БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Задание № 3

Вариант 4

Выполнил: Жерело Владимир Анатольевич,

студент 2 курса 9 группы

Преподаватель: Будник Анатолий Михайлович

Минск, 2021

***Метод наименьших квадратов***

Постановка задачи:

[𝑎,𝑏]=[0.3,1.3]

.

Точки восстановления:

Методом наименьших квадратов построить аппроксимирующий многочлен степени m=n/2, полагая, что весовая функция в рассматриваемом пространстве равна 1. Вычислить приближенные значения функции f(x) в точках x\*,x\*\* и x\*\*\*. Оценить погрешность.

Алгоритм решения:

Элемент наилучшего приближения (ЭНП) будем искать в пространстве всех действительных квадратично интегрируемых по весу функций, т.е. в пространстве с нормой

ЭНП определим, как линейную комбинацию вида , где , а константы определяются из условия минимума нормы . При этом, учитывая, что функция задана таблично и , скалярное произведение можно вычислить по формуле:

Для определения необходимо решить следующую систему линейных алгебраических уравнений:

Погрешность приближения определяется среднеквадратичным отклонением, определяемым по формуле:

Листинг программы:

std::vector<double> get\_nodes() {

std::vector<double> x(N + 1);

for (int i = 0; i <= N; ++i) {

x[i] = Aj + (double)i / N;

}

return x;

}

double func(double x) {

return Aj \* exp(-x) + (1 - Aj) \* cos(x);

}

std::vector<std::vector<double>> build\_matrix(std::vector<double> fVals, std::vector<double> xVals) {

if (fVals.size() != N + 1 || xVals.size() != N + 1) {

throw new std::exception("fVals and xVals are wrong size");

}

std::vector<std::vector<double>> matr(M + 1, std::vector<double>(M + 2, 0));

for (int k = 0; k <= M; ++k) {

for (int i = 0; i <= M; ++i) {

for (int j = 0; j <= N; ++j) {

matr[k][i] += pow(xVals[j], i + k);

}

}

for (int j = 0; j <= N; ++j) {

matr[k][M + 1] += pow(xVals[j], k) \* fVals[j];

}

}

return matr;

}

/// <summary>

/// решает систему методом гаусса

/// </summary>

/// <param name="matr">расширенная матрица(со столбцом свободных членов)</param>

/// <returns></returns>

std::vector<double> solve\_Gauss(std::vector<std::vector<double>> matr) {

double cur;

for (int i = 0; i < matr.size(); ++i) {

cur = matr[i][i];

for (int j = i; j < matr[0].size(); ++j) {

matr[i][j] /= cur;

}

for (int j = i + 1; j < matr.size(); ++j) {

for (int k = i + 1; k < matr[0].size(); ++k) {

matr[j][k] -= matr[j][i] \* matr[i][k];

}

matr[j][i] = 0;

}

}

std::vector<double> sol(matr.size());

for (int j = matr.size() - 1; j >= 0; --j) {

double tmp = matr[j][matr[0].size() - 1];

for (int i = matr.size() - 1; i > j; --i) {

tmp -= matr[j][i] \* sol[i];

}

sol[j] = tmp;

}

return sol;

}

double polynomial(std::vector<double>& coefs, double arg) {

double res{ 0 };

for (int i = 0; i < coefs.size(); ++i) {

res += coefs[i] \* pow(arg, i);

}

return res;

}

void print(std::vector<double>& vec) {

for (int i = 0; i < vec.size(); ++i) {

std::cout << vec[i] << ' ';

}

}

double mean\_sqr\_error(std::vector<double>& coefs, std::vector<double>& fVals, std::vector<double>& xVals) {

double sum = 0;

for (int j = 0; j <= N; ++j) {

sum += (fVals[j] - polynomial(coefs, xVals[j])) \* (fVals[j] - polynomial(coefs, xVals[j]));

}

return sqrt(sum);

}

void print\_interpolation\_data(std::vector<double>& x, std::vector<double>& f) {

std::cout << "interpolation nodes\n";

print(x);

std::cout << std::endl;

std::cout << "interpolation function values\n";

print(f);

}

void min\_sqr\_method(std::vector<double>& x, std::vector<double>& f) {

print\_interpolation\_data(x, f);

std::cout << std::endl;

std::vector<double> coefs = solve\_Gauss(build\_matrix(f, x));

std::cout << "coefficients: ";

print(coefs);

std::cout << '\n';

double x1 = x[0] + 2. / (3 \* N);

double x2 = x[N / 2] + 1. / (2 \* N);

double x3 = x[N] - 1. / (3 \* N);

std::cout << "x\* = " << x1 << '\n';

std::cout << "x\*\* = " << x2 << '\n';

std::cout << "x\*\*\* = " << x3 << '\n';

std::cout << "Polynomial values: \n";

std::cout << "P(x\*) = " << polynomial(coefs, x1) << '\n';

std::cout << "P(x\*\*) = " << polynomial(coefs, x2) << '\n';

std::cout << "P(x\*\*\*) = " << polynomial(coefs, x3) << '\n';

std::cout << "true residuals: \n";

std::cout << abs(polynomial(coefs, x1) - func(x1)) << '\n';

std::cout << abs(polynomial(coefs, x2) - func(x2)) << '\n';

std::cout << abs(polynomial(coefs, x3) - func(x3)) << '\n';

std::cout << "mean sqr error: " << mean\_sqr\_error(coefs, f, x);

}

Результаты:

interpolation nodes

0.3 0.4 0.5 0.6 0.7 0.8 0.9 1 1.1 1.2 1.3

interpolation function values

0.890981 0.845839 0.796267 0.742378 0.684365 0.622493 0.557098 0.488575 0.417379 0.344009 0.269009

coefficients: 1.00007 -0.300663 -0.19745 -0.0549626 0.046861 -0.00527674

x\* = 0.366667

x\*\* = 0.85

x\*\*\* = 1.26667

Polynomial values:

P(x\*) = 0.861381

P(x\*\*) = 0.590213

P(x\*\*\*) = 0.294155

true residuals:

3.46691e-07

1.65904e-07

2.49988e-07

mean sqr error: 5.58176e-07

Погрешности имеют -7 порядок, значит, что с помощью полученного аппроксимирующего многочлена Ф(x) можно рассчитать значения функции на отрезке [0.3,1.3] с точностью до 6 знака включительно.

***Интерполяционный многочлен Лагранжа***

Постановка задачи:

[𝑎,𝑏]=[0.3,1.3]

.

Точки восстановления:

По заданной таблице значений функции 𝑓(𝑥) построить интерполяционный многочлен Лагранжа. Вычислить приближенные значения функции в точках 𝑥∗, 𝑥∗∗ и 𝑥∗∗∗. Оценить погрешность.

Алгоритм решения:

В случае алгебраического интерполирования многочлен , построенный из условий , определяется однозначно.

Для приближения заданной функции используем интерполяционный многочлен в форме Лагранжа: .

Погрешность приближения определяется остатком интерполирования , определяемым по формуле:

Листинг программы:

double lagrangia\_polynomial(std::vector<double>& xVals, std::vector<double>& fVals, double arg) {

if (xVals.size() != fVals.size()) {

throw new std::exception("sizes of xVals and fVals must match");

}

double res = 0;

double l;

for (int i = 0; i < fVals.size(); ++i) {

l = 1;

for (int j = 0; j < i; ++j) {

l \*= (arg - xVals[j]) / (xVals[i] - xVals[j]);

}

for (int j = i + 1; j < xVals.size(); ++j) {

l \*= (arg - xVals[j]) / (xVals[i] - xVals[j]);

}

res += l \* fVals[i];

}

return res;

}

void lagrangia\_method(std::vector<double>& x, std::vector<double>& f) {

print\_interpolation\_data(x, f);

std::cout << std::endl;

double x1 = x[0] + 2. / (3 \* N);

double x2 = x[N / 2] + 1. / (2 \* N);

double x3 = x[N] - 1. / (3 \* N);

std::cout << "x\* = " << x1 << '\n';

std::cout << "x\*\* = " << x2 << '\n';

std::cout << "x\*\*\* = " << x3 << '\n';

std::cout << "Polynomial values: \n";

std::cout << "P(x\*) = " << lagrangia\_polynomial(x, f, x1) << '\n';

std::cout << "P(x\*\*) = " << lagrangia\_polynomial(x, f, x2) << '\n';

std::cout << "P(x\*\*\*) = " << lagrangia\_polynomial(x, f, x3) << '\n';

std::cout << "true residuals: \n";

std::cout << abs(lagrangia\_polynomial(x, f, x1) - func(x1)) << '\n';

std::cout << abs(lagrangia\_polynomial(x, f, x2) - func(x2)) << '\n';

std::cout << abs(lagrangia\_polynomial(x, f, x3) - func(x3)) << '\n';

double rx1 = DER\_11\_MAX;

double rx2 = DER\_11\_MAX;

double rx3 = DER\_11\_MAX;

for (int i = 0; i <= N; ++i) {

rx1 \*= (x1 - x[i]) / ((double)i + 1);

rx2 \*= (x2 - x[i]) / ((double)i + 1);

rx3 \*= (x3 - x[i]) / ((double)i + 1);

}

std::cout << "interpolation remainder in x\*: " << rx1 << '\n';

std::cout << "interpolation remainder in x\*\*: " << rx2 << '\n';

std::cout << "interpolation remainder in x\*\*\*: " << rx3 << '\n';

}

Результаты:

interpolation nodes

0.3 0.4 0.5 0.6 0.7 0.8 0.9 1 1.1 1.2 1.3

interpolation function values

0.890981 0.845839 0.796267 0.742378 0.684365 0.622493 0.557098 0.488575 0.417379 0.344009 0.269009

x\* = 0.366667

x\*\* = 0.85

x\*\*\* = 1.26667

Polynomial values:

P(x\*) = 0.861381

P(x\*\*) = 0.590213

P(x\*\*\*) = 0.294155

true residuals:

1.65423e-14

3.33067e-16

4.08562e-14

interpolation remainder in x\*: 2.84373e-14

interpolation remainder in x\*\*: -7.08961e-16

interpolation remainder in x\*\*\*: -6.0611e-14

Получили достаточно высокую точность приближенного вычисления f(x\*), f(x\*\*), f(x\*\*\*), что подтверждается вычисленными остатками интерполирования и истинными погрешностями

***Интерполяционный многочлен Ньютона***

Постановка задачи:

[𝑎,𝑏]=[0.3,1.3]

.

Точки восстановления:

По заданной таблице значений функции 𝑓(𝑥) построить интерполяционный многочлен Ньютона. Вычислить приближенные значения функции в точках 𝑥∗, 𝑥∗∗ и 𝑥∗∗∗. Оценить погрешность.

Алгоритм решения:

В случае алгебраического интерполирования многочлен , построенный из условий определяется однозначно.

Для приближения заданной функции используем интерполяционный многочлен в форме Ньютона: .

Погрешность приближения определяется остатком интерполирования 𝑟𝑛(𝑥)=𝑓(𝑥)−𝑃𝑛(𝑥), определяемым по формуле: 𝑟𝑛(𝑥)=𝜔(𝑥)𝑓(𝑥,𝑥0,𝑥1,…,𝑥𝑛), где разделенную разность -го порядка можно вычислить, пользуясь полученным приближенным значением функции в точке 𝑥.

Листинг программы:

double newton\_polynomial(std::vector<double> rr, std::vector<double> x, double arg) {

double tmp = 1;

double result = rr[0];

for (int i = 0; i < rr.size() - 1; ++i) {

tmp \*= (arg - x[i]);

result += tmp \* rr[i + 1];

}

return result;

}

void newton\_method(std::vector<double>& x, std::vector<double>& f) {

print\_interpolation\_data(x, f);

std::cout << std::endl;

double x1 = x[0] + 2. / (3 \* N);

double x2 = x[N / 2] + 1. / (2 \* N);

double x3 = x[N] - 1. / (3 \* N);

std::cout << "x\* = " << x1 << '\n';

std::cout << "x\*\* = " << x2 << '\n';

std::cout << "x\*\*\* = " << x3 << '\n';

std::vector<std::vector<double>> rr(N + 1, std::vector<double>(N + 1));

for (int i = 0; i < rr.size(); ++i) {

rr[i][0] = f[i];

}

for (int j = 1; j < rr[0].size(); ++j) {

for (int i = 0; i < rr.size() - j; ++i) {

rr[i][j] = (rr[i + 1][j - 1] - rr[i][j - 1]) / (x[i + j] - x[i]);

}

}

std::cout << "Polynomial values: \n";

std::cout << "P(x\*) = " << newton\_polynomial(rr[0], x, x1) << '\n';

std::cout << "P(x\*\*) = " << newton\_polynomial(rr[0], x, x2) << '\n';

std::cout << "P(x\*\*\*) = " << newton\_polynomial(rr[0], x, x3) << '\n';

std::cout << "true residuals: \n";

std::cout << abs(newton\_polynomial(rr[0], x, x1) - func(x1)) << '\n';

std::cout << abs(newton\_polynomial(rr[0], x, x2) - func(x2)) << '\n';

std::cout << abs(newton\_polynomial(rr[0], x, x3) - func(x3)) << '\n';

double w1 = 1;

double w2 = 1;

double w3 = 1;

for (int i = 0; i <= N; ++i) {

w1 \*= (x1 - x[i]);

w2 \*= (x2 - x[i]);

w3 \*= (x3 - x[i]);

}

std::vector<double> rr1(N + 1);

rr1[0] = func(x1);

std::vector<double> rr2(N + 1);

rr2[0] = func(x2);

std::vector<double> rr3(N + 1);

rr3[0] = func(x3);

for (int i = 1; i < N + 1; ++i) {

rr1[i] = (rr[0][i - 1] - rr1[i - 1]) / (x[i - 1] - x1);

rr2[i] = (rr[0][i - 1] - rr2[i - 1]) / (x[i - 1] - x2);

rr3[i] = (rr[0][i - 1] - rr3[i - 1]) / (x[i - 1] - x3);

}

std::cout << "interpolation remainder in x\*: " << w1 \* rr1[N] << '\n';

std::cout << "interpolation remainder in x\*\*: " << w2 \* rr2[N] << '\n';

std::cout << "interpolation remainder in x\*\*\*: " << w3 \* rr3[N] << '\n';

}

Результаты:

interpolation nodes

0.3 0.4 0.5 0.6 0.7 0.8 0.9 1 1.1 1.2 1.3

interpolation function values

0.890981 0.845839 0.796267 0.742378 0.684365 0.622493 0.557098 0.488575 0.417379 0.344009 0.269009

x\* = 0.366667

x\*\* = 0.85

x\*\*\* = 1.26667

Polynomial values:

P(x\*) = 0.861381

P(x\*\*) = 0.590213

P(x\*\*\*) = 0.294155

true residuals:

1.63203e-14

5.55112e-16

4.04676e-14

interpolation remainder in x\*: -2.01018e-13

interpolation remainder in x\*\*: 4.8203e-15

interpolation remainder in x\*\*\*: 3.97389e-13

Получили достаточно высокую точность приближенного вычисления f(x\*), f(x\*\*), f(x\*\*\*), что подтверждается вычисленными остатками интерполирования и истинными погрешностями

***Минимизация остатка интерполирования***

Алгоритм решения:

Для того чтобы минимизировать остаток интерполирования необходимо минимизировать многочлен . Таким многочленом, наименее отклоняющимся от 0, является многочлен Чебышева . Тогда новые узлы интерполирования есть корни этого многочлена:

По новой таблице построим интерполяционный многочлен Лагранжа. При это для остатка интерполирования справедлива оценка:

Листинг программы:

std::vector<double> min\_nodeset() {

std::vector<double> x(N + 1);

for (int k = 0; k <= N; ++k) {

double a = (double)(1 + 2 \* Aj) / 2;

double b = cos((2 \* k + 1) \* M\_PI / (2 \* (N + 1)));

x[N - k] = a + b / 2;

}

return x;

}

void minimization(std::vector<double>& x, std::vector<double>& f) {

double x1 = x[0] + 2. / (3 \* N);

double x2 = x[N / 2] + 1. / (2 \* N);

double x3 = x[N] - 1. / (3 \* N);

std::cout << "x\* = " << x1 << '\n';

std::cout << "x\*\* = " << x2 << '\n';

std::cout << "x\*\*\* = " << x3 << '\n';

std::vector<double> newX = min\_nodeset();

std::vector<double> newF(newX.size());

std::cout << "Values in these nodes:\n";

for (int i = 0; i < newX.size(); ++i) {

newF[i] = func(newX[i]);

}

print\_interpolation\_data(newX, newF);

std::cout << '\n';

std::cout << "Polynomial values: \n";

std::cout << "P(x\*) = " << lagrangia\_polynomial(newX, newF, x1) << '\n';

std::cout << "P(x\*\*) = " << lagrangia\_polynomial(newX, newF, x2) << '\n';

std::cout << "P(x\*\*\*) = " << lagrangia\_polynomial(newX, newF, x3) << '\n';

std::cout << "true residuals: \n";

std::cout << abs(lagrangia\_polynomial(newX, newF, x1) - func(x1)) << '\n';

std::cout << abs(lagrangia\_polynomial(newX, newF, x2) - func(x2)) << '\n';

std::cout << abs(lagrangia\_polynomial(newX, newF, x3) - func(x3)) << '\n';

double rx1 = 0.59 / pow(2, 2 \* N + 1);

for (int i = 0; i <= N; ++i) {

rx1 \*= 1. / ((double)i + 1);

}

std::cout << "interpolation remainder: " << rx1 << '\n';

}

Результаты:

x\* = 0.366667

x\*\* = 0.85

x\*\*\* = 1.26667

interpolation nodes

0.305089 0.345184 0.422125 0.52968 0.659134 0.8 0.940866 1.07032 1.17787 1.25482 1.29491

interpolation function values

0.888791 0.871136 0.83525 0.780716 0.708556 0.622493 0.529449 0.43876 0.360402 0.303062 0.272856

Polynomial values:

P(x\*) = 0.861381

P(x\*\*) = 0.590213

P(x\*\*\*) = 0.294155

true residuals:

3.77476e-15

4.10783e-15

2.88658e-15

interpolation remainder: 7.04801e-15

Получили более высокую точность приближения по сравнению с методами Лагранжа и Ньютона. Причем оценка остатка интерполирования верна для любого х из отрезка(в отличие от предыдущих методов, где она зависит от конкретной точки)

***Интерполирования по равноотстоящим узлам в начале таблицы***

Постановка задачи:

Используя равномерную сетку узлов, при степени многочлена k=3 добиться точности

Алгоритм решения:

В данном случае остаток интерполирования имеет формулу:

Тогда найдем такое h, чтобы =>

Зададим новые узлы интерполирования с учетом, что :

Формула интерполяционного многочлена:

Где – конечные разности соответствующих порядков

Листинг программы:

double beg\_interpolation\_polynomial(std::vector<double> fin\_sub, double t) {

double result = fin\_sub[0];

double tmp = 1;

for (int i = 0; i < fin\_sub.size() - 1; ++i) {

tmp \*= (t - i) / (i + 1);

result += fin\_sub[i + 1] \* tmp;

}

return result;

}

void interpolation\_in\_beginning(std::vector<double>& x, std::vector<double>& f) {

double x1 = x[0] + 2. / (3 \* N);

//новое расстояние между узлами(чтобы достигнуть точности 10^-7)

const double NEW\_STEP = 0.04;

std::vector<double> newX(4);

std::vector<double> newF(4);

newX[0] = x1 - NEW\_STEP / 10;

newF[0] = func(newX[0]);

for (int i = 1; i < newX.size(); ++i) {

newX[i] = newX[i - 1] + NEW\_STEP;

newF[i] = func(newX[i]);

}

print\_interpolation\_data(newX, newF);

std::cout << "x\* = " << x1 << '\n';

std::cout << '\n';

// таблица конечных разностей

std::vector<std::vector<double>> fin\_sub(4, std::vector<double>(4));

for (int i = 0; i < fin\_sub.size(); ++i) {

fin\_sub[i][0] = newF[i];

}

for (int j = 1; j < fin\_sub[0].size(); ++j) {

for (int i = 0; i < fin\_sub.size() - j; ++i) {

fin\_sub[i][j] = fin\_sub[i + 1][j - 1] - fin\_sub[i][j - 1];

}

}

std::cout << "P(x\*) = " << beg\_interpolation\_polynomial(fin\_sub[0], (x1 - newX[0]) / NEW\_STEP) << '\n';

std::cout << "true residual: ";

std::cout << abs(beg\_interpolation\_polynomial(fin\_sub[0], (x1 - newX[0]) / NEW\_STEP) - func(x1)) << '\n';

}

Результаты:

interpolation nodes

0.362667 0.402667 0.442667 0.482667

interpolation function values

0.863213 0.844574 0.825225 0.805173

x\* = 0.366667

P(x\*) = 0.861382

true residual: 4.44485e-08

Исходя из выбранного h, остаток интерполирования меньше . По истинной погрешности также видно, что мы достигли нужной точности приближая исходную функцию полиномом третей степени