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»**

**Выполнила**:

студентка группы 3824Б1ПМ1 Сайгушева В.И.

**Проверил**:

преподаватель каф. ВВСП,

Волокитин В.Д.

Нижний Новгород

2025

**Содержание**

[Постановка задачи 3](#_Toc26962562)

[Метод решения 4](#_Toc26962563)

[Руководство пользователя 5](#_Toc26962564)

[Описание программной реализации 6](#_Toc26962565)

[Подтверждение корректности 7](#_Toc26962566)

[Результаты экспериментов 8](#_Toc26962567)

[Заключение 9](#_Toc26962568)

[Приложение 10](#_Toc26962569)

# Постановка задачи

Требуется реализовать вычисление математических функций , , , в точке x при помощи рядов Тейлора. Также необходимо использовать разные виды суммирования (прямое, обратное и попарное суммирование) для сложения членов рядов Тейлора, а затем сравнить результаты вычислений с математическими функциями из библиотеки math.h, чтобы выяснить, какой способ суммирования позволяет получить более точный результат.

# Метод решения

Для вычисления *i-ого* члена ряда Тейлора я использовала его выражение через *(i+1)-ый*, что позволяет упростить вычисления и ускорить программу. Формулы для *i-ого* члена каждой функции выглядят следующим образом:

: , где ;

: , где ;

: , где ;

: , где .

Все элементы ряда записываются в массиве, а затем складываются.

Каждый из видов суммирования выгодно использовать в разных обстоятельствах в зависимости от того, какой ряд мы суммируем. Например, для суммирования ряда Тейлора для синуса или косинуса более эффективно использовать попарное суммирование, поскольку элементы ряда этих тригонометрических функций чередуют знак, а значит, если мы сначала сложим отрицательное и положительное число, то получим маленькое число относительно модуля изначальных слагаемых. Попарное суммирование в этом случае даст высокую точность, поскольку числа с близким порядком величины складываются сначала между собой. Попарное суммирование реализовано в функции *pairwise\_summation*.

Для рядов, которые быстро убывают предпочтительнее будет использовать обратное суммирование (функция *reverse\_summation*). В этом случае элементы складываются начиная с конца, что позволяет учесть дробные части старших членов и достичь большей точности.

Прямое суммирование, реализованное в функции *direct\_summation*, является самым простым в реализации и подходит для рядов, которые быстро сходятся, иначе оно даёт большую погрешность из-за округления дробной части, когда младшие члены могут «потеряться».

# Руководство пользователя

Пользователь при запуске программы должен выбрать математическую функцию, для которой программа будет вычислять значение. Для этого ему нужно ввести одну из цифр от 1 до 4. (рис. 1)



Рисунок 1. Выбор математической функции

Затем пользователю предлагается выбрать суммирование, с помощью которого будут складываться члены ряда Тейлора. (рис. 2)

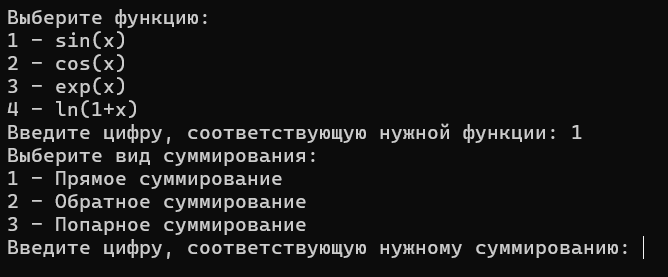


Рисунок 2. Выбор вида суммирования

После чего необходимо ввести значение x, то есть, в какой точке будет вычислено разложение, и количество элементов в ряде. (рис. 3)

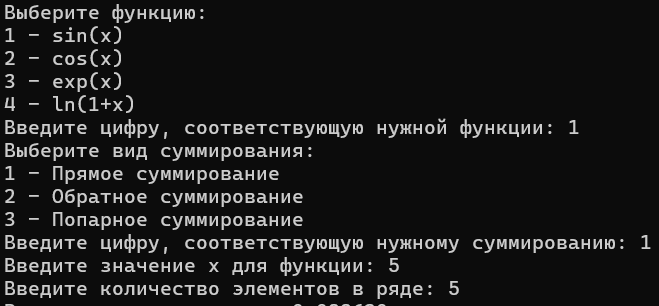


Рисунок 3. Ввод значения х и количества элементов в ряде

На экране появится вычисленное значение и погрешность, полученная в результате вычислений. (рис. 4)



Рисунок 4. Результат работы программы

Программа не позволит пользователю ввести некорректное значение и прекратит свою работу. (рис. 5, 6, 7)

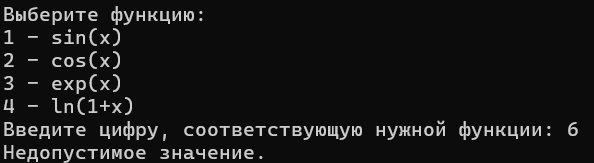


Рисунок 5. Проверка введённых значений

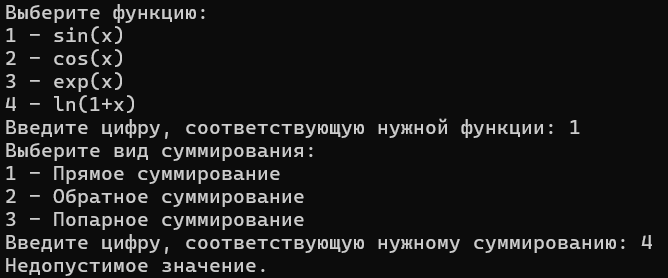


Рисунок 6. Проверка введённых значений

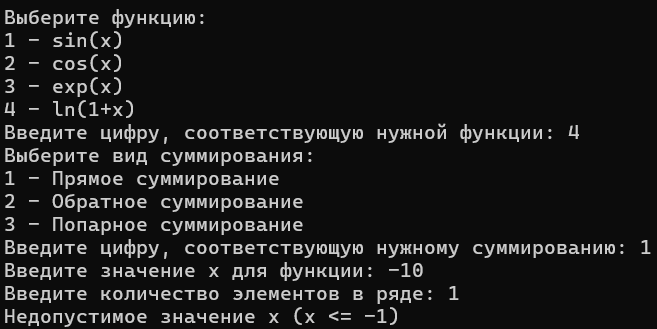


Рисунок 7. Проверка введённых значений

# Описание программной реализации

Для того, чтобы писать на русском языке я использовала функцию *setlocale()* из библиотеки “*locale.h*”. Из библиотеки “*math.h*” мне понадобились математические функции синуса, косинуса, логарифма и экспоненты, так как нужно было вычислить погрешность в расчётах. Библиотека “*stdlib.h*” нужна для использования функции *malloc*. Для удобства тип переменной “*unsigned int*” был назван “*uint*”.

Для каждой математической функции в программе есть аналог: – *sin\_*, – *cos\_*, – *exp\_*, – *ln\_*.

void sin\_(float\* data, uint n, float x)

void cos\_(float\* data, uint n, float x)

void exp\_(float\* data, uint n, float x)

void ln\_(float\* data, uint n, float x)

В каждой из них используется цикл *for*, в котором вычисляется *i-ый* элемент ряда и записывается в массив *data*. Для аргумента тригонометрических функций и я использую функцию “*fmodf*” из библиотеки “*math.h*”. Она позволяет перевести введённый пользователем угол *x* в первую или вторую четверть с помощью вычисления остатка от деления *x* на 2π.

Для суммирования полученных элементов массива реализованы три функции: *direct\_summation* – прямое суммирование, *reverse\_summation* – обратное суммирование и *pairwise\_summation* – попарное суммирование.

float direct\_summation(float\* data, uint n)

float reverse\_summation(float\* data, uint n)

float pairwise\_summation(float\* data, uint n)

В функциях используется цикл *for*, где в переменную *sum* записывается результат суммирования. В функции прямого суммирования складываются все члены ряда из массива *data* по очереди, начиная с начала, в функции обратного суммирования – начиная с конца, в функции попарного суммирования рассматриваются два варианта, в обоих в новый массив *new\_data* записывается результат суммы двух рядом стоящих элементов. Если количество элементов нечётное, то в новый массив *new\_data* добавляется последний элемент из *data*. Затем элементы *new\_data* суммируются в переменной *sum*.

Также создана функция *common*, которая контролирует вызов функций. Она принимает на вход точку *x*, количество членов ряда *n*, указатель на функцию математических функций и указатель на функцию суммирования. Внутри неё вызывается одна из математических функций и функция суммирования, возвращается итог суммирования.

common(float x, uint n, void(\*functions)(float\*mas, uint n, float x), float(\*summations)(float\*mas, uint n))

Функция *choice* осуществляет реализацию выбора пользователем функций. В зависимости от выбора пользователя меняются аргументы у функции common.

choice(int flag\_f, int flag\_s, float x, uint n)

Функция *error\_rate* вычисляет модуль разности получившегося значения и значения в данной точке математической функции из библиотеки *math.h*.

error\_rate(float result, int flag\_f, float x)

# Подтверждение корректности

Для подтверждения корректности в программе я использовала математические функции синуса, косинуса, логарифма и экспоненты из библиотеки *math.h*. С их помощью можно узнать погрешность, с которой удалось вычислить значения математических функций.

# Результаты экспериментов

Рассмотрим по отдельности разные виды суммирования для каждой функции и построим графики зависимости погрешности – ось Y – от количества элементов в ряде – ось X.

, .

Прямое суммирование:

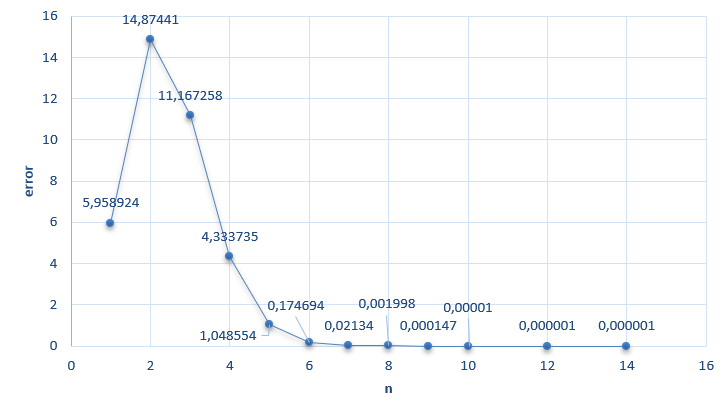


Рисунок 8. График зависимости погрешности от количества элементов в ряде для , прямое суммирование

Обратное суммирование:

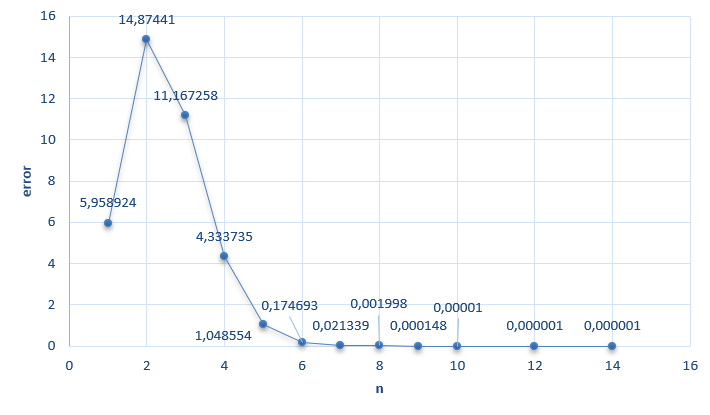


Рисунок 9. График зависимости погрешности от количества элементов в ряде для , обратное суммирование

Попарное суммирование:

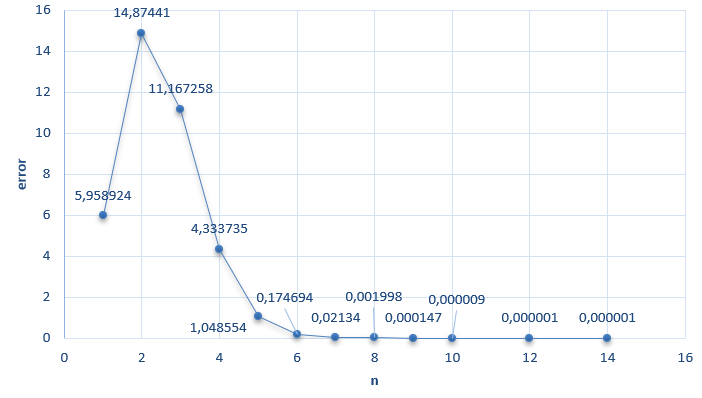


Рисунок 10. График зависимости погрешности от количества элементов в ряде для , попарное суммирование

Значение синуса при использовании разных видов суммирования отличаются в младших разрядах чисел погрешности, что может быть значимым для вычислений, где требуется очень высокая точность. (рис. 8, 9, 10)

Вычисление синуса при увеличении количества элементов ряда Тейлора даёт более точный результат. В моём примере уже при полученное значение достаточно близко к правильному. (рис. 8, 9, 10)

, .

Прямое суммирование:

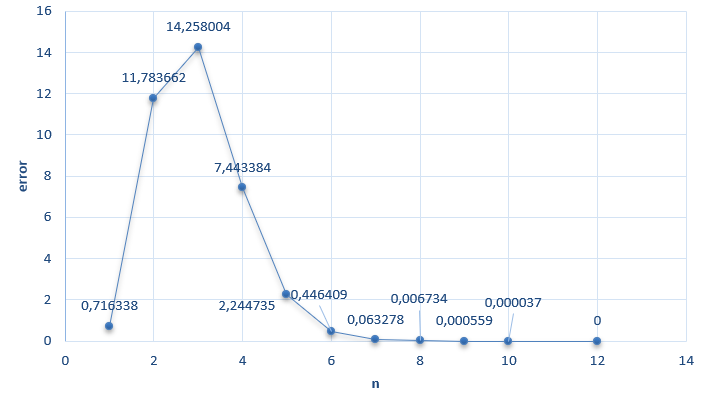


Рисунок 11. График зависимости погрешности от количества элементов в ряде для, прямое суммирование

Обратное суммирование:

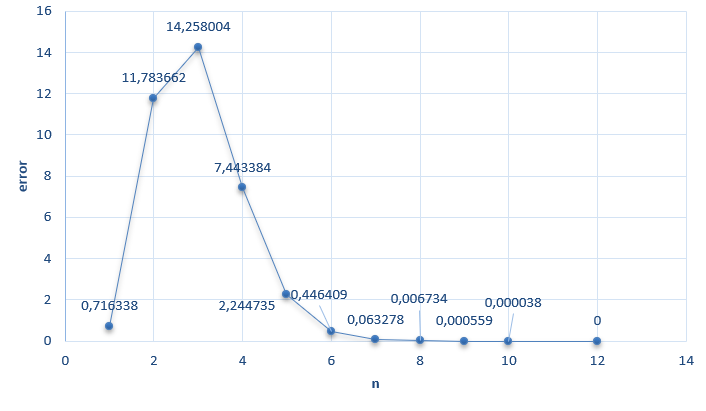


Рисунок 12. График зависимости погрешности от количества элементов в ряде для, обратное суммирование

Попарное суммирование:

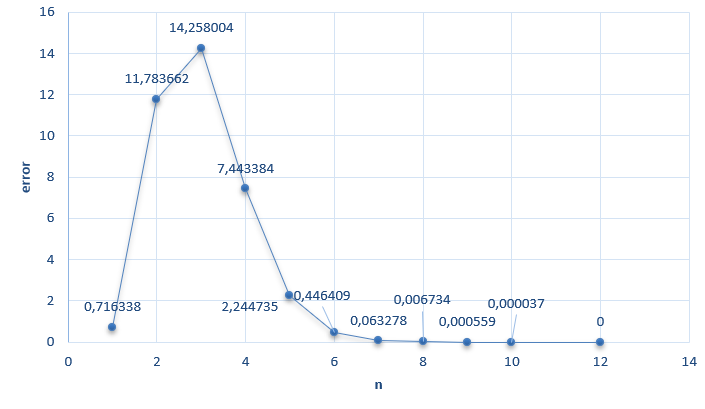


Рисунок 13. График зависимости погрешности от количества элементов в ряде для, попарное суммирование

Значения косинуса тоже приближаются к правильным с увеличением количества элементов ряда Тейлора. При полученное значение уже имеет достаточно маленькую погрешность. (рис. 11, 12, 13)

, .

Прямое суммирование:

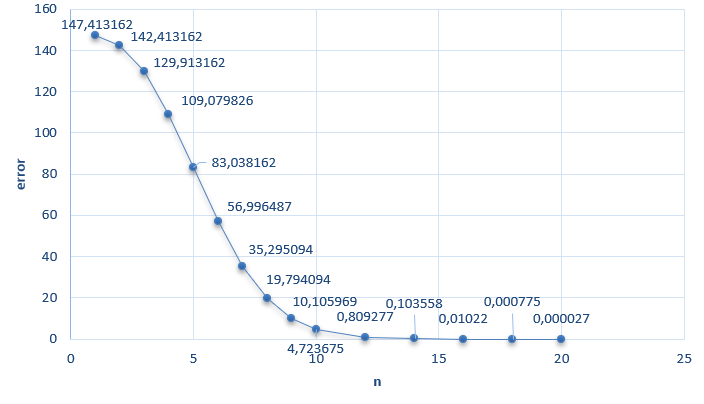


Рисунок 14. График зависимости погрешности от количества элементов в ряде для , прямое суммирование

Обратное суммирование:

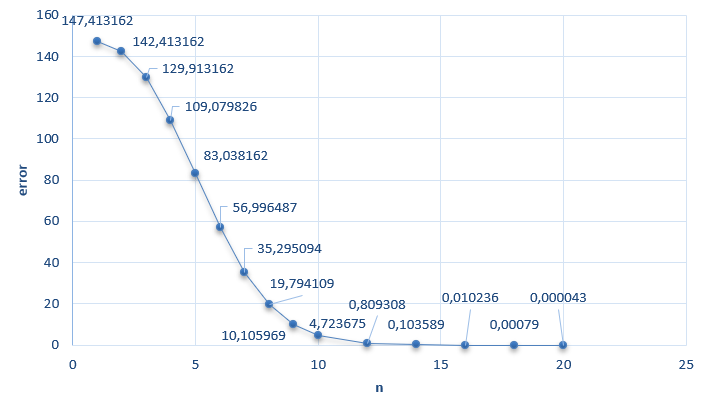


Рисунок 15. График зависимости погрешности от количества элементов в ряде для , обратное суммирование

Попарное суммирование:

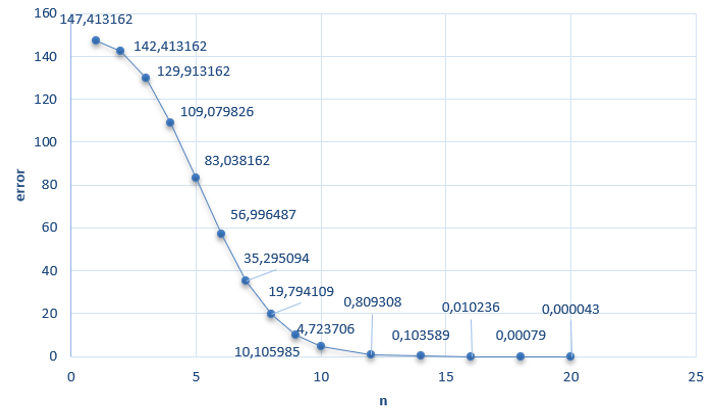


Рисунок 16. График зависимости погрешности от количества элементов в ряде для , попарное суммирование

Значения при разных видах суммирования также имеют отличия только в младших разрядах. При достаточно больших n экспонента близка к истинному значению. (рис. 14, 15, 16)

, .

Прямое суммирование:

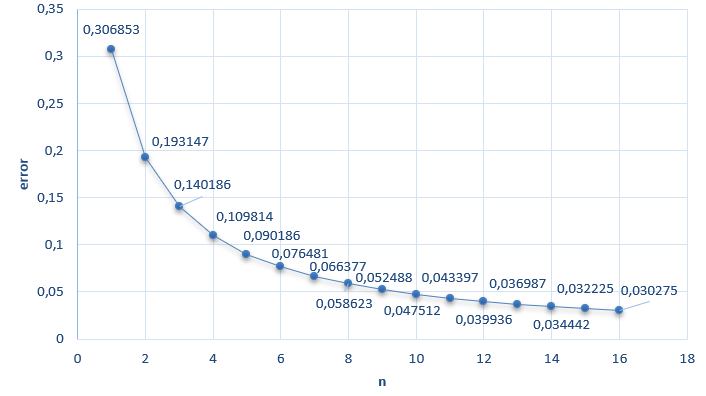


Рисунок 17. График зависимости погрешности от количества элементов в ряде для , прямое суммирование

Обратное суммирование:

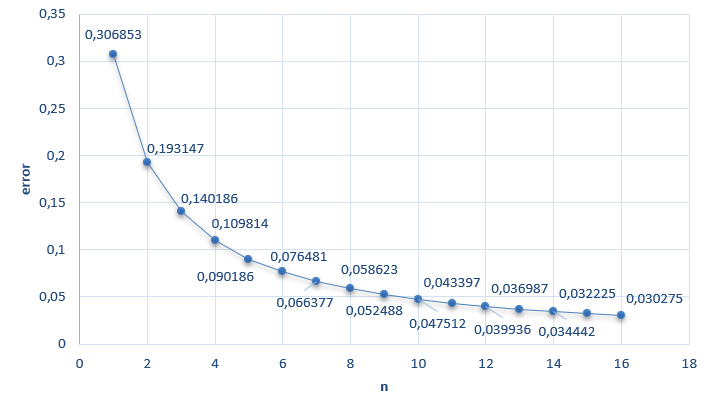


Рисунок 18. График зависимости погрешности от количества элементов в ряде для , обратное суммирование

Попарное суммирование:

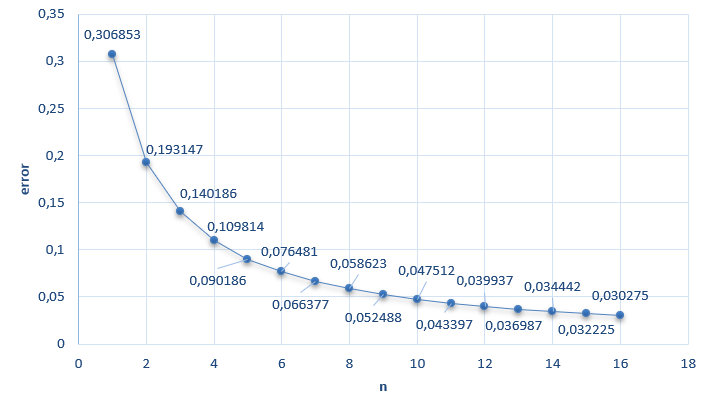


Рисунок 19. График зависимости погрешности от количества элементов в ряде для , попарное суммирование

Логарифм же даже при достаточно больших n при любом суммировании даёт достаточно малую погрешность, но она гораздо больше, чем у синуса и косинуса при том же n. (рис. 17, 18, 19)

Для того, чтобы узнать, какое суммирование показывает более точные результаты для каждой функции, я посчитала среднюю погрешность для каждой функции при разных видах суммирования для фиксированного количества элементов в ряде Тейлора и при изменяющемся x ().

Результаты для :

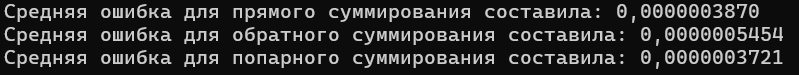


Рисунок 20. Подсчёт средней ошибки для

Для синуса , . При суммировании ряда Тейлора для синуса оказались более точным оказалось попарное суммирование, которое дало меньшую среднюю погрешность. Наименьшую точность дало обратное суммирование.

Результаты для :

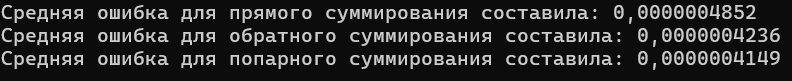


Рисунок 21. Подсчёт средней ошибки для

Для косинуса , . При суммировании ряда Тейлора для косинуса более точным оказалось попарное суммирование, которое дало меньшую среднюю погрешность. Наименьшую точность дало прямое суммирование.

Результаты для :

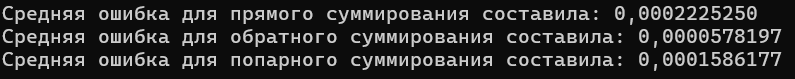


Рисунок 22. Подсчёт средней ошибки для

Для экспоненты , . При суммировании ряда Тейлора для экспоненты более точным оказалось обратное суммирование, а наименее точным – прямое суммирование.

Результаты для :

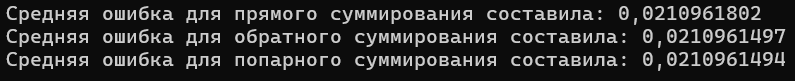


Рисунок 23. Подсчёт средней ошибки для

Для логарифма , . При суммировании ряда Тейлора для логарифма более точным оказалось попарное суммирование, а наименее точным – прямое суммирование.

Таким образом, для всех рассматриваемых математических функций наиболее точным было либо попарное, либо обратное суммирование, а наименее точным – прямое суммирование для всех, кроме синуса.

Результаты данного эксперимента подтверждают, что для функций, в формулах которых у элементов чередуются знаки, предпочтительнее использовать попарное суммирование. Для экспоненты наиболее выгодным было использование обратного суммирования, поскольку не в таком случае дробная часть не теряется.

# Заключение

В ходе лабораторной работы мне удалось вычислить значения , , , с помощью рядов Тейлора с использованием формулы для выражения текущего элемента ряда через предыдущий. Также были реализованы три вида суммирования – прямое, обратное и попарное. По результатам экспериментов выяснилось, что разные виды суммирования влияют на результат вычислений. Для математических функций, члены ряда Тейлора которых чередуют знак, предпочтительнее использовать попарное суммирование, а наименее точным оказалось прямое суммирование.

# Приложение

#include <stdio.h>

#define \_USE\_MATH\_DEFINES

#include <math.h>

#include <locale.h>

#include <stdlib.h>

typedef unsigned int uint;

void sin\_(float\* data, uint n, float x) {

x = fmodf(x, 2.0f \* (float)M\_PI);

data[0] = x;

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

data[i] = ((-1) \* data[i - 1] \* x \* x) / ((2 \* i + 1) \* (2 \* i));

}

}

void cos\_(float\* data, uint n, float x) {

x = fmodf(x, 2.0 \* (float)M\_PI);

data[0] = 1.0;

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

data[i] = ((-1) \* data[i - 1] \* x \* x) / ((2 \* i) \* (2 \* i - 1));

}

}

void exp\_(float\* data, uint n, float x) {

data[0] = 1.0;

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

data[i] = data[i - 1] \* x / i;

}

}

void ln\_(float\* data, uint n, float x) {

data[0] = x;

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

data[i] = ((-1) \* data[i - 1] \* x \* i) / (i + 1);

}

}

float direct\_summation(float\* data, uint n) {

float sum = 0.0;

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

sum = sum + data[i];

}

return sum;

}

float reverse\_summation(float\* data, uint n) {

float sum = 0.0;

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

sum = sum + data[i];

}

return sum;

}

float pairwise\_summation(float\* data, uint n) {

if (n == 1) { return data[0]; }

float sum = 0.0;

int new\_n = (n + 1) / 2;

float\* new\_data = (float\*)malloc(sizeof(float) \* (new\_n));

if ((n % 2) == 0) {

for (int i = 0; i < new\_n; i++) {

new\_data[i] = data[i \* 2] + data[i \* 2 + 1];

}

}

else {

for (int i = 0; i < (new\_n - 1); i++) {

new\_data[i] = data[2 \* i] + data[2 \* i + 1];

}

new\_data[new\_n - 1] = data[n - 1];

}

for (int i = 0; i < new\_n; i++) {

sum = sum + new\_data[i];

}

free(new\_data);

return sum;

}

float common(float x, uint n, void(\*functions)(float\*mas, uint n, float x), float(\*summations)(float\*mas, uint n)){

float\* data = (float\*)malloc(sizeof(float) \* n);

float result;

functions(data, n, x);

result = summations(data, n);

free(data);

return result;

}

float choice(int flag\_f, int flag\_s, float x, uint n) {

if (flag\_f == 1) {

if (flag\_s == 1) { return common(x, n, sin\_, direct\_summation); }

if (flag\_s == 2) { return common(x, n, sin\_, reverse\_summation); }

if (flag\_s == 3) { return common(x, n, sin\_, pairwise\_summation); }

}

if (flag\_f == 2) {

if (flag\_s == 1) { return common(x, n, cos\_, direct\_summation); }

if (flag\_s == 2) { return common(x, n, cos\_, reverse\_summation); }

if (flag\_s == 3) { return common(x, n, cos\_, pairwise\_summation); }

}

if (flag\_f == 3) {

if (flag\_s == 1) { return common(x, n, exp\_, direct\_summation); }

if (flag\_s == 2) { return common(x, n, exp\_, reverse\_summation); }

if (flag\_s == 3) { return common(x, n, exp\_, pairwise\_summation); }

}

if ((flag\_f == 4) && (x > -1) && (x <= 1)) {

if (flag\_s == 1) { return common(x, n, ln\_, direct\_summation); }

if (flag\_s == 2) { return common(x, n, ln\_, reverse\_summation); }

if (flag\_s == 3) { return common(x, n, ln\_, pairwise\_summation); }

}

else { printf("Недопустимое значение х (х <= -1)\n"); }

return 0.0;

}

double error\_rate(float result, int flag\_f, float x) {

float error;

if (flag\_f == 1) {

return fabs(sin(x) - (double)result);

}

if (flag\_f == 2){

return fabs(cos(x) - (double)result);

}

if (flag\_f == 3) {

return fabs(exp(x) - (double)result);

}

if (flag\_f == 4) {

return fabs(log(x + 1) - (double)result);

}

}

int main()

{

setlocale(LC\_ALL, "rus");

float error;

float result;

double result\_1 = 0.0, result\_2 = 0.0, result\_3 = 0.0;

int flag\_f;

int flag\_s;

float x;

uint n;

printf("Выберите функцию:\n1 - sin(x)\n2 - cos(x)\n3 - exp(x)\n4 - ln(1+x)\n");

printf("Введите цифру, соответствующую нужной функции: ");

scanf\_s("%d", &flag\_f);

if ((flag\_f <= 0)||(flag\_f > 4)){printf("Недопустимое значение."); }

else {

printf("Выберите вид суммирования:\n1 - Прямое суммирование\n2 - Обратное суммирование\n3 - Попарное суммирование\n");

printf("Введите цифру, соответствующую нужному суммированию: ");

scanf\_s("%d", &flag\_s);

if ((flag\_s <= 0) || (flag\_s > 3)) { printf("Недопустимое значение."); }

else {

printf("Введите значение х для функции: ");

scanf\_s("%f", &x);

printf("Введите количество элементов в ряде: ");

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

if (n <= 0) { printf("Недопустимое значение."); }

else {

result = choice(flag\_f, flag\_s, x, n);

printf("Результат суммирования: %f\n", result);

error = error\_rate(result, flag\_f, x);

printf("Погрешность составила: %f\n", error);

}

}

}