Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное   
учреждение высшего образования

Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

**Отчет по лабораторной работе**

**«Вычисление математических функций с использованием рядов»**

**Выполнил**:

студент группы 3824Б1-ПМ1

Блеклов С.С.

**Проверила**:

Бусько П.В.

Нижний Новгород

2025

**Содержание**

[Введение 3](#_Toc182419046)

[Постановка задачи 4](#_Toc182419047)

[Руководство пользователя 5](#_Toc182419048)

[Описание программной реализации 6](#_Toc182419049)

[Результаты экспериментов 7](#_Toc182419050)

[Заключение 9](#_Toc182419051)

[Литература 10](#_Toc182419052)

[Приложение 11](#_Toc182419053)

# Введение

Вычисление математических функций с заданной точностью является важной задачей в вычислительной математике. Одним из распространенных методов аппроксимации функций является использование рядов Тейлора и Маклорена. Эти ряды позволяют представить сложные функции в виде суммы бесконечного количества простых слагаемых, что упрощает их вычисление.

Цель данной работы — исследовать зависимость погрешности вычисления функций (синус, косинус, экспонента, натуральный логарифм) от количества членов ряда. Практическая часть включает реализацию алгоритмов на языке C и сравнение результатов с библиотечными функциями.

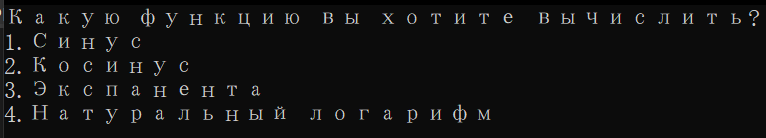
# Постановка задачи

Основная задача — экспериментальное исследование точности вычисления функций с использованием рядов. Этапы работы:

1. **Реализация**:
   * Написание функций для вычисления синуса, косинуса, экспоненты и натурального логарифма через ряды.
   * Реализация двух методов суммирования: прямого (от первого члена к последнему) и обратного (от последнего к первому).
2. **Тестирование**:
   * Сравнение результатов с функциями из библиотеки math.h.
   * Измерение погрешности для разного количества членов ряда.
3. **Анализ**:
   * Определение скорости сходимости рядов.
   * Сравнение эффективности методов суммирования.
4. **Выводы**:
   * Рекомендации по выбору метода и количества членов ряда для минимизации погрешности.

# Руководство пользователя

1. Программа запускается в консоли (Приложение №1). Пользователю предлагается выбрать функцию для вычисления:



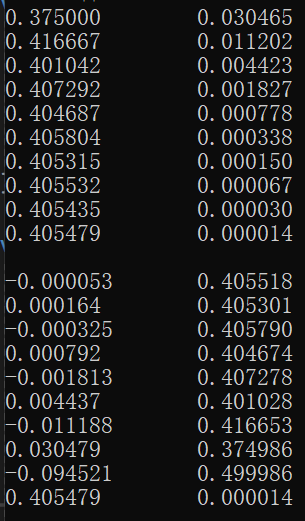
(Рис.1 – изображение меню в консоли)

1. После выбора функции пользователь вводит значение аргумента x и количество итераций (членов ряда) (Приложение №2).



(Рис.2 – ввод значений х и количества итераций)

1. Программа выводит таблицу (Приложение №3) с промежуточными значениями суммы ряда и погрешности на каждом шаге.



(Рис.3 – таблица значений для натурального логарифма)

1. Результаты отображаются для обоих методов суммирования ( прямого и обратного).

# Описание программной реализации

Проект состоит из трех файлов:

1. **lab2.h** — заголовочный файл с объявлениями функций.
2. **lab2.cpp** — реализации функций вычисления рядов.
3. **Лабораторная работа №2.cpp** — основной файл, где прописан интерфейс, структура, а также вспомогательные функции для создания и удаления динамического массива в структуре.

**Подключаемые библиотеки**:

1. **Stdio.h** – стандартная библиотека С для ввода и вывода информации.
2. **Stdlib.h** – библиотека для работы с памятью (динамическими массивами).
3. **Math.h** – библиотека со стандартными функциями, которые используются в работе.

**Функции (Приложение №4)**:

* factorial — вычисление факториала рекурсией для натуральных чисел.
* sinXn, cosXn, expXn, lnXn — реализация прямого суммирования ряда для стандартных функций по Маклорену. Принимают в себя количество итераций, значение х, пустой массив, который будет заполняться погрешностью и пустой массив, который будет заполняться суммой при каждой итерации. Заполнениями в массиве sum реализовано рекурсией.
* sinXnU, cosXnU, expXnU, lnXnU — альтернатива функциям sinXn, cosXn, expXn, lnXn, где реализована обратная сумма.
* test — функция, которая создает и заполняет структуру данными, которые вводятся пользователем, после чего при помощи указателя на функцию использует нужную и выводит прямое и обратное суммирование.
* create\_array, free\_array — функции для создания и удаления массива из структуры. Принимают в себя ссылку на массив, который они заполняют/очищают.

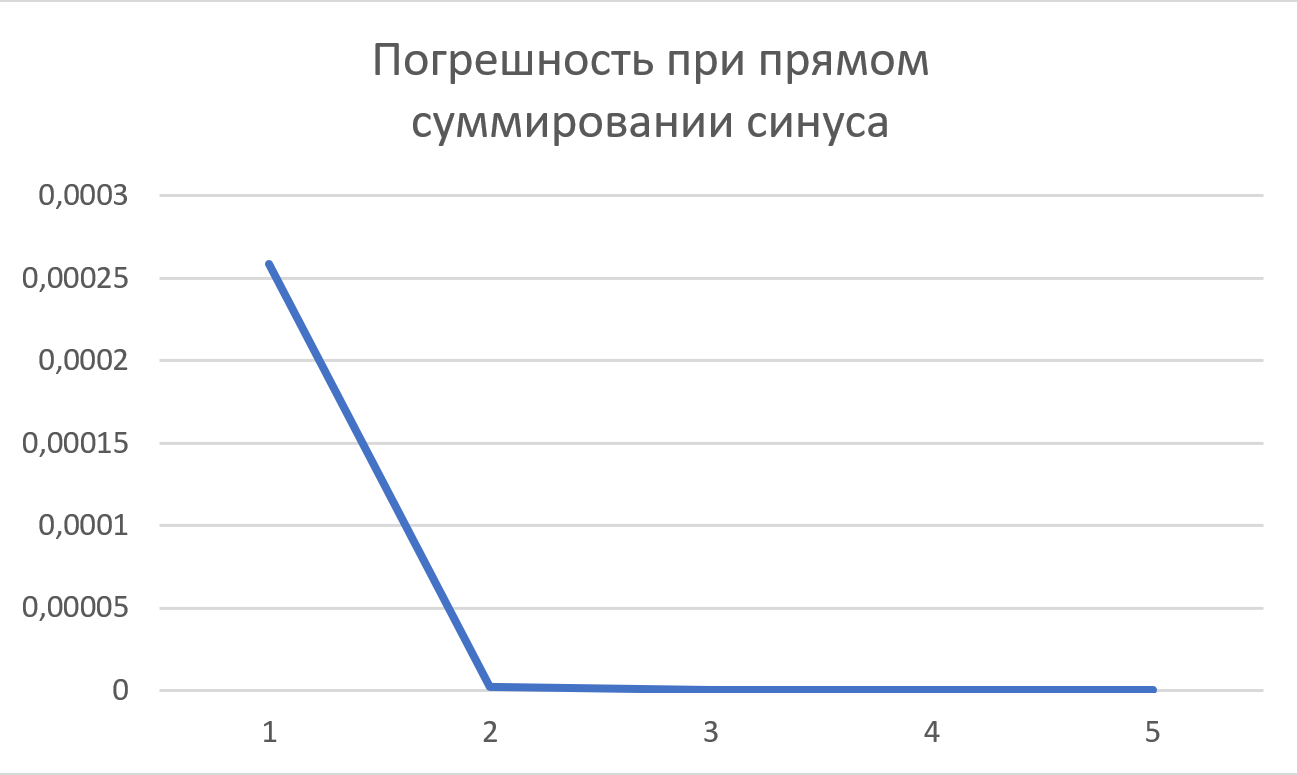
**Особенности реализации**:

* Погрешность вычисляется как абсолютная разность между значением ряда и результатом функции из math.h.

# Результаты экспериментов

Для проверки я решил взять значение х = 0,5 , и увидел, какие результаты разложение по Маклорену показывает при разном количестве итераций:

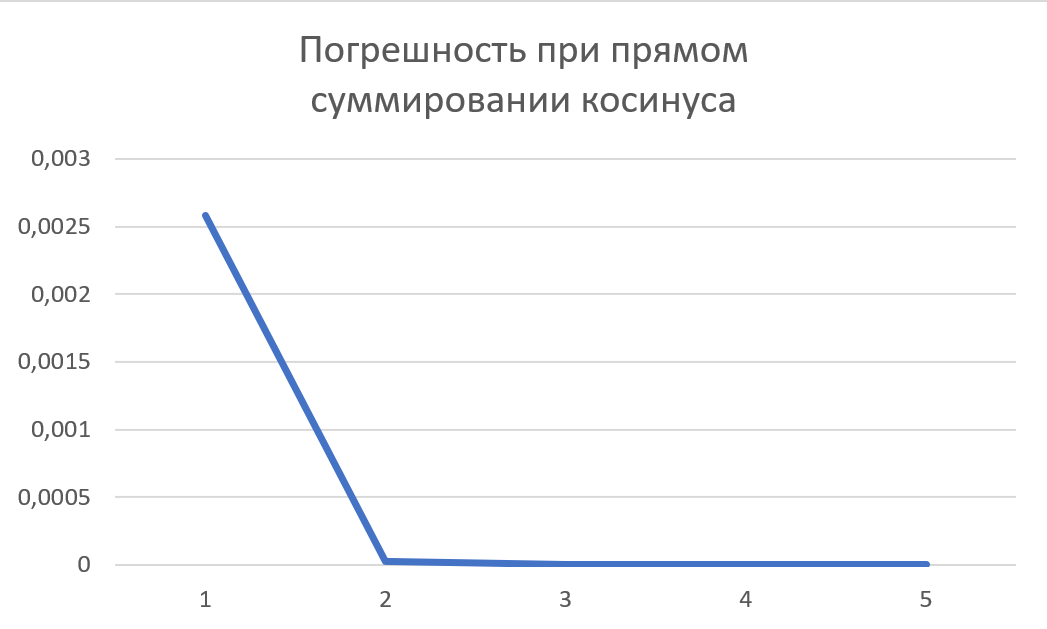
1. Sin – при суммировании достаточно 2 итерации, после чего погрешность при значениях типа double равна 0. (Приложение №5)

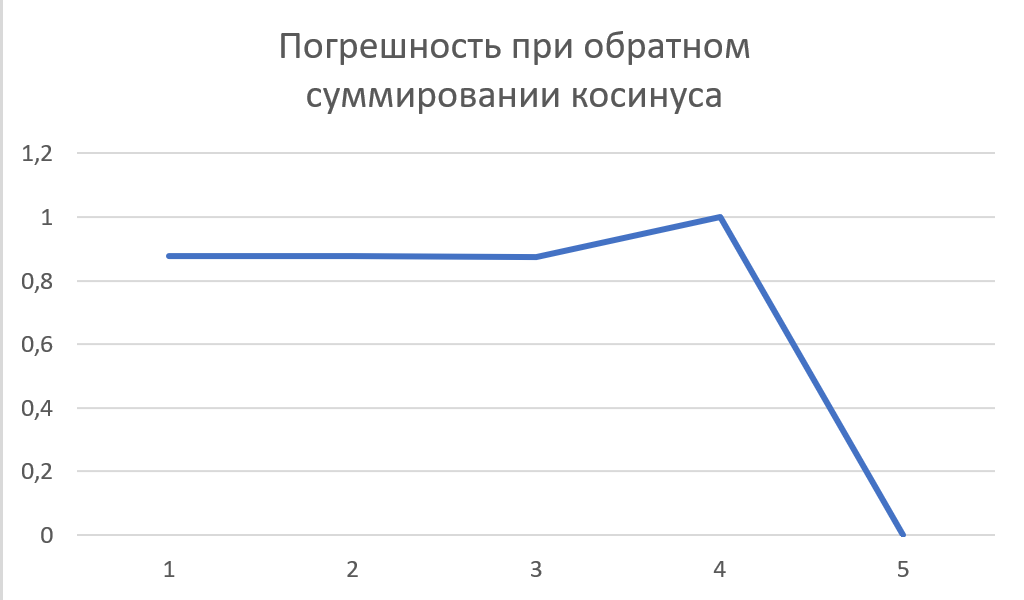


(Рис.4 – график погрешности при прямом суммировании синуса) 

(Рис.5 – график погрешности при обратном суммировании синуса)

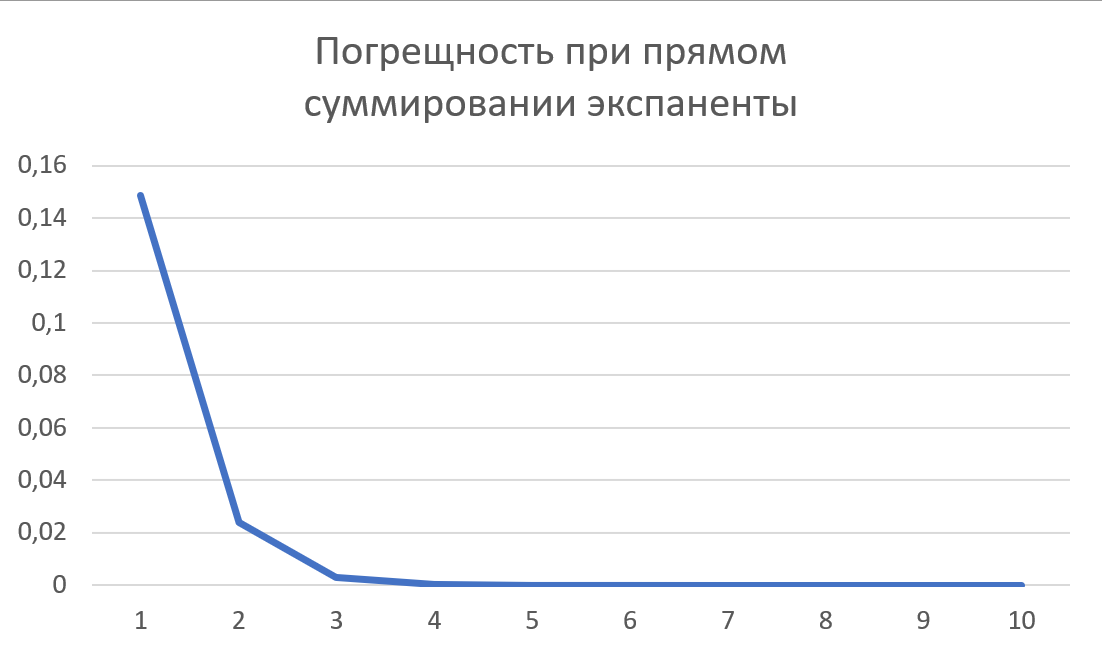
1. Cos – при суммировании достаточно 2 итерации, после чего погрешность при значениях типа double равна 0. (Приложение №6)



(Рис.6 – график погрешности при прямом суммировании косинуса) 

(Рис.7 – график погрешности при обратном суммировании косинуса)

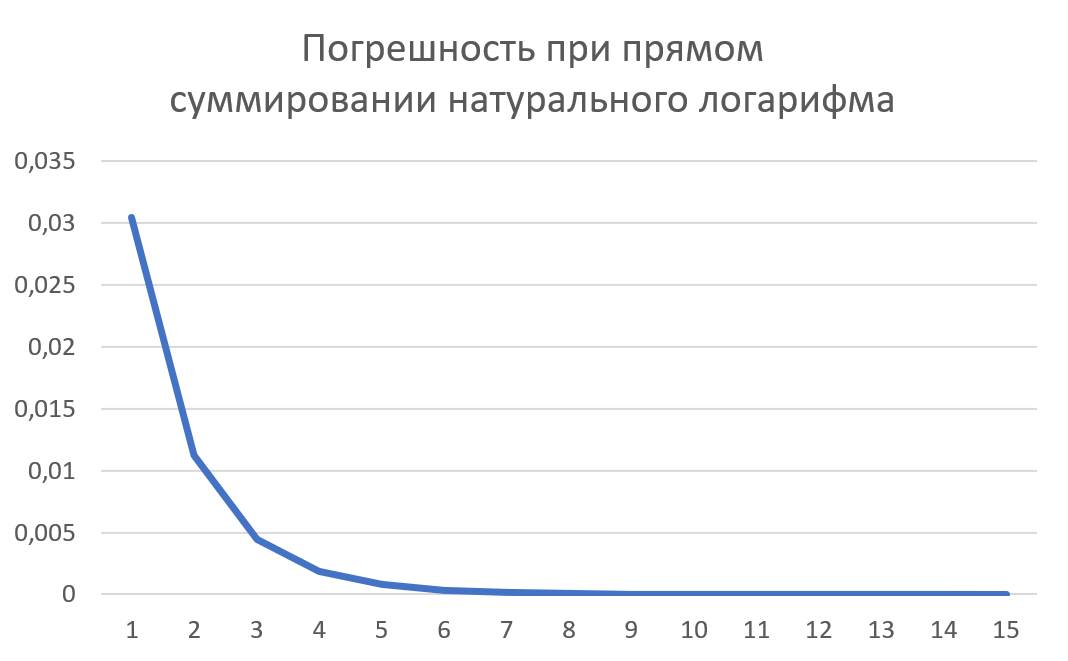
1. Exp – при суммировании достаточно 6 итераций, после чего погрешность при значениях типа double равна 0. (Приложение №7)



(Рис.8 – график погрешности при прямом суммировании экспоненты) 

(Рис.9 – график погрешности при обратном суммировании экспоненты)

1. Ln – при суммировании достаточно 14 итераций, после чего погрешность при значениях типа double равна 0. (Приложение №8)



(Рис.10 – график погрешности при прямом суммировании натурального логарифма) 

(Рис.11 – график погрешности при обратном суммировании натурального логарифма)

# Заключение

В ходе работы были реализованы алгоритмы вычисления математических функций с использованием рядов. Эксперименты подтвердили, что с увеличением количества членов ряда погрешность уменьшается. Прямое суммирование показало более высокую точность на ранних итерациях. При обратном суммировании возникла некоторая особенность – при большом количестве итераций (n>5) первые члены суммы никак не влияют на общую сумму и значения начинают приближаться к нужным только на последних итерациях.

**Рекомендации**:

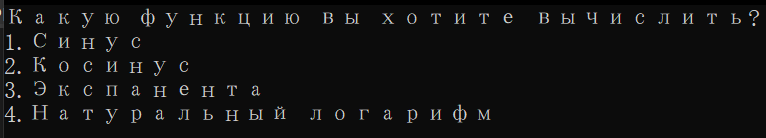
* Для функций с быстрой сходимостью (синус, косинус) достаточно 2-4 членов ряда.
* Для экспоненты и логарифма рекомендуется увеличивать количество итераций (6-20).

# Литература

1. Кнут Д. Э. Искусство программирования. Том 2.
2. Бахвалов Н.С. Численные методы.
3. Документация стандартной библиотеки C (math.h).

# Приложение

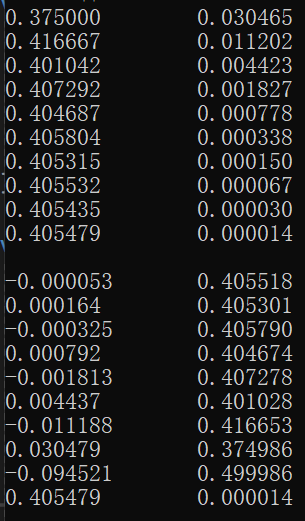
**Приложение №1**



**Приложение №2**



**Приложение №3**



**Приложение №4**

Алгоритм 1.

int factorial(int n) {

if (n < 0) return 0;

if (n <= 1) return 1;

return n \* factorial(n - 1);

}

Алгоритм 2.

void sinXn(int count, double x, double\* err, double\* sum) {

sum[0] = x;

err[0] = fabs(x - sin(x));

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

sum[i] = sum[i - 1] + pow(-1, i) \* (pow(x, 2 \* i + 1) / factorial(2 \* i + 1));

err[i] = fabs(sum[i] - sin(x));

printf("%lf\t%lf\n",sum[i], err[i]);

}

printf("\n");

}

Алгоритм 3.

void cosXn(int count, double x, double\* err, double\* sum) {

sum[0] = 1;

err[0] = fabs(x - cos(x));

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

sum[i] = sum[i - 1] + pow(-1, i) \* (pow(x, 2 \* i) / factorial(2 \* i));

err[i] = fabs(sum[i] - cos(x));

printf("%lf\t%lf\n",sum[i], err[i]);

}

printf("\n");

};

Алгоритм 4.

void expXn(int count, double x, double\* err, double\* sum) {

sum[0] = 1;

err[0] = fabs(x - exp(x));

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

sum[i] = sum[i - 1] + (pow(x, i) / factorial(i));

err[i] = fabs(sum[i] - exp(x));

printf("%lf\t%lf\n",sum[i], err[i]);

}

printf("\n");

}

Алгоритм 5.

void lnXn(int count, double x, double\* err, double\* sum) {

sum[0] = x;

err[0] = fabs(x - log(x+1));

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

sum[i] = sum[i - 1] + pow(-1, i) \* (pow(x, i+1) / (i+1));

err[i] = fabs(sum[i] - log(x+1));

printf("%lf\t%lf\n",sum[i], err[i]);

}

printf("\n");

}

Алгоритм 6.

void sinXnU(int count, double x, double\* err, double\* sum) {

int last = count - 1;

sum[last] = pow(-1, last) \* (pow(x, 2 \* last + 1) / factorial(2 \* last + 1));

err[last] = fabs(sum[last] - sin(x));

for (int i = last - 1; i >= 0; i--) {

sum[i] = sum[i + 1] + pow(-1, i) \* (pow(x, 2 \* i + 1) / factorial(2 \* i + 1));

err[i] = fabs(sum[i] - sin(x));

printf("%lf\t%lf\n",sum[i], err[i]);

}

printf("\n");

}

Алгоритм 7.

void cosXnU(int count, double x, double\* err, double\* sum) {

int last = count - 1;

sum[last] = pow(-1, last) \* (pow(x, 2 \* last) / factorial(2 \* last));

err[last] = fabs(sum[last] - cos(x));

for (int i = last - 1; i >= 0; i--) {

sum[i] = sum[i + 1] + pow(-1, i) \* (pow(x, 2 \* i) / factorial(2 \* i));

err[i] = fabs(sum[i] - cos(x));

printf("%lf\t%lf\n",sum[i], err[i]);

}

printf("\n");

}

Алгоритм 8.

void expXnU(int count, double x, double\* err, double\* sum) {

int last = count - 1;

sum[last] = pow(x, last) / factorial(last);

err[last] = fabs(sum[last] - exp(x));

for (int i = last - 1; i >= 0; i--) {

sum[i] = sum[i + 1] + pow(x, i) / factorial(i);

err[i] = fabs(sum[i] - exp(x));

printf("%lf\t%lf\n",sum[i], err[i]);

}

printf("\n");

}

Алгоритм 9.

void lnXnU(int count, double x, double\* err, double\* sum) {

int last = count - 1;

sum[last] = pow(-1, last) \* pow(x, last + 1) / (last + 1);

err[last] = fabs(sum[last] - log(x + 1));

for (int i = last - 1; i >= 0; i--) {

sum[i] = sum[i + 1] + pow(-1, i) \* pow(x, i + 1) / (i + 1);

err[i] = fabs(sum[i] - log(x + 1));

printf("%lf\t%lf\n",sum[i], err[i]);

}

printf("\n");

}

Алгоритм 10.

void test(void (\*f)(int count, double x, double\* err, double\* sum), void (\*f1)(int count, double x, double\* err, double\* sum)) {

struct xn array;

double v;

int n;

printf("Введите x:");

scanf\_s("%lf", &v);

printf("Введите количество итераций:");

scanf\_s("%i", &n);

array.x = v;

create\_array(&array, n+1);

f(array.count, array.x, array.err, array.sum);

f1(array.count, array.x, array.err, array.sum);

free\_array(&array);

}

Алгоритм 11.

void create\_array(struct xn\* array, int count) {

array->count = count;

array->sum = (double\*)malloc(sizeof(double) \* array->count);

array->err = (double\*)malloc(sizeof(double) \* array->count);

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

array->err[i] = 0;

array->sum[i] = 0;

}

}

Алгоритм 12.

void free\_array(struct xn\* array) {

free(array->sum);

free(array->err);

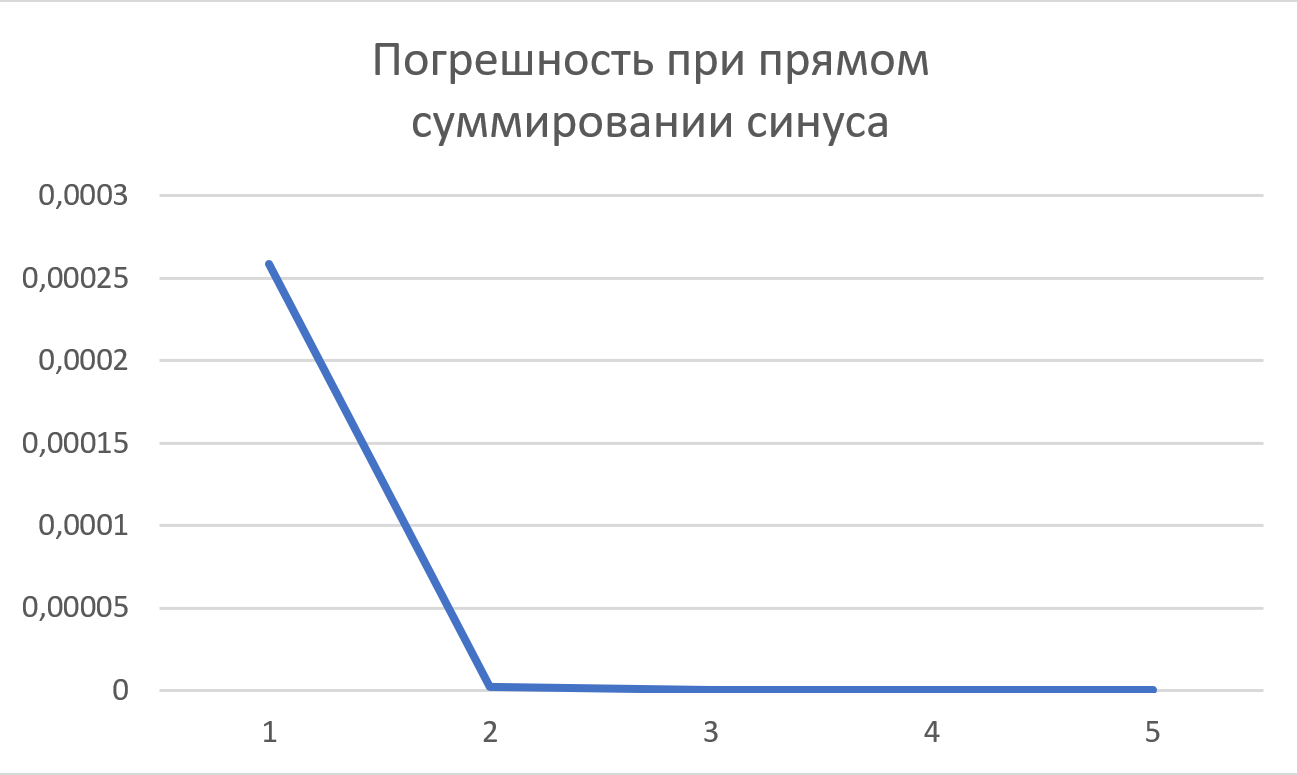
array->sum = NULL;

array->err = NULL;

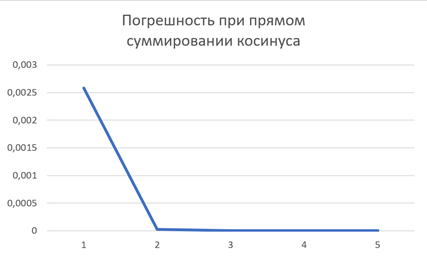
array->count = 0;

}

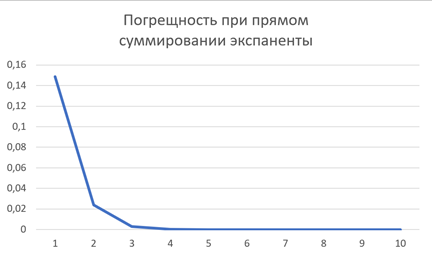
**Приложение №5**



**Приложение №6**



**Приложение №7**



**Приложение №8**

