;i< _row_count;++i) {
            _marks[i] = pair<int, string>(i, "x" + to_string(i+1));
        }
    }
    matrix(int rows, int columns, initializer_list<string> marks){
        _row_count = rows;
        _column_count = columns;
        fills = new double[rows * columns];
        int i = 0;
        _marks = new pair<int, string>[_row_count];
        for(const auto& p: marks){
            if(i < _row_count)
                _marks[i++] = pair<int, string>(i, p);
        }
    }

```

```

    }
    for (;i< _row_count;++i) {
        _marks[i] = pair<int, string>(i, "x" + to_string(i+1));
    }
}

matrix(const matrix& m){
    _is_error = m._is_error;
    _row_count = m._row_count;
    _column_count = m._column_count;
    fills = new double[_row_count * _column_count];
    for (int i = 0; i < _row_count * _column_count; ++i) {
        fills[i] = m.fills[i];
    }
    if(m.is_marked()){
        _marks = new pair<int, string>[_row_count];
        for (int i = 0; i < _row_count; ++i) {
            _marks[i] = m._marks[i];
        }
    }
}

int row_count() const { return _row_count; }
int column_count() const { return _column_count; }
void print(int w = 8) const;
void print_with_title(const string& title, int w = 8) const;

//向量的二范数
double vector_norm2();

matrix t();
matrix get_dialog();

~matrix(){
    if(fills != nullptr){
        delete[] fills;
    }
}

friend matrix operator + (const matrix& m1, const matrix& m2);
friend matrix operator - (const matrix& m1, const matrix& m2);
friend matrix operator * (const matrix& m1, const matrix& m2);
friend matrix operator * (double a, const matrix& m);
friend matrix operator ^ (const matrix& m1, int p);

matrix& operator = (const matrix& m);

```

```

matrix& operator -();

static matrix error_instance(){
    matrix m(0,0);
    m._is_error = true;
    return m;
}
static matrix e(int row);
static matrix dialog(initializer_list<double> data);

static matrix vector_add(const matrix& m1, const matrix& m2) ;
static matrix vector_multiply(const matrix& m1, const matrix& m2);
double expression_invoke(double v) const;
string to_expression() const;
void vector_adjust();
double& at(int row, int column) const;
matrix row_at(int row) const;
matrix column_at(int column) const;
bool is_cell() const;
bool is_vector() const;
bool is_row_vector() const;
bool is_column_vector() const;
bool is_marked() const;

pair<double, double> min_point(double l, double r) const;
//交换行
void swap_row(int row1, int row2);
void swap_row_only(int row1, int row2);
//交换列
void swap_column(int column1, int column2);
//int row1+=row2*p;
void apply_row(int row1, int row2, double p);
void main_gs();
void sort_with_mark();
matrix solve() const;
private:
    matrix(int _row_count, int _column_count): _row_count(_row_count),
    _column_count(_column_count){}
    static void
get_row_column(initializer_list<initializer_list<double>>& data, int&
row_count, int& column_count);
    static void init_data(initializer_list<initializer_list<double>>&
data, double*& fills, int& row_count, int& column_count);

```



```

static void init_array(double*& data, int length);
matrix inverse() const;
bool is_dialog() const;

//void resize(int _rows, int _columns);

bool _is_error = false;
// 行数
int _row_count;
// 列数
int _column_count;
// 填充
double* fills = nullptr;
// 辅助标记
pair<int, string>* _marks = nullptr;

};

```

```

matrix matrix::vector_add(const matrix &m1, const matrix &m2) {
    if(m1._row_count == 1 && m2._row_count == 1){
        const matrix* a = &m1;
        const matrix* b = &m2;
        if(a->_column_count < b->_column_count){
            a = &m2;
            b = &m1;
        }
        matrix m(1, a->_column_count);
        m.fills = new double[a->_column_count];
        for(int i = 0; i < b->_column_count; ++i)
            m.fills[i] = a->fills[i] + b->fills[i];
        for(int i = b->_column_count; i<a->_column_count;++i)
            m.fills[i] = a->fills[i];
        //m.vector_adjust();
        return m;
    } else {
        return matrix::error_instance();
    }
}

matrix matrix::vector_multiply(const matrix& m1, const matrix& m2) {
    if(m1._row_count == 1 && m2._row_count == 1){
        int length = m1._column_count + m2._column_count -1;
        matrix m(1, length);
    }
}

```

```

        m.fills = new double[length];
        fill(m.fills, length);
        for(int i = 0; i < m1._column_count; ++i){
            for(int j = 0; j < m2._column_count; ++j){
                m.fills[i+j] += m1.fills[i] * m2.fills[j];
            }
        }
        m.vector_adjust();
        return m;
    } else {
        return matrix::error_instance();
    }
}

matrix matrix::row_at(int row) {
    matrix m(1, _column_count);
    m.fills = new double[_column_count];
    for (int i = 0; i < _column_count; ++i) {
        m.fills[i] = fills[row * _column_count + i];
    }
    return m;
}

matrix matrix::column_at(int column) {
    matrix m(_row_count, 1);
    m.fills = new double[_row_count];
    for (int i = 0; i < _row_count; ++i) {
        m.fills[i] = fills[i * _column_count + column];
    }
    return m;
}

double matrix::at(int row, int column) {
    return fills[row * _column_count + column];
}

matrix operator*(double a, const matrix& m) {
    matrix m1 = matrix(m._row_count, m._column_count);
    int length = m._row_count * m._column_count;
    m1.fills = new double[length];
    for (int i = 0; i < length; ++i) {
        m1.fills[i] = a * m.fills[i];
    }
}

```

```

        return m1;
    }

    void matrix::vector_adjust() {
        if(_row_count == 1){
            int length = _column_count;
            int i;
            for (i = length - 1 ; i > 0; -- i) {
                if(fills[i] != 0)
                    break;
            }
            auto* f = new double[i + 1];
            for (int j = 0; j < i + 1; ++j) {
                f[j] = fills[j];
            }
            delete fills;
            fills = f;
        }
    }

    string matrix::to_expression() {
        if(_row_count == 1){
            ostringstream s;
            for (int i = _column_count - 1; i >= 0; --i) {
                if(i < _column_count - 1 && fills[i] > 0)
                    s << "+";
                s << fills[i];
                if(i > 0){
                    s << "*";
                    if(i == 1)
                        s << "x";
                    else
                        s << "power(x," << i << ")";
                }

            }
            return s.str();
        } else {
            return "";
        }
    }

    double matrix::expression_invoke(double v) {
        if(_row_count == 1){

```

```

        double r = 0;
        for (int i = 0; i < _column_count; ++i) {
            r += fills[i] * pow(v,i);
        }
        return r;
    } else {
        return -1;
    }
}

matrix matrix::solve() const {
    matrix m = *this;
    m.main_gs();
    //m.print_with_title("m");
    //init the r
    matrix r = matrix(_row_count, 1);
    r.fills = new double[_row_count];
    r._marks = new pair<int, string>[_row_count];
    for (int i = 0; i < _row_count; ++i) {
        r._marks[i] = m._marks[i];
    }

    //m.print_with_title("test::m");
    //r.print_with_title("test::r");
    //r.fills = new double[_row_count];
    for(int i = _row_count - 1; i >= 0; --i){
        double v = m.fills[i * _column_count + _row_count];
        for(int j = i + 1; j < _row_count; ++j){
            v -= m.fills[i * _column_count + j] * r.fills[j];
        }
        r.fills[i] = v / m.fills[i * _column_count + i];
    }
    //r.print_with_title("test::r");
    //r.sort_with_mark();
    //r.print_with_title("test::r::sorted");
    return r;
}

```

points.h points.cpp

```

#pragma once
#include "matrix.h"
#include "ploy.h"
#include <vector>
#include "multi_ploy.h"
#include "f_generator.h"

```

```

using namespace std;

// 用矩阵来表示的点集, 封装了求插值多项式, 拟合函数的一系列操作
class points {
public:
    explicit points(const matrix& m): _data(m){}
    points(const matrix& m, f_generator* generator): _data(m),
    _generator(generator){}
    // 获取插值多项式
    multi_ploy insertion_exp(int exp);
    // 获取最小二乘法曲线
    matrix min_2_multi(int exp, double& diff);

    matrix& data(){
        return _data;
    }
private:
    matrix _data;
    f_generator* _generator = nullptr;
};

```

```

#include "points.h"

matrix points::min_2_multi(int exp, double& diff) {
    // generator the matrix
    int rows = exp + 1;
    matrix m(rows, rows + 1, {});
    //m.print_with_title("m");
    for(int i = 0; i < rows; ++i){
        for(int j = 0; j < rows + 1; ++j){
            double r = 0;
            for(int k = 0; k < _data.row_count(); ++k){
                if(j != rows){ //not the last column
                    r += _generator->value(_data.at(k,0),i) *
                        _generator->value(_data.at(k,0),j);
                } else {
                    r += _generator->value(_data.at(k,0),i) *
                        _data.at(k,1);
                }
            }
            m.at(i,j) = r;
        }
    }
}

```

```

    }
    //m.print_with_title("test::m");
    matrix s = m.solve();
    //s.print_with_title("s");
    diff = 0;
    for (int i = 0; i < _data.row_count(); ++i) {
        diff += pow( s.expression_invoke(_data.at(i,0)) - _data.at(i,1) ,
2);
    }

    return s;
}

```

## ploy.h

```

#pragma once
#include "matrix.h"

class ploy {
public:
    ploy(double start, double end, const matrix& ploy_ana): _start(start),
_end(end), _curve(ploy_ana){}
    double start(){
        return _start;
    }
    double end(){
        return _end;
    }
    matrix curve(){
        return _curve;
    }
private:
    double _start;
    double _end;
    matrix _curve;
};

```

## insertion\_result.h insertion\_result.cpp

```

#pragma once
#include "matrix.h"
#include "ploy.h"
#include <utility>
#include <vector>
#include <initializer_list>

```

```

using namespace std;

class insertion_result {
public:
    insertion_result(vector<ploy> d): data(std::move(d)){}
    friend ostream& operator << (ostream& o, const insertion_result& r);
    void print_result(initializer_list<double> v);
private:
    vector<ploy> data;
};

```

```

#include "insertion_result.h"

ostream &operator<<(ostream &o, const insertion_result& r) {
    int i = 0;
    for(auto p:r.data){
        o << setw(8) << p.start() << setw(8) << p.end() << " " <<
p.curve().to_expression() << endl;
    }
    return o;
}

void insertion_result::print_result(initializer_list<double> v) {
    for(auto p:v){
        cout << setw(8);
        for(auto d:data){
            if(p >= d.start() && p<= d.end()){
                cout << setw(8) << d.curve().expression_invoke(p);
                break;
            }
        }
        cout << endl;
    }
}

```