Московский авиационный институт

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Лабораторная работа № 3
по дисциплине
"Численные методы"

Студент: Балес А.И.
Преподаватель: Ревизников Д.Л.
Дата:
Оценка:
Подпись:

Москва, 2016

Метод 1 — Интерполяция

Задание
Используя таблицу значений Y_i функции $y = f(x)$, вычисленных в точках X_i, $i = 0, ..., 3$ построить интерполяционные многочлены Лагранжа и Ньютона, проходящие через точки X_i, Y_i. Вычислить значение погрешности интерполяции в точке X^*.

Вариант: 3
$y = tg(x)$, а) $X_i = 0, \frac{\pi}{8}, \frac{2\pi}{8}, \frac{3\pi}{8}$; б) $X_i = 0, \frac{\pi}{8}, \frac{\pi}{3}, \frac{3\pi}{8}$; $X^* = \frac{3\pi}{16}$

Описание алгоритма
Пусть на отрезке $[a, b]$ задано множество несовпадающих точек x_i (интерполяционных узлов), в которых известны значения функции $f_i = f(x_i), i = 0, ..., n$. Приближающая функция $\varphi(x, a)$ такая, что выполняются равенства

$$\varphi(x_i, a_0, ..., a_n) = f(x_i) = f_i, i = 0, ..., n$$

называется интерполяционной.
Наиболее часто в качестве приближающей функции используют многочлены степени n:

$$P_n(x) = \sum_{i=0}^{n} a_i x^i$$

Произвольный многочлен может быть записан в виде:

$$L_n(x) = \sum_{i=0}^{n} f_i l_i(x)$$

Здесь $l_i(x)$ — многочлены степени n, так называемые лагранжевы многочлены влияния, которые удовлетворяют условию $l_i(x_j) = \begin{cases} 1, i = j \\ 0, i \neq j \end{cases}$ и, соответственно, $l_i(x) = \prod\limits_{j=0, j\neq i}^{n} \frac{(x-x_i)}{(x_i-x_j)}$, а интерполяционный многочлен запишется в виде:

$$L_n(x) = \sum_{i=0}^{n} f_i \prod_{j=0, j\neq i}^{n} \frac{(x - x_i)}{(x_i - x_j)}$$

Данный интерполяционный многочлен называется интерполяционным многочленом Лагранжа.

Введем понятие разделенной разности. Разделенные разности нулевого порядка совпадают со значениями функции в узлах. Разделенные разности первого порядка обозначаются $f(x_i, x_j)$ и определяются через разделенные разности нулевого порядка:

$$f(x_i, x_j) = \frac{f_i - f_j}{x_i - x_j}$$

разделенные разности второго порядка определяются через разделенные разности первого порядка:

$$f(x_i, x_j, x_k) = \frac{f(x_i, x_j) - f(x_j, x_k)}{x_i - x_k}$$

Разделенная разность порядка $n - k + 2$ определяется соотношениями:

$$f(x_i, x_j, x_k, ..., x_{n-1}, x_n) = \frac{f(x_i, x_j, x_k, ..., x_{n-1}) - f(x_j, x_k, ..., x_n)}{x_i - x_n}$$

Пусть известны значения аппроксимируемой функции $f(x)$ в точках $x_0, x_1, ..., x_n$. Интерполяционный многочлен, значения которого в узлах интерполяции совпадают со значениями функции $f(x)$ может быть записан в виде:

$P_n(x) = f(x_0) + (x - x_0)f(x_1, x_0) + (x - x_0)(x - x_1)f(x_0, x_1, x_2) + ...+$
$+(x - x_0)(x - x_1)...(x - x_n)f(x_0, x_1, ..., x_n)$

Данная запись многочлена есть так называемый интерполяционный многочлен Ньютона.

Реализация

Лагранж

```
void PolynomialLagrange() {
  auto Y = [](double x) -> double { return tan(x);
     };
  auto Product = [](const vector<double>& X, double
     curX, size_t pos) -> double {
    double temp = 1.0;
    for (size_t i = 0; i < 4; ++i) {
      if (i == pos) continue;
```

```
      temp *= (curX - X[i]) / (X[pos] - X[i]);
    }
    return temp;
  };

  vector<double> X;
  // task a)
  /*
  X.push_back(0);
  X.push_back(1.0 * M_PI / 8);
  X.push_back(2.0 * M_PI / 8);
  X.push_back(3.0 * M_PI / 8);
  */
  //task b)
  X.push_back(0);
  X.push_back(1.0 * M_PI / 8);
  X.push_back(1.0 * M_PI / 3);
  X.push_back(3.0 * M_PI / 8);

  double perfectX = 3.0 * M_PI / 16;
  double sum = 0.0;
  for (size_t i = 0; i < 4; ++i) {
    sum += Y(X[i]) * Product(X, perfectX, i);
  }
  cout << "L(" << perfectX << ") = " << sum << endl;
  cout << "y(" << perfectX << ") = " << Y(perfectX)
     << endl;
  cout << "|L(" << perfectX << ") - y(" << perfectX
     << ")| = " << fabs(Y(perfectX) - sum) << endl;
}
```

Ньютон

```
void PolynomialNewtoon() {
  auto Y = [](double x) -> double { return sin(M_PI
     * x / 6); };
  function<double(const vector<double>&, double,
     size_t, size_t)> FuncY;
  FuncY = [Y, &FuncY](const vector<double>& X,
     double curX, size_t start, size_t end) ->
     double {
```

```
  if (start == end) {
    return Y(X[start]);
  }
  else {
    return (FuncY(X, curX, start, end - 1) -
       FuncY(X, curX, start + 1, end)) /
       (X[start] - X[end]);
  }
};
auto Product = [](const vector<double>& X, double
   curX, size_t sz) -> double {
  double prod = 1.0;
  for (size_t i = 0; i < sz; ++i)
    prod *= (curX - X[i]);
  return prod;
};

vector<double> X;
// task a)
/*
X.push_back(0);
X.push_back(1.0 * M_PI / 8);
X.push_back(2.0 * M_PI / 8);
X.push_back(3.0 * M_PI / 8);
*/
//task b)
X.push_back(0);
X.push_back(1.0 * M_PI / 8);
X.push_back(1.0 * M_PI / 3);
X.push_back(3.0 * M_PI / 8);


double perfectX = 1.5;
double sum = 0.0;
for (size_t i = 1; i < 4; ++i) {
  sum += FuncY(X, perfectX, 0, i) * Product(X,
     perfectX, i);
}
cout << "P(" << perfectX << ") = " << sum << endl;
cout << "y(" << perfectX << ") = " << Y(perfectX)
   << endl;
```

```
  cout << "|P(" << perfectX << ") - y(" << perfectX
     << ")| = " << fabs(Y(perfectX) - sum) << endl;

}
```

Тестирование

Выходной файл

```
a)
Polynomial Lagrange
L(0.589049) = 0.644607
y(0.589049) = 0.668179
|L(0.589049) - y(0.589049)| = 0.0235719
****************************************
Polynomial Newtoon
P(1.5) = 0.70664
y(1.5) = 0.707107
|P(1.5) - y(1.5)| = 0.000467104
****************************************

б)
Polynomial Lagrange
L(0.589049) = 0.585251
y(0.589049) = 0.668179
|L(0.589049) - y(0.589049)| = 0.0829278
****************************************
Polynomial Newtoon
P(1.5) = 0.706792
y(1.5) = 0.707107
|P(1.5) - y(1.5)| = 0.000314796
****************************************
```

Метод 2 — Кубический сплайн

Задание
Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при $x = x_0$ и $x = x_4$. Вычислить значение функции в точке $x = X^*$.

Вариант: 3
$X^* = 1.5$

i	0	1	2	3	4
x_i	0.0	0.9	1.8	2.7	3.6
f_i	0.0	0.36892	0.85408	1.7856	6.3138

Описание алгоритма
Наиболее широко применяемым является случай, когда между любыми двумя точками разбиения исходного отрезка строится многочлен n–й степени:

$$S(x) = \sum_{k=0}^{n} a_{ik}x^k, x_{i-1} \leq x \leq x_i, i = 1, ..., n$$

который в узлах интерполяции принимает значения аппроксимируемой функции и непрерывен вместе со своими $(n-1)$ производными. Такой кусочно-непрерывный интерполяционный многочлен называется сплайном. Его коэффициенты находятся из условий равенства в узлах сетки значений сплайна и приближаемой функции, а также равенства $n-1$ производных соответствующих многочленов. На практике наиболее часто используется интерполяционный многочлен третьей степени, который удобно представить как:

$S(x) = a_i + b_i(x - x_{i-1}) + c_i(x - x_{i-1})^2 + d_i(x - x_{i-1})^3$
$x_{i-1} \leq x \leq x_i, i = 1, 2, ..., n$

Для построения кубического сплайна необходимо построить n многочленов третьей степени, т.е. определить $4n$ неизвестных a_i, b_i, c_i, d_i. Эти коэффициенты ищутся из условий в узлах сетки.

$S(x_{i-1}) = a_i = a_{i-1} + b_{i-1}(x_{i-1} - x_{i-2}) + c_{i-1}(x_{i-1} - x_{i-2})^2 + d_i(x_{i-1} - x_{i-2})^3 = f_{i-1}$
$S'(x_{i-1}) = b_i = b_{i-1} + 2c_{i-1}(x_{i-1} - x_{i-2}) + 3d_{i-1}(x_{i-1} - x_{i-2})^2$
$S''(x_{i-1}) = 2c_i = 2c_{i-1} + 6d_{i-1}(x_{i-1} - x_{i-2}), i = 2, 3, ..., n$

$S(x_0) = a_1 = f_0$
$S''(x_0) = c_1 = 0$
$S(x_n) = a_n + b_n(x_n - x_{i-1}) + c_n(x_n - x_{i-1})^2 + d_n(x_n - x_{i-1})^3 = f_n$
$S''(x_n) = c_n + 3d_n(x_n - x_{i-1}) = 0$

Предполагается, что сплайны имеют нулевую кривизну на концах отрезка.
Если ввести обозначение $h_i = x_i - x_{i-1}$, и исключить из системы a_i, b_i, d_i, то можно получить систему из $n-1$ линейных алгебраических уравнений относительно $c_i, i = 2, ..., n$ с трехдиагональной матрицей:

$2(h_1 + h_2)c_2 + h_2c_3 = 3[(f_2 - f_1)/h_2 - (f_1 - f_0)/h_1]$
$h_{i-1}c_{i-1} + 2(h_{i-1} + h_i)c_i + h_ic_{i+1} = 3[(f_i - f_{i-1})/h_i - (f_{i-1} - f_{i-2})/h_{i-1}], i = 3, ..., n-1$
$h_{n-1}c_{n-1} + 2(h_{n-1} + h_n)c_n = 3[(f_n - f_{n-1})/h_n - (f_{n-1} - f_{n-2})/h_{n-1}]$

Остальные коэффициенты сплайнов могут быть восстановлены по формулам:

$a_i = f_{i-1}, i = 1, ..., n; \quad b_i = (f_i - f_{i-1})/h_i - \frac{1}{3}h_i(c_{i+1} + 2c_i), \quad d_i = \frac{c_{i+1} - c_i}{3h_i}, i = 1, ..., n-1$
$c_1 = 0, b_n = (f_n - f_{n-1})/h_n - \frac{2}{3}h_nc_n, d_n = -\frac{c_n}{3h_n}$

Реализация

```
#include "./dependens/TSolve.h"
#include <map>
#include <utility>

using namespace std;

static const size_t powerSplane = 3;

struct TData {
  pair<double, double> pSection;
  vector<double> vKoef;
};

int main(int argc, char* argv[]) {
  if (argc != 2) {
    cerr << "Error: count of args is incorrect" <<
```

```
      endl;
    exit(-1);
}
string dataFile = argv[1];
ifstream in(dataFile, ios::in);

size_t N;
vector<double> X, Y;
vector<TData> table;
double perfectX;
double temp;

in >> N;
for (size_t i = 0; i < N; ++i) {
  in >> temp;
  X.push_back(temp);
}
for (size_t i = 0; i < N; ++i) {
  in >> temp;
  Y.push_back(temp);
}
in >> perfectX;
in.close();

for (size_t i = 1; i < N; ++i) {
  TData temp;
  temp.pSection = make_pair(X[i - 1], X[i]);
  table.push_back(temp);
}

string inputFile  = "./dependens/inputData";
string outputFile   = "./dependens/outputData";

ofstream out(inputFile, ios::out);
N--;
out << N - 1 << endl;
out << 2 * (X[2] - X[0]) << " " << (X[2] - X[1])
   << " " << 0.0 << " " << 3 * ((Y[2] - Y[1]) /
   (X[2] - X[1]) - (Y[1] - Y[0]) / (X[1] - X[0]))
   << endl;
for (size_t i = 3; i <= N - 1; ++i) {
```

```
    out << (X[i - 1] - X[i - 2]) << " " << 2 *
        (X[i] - X[i - 2]) << " " << (X[i] - X[i -
        1]) << " " << 3 * ((Y[i] - Y[i - 1]) / (X[i]
        - X[i - 1]) - (Y[i - 1] - Y[i - 2]) / (X[i -
        1] - X[i - 2])) << endl;
}
out << (X[N - 1] - X[N - 2]) << " " << 2 * (X[N]
    - X[N - 2]) << " " << 0.0 << " " << 3 * ((Y[N]
    - Y[N - 1]) / (X[N] - X[N - 1]) - (Y[N - 1] -
    Y[N - 2]) / (X[N - 1] - X[N - 2])) << endl;
out.close();
TSolve sol(inputFile, outputFile);
if (!!sol.ToSolveByTripleDiagMatrix()) {
    cerr << "Error: Troubles with method
        TripleDiagMatrix!" << endl;
    exit(-1);
}
in.close();
in.open(outputFile);
vector<double> vecC;
vecC.push_back(0.0);
while (in >> temp) {
    vecC.push_back(temp);
}
in.close();
for (size_t i = 1; i < N; ++i) {
    vector<double> t(powerSplane + 1);
    t[0] = Y[i - 1];
    t[1] = (Y[i] - Y[i - 1]) / (X[i] - X[i - 1]) -
        1.0 / 3 * (X[i] - X[i - 1]) * (vecC[i] + 2 *
        vecC[i - 1]);
    t[2] = vecC[i - 1];
    t[3] = (vecC[i] - vecC[i - 1]) / (3 * (X[i] -
        X[i - 1]));
    table[i - 1].vKoef = t;
}
vector<double> vecTemp;
vecTemp.push_back(Y[N - 1]);
vecTemp.push_back((Y[N] - Y[N - 1]) / (X[N] - X[N
    - 1]) - 2.0 / 3 * (X[N] - X[N - 1]) * vecC[N -
    1]);
```

```
  vecTemp.push_back(vecC[N - 1]);
  vecTemp.push_back(-vecC[N - 1] / (3 * (X[N] - X[N
     - 1])));
  table[N - 1].vKoef = vecTemp;
  // draw graphics
  double deltaX = 0.01;
  for (size_t i = 0; i < table.size(); ++i) {
    double start = table[i].pSection.first;
    double end = table[i].pSection.second;
    ofstream out("plotData" + to_string(i),
       ios::out);
    for (double cur = start; cur <= end; cur +=
       deltaX) {
      double f = .0;
      for (size_t k = 0; k < table[i].vKoef.size();
         ++k) {
        f += table[i].vKoef[k] * pow(cur - start, k
           * 1.0);
      }
      out << cur << " " << f << endl;
    }
    cout << "[" << table[i].pSection.first << "; "
       << table[i].pSection.second << "]\t";
    for (size_t j = 0; j < table[i].vKoef.size();
       ++j)
      cout << table[i].vKoef[j] << " ";
    cout << endl;
  }
  return 0;
}
```

Тестирование

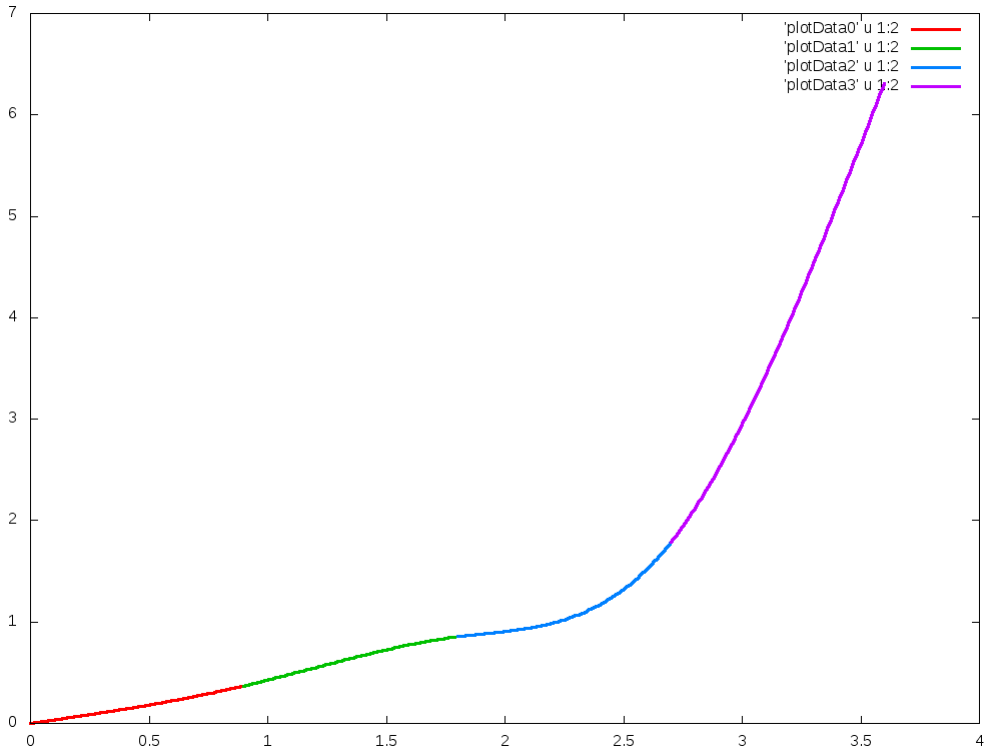
Входной файл

```
5
0.0 0.9 1.8 2.7 3.6
0.0 0.36892 0.85408 1.7856 6.3138
1.5
```

Выходной файл

```
[0; 0.9] 0  0.339411  0  0.087037
[0.9; 1.8] 0.36892  0.551067  0.235  -0.275926
[1.8; 2.7] 0.85408  0.303022  -0.51  1.47037
[2.7; 3.6] 1.7856  2.95533  3.46  -1.28148
```

График



Метод 3 — МНК–аппроксимация

Задание
Для таблично заданной функции путем решения нормальной системы МНК найти приближающие многочлены а) 1–ой и б) 2–ой степени. Для каждого из приближающих многочленов вычислить вычислить сумму квадратов ошибок. Построить графики приближаемой функции и приближающих многочленов.

Вариант: 3

i	0	1	2	3	4	5
x_i	-0.9	0.0	0.9	1.8	2.7	3.6
y_i	-0.36892	0.0	0.36892	0.85408	1.7856	6.3138

Описание алгоритма
Пусть задана таблично в узлах x_j функция $y_j = f(x_j), j = 0, 1, ..., N$. При этом значения функции y_j определены с некоторой погрешностью, также из физических соображений известен вид функции, которой должны приближенно удовлетворять табличные точки, например: многочлен степени n, у которого неизвестны коэффициенты $a_i, F_n(x) = \sum\limits_{i=0}^{n} a_i x^i$. Неизвестные коэффициенты будем находить из условия минимума квадратичного отклонения многочлена от таблично заданной функции.

$$\Phi = \sum_{j=0}^{N} [F_n(x_j) - y_j]^2$$

Минимума Ф можно добиться только за счет изменения коэффициентов многочлена $F_n(x)$. Необходимые условия экстремума имеют вид

$$\frac{\partial \Phi}{\partial a_k} = 2 \sum_{j=0}^{N} [\sum_{i=0}^{n} a_i x_j^i - y_j] x_j^k = 0, k = 0, 1, ..., n$$

Эту систему для удобства преобразуют к следующему виду:

$$\sum_{i=0}^{n} a_i \sum_{j=0}^{N} x_j^{k+i} = \sum_{j=0}^{N} y_j x_j^k, k = 0, 1, ..., n$$

Затем нужно решить систему и, тем самым, получить коэффициенты $a_0, a_1, ..., a_n$ для полинома.

Реализация

```
#include "./dependens/TSolve.h"
#include <fstream>
#include <functional>

using namespace std;

static size_t powMNK = 2;

int ToInt(const string& str) {
  int res = 0;
  for (int i = str.length() - 1; i >= 0 ; --i) {
    res = res * 10 + (str[i] - '0');
  }
  return res;
}

int main(int argc, char* argv[]) {
  if (argc != 3) {
    cerr << "Error: arg is incorrect" << endl;
    exit(-1);
  }
  string dataFile = argv[1];
  powMNK = max(0, ToInt(argv[2]));
  ifstream in(dataFile, ios::in);

  size_t N;
  vector<double> X, Y;
  double temp;

  in >> N;
  for (size_t i = 0; i < N; ++i) {
    in >> temp;
    X.push_back(temp);
  }
  for (size_t i = 0; i < N; ++i) {
    in >> temp;
    Y.push_back(temp);
  }
  in.close();
```

```
TMatrix matrixFI(N, powMNK + 1, Zero);
for (size_t i = 0; i < N; ++i) {
  for (size_t j = 0; j < powMNK + 1; ++j) {
    matrixFI[i][j] = pow(X[i], 1.0 * j);
  }
}
TVector vecY(Y);
string inputData  = "./dependens/inputData";
string outputData   = "./dependens/outputData";
ofstream input(inputData, ios::out);
ifstream output(outputData, ios::in);
TSolve solution(inputData, outputData);

TMatrix writeMatrix = matrixFI.Rotate() *
   matrixFI;
TVector writeVector = (matrixFI.Rotate() *
   vecY.Rotate()).Rotate();
input << powMNK + 1 << endl;
for (size_t j = 0; j < writeMatrix.GetSizeRow();
   ++j) {
  for (size_t i = 0; i <
     writeMatrix.GetSizeCol(); ++i) {
    input << writeMatrix[j][i] << " ";
  }
  input << writeVector[j] << endl;
}
if (!!solution.ToSolveByGauss()) {
  cerr << "Error: troubles with LUP method, check
     input data" << endl;
  exit(-1);
}
input.close();
output.close();

vector<double> aproxVec(powMNK + 1);
in.open(outputData);

for (size_t i = 0; i < aproxVec.size(); ++i) {
  in >> aproxVec[i];
}
```

```
in.close();

auto CalcApproxFunction = [](double x, const
   vector<double>& arr) -> double {
  double res = 0.0;
  for (size_t i = 0; i < arr.size(); ++i)
    res += arr[i] * pow(x, 1.0 * i);
  return res;
};

string resultFile = "./solution3_3";
ofstream o(resultFile, ios::out);

o << "X: ";
for (size_t i = 0; i < N; ++i) {
  o << X[i] << " ";
}
o << endl;
o << "Y: ";
for (size_t i = 0; i < N; ++i) {
  o << Y[i] << " ";
}
o << endl;
o << "Approx koef: ";
double sumOfError = .0;
for (size_t i = 0; i < N; ++i) {
  o << CalcApproxFunction(X[i], aproxVec) << " ";
  sumOfError += (CalcApproxFunction(X[i],
     aproxVec) - Y[i]) *
     (CalcApproxFunction(X[i], aproxVec) - Y[i]);
}
o << endl;
o << "Summ of errors: " << sumOfError << endl;
o.close();

string pointFiles = "./plotDataPoint";
string approxFiles = "./plotDataApprox" +
   to_string(powMNK);
ofstream outPoint(pointFiles, ios::out);
ofstream outApprox(approxFiles, ios::out);
```

```
  for (size_t i = 0; i < N; ++i) {
    outPoint << X[i] << " " << Y[i] << endl;
  }
  outPoint.close();
  double deltaX = 0.01;
  for (size_t i = 0; i < N - 1; ++i) {
    for (double cur = X[i]; cur <= X[i + 1]; cur +=
       deltaX) {
      outApprox << cur << " " <<
         CalcApproxFunction(cur, aproxVec) << endl;
    }
  }
  outApprox.close();
  return 0;
}
```

Тестирование

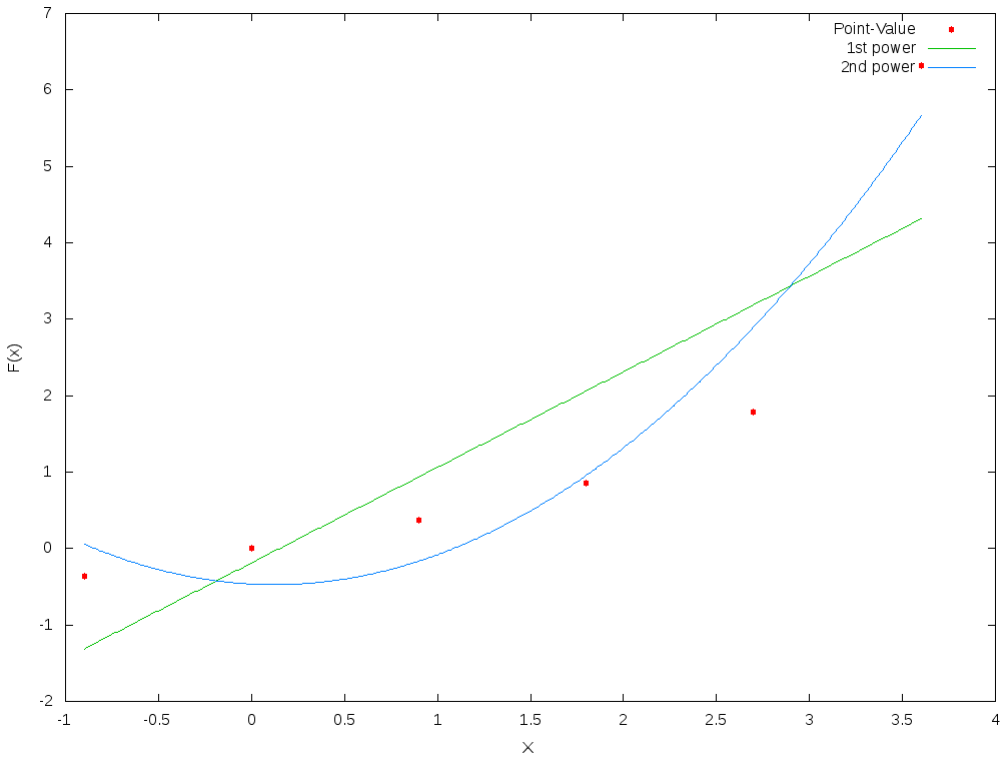
Входной файл

```
6
-0.9 0.0 0.9 1.8 2.7 3.6
-0.36892 0.0 0.36892 0.85408 1.7856 6.3138
```

Выходной файл

```
-0.9  -0.36892
0  0
0.9  0.36892
1.8  0.85408
2.7  1.7856
3.6  6.3138
```

График


Point-Value
1st power
2nd power
F(x)
X
7
6
5
4
3
2
1
0
-1
-2
-1
-0.5
0
0.5
1
1.5
2
2.5
3
3.5
4

Метод 4 — Численное дифференцирование

Задание

Вычислить первую и вторую производную от таблично заданной функции $y_i = f(x_i), i = 0, 1, 2, 3, 4$ в точке $x = X^*$.

Вариант: 3

$X^* = 2.0$

i	0	1	2	3	4
x_i	1.0	1.5	2.0	2.5	3.0
y_i	0.0	0.40547	0.69315	0.91629	1.0986

Описание алгоритма

Формулы численного дифференцирования в основном используется при нахождении производных от функции $y = f(x)$, заданной таблично. Исходная функция $y_i = f(x_i), i = 0, 1, ..., M$ на отрезках $[x_j, x_{j+k}]$ заменяется некоторой приближающей, легко вычисляемой функцией $\varphi(x, \overline{a}), y = \varphi(x, \overline{a}) + R(x)$, где $R(x)$ — остаточный член приближения, $\overline{a}$ — набор коэффициентов. Наиболее часто в качестве приближающей функции $\varphi(x, \overline{a})$ берется интерполяционный многочлен $\varphi(x, \overline{a}) = P_n(x) = \sum_{i=0}^{n} a_i x^i$, а производные соответствующих порядков определяются дифференцированием многочлена.

Первая производная:

$$y'(x) \approx \varphi'(x) = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} + \frac{\frac{y_{i+2} - y_{i+1}}{x_{i+2} - x_{i+1}} - \frac{y_{i+1} - y_i}{x_{i+1} - x_i}}{x_{i+2} - x_i}(2x - x_i - x_{i+1}), x \in [x_i, x_{i+1}]$$

Вторая производная:

$$y''(x) \approx \varphi''(x) = 2\frac{\frac{y_{i+2} - y_{i+1}}{x_{i+2} - x_{i+1}} - \frac{y_{i+1} - y_i}{x_{i+1} - x_i}}{x_{i+2} - x_i}, x \in [x_i, x_{i+1}]$$

Реализация

```
#include <iostream>
#include <functional>
#include <fstream>
#include <vector>
```

```
using namespace std;

int main(int argc, char* argv[]) {
  auto firstDerivative2 = [](const vector<double>&
     Y, const vector<double>& X, int pos, double x)
     -> double {
    if (pos >= Y.size() - 2) return 0;
    return (Y[pos + 1] - Y[pos]) / (X[pos + 1] -
       X[pos]) +
      ((Y[pos + 2] - Y[pos + 1]) / (X[pos + 2] -
         X[pos + 1]) - (Y[pos + 1] - Y[pos]) /
         (X[pos + 1] - X[pos]))
        / (X[pos + 2] - X[pos]) * (2 * x - X[pos] -
           X[pos + 1]);
  };
  auto secondDerivative2 = [](const vector<double>&
     Y, const vector<double>& X, int pos, double x)
     -> double {
    if (pos >= Y.size() - 2) return 0;
    return 2 * ((Y[pos + 2] - Y[pos + 1]) / (X[pos
       + 2] - X[pos + 1])
        - (Y[pos + 1] - Y[pos]) / (X[pos + 1] -
           X[pos]))
      / (X[pos + 2] - X[pos]);
  };

  if (argc != 2) {
    cerr << "Error: arg is incorrect" << endl;
    exit(-1);
  }

  string dataFile = argv[1];
  vector<double> X, Y;
  size_t N;
  double temp, perfectX;
  ifstream in(dataFile, ios::in);

  in >> N;
  for (size_t i = 0; i < N; ++i) {
    in >> temp;
    X.push_back(temp);
```

```
  }
  for (size_t i = 0; i < N; ++i) {
    in >> temp;
    Y.push_back(temp);
  }
  in >> perfectX;
  in.close();

  for (size_t i = 0; i < N - 1; ++i) {
    if (X[i] <= perfectX && perfectX <= X[i + 1]) {
      cout << "F'(x) = " << firstDerivative2(Y, X,
        i, perfectX) << endl;
      cout << "F\"(x) = " << secondDerivative2(Y,
        X, i, perfectX) << endl;
      break;
    }
  }
  return 0;
}
```

Тестирование

Входной файл

```
5
1.0 1.5 2.0 2.5 3.0
0.0 0.40547 0.69315 0.91629 1.0986
2.0
```

Выходной файл

```
F'(x) = 0.51082
F"(x) = -0.25816
```

Метод 5 — Численное интегрирование

Задание

Вычислить определенный интеграл $F = \int\limits_{X_0}^{X_1} ydx$, методами прямоугольников, трапеций, Симпсона с шашами h_1, h_2. Оценить погрешность вычислений, используя метод Рунге-Ромберга.

Вариант: 3
$y = \frac{x}{(3x+4)^3}, \quad X_0 = -1, X_k = 1, h_1 = 0.5, h_2 = 0.25$

Описание алгоритма

Формулы численного интегрирования используются в тех случаях, когда вычислить аналитически интеграл $F = \int\limits_a^b f(x)dx$ не удается. Отрезок $[a, b]$ разбивают точками $x_0, ..., x_N$ так, что $a = x_0 \leq x_1 \leq ... \leq x_N = b$ с достаточно мелким шагом $h_i = x_i - x_{i-1}$ и на одном или нескольких отрезках h_i подынтегральную функцию $f(x)$ заменяют такой приближающей $\varphi(x)$ так, что она, во–первых, близка $f(x)$, а, во–вторых, интеграл от $\varphi(x)$ легко вычисляется.
Заменим подынтегральную функцию интерполяционным многочленом Лагранжа нулевой степени, проходящим через середину отрезка — точку $\overline{x_i} = (x_{i-1} + x_i)/2$, получим формулу прямоугольников:

$$\int\limits_a^b f(x)dx \approx \sum_{i=1}^{N} h_i f(\frac{x_{i-1} + x_i}{2})$$

В случае таблично заданных функций удобно в качестве узлов интерполяции выбрать начало и конец отрезка интегрирования, т.е. заменить функцию $f(x)$ многочленом Лагранжа первой степени. В результате получим формулу трапеций:

$$\int\limits_a^b f(x)dx \approx \frac{1}{2} \sum_{i=1}^{N} (f_i + f_{i-1})h_i$$

Для повышения порядка точности формулы численного интегрирования заменим подынтегральную кривую параболой — интерполяционным многочленом второй степени, выбрав в качестве узлов интерполяции концы и середину отрезка интегрирования: $x_{i-1}, x_{i-\frac{1}{2}} = (x_{i-1} + x_i)/2, x_i$

Для случая $h_i = \frac{x_i - x_{i-1}}{2}$, получим формулу Симпсона (парабол):

$$\int_a^b f(x)dx \approx \frac{1}{3}\sum_{i=1}^{N}(f_{i-1} + 4f_{i-\frac{1}{2}} + f_i)h_i$$

В случае интегрирования с постоянным шагом формулы принимают следующий вид:

Метод прямоугольников

$$F = h[y(\frac{x_0 + x_1}{2}) + y(\frac{x_1 + x_2}{2}) + ... + y(\frac{x_{N-1} + x_N}{2})]$$

Метод Трапеций

$$F = h[\frac{y_0}{2} + y_1 + y_2 + ... + y_{N-1} + \frac{y_N}{2}]$$

Метод Симпсона

$$F = \frac{h}{3}[y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + ... + 2y_{N-2} + 4y_{N-1} + y_N]$$

Реализация

```
#include <iostream>
#include <fstream>
#include <functional>
#include <cmath>

using namespace std;

int main(int argc, char* argv[]) {
  if (argc != 2) {
    cerr << "Error: arg is incorrect" << endl;
    exit(-1);
  }

  string dataFile = argv[1];
  double h1, h2, X1, Xn;

  ifstream in(dataFile, ios::in);
  in >> X1 >> Xn >> h1 >> h2;
```

```
  in.close();

  auto Y = [](double x) -> double {
    return x / pow(3 * x + 4, 3.0);
  };

  auto secondDerivateY = [](double x) -> double {
    return 18 * (3 * x - 4) / pow(3 * x + 4, 5.0);
  };

  auto fourthDerivateY = [](double x) -> double {
    return 3240 * (3 * x - 8) / pow(3  * x + 4,
       7.0);
  };

  double M2_h1 = 0.0, M2_h2 = 0.0, M4_h1 = 0.0,
     M4_h2 = 0.0;

  for (double cur = X1; cur <= Xn; cur += h1) {
    M2_h1 = max(M2_h1, abs(secondDerivateY(cur)));
    M4_h1 = max(M4_h1, abs(fourthDerivateY(cur)));
  }
  for (double cur = X1; cur <= Xn; cur += h2) {
    M2_h2 = max(M2_h2, abs(secondDerivateY(cur)));
    M4_h2 = max(M4_h2, abs(fourthDerivateY(cur)));
  }

  auto rectangleMethod = [&Y](double startX, double
     endX, double step) -> double {
    double res = 0.0;
    for (double cur = startX + step; cur <= endX;
       cur += step) {
      res += Y(0.5 * (2 * cur - step));
    }
    return step * res;
  };

//  auto estimateRectangleMethod = [](double
   startX, double endX, double step, double M) ->
   double {
//    return step * step * M * (endX - startX) / 24;
```

```
//  };

  auto trapezoidalMethod = [&Y](double startX,
     double endX, double step) -> double {
    double res = 0.0;
    for (double cur = startX + step; cur <= endX;
       cur += step) {
      res += Y(cur) + Y(cur - step);
    }
    return 0.5 * step * res;
  };

//  auto estimateTrapezoidalMethod = [](double
   startX, double endX, double step, double M) ->
   double {
//    return step * step * M * (endX - startX) / 12;
//  };

  auto SimpsonMethod = [&Y](double startX, double
     endX, double step) -> double {
    double res = 0.0;
    res += Y(startX) + Y(endX);
    for (double cur = startX + step; cur < endX;
       cur += 2 * step) {
      res += 4 * Y(cur);
    }
    for (double cur = startX + 2 * step; cur <
       endX; cur += 2 * step) {
      res += 2 * Y(cur);
    }
    return step * res / 3;
  };

//  auto estimateSimpsonMethod = [](double startX,
   double endX, double step, double M) -> double {
//    return (endX - startX) * pow(step, 4.0) * M /
   180;
//  };

  double k = max(h1, h2) / min(h1, h2);
```

```
    cout << "Rectangle method (h1): " <<
        rectangleMethod(X1, Xn, h1) << endl;
    cout << "Rectangle method (h2): " <<
        rectangleMethod(X1, Xn, h2) << endl;
    cout << "estimate: ";
    if (h1 >= h2) cout << rectangleMethod(X1, Xn, h2)
        + (rectangleMethod(X1, Xn, h2) -
        rectangleMethod(X1, Xn, h1)) / (pow(k, 2.0) -
        1.0) << endl;
    else cout << rectangleMethod(X1, Xn, h1) +
        (rectangleMethod(X1, Xn, h1) -
        rectangleMethod(X1, Xn, h2)) / (pow(k, 2.0) -
        1.0) << endl;
    cout << "Trapezoidal method (h1): " <<
        trapezoidalMethod(X1, Xn, h1) << endl;
    cout << "Trapezoidal method (h2): " <<
        trapezoidalMethod(X1, Xn, h2) << endl;
    cout << "estimate: ";
    if (h1 >= h2) cout << trapezoidalMethod(X1, Xn,
        h2) + (trapezoidalMethod(X1, Xn, h2) -
        trapezoidalMethod(X1, Xn, h1)) / (pow(k, 2.0)
        - 1.0) << endl;
    else cout << trapezoidalMethod(X1, Xn, h1) +
        (trapezoidalMethod(X1, Xn, h1) -
        trapezoidalMethod(X1, Xn, h2)) / (pow(k, 2.0)
        - 1.0) << endl;
    cout << "Simpson method (h1): " <<
        SimpsonMethod(X1, Xn, h1) <<  endl;
    cout << "Simpson method (h2): " <<
        SimpsonMethod(X1, Xn, h2) <<  endl;
    cout << "estimate: ";
    if (h1 >= h2) cout << SimpsonMethod(X1, Xn, h2) +
        (SimpsonMethod(X1, Xn, h2) - SimpsonMethod(X1,
        Xn, h1)) / (pow(k, 4.0) - 1.0) << endl;
    else cout << SimpsonMethod(X1, Xn, h1) +
        (SimpsonMethod(X1, Xn, h1) - SimpsonMethod(X1,
        Xn, h2)) / (pow(k, 4.0) - 1.0) << endl;
    return 0;
}
```

Тестирование

Входной файл

```
-1 1 0.5 0.25
```

Выходной файл

```
Rectangle method (h1): -0.0709098
Rectangle method (h2): -0.102439
Runge-Romberg estimate: 0.112949
Trapezoidal method (h1): -0.263769
Trapezoidal method (h2): -0.167339
Runge-Romberg estimate: 0.135196
Simpson method (h1): -0.185511
Simpson method (h2): -0.135196
Runge-Romberg estimate: 0.131842
```