ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»

Факультет безопасности информационных технологий
Кафедра проектирования и безопасности компьютерных систем

Дисциплина:
«Вычислительная математика»

ОТЧЕТ О ВЫПОЛНЕНИИ ДОМАШНИХ ЗАДАНИЙ

Выполнил:
Студент группы N3247
Гаврилова Вероника Викторовна

Проверил:
Гришенцев Алексей Юрьевич

Санкт-Петербург
2021г.

Оглавление

1. Приближенные числа и источники погрешностей

1.1 Абсолютная и относительные погрешности

Задание:
Провести многократные независимые измерения одной величины в ходе эксперимента и рассчитать абсолютную и относительную погрешности полученных измерений, среднее арифметическое и среднее геометрическое полученной величины.

Цель:
Реализация программы, рассчитывающей погрешности независимых измерений величины средствами языка C++.

Теория:
Абсолютной погрешностью числа a называют величину равную разнице между этим числом и его точным значением

$$\Delta = |A - a|$$

Абсолютная погрешность при неизвестном $X_{\text{истинн.}}$: $\Delta X = S_{\overline{X}} * t$, где t – коэффициент Стьюдента, а $S_{\overline{X}}$ - среднеквадратичная ошибка среднего арифметического.

Среднеквадратичная ошибка среднего арифметического: $S_{\overline{X}} = \sqrt{\frac{\sum_{i=1}^{n}(\overline{X}-X_i)\text{^}2}{n(n-1)}}$

Относительной погрешностью приближенного числа называется отношение абсолютной погрешности числа к самому числу:

$$\delta = \frac{\delta x}{x}$$

Среднее арифметическое: $(\sum_{i=1}^{n} X_i)/n$
Среднее геометрическое: $\sqrt[n]{\prod_{i=1}^{n} X_i}$
Соотношение между средним арифметическим и средним геометрическим
$\frac{a+b}{2} \geq \sqrt{ab}$

Программа:

```
#include <bits/stdc++.h>

using namespace std;

int main() {
string s1, s2;
double a, da, b, db;
string tmp1, tmp2, tmp3, tmp4;
cin >> tmp1 >> s1 >> tmp2 >> s2 >> tmp3 >> s1 >> tmp4;

int a1 = tmp1.size() - tmp1.find(".") - 1;
int a3 = tmp3.size() - tmp3.find(".") - 1;
int a2 = tmp2.size() - tmp2.find(".") - 1;
int a4 = tmp4.size() - tmp4.find(".") - 1;
int k1 = max(a1, a3);
```

```
int k2 = max(a2, a4);
int k = max(k1, k2);
a = stod(tmp1);
da = stod(tmp2);
b = stod(tmp3);
db = stod(tmp4);
double rez, drez;
if(s2 == "+") {
rez = a + b;
drez = da + db;
cout.setf(std::ios::fixed);
cout.precision(k);
cout << rez << " +- ";
cout.precision(k);
cout « drez;

}
if(s2 == "-") {
rez = a - b;
drez = da + db;
cout.setf(std::ios::fixed);
cout.precision(k);
cout << rez << " +- ";
cout.precision(k);
cout « drez;
}
if(s2 == "*") {
rez = a * b;
drez = (da/a + db/b)*rez;
cout.setf(std::ios::fixed);
cout.precision(k);
cout << rez << " +- ";
cout.precision(k);
cout << drez;
}
if(s2 == "/") {
rez = a / b;
drez = (da/a + db/b)*rez;
cout.setf(std::ios::fixed);
cout.precision(k);
cout << rez << " +- ";
cout.precision(k);
cout << drez;
}
return 0;
}
```

Пример вывода:

```
5.0 +- 0.1 * 3.0 +- 0.001
15.000 +- 0.305
```

```
2.2 +- 0.0 + 3.8 +- 0.0
6.0 +- 0.0

7.0 +- 0.2 / 3.5 +- 0.4
2.0 +- 0.3
```

Вывод:
Данная программа может быть использована для вычисления погрешностей в ходе какого-либо эксперимента.

2. Вычисление функций

2.1 Схема Горнера

Задание:
Разработать программу на языке C++ реализующую схему на языке C++ реализующую схему Горнера для деления многочленов на двучлены и вычисляющую коэффициенты многочлена-частного и остаток. Результат выводить в виде многочлена-частного и остатка.

Цель:
Научиться производить деление многочленов на двучлены схемой Горнера и реализовать данный алгоритм средствами языка C++.

Теория:
Пусть в данном полиноме $P(x) = a_0x^n + a_1x^{n-1} + \ldots + a_n$ (1) по каким-либо соображениям требуется произвести замену переменной x по формуле $x = y + \xi$, где ξ – фиксированное число и y – новая переменная.
Подставив второе выражение в данный полином и произведя указанные действие, после приведения подобных членов получим новый полином относительно переменной y:

$$P(y + \xi) = A_0y^n + A_1y^{n-1} + \ldots + A_n. \quad (2)$$

Так как полученный полином можно рассматривать как разложение Тейлора по степеням y функции $P(y + \xi)$, то коэффициенты A_i (i = 0, 1, …, n) могут быть вычислены по формуле:

$$A_i = \frac{p^{(n-i)}(\xi)}{(n - i)!} \quad (i = 0, 1, \ldots, n). \quad (3)$$

Рассмотрим более удобный на практике способ нахождения коэффициентов A_i $(i = 0, 1, \ldots, n)$ с помощью схемы Горнера. Положим сначала y=0 в полученном полиноме (3). Тогда будем иметь $A_n = P(\xi)$.
Разделив полином (1) на двучлен x- ξ, получим:

$$P(x) = (x - \xi)P_1(x) + P(\xi), \quad (4)$$

$$\text{где } P_1(x) = b_0x^{n-1} + b_1x^{n-2} + \cdots + b_{n-1}.$$

Далее, если в выражение (3) вместо y подставить его значение y = x - ξ, то получим:

$$P(x) = (x - \xi)[A_0(x - \xi)^{n-1} + A_1(x - \xi)^{n-2} + \ldots + A_{n-1}] + P(\xi). \quad (5)$$

Сопоставляя формулы (4) и (5), заключаем, что:

$$P_1(x) = A_0(x - \xi)^{n-1} + A_1(x - \xi)^{n-2} + \ldots + A_{n-1}. \quad (6)$$

Отсюда $A_{n-1} = P_1(\xi)$. (7)
Аналогично, разделив полином $P_1(x)$ на двучлен $x - \xi$, можем положить:

$$P_1(x) = (x - \xi)P_2(x) + P_1(\xi), \quad (8)$$

$$\text{где } P_2(x) = c_0x^{n-2} + c_1x^{n-3} + \cdots + c_{n-2}.$$

Из формул (6) и (7) имеем:
$P_1(x) = (x - \xi)[A_0(x - \xi)^{n-2} + A_1(x - \xi)^{n-3} + \ldots + A_{n-2}] + P_1(\xi)$. (9)

Сопоставляя формулы (8) и (9), заключаем, что:

$$P_2(x) = A_0(x - \xi)^{n-2} + A_1(x - \xi)^{n-3} + \ldots + A_{n-2}.$$

Отсюда $A_{n-2} = P_2(\xi)$.
Продолжая этот процесс, мы последовательно выразим все коэффициенты A_i $(i = 0, 1, \ldots, n)$ через значения соответствующих полиномов $P_0 = P(x)$ и $P_1(x), \ldots, P_n(x) = a_0$ при $x = \xi$:

$$A_n = P(\xi);$$
$$A_{n-1} = P_1(\xi);$$
$$A_{n-2} = P_2(\xi);$$
$$\ldots$$
$$A_0 = P_n(\xi),$$

Где полиномы $P_{k+1}(x)$, исходя из полиномов $P_k(x)$, строятся по формуле:

$P_k(x) = (x - \xi)P_{k+1}(x) + P_k(\xi)$, где $k = 0, 1, \ldots, n$.

Для вычислений значений $P(\xi)$, $P_1(\xi)$, $P_2(\xi)$... пользуются обобщенной схемой Горнера:

$$\begin{array}{l} a_0\ a_1\ a_2\ a_3 \ldots a_{n-1}\ a_n \qquad |\xi\_\_ \\ \underline{\quad b_0\xi\ b_1\xi\ b_2\xi \ldots\ b_{n-1}\xi\quad} \\ b_0\ b_1\ b_2\ b_3 \ldots b_{n-1}\ b_n = P(\xi) \end{array}$$

$$\begin{array}{c} c_0\xi\ c_1\xi\ c_2\xi \ldots\ c_{n-2}\xi \\ \hline c_0\ c_1\ c_2\ c_3 \ldots c_{n-1} = P_1(\xi) \end{array}$$
$$\ldots\ldots\ldots$$

Программа:

```
#include <stdlib.h>
#include <stdio.h>
int main(){
      printf(" Введите степень многочлена: ");
      int n, e;
      scanf("%d", &n);
      int* a = (int*) calloc(n+1, sizeof(int));
      int* b = (int*) calloc(n, sizeof(int));
      printf(" Введите коэффициенты многочлена (по
      возрастанию степени): ");
      for (int i = 0; i < (n+1); i++){
      scanf("%d", &a[i]);
      }
      printf(" Введите  свободный член делящего двучлена: ");
      scanf("%d", &e);
      b[n-1] = a[n];
printf("Результат деления: ");
for (int i = n - 1; i > 0; i--){
b[i-1] = b[i]*e + a[i];
printf("%d*x^%d ", b[i], i);
```

```
if (i>1) printf("+ ");
}
printf(";  Остаток: %d\n", b[0]);
int* k = (int*) calloc(n, sizeof(int));
k[n-1] = b[n-1];
for (int i = n - 2; i > 0; i--){
k[i] = b[i-1] - e*b[i];
if (k[i]!=a[i]){
free(k);
free(a);
free(b);
printf("Проверка не пройдена \n");
return 0;
}
}
free(k);
free(a);
free(b);
printf("Проверка пройдена \n");
return 0;
}
```

Вывод программы:

```
Введите степень многочлена: 4
Введите коэффициенты многочлена (по возрастанию степени): 23 45 3 5 6
Введите свободный член делящего двучлена: 7
Результат деления: 6*x^3 + 47*x^2 + 332*x^1 ;  Остаток: 2369
Проверка пройдена
```

Выводы:

Мы ознакомились с обобщённой схемой Горнера, а также реализовали программу, осуществляющую деление многочленов на двучлены и вычисляющую коэффициенты многочлена-частного и остаток.

2.2 Ряд Тейлора и Маклорена

Задание:

Разработать и написать программу для вычисления логарифмической и гармонической функции с помощью рядов Тейлора (Маклорена), с возможностью выбора числа членов рядов вычисления функций, точность вычисления определить с помощью расчета остаточного члена. Осуществить проверку вычислений: путем вычисления значений функций в точке и сравнения ошибки вычисления и величины остаточного члена. Построить график зависимости ошибки, точного значения и остаточного члена в зависимости от числа членов ряда.

Цель:

Ознакомиться с методами вычисления логарифмической и гармонической функций с помощью рядов Тейлора и Маклорена.

Теория:

Пусть дана действительная функция f(x). Данная функция называется аналитической в точке ξ, если в некоторой окрестности | x - ξ | $< R$ этой точки разлагается в степенной ряд.

Теорема Тейлора: Функция f(x) дифференцируемая n+1 раз в некотором интервале, содержащем точку ξ, может быть представлена в виде суммы многочлена n-ой степени и остаточного члена $R_n(x)$.

Ряд Тейлора:

$$f(x) = f(\xi) + f'(\xi)(x-\xi) + \frac{f''(\xi)}{2!}(x-\xi)^2 + \cdots + \frac{f^{(n)}(\xi)}{n!}(x-\xi)^n + \cdots$$

Полином Тейлора:

$$P_n(x) = \sum_{k=0}^{n} \frac{f^{(k)}(\xi)}{k!}(x-\xi)^k$$

Остаточным членом называется разность $R_n(x) = f(x) - \sum_{k=0}^{n} \frac{f^{(k)}(\xi)}{k!}(x-\xi)^k$, которая так же является ошибкой при замене функции f(x) полиномом Тейлора.
При неизвестном значении f(x) используют формулы вычисления остаточного члена, не содержащие f(x):

$$R_n(x) = \frac{f^{(n+1)}\big(\xi + \theta(x-\xi)\big)}{(n+1)!}(x-\xi)^{n+1}, 0 < \theta < 1$$

При ξ = 0 получаем ряд Маклорена:

$$f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \cdots + \frac{f^{(n)}(0)}{n!}x^n + \cdots$$

Примеры разложения функций:

Экспоненциальная функция: $e^x = 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!}$

Остаточный член: $R_n(x) = \frac{e^{\theta x}}{(n+1)!}x^{n+1}, 0 < \theta < 1$

Логарифмическая функция $0 < (1 + x) \leq 2$:

$$\ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots + (-1)^{n-1}\frac{x^n}{n} + \cdots$$

Логарифмическая функция $0 < (\frac{1-x}{1+x}) < \infty$:

Остаточный член:

$$R_n = 2(\frac{\xi^{2n+1}}{2n+1} + \frac{\xi^{2n+3}}{2n+3} + \frac{\xi^{2n+5}}{2n+5} + \ldots)$$

Разложение гармонической функции $0 \leq x \leq \frac{\pi}{4}$:

$$sin\ x\ = \sum_{n=0}^{\infty}(-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

При $\frac{\pi}{4} \leq x \leq \frac{\pi}{2}$:

$$sin\ x = \ cos\ z\ = \sum_{n=0}^{\infty}(-1)^n \frac{z^{2n}}{(2n)!}, \qquad z = \frac{\pi}{2} - \ x\ , 0 \leq z \leq \frac{\pi}{4}$$

Остаточный член:

$$|R_n| \leq \frac{x^{2n+1}}{(2n+1)!}$$

$$X^{(k+1)} = \beta + \alpha X^{(k)}$$

Программа:

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

using namespace std;

double* r_ln(double x, int n){
	double* pack = new double[3];
	pack[0] = 0; // result
	for (int i = 1; i <= n; i++) pack[0] += pow(-1, i + 1) * pow(x,
i) / i;
	double q = 0.9999;
	pack[1] = pow(-1, n + 1) * (1 + x) / (n * (n + 1) * pow(q, n));
// tail part
	pack[2] = pack[0] - log(1 + x); // mistake
	return pack;
}
double factor(int n){
	double k = 1.0;
	for (int i = 1; i <= n ; i++) k *= i;
	return k;
}
double* r_sin(double x, int n){
	double* pack = new double[3];
```

```
	pack[0] = 0;
	for (int i = 0; i <= n; i++) pack[0] += pow(-1, i) * pow(x, 2*i + 
1) / factor(2*i + 1);
	pack[1] = pow(x, 2*n + 1) / factor(2*n + 1);
		pack[2] = pack[0] - sin(x);
	return pack;
}

double* r_cos(double x, int n){
	double* pack = new double[3];
	pack[0] = 0;
	for (int i = 0; i <= n; i++) pack[0] += pow(-1, i) * pow(x, 
2*i) / factor(2*i);
	pack[1] = pow(x, 2*n) / factor(2*n);
	pack[2] = pack[0] - sin(x);
	return pack;
}
int main(){
	while (1) {
		cout << "[*]Режим работы программы: \n";
		cout << "[1] – Демонстрация\n";
		cout << "[2] - Вывод в файл зависимостей точных значений, 
остаточных членов и ошибок вычисления от числа членов ряда \n";
		cout << "[3] – Выход\n";
		int regime;
		cin >> regime;
		if (regime == 1){
			cout << "[*]Для выбора введите число и нажмите Enter: 
\n";
			cout << "[1] - Логарифмическая функция\n”;
			cout << "[2] - Гармоническая функция\n";
			int var;
			cin >> var;
			if (var == 1){
				double x;
				int n;
				cout << "[*]Введите значение x: ";
				cin >> x;
				if (x <= 1.0 && x > -1.0){
					cout << "[*]Введите значение n: ";
					cin >> n;
					double* res = r_ln(x, n);
					cout << "--Значение функции в заданной 
точке: " << res[0] << "\n";
					cout << "--Значение остаточного члена: " << 
res[1] << "\n";
					cout << "--Значение ошибки вычисления: " << 
res[2] << "\n";
				}
				else cout << "[*]Значение x для разложения 
ln(1+x) в ряд Маклорена должно быть в полуинтервале (-1.0, 1.0]\n";
```

```
                }
                if (var == 2){
                      double x;
                      int n;
                      cout << "[*]Введите значение x: ";
                      cin >> x;
                      if (x >= 0 && x <= M_PI/4){
                            cout << "[*]Введите значение n: ";
                            cin >> n;
                            double* res = r_sin(x, n);
                            cout << "--Значение функции в заданной
точке: " << res[0] << "\n";
                            cout << "--Абсолютное значение остаточного
члена <=  " << res[1] << "\n";
                            cout << "--Значение ошибки вычисления: " <<
res[2] << "\n";
                      }
                      else if (x >= M_PI/4 && x <= M_PI/2){
                            cout << "[*]Введите значение n: ";
                            cin >> n;
                            double* res = r_sin(M_PI - x, n);
                            cout << "--Значение функции в заданной
точке: " << res[0] << "\n";
                            cout << "--Абсолютное значение остаточного
члена <=  " << res[1] << "\n";
                            cout << "--Значение ошибки вычисления: " <<
res[2] << "\n";
                      }
                      else cout << "[*]Значение x для разложения Sin(x)
в ряд Маклорена должно находится в отрезке [0, pi/2]\n";
                }
          }
if (regime == 2){
                ofstream fout;
                double x1, x2;
                int Nmax;
                cout << "[*]Введите максимальное кол-во членов для
разложения в ряды Маклорена логарифмической и гармонической функций:
";
                cin >> Nmax;
                fout.open("logsLN.txt");
                if (!fout.is_open()) {
                      cout << "[*]Ошибка при открытии файла для записи.
Файл logsLN.txt не существует или поврежден, создаю новый файл.\n";
                      ofstream fout("logsLN.txt");
                }
                cout << "[*]Введите значение x1 (для ln(1+x1)): ";
                cin >> x1;
                if (x1 <= 1.0 && x1 > -1.0){
                      fout << "n,точное значение,остаточный
член,ошибка\n";
```

```
                    for (int n = 1; n <= Nmax; n++){
                         double* res = r_ln(x1, n);
                         fout << n << "," << res[0] << "," << res[1]
<< "," << res[2] <<  "\n";
                    }
               }
               else cout << "[*]Значение x для разложения ln(1+x) в
ряд Маклорена должно быть в полуинтервале (-1.0, 1.0]\n";
               fout.close();
               fout.open("logsSIN.txt");
               if (!fout.is_open()){
                         cout << "[*]Ошибка при открытии файла для
записи. Файл logsSIN.txt не существует или поврежден, создаю новый
файл.\n";
                    ofstream fout("logsSIN.txt");
               }
               cout << "[*]Введите значение x2 (для Sin(x2)): ";
               cin >> x2;
               fout << "n,точное значение,остаточный член,ошибка\n";
               if (x2 >= 0 && x2 <= M_PI/4){
                    for (int n = 1; n <= Nmax; n++){
                         double* res = r_sin(x2, n);
                         fout << n << "," << res[0] << "," << res[1]
<< "," << res[2] <<  "\n";
                    }
               }
               else if (x2 >= M_PI/4 && x2 <= M_PI/2){
                    for (int n = 1; n <= Nmax; n++){
                         double* res = r_cos(M_PI - x2, n);
                                                  fout << n
<< "," << res[0] << "," << res[1] << "," << res[2] <<  "\n";
                    }
               }
               else cout << "[*]Значение x для разложения Sin(x) в
ряд Маклорена должно находится в отрезке [0, pi/2]\n";
          }
          if (regime == 3) break;
     }
     return 0;
}
```

Примеры вывода:

1)

```
[*]Режим работы программы:
[1] - Демонстрация
[2] - Вывод в файл зависимостей точных значений, остаточных членов и
ошибок вычисления от числа членов ряда
[3] - Выход
1
[*]Для выбора введите число и нажмите Enter:
[1] - Логарифмическая функция
```

```
[2] - Гармоническая функция
1
[*]Введите значение x: 4
[*]Значение x для разложения ln(1+x) в ряд Маклорена должно быть в полуинтервале (-1.0, 1.0]
```

2)

```
[*]Режим работы программы:
[1] - Демонстрация
[2] - Вывод в файл зависимостей точных значений, остаточных членов и ошибок вычисления от числа членов ряда
[3] - Выход
1
[*]Для выбора введите число и нажмите Enter:
[1] - Логарифмическая функция
[2] - Гармоническая функция
1
[*]Введите значение x: 0.3
[*]Введите значение n: 20
--Значение функции в заданной точке: 0.262364
--Значение остаточного члена: -0.00310144
--Значение ошибки вычисления: -3.87301e-13
```

3)

```
[*]Режим работы программы:
[1] - Демонстрация
[2] - Вывод в файл зависимостей точных значений, остаточных членов и ошибок вычисления от числа членов ряда
[3] - Выход
1
[*]Для выбора введите число и нажмите Enter:
[1] - Логарифмическая функция
[2] - Гармоническая функция
2
[*]Введите значение x: 5
[*]Значение x для разложения Sin(x) в ряд Маклорена должно находится в отрезке [0, pi/2]
```

4)

```
[*]Режим работы программы:
[1] - Демонстрация
[2] - Вывод в файл зависимостей точных значений, остаточных членов и ошибок вычисления от числа членов ряда
[3] - Выход
1
[*]Для выбора введите число и нажмите Enter:
[1] - Логарифмическая функция
[2] - Гармоническая функция
2
[*]Введите значение x: 1.0466
[*]Введите значение n: 20
```

```
--Значение функции в заданной точке: 0.865726
--Абсолютное значение остаточного члена <=  4.40609e-37
--Значение ошибки вычисления: 1.11022e-16
```

Графики:

График зависимости точного значения от числа членов ряда для гармонической функции (x=1.0466):

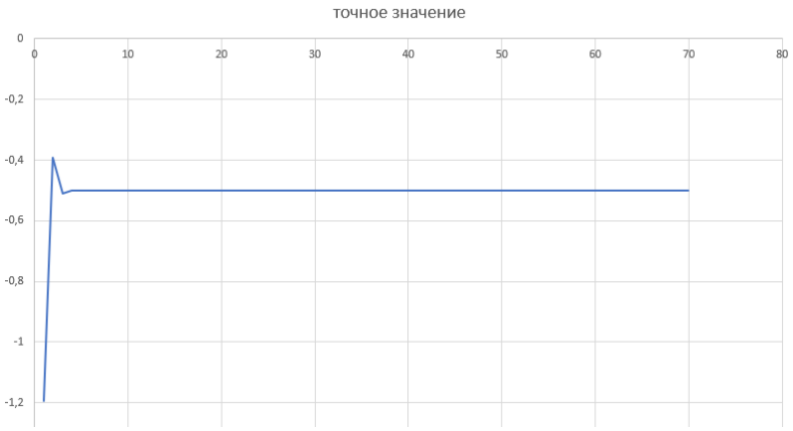


График зависимости остаточного члена от числа членов ряда для гармонической функции (x=1.0466):

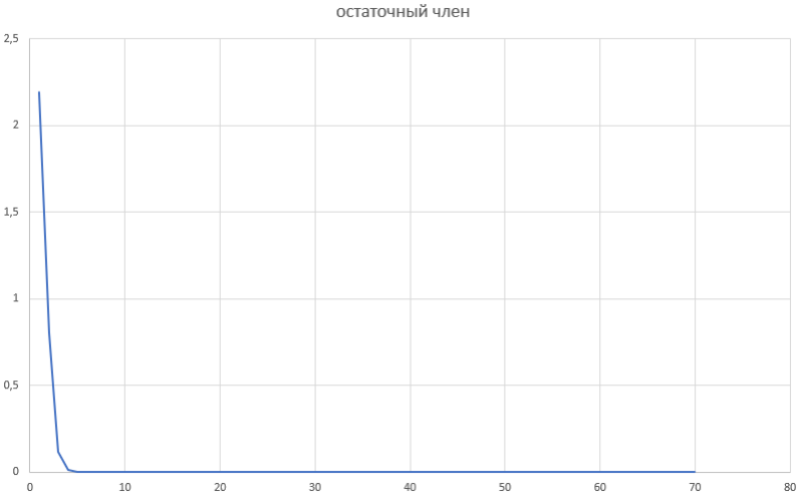


График зависимости ошибки от числа членов ряда для гармонической функции (x=1.0466):

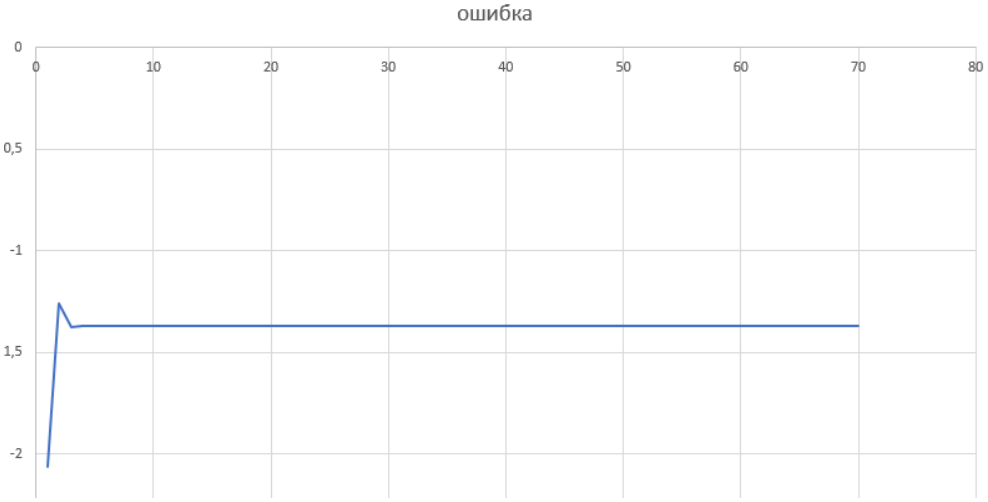


График зависимости точного значения от числа членов ряда для логарифмической функции (x=0.3):

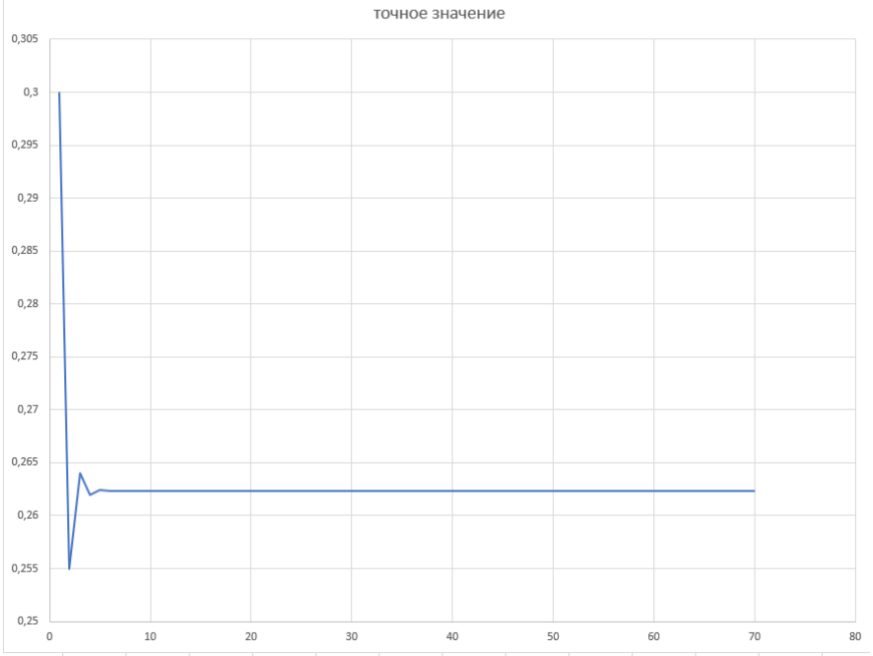


График зависимости остаточного члена от числа членов ряда для логарифмической функции (x=0.3):

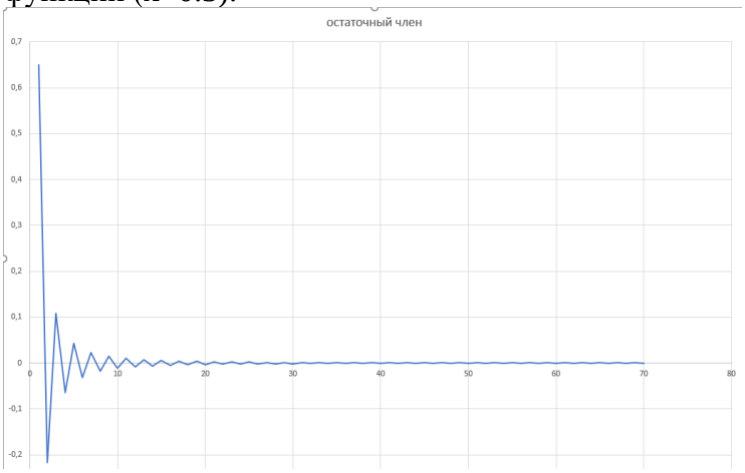
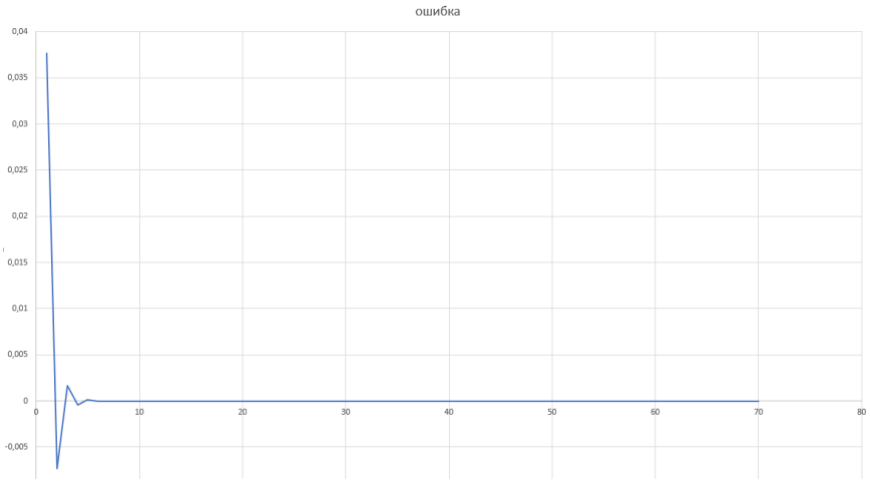


График зависимости ошибки от числа членов ряда для логарифмической функции (x=0.3):



Вывод:
Мы ознакомились с методом вычисления логарифмической и гармонической функций с помощью рядов Тейлора (Маклорена), так же реализовали программный код, использующий данный метод.

3. Приближённое решение уравнений

3.1 Графическое решение уравнений

Задание:

Графически определить верхние и нижние границы корней уравнений, проверить справедливость теоремы об отделении корней:

$$5x^4 - 2x^3 + 3x^2 + x - 4 = 0$$

$$x^5 - 3x^4 + 7x^2 + x - 8 = 0$$

Цель:
Ознакомиться с методом графического решения уравнений.

Теория:
Действительные корни уравнения f(x)=0 приближенно можно определить как абсциссы точек пересечения графика функции y = f(x) с осью Ox (рисунок 1):

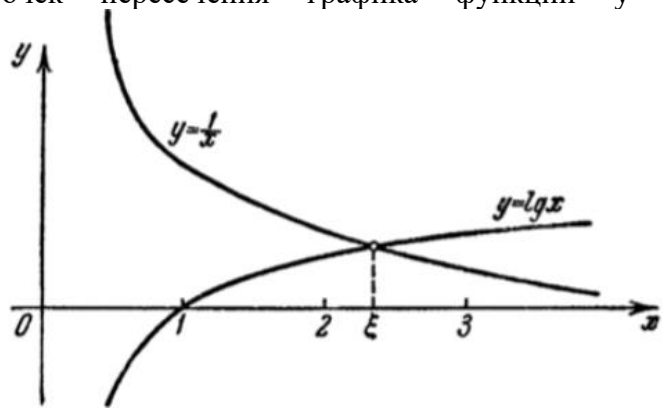


Рис.1

Если уравнение f(x)=0 не имеет близких между собой корней, то этим способом его корни легко отделяются. На практике часто бывает выгодно заменить уравнение f(x)=0 на равносильное ему уравнение $\varphi(x) = \psi(x)$, где функции $\varphi(x)$ и $\psi(x)$ – более простые, чем функция f(x). Тогда6 построив графики функций y = $\varphi(x)$ и y = $\psi(x)$, искомые корни получим как абсциссы точек пересечения этих графиков.

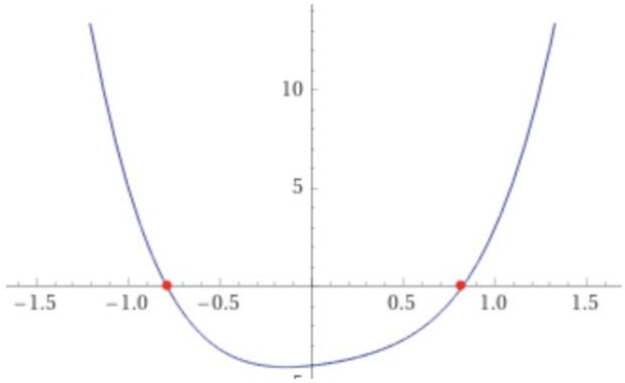
Программа:

```
#include <bits/stdc++.h>

using namespace std;

double f(double x, int k ) {
switch (k) {
case 1:
return 5*pow(x, 4) - 2*pow(x, 3) + 3*pow(x, 2) + x - 4;
case 2:
return pow(x, 5) - 3*pow(x, 4) + 7*pow(x,2) + x - 8;
}
}

int main() {
double a = -10.0, b = 10.0; // левая и правые границы
double n = 100000; // число отрезков
```

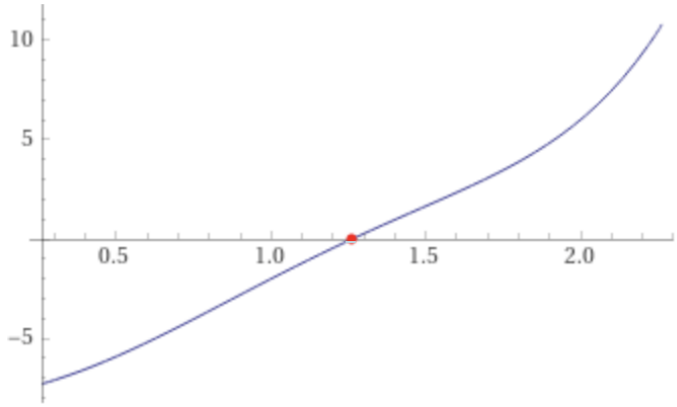
```
double h = (b - a)/n; // шаг
int k = 2; // номер функции
double x0 = a, y0 = f(x0,k), x = x0 + h;
double y;

while(x <= b) {
y = f(x,k);
if(y * y0 < 0)
cout « x0 « " "« x « endl;
x0 = x;
y0 = y;
x += h;
}

return 0;
}
```

Примеры вывода:

```
для k = 1
-0.7892 -0.789
0.8202 0.8204
```




```
для k = 2
1.2616 1.2618
```



Вывод:

Хотя графические методы решения уравнений просты и удобны, как правило, они применимы лишь для грубого определения корней. Особенно неблагоприятным случаем в смысле потери точности является случай, когда линии пересекаются под очень острым углом и практически сливаются по некоторой дуге.

3.2 Метод половинного деления

Задание:
Методом половинного деления найти хотя бы один вещественный корень уравнения, решение снабдить графиками функций и указать найденные корни:

$$2x^3 + x^2 - 7 = 0$$
$$5cos(3x) + 0.5x = 2, x \in [0; 2\pi]$$
$$x^5 - 2x^4 + 6x^2 + 2x - 4 = 0$$
$$x^3 - 0.2x^2 - 0.2x - 1.2 = 0$$
$$ln(|x^3| + 1) + x^3 = 2$$

Произвести оценку вычислительной сложности метода. Посчитать число итераций для решения уравнения с заданной точностью.

Цель:
Найти хотя бы один корень уравнения методом половинного деления, определить работоспособность и эффективность, оценив вычислительную сложность метода.

Теория:
Пусть дано уравнение f(x) = 0, где функция f(x) непрерывна на [a,b] и f(a)*f(b) < 0. Для нахождения одного из корней уравнения можно применить метод половинного деления: делим отрезок [a,b] пополам. Если $f\left(\frac{a+b}{2}\right) * f(b) < 0$, то рассматриваем отрезок $[\frac{a+b}{2}, b]$, иначе - отрезок $[a, \frac{a+b}{2}]$. Новый суженный отрезок $[a_1, b_1]$ снова делим пополам и проводим то же рассмотрение и т.д. В результате на каком-то этапе получим либо точный корень уравнения, лежащий в середине одного из отрезков, либо бесконечную последовательность вложенных друг в друга отрезков. В последнем случае корень уравнения определяется с заданной точностью так, что на каком-то этапе получим: $b_n - a_n < \varepsilon$, где ε – точность. Тогда искомый корень находится как $\frac{a_n+b_n}{2}$.

Программа:

```
#define _USE_MATH_DEFINES

#include <iostream>
#include <cmath>
#include <iomanip>
using namespace std;

double func(double x, int number) {
    switch (number) {
        case 1:
            return 2 * pow(x, 3) + pow(x, 2) - 7;
        case 2:
            return 5 * cos(3 * x) + 0.5 * x - 2;
        case 3:
            return pow(x, 5) - 2 * pow(x, 4) + 6 * pow(x, 2) + 2 * x -
4;
        case 4:
            return pow(x, 3) - 0.2 * pow(x, 2) - 0.2 * x - 1.2;
        case 5:
            return log(abs(pow(x, 3)) + 1) + pow(x, 3) - 2;
    }
```

```
}

void roots(double a, double b, double eps, int number) {
    if (b - a > eps) {
        double mid = (a + b) / 2;
        if (func(mid,number) == 0) {
            cout << "The root of the equation is : ";
            cout << fixed << setprecision(abs(log10(eps))) << (a + b)
/ 2;
            cout << "\n Check the root : " << func((a + b) / 2,
number) << endl;
        }
        else if (func(a, number) * func(mid, number) < 0)
            roots(a, mid, eps, number);
        else
            roots(mid, b, eps, number);
    }
    else {
        cout << "The root of the equation is : ";
        cout << fixed << setprecision(abs(log10(eps))) << (a + b) / 2;
        cout << "\n Check the root : " << func((a + b) / 2, number) <<
endl;
    }
    return;
}

int main() {
    double eps = 0;
    int number = 0;
    double a = 0, b = 0;
    cout << "Enter the number of task : ";
    cin >> number;
    cout << "\nEnter the section from a to b : ";
    cin >> a >> b;
    cout << "\nEnter the value of the error : ";
    cin >> eps;
    if (func(a, number) * func(b, number) < 0)
        roots(a, b, eps, number);
    else
        cout << "\n No root in this section";
    return 0;
}
```

Пример вывода:

```
Enter the number of task : 1

Enter the section from a to b : 0 2

Enter the value of the error : 0.0000000001
The root of the equation is : 1.3685984183
```

```
Check the root : 0.0000000003
```

Решение уравнений:

Найдем корни данных функций с помощью сервиса Wolfram Alpha, построим их графики. На рис. 1 представлен график решения в вещественных числах, рис. 2 – корни уравнения, рис. 3 – решение в комплексных числах.

Для 1):

Рис.1

$$2x^3 + x^2 - 7 = 0;$$

Рис.2

Real solution:

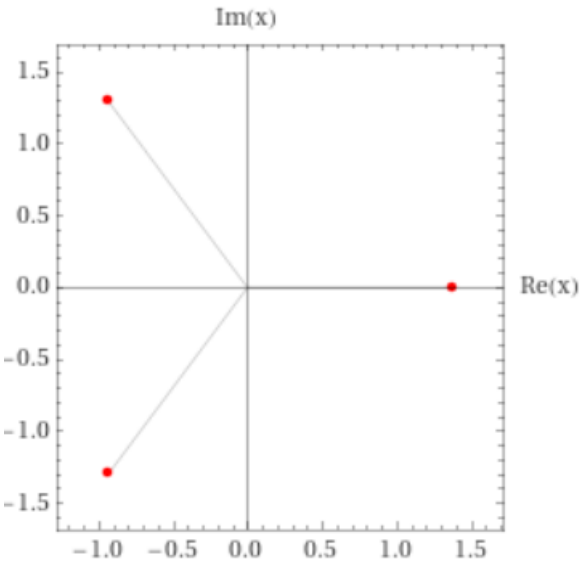
$x \approx 1.3686$

Complex solutions:

$x \approx -0.9343 - 1.2979\,i$

$x \approx -0.9343 + 1.2979\,i$

Рис.3



Для 2):

Рис.4

$$5\cos(3x) + 0.5x = 2, x \in [0; 2\pi];$$

Рис.5 - все корни уравнения на отрезке $[0;2\pi]$.

$x \approx 0.400874$

$x \approx 1.64987$

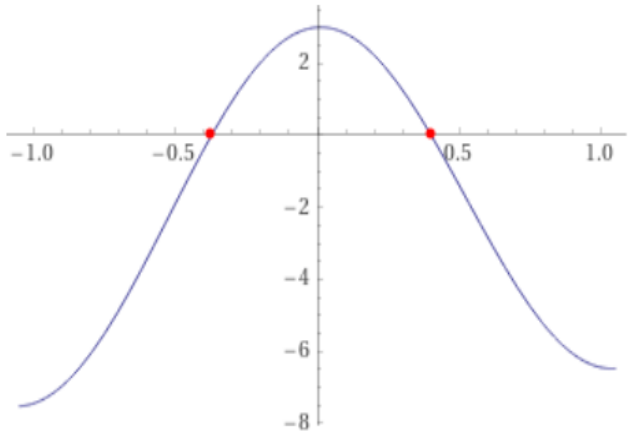
$x \approx 2.57017$

$x \approx 3.67599$

$x \approx 4.73698$

$x \approx 5.70256$

Рис.6 - представлены корни периодической функции



Для 3):

$$x^5 - 2x^4 + 6x^2 + 2x - 4 = 0;$$

Рис.7 - представлен вещественный корень уравнения

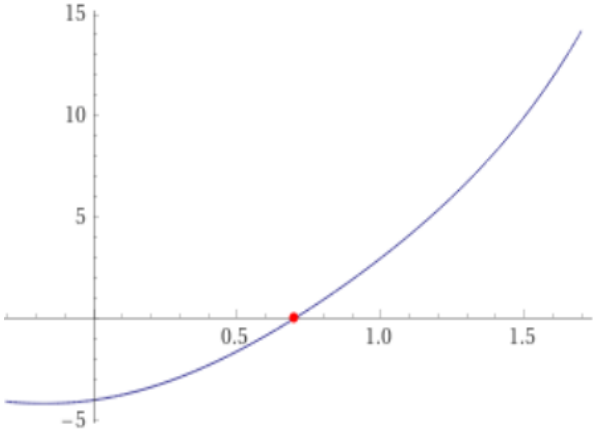


Рис.8

Real solution:

$x \approx 0.696849$

Complex solutions:

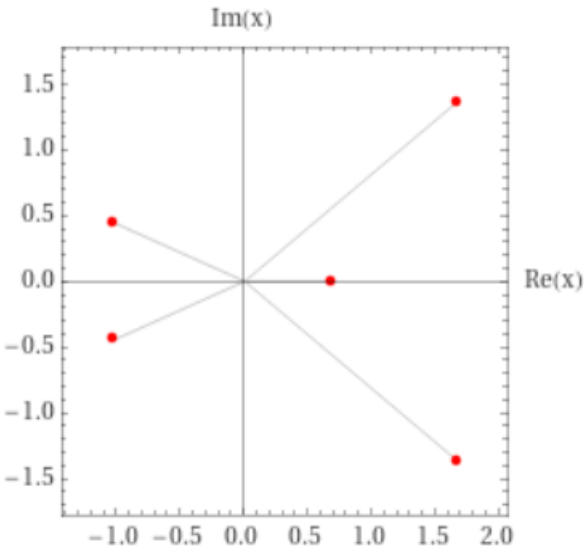
$x \approx -1.01796 - 0.442938\,i$

$x \approx -1.01796 + 0.442938\,i$

$x \approx 1.66954 - 1.36755\,i$

$x \approx 1.66954 + 1.36755\,i$

Рис.9



Для 4):

$x^3 - 0.2x^2 - 0.2x - 1.2 = 0.$

Рис.10 - представлен вещественный корень уравнения

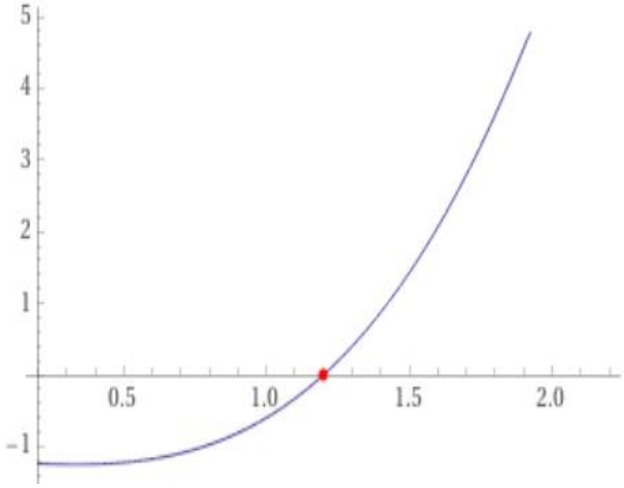


Рис.11- все корни уравнения

Real solution:

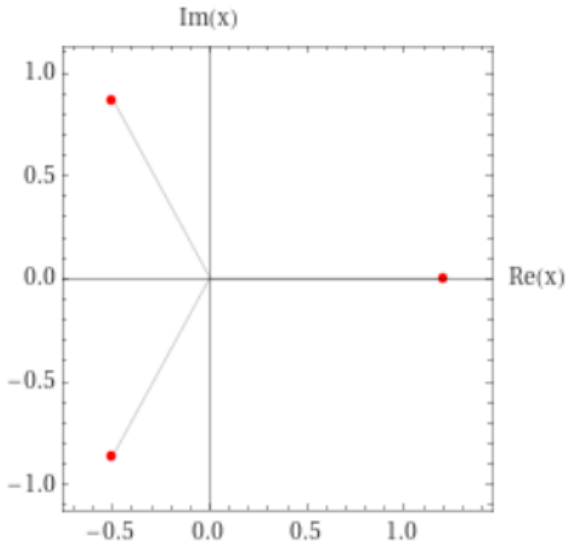
$x \approx 1.2$

Complex solutions:

$x = -0.5 - 0.866025\,i$

$x = -0.5 + 0.866025\,i$

Рис.12- отображает все корни в комплексной плоскости



Для 5):

$$\ln(|x^3|+1)+x^3=2.$$

Рис.13- представлен вещественный корень уравнения

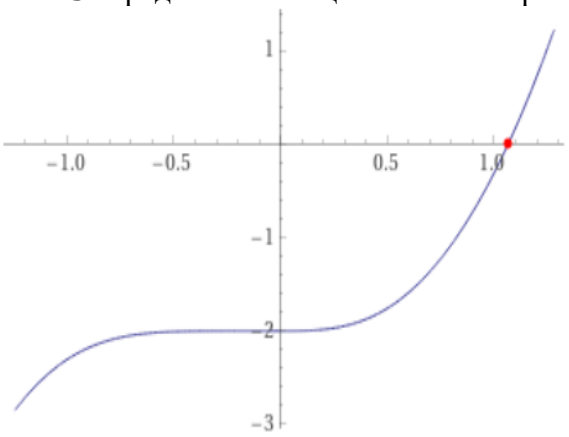


Рис.14- все корни уравнения под общей формулой.

$$x=\sqrt[3]{W(e^3)-1}$$

Оценка вычислительной сложности:

Посчитаем количество итераций в общем виде в зависимости от вводимой точности. Обозначим k – количество итераций (сокращения отрезка в два раза), [a,b] – отрезок, на котором находится корень уравнения, ε – введенная точность расчета. Тогда поиск корня останавливается при выполнении неравенства:

$$\frac{b-a}{2^k}\le\varepsilon;$$

Рассмотрим крайний случай, когда

$$\frac{b-a}{2^k}=\varepsilon;$$

Отсюда,

$$k=\log_2\frac{b-a}{\varepsilon}.$$

Тогда количество итераций для нахождения корня с заданной точностью определяется формулой выше, а вычислительная сложность решения равна O(log n).

Вывод:

В результате проделанной работы можно заключить, что метод является эффективным по точности и его удобно применять для грубого нахождения корня уравнения, когда проблематично найти общее решение. Однако, при большой величине вводимого отрезка или маленькой заданной точности объем вычислительной работы логарифмически растет.

3.3 Метод хорд

Задание:
Методом хорд найти хотя бы один вещественный корень уравнения, решение снабдить графиками функций и указать найденные корни:

$$5^x\sqrt{8^{x-1}} - 189 = 0$$
$$x^3 - x^2 + 2x - 5 = 0$$
$$2lgx^2 - 5lg^2x + 4 = 0$$
$$2sin(2x) - cos(3x) = 0.5, x \in [0; 2\pi]$$
$$2x^3 - 7x^2 - 7x\text{-}2.5=0$$

Цель:
Научиться использовать метод хорд для нахождения вещественных корней уравнения

Теория:
Метод хорд является усовершенствованным методом половинного деления.
Метод имеет более быструю сходимость.
Пусть x_1, x_2 — абсциссы концов хорды, $f(x) = 0$ — уравнение функции, решаемое методом секущих. Найдём коэффициенты k и b из системы уравнений

$$\begin{cases} f(x_1) - kx_1 + b, \\ f(x_2) = kx_2 + b. \end{cases}$$

$$k = \frac{f(x_2) - f(x_1)}{x_2 - x_1} \; b = f(x_1) - \frac{\big(f(x_2) - f(x_1)\big)x_1}{x_2 - x_1}$$

Уравнение принимает вид:

$$y = \frac{f(x_2) - f(x_1)}{x_2 - x_1}(x - x_1) + f(x_1)$$

Следовательно, первое приближение к корню, полученное методом хорд, принимает следующий вид:$x_3 = x_1 - \frac{(x_2 - x_1)f(x_1)}{f(x_2) - f(x_1)}$.

Таким образом, итерационная формула метода хорд имеет вид:

$$x_{i+1} = x_{i-1} - \frac{f(x_{i-1})(x_i - x_{i-1})}{f(x_i) - f(x_{i-1})}$$

Повторять операцию следует до тех пор, пока $|x_i - x_{i-1}|$ не станет меньше или равно заданному значению погрешности.

Программа:

```
#include <iostream>
#include <cmath>
#include <windows.h>

using namespace std;
double x, l, r, d, epsilon;
int number;

double func(double x) {
    if (number == 1)
    {
```

```
        return (pow(5, x) * sqrt(pow(8, (x - 1))) - 189);
    }
    if (number == 2)
    {
        return (pow(x, 3) - pow(x, 2) + 2 * x - 5);
    }
    if (number == 3)
    {
        return (2 * log10(pow(x, 2)) - 5 * pow(log10(x), 2) - 4);
    }
    if (number == 4)
    {
        return (2 * sin(2 * x) - cos(3 * x) - 0.5);
    }
    if (number == 5)
    {
        return (2 * pow(x, 3) - 7 * pow(x, 2) - 7 * x - 2.5);
    }
    else
    {
        cout << "Некорректный выбор" << endl;
        exit(0);
    }
}

double dfunc_2(double x) {
    if (number == 1)
    {
        return (sqrt(2) * pow(5, x) * (pow(log(8), 2) + 4 *
pow(log(5), 2) + 4 * log(5) * log(8)) * sqrt(pow(8, x)) / 16);
    }
    if (number == 2)
    {
        return (6 * x - 2);
    }
    if (number == 3)
    {
        return (2 * (5 * log10(x) - 5 / (log(10)) - 2) / (pow(x, 2) *
log(10)));
    }
    if (number == 4)
    {
        return (-8 * sin(2 * x) + 9 * cos(3 * x));
    }
    if (number == 5)
    {
        return (2 * (6 * x - 7));
    }

}
```

```
void solve() {
    if (dfunc_2(r) * func(r) < 0) x = l;
    if (dfunc_2(l) * func(l) < 0) x = r;
    do {
        d = x;
        x = l - func(l) * (r - l) / (func(r) - func(l));
        d = abs(d - x);
        if (func(x) * func(r) < 0) l = x;
        if (func(x) * func(l) < 0) r = x;
        cout << l << ", " << r << " " << d << endl;
    } while (d > epsilon);
}


int main() {
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    cout << "Выберите уравнение." << endl << "1 - '5^x*sqrt(8^(x-1)) -
189';" << endl << "2 - 'x^3 - x^2 + 2*x - 5';" << endl << "3 -
'2*lg(x^2) - 5*(lg(x)^2) - 4';" << endl << "4 - '2*sin(2*x) - cos(3*x)
- 0.5;'" << endl << "5 - '2*x^3 - 7*x^2 - 7*x - 2.5'" << endl << endl;
    cin >> number;
    cout << "Введите левую и правую границы" << endl;
    cin >> l >> r;
    cout << "Введите погрешность" << endl;
    cin >> epsilon;
    if (abs(r - l) <= epsilon) {
        cout << "Некорректный выбор границ интервала" << endl;
        exit(0);
    }
    if (func(l) * func(r) > 0) {
        cout << "Некорректный выбор границ интервала" << endl;
        return 2;
    }
    solve();
    cout << "x = " << x << ", e = " << d << endl;
}
```

Пример вывода:

Выберите уравнение.
1 - '5^x*sqrt(8^(x-1)) - 189';
2 - 'x^3 - x^2 + 2*x - 5';
3 - '2*lg(x^2) - 5*(lg(x)^2) - 4';
4 - '2*sin(2*x) - cos(3*x) - 0.5;'
5 - '2*x^3 - 7*x^2 - 7*x - 2.5'

1
Введите левую и правую границы
2
3
Введите погрешность
0.0001

x = 2.37112, e = 0

2
Введите левую и правую границы
1
2
Введите погрешность
0.001
x = 1.63963, e = 0.00075758

3
Введите левую и правую границы
0
2
Введите погрешность
0.0001
Корень уравнения = nan

4
Введите левую и правую границы
0
1
Введите погрешность
0.0001
x = 0.298642, e = 8.557e-06

5
Введите левую и правую границы
4
5
Введите погрешность
0.001
x = 4.3668, e = 0.000740879

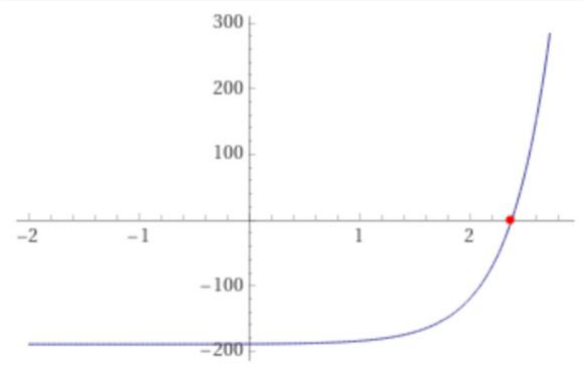
Графики:

$$5^x\sqrt{8^{x-1}} - 189 = 0$$

1 корень на интервале [2;3]

Numerical solution:

$x \approx 2.37111798004356\ldots$



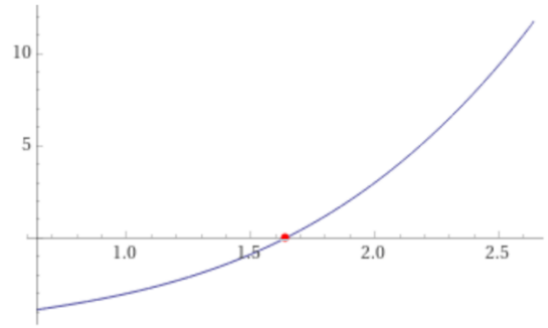
$$x^3 - x^2 + 2x - 5 = 0$$

1 корень на интервале [1;2]

Real solution:

$x \approx 1.6398$

Root plot:



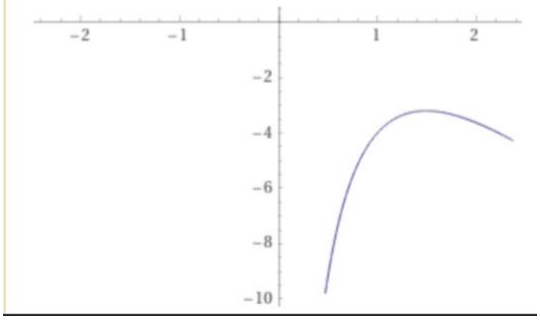
$$2lgx^2 - 5lg^2x - 4 = 0$$

Корней нет

Input:

$2\log(x^2) - 5\log^2(x) - 4 = 0$

Plot:



$$2sin(2x) - cos(3x) = 0.5, x \in [0; 2\pi]$$

4 корня на: [0;1], [1;2], [2;4], [4;5]

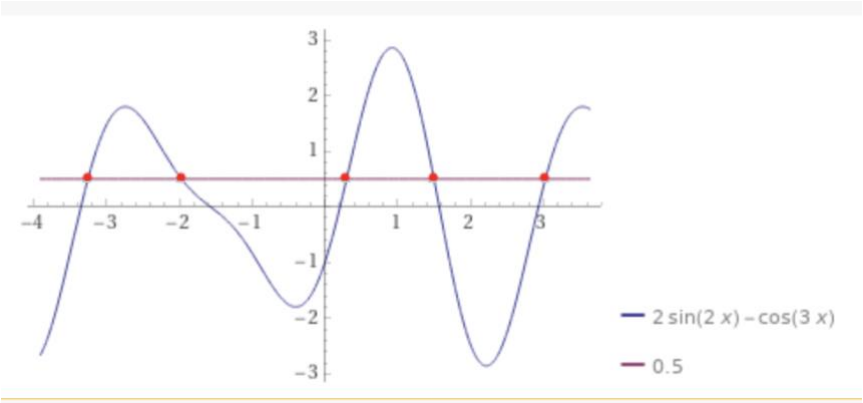
$x \approx 2\,(3.14159\,n - 0.984611)\,, \quad n \in \mathbb{Z}$

$x \approx 2\,(3.14159\,n + 0.149321)\,, \quad n \in \mathbb{Z}$

$x \approx 2\,(3.14159\,n + 0.749495)\,, \quad n \in \mathbb{Z}$

$x \approx 2\,(3.14159\,n + 1.51481)\,, \quad n \in \mathbb{Z}$

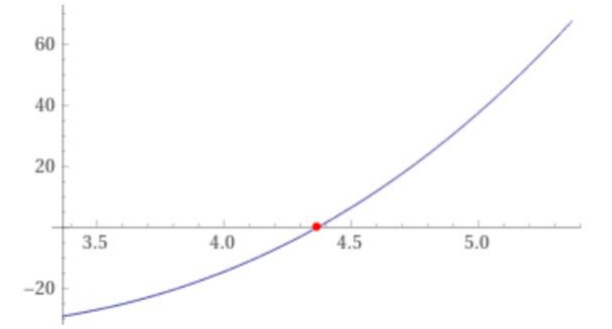
$x \approx 2\,(3.14159\,n - (0.71451 + 0.34885\,i))\,, \quad n \in \mathbb{Z}$



$2x^3 - 7x^2 - 7x$-2.5=0

1 корень на [4;5]

$x \approx 4.36701$



Вывод:

Мы ознакомились с методом хорд, который имеет более быструю сходимость, чем метод половинного деления.

3.4 Метод Ньютона

Задание:

Методом Ньютона найти хотя бы один вещественный корень уравнений, решения снабдить графиками функций и указать найденные корни:

$$2lg(x) - cos(x) = 0, x \in (0; 4\pi]$$
$$2x^3 - 5x^2 - 1 = 0$$
$$2sin^3(2x) - cos(x) = 0, x \in [0; \pi]$$
$$x^5 - 3x^4 + 8x^2 + 2x - 7 = 0$$
$$0.5x^2 5cos(2x) - 2 = 0, x \in [-\pi; \pi]$$

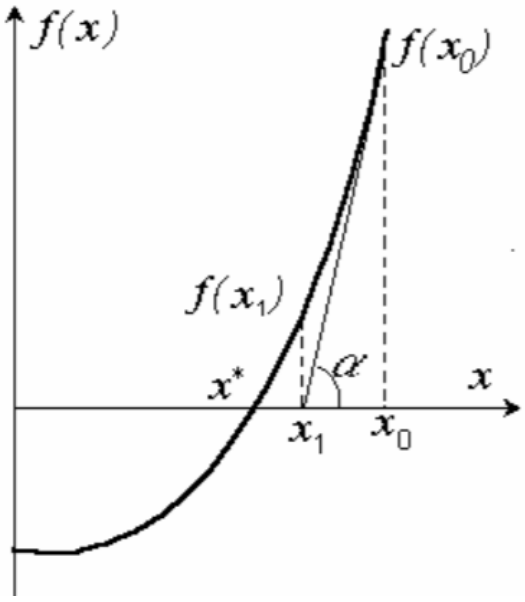
Цель:

Ознакомиться с методом Ньютона для нахождения вещественных корней уравнения.

Теория:

Основная идея метода заключается в следующем: задаётся начальное приближение вблизи предположительного корня, после чего строится касательная к графику исследуемой функции в точке приближения, для которой находится пересечение с осью абсцисс. Эта точка берётся в качестве следующего приближения. И так далее, пока не будет достигнута необходимая точность.

Пусть x* есть корень уравнения f(x) = 0, единственный на отрезке [a, b]. Предположим, что каким-либо способом, например, графически определено начальное приближение x0 к корню. В этой точке вычислим значение функции f(x0) и ее производной f ′(x_0 = x) . Значение этой производной равно tg(α) – тангенсу угла наклона соответствующей касательной к оси абсцисс. Точка x_1 пересечения этой касательной с осью абсцисс есть следующее приближение к корню. Эта точка x_1 принимается за новое начальное приближение и процесс повторяется. Из рисунка видно, что процесс сходится к искомому корню x*.



Процесс уточнения корня закончится, когда выполнится условие $|x_{n+1} - x_n| < \varepsilon$ или $|f(x_{n+1})| < \varepsilon$ (т.е.погрешность в нужных пределах).

Где ε – допустимая погрешность определения корня. Из геометрических соображений можно получить расчетную формулу для метода Ньютона в виде:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Программа:

```
#include <iostream>
#include <cmath>

using namespace std;
double l, r, epsilon, x0, x1, amendment;
int number;

double f7(double x) {
    if (number == 1)
    {
        return (2 * log10(x) - cos(x));
    }
    if (number == 2)
    {
        return (2*pow(x,3) - 5 * pow(x,2) -1);
    }
    if (number == 3)
    {
        return (2* pow(sin(2*x),3) - cos(x));
    }
    if (number == 4)
    {
        return (pow(x, 5) - 3 * pow(x, 4) + 8 * pow(x, 2) + 2 * x -
7);
    }
    if (number == 5)
    {
        return (0.5 * pow(x,2) + 5 * cos(2*x) - 2);
    }
    else
    {
        cout << "Некорректный выбор" << endl;
        exit(0);
    }
}
double df7(double x) {
    if (number == 1)
    {
        return (2 / (x * log(10)) + sin(x));
    }
    if (number == 2)
    {
        return (6 * pow(x, 2) - 10 * x);
    }
    if (number == 3)
    {
        return (12 * pow(sin(2 * x), 2) * cos(2*x) + sin(x));
    }
    if (number == 4)
    {
```

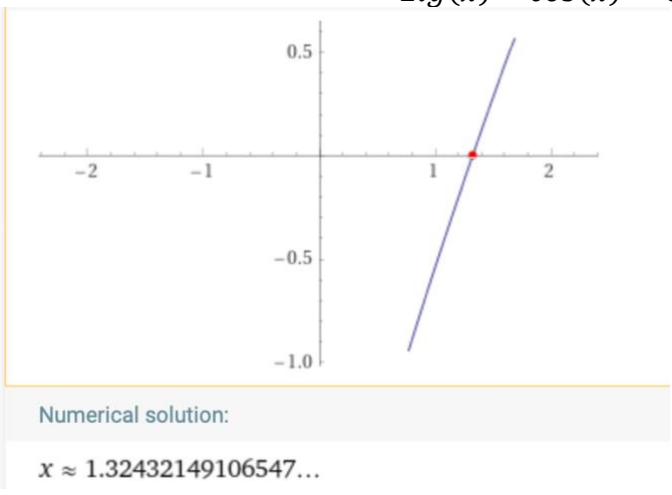
```
        return (5 * pow(x, 4) - 12 * pow(x, 3) + 16 * x + 2);
    }
    if (number == 5)
    {
        return (x - 10 * sin(2 * x));
    }
    else
    {
        cout << "Некорректный выбор" << endl;
        exit(0);
    }
}


int main() {
    //SetConsoleCP(1251);
    //SetConsoleOutputCP(1251);
    cout << "Выберите уравнение." << endl << "1 - '2*lg(x) - cos(x)';"
<< endl << "2 - '2*x^3 - 5*x^2 -1';" << endl << "3 - '2*(sin(2*x))^3 -
cos(x)';" << endl << "4 - 'x^5 - 3*x^4 + 8*x^2+2*x-7';" << endl << "5
- '0.5*x^2 + 5*cos(2*x)-2'" << endl << endl;
    cin >> number;
    cout << "Введите грубое приближение x0, EPS" << endl;
    cin >> x0 >> epsilon;
    cout << "Введите левую и правую границы" << endl;
    cin >> l >> r;

    if (epsilon < 0)
        cout << "Шаг не может быть отрицательным числом" << endl;
    if (x0 > r || x0 < l)
        cout << "Грубое приближения корня должно входить в интервал"
<< endl;

    do
    {
        amendment = f7(x0) / df7(x0);
        x1 = x0 - amendment;
        if (f7(x1) == 0)
            break;
        x0 = x1;
    } while (abs(f7(x1)) > epsilon);
    cout << "Корень уравнения = " << x1 << endl;
    exit(0);
}
```

Пример вывода:

```
1
Введите грубое приближение x0, EPS
1 0.001
Введите левую и правую границы
0 2
Корень уравнения = 1.32432

2
Введите грубое приближение x0, EPS
2 0.001
Введите левую и правую границы
1.5 3
Корень уравнения = 2.57539

3
Введите грубое приближение x0, EPS
1 0.1
Введите левую и правую границы
0 2
Корень уравнения = 1.29304

4
Введите грубое приближение x0, EPS
1 0.1
Введите левую и правую границы
0 2
Корень уравнения = 0.909091

5
Введите грубое приближение x0, EPS
1 0.1
Введите левую и правую границы
-3 3
Корень уравнения = 0.599805
```

Графики:

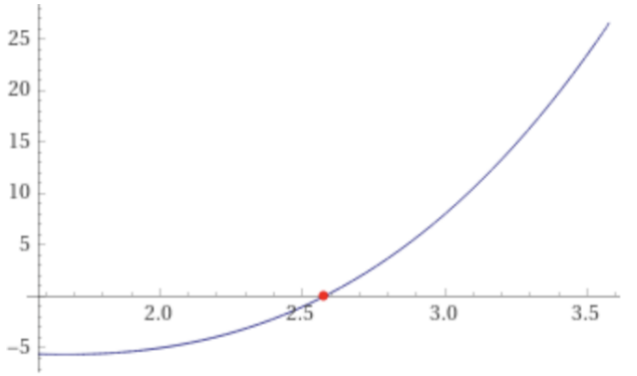
$$2lg(x) - cos(x) = 0, x \in (0; 4\pi]$$



$$2x^3 - 5x^2 - 1 = 0$$

Real solution:

$x \approx 2.5754$

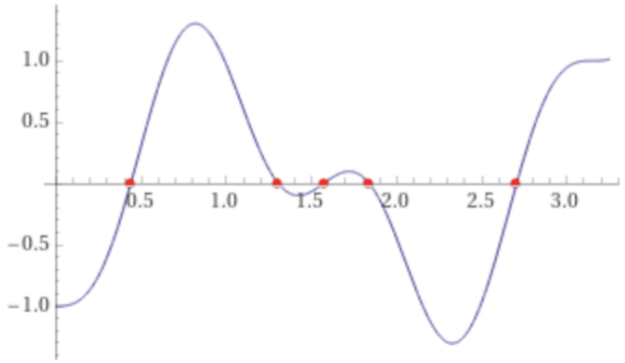


$$2sin^3(2x) - cos(x) = 0, x \in [0; \pi]$$

корень 1.19912

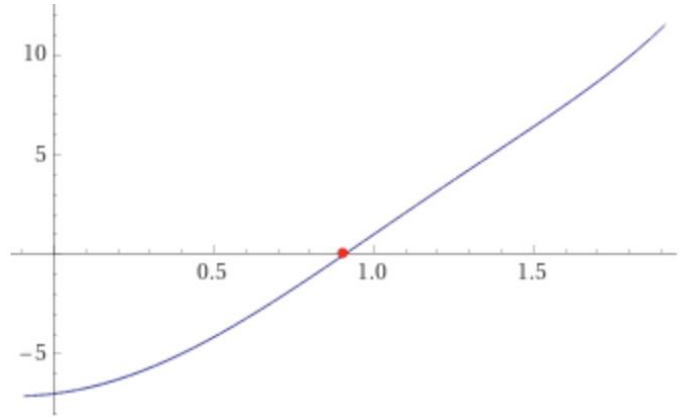
plot $2\sin^3(2x) - \cos(x) = 0$ $x = 0$ to π

Root plot:



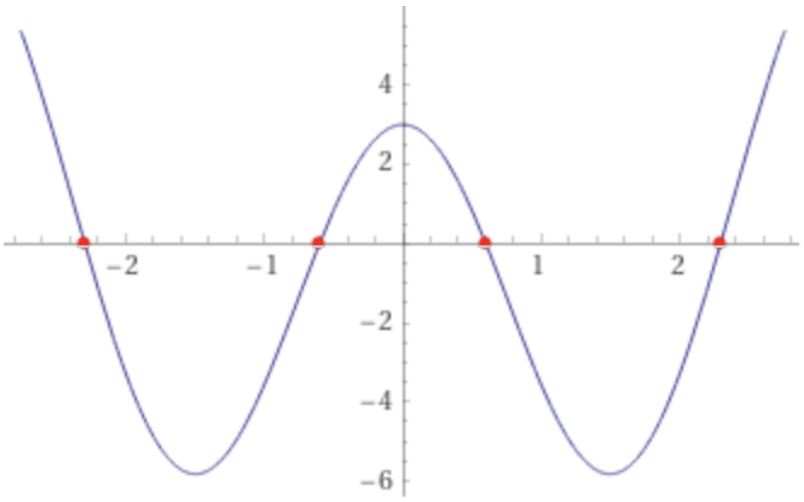
$$x^5 - 3x^4 + 8x^2 + 2x - 7 = 0$$

$x \approx 0.908942$



$$0.5x^25cos(2x) - 2 = 0, x \in [-\pi; \pi]$$

$x \approx 0.599058$



Вывод:

Метод может быть применён не всегда:

1) Если начальное приближение недостаточно близко к решению, то метод может не сойтись

2) Если производная в точке корня равна нулю, то скорость сходимости не будет квадратичной, а сам метод может преждевременно прекратить поиск, и дать неверное для заданной точности приближение

3) Если не существует вторая производная в точке корня, то скорость сходимости метода может быть заметно снижена

4) Если производная не непрерывна в точке корня, то метод может расходиться в любой окрестности корня.

3.5 Метод итераций

Задание:

Методом итерации найти хотя бы один вещественный корень уравнения, решение снабдить графиками функций и указать найденные корни:

$$x^3 + x = 1000$$
$$cos(x) = 0.1, 0 < x \leq 4\pi$$
$$x^5 - x^4 - x^2 - x\text{-}5=0$$
$$x^3 - x - 1 = 0$$
$$ln(x) + x = 2.25$$

Теория:

Суть метода итерации заключается в следующем: дано нелинейное уравнение f(x) = 0, заменим его эквивалентным уравнением x = φ(x) . Пусть на отрезке [a, b] расположен единственный корень. Примем за x0 любое значение из интервала [a, b]. Вычислим значение функции при x = φ(x) при x = x0 и найдем уточненное значение x1 = φ(x0) . Продолжая этот процесс неограниченно, получим последовательность приближений к корню xn+1 = φ(xn).

Сходимость: если функция φ(x) определена и непрерывна на интервале [a, b] и |φ'(x)| < 1, x∈[a, b], то процесс итераций сходится с любой точностью при любом начальном значении x0 из интервала [a, b].

Геометрическая иллюстрация метода: корнем исходного нелинейного уравнения является абсцисса точки пересечения линии y =φ(x) с прямой y = x.

Из графиков можно увидеть, что возможны как сходящиеся (Рис.1) так и расходящееся (Рис.2) итерационные процессы. Скорость сходимости зависит от абсолютной величины φ'(x).

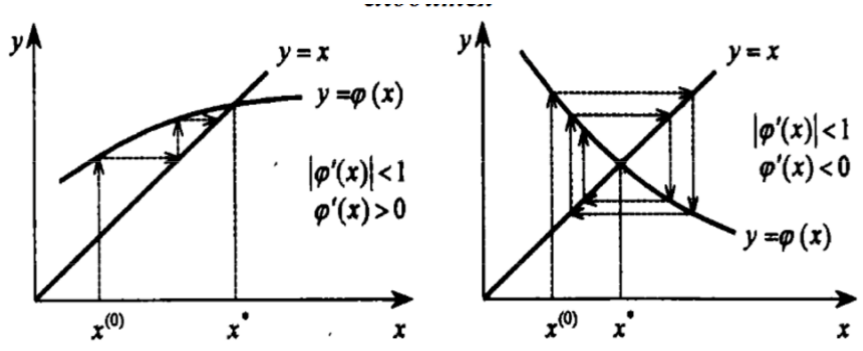


Рис.1 – метод сходится

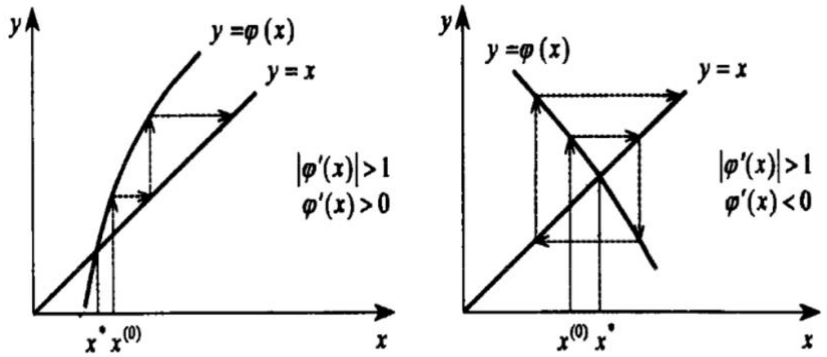


Рис. 2 – метод расходится

Критерий окончания: при заданной точности вычисления нужно вести до тех пор, пока не будет выполнено неравенство:

$$|x_n - x_{n-1}| < \frac{1-q}{q}\varepsilon$$

Если q⩽ 0.5, можно использовать упрощенное условие:

$$|x_n - x_{n-1}| < \varepsilon$$

Программа:

```
#include <iostream>
#include <cmath>
#include <iomanip>

#define e 2.7182818284

using namespace std;

double IterativeFunction(int taskNumber, double argument) {
    double x = argument;
    switch (taskNumber) {
    case(1):
        return acos(1./10);
    case(2):
        return pow(1000 - x, 1./3);
    case(3):
        return pow(pow(x,4)+pow(x,2)+x+5, 1./5);
    case(4):
        return pow(1+x, 1./3);
    case(5):
        return 2.25-log(x);
    default:
        return 0;
    }
}
double CheckResult(int taskNumber, double argument) {
    double x = argument;
    switch (taskNumber) {
    case(1):
        return cos(x) - 0.1;
    case(2):
        return pow(x, 3) + x - 1000;
    case(3):
        return pow(x, 5) - pow(x, 4) - pow(x, 2) - x - 5;
    case(4):
        return pow(x, 3) - x - 1;
    case(5):
        return log(x) + x - 2.25;
    default:
        return 0;
    }
}

void SolveEquation(int taskNumber, double accuracy) {
    double root = 1;
```

```
    double previousRoot = 0;

    root = IterativeFunction(taskNumber, root);
    do {
         previousRoot = root;
         root = IterativeFunction(taskNumber, root);
    } while (abs(root - previousRoot) > accuracy);

    cout << "Корень x равен " << fixed <<
setprecision((abs(log10(accuracy)))) << root;
    cout << "\nПроверка решения. f(x) = " << fixed <<
setprecision((abs(log10(accuracy)))) << CheckResult(taskNumber, root);
}

int main()
{
    setlocale(LC_ALL, "Russian");

    double epsilon;
    int taskNumber;
    cout << "Введите номер задачи: ";
    cin >> taskNumber;
    cout << "Введите точность нахождения решения: ";
    cin >> epsilon;
    SolveEquation(taskNumber, epsilon);
    return 0;}
```

Пример вывода:

Введите номер уравнения, для которого необходимо вычислить корень:

1) x^3 + x = 1000

2) cos(x) = 0.01, x∈ (0, 4π]

3) x^5 - x^4 - x^2 - x - 5 = 0

4) x^3 - x - 1 = 0

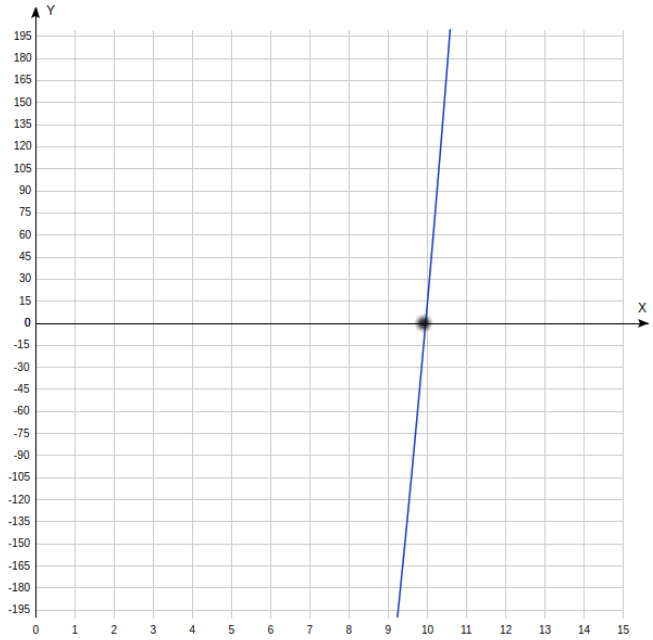
5) ln(x) + x = 2.25

1

x = 9.966667

f(x) = -0.0000000000

Число итераций = 4

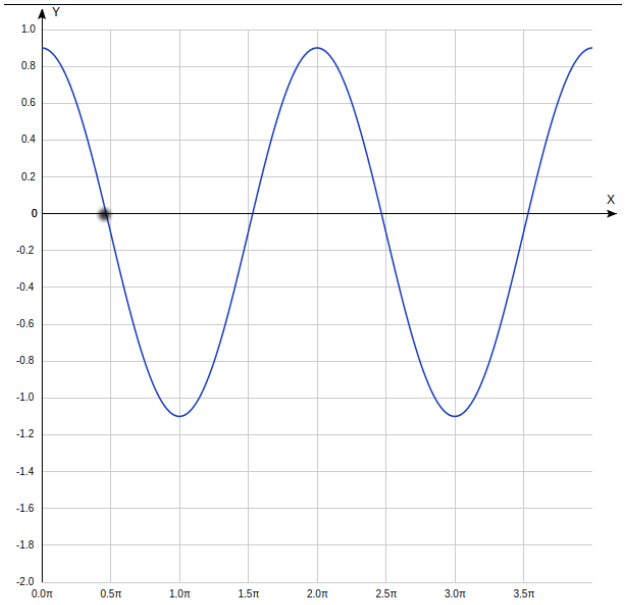


2

x = 1.470629

f(x) = -0.0000000000

Число итераций = 2

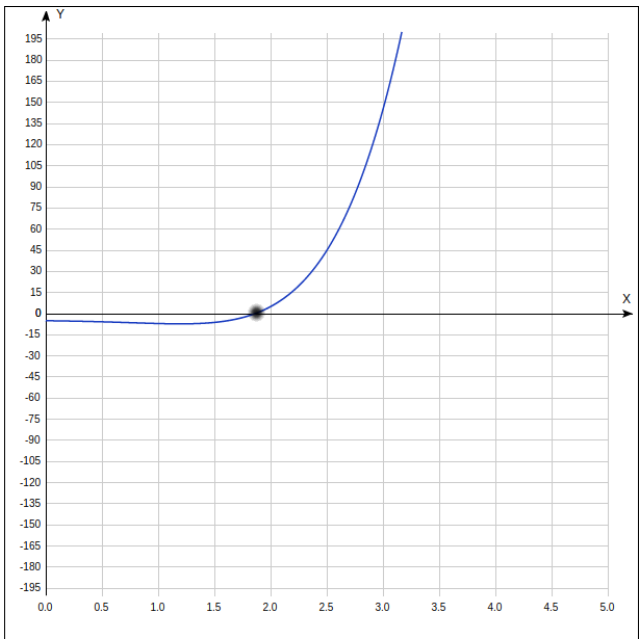


3

f(x) = -0.0000000000

x = 1.860891

Число итераций = 17

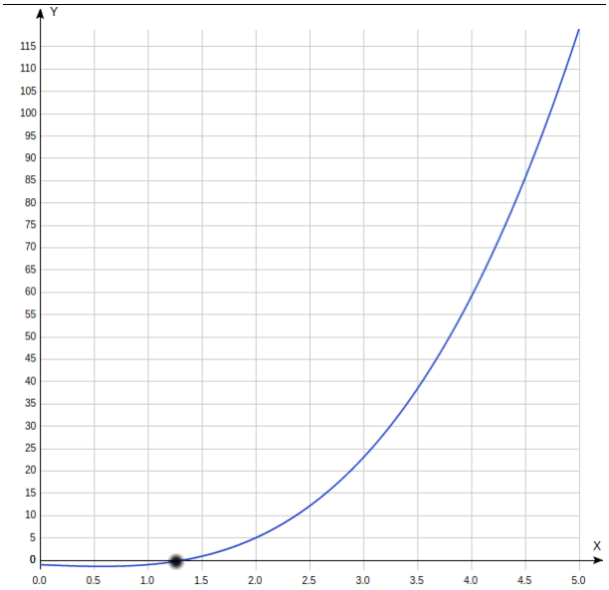


4

x = 1.324717

f(x) = -0.0000000000

Число итераций = 8



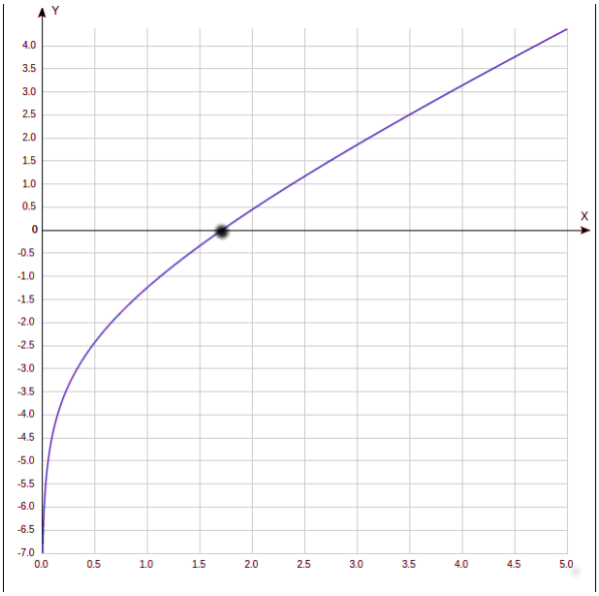
Введите номер уравнения, для которого необходимо вычислить корень:

5

x = 1.712217

f(x) = 0.0000000000

Число итераций = 23



Вывод:

По результатам использования программы, можно сделать вывод, что метод итераций считает корни с очень высокой точностью. Стоит отметить, что от пользователя алгоритма требуется небольшая сноровка в составлении итеративной функции, которая бы удовлетворяла условию теоремы. Также важным является факт, что сложность алгоритма сильно зависит от исходного уравнения. После нахождения итеративной функции метод реализуется довольно просто.

3.6 Метод Ньютона для комплексных корней

Задание:
Методом Ньютона с заданной точностью найти хотя бы один комплекснозначный корень следующих уравнений:

$$4z^4 + 2z^2 + 1.3 = 0$$
$$z^2 + 2.71 = 0$$
$$2e^z + \sqrt{2} = 0$$

Теория:
Пусть f(z)(z=x+iy), $i^2 = -1$- аналитическая функция в некоторой выпуклой окрестности U ее простого изолированного нуля

$$\zeta = \xi + \iota\eta$$
$$(f(\zeta) = 0, f'(\zeta) \neq 0),$$

Который является комплексным. Пусть z_n – приближенное значение корня, принадлежащее окрестности U, и $z_{n+1} = z_n + \Delta z_n$- уточненное значение корня. Применяя разложение в ряд Тейлора в точке z_n и считая, что $f(z_{n+1}) \approx 0$ с точностью до Δz_n будем иметь $f(z_{n+1}) \approx f(z_n) + \Delta z_n f'(z_n)$ отсюда $\Delta z_n = -\frac{l(z_n)}{l\prime(z_n)}$. Если z_n принадлежит U(n= 1,2, …) и последовательность $\{z_n\}$ сходится, то предел $\zeta = \lim\limits_{n\to\infty} z_n$ является искомым корнем уравнения.

Программа:

```
#define _USE_MATH_DEFINES

#include <iostream>
#include <cmath>
#include <complex>

using namespace std;
double a, b, x, y, eps1 ,eps2;
complex <float> z0, z1, amendment, epsilon, k1, k2;
complex <float> kfree, kexp;
int number;




complex <float> fz(complex <float> z, int number) {
    switch (number) {
        case 1:
            k1 = 4.0;
            k2 = 2.0;
            kfree = (1.3,0);
            return (k1 * pow(z, 4) + k2 * pow(z, 2) + kfree);
        case 2:
            kfree = 2.71;
            return (pow(z, 2) + kfree);
        case 3:
            k1 = 2.0;
            kfree = sqrt(2);
            kexp = M_E;
```

```
            return (k1 * pow(kexp, z) + kfree);
    }
}

complex <float> dfz(complex <float> z, int number) {
    switch (number) {
        case 1:
            k1 = 16.0;
            k2 = 4.0;
            return (k1 * pow(z, 3) + k2 * z);
        case 2:
            k1 = 2.0;
            return (k1 * z);
        case 3:
            k1 = 2.0;
            kexp = M_E;
            return (k1 * pow(kexp, z));
    }
}


int main() {

    float a, b;
    cout << "Выберите уравнение." << endl << "1 - '4*z^4 + 2*z^2 + 1.3
= 0';\n" << "2 - 'z^2 + 2.71';\n" << "3 - '2*e^z + sqrt(2)'\n";
    cin >> number;
    cout << "Введите грубое приближение z0 (Re и Im), EPS (Re и
Im)\n";
    cin >> a >> b >> eps1 >> eps2;
    cout << "Введите левую и правую границы\n";
    cin >> x >> y;
    complex <float> z0(a, b);
    complex <float> zero = 0.0;
    epsilon = (eps1, eps2);
    do
    {
        amendment = fz(z0,number) / dfz(z0,number);
        z1 = z0 - amendment;
        if (fz(z1,number) == zero)
            break;
        z0 = z1;
    } while (abs(fz(z1,number)) > abs(epsilon));
    cout << "Корень уравнения = " << z1 << endl;

}
```

Пример вывода:

```
1
Введите грубое приближение z0 (Re и Im), EPS (Re и Im)
1
```

```
-1
0.01
0.01
Введите левую и правую границы
-2
3
Корень уравнения = (-0.0152977,-0.0465125)

2
Введите грубое приближение z0 (Re и Im), EPS (Re и Im)
0 -1.6
0.01 0.01
Введите левую и правую границы
-2 2
Корень уравнения = (0,-1.64688)
```

Вывод:
Мы ознакомились с методом Ньютона для комплексных корней.

3.7 Метод Бернулли решения алгебраических уравнений

Задание:
Методом Бернулли с заданной точностью найти хотя бы один корень следующих уравнений:

$$5x^4 - 2x^3 + 3x^2 + x - 4 = 0$$
$$x^5 + 5x^4 - 5 = 0$$

Теория:

Метод Бернулли позволяет найти наибольший и наименьший по модулю корень алгебраического уравнения, но и несколько ближайших к нему (по модулю) корней.

Вычисления по методу Бернулли сводятся в основном к построению некоторой последовательности чисел $\mu_1, \mu_2, \mu_3 \ldots$, для построения которой выбираются вначале некоторые, вообще говоря, произвольные значения $\mu_0, \mu_{-1}, \mu_{-2}, .. \mu_{-(n-1)}$. После этого значения $\mu_1, \mu_2, \mu_3 \ldots$ вычисляются с помощью рекуррентной формулы:

$$\mu_k = p_1\mu_{k-1} + p_2\mu_{k-2} + \ldots + p_n\mu_{k-n}, \text{ k= 1,2 3}\ldots$$

Далее по виду последовательности определяется вид наибольшего (наименьшего) по модулю корня и значение этого корня.

Далее после того, как наибольший корень вычислен с достаточной степенью точности, определяется второй по величине модуля корень. Для второго корня строиться новая последовательность $\{\mu_k\}$, вид которой определяется на основании типа сходимости последовательности, построенной для предыдущего корня.

После того как найден второй по модулю корень, аналогично находятся третий и последующие корни.

Пусть погрешность округления во всех вычислениях постоянна и равна ε. Тогда относительная погрешность первого корня равна

$$\delta x_1 = n\delta\mu_k, \text{ где } \delta\mu_k = \frac{\varepsilon}{\mu_k}.$$

Потеря точности для последующих корней может быть значительно больше.

Программа:

```
#include <iostream>
#include <vector>

using namespace std;

int main(){

vector<double> coef;
int n;
cout << "Введите степень алгебраического уравнения: ";
cin >> n;
```

```
                            cout << "Введите коэффициенты при степенях
x и свободный член: \n";
                            for (int i = 0; i <= n; i++){
                                cout << "при x^" << n-i << ":  ";
                                double a;
                                cin >> a;
                                coef.push_back(a);
                            }
                            int k;
                            cout << "Введите точность (число членов
последовательности, удовлетворяющей конечно-разностному уравнению в
методе Бернулли): \n";
                            cin >> k;
                            vector<double> y(n-1, 0.0);
                            y.push_back(1.0);
                            for (int i = 0; i <= k-n; i++){
                                double S = 0;
                                for (int j = 0; j < n; j++)
                                     S += coef[n-j]*y[i+j];
                                S /= -1*coef[0];
                                y.push_back(S);
                            }
                            cout << "Наибольший по абсолютной величине
корень алгебраического уравнения:  " << y[y.size()-1]/y[y.size()-2] <<
"\n";


                            return 0;
}
```

Пример вывода:

```
Введите степень алгебраического уравнения: 5
Введите коэффициенты при степенях x и свободный член:
при x^5:  1
при x^4:  5
при x^3:  0
при x^2:  0
при x^1:  0
при x^0:  -5
Введите точность (число членов последовательности, удовлетворяющей
конечно-разностному уравнению в методе Бернулли):
13
Наибольший по абсолютной величине корень алгебраического уравнения:  -
4.99195

Введите степень алгебраического уравнения: 4
Введите коэффициенты при степенях x и свободный член:
при x^4:  5
при x^3:  -2
```

```
при x^2:  3
при x^1:  1
при x^0:  -4
Введите точность (число членов последовательности, удовлетворяющей
конечно-разностному уравнению в методе Бернулли):
20
Наибольший по абсолютной величине корень алгебраического уравнения:  -
0.0126227
```

Выводы:
Метод Бернулли обладает очень простой вычислительной схемой. Основные вычисления сводятся к повторению операции накопления, что делает метод удобным для вычисления на ЭВМ. Кроме того, корни в методе Бернулли определяются не все сразу, а один или несколько наибольших (наименьших) по модулю корней, что приводит к потере точности для остальных корней.

4. Элементы теории матриц

4.1 Сложение и умножение матриц

Задание:
Разработать алгоритм и написать программу, реализующую: ввод комплекснозначных матриц из консоли или из файла и следующие операции над матрицами, включая проверку реализуемости операции: сложение, вычитание, умножение на комплексное число, умножение матрицы на матрицу.

Теория:
Суммой двух матриц $\boldsymbol{A} = (\boldsymbol{a_{ij}})$ и $\mathbf{B} = (\boldsymbol{b_{ij}})$ одинакового порядка называют матрицу $\mathbf{C} = (\boldsymbol{c_{ij}})$ такого же порядка, элементы которой равны сумме соответствующих элементов матриц $\boldsymbol{A}$ и **B**. Т. е. $\boldsymbol{c_{ij}} = \boldsymbol{a_{ij}} + \boldsymbol{b_{ij}}$
Аналогично, разностью двух матриц $\boldsymbol{A} = (\boldsymbol{a_{ij}})$ и $\mathbf{B} = (\boldsymbol{b_{ij}})$ одинакового порядка называют матрицу $\mathbf{C} = (\boldsymbol{c_{ij}})$ такого же порядка, элементы которой равны разности соответствующих элементов матриц $\boldsymbol{A}$ и **B**. Т. е. $\boldsymbol{c_{ij}} = \boldsymbol{a_{ij}} - \boldsymbol{b_{ij}}$
Произведением матрицы $A = (a_{ij})$ на число k есть матрица $\mathrm{C} = (c_{ij})$ того же порядка, что и матрица A, элементы которой получены умножением соответствующих элементов матрицы A на число k, то есть $c_{ij} = k * a_{ij}$
Если количество столбцов матрицы $A(k \times n)$ равно числу строк матрицы $\mathrm{B}(n \times p)$, то для них определена матрица $\mathrm{C}(k \times p)$, которую называют её произведением. Элементы матрицы C находятся по правилу: элемент c_{ij} равен сумме попарных произведений элементов $i-$ й строки матрицы A и j−го столбца матрицы B

Программа:
```
#include <iostream>
#include <complex>
#include <cmath>

using namespace std;

class Matrix
{
private:
    complex <double>** matrix;
    int n, m;
    bool isSquared = false;
public:
    Matrix(int numberOfRows, int numberOfColumns)
    {
        n = numberOfRows;
        m = numberOfColumns;
        if (n == m)
            isSquared = true;
        matrix = new complex <double> * [n];
        for (int i = 0; i < n; ++i)
        {
            matrix[i] = new complex <double>[m];
```

```
            for (int j = 0; j < m; ++j)
                matrix[i][j] = 0;
        }
    }
    Matrix()
    {
        matrix = 0;
        n = 0;
        m = 0;
    }
    int numberOfRows()
    {
        return n;
    }
    int numberOfColumns()
    {
        return m;
    }
    bool IsSquared()
    {
        return isSquared;
    }
    complex <double>* operator [] (int index)
    {
        return getRow(index);
    }
    // получить строку матрицы
    complex <double>* getRow(int index)
    {
        if (index < 0 || index >= n)
            return 0;
        return matrix[index];
    }
};
    // прочитать матрицу из консоли
    Matrix MarixInput()
    {
        int rows, columns;
        cout << "Количество строк: ";
        cin >> rows;
        cout << "Количество столбцов: ";
        cin >> columns;
        Matrix a = Matrix(rows, columns);
        cout << "Введите элементы матрицы через Enter: " << endl;
        for (int i = 0; i < rows; ++i)
            for (int j = 0; j < columns; ++j)
                cin >> a[i][j];
        return a;
    }
    // вывести матрицу в консоль
    void printMatrix(Matrix& a)
```

```
    {
        for (int i = 0; i < a.numberOfRows(); ++i)
        {
            for (int j = 0; j < a.numberOfColumns(); ++j)
                cout << a[i][j] << " ";
            cout << endl;
        }
    }

    // сложить две матрицы
    Matrix MatrixSumm(Matrix& firstMatrix, Matrix& secondMatrix)
    {
        Matrix a = firstMatrix;
        Matrix b = secondMatrix;
        if (a.numberOfRows() != b.numberOfRows() ||
a.numberOfColumns() != b.numberOfColumns())
        {
            cout << "Ошибка. Матрицы нельзя сложить между собой.
(Несовпадение размеров)" << endl;
            return Matrix();
        }
        Matrix result = Matrix(a.numberOfRows(), b.numberOfColumns());
        for (int i = 0; i < result.numberOfRows(); ++i)
            for (int j = 0; j < result.numberOfColumns(); ++j)
                result[i][j] = a[i][j] + b[i][j];
        return result;
    }
    // вычесть две матрицы
    Matrix MatrixSubstraction(Matrix& firstMatrix, Matrix&
secondMatrix)
    {
        Matrix a = firstMatrix;
        Matrix b = secondMatrix;
        if (a.numberOfRows() != b.numberOfRows() ||
a.numberOfColumns() != b.numberOfColumns())
        {
            cout << "Ошибка. Произвести вычитание невозможно
(Несовпадение размеров)." << endl;
            return Matrix();
        }
        Matrix result = Matrix(a.numberOfRows(), b.numberOfColumns());
        for (int i = 0; i < result.numberOfRows(); ++i)
            for (int j = 0; j < result.numberOfColumns(); ++j)
                result[i][j] = a[i][j] - b[i][j];
        return result;
    }

    // умножение матрицы на вещественное число
    Matrix MatrixMultiplication(Matrix& matrix, double multiplicator)
    {
        int k = multiplicator;
```

```
        Matrix result = Matrix(matrix.numberOfRows(),
matrix.numberOfColumns());
        for (int i = 0; i < matrix.numberOfRows(); ++i)
            for (int j = 0; j < matrix.numberOfColumns(); ++j)
                result[i][j] = matrix[i][j] * multiplicator;
        return result;
    }

    // умножение матрицы на комплексное число
    Matrix MatrixMultiplication(Matrix& matrix, complex<double>
multiplicator)
    {
        complex<double> k = multiplicator;
        Matrix result = Matrix(matrix.numberOfRows(),
matrix.numberOfColumns());
        for (int i = 0; i < matrix.numberOfRows(); ++i)
            for (int j = 0; j < matrix.numberOfColumns(); ++j)
                result[i][j] = matrix[i][j] * multiplicator;
        return result;
    }
    // умножение двух матриц
    Matrix MatrixMultiplication(Matrix& firstMatrix, Matrix&
secondMatrix)
    {
        Matrix a = firstMatrix;
        Matrix b = secondMatrix;
        if (a.numberOfColumns() != b.numberOfRows())
        {
            cout << "Ошибка. Нельзя произвести перемножение матриц.
(Кол-во столбцов A != кол-во срок B" << endl;
            return Matrix();
        }
        Matrix c = Matrix(a.numberOfRows(), b.numberOfColumns());
        for (int i = 0; i < a.numberOfRows(); ++i)
            for (int j = 0; j < b.numberOfColumns(); ++j)
                for (int k = 0; k < a.numberOfColumns(); ++k)
                    c[i][j] += a[i][k] * b[k][j];
        return c;
    }

    int main()
    {
        setlocale(LC_ALL, "Russian");

        complex<double> deter;
        cout << "Ввод первой матрицы. Ввод комплексных чисел в формате
(real, image)" << endl;
        Matrix firstMatrix = MarixInput();
        cout << "Ввод второй матрицы. " << endl;
        Matrix secondMatrix = MarixInput();
        cout << "Первая матрица A: " << endl;
```

```
        printMatrix(firstMatrix);
        cout << "Вторая матрица B:  " << endl;
        printMatrix(secondMatrix);

        cout << "A + B = " << endl;
        Matrix sumOfTwo = MatrixSumm(firstMatrix, secondMatrix);
        printMatrix(sumOfTwo);

        cout << "A - B = " << endl;
        Matrix subtrOfTwo = MatrixSubstraction(firstMatrix,
secondMatrix);
        printMatrix(subtrOfTwo);

        cout << "Введите вещественный множитель: ";
        double multNumber;
        cin >> multNumber;
        cout << "A * " << multNumber << " = " << endl;
        Matrix multToNumber = MatrixMultiplication(firstMatrix,
multNumber);
        printMatrix(multToNumber);


        cout << "Введите комплексный множитель: ";
        complex<double> multNumber2;
        cin >> multNumber2;
        cout << "A * " << multNumber2 << " = " << endl;
        multToNumber = MatrixMultiplication(firstMatrix, multNumber2);
        printMatrix(multToNumber);

        cout << "A * B = " << endl;
        Matrix multOfTwo = MatrixMultiplication(firstMatrix,
secondMatrix);
        printMatrix(multOfTwo);
        return 0;
    }
```

Пример вывода:

```
Ввод первой матрицы. Ввод комплексных чисел в формате (real, image)
Количество строк: 2
Количество столбцов: 2
Введите элементы матрицы через Enter:
1
4
7
-9
Ввод второй матрицы.
Количество строк: 2
Количество столбцов: 2
Введите элементы матрицы через Enter:
5
4
8
```

```
0.1
Первая матрица A:
(1,0) (4,0)
(7,0) (-9,0)
Вторая матрица B:
(5,0) (4,0)
(8,0) (0.1,0)
A + B =
(6,0) (8,0)
(15,0) (-8.9,0)
A - B =
(-4,0) (0,0)
(-1,0) (-9.1,0)
Введите вещественный множитель: 88
A * 88 =
(88,0) (352,0)
(616,0) (-792,0)
Введите комплексный множитель: 4
A * (4,0) =
(4,0) (16,0)
(28,0) (-36,0)
A * B =
(37,0) (4.4,0)
(-37,0) (27.1,0)
```

Вывод:

Мы ознакомились с операциями сложения, вычитания и умножения (на число, на матрицу) матриц.

4.2 Определитель

Задание:
Разработать алгоритм и написать программу, реализующую: вычисление определителя методом Гаусса, для матриц заданных на множестве вещественных чисел.

Теория:
Чтобы найти определитель матрицы методом Гаусса, необходимо: привести матрицу к верхне-треугольной или нижне-треугольной форме используя разрешённые над матрицей преобразования, называемые также элементарными. Сосчитать произведение всех членов матрицы, принадлежащих главной матричной диагонали полученной треугольной матрицы (эта диагональ проходит слева-направо сверху-вниз). При осуществлении подсчётов для вычисления определителя матрицы методом Гаусса нужно помнить, что при перестановке строчек или столбцов необходимо поменять знак детерминанта в конце решения на противоположный.

Программа:
```
#include <bits/stdc++.h>

using namespace std;

typedef vector<vector<double>> real_matrix;
const double EPS = 1E-9;

void input_matrix(real_matrix &arr, int n) {
    cout << "Вводите матрицу" << endl;
    arr.resize(n);
    for (int i = 0; i < n; i++) {
        cout << "Введите строчку " << to_string(i) << endl;
        arr[i].resize(n);
        for (int j = 0; j < n; j++)
            cin >> arr[i][j];
    }
}

double determinant(real_matrix a, int n) {
    double d = 1;
    for (int i = 0; i < n; ++i) {
        int k = i;
        for (int j = i + 1; j < n; ++j)
            if (abs(a[j][i]) > abs(a[k][i]))
                k = j;
        if (abs(a[k][i]) < EPS) {
            d = 0;
            break;
        }
        swap(a[i], a[k]);
        if (i != k)
            d = -d;
        d = a[i][i] * d;
        for (int j = i + 1; j < n; ++j)
```

```
            a[i][j] /= a[i][i];
        for (int j = 0; j < n; ++j)
            if (j != i && abs(a[j][i]) > EPS)
                for (int k = i + 1; k < n; ++k)
                    a[j][k] -= a[i][k] * a[j][i];
    }

    return d;
}
int main() {
    int n;
    cout << "Введите n: " << endl;
    cin >> n;

    real_matrix a;
    input_matrix(a, n);
    double det = determinant(a, n);
    cout << "Определитель: " << det << endl;

return 0;
}
```

Пример вывода:

```
Введите n:
2
Вводите матрицу
Введите строчку 0
0.1 0.02
Введите строчку 1
0.004 34
Определитель: 3.39992
```

Вывод:

Таким образом, мы можем с помощью алгоритма Гаусса вычислять определитель матрицы за O (N^3).

4.3 Обратная матрица

Задание:
Вычисление обратной матрицы. Разработать алгоритм и написать программу, реализующую: ввод вещественнозначных матриц из консоли и из файла и вычисление обратной матрицы на основе элементарных преобразований матрицы. В алгоритм программы интегрировать проверку путём умножения вычисленной обратной матрицы на исходную. Оценить вычисленную сложность вычисления обратной матрицы на основе элементарных преобразований.

Теория:
Обратной матрицей по отношению к данной называется матрица, которая, будучи умноженная как справа, так и слева на данную матрицу, дает единичную матрицу: $A^{-1} \times A = A \times A^{-1} = E$, где E – единичная матрица.
Транспонированная относительно матрицы A матрица A' получается, если из строк матрицы A сделать столбцы, а из её столбцов - наоборот, строки, то есть заменить строки столбцами.
Нахождение обратной матрицы для данной называется обращением данной матрицы.
Квадратная матрица обратима тогда и только тогда, когда ее определитель не равен нулю (матрица не вырождена).
Для не квадратных и вырожденных матриц обращение невозможно.
Нахождение обратной матрицы:
Матрицу, обратную данной матрице A представим в виде:
$A^{-1} = \frac{1}{|A|} * A^{-}$, где– A^{-} это союзная матрица (составленная из алгебраических дополнений для соответствующих элементов транспонированной матрицы).
Алгоритм:
1)Найти матрицу, транспонированную относительно A.
2) Вычислить элементы союзной матрицы как алгебраические дополнения матрицы, найденной на шаге 1.
3) Применить формулу: умножить число, обратное определителю матрицы A, на союзную матрицу, найденную на шаге 4.
4) Проверить полученный на шаге 3 результат, умножив данную матрицу A на обратную матрицу. Если произведение этих матриц равно единичной матрицы, значит обратная матрица была найдена верно. В противном случае начать процесс решения снова.
Сложность алгоритма $O(n^2) \times O_{\Delta}$.

Код программы:
```
#include <iostream>
using namespace std;

void inversion(double **A, int N)
{
    double temp;

    double **E = new double *[N];

    for (int i = 0; i < N; i++)
        E[i] = new double [N];
```

```
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
    {
        E[i][j] = 0.0;

        if (i == j)
            E[i][j] = 1.0;
    }

for (int k = 0; k < N; k++)
{
    temp = A[k][k];

    for (int j = 0; j < N; j++)
    {
        A[k][j] /= temp;
        E[k][j] /= temp;
    }

    for (int i = k + 1; i < N; i++)
    {
        temp = A[i][k];

        for (int j = 0; j < N; j++)
        {
            A[i][j] -= A[k][j] * temp;
            E[i][j] -= E[k][j] * temp;
        }
    }
}

for (int k = N - 1; k > 0; k--)
{
    for (int i = k - 1; i >= 0; i--)
    {
        temp = A[i][k];

        for (int j = 0; j < N; j++)
        {
            A[i][j] -= A[k][j] * temp;
            E[i][j] -= E[k][j] * temp;
        }
    }
}

for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        A[i][j] = E[i][j];

for (int i = 0; i < N; i++)
    delete [] E[i];
```

```
    delete [] E;
}

int main()
{
    int N;

    std::cout << "Введите размерность квадратной матрицы N: ";
    std::cin >> N;

    double **matrix = new double *[N];
    double **matrix2 = new double *[N];
    double **res = new double *[N];
    for (int i = 0; i < N; i++){
        matrix[i] = new double [N];
        matrix2[i] = new double [N];
        res[i] = new double [N];
    }
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
        {
            cout << "Введите элемент матрицы[" << i << "][" << j
<< "] = ";
            cin >> matrix[i][j];
            matrix2[i][j] = matrix[i][j];
        }

    inversion(matrix, N);
    cout << "Обратная матрица:"<< '\n';
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            cout <<matrix[i][j] << "  ";

        cout << std::endl;
    }

    for (int i = 0; i < N; i++){
        for (int k = 0; k < N; k++){
            double el = 0.0;
            for (int j = 0; j<N; j++)
                el += matrix2[i][j]*matrix[j][k];
            res[i][k] = el;
        }
    }
    cout << "Проверка:"<< '\n';
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            cout <<res[i][j] << "  ";
```

```
        cout << std::endl;
    }

    for (int i = 0; i < N; i++){
        delete [] matrix[i];
        delete [] matrix2[i];
        delete [] res[i];
    }


    delete [] matrix;
    delete [] matrix2;
    delete [] res;

    cin.get();
    return 0;
}
```

Пример вывода:

```
Введите размерность квадратной матрицы N: 2
Введите элемент матрицы[0][0] = 1
Введите элемент матрицы[0][1] = 2
Введите элемент матрицы[1][0] = 1
Введите элемент матрицы[1][1] = 4
Обратная матрица:
2  -1
-0.5  0.5
Проверка:
1  0
0  1

Введите размерность квадратной матрицы N: 6

Введите элемент матрицы[0][0] = 3

Введите элемент матрицы[0][1] = 5

Введите элемент матрицы[0][2] = 2

Введите элемент матрицы[0][3] = 7

Введите элемент матрицы[0][4] = 5

Введите элемент матрицы[0][5] = 1

Введите элемент матрицы[1][0] = 3

Введите элемент матрицы[1][1] = 7

Введите элемент матрицы[1][2] = 8
```

```
Введите элемент матрицы[1][3] = 5
Введите элемент матрицы[1][4] = 3
Введите элемент матрицы[1][5] = 6
Введите элемент матрицы[2][0] = 7
Введите элемент матрицы[2][1] = 5
Введите элемент матрицы[2][2] = 6
Введите элемент матрицы[2][3] = 7
Введите элемент матрицы[2][4] = 3
Введите элемент матрицы[2][5] = 4
Введите элемент матрицы[3][0] = 5
Введите элемент матрицы[3][1] = 2
Введите элемент матрицы[3][2] = 6
Введите элемент матрицы[3][3] = 8
Введите элемент матрицы[3][4] = 5
Введите элемент матрицы[3][5] = 6
Введите элемент матрицы[4][0] = 3
Введите элемент матрицы[4][1] = 4
Введите элемент матрицы[4][2] = 9
Введите элемент матрицы[4][3] = 7
Введите элемент матрицы[4][4] = 6
Введите элемент матрицы[4][5] = 2
Введите элемент матрицы[5][0] = 4
Введите элемент матрицы[5][1] = 5
Введите элемент матрицы[5][2] = 4
Введите элемент матрицы[5][3] = 6
Введите элемент матрицы[5][4] = 5
Введите элемент матрицы[5][5] = 4
Обратная матрица:
```

```
-0.232697  -0.203527  0.232143  -0.133319  0.0250665  0.318767

0.0805235  0.105701  0.0178571  -0.14929  -0.0468057  0.0507986

-0.0874002  0.0312777  0.0357143  -0.0252884  0.136202  -0.0909494

0.391526  0.136535  -0.0178571  0.230035  -0.0898403  -0.58496

-0.240683  -0.205745  -0.125  -0.110248  0.128882  0.59472

-0.066992  0.0925022  -0.107143  0.137977  -0.129104  0.092724

Проверка:

1  -6.93889e-17  8.32667e-17  -5.55112e-17  1.11022e-16  1.38778e-
17

-3.33067e-16  1  -1.11022e-16  0  0  3.33067e-16

-6.10623e-16   -2.77556e-16   1   -5.55112e-16   2.22045e-16   -
1.94289e-15

-9.99201e-16   -7.77156e-16   3.33067e-16   1   3.33067e-16   -
1.22125e-15

-6.38378e-16   -1.38778e-16   -1.66533e-16   -3.33067e-16   1   -
4.16334e-16

-6.10623e-16  -5.55112e-17  0  -1.11022e-16  0  1

Введите размерность квадратной матрицы N: 3

Введите элемент матрицы[0][0] = 0.1

Введите элемент матрицы[0][1] = 0.003

Введите элемент матрицы[0][2] = 0.77

Введите элемент матрицы[1][0] = 45

Введите элемент матрицы[1][1] = 0.4

Введите элемент матрицы[1][2] = -5

Введите элемент матрицы[2][0] = -0.99

Введите элемент матрицы[2][1] = 9

Введите элемент матрицы[2][2] = 5

Обратная матрица:

0.148643  0.0218694  -0.00102152

-0.695932  0.00399216  0.111166
```

```
1.28211  -0.00285574  -0.000300448

Проверка:

1  4.33681e-19  5.42101e-20

-8.88178e-16  1  -6.07153e-18

-8.99725e-13  1.1692e-15  1
```

Вывод:
При проверке (перемножении матриц) возникает погрешность в элементах матрицы, не лежащих на главной диагонали. Это возникает из-за использования типа double. Чем больше операций деления и умножения, тем больше погрешность.

5. Линейные уравнения

5.1 Метод Крамера

Задание:
Разработать алгоритм и написать программу, реализующую: решение системы линейных уравнений, на множестве вещественных чисел, методом Крамера. Предусмотреть возможность решения уравнений различного порядка без перекомпиляции программы. Исходные данные (матрицу коэффициентов уравнения, вектор правых частей), вводить в программу из консоли или из файла.

Теория:

Для системы n-линейных уравнений с n неизвестными

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \cdots\ \cdots\ \cdots\ \cdots\ \cdots\ \cdots\ \cdots\ \cdots\ \cdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{cases}$$

с определителем матрицы системы Δ, отличным от нуля, решение по методу Крамера находится как

$$x_i = \frac{1}{\Delta} \begin{vmatrix} a_{11} \dots a_{1,i-1} & b_1 & a_{1,i+1} \dots a_{1n} \\ a_{21} \dots a_{2,i-1} & b_2 & a_{2,i+1} \dots a_{2n} \\ \cdots\ \cdots\ \cdots\ \cdots\ \cdots\ \cdots\ \cdots \\ a_{n-1,1} \dots a_{n-1,i-1} & b_{n-1} & a_{n-1,i+1} \dots a_{n-1,n} \\ a_{n1} \dots a_{n,i-1} & b_n & a_{n,i+1} \dots a_{nn} \end{vmatrix}$$

(i-ый столбец матрицы системы заменяется столбцом свободных членов).
Алгоритм решения системы линейных уравнений методом Крамера:

1. Вычислить определитель Δ основной матрицы A.
2. Замена столбца 1 матрицы A на вектор свободных членов b.
3. Вычисление определителя Δ1 полученной матрицы A1.
4. Вычислить переменную x1=Δ1/Δ.
5. Повторить шаги 2−4 для столбцов 2, 3, ..., n матрицы A.

Программа:

```
#include <iostream>
#include <cmath>
using namespace std;
double det(int n,double **a);
void Kramer(int n,double **aMain,double *freeVar,double *answer);
int main()
{
    int n;
    cout<<"Введите размер матриц коэффициентов: ";cin>>n;
    double **aMain;
    aMain= new double* [n];
    double *freeVar;
```

```
    freeVar = new double [n];

    double *answer = new double [n];
    for(int i=0;i<n;i++)
        aMain[i]=new double [n];

    for(int i=0;i<n;i++)
    for(int j=0;j<n;j++)
            {cout<<"A["<<i<<"]["<<j<<"]=";cin>>aMain[i][j];}

    cout<<endl<<"Введите коэффициенты вектора правых частей:"<<endl;
    for(int i=0;i<n;i++)
    {
        cout<<"b["<<i<<"]=";cin>>freeVar[i];
    }

    Kramer(n,aMain,freeVar,answer);

    delete [] aMain;
    delete [] freeVar;
    delete [] answer;
    return 0;

}

double det(int n, double **mat)
{
    double d=0;
    int c, subi, i, j, subj;
    double **submat;
    submat= new double*[n];
    for(int i=0;i<n;i++)
        submat[i]= new double [n];

    if (n == 2)
    {
        return( (mat[0][0] * mat[1][1]) - (mat[1][0] * mat[0][1]));
    }
    else
    {
        for(c = 0; c < n; c++)
        {
            subi = 0;
            for(i = 1; i < n; i++)
            {
                subj = 0;
                for(j = 0; j < n; j++)
                {
                    if (j == c)
                    {
                        continue;
```

```
                    }
                    submat[subi][subj] = mat[i][j];
                    subj++;
                }
                subi++;
            }
        d = d + (pow(-1 ,c) * mat[0][c] * det(n - 1 ,submat));
        }
    }
    delete [] submat;
    return d;
}

void Kramer(int n,double **aMain,double *freeVar,double *answer)
{
    double *detArray = new double [n];
    double deterMain;
    double *temp= new double [n];
     cout<<"Определитель:"<<endl;
    cout<<"detM = ";deterMain=det(n,aMain);cout<<deterMain<<endl;

    for(int i=0;i<n;i++)
        {for(int j=0;j<n;j++)
    {
        temp[j]=aMain[i][j];
        aMain[i][j]=freeVar[j];

    }
    detArray[i]=det(n,aMain);
    for(int k=0;k<n;k++)
    {
        aMain[i][k]=temp[k];
    }
        }

        for(int i=0;i<n;i++)
        {answer[i]=detArray[i]/deterMain;
        cout<<answer[i]<<endl;
        }
        delete [] detArray;
        delete [] temp;

}
```

Пример вывода:

```
1)
Введите размер матриц коэффициентов: 2
A[0][0]=0.1
A[0][1]=0.9
A[1][0]=3
A[1][1]=1
```

```
Введите коэффициенты вектора правых частей:
b[0]=1
b[1]=4
Определитель:
detM = -2.6
4.23077
0.192308

2)
Введите размер матриц коэффициентов: 3
A[0][0]=234 A[0][1]=675 A[0][2]=844 A[1][0]=324 A[1][1]=66666
A[1][2]=3222 A[2][0]=76 A[2][1]=433 A[2][2]=23
Введите коэффициенты вектора правых частей: b[0]=11 b[1]=234 b[2]=6665
Определитель: detM = -3.96522e+09
8.26609
0.0865772
-25.6752
```

Выводы:
Мы научились реализовывать алгоритм решения СЛУ методом Крамера.

5.2 Метод Гаусса

Задание:
Разработать алгоритм и написать программу, реализующую: решение системы линейных уравнений, на множестве вещественных чисел, методом Гаусса. Предусмотреть возможность решения уравнений различного порядка без перекомпиляции программы. Исходные данные (матрицу коэффициентов уравнения, вектор правых частей), вводить в программу из консоли или из файла.

Теория:

Кратко говоря, алгоритм заключается в последовательном исключении переменных из каждого уравнения до тех пор, пока в каждом уравнении не останется только по одной переменной. Если n=m, то можно говорить, что алгоритм Гаусса-Жордана стремится привести матрицу A системы к единичной матрице — ведь после того, как матрица стала единичной, решение системы очевидно — решение единственно и задаётся получившимися коэффициентами b_i.

При этом алгоритм основывается на двух простых эквивалентных преобразованиях системы: во-первых, можно обменивать два уравнения, а во-вторых, любое уравнение можно заменить линейной комбинацией этой строки (с ненулевым коэффициентом) и других строк (с произвольными коэффициентами).

На первом шаге алгоритм Гаусса-Жордана делит первую строку на коэффициент a_{11}. Затем алгоритм прибавляет первую строку к остальным строкам с такими коэффициентами, чтобы их коэффициенты в первом столбце обращались в нули — для этого, очевидно, при прибавлении первой строки к i-ой надо домножать её на a_{i1}. При каждой операции с матрицей A (деление на число, прибавление к одной строке другой) соответствующие операции производятся и с вектором b; в некотором смысле, он ведёт себя, как если бы он был m+1-ым столбцом матрицы A.

В итоге, по окончании первого шага первый столбец матрицы A станет единичным (т. е. будет содержать единицу в первой строке и нули в остальных).

Аналогично производится второй шаг алгоритма, только теперь рассматривается второй столбец и вторая строка: сначала вторая строка делится на a_{22}, а затем отнимается от всех остальных строк с такими коэффициентами, чтобы обнулять второй столбец матрицы A.

И так далее, пока мы не обработаем все строки или все столбцы матрицы A. Если n=m, то по построению алгоритма очевидно, что матрица A получится единичной, что нам и требовалось.

Программа:
```
#include <bits/stdc++.h>

using namespace std;

typedef vector<vector<double>> real_matrix;
const double EPS = 1E-9;

void input_matrix(real_matrix &arr, int n) {
    cout << "Вводите матрицу" << endl;
    arr.resize(n);
    for (int i = 0; i < n; i++) {
        cout << "Введите строчку " << to_string(i) << endl;
        arr[i].resize(n);
```

```
        for (int j = 0; j < n; j++)
            cin >> arr[i][j];
    }
}

double determinant(real_matrix a, int n) {
    double d = 1;
    for (int i = 0; i < n; ++i) {
        int k = i;
        for (int j = i + 1; j < n; ++j)
            if (abs(a[j][i]) > abs(a[k][i]))
                k = j;
        if (abs(a[k][i]) < EPS) {
            d = 0;
            break;
        }
        swap(a[i], a[k]);
        if (i != k)
            d = -d;
        d = a[i][i] * d;
        for (int j = i + 1; j < n; ++j)
            a[i][j] /= a[i][i];
        for (int j = 0; j < n; ++j)
            if (j != i && abs(a[j][i]) > EPS)
                for (int k = i + 1; k < n; ++k)
                    a[j][k] -= a[i][k] * a[j][i];
    }
    return d;
}

vector<double> gauss_method(real_matrix a, vector<double> b, int n) {
    double max;
    int index;
    int k = 0;
    vector<double> x(n);

    while (k < n) {
        max = abs(a[k][k]);
        index = k;
        for (int i = k + 1; i < n; i++) {
            if (abs(a[i][k]) > max) {
                max = abs(a[i][k]);
                index = i;
            }
        }
        if (max < EPS) {
            cout << "Решение получить невозможно" << endl;
            return vector<double>(0, 0);
        }
        for (int j = 0; j < n; j++) {
            double temp = a[k][j];
```

```
            a[k][j] = a[index][j];
            a[index][j] = temp;
        }

        double t = b[k];
        b[k] = b[index];
        b[index] = t;
        for (int i = k; i < n; i++) {
            t = a[i][k];
            if (abs(t) < EPS) continue;
            for (int j = 0; j < n; j++)
                a[i][j] = a[i][j] / t;
            b[i] = b[i] / t;
            if (i == k) continue;
            for (int j = 0; j < n; j++)
                a[i][j] = a[i][j] - a[k][j];
            b[i] = b[i] - b[k];
        }
        k++;
    }

    for (k = n - 1; k >= 0; k--) {
        x[k] = b[k];
        for (int i = 0; i < k; i++)
            b[i] = b[i] - a[i][k] * x[k];
    }
    return x;
}

int main() {
    int n;
    cout << "Введите n: " << endl;
    cin >> n;

    real_matrix a;
    input_matrix(a, n);

    vector<double> b;
    b.resize(n);
    cout << "Введите вектор свободных членов: " << endl;
    for (int i = 0; i < n; i++)
        cin >> b[i];

    double det = determinant(a, n);
    cout << "Определитель: " << det << endl;

    vector<double> x = gauss_method(a, b, n);
    for (int i = 0; i < x.size(); i++)
        cout << "x" << to_string(i) << " = " << x[i] << endl;
}
```

Пример вывода:
1)
Введите n:
3
Вводите матрицу
Введите строчку 0
123
4
5
Введите строчку 1
4
5
6
Введите строчку 2
4
6
5
Введите вектор свободных членов:
3
4
6
Определитель: -1317
x0 = -0.000759301
x1 = 1.45482
x2 = -0.545178

2)
Введите n:
2
Вводите матрицу
Введите строчку 0
3.76
5.99
Введите строчку 1
900
43
Введите вектор свободных членов:
1
3
Определитель: -5229.32
x0 = -0.00478647
x1 = 0.169949

Вывод:

Оценим асимптотику полученного алгоритма. Алгоритм состоит из m фаз, на каждой из которых происходит:

1. поиск и перестановка опорного элемента — за время O(n+m)
2. если опорный элемент в текущем столбце был найден — то прибавление текущего уравнения ко всем остальным уравнениям — за время O(nm).

Очевидно, первый пункт имеет меньшую асимптотику, чем второй. Заметим также, что второй пункт выполняется не более min(n, m) раз — столько, сколько может быть зависимых переменных в СЛАУ. Таким образом, итоговая асимптотика алгоритма принимает вид O (min(n, m)*nm). При n=m эта оценка превращается в O (n^3).

5.3 Схема Холецкого

Задание:
Разработать алгоритм и написать программу, реализующую: решение системы линейных уравнений, на множестве вещественных чисел, методом схемы Холецкого. Предусмотреть возможность решения уравнений различного порядка без перекомпиляции программы. Исходные данные (матрицу коэффициентов уравнения, вектор правых частей), вводить в программу из консоли или из файла.

Теория:
Пусть дана положительно определённая симметрическая матрица A (элементы a_{ij}). Тогда необходимо найти нижнюю треугольную матрицу L (элементы l_{ij}). Разложение Холецкого определяется формулами:

$$l_{11} = \sqrt{a_{11}}$$

$$l_{j1} = \frac{a_{j1}}{l_{11}}, j \in [2, n],$$

$$l_{ii} = \sqrt{a_{ii} - \sum_{p=1}^{i-1} l_{ip}^2}, \qquad i \in [2, n],$$

$$l_{ji} = \frac{a_{ji} - \sum_{p=1}^{i-1} l_{ip} l_{jp}}{l_{ii}} \qquad i \in [2, n-1], j \in [i+1, n].$$

После этого вектор корней СЛАУ X находится как решение системы уравнений LY = B, $L^T X = Y$

Программа:
```
#include <bits/stdc++.h>

using namespace std;

typedef vector<vector<double>> real_matrix;

void input_matrix(real_matrix &arr, int n) {
    cout << "Вводите матрицу" << endl;
    arr.resize(n);
    for (int i = 0; i < n; i++) {
        cout << "Введите строчку " << to_string(i) << endl;
        arr[i].resize(n);
        for (int j = 0; j < n; j++)
            cin >> arr[i][j];
    }
}

void transpose(real_matrix & from, real_matrix & to, int n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            to[j][i] = from[i][j];
}

void output_matrix(real_matrix & a) {
```

```
    for (int i = 0; i < a.size(); i++) {
        for (int j = 0; j < a[i].size(); j++)
            cout << a[i][j] << '\t';
        cout << endl;
    }
    cout << endl;
}

vector<double> choletsky_method(real_matrix a, vector<double> b, int
n) {
    double pk, sum;
    real_matrix L(n, vector<double>(n, 0.0));
    real_matrix T(n, vector<double>(n, 0.0));
    vector<double> x(n), y(n);

    L[0][0] = sqrt(a[0][0]);
    for (int i = 1; i < n; i++)
        L[i][0] = a[i][0] / L[0][0];

    for (int i = 1; i < n; i++) {
        sum = 0;
        for (int k = 0; k < i; k++)
            sum += L[i][k] * L[i][k];
        pk = a[i][i] - sum;
        L[i][i] = sqrt(pk);

        for (int j = i + 1; j < n; j++) {
            sum = 0;
            for (int k = 0; k < i; k++)
                sum += L[j][k] * L[i][k];
            pk = a[j][i] - sum;
            L[j][i] = pk / L[i][i];
        }
    }

    transpose(L, T, n);

    cout << "Матрица L: " << endl;
    output_matrix(L);
    cout << "Матрица T: " << endl;
    output_matrix(T);

    // Решение L*y=b
    y[0] = b[0] / L[0][0];
    for (int i = 1; i < n; i++) {
        sum = 0;
        for (int k = 0; k < i; k++)
            sum += L[i][k] * y[k];
        pk = b[i] - sum;
        y[i] = pk / L[i][i];
    }
```

```
    // Решение T*x=y
    x[n - 1] = y[n - 1] / T[n - 1][n - 1];
    for (int i = n - 2; i >= 0; i--) {
        sum = 0;
        for (int k = i + 1; k < n; k++)
            sum += T[i][k] * x[k];
        pk = y[i] - sum;
        x[i] = pk / T[i][i];
    }

    return x;
}

int main() {
    int n;
    cout << "Введите n: " << endl;
    cin >> n;

    real_matrix a;
    input_matrix(a, n);

    vector<double> b;
    b.resize(n);
    cout << "Введите вектор свободных членов: " << endl;
    for (int i = 0; i < n; i++)
        cin >> b[i];

    vector<double> x = choletsky_method(a, b, n);
    for (int i = 0; i < x.size(); i++)
        cout << "x" << to_string(i) << " = " << x[i] << endl;
}
```

Пример вывода:
Введите n:
3
Вводите матрицу
Введите строчку 0
81
45
-45
Введите строчку 1
-45
50
-15
Введите строчку 2
45
-15
38
Введите вектор свободных членов:
531
-460

193
Матрица L:
9 0 0
-5 5 0
5 2 3

Матрица T:
9 -5 5
0 5 2
0 0 3

x0 = 6
x1 = -5
x2 = -4

Вывод:
Мы ознакомились с методом решения схемой Холецкого.

5.4 Метод итераций

Задание:
Разработать алгоритм и написать программу, реализующую: решение системы линейных уравнений, на множестве вещественных чисел, методом итераций. Предусмотреть возможность решения уравнений различного порядка без перекомпиляции программы. Исходные данные (матрицу коэффициентов уравнения, вектор правых частей), вводить в программу из консоли или из файла. Оценить вычислительную сложность решения задачи.

Теория:
Прежде чем применять метод итераций, необходимо переставить строки исходной системы таким образом, чтобы на диагонали стояли наибольшие по модулю коэффициенты матрицы.
Имеем СЛАУ $AX = B$
Предполагая, что $a_{ii} \neq 0$ разрешим новое уравнение системы относительно x_1, второе – относительно x_2,..., n-ое уравнение – относительно x_n. В результате получим:

$$\{x_1 = \beta_1 - \alpha_{12}x_2 - \alpha_{13}x_3 - \cdots - \alpha_{1n}x_n \; x_2 = \beta_2 - \alpha_{21}x_1 - \alpha_{23}x_3 - \cdots - \alpha_{2n}x_n \; x_n = \beta_n - \alpha_{n1}x_n - \alpha_{n3}x_3 - \cdots - \alpha_{nn-1}x_{n-1}$$

где $\beta_i = \frac{b_i}{a_{ii}}$; $\alpha_{ij} = \frac{a_{ij}}{a_{ii}}$ при i ≠ j; $\alpha_{ii} = 0$

Система в матричной форме имеет вид $X = \beta - \alpha X$
Систему будем решать методом последовательных приближений. Примем за первое приближение $X^{(0)} = \beta$, тогда последующие:

$$X^{(1)} = \beta + \alpha X^{(0)}$$
$$X^{(2)} = \beta + \alpha X^{(1)}$$
$$\cdots$$
$$X^{(k+1)} = \beta + \alpha X^{(k)}$$

Оценка погрешности:

$$\varepsilon = \left\| x - x^{(k)} \right\| \leq \frac{\|\alpha\|^{k+1}\|\beta\|}{1 - \|\alpha\|}$$
$$\varepsilon = \|x - x\^{(k)}\| \leq \|\alpha\|/(1 - \|\alpha\|) * \|x\^(k) - x\^(k-1)\|$$

Программа:
```
#include <bits/stdc++.h>

using namespace std;

typedef vector<vector<double>> real_matrix;
const double EPS = 1E-9;

void input_matrix(real_matrix &arr, int n) {
    cout << "Вводите матрицу" << endl;
    arr.resize(n);
    for (int i = 0; i < n; i++) {
        cout << "Введите строчку " << to_string(i) << endl;
        arr[i].resize(n);
```

```
        for (int j = 0; j < n; j++)
            cin >> arr[i][j];
    }
}

vector<double> iterations_method(real_matrix a, vector<double> b, int
n) {
    vector<double> previous_x(n, 0.0);
    for (int i = 0; i < n; i++)
        previous_x[i] = b[i];

    while (true) {
        vector<double> current_x(n);

        for (int i = 0; i < n; i++) {
            current_x[i] = b[i];
            for (int j = 0; j < n; j++)
                if (i != j)
                    current_x[i] -= a[i][j] * previous_x[j];
            current_x[i] /= a[i][i];
        }

        double error = 0.0;
        for (int i = 0; i < n; i++)
            error += abs(current_x[i] - previous_x[i]);
        if (error < EPS)
            break;

        previous_x = current_x;
    }

    return previous_x;
}

int main() {
    int n;
    cout << "Введите n: " << endl;
    cin >> n;

    real_matrix a;
    input_matrix(a, n);

    vector<double> b;
    b.resize(n);
    cout << "Введите вектор свободных членов: " << endl;
    for (int i = 0; i < n; i++)
        cin >> b[i];

    vector<double> x = iterations_method(a, b, n);

    for (int i = 0; i < x.size(); i++)
```

```
        cout << "x" << to_string(i) << " = " << x[i] << endl;
}
```

Пример вывода

```
Введите n:
3
Вводите матрицу
Введите строчку 0
10
2
-1
Введите строчку 1
-2
-6
-1
Введите строчку 2
1
-3
12
Введите вектор свободных членов:
5
24.42
36
x0 = 1.64221
x1 = -4.89081
x2 = 1.64045
```

Вывод:

Метод итерации применяют в случае, если сходится последовательность приближений по указанному алгоритму

5.5 Метод Зейделя

Задание:
Разработать алгоритм и написать программу, реализующую: решение системы линейных уравнений, на множестве вещественных чисел, методом Зейделя. Предусмотреть возможность решения уравнений различного порядка без перекомпиляции программы. Исходные данные (матрицу коэффициентов уравнения, вектор правых частей), вводить в программу из консоли или из файла.

Теория:
Метод Зейделя представляет собой некоторую модификацию метода простой итерации. Основная его идея заключается в том, что при вычислении (k+1)-го приближения неизвестной x_i учитываются уже вычисленные ранее (k+1)-е приближения неизвестных x_1, x_2, ... x_{i-1}
Пусть дана система из трех линейных уравнений. Представим ее так, чтобы в левой части i-го уравнения была только i-я неизвестная. Выберем произвольно начальные приближения корней, стараясь, чтобы они в какой-то мере соответствовали искомым неизвестным. За нулевое приближение можно принять столбец свободных членов, т. е. $x_1^{(0)}=b_1$, $x_2^{(0)}=b_2$, $x_3^{(0)}=b_3$. Найдем первое приближение $x^{(1)}$:

$$\begin{cases} x_1^{(1)} = \alpha_{12}x_2^{(0)} + \alpha_{13}x_3^{(0)} + \beta_1 \\ x_2^{(1)} = \alpha_{21}x_1^{(1)} + \alpha_{23}x_3^{(0)} + \beta_2 \\ x_3^{(1)} = \alpha_{31}x_1^{(1)} + \alpha_{32}x_2^{(1)} + \beta_3 \end{cases}$$

Следует обратить внимание на особенность метода Зейделя, которая состоит в том, что полученное в первом уравнении значение $x_1^{(l)}$ сразу же используется во втором уравнении, а значения $x_1^{(1)}$, $x_2^{(1)}$ – в третьем уравнении и т. д. То есть все найденные значения $x_1^{(1)}$ подставляются в уравнения для нахождения $x_{i+1}^{(1)}$
Формулы для метода Зейделя в общем виде имеют следующий вид:

$$\begin{cases} x_1^{(k+1)} = \alpha_{12}x_2^{(k)} + \alpha_{13}x_3^{(k)} + \cdots + \alpha_{1n}x_n^{(k)} + \beta_1 \\ x_2^{(k+1)} = \alpha_{21}x_1^{(k+1)} + \alpha_{23}x_3^{(k)} + \cdots + \alpha_{2n}x_n^{(k)} + \beta_2 \\ \cdots \\ x_n^{(k+1)} = \alpha_{n1}x_1^{(k+1)} + \alpha_{n2}x_2^{(k+1)} + \cdots + \alpha_{n,n-1}x_{n-1}^{(k+1)} + \beta_n \end{cases}$$

Программа:

```
#include <bits/stdc++.h>

using namespace std;

typedef vector<vector<double>> real_matrix;
const double EPS = 1E-9;

void input_matrix(real_matrix &arr, int n) {
    cout << "Вводите матрицу" << endl;
    arr.resize(n);
    for (int i = 0; i < n; i++) {
        cout << "Введите строчку " << to_string(i) << endl;
        arr[i].resize(n);
        for (int j = 0; j < n; j++)
            cin >> arr[i][j];
    }
```

```
}

vector<double> Seidel_method(real_matrix a, vector<double> b, int n) {
    vector<double> previous_x(n, 0.0);
    for (int i = 0; i < n; i++)
        previous_x[i] = b[i];

    while (true) {
        vector<double> current_x(n);

        for (int i = 0; i < n; i++) {
            current_x[i] = b[i];
            for (int j = 0; j < n; j++) {
                if (j < i)
                    current_x[i] -= a[i][j] * current_x[j];
                if (j > i)
                    current_x[i] -= a[i][j] * previous_x[j];
            }
            current_x[i] /= a[i][i];
        }

        double error = 0.0;
        for (int i = 0; i < n; i++)
            error += abs(current_x[i] - previous_x[i]);
        if (error < EPS)
            break;

        previous_x = current_x;
    }

    return previous_x;
}

int main() {
    int n;
    cout << "Введите n: " << endl;
    cin >> n;

    real_matrix a;
    input_matrix(a, n);

    vector<double> b;
    b.resize(n);
    cout << "Введите вектор свободных членов: " << endl;
    for (int i = 0; i < n; i++)
        cin >> b[i];

    vector<double> x = Seidel_method(a, b, n);

    for (int i = 0; i < x.size(); i++)
        cout << "x" << to_string(i) << " = " << x[i] << endl;
```

```
}
```
Пример вывода:

```
Введите n:
3
Вводите матрицу
Введите строчку 0
10
1
-1
Введите строчку 1
1
10
-1
Введите строчку 2
-1
1
10
Введите вектор свободных членов:
11
10
10
x0 = 1.10202
x1 = 0.990909
x2 = 1.01111
```

Вывод:
Условие окончания итерационного процесса Зейделя при достижении точности ε в упрощенной форме имеет вид: $||x^{(k+1)} - x^{(k)}|| \leq \varepsilon$.

6. Векторные преобразования

6.1 Скалярное и векторное произведения векторов

Задание:
Разработать алгоритм и написать программу, реализующую: скалярное и векторное произведение векторов (строк на строки и столбцов на столбцы) из одной матрицы. Матрицу и её размер вводить в программу из консоли или из файла. Предусмотреть возможность выбора размера матрицы без перекомпиляции программы, размер матрицы $N \times N$, где $N = 1,2,3, \ldots$.

Теория:
Пусть в n-мерном пространстве E_n имеем векторы:
$x = (x_1, x_2, \ldots, x_n)$ и $y = (y_1, y_2, \ldots, y_n)$

Скалярным произведением векторов n-мерного пространства называется сумма произведений одноименных координат.
Векторным произведением двух векторов называется такой вектор, равный по величине значению мнемонического определителя матрицы, составленной из этих же двух исходных векторов.

$$[\vec{a}, \vec{b}] = \begin{vmatrix} i & j & k \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}, \text{где } i = (1,0,0), j = (0,1,0), k = (0,0,1)$$

Программа:

```
#include <bits/stdc++.h>

using namespace std;

typedef vector<vector<double>> real_matrix;

void input_matrix(real_matrix &arr, int n) {
    arr.resize(n);
    arr[0].resize(n);
    for (int j = 0; j < n; j++)
        arr[0][j] = 1;
    for (int i = 1; i < n; i++) {
        cout << "Введите вектор " << to_string(i) << endl;
        arr[i].resize(n);
        for (int j = 0; j < n; j++)
            cin >> arr[i][j];
    }
}

double scalar_multiply(const real_matrix & mat, int n) {
    double result, t;
    for (int j = 0; j < n; j++) {
        t = 1;
        for (int i = 0; i < n; i++)
            t *= mat[i][j];
```

```
        result += t;
    }
    return result;
}

double vector_multiply(const real_matrix & mat, int n) {
    double a;
    real_matrix submat(n, vector<double>(n));

    if (n == 2) {
        return ((mat[0][0] * mat[1][1]) - (mat[1][0] * mat[0][1]));
    }
    else {
        for (int k = 0; k < n; k++) {
            int subi = 0;
            for (int i = 1; i < n; i++) {
                int subj = 0;
                for (int j = 0; j < n; j++) {
                    if (j == k)
                        continue;
                    submat[subi][subj] = mat[i][j];
                    subj++;
                }
                subi++;
            }
            a = (pow(-1, k) * mat[0][k] * vector_multiply(submat, n -
1));
            cout << a << " ";
        }
        return 0;
    }
}

int main() {
    int n;
    cout << "Введите n: " << endl;
    cin >> n;

    real_matrix a;
    input_matrix(a, n);

    cout << endl << "Скалярное произведение: " << scalar_multiply(a,
n);
    cout << endl << "Векторное произведение: ";
    vector_multiply(a, n);

    return 0;
}
```

Пример вывода:

Введите n:
4
Введите вектор 1
1 -9 0 6
Введите вектор 2
33 0.1 5
6
Введите вектор 3
3 7 9 12

Скалярное произведение: 524.7
Векторное произведение: -54 0 -204.6 0 6 -0 1692 -0 -40.8 3402 1384.2 0 -34.1 2538 0 -0

Вывод: было изучено скалярное и векторное произведение векторов, а также реализована программа.

6.2 Ортогонализация методом Грама-Шмидта

Задание: разработать и написать программу реализующую ортогонализацию методом Грама-Шмидта строк из одной матрицы. Матрицу и её размер вводить в программу из консоли или из файла. Предусмотреть возможность выбора размера матрицы без перекомпиляции программы, размер матрицы N на N, где N = 1,2, 3…. Выполнить проверку ортогональности и нормировки с помощью вычисления скалярного произведения и расчёта длин векторов.

Теория:

Вещественное линейное пространство E называется *евклидовым пространством,* если каждой упорядоченной паре элементов x и y из E поставлено в соответствие вещественное число (x, y) , называемое *скалярным произведением*, так, что выполнены аксиомы:

1. $\forall x,y \in E, (x,y)=(y,x)$;
2. $\forall x,y \in E, \lambda \in R, (\lambda x,y)=\lambda(x,y)$;
3. $\forall x1,x2,y \in E, (x1+x2,y)=(x1,y)+(x2,y)$;
4. $\forall x \in E, x \neq o, (x,x)>0$.

Необходимое условие для ортогональности векторов — два вектора $\vec{a}$ и $\vec{b}$ являются ортогональными (перпендикулярными), если их скалярное произведение равно нулю: $\vec{a} \times \vec{b} = 0$. Если длина вектора равна единице, он называется нормированным вектором: $(x, x) = 1$, $|x| = 1$. Если векторы системы векторов $e_1, e_2, ..., e_n$ попарно ортогональны и нормированы, то система векторов называется ортонормированной системой: $(e_i, e_j) = 0$

Суть метода Грама-Шмидта состоит во взятии первого ортогонального вектора равным первому исходному вектору и построении каждого нового ортогонального вектора равным текущему исходному вектору, скорректированному на величины проекций текущего вектора на предыдущие ортогональные векторы.

Пусть $f = (f_1 \dots f_n)$ - некоторый базис в n-мерном евклидовом пространстве E.

Модифицируя этот базис, мы будем строить новый базис $e = (e_1 \dots e_n)$, который будет ортонормированным. Последовательно вычисляем векторы g_1 и e_1, g_2 и e_2 и т. д. по формулам (1):

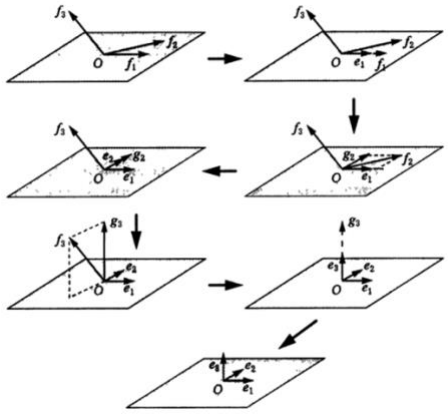
$$g_1 = f_1, \quad e_1 = \frac{g_1}{||g_1||}$$

$$\dots$$

$$g_n = f_n - (f_n, e_1)e_1 - \dots - (f_n, e_{n-1})e_{n-1}, \quad e_n = \frac{g_n}{||g_n||}$$

Для обоснования алгоритма нужно показать, что ни один из последовательно вычисляемых векторов g_i не является нулевым вектором (иначе процесс оборвался бы преждевременно) и что все векторы g_i , $i = 1, n$, попарно ортогональны. Тогда и векторы e_i, i = 1, n, образуют ортогональную систему, но при этом норма каждого из этих векторов равна единице. Ортогональная система из n ненулевых векторов, линейно независима и поэтому в n-мерном евклидовом пространстве является базисом.

Доказательство опирается на метод математической индукции. В соответствии с этим методом мы будем доказывать, что для любого k, $k = 1, n$, векторы $e_1, \dots, e_k$ образуют ортогональную систему и длины их равны единице.



Геометрическая иллюстрация последовательности вычислений при n = 3 (линейное пространство V_3)

Это утверждение очевидно при $k = 1$, так как в этом случае вектор g_1 ненулевой, потому что равен вектору $f_1/||f_1||$ единичной длины, а систему векторов, состоящую из одного вектора, считают ортогональной по определению.
Пусть векторы $e_1, ..., e_k$ образуют ортогональную систему. Вычислим новый вектор g_{k+1} по формуле
$g_{k+1} = f_{k+1} - (f_{k+1}, e_1)\, e_1 - ... - (f_{k+1}, e_k)\, e_k$. (2)
Предположив, что $g_{k+1} = 0$, заключаем, что вектор f_{k+1} является линейной комбинацией векторов $e_1 ..., e_k$, которые в силу (1) выражаются через векторы $f_1, ..., f_k$. Следовательно, этот вектор является линейной комбинацией системы векторов $f_1, ..., f_k$, а система векторов $f_1, ..., f_k, f_{k+1}$ линейно зависима. Но это противоречит условию линейной независимости системы $f_1, ..., f_n$.
Итак, предположение о том, что $g_{k+1} = 0$, привело к противоречию и потому неверно. Нам остается убедиться, что вектор g_{k+1} ортогонален каждому из векторов $e_1, ..., e_k$. Умножим равенство (2) скалярно на вектор e_i, где $i \le k$. Учитывая, что векторы e_j попарно ортогональны при $j \le k$, получим:
$(g_{k+1}, e_i) = (f_{k+1}, e_i) - (f_{k+1}, e_i)\,(e_i\, e_i) = (f_{k+1}, e_i) - (f_{k+1}, e_i) = 0$,
так как $(e_i, e_i) = 1$. Следовательно, векторы $e_1, ..., e_k, e_{k+1}$, где $e_{k+1} = g_{k+1}/||g_{k+1}||$ образуют ортогональную систему векторов и имеют единичную длину.
В конечномерном евклидовом пространстве существует ортонормированный базис.
При практических применениях процесс Грама - Шмидта удобно модифицировать так, чтобы ограничиться вычислением векторов g_i и не использовать их нормированные варианты e_i.
В этом случае нужно последовательно вычислить векторы $g_1, ..., g_n$, а затем провести их нормировку, приводящую к векторам e_i. Чтобы модифицировать алгоритм вычислений, заменим векторы e_i на g_i согласно формулам.
Получим:

$$g_1 = f_1$$

$$g_2 = f_2 - \frac{(f_2, g_1)}{||g_1||^2} g_1$$

$$\dots$$

$$g_n = f_2 - \frac{(f_n, g_1)}{||g_1||^2} g_1 - \frac{(f_n, g_1)}{||g_2||^2} g_2 - \dots - \frac{(f_n, g_{n-1})}{||g_{n-1}||^2} g_{n-1}$$

Для реализации программы используем:
Пусть дана система из n ЛНЗ векторов $\{a\}_{i=1}^{n}$

Тогда векторы результирующей системы $\{b\}_{j=1}^{n}$ будут выражаться как:

$$b_n = a_n - \sum_{i=1}^{n-1} proj_{b_i} a_n$$

Где $proj_b a$ - проекция вектора a на вектор b, которую можно вычислить по формуле:

$$proj_b a = \frac{\langle a, b \rangle}{\langle b, b \rangle} b$$

Программа:

```
#include <iostream>
#include <cmath>
#include <vector>

using namespace std;

vector<double> operator-(const vector<double> & l, const
vector<double> & r) {
    vector<double> result;
    for (int i = 0; i < l.size(); i++)
        result.push_back(l[i] - r[i]);
    return result;
}

vector<double> operator+(const vector<double> & l, const
vector<double> & r) {
    vector<double> result;
    for (int i = 0; i < l.size(); i++)
        result.push_back(l[i] + r[i]);
    return result;
}

double operator*(const vector<double> & l, const vector<double> & r) {
    double result = 0;
    for (int i = 0; i < l.size(); i++)
        result += l[i] * r[i];
    return result;
}

vector<double> operator*(double l, const vector<double> & r) {
    vector<double> result;
    for (double i : r) result.push_back(l * i);
    return result;
}

vector<double> proj(const vector<double> & a, const vector<double> &
b) {
    return ((a * b) / (b * b)) * b;
}

int main() {
```

```
    vector<vector<double>> arr;

    int n;
    cout <<" " <<endl<< "Введите n: " << endl;
    cin >> n;

    for (int i = 0; i < n; i++) {
        vector<double> t;
        cout << "Введите значения для вектора " << i << endl;
for (int j = 0; j < n; j++) {
            double x;
            cin >> x;
            t.push_back(x);
        }
        arr.push_back(t);
    }

    vector<vector<double>> result;
    for (auto & a : arr) {
        vector<double> t = a;
        for (auto & b : result)
            t = t - proj(a, b);
        result.push_back((1.0/sqrt(t * t) * t));
    }

    for (auto & i : result) {
        for (auto & j : i)
            cout << j << " ";
        cout << endl;
    }
cout <<"Проверка нормировки: "<< endl;
    for (auto a : result){
        cout <<  sqrt(a*a) << endl;
    }

    cout <<"Проверка ортогональности: "<< endl;
for (int i = 0; i< result.size(); i++) {
    for(int j=0; j<result.size(); j++){
        if (i!=j) cout << result[i] * result[j]<< endl;
    }
}
    return 0;
}
```

Пример вывода:

Введите n:
3
Введите значения для вектора 0
1 -1 2.1
Введите значения для вектора 1
5 8 -2
Введите значения для вектора 2

3 9 2
0.394976 -0.394976 0.82945
0.664501 0.746272 0.0389389
-0.634376 0.53579 0.557222
Проверка нормировки:
1
1
1
Проверка ортогональности:
 4.16334e-17
-1.11022e-16
4.16334e-17
4.05925e-16
-1.11022e-16
4.05925e-16

Введите n:
4
Введите значения для вектора 0
5 3 6 -9
Введите значения для вектора 1
0.9 8.5 4 2
Введите значения для вектора 2
-3 88 2 4
Введите значения для вектора 3
-1 0.4 3 5
0.406894 0.244137 0.488273 -0.73241
-0.0317755 0.846985 0.279566 0.451053
-0.205989 0.471426 -0.732589 -0.445689
-0.889379 -0.0277547 0.383073 -0.247969
Проверка нормировки:
1 1 1 1
Проверка ортогональности:
5.55112e-17
0
-1.11022e-16
5.55112e-17
-5.55112e-17
8.18789e-16
0
-5.55112e-17
-2.77556e-17
-1.11022e-16
8.18789e-16
-2.77556e-17

Выводы:
В результате проверки на ортогональность из-за использования типа double возникает погрешность.

6.3 Преобразование Фурье

Задание:
Разработать алгоритм и написать программу, реализующую: преобразование Фурье для разложения последовательности комплекснозначных чисел в ряд Фурье. Длина последовательности произвольная.

Теория:
Преобразование Фурье — операция, сопоставляющая одной функции вещественной переменной другую функцию вещественной переменной. Эта новая функция описывает коэффициенты («амплитуды») при разложении исходной функции на элементарные составляющие — гармонические колебания с разными частотами (подобно тому, как музыкальный аккорд может быть выражен в виде суммы музыкальных звуков, которые его составляют). Преобразование Фурье функции f вещественной переменной является интегральным и задаётся следующей формулой:

$$\hat{f}(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} f(x) e^{-i\omega t}\, dx$$

Разные источники могут давать определения, отличающиеся от приведённого выше выбором коэффициента перед интегралом, а также знака «−» в показателе экспоненты.

Программа:

```
#include <complex>
#include <iostream>
#include <valarray>

using namespace std;
const double PI = 3.14159265358979323460;
typedef complex<double> Complex;
typedef valarray<Complex> CArray;
void Fast_Fourier_transform(CArray& x)
{
    const size_t N = x.size();
    if (N <= 1) return;
    CArray even = x[slice(0, N/2, 2)];
    CArray  odd = x[slice(1, N/2, 2)];
    Fast_Fourier_transform(even);
    Fast_Fourier_transform(odd);
    for (size_t k = 0; k < N/2; ++k)
    {
        Complex t = polar(1.0, -2 * PI * k / N) * odd[k];
        x[k    ] = even[k] + t;
        x[k+N/2] = even[k] - t;
    }
}
void ifft(CArray& x)
{
    x = x.apply(conj);
    Fast_Fourier_transform(x);
    x = x.apply(conj);
    x /= x.size();
```

```
}

int main()
{
    const Complex test[] = { 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0 };
    CArray data(test, 8);
    Fast_Fourier_transform(data);
    cout << "Быстрое преобразование Фурье:" << endl;
    for (int i = 0; i < 8; ++i)
    {
        cout << data[i] << endl;
    }
    ifft(data);
    cout << endl << "Обратное преобразование Фурье:" << endl;
    for (int i = 0; i < 8; ++i)
    {
        cout << data[i] << endl;
    }
    return 0;
}
```

Пример вывода:
Быстрое преобразование Фурье
(3,0)
(4.979e-17,-0.414214)
(-1,0)
(2.83277e-16,-2.41421)
(-1,0)
(-1.72255e-16,2.41421)
(-1,0)
(-1.60812e-16,0.414214)

Обратное преобразование Фурье
(0,-0)
(1,-0)
(0,-6.16298e-33)
(1,-0)
(0,-0)
(1.11022e-16,0)
(1,6.16298e-33)
(0,-0)

Вывод: был изучен метод быстрого и обратного преобразования Фурье.

6.4 Матрица унитарного преобразования Фурье

Задание: разработать алгоритм и написать программу, реализующую: вычисление, вывод в консоль матрицы преобразования Фурье и проверки полученной матрицы на унитарность. Размер матрицы
$N \times N$, где $N = 2^n$.

Теория:
Пару преобразования Фурье:

$$\begin{cases} X[m] = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x[n] e^{\frac{-j2\pi mn}{N}} \\ x[n] = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} X[k] e^{\frac{j2\pi mn}{N}} \end{cases}$$

взаимного отображения сигнала *x[n]* , длительностью *N* отсчетов, и его частотного образа *X[m]* можно интерпретировать как линейное унитарное преобразование в пространстве и записать в матричном виде:

$$X = A_{FT} x$$

$$x = A_{FT}^{*} X$$

где *x* – вектор сигнала в пространстве координат-отсчетов по времени, *X* – вектор сигнала в пространстве координат-отсчетов по частоте. Значения элементов матрицы линейного унитарного преобразования Фурье *AFT*, вычислимы как:

$$A_{FT} = (a_{mn}) = \frac{1}{\sqrt{N}} e^{-j\frac{2\pi mn}{N}}$$

Таким образом, пара преобразования Фурье образует линейное унитарное преобразование координат.
Свойства матрицы *AFT* : симметрична, унитарна.

Программа:

```
#include <bits/stdc++.h>
#define _USE_MATH_DEFINES // for C++
#include <cmath>

using namespace std;

void print_complex(complex <double> c) {
    double a = c.real();
    double b = c.imag();
    string sign = "";
    if(b >= 0)
        sign = "+";
    else
        sign = "-";
    cout << a << sign << "j" << abs(b) << " ";

}

int main(){
    cout << "Введите n" << endl;
```

```
    int l = 2;
    cin >> l;
  int N = pow(2, l);
  cout << "Введите вектор сигнала в пространстве координат-отсчетов по
времени, состоящий из " << N << " элементов" << ":" << endl;
  complex <double> j(0, -1);
  vector <vector<complex<double> > > matrix (N, vector<complex<double>
>(N));
  for(int k = 0; k < N; k++) {
      for(int n = 0; n < N; n++) {
          matrix[k][n] = (1/sqrt(N)) * pow(M_E, (j * (2 * M_PI * k *
n/N)));
          double a = matrix[k][n].real();
          double b = matrix[k][n].imag();
          if(abs(a) < 1e-7)
              a = 0;
          if(abs(b) < 1e-7)
              b = 0;
          complex<double> tmp (a, b);
          matrix[k][n] = tmp;
      }
  }
    vector <vector<complex<double> > > matrix_s (N,
vector<complex<double> >(N));
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            matrix_s[i][j] = conj(matrix[i][j]);
            double a = matrix_s[i][j].real();
            double b = matrix_s[i][j].imag();
            if(abs(a) < 1e-7)
                a = 0;
            if(abs(b) < 1e-7)
                b = 0;
            complex<double> tmp (a, b);
            matrix_s[i][j] = tmp;
        }
        }
    vector <complex<double> >  x (N, (0, 0));
    vector <complex<double> > X (N, (0, 0));
    vector <complex<double> > x_t (N, (0, 0));

    double c;
    for(int i = 0; i < N; i++) {
        cin >> c;
        complex <double> tmp(c, 0);
        x[i] = tmp;
    }
```

```
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
                X[i] += matrix[i][j] * x[j];
                double a = X[i].real();
                double b = X[i].imag();
                if(abs(a) < 1e-7)
                    a = 0;
                if(abs(b) < 1e-7)
                    b = 0;
                complex<double> tmp (a, b);
                X[i] = tmp;

        }

    }
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            x_t[i] += matrix_s[i][j] * X[j];
            double a = x_t[i].real();
            double b = x_t[i].imag();
            if(abs(a) < 1e-7)
                a = 0;
            if(abs(b) < 1e-7)
                b = 0;
            complex<double> tmp (a, b);
            x_t[i] = tmp;

        }

    }

    cout << "Прямое преобразование Фурье: " << endl;
      for(int i = 0; i < N; i++) {
          print_complex(X[i]);
          cout << endl;
      }
     cout << "Обратное преобразование Фурье: " << endl;
    for(int i = 0; i < N; i++) {
        print_complex(x_t[i]);
        cout << endl;
    }
return 0;
}
```

Пример вывода:
Введите n
2
Матрица унитарного преобразования Фурье размерности 4x4:
0.5+j0 0.5+j0 0.5+j0 0.5+j0
0.5+j0 0-j0.5 -0.5+j0 0+j0.5
0.5+j0 -0.5+j0 0.5+j0 -0.5+j0
0.5+j0 0+j0.5 -0.5+j0 0-j0.5
==========
Результат умножения полученной матрицы на эрмитово сопряженную:
1+j0 0+j0 0+j0 0+j0
0+j0 1+j0 0+j0 0+j0
0+j0 0+j0 1+j0 0+j0
0+j0 0+j0 0+j0 1+j0

Введите n
3
Матрица унитарного преобразования Фурье размерности 8x8:
0.353553+j0 0.353553+j0 0.353553+j0 0.353553+j0 0.353553+j0 0.353553+j0 0.353553+j0 0.353553+j0
0.353553+j0 0.25-j0.25 0-j0.353553 -0.25-j0.25 -0.353553+j0 -0.25+j0.25 0+j0.353553 0.25+j0.25
0.353553+j0 0-j0.353553 -0.353553+j0 0+j0.353553 0.353553+j0 0-j0.353553 -0.353553+j0 0+j0.353553
0.353553+j0 -0.25-j0.25 0+j0.353553 0.25-j0.25 -0.353553+j0 0.25+j0.25 0-j0.353553 -0.25+j0.25
0.353553+j0 -0.353553+j0 0.353553+j0 -0.353553+j0 0.353553+j0 -0.353553+j0 0.353553+j0 -0.353553+j0
0.353553+j0 -0.25+j0.25 0-j0.353553 0.25+j0.25 -0.353553+j0 0.25-j0.25 0+j0.353553 -0.25-j0.25
0.353553+j0 0+j0.353553 -0.353553+j0 0-j0.353553 0.353553+j0 0+j0.353553 -0.353553+j0 0-j0.353553
0.353553+j0 0.25+j0.25 0+j0.353553 -0.25+j0.25 -0.353553+j0 -0.25-j0.25 0-j0.353553 0.25-j0.25
==========
Результат умножения полученной матрицы на эрмитово сопряженную:
1+j0 0+j0 0+j0 0+j0 0+j0 0+j0 0+j0 0+j0
0+j0 1+j0 0+j0 0+j0 0+j0 0+j0 0+j0 0+j0
0+j0 0+j0 1+j0 0+j0 0+j0 0+j0 0+j0 0+j0
0+j0 0+j0 0+j0 1+j0 0+j0 0+j0 0+j0 0+j0
0+j0 0+j0 0+j0 0+j0 1+j0 0+j0 0+j0 0+j0
0+j0 0+j0 0+j0 0+j0 0+j0 1+j0 0+j0 0+j0
0+j0 0+j0 0+j0 0+j0 0+j0 0+j0 1+j0 0+j0
0+j0 0+j0 0+j0 0+j0 0+j0 0+j0 0+j0 1+j0

Вывод: интерпретация преобразования Фурье как линейного унитарного преобразования на основе матрицы *AFT*, очень удобно для реализации на электронно-вычислительных машинах. Кроме того, она может быть полезна для понимания сущности алгоритма быстрого преобразования Фурье (БПФ).

6.5 Прямое и обратное преобразование Фурье на основе умножения на матрицу преобразования Фурье

Задание:
Разработать алгоритм и написать программу, реализующую: прямое и обратное преобразование Фурье на основе умножения на матрицу преобразования Фурье. Предусмотреть возможность выбора размера последовательности преобразования без перекомпиляции программы, размер последовательности N, где $N = 2^n, n = 1,2,3, \ldots$.

Теория:
Дискретное преобразование Фурье является линейным преобразованием, которое переводит вектор временных отсчётов $\vec{x}$ в вектор спектральных отсчётов той же длины. Таким образом преобразование может быть реализовано как умножение симметричной квадратной матрицы на вектор:

$$\vec{X} = \hat{A} \cdot \vec{x}$$

Матрица $\hat{A}$:

$$\hat{A} = \begin{pmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & e^{-\frac{2\pi i}{N}} & e^{-\frac{4\pi i}{N}} & e^{-\frac{6\pi i}{N}} & \dots & e^{-\frac{2\pi i}{N}(N-1)} \\ 1 & e^{-\frac{4\pi i}{N}} & e^{-\frac{8\pi i}{N}} & e^{-\frac{12\pi i}{N}} & \dots & e^{-\frac{2\pi i}{N}2(N-1)} \\ 1 & e^{-\frac{6\pi i}{N}} & e^{-\frac{12\pi i}{N}} & e^{-\frac{18\pi i}{N}} & \dots & e^{-\frac{2\pi i}{N}3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & e^{-\frac{2\pi i}{N}(N-1)} & e^{-\frac{2\pi i}{N}2(N-1)} & e^{-\frac{2\pi i}{N}3(N-1)} & \dots & e^{-\frac{2\pi i}{N}(N-1)^2} \end{pmatrix}$$

Элементы матрицы задаются следующей формулой:

$A(m, n) = \frac{1}{\sqrt{N}} exp(-2\pi i \frac{(m-1)(n-1)}{N})$

Программа:

```
#include <bits/stdc++.h>
#define _USE_MATH_DEFINES
#include <cmath>

using namespace std;
```

```
void print_complex(complex <double> c) {
    double a = c.real();
    double b = c.imag();
    string sign = "";
    if(b >= 0)
        sign = "+";
    else
        sign = "-";
    cout << a << sign << "j" << abs(b) << " ";

}

int main(){
    cout << "Введите n" << endl;
    int l = 2;
    cin >> l;
  int N = pow(2, l);
  cout << "Введите вектор сигнала в пространстве координат-отсчетов по
времени, состоящий из " << N << " элементов" << ":" << endl;
  complex <double> j(0, -1);
  vector <vector<complex<double> > > matrix (N, vector<complex<double>
>(N));
  for(int k = 0; k < N; k++) {
      for(int n = 0; n < N; n++) {
          matrix[k][n] = (1/sqrt(N)) * pow(M_E, (j * (2 * M_PI * k *
n/N)));
          double a = matrix[k][n].real();
          double b = matrix[k][n].imag();
          if(abs(a) < 1e-7)
              a = 0;
          if(abs(b) < 1e-7)
              b = 0;
          complex<double> tmp (a, b);
          matrix[k][n] = tmp;
      }
  }
    vector <vector<complex<double> > > matrix_s (N,
vector<complex<double> >(N));
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            matrix_s[i][j] = conj(matrix[i][j]);
            double a = matrix_s[i][j].real();
            double b = matrix_s[i][j].imag();
            if(abs(a) < 1e-7)
                a = 0;
            if(abs(b) < 1e-7)
                b = 0;
            complex<double> tmp (a, b);
            matrix_s[i][j] = tmp;
        }
        }
```

```
vector <complex<double> >  x (N, (0, 0));
vector <complex<double> > X (N, (0, 0));
vector <complex<double> > x_t (N, (0, 0));

double c;
for(int i = 0; i < N; i++) {
    cin >> c;
    complex <double> tmp(c, 0);
    x[i] = tmp;
}

for (int i = 0; i < N; i++)
{
    for (int j = 0; j < N; j++)
    {
            X[i] += matrix[i][j] * x[j];
            double a = X[i].real();
            double b = X[i].imag();
            if(abs(a) < 1e-7)
                a = 0;
            if(abs(b) < 1e-7)
                b = 0;
            complex<double> tmp (a, b);
            X[i] = tmp;

    }

}
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < N; j++)
    {
        x_t[i] += matrix_s[i][j] * X[j];
        double a = x_t[i].real();
        double b = x_t[i].imag();
        if(abs(a) < 1e-7)
            a = 0;
        if(abs(b) < 1e-7)
            b = 0;
        complex<double> tmp (a, b);
        x_t[i] = tmp;

    }

}

cout << "Прямое преобразование Фурье: " << endl;
  for(int i = 0; i < N; i++) {
      print_complex(X[i]);
      cout << endl;
  }
```

```
     cout << "Обратное преобразование Фурье: " << endl;
    for(int i = 0; i < N; i++) {
        print_complex(x_t[i]);
        cout << endl;
    }
return 0;
}
```

Пример вывода:

Введите n
3
Введите вектор сигнала в пространстве координат-отсчетов по времени, состоящий из 8 элементов:
1
2
3
4
5
6
7
8
Прямое преобразование Фурье:
12.7279+j0
-1.41421+j3.41421
-1.41421+j1.41421
-1.41421+j0.585786
-1.41421+j0
-1.41421-j0.585786
-1.41421-j1.41421
-1.41421-j3.41421
Обратное преобразование Фурье:
1+j0
2+j0
3+j0
4+j0
5+j0
6+j0
7+j0
8+j0

Process finished with exit code 0
Введите n
4
Введите вектор сигнала в пространстве координат-отсчетов по времени, состоящий из 16 элементов:
1
2
3
4
5
6
7

8
9
10
11
12
13
14
15
16
Прямое преобразование Фурье:
34+j0
-2+j10.0547
-2+j4.82843
-2+j2.99321
-2+j2
-2+j1.33636
-2+j0.828427
-2+j0.397825
-2+j0
-2-j0.397825
-2-j0.828427
-2-j1.33636
-2-j2
-2-j2.99321
-2-j4.82843
-2-j10.0547
Обратное преобразование Фурье:
1+j0
2+j0
3+j0
4+j0
5+j0
6+j0
7+j0
8+j0
9+j0
10+j0
11+j0
12+j0
13+j0
14+j0
15+j0
16+j0

Вывод: был изучен метод прямого и обратного преобразования Фурье на основе умножения на матрицу преобразования Фурье.

7. Интерполяция и приближение функций

7.1 Интерполяционные формулы Ньютона

Задание:
Разработать алгоритм и написать программу, реализующую: первую интерполяционную формулу Ньютона. Отчёт снабдить графиками. Произвести анализ результатов. Предусмотреть возможность выбора размера последовательности преобразования без перекомпиляции программы, размер последовательности N, где N=5,6,7….

Теория:
Первая интерполяционная формула Ньютона практически неудобна для интерполирования функций в конце таблицы. Поэтому, когда точка интерполирования лежит вблизи точки x_nудобно пользоваться второй интерполяционной формулой Ньютона, которая имеет вид

$$P_n(x) = y_n + \frac{\Delta y_n}{1!\,h}(x - x_n) + \cdots + \frac{\Delta^n y_n}{n!\,h^n}(x - x_n)(x - x_{n-1}) * \ldots * (x - x_1)$$

Вводя новую переменную $q = \frac{x - x_n}{h}$, эту формулу перепишем в виде

$$P_n\big(x(q)\big) = y_n + q\Delta y_{n-1} + \frac{q(q+1)}{2!}\Delta^2 y_{n-2} + \cdots + \frac{q(q+1)(q+2)\ldots(q+n-1)}{n!}\Delta^n y_0$$

В этой формуле из таблицы конечных разностей используются $\Delta^k f_i$ нижней диагонали.

Остаточный член имеет вид

$$R_n(x) = h^n + \frac{q(q+1)\ldots(q+n)}{(n+1)!} f^{(n+1)}(\varepsilon)$$

где точка ε имеет тот же смысл, что и ранее.

Отметим, что формулы Ньютона используются и для экстраполирования функций. Если $\bar{x} < x_0$, то для экстраполирования назад используют первую интерполяционную формулу Ньютона. Если $\bar{x} > x_0$, то для экстраполирования вперед используют вторую интерполяционную формулу Ньютона. Следует заметить, что операция экстраполирования менее точна, чем операция интерполирования в узком смысле.

Программа:

```
#include <iostream>
#include <math.h>
#include <vector>

using namespace std;
struct dat {
  double y;
  double x;
};
int fack(int x) {
  if( x == 0 ) return 1;
  return x * fack(x - 1);
  }
double delta(vector<dat> a, int index, int num) {
  if (index == 1) {
    return a[num+1].y - a[num].y;
    }
  else {
    return delta(a, index-1, num+1) - delta(a, index-1, num);
    }
}

int main() {
  int n;
  vector<dat> arr;
  dat z;
  cout << "Введите число точек: " << endl;
  cin >> n;
  cout << "Введите точки, с разделением координат пробелом: " << endl;
  for(int i = 0; i < n; i++) {
    cin >> z.y >> z.x;
    arr.push_back(z);
  }
  cout << "Введите искомую точку: " << endl;
  double X;
  cin >> X;
  double summ = arr[0].y;
  double step = arr[1].x - arr[0].x;
  double q =  X - arr[0].x / step;
  double thing = q;
  for(int i = 1; i < n; i++) {
    summ += thing*(delta(arr, i, 0))/fack(i);
    thing *= (q - i);
    }
  cout << summ << endl;
}
```

Пример вывода:

Введите число точек:
3

Введите точки, с разделением координат пробелом:
2 3
-1 5
0 7
Введите искомую точку:
5
9

Вывод: был изучен алгоритм, реализующий первую интерполяционную формулу Ньютона.

7.2 Интерполяция Гаусса

Задание:
Разработать алгоритм и написать программу, реализующую: интерполяцию на основе первой и второй интерполяционной формулы Гаусса. Произвести сравнение и анализ результатов. Предусмотреть возможность выбора размера последовательности преобразования без перекомпиляции программы, размер последовательности N, где N=5,6,7….

Теория:
Основным недостатком интерполяционных формул Ньютона является то, что они используют лишь односторонние значения функции. На практике часто оказывается полезным использовать формулы, в которых присутствуют как последующие, так и предыдущие значения функции по отношению к ее начальному значениюy_0 .
Рассмотрим $2n+1$ равноотстоящих узлов $x_{-n}, x_{-n+1}, \dots, x_{-1}, x_0, x_1, x_2, \dots, x_n$, в которых заданы значения некоторой функции $y_i = f(x_i), i = -n, \dots, n$. Требуется найти полином степени не выше, такой, чтобы выполнялось условие $P_{2n}(x_i) = y_i = f(x_i), i = 0, \pm1, \pm2, \dots, \pm n.$

Будем искать полином в виде$P_{2n}(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + a_3(x - x_0)(x - x_1)(x - x_2) + \cdots + a_{2n}(x - x_{-n+1}) \dots (x - x_{-1})(x - x_0)(x - x_1) \dots (x - x_n)$

Поступая по аналогии с выводом первой интерполяционной формулы Ньютона, для коэффициентов a_i получим следующие выражения

$$a_0 = y_0, a_1 = \frac{\Delta y_0}{1!\,h}, a_2 = \frac{\Delta^2 y_{-1}}{2!\,h^2}, a_3 = \frac{\Delta^3 y_{-1}}{3!\,h^3}, \dots, a_{2n} = \frac{\Delta^{2n} y_{-n}}{(2n)!\,h^{2n}}$$

Введем новую переменную $q = \frac{x - x_0}{h}$ и, подставляя преобразованные выражения для коэффициентов (1.20) в соотношение (1.19), получим первую интерполяционную формулу Гаусса (для интерполирования вперёд)

$$P_{2n}(x) = y_0 + q\Delta y_0 + \frac{q(q-1)}{2!}\Delta^2 y_{-1} + \frac{(q+1)q(q-1)}{3!}\Delta^3 y_{-1} + \cdots + \frac{(q+n-1)\dots(q+1)q(q-1)\dots(q-n+1)}{(2n-1)!}\Delta^{2n-1} y_{-n+1} + \frac{(q+n-1)\dots(q+1)q(q-1)\dots(q-n)}{(2n)!}\Delta^{2n} y_{-n}$$

Разности $\Delta y_0, \Delta^2 y_{-1}, \Delta^3 y_{-1}, \Delta^4 y_{-2}, \Delta^5 y_{-2}$, ..., используемые в этой формуле, образуют нижнюю ломаную линию в диагональной таблице разностей 1.2. (см. внизу)

Если полином $P_{2n}(x)$ искать в виде

$$\begin{aligned} P_{2n}(x) = a_0 &+ a_1(x - x_0) + a_2(x - x_{-1})(x - x_0) + a_3(x - x_{-1})(x - x_0)(x - x_1) \\ &+ a_4(x - x_{-2})(x - x_{-1})(x - x_0)(x - x_1) + \cdots \\ &+ a_{2n-1}(x - x_{-n+1}) \dots (x - x_{-1})(x - x_0)(x - x_1) \dots (x - x_{n-1}) \\ &+ a_{2n}(x - x_{-n}) \dots (x - x_{-1})(x - x_0)(x - x_1) \dots (x - x_{n-1}) \end{aligned}$$

то аналогично интерполяционной формуле Гаусса для интерполирования вперед можно получить вторую интерполяционную формулу Гаусса (для интерполирования назад)

$$\begin{aligned} P_{2n}(x) = y_0 + q\Delta y_{-1} + \frac{q(q+1)}{2!}\Delta^2 y_{-1} + \frac{(q+1)q(q-1)}{3!}\Delta^3 y_{-2} + \cdots \\ + \frac{(q+n-1)\dots(q+1)q(q-1)\dots(q-n+1)}{(2n-1)!}\Delta^{2n-1} y_{-n} \\ + \frac{(q+n-1)\dots(q+1)q(q-1)\dots(q-n)}{(2n)!}\Delta^{2n} y_{-n} \end{aligned}$$

Разности $\Delta y_{-1}, \Delta^2 y_{-1}, \Delta^3 y_{-2}, \Delta^4 y_{-2}, \Delta^5 y_{-3}$, ..., , используемые в этой формуле, образуют верхнюю ломаную линию в диагональной таблице разностей.

Формулы Гаусса применяются для интерполирования в середине таблицы вблизи x_0. При этом первая формула Гаусса применяется при $x > x_0$, а вторая – при $x < x_0$.

Программа:

```
#include <iostream>
#include <math.h>
#include <vector>

using namespace std;
struct dat {
  double y;
  double x;
};
int fack(int x) {
  if( x == 0 ) return 1;
  return x * fack(x - 1);
  }
double delta(vector<dat> a, int index, int num) {
  if (index == 1) {
    return a[num+1].y - a[num].y;
    }
  else {
    return delta(a, index-1, num+1) - delta(a, index-1, num);
    }
}

int main() {
  int n;
  vector<dat> arr;
```

```
  dat z;
  cout << "Введите число точек: " << endl;
  cin >> n;
  cout << "Введите точки, с разделением координат пробелом: "<< endl;
  for(int i = 0; i < n; i++) {
    cin >> z.y >> z.x;
    arr.push_back(z);
  }
  cout << "Введите искомую точку: " << endl;
  double X;
  cin >> X;
  double step = arr[1].x - arr[0].x;
  double q =  X - arr[0].x / step;
  int n1 = 0;
  if (n % 2 != 1) {
    n1 = n*0.5;
  }
  else {
    n1 = (n+1)*0.5;
  }
  double summ = arr[n1].y + q*(arr[n1+1].y-arr[n1].y);
  int key = 0;
  for(int i = 3; i < n1; i++) {
    if (key % 2 == 0) {
      summ += pow(q,i)*delta(arr,n1-i,key*0.5)/fack(i);
    }
    else {
      q += i*0.5-1;
      summ += pow(q,i)*delta(arr, n1-1,(key-1)*0.5)/fack(i);
    }
    key++;
    }
  cout << "Формула 1: " << summ << endl;
  summ = arr[0].y;
  key = 2;
  for(int i = 1; i < n1; i++) {
    if (key % 2 == 0) {
      summ += pow(q,i)*delta(arr,n1-i,key*0.5)/fack(i);
    }
    else {
      q += i*0.5-1;
      summ += pow(q,i)*delta(arr, n1-1,(key-1)*0.5)/fack(i);
    }
    key++;
    }
  cout << "Формула 2:" << summ << endl;
}
```

Пример вывода:

Введите число точек:
3

Введите точки, с разделением координат пробелом:
4 3
9 1
4 4
Введите искомую точку:
2
Формула 1: -10
Формула 2: -13.5

Вывод: был изучен алгоритм интерполяции Гаусса.

7.3 Сплайн интерполяция

Задание:
Разработать алгоритм и написать программу, реализующую: найти приближение функции, заданной в равноотстоящих точках, т. е. функции, заданной в виде последовательности чисел, с помощью сплайна третьей степени. Произвести анализ результатов. Предусмотреть возможность выбора размера последовательности преобразования без перекомпиляции программы, размер последовательности N, где N=5,6,7….

Теория:
Сплайн – функция, которая вместе с несколькими производными непрерывна на всем заданном отрезке $[a, b]$, а на каждом частичном отрезке $[x_i, x_{i+1}]$в отдельности является некоторым алгебраическим многочленом.
Степенью сплайна называется максимальная по всем частичным отрезкам степень многочленов, а дефектом сплайна - разность между степенью сплайна и порядком наивысшей непрерывной на $[a, b]$ производной. Например, непрерывная ломанная является сплайном степени 1 с дефектом 1 (так как сама функция – непрерывна, а первая производная уже разрывна).
На практике наиболее часто используются кубические сплайны $S_3(x)$ - сплайны третьей степени с непрерывной, по крайней мере, первой производной. При этом величина$m_i = S_3'(x_i)$, называется наклоном сплайна в точке x_i.
Разобьём отрезок [a, b] на N равных отрезков $[x_i, x_{i+1}]$, где $x_i = a + ih$, i=0, 1, …, N-1,$x_N = b, h = (b-a)/N$.
Если в узлах x_i, x_{i+1} заданы значения f_i, f_{i+1}, которые принимает кубический сплайн, то на частичном отрезке $[x_i, x_{i+1}]$он принимает вид:

$$S_3(x) = \frac{(x_{i+1}-x)^2(2(x-x_i)+h)}{h^3} f_i + \frac{(x-x_i)^2(2(x_{i+1}-x)+h)}{h^3} f_{i+1} + \frac{(x_{i+1}-x)^2(x-x_i)}{h^2} m_i + \frac{(x-x_i)^2(-x_{i+1}+x)}{h^2} m_{i+1} \quad (1)$$

В самом деле, это легко проверить, рассчитав $S_3(x)$ и $S_3'(x)$ в точках x_i, x_{i+1}.
Можно доказать, что если многочлен третьей степени принимает в точках x_i, x_{i+1} значения f_i, f_{i+1} и имеет в этих точках производные, соответственно, m_i, m_{i+1}, то он совпадает с многочленом (1).
Таким образом, для того, чтобы задать кубический сплайн на отрезке, необходимо задать значения f_i, m_i i=0,1…, N в N+1 в узлеx_i.
Кубический сплайн, принимающий в узлах те же значения f_i, что и некоторая функция, называется интерполяционным и служит для аппроксимации функции f на отрезке $[a, b]$ вместе с несколькими производными.

Программа:

```
#include <vector>
#include <array>
#include <stdio.h>
#include <cmath>
using namespace std;

typedef array<double,4> polynome;

int main() {
  double a, b;
  int n;
      vector<double> m = {};
      vector<double> x = {};
  vector<double> f = {};
  printf("Введите границы: ");
  scanf("%lf %lf",&a,&b);
  printf("Введите N: ");
  scanf("%d",&n);
  double h = (b-a)/n;
  for (int i = 0; i <= n; i++) {
    x.push_back(a+i*h);
    printf("f(%lf): ",x[i]);
    double tmp;
    scanf("%lf",&tmp);
    f.push_back(tmp);
  }
  m.push_back((4*f[1]-f[2]-3*f[0])/(2*h));
  for (int i = 1; i < n; i++)
    m.push_back((f[i+1]-f[i-1])/(2*h));
  m.push_back((3*f[n]-f[n-2]-3*f[n-1])/(2*h));

  for (int i = 0; i < n; i++) {
    polynome s = {

      (-2*x[i]*pow(x[i+1],2)+h*pow(x[i+1],2))/pow(h,3)*f[i]+
      (2*x[i+1]*pow(x[i],2)+h*pow(x[i],2))/pow(h,3)*f[i+1]+
      (x[i]*pow(x[i+1],2))/pow(h,2)*m[i]+
      (x[i+1]*pow(x[i],2))/pow(h,2)*m[i+1],

      (2*pow(x[i+1],2)+4*x[i]*x[i+1]-2*x[i+1]*h)/pow(h,3)*f[i]+
      (-4*x[i]*x[i+1]-2*x[i]*h-2*pow(x[i],2))/pow(h,3)*f[i+1]+
      (pow(x[i+1],2)+2*x[i]*x[i+1])/pow(h,2)*m[i]+
      (2*x[i]*x[i+1]+pow(x[i],2))/pow(h,2)*m[i+1],

      (-4*x[i+1]-2*x[i]+h)/pow(h,3)*f[i]+
      (4*x[i]+2*x[i+1]+h)/pow(h,3)*f[i+1]+
      (-2*x[i+1]-x[i])/pow(h,2)*m[i]+
      (-2*x[i]-x[i+1])/pow(h,2)*m[i+1],

      2/pow(h,3)*f[i]-
```

```
      2/pow(h,3)*f[i+1]+
      1/pow(h,2)*m[i]+
      1/pow(h,2)*m[i+1],

    };
    double check =  (s[3]*(pow(x[i],3)+pow(x[i+1],3))+
                    s[2]*(pow(x[i],2)+pow(x[i+1],2))+
                    s[1]*(x[i]+x[i+1]))/2+s[0];
    s[0] -= (check-(f[i]+f[i+1])/2);
    printf(
      "x ∈ [%lf, %lf], S3(x) = (%lf)x^3 + (%lf)x^2 + (%lf)x +
(%lf)\n",
      x[i],x[i+1],
      s[3],s[2],s[1],s[0]
    );
  }
}
```

Пример вывода:
Введите границы: 0 10
Введите N: 3
f(0.000000): 1
f(3.333333): 2
f(6.666667): 5
f(10.000000): 6
x ∈ [0.000000, 3.333333], S3(x) = (0.000000)x^3 + (0.090000)x^2 + (0.000000)x + (1.000000)
x ∈ [3.333333, 6.666667], S3(x) = (-0.054000)x^3 + (0.810000)x^2 + (-3.000000)x + (5.000000)
x ∈ [6.666667, 10.000000], S3(x) = (0.013500)x^3 + (-0.405000)x^2 + (4.200000)x + (-9.000000)

Вывод:
Был изучен алгоритм, реализующий приближение функции, заданной в равноотстоящих точках.

7.4 Фурье интерполяция

Задание:
Разработать алгоритм и написать программу, реализующую: найти приближение функции, заданной в равноотстоящих точках, т. е. функции, заданной в виде последовательности чисел, с помощью интерполяции Фурье. Произвести анализ результатов. Предусмотреть возможность выбора размера последовательности преобразования без перекомпиляции программы, размер последовательности N, где N=2n, n=1, 2, 3, ….

Теория:
Допустим, что функция *f(x)* является суммой тригонометрического ряда, и этот ряд равномерно сходится на $-\pi \le x \le \pi$, тогда определить его коэффициенты достаточно легко. Умножим равенство

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} (a_n \cos \mathrm{nx} + b_n \sin \mathrm{nx})$$

на *cos kx* или на *sin kx*, проинтегрируем его в пределах от $-\pi$ до $+\pi$ и в результате получим

$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) cosnxdx\,; b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) sinnxdx$$

Эти формулы называются формулами Фурье, числа a_n и b_n – коэффициентами Фурье, а ряд, коэффициенты которого определяются по формулам Фурье, отправляясь от функции *f(x),* носит название ряда для функции *f(x).*

Программа:
```
#include <cstdlib>
#include <iostream>
#include <cmath>

using namespace std;
#define pi 3.1415926
int n;
double* a,*b;
double Function(double);
double A(int j)
{
    double S=0;
    int ii;
```

```
    for (int i=-n;i<n+1;i++)
    {


S=S+Function(2*pi*double(i)/(2*n+1))*cos(2*pi*double(j)*double(i)/(2*n
+1));
    }
    if (j==0) return 1/double(2*n+1)*S;

    return 2/double(2*n+1)*S;
}
double B(int j)
{
    int ii;
    double S=0;
    for (int i=-n;i<n+1;i++)
    {
        ii=-n+i;

S=S+Function(2*pi*double(i)/double(2*n+1))*sin(2*pi*double(j*i)/double
(2*n+1));
    }
    return 2/double(2*n+1)*S;
}
double Function(double x)
{
    if (x<-1.5) return double(1);
    if (x>=-1.5 && x<1.6) return 0.5;
    return -0.333;
}
double Interpolate(double x)
{
    double S=a[0];
    for (int i=1;i<n;i++)
    {
        S=S+a[i]*cos(double(i)*x)+b[i]*sin(double(i)*x);
    }
    return S;
}
int main()
{
    double size;
    cout <<"Введите число интерполяционных членов: ";
    cin>>n;
    cout<<"Введите размер результирующего вектора: ";
    cin >>size;


    a=new double[n];
    b=new double[n];
```

```
    cout <<endl;
    cout<<"Коэффициент a :"<<endl << endl;
    for (int j=0; j<n;j++)
    {
        a[j]=A(j);
        cout<<a[j]<<endl;
    }
    cout <<endl;
    cout<<"Коэффициент b :"<<endl << endl;
    for (int j=1; j<n;j++)
    {
        b[j]=B(j);
        cout<<b[j]<<endl;
    }
    cout <<endl;
    double x;
    cout<< "Координаты и значения в них:"<<endl;
    for (int j=0;j<size;j++)
    {
        x=-pi+2*pi/size*j;
        cout<<x<<"\t"<<Function(x)<<endl;
    }
    for (int j=0;j<size;j++)
    {
        x=-pi+2*pi/size*j;
        cout<<x<<"\t"<<Interpolate(x)<<endl;
    }
    cout<< endl<<"Ошибка: "<<endl;
    for (int j=0;j<n;j++)
    {
        x=-pi+2*pi/size*j;
        cout<<x<<"\t"<<Interpolate(x)-Function(x)<<endl;
    }

    return 0;
}
```

Пример вывода:
Введите число интерполяционных членов: 4
Введите размер результирующего вектора: 9

Коэффициент a :

0.426
0.106537
-0.0196873
-0.037

Коэффициент b :

-0.35785
0.446944

-0.256536

Координаты и значения в них:
-3.14159 1
-2.44346 1
-1.74533 1
-1.0472 0.5
-0.349066 0.5
0.349066 0.5
1.0472 0.5
1.74533 -0.333
2.44346 -0.333
-3.14159 0.336776
-2.44346 1.21481
-1.74533 0.690611
-1.0472 0.448955
-0.349066 0.549799
0.349066 0.435262
1.0472 0.60327
1.74533 0.124389
2.44346 -0.569873

Ошибка:
-3.14159 -0.663224
-2.44346 0.214811
-1.74533 -0.309389
-1.0472 -0.0510454

Выводы: был изучен алгоритм, реализующий приближение функции, заданной в равноотстоящих точках.

8. Приближённое дифференцирование
8.1 Конечные разности

Задание:
Разработать алгоритм и написать программу, реализующую: дифференцирование функции (найти производную заданного порядка), заданной в равноотстоящих точках, с помощью правых и левых конечных разностей.

Теория:
В ряде случаев возникает необходимость найти производные от функции y=f(x) , заданной таблично. Возможно также, что непосредственное дифференцирование функции оказывается слишком сложным в силу особенностей аналитического задания функции. В этих случаях прибегают к приближенному дифференцированию.

Для вывода формулы приближенного дифференцирования данную функцию $f(x)$ заменяют интерполяционным полиномом $P(x)$, и полагают:

$f'(x) = P'(x)$ на отрезке [a, b].

Погрешность интерполирующей функции P(x) определяют разностью: $R(x) = f(x) - P(x)$ и тогда погрешность производной $P'(x)$ выражается формулой: $r(x) = f'(x) - P'(x) = R'(x)$.

Получим формулы приближенного дифференцирования, основанные на первой интерполяционной формуле Ньютона.

Пусть функция $y = f(x)$ задана в равноотстоящих точках $x_i (i = 0,1,2, \dots, n)$ отрезка [a, b]. Функцию *y* приближенно заменим интерполяционным полиномом Ньютона:

$$y = y_0 + q\Delta y_0 + \frac{q(q-1)}{2!}\Delta^2 y_0 + \frac{q(q-1)(q-2)}{3!}\Delta^3 y_0 + \frac{q(q-1)(q-2)(q-3)}{4!}\Delta^4 y_0 + \cdots$$

Здесь $q = \frac{x-x_0}{h}, h = x_{i+1} - x_i$ - шаг интерполяции.

Δy_0 - первая конечная разность:

$$\Delta y_0 = \Delta f(x_0) = f(x_0 + h) - f(x_0)$$

$\Delta^2 y_0$ - вторая конечная разность:$\Delta^2 y_0 = \Delta(\Delta y_0)$

$\Delta^n y_0$ - конечные разности высших порядков:

$$\Delta^n y_0 = \Delta(\Delta^{n-1} y_0)$$

Производя перемножение в формуле (3.12) и раскрывая факториал, получаем:

$$y = y_0 + q\Delta y_0 + \frac{q^2 - q}{1 * 2}\Delta^2 y_0 + \frac{q^3 - 3q^2 + 2q}{1 * 2 * 3}\Delta^3 y_0 + \frac{q^4 - 6q^3 + 11q^2 - 6q}{1 * 2 * 3 * 4}\Delta^4 y_0 + \cdots$$

Учитывая, что $y' = \frac{dy}{dx} = \frac{dy}{dq} * \frac{dq}{dx} = \frac{1}{h} * \frac{d(y\ \ )}{dq}$, получаем формулу приближенного дифференцирования:

$$y' = \frac{1}{h}\left[\Delta y_0 + \frac{2q - 1}{2}\Delta^2 y_0 + \frac{3q^2 - 6q + 2}{6}\Delta^3 y_0 + \frac{2q^3 - 9q^2 + 11q - 3}{12}\Delta^4 y_0 + \cdots\right]$$

Аналогично для второй производной:

$$y'' = \frac{d(y')}{dx} = \frac{d(x')}{dq} * \frac{dq}{dx} = \frac{1}{h} * \frac{d(y')}{dq}$$

$$y'' = \frac{1}{h^2}\left[\Delta y_0 + \frac{2q - 1}{2}\Delta^2 y_0 + \frac{3q^2 - 6q + 2}{6}\Delta^3 y_0 + \frac{2q^3 - 9q^2 + 11q - 3}{12}\Delta^4 y_0 + \cdots\right]$$

Таким же способом можно вычислить производную любого порядка.

Если функция задана таблично, и значение производной нужно вычислить в узловых точках x_i, то каждое табличное значение принимают за начальное x = x0и тогда $q = 0$. Формулы численного дифференцирования существенно упрощаются. Полагая в формуле (3.14) $q = 0$, получаем:

$$y' = \frac{1}{h}(\Delta y_0 - \frac{\Delta^2 y_0}{2} + \frac{\Delta^3 y_0}{3} - \frac{\Delta^4 y_0}{4} + \cdots)$$

Для второй производной:

$$y'' = \frac{1}{h^2}(\Delta^2 y_0 - \Delta^3 y_0 + \frac{11}{12}\Delta^4 y_0 - \cdots)$$

Опустим теоретический вывод и приведем конечную формулу для вычисления погрешности производной:

$$R'(x_0) \approx \frac{(-1)^k \Delta^{k+1} y_0}{h(k+1)}$$

где k — это максимальный порядок конечной разности, входящей в интерполяционный полином Ньютона P(x).

Программа:

```
#include <iostream>

#include <vector>


std::vector <double> dif_finite_differences(const std::vector<double>
&data, double h)

{

     if (data.size() == 0 || h <= 0)

          throw "Неверные входные данные";
```

```
	std::vector <double> res(data.size() - 1);


	for (int index = 0; index < res.size(); ++index)
		res[index] = (data[index + 1] - data[index]) / h;


	return res;
}


int main()
{
	std::vector<double> func{ 3.912, 3.951, 3.989, 4.025, 4.06, 4.094, 4.127, 4.159, 4.19, 4.22 };
	double h = 2;
	double x0 = 50;
	std::vector<double> first_differences = dif_finite_differences(func, h);
std::vector<double> second_differences = dif_finite_differences(first_differences, h);
	std::vector<double> third_differcences = dif_finite_differences(second_differences, h);


	std::cout << "Функция:\n";
	for (int index = 0; index < func.size(); ++index)
		std::cout << "f(" << x0 + index * h << ") = " << func[index] << std::endl;


	std::cout << "Первая производная через левые конечные разности: \n";
	for (int index = 0; index < first_differences.size(); ++index)
		std::cout << "f'(" << x0 + h + index * h << ") = " << first_differences[index] << std::endl;


	std::cout << "Вторая производная через левые конечные разности: \n";
	for (int index = 0; index < second_differences.size(); ++index)
		std::cout << "f''(" << x0 + 2 * h + index * h << ") = " << second_differences[index] << std::endl;
```

```
	std::cout << "Третья производная через левые конечные разности: \n";
	for (int index = 0; index < third_differcences.size(); ++index)
		std::cout << "f'’’(" << x0 + 3 * h + index * h << ") = " << third_differcences[index] << std::endl;


	std::cout << "Первая производная через правые конечные разности: \n";
	for (int index = 0; index < first_differences.size(); ++index)
		std::cout << "f'(" << x0  + index * h << ") = " << first_differences[index] << std::endl;


	std::cout << "Вторая производная через правые конечные разности: \n";
	for (int index = 0; index < second_differences.size(); ++index)
		std::cout << "f''(" << x0  + index * h << ") = " << second_differences[index] << std::endl;


	std::cout << "Третья производная через правые конечные разности: \n";
	for (int index = 0; index < third_differcences.size(); ++index)
		std::cout << "f'’’(" << x0 + index * h << ") = " << third_differcences[index] << std::endl;


	system("pause");
	return 0;
}
```

Пример вывода:

Функция:

f(50) = 3.912

f(52) = 3.951

f(54) = 3.989

f(56) = 4.025

f(58) = 4.06

f(60) = 4.094

f(62) = 4.127

f(64) = 4.159

f(66) = 4.19

f(68) = 4.22

Первая производная через левые конечные разности:

f'(52) = 0.0195

f'(54) = 0.019

f'(56) = 0.018

f'(58) = 0.0175

f'(60) = 0.017

f'(62) = 0.0165

f'(64) = 0.016

f'(66) = 0.0155

f'(68) = 0.015

Вторая производная через левые конечные разности:

f''(54) = -0.00025

f''(56) = -0.0005

f''(58) = -0.00025

f''(60) = -0.00025

f''(62) = -0.00025

f''(64) = -0.00025

f''(66) = -0.00025

f''(68) = -0.00025

Третья производная через левые конечные разности:

f'''(56) = -0.000125

f'''(58) = 0.000125

f'''(60) = 3.33067e-16

f’’(62) = -3.33067e-16

f’’(64) = 2.22045e-16

f’’(66) = 0

f’’(68) = -2.22045e-16

Первая производная через правые конечные разности:

f'(50) = 0.0195

f'(52) = 0.019

f'(54) = 0.018

f'(56) = 0.0175

f'(58) = 0.017

f'(60) = 0.0165

f'(62) = 0.016

f'(64) = 0.0155

f'(66) = 0.015

Вторая производная через правые конечные разности:

f’(50) = -0.00025

f’(52) = -0.0005

f’(54) = -0.00025

f’(56) = -0.00025

f’(58) = -0.00025

f’(60) = -0.00025

f’(62) = -0.00025

f’(64) = -0.00025

Третья производная через правые конечные разности:

f’’(50) = -0.000125

f’’(52) = 0.000125

f’’(54) = 3.33067e-16

f’’(56) = -3.33067e-16

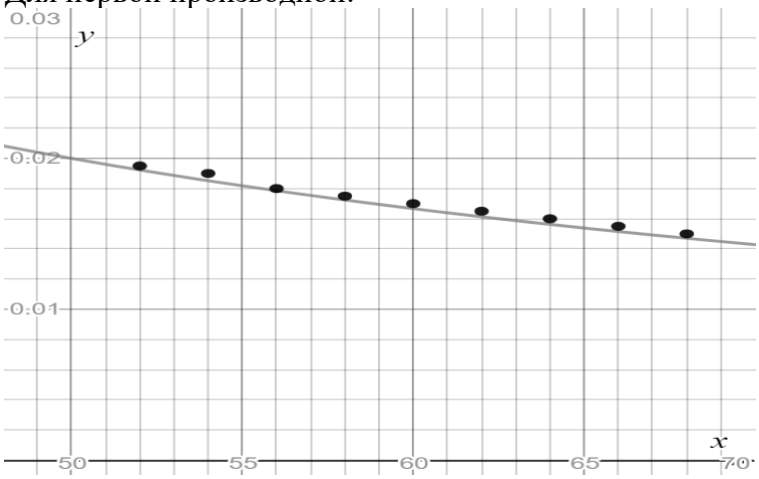
f’’’(58) = 2.22045e-16

f’’’(60) = 0

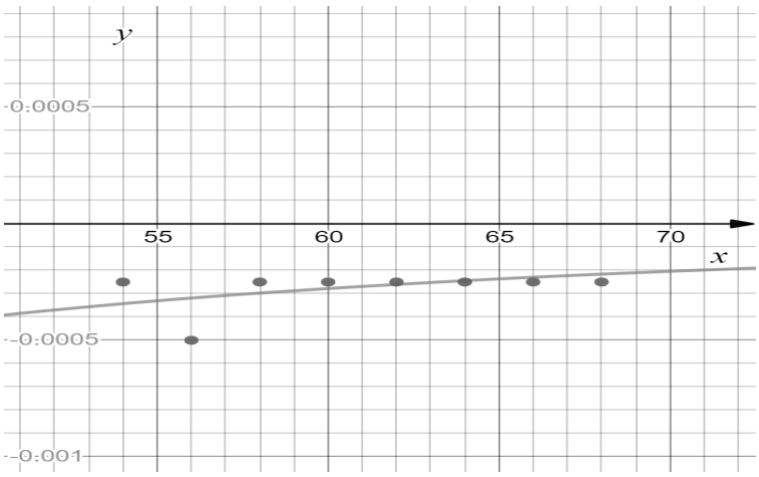
f’’’(62) = -2.22045e-16

Сравнение производной с рассчитанной методом левых конечных разностей:

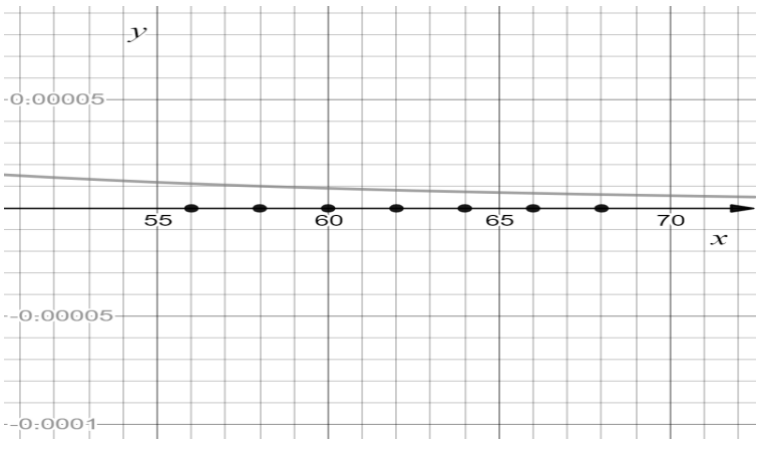
Для первой производной:



Для второй производной:



Для третьей производной:



8.2 Производная с помощью преобразования Фурье

Задание: разработать алгоритм и написать программу, реализующую: вычисление производной с помощью преобразования Фурье.

Теория:
Пусть функция *f* абсолютно интегрируема на $(-\infty, \infty)$ и *f'* непрерывна и абсолютно интегрируема на $(-\infty, \infty)$. Тогда

$$F[f'](y) = (iy)F[f](y), \quad y \in (-\infty, \infty)$$

Представим функцию *f* в виде

$$f(x) = f(0) + \int_0^x f'(t)dt$$

из сходимости интеграла $\int_0^{+\infty} f'(t)dt$ следует существование пределов $\lim_{x\to+\infty} f(x)$, $\lim_{x\to-\infty} f(x)$. Они не могут быть отличными от нуля в силу сходимости интеграла $\int_{-\infty}^{\infty} |f(x)|dx$. С помощью интегрирования по частям получаем

$$F[f'](y) = \frac{1}{2\pi}\int_{-\infty}^{\infty} f'(x)e^{-ixy}dx = \frac{1}{\sqrt{2\pi}}f(x)e^{-ixy}\Big|_{-\infty}^{\infty} + \frac{iy}{\sqrt{2\pi}}\int_{-\infty}^{\infty} f(x)e^{-ixy}dy = iyF[f](y).$$

Пусть функции *f* непрерывна на $(-\infty, \infty)$, а функция $f_1 = x\,f(x)$ абсолютно интегрируема на $(-\infty, \infty)$. Тогда

$$\frac{d}{dy}F[f](y) = F[-if_1](y) = F[-ixf(x)](y).$$

$$F[f](y) := v.p.\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} f(x)e^{-iyx}dx$$

Дифференцируя интеграл по параметру *y*, получаем на основании теоремы

$$\frac{d}{dy}F[f](y) = \frac{1}{\sqrt{2\pi}}\frac{d}{dy}\int_{-\infty}^{\infty} f(x)e^{-iyx}dx = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty}(-ix)f(x)e^{-iyx}dx$$

Заметим, что последний интеграл сходится равномерно на $(-\infty, \infty)$ по признаку Вейерштрасса с мажорантном $\varphi(x) = |\,x\,f(x)\,|$.

Программа:

```
#define _USE_MATH_DEFINES
#include <iostream>
#include <cmath>
#include <complex>
#include <vector>

using namespace std;

int main() {
   setlocale(LC_ALL, "Russian");
   cout << "Введите n" << endl;
```

```
    int l = 2;
    cin >> l;
    int N = pow(2, l);
    cout << "Введите вектор сигнала в пространстве координат-отсчетов по времени,
состоящий из " << N << " элементов" << ":" << endl;
    complex <double> j(0, -1);
    vector <vector<complex<double> > > matrix(N, vector<complex<double> >(N));
    for (int k = 0; k < N; k++) {
        for (int n = 0; n < N; n++) {
            matrix[k][n] = (1 / sqrt(N)) * pow(M_E, (j * (2 * M_PI * k * n / N)));
            double a = matrix[k][n].real();
            double b = matrix[k][n].imag();
            if (abs(a) < 1e-7)
                a = 0;
            if (abs(b) < 1e-7)
                b = 0;
            complex<double> tmp(a, b);
            matrix[k][n] = tmp;
        }
    }
    complex <double> om(2 * M_PI/ N, 0);
    vector <vector<complex<double> > > matrix_s(N, vector<complex<double> >(N));
    for (int i = 0; i < N; i++) {
        for (int k = 0; k < N; k++) {
            matrix_s[i][k] = conj(matrix[i][k]);
            //matrix_s[i][k] *= om * j * real(k);// real(i)* real(k);
            double a = matrix_s[i][k].real();
            double b = matrix_s[i][k].imag();
            if (abs(a) < 1e-7)
                a = 0;
            if (abs(b) < 1e-7)
                b = 0;
            complex<double> tmp(a, b);
            matrix_s[i][k] = tmp;
        }
    }
    vector <complex<double> >  x(N, (0, 0));
    vector <complex<double> > X(N, (0, 0));
    vector <complex<double> > x_t(N, (0, 0));

    double c;
    for (int i = 0; i < N; i++) {
        cin >> c;
        complex <double> tmp(c, 0);
        x[i] = tmp;
    }

    for (int i = 0; i < N; i++)
    {
        for (int k = 0; k < N; k++)
        {
            X[i] += matrix[i][k] * x[k];
```

```
            double a = X[i].real();
            double b = X[i].imag();
            if (abs(a) < 1e-7)
                a = 0;
            if (abs(b) < 1e-7)
                b = 0;
            complex<double> tmp(a, b);
            X[i] = tmp;

        }

    }

    for (int i = 0; i < N; i++)
    {
        for (int k = 0; k < N; k++)
        {
            x_t[i] += matrix_s[i][k] * X[k] * j * om * real(k);
            double a = x_t[i].real();
            double b = x_t[i].imag();
            if (abs(a) < 1e-7)
                a = 0;
            if (abs(b) < 1e-7)
                b = 0;
            complex<double> tmp(a, b);
            x_t[i] = tmp;

        }

    }

    cout << "Производная через преобразование Фурье: " << endl;
    for (int i = 0; i < N; i++) {
        cout << abs(x_t[i]);
        cout << endl;
    }
    return 0;
}
```

Пример вывода:

Введите n
3
Введите вектор сигнала в пространстве координат-отсчетов по времени, состоящий из 8 элементов:
1 9 -4 2 56 4 8 7
Производная через преобразование Фурье:
29.6195
4.45852
47.4232
61.9261
143.473
47.1907

19.1306
17.9087

Вывод: был изучен алгоритм вычисления производной с помощью преобразования Фурье

8.3 Интерполяционные формулы Ньютона

Задание:
Разработать алгоритм и написать программу, реализующую: вычисление производной с помощью интерполяционных формул Ньютона. Результаты, для производной одной и той же последовательности, полученной с помощью интерполяционных формул Ньютона и правых и левых конечных разностей отобразить на графике и сравнить.

Теория:
Для вычисления производной в табличной точке значения функции можно воспользоваться интерполяционными формулами Ньютона.
Значения первых двух производных вычисляются по следующим формулам:

$$y' = \frac{1}{h}(\Delta y_0 - \frac{\Delta^2 y_0}{2} + \frac{\Delta^3 y_0}{3} - \frac{\Delta^4 y_0}{4} + \frac{\Delta^5 y_0}{5} \dots)$$

$$y'' = \frac{1}{h^2}(\Delta^2 y_0 - \Delta^3 y_0 + \frac{11}{12}\Delta^4 y_0 - \frac{5}{6}\Delta^5 y_0 + \cdots)$$

Программа:
```
#include <iostream>
#include <vector>
using namespace std;

double count_finite_differences(const vector<double> &data, int start,
int k)
{
    if (start < 0 || k <= 0)
        throw "Неверные входные данные";
    if (start + k > data.size() - 1)
        return 0;

    vector<double> differences(k);
    for (int index = 0; index < differences.size(); ++index)
        differences[index] = data[start + index + 1] - data[start +
index];

    while (differences.size() > 1)
    {
        for (int index = 0; index < differences.size() - 1; ++index)
            differences[index] = differences[index + 1] -
differences[index];
        differences.pop_back();
    }
```

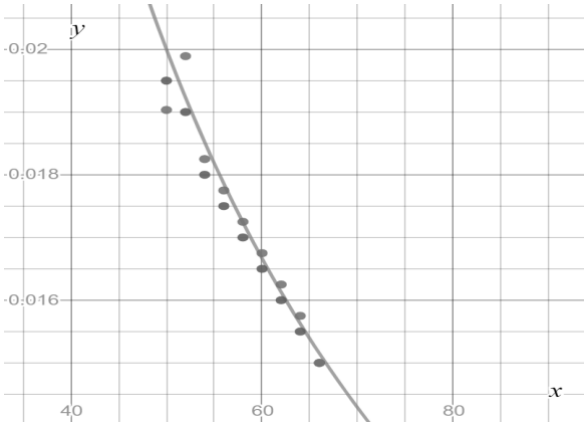
```
    return differences[0];
}
vector<double> count_first_diffirencial(const vector<double> &data,
double h)
{
    vector<double> res(data.size());
    for (int index = 0; index < res.size(); ++index)
    {
        res[index] = (count_finite_differences(data, index, 1) -
count_finite_differences(data, index, 2) / 2 +
count_finite_differences(data, index, 3) / 3 -
count_finite_differences(data, index, 4) / 4 +
count_finite_differences(data, index, 5) / 5 ) / h;
    }
    return res;
}

vector<double> count_second_diffirencial(const std::vector<double>
&data, double h)
{
    vector<double> res(data.size());
    for (int index = 0; index < res.size(); ++index)
    {
        res[index] = (count_finite_differences(data, index, 2) -
count_finite_differences(data, index, 3) +
count_finite_differences(data, index, 4) * 11/ 12 -
count_finite_differences(data, index, 5) * 5 / 6) / h / h;
    }
    return res;
}

int main()
{
    vector<double> func{ 3.912, 3.951, 3.989, 4.025, 4.06, 4.094,
4.127, 4.159, 4.19, 4.22 };
    double h = 2;
    double x0 = 50;

    auto first_differences = count_first_diffirencial(func, h);
    auto second_differences = count_second_diffirencial(func, h);

    cout << "Функция:\n";
    for (int index = 0; index < func.size(); ++index)
        cout << "f(" << x0 + index * h << ") = " << func[index] <<
std::endl;

    cout << "Первая производная через левые конечные разности: \n";
    for (int index = 0; index < first_differences.size(); ++index)
        cout << "f'(" << x0 + h + index * h << ") = " <<
first_differences[index] << std::endl;
```

```
    cout << "Вторая производная через левые конечные разности: \n";
    for (int index = 0; index < second_differences.size(); ++index)
        cout << "f''(" << x0 + 2 * h + index * h << ") = " <<
second_differences[index] << std::endl;
    return 0;
}
```

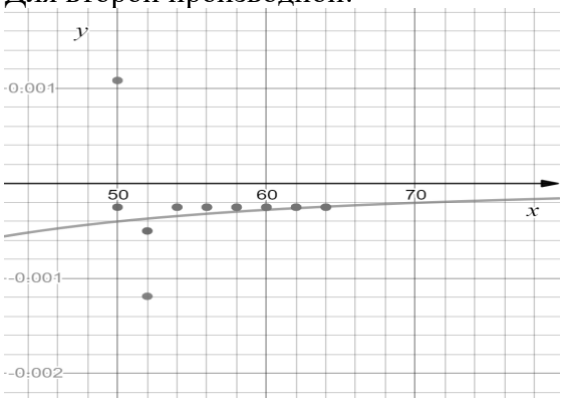
Пример вывода:
Функция:
f(50) = 3.912
f(52) = 3.951
f(54) = 3.989
f(56) = 4.025
f(58) = 4.06
f(60) = 4.094
f(62) = 4.127
f(64) = 4.159
f(66) = 4.19
f(68) = 4.22
Первая производная через левые конечные разности:
f'(52) = 0.0190333
f'(54) = 0.0198917
f'(56) = 0.01825
f'(58) = 0.01775
f'(60) = 0.01725
f'(62) = 0.01675
f'(64) = 0.01625
f'(66) = 0.01575
f'(68) = 0.015
f'(70) = 0
Вторая производная через левые конечные разности:
f''(54) = 0.00108333
f''(56) = -0.0011875
f''(58) = -0.00025
f''(60) = -0.00025
f''(62) = -0.00025
f''(64) = -0.00025
f''(66) = -0.00025
f''(68) = -0.00025
f''(70) = 0
f''(72) = 0

Сравнение производной с рассчитанной методом интерполяционных формул Ньютона.

Для первой производной:



Для второй производной:



Вывод: был изучен алгоритм вычисление производной с помощью интерполяционных формул Ньютона.

9. Приближенное интегрирование

9.1 Функции для интегрирования

1) $f(x) = x^2, x \in [-5; 5]$;
2) $f(x) = sin^2x, x \in [-\pi; \pi]$;
3) $f(x) = sin2x + cos7x + 8, x \in [-\pi; \pi]$;
4) $f(x) = 2x^4 + x^3 + 2x^2 + 3x + 24, x \in [-1; 3]$;
5) $f(x) = ln(x^2 + 1) + sin\frac{x}{3} + 17, x \in [-100; 100]$;
6) $f(x) = 5^x + sinx + x + 11, x \in [-\pi; \pi]$;
7) $f(x) = x^5 + 2x^4 + 3x^3 + 4x^2 + 5x + 6, x \in [-7; 7]$;

9.2 Метод прямоугольников

Задание: разработать алгоритм и написать программу, реализующую: интегрирование функций на заданном интервале методом прямоугольников, расчёт остаточного члена. При равном шаге сетки сравнить полученный результат с результатами полученными иными методами численного интегрирования.

Функция для интегрирования:

$$f(x) = 2x^4 + x^3 + 2x^2 + 3x + 24, x \in [-1; 3];$$

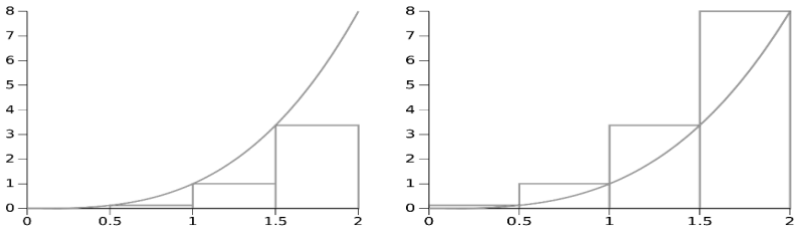
Теория:
Весь участок $[a, b]$делим на n равных частей с шагом

$$h = \frac{(b - a)}{n}$$

Определяем значение y_i подынтегральной функции $f(x)$ в каждой части деления,

$$y_i = f(x_i)$$

Для каждой части деления определяем площадь S_i частичного прямоугольника. Суммируем эти площади. Приближенное значение интеграла равно сумме площадей частичных прямоугольников.



Методы левых прямоугольников (слева) и правых прямоугольников (справа).

Программа:

```
#include <iostream>
#include <cmath>
using namespace std;

double f(double x) {
    return 2 * pow(x, 4) + pow(x, 3) + 2 * pow(x, 2) + 3 * x +
24;
}
int main() {
    double a, b;
    int n;
    cout << "Введите границы интегрирования: " << endl;
    cin >> a >> b;
    cout << "Введите N: " << endl;
    cin >> n;

    double x = a;
    double h = (b - a) / n;
    double left_sum = 0;
    double right_sum = 0;

    for (int i = 0; i < n; ++i) {
        left_sum += f(x) * h;
        x += h;
        right_sum += f(x) * h;
    }
    cout << "Результат метода левых прямоугольников: " << left_sum
<< endl;
    cout << "Результат метода правых прямоугольников: " <<
right_sum << endl;
}
```

Пример вывода:

Введите границы интегрирования:
-2 1
Введите N:
6000
Результат метода левых прямоугольников: 82.9545
Результат метода правых прямоугольников: 82.9455

Выводы: был изучен алгоритм интегрирования функций на заданном интервале методом прямоугольников для данной функции.

9.3 Метод трапеций

Задание:
Разработать алгоритм и написать программу, реализующую: интегрирование функций на заданном интервале методом трапеций, расчёт остаточного члена. При равном шаге сетки сравнить полученный результат с результатами полученными иными методами численного интегрирования.

Функция для интегрирования:

$$f(x) = 2x^4 + x^3 + 2x^2 + 3x + 24, x \in [-1; 3];$$

Теория:
Метод трапеций аналогичен методу прямоугольников, но функция на каждом из частичных отрезков аппроксимируется прямой, проходящей через конечные значения.

Площадь трапеции на каждом отрезке:

$$I_i \approx \frac{f(x_{i-1}) + f(x_i)}{2}(x_i - x_{i-1}).$$

Полная формула трапеций в случае деления всего промежутка интегрирования на отрезки одинаковой длины h:

$$I \approx h\left(\frac{f(x_0) + f(x_n)}{2}\sum_{i=1}^{n-1} f(x_i)\right) \text{где } h = \frac{b-a}{n}$$

Программа:

```
#include <iostream>
#include <cmath>
using namespace std;

double f(double x) {
```

```
return 2 * pow(x, 4) + pow(x, 3) + 2 * pow(x, 2) + 3 * x + 24;
}

int main() {
    double a, b;
    int n;
    cout << "Введите границы интегрирования: " << endl;
    cin >> a >> b;
    cout << "Введите N: " << endl;
    cin >> n;
    double h = (b - a) / n;
    double sum = 0;
    for (int i = 0; i < n; i++) {
        const double x1 = a + i * h;
        const double x2 = a + (i + 1) * h;
        sum += 0.5 * (x2 - x1) * (f(x1) + f(x2));
    }
    cout << sum << endl;
}
```

Пример вывода:
Введите границы интегрирования:
0 5
Введите N:
3000
1647.08

Вывод: был изучен алгоритм интегрирования функций на заданном интервале методом трапеций для данной функции.

9.4. Метод Симпсона

Задание: разработать алгоритм и написать программу, реализующую: интегрирование функций на заданном интервале методом Симпсона, расчет остаточного члена.

Функция для интегрирования:

$$f(x) = 2x^4 + x^3 + 2x^2 + 3x + 24, x \in [-1; 3];$$

Теория:

Формула носит название формулы Симпсона. Геометрически эта формула получается в результате замены кривой $y = f(x)$ параболой $y = L_2(x)$, проходящей через 3 точки $M_0\,(x_0, y_0)$, $M_1\,(x_1, y_1)$, $M_2\,(x_2, y_2)$

$$\int_{x_0}^{x_2} y dx = \frac{h}{3}(y_0 + 4y_1 + y_2)$$

Остаточный член формулы Симпсона равен:

$$R = -\frac{h^5}{90} y^{IV}(\xi), \text{ где } \xi \in (x_0, x_2).$$

Программа:

```
#include <iostream>
#include <cmath>

using namespace std;
double f(double x) {
          return 2 * pow(x, 4) + pow(x, 3) + 2 * pow(x, 2) + 3 * x +
     24;
     }

int main() {
```

```
    double a, b;
    int n;
     cout << "Введите границы интегрирования: " << endl;
     cin >> a >> b;
     cout << "Введите N: " << endl;
     cin >> n;
     double h, sum2 = 0, sum4 = 0, sum = 0;
     h = (b - a) / (2 * n);
     for (int i = 1; i <= 2 * n - 1; i += 2) {
         sum4 += f(a + h * i);
         sum2 += f(a + h * (i + 1));
     }
     sum = f(a) + 4 * sum4 + 2 * sum2 - f(b);
     cout << (h / 3) * sum << endl;
}
```

Пример вывода:

Введите границы интегрирования:
3 8
Введите N:
6000
14539.6

Вывод: был изучен алгоритм интегрирования функций на заданном интервале методом Симпсона.

9.5 Метод Ньютона-Кортеса

Задание: разработать алгоритм и написать программу, реализующую: интегрирование функций на заданном интервале методом Ньютона-Кортеса 3- го и 4-го порядков, расчет остаточного члена.

Теория:
Основной идеей метода является замена подынтегральной функции каким-либо интерполяционным многочленом. После взятия интеграла можно написать

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n} H_i f(x_i)$$

где числа H_i называются коэффициентами Котеса и вычисляются как интегралы от соответствующих многочленов, стоящих в исходном интерполяционном многочлене для подынтегральной функции при значении функции в узле $x_i = a + ih$ ($h = \frac{(b-a)}{n}$ — шаг сетки; n — число узлов сетки, а индекс узлов $\overrightarrow{i = 0 \ldots n}$).

Программа:

```
#include <iostream>
#include <cmath>

using namespace std;

double f(double x) {
    return 2 * pow(x, 4) + pow(x, 3) + 2 * pow(x, 2) + 3 * x + 24;
}
double NewtonCotes(double a, double b, int degree, int n) {
    int H[10][10] = {
            1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            1, 1, 0, 0, 0, 0, 0, 0, 0, 0,
            1, 4, 1, 0, 0, 0, 0, 0, 0, 0,
```

```
            1, 3, 3, 1, 0, 0, 0, 0, 0, 0,
            7, 32, 12, 32, 7, 0, 0, 0, 0, 0,
            19, 75, 50, 50, 75, 19, 0, 0, 0, 0,
            41, 216, 27, 272, 27, 216, 41, 0, 0, 0,
            751, 3577, 1323, 2989, 2989, 1323, 3577, 751, 0, 0,
            989, 5888, -928, 10496, -4540, 10496, -928, 5888, 989, 0,
            2857, 15741, 1080, 19344, 5778, 5778, 19344, 1080, 15741,
2857
    };
    double k[10] = {1, 1.0 / 2, 1.0 / 3, 3.0 / 8, 2.0 / 45, 5.0 / 288,
1.0 / 140, 7.0 / 17280, 4.0 / 14175, 9.0 / 89600};
    double result, sum;
    double h = (b - a) / (degree * n);
    result = 0;
    for (int j = 0; j < n; j++) {
        sum = 0;
        for (int i = 0; i <= degree; i++)
            sum += H[degree][i] * f(a + (i + j * degree) * h);
        result += k[degree] * sum * h;
    }
    return result;
}
int main() {
    double a, b;
    int n;
    cout << "Введите границы интегрирования: " << endl;
    cin >> a >> b;
    cout << "Введите N: " << endl;
    cin >> n;
    cout << NewtonCotes(a, b, 3, n) << endl;
}
```

Пример вывода:

Введите границы интегрирования:
2 5
Введите N:
4000
1570.95

Вывод: был изучен алгоритм интегрирования функций на заданном интервале методом Ньютона-Кортеса 3- го и 4-го порядков.