```cpp
/*
 * Ryan Li
 * APP_Assignment_0
 * main.cpp
 * Sets up Vandermonde matrix v for polynomial, computes vT and vTv,
 * then solves for the polynomial coefficient matrix for a general polynomial.
 */

#include <iostream>
#include <iomanip>
#include <cmath>
#include <string>
#include <vector>

using namespace std;
static bool g_error = false;            // checks if inverse is possible

// define 2d vectors as matrices for clarity
typedef vector<vector<double>> matrix;

// set up matrix with passed dimensions
matrix createMatrix(int numRows, int numCols) {
    // define a single row of matrix; size = number of columns
    vector<double> row(numCols);

    // create a matrix with n rows of equal size; typecast for compatibility
    return matrix{ static_cast<unsigned int>(numRows), row };
}

// create vandermonde matrix with passed data
matrix getVandermonde(matrix xVals, int degree) {
    // set up (degree + 1) columns to account for constant
    matrix vandermonde{ createMatrix(xVals.size(), degree + 1) };

    // fill vandermonde
    for (int i = 0; i < xVals.size(); i++)
        for (int j = 0; j < degree + 1; j++)
            (j == 0) ?                  // check if degree is 0 (constants);
            vandermonde[i][j] = 1 :     // set constants col to 1, avoid 0^0
            vandermonde[i][j] = pow(xVals[i][0], j);

    return vandermonde;
}

// create transpose of a given matrix
matrix getTranspose(matrix original) {
    // transpose has flipped number of cols and rows
    matrix transposed{ createMatrix(original[0].size(), original.size()) };

```

```cpp
    // place element from (r,c) into (c,r) of transposed matrix
    for (int i = 0; i < transposed.size(); i++)
        for (int j = 0; j < transposed[0].size(); j++)
            transposed[i][j] = original[j][i];

    return transposed;
}

// multiply two matrices
matrix multiplyMatrix(matrix A, matrix B) {
    // for two matrices of dimensions axb, cxd, result will have size axd
    matrix result{ createMatrix(A.size(), B[0].size()) };

    // filling product matrix
    for (int i = 0; i < A.size(); i++) {
        for (int j = 0; j < B[0].size(); j++) {
            // element is first set to 0 then essentially the dot
            // product of A's row and B's column is calculated
            result[i][j] = 0;
            for (int k = 0; k < A[0].size(); k++) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return result;
}

// display matrix with given name
void printMatrix(matrix mtrx, string name) {
    // print buffer line
    cout << "------------\n" << name << ":\n\n";

    // print row by row
    for (int i = 0; i < mtrx.size(); i++) {
        for (int j = 0; j < mtrx[0].size(); j++) {
            // right align; setw gives 12 spaces to maintain matrix shape
            cout << setw(12) << mtrx[i][j];
        }
        cout << "\n";
    }
}

// displays input data
void printData(matrix xs, matrix ys) {
    // print buffer line
    cout << "------------\n" << "Input Data:\n\n";

    // print index, then x/y values in order
    // setw gives 10 spaces for formatting
```

```
 99      cout << "index:    {";          // index
100      for (int i = 0; i < xs.size(); i++) {
101          cout << setw(7) << i;
102      }
103      cout << "}\n" << "x-values: {"; // x vals
104      for (int i = 0; i < xs.size(); i++) {
105          cout << setw(7) << xs[i][0];
106      }
107      cout << "}\n" << "y-values: {"; // y vals
108      for (int i = 0; i < ys.size(); i++) {
109          cout << setw(7) << ys[i][0];
110      }
111      cout << "}\n";
112  }
113
114  // prints final polynomial
115  void printAnswer(matrix ans) {
116      cout << "------------\n"<< "Your final line of best fit is: y = ";
117
118      // organizes equation in ax^n+bx^(n-1)+...
119      for (int i = ans.size() - 1; i >= 0; i--) { // highest order first
120          if (ans.size() != 1) {      // check if degree > 0
121              // check to write + sign
122              if (ans[i][0] > 0 && i != ans.size() - 1)
123                  cout << "+";
124              // write coeff, then x^i
125              if (ans[i][0] != 0) {
126                  if (i == 0 || (i != 0 && ans[i][0] != 1))   // avoid 1x^n
127                      cout << ans[i][0];
128                  if (i != 0) {       // write x if not constant c
129                      cout << "x";
130                      if (i != 1)     // only write exponent if > 1
131                          cout << "^" << i;
132                  }
133              }
134          }
135          else
136              cout << ans[0][0];
137      }
138  }
139
140  // returns cofactor of mtrx at the break column;
141  // recognized as the matrix that is produced by
142  // removing the row and column of the chosen index
143  matrix getCofactor(matrix mtrx, int breakRow, int breakCol) {
144      int size = mtrx.size();
145
146      // cofactor is square with size 1 less than original
147      matrix ret{ createMatrix(size - 1, size - 1) };
```

```
148
149     // set up new row/col indices to track index of ret array
150     int newRow{ 0 };
151     int newCol{ 0 };
152
153     // fill cofactor matrix with values of original matrix
154     for (int i = 0; i < size; i++) {
155         for (int j = 0; j < size; j++) {
156             if (i != breakRow && j != breakCol) {// ignore break row/col
157                 ret[newRow][newCol] = mtrx[i][j];
158
159                 newCol++;                       // move to next empty index;
160                 if (newCol == size - 1) {       // set to start of new row
161                     newRow++;                   // when end of row is reached
162                     newCol = 0;
163                 }
164             }
165         }
166     }
167
168     return ret;
169 }
170
171 // calculates determinant of nxn matrix; recursive, resolving
172 // the cofactor until size 2 base case is achieved
173 double getDeterminant(matrix mtrx) {
174     double det = 0;
175     int size = mtrx.size();
176
177     if (size == 1)          // det of 1x1 is just the element
178         return mtrx[0][0];
179     else if (size == 2)     // det of 2x2 is ad-bc
180         return mtrx[0][0] * mtrx[1][1] - mtrx[0][1] * mtrx[1][0];
181     else {                  // nxn, n > 2
182         for (int i = 0; i < size; i++) {
183             // get cofactors for each element of top row
184             matrix cofactMtrx{ getCofactor(mtrx, 0, i) };
185
186             // formula for det: current element in matrix times cofactor's
187             // determinant. the -1^i indicates that adjacent elements
188             // are treated with opposite signs.
189             det += mtrx[0][i] * pow(-1, i) * getDeterminant(cofactMtrx);
190         }
191     }
192
193     return det;
194 }
195
196 // returns the inverse of the passed matrix if possible;
```

```
197 // uses adjoint and determinant to calculate where
198 // A-1 = adjoint(A)/det(A)
199 matrix getInverse(matrix mtrx) {
200     if (getDeterminant(mtrx) == 0) {    // error message when det = 0
201         g_error = true;
202         cout << "The determinant is 0 and the matrix has no inverse.\n";
203         return matrix{ 0 };
204     }
205     else if (mtrx.size() == 1) {        // 1x1 matrices all have det = [1]
206         matrix ret{ createMatrix(1, 1) };
207         ret[0][0] = 1;
208 
209         return ret;
210     }
211 
212     // calculating inverse
213 
214     matrix inverse{ createMatrix(mtrx.size(), mtrx[0].size()) };
215 
216     // calculates the adjoint matrix by replacing each element with the
217     // determinant of its cofactor and transposing the result, divides
218     // by original's determinant to get inverse
219     for (int i = 0; i < mtrx.size(); i++) {
220         for (int j = 0; j < mtrx[0].size(); j++) {
221             // power of -1 because value is pos when (row+col)%2==0
222             inverse[i][j] = pow(-1, i + j)
223                 * getDeterminant(getCofactor(mtrx, i, j));
224             inverse[i][j] /= getDeterminant(mtrx);
225         }
226     }
227 
228     return getTranspose(inverse);       // transpose to get inverse
229 }
230 
231 int main() {
232     g_error = false;        // becomes true if error is detected
233 
234     // get data points
235     cout << "Enter number of data points: ";
236     int numEntries{};
237     cin >> numEntries;
238 
239     // initialize data sets as one column matrices so we can use
240     // multiplyMatrix() after when solving for coefficients
241     matrix xVals{ createMatrix(numEntries, 1) };
242     matrix yVals{ createMatrix(numEntries, 1) };
243 
244     for (int i = 0; i < numEntries; i++) {
245         // (i+1) since array starts at 0, but we prompt for (i+1)th
```

```
246         cout << "---\n" << "Enter x" << i + 1 << ": ";
247         cin >> xVals[i][0];
248         cout << "Enter y" << i + 1 << ": ";
249         cin >> yVals[i][0];
250     }
251
252     // acquire degree of polynomial
253     int degree{};
254     cout << "------------\n";
255     cout << "Enter the integer degree of desired polynomial: " ;
256     cin >> degree;
257
258     // create v, vT, vTv
259     matrix v{ getVandermonde(xVals, degree) };
260     matrix vT{ getTranspose(v) };
261     matrix vTv{ multiplyMatrix(vT, v) };
262
263     // solve for coefficients for line of best fit
264     // solution is a = (vTv)-1*vT*y, for polynomial coeffs matrix a
265     matrix vTvInverse{ getInverse(vTv) };
266     matrix inverseMultiplied{ multiplyMatrix(vTvInverse, vT) };
267     matrix solution{ multiplyMatrix(inverseMultiplied, yVals) }; // = a
268
269     // print input data and three matrices
270     printData(xVals, yVals);
271     printMatrix(v, "v");
272     printMatrix(vT, "vT");
273     printMatrix(vTv, "vTv");
274
275     if (!g_error) {         // if inverse was calculated
276         // print solution
277         printMatrix(solution, "Solving for Coefficients in Increasing Order");
278         // print final polynomial
279         printAnswer(solution);
280     }
281
282     // message just in case a coeff is close to 0 or something
283     cout << "\n------------\n";
284     cout << "Values can have slight error, usually less than 1e-12.\n" ;
285
286     return 0;
287 }
```