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 Тейлора»**

**Выполнил**:

студент группы 3824Б1ПМ1

Котельников И.Е.

**Проверил**:

преподаватель каф. ВВСП,

Волокитин В.Д.

Нижний Новгород

2025

**Содержание**

[Постановка задачи 3](#_Toc14085)

[Метод решения 4](#_Toc21854)

[Руководство пользователя 5](#_Toc3015)

[Описание программной реализации 8](#_Toc11747)

[Подтверждение корректности 10](#_Toc19062)

[Результаты экспериментов 11](#_Toc13111)

[Заключение 28](#_Toc10235)

[Приложение 29](#_Toc15686)

# Постановка задачи

Необходимо реализовать вычисление функций , , , с помощью рядов Тейлора и сравнить различные способы подсчёта, такие как прямое, обратное и попарное суммирования. Сравнить результаты работы программной реализации с соответствующими значениями функций встроенных в библиотеку math.h от того же аргумента. Изучить зависимость значения абсолютной погрешности от количества членов в ряду. Изучить зависимость значения абсолютной погрешности от значения аргумента (его отдалённости от точки разложения, в нашем случае x = 0).

# Метод решения

В первую очередь стоит обратить внимание на принцип вычисления элементов рядов. В разложении функции по Тейлору на полином используется вычисление производной n-ого порядка. Поскольку вычислять производную в данном случае не рационально, было взято базовое разложение по Маклорену. Используя его можно вывести соответствующие формулы N + 1 элемента, тем самым избавившись от необходимости брать производную. Вот они:

Sin: Cos: Exp: Ln:

*Примечание: В разложении логарифма используется формула ln(1 + x), но программа принимает аргумент целиком, поэтому формула последующего члена выглядит так и именно поэтому диапазон применения программы для логарифма (0; 2].*

В силу необходимости для вычисления и предпоследнего элемента, общий прототип всех функций был дополнен этим элементом. Можно было обойтись уменьшением длины полинома в 2 раза, но тогда бы пришлось добавить лишнее условие в функцию сложения, что было бы, возможно, несколько труднее. В силу этой особенности, перед заполнением массива ряда программа высчитывает нулевой и первый элементы ряда.

Общий принцип решения состоит в том, что программа заполняет некоторый массив элементами полинома, после чего суммирует их различными способами и сравнивает полученный каждым способом результат с идеалом, которым является значение встроенной библиотечной функции от того же аргумента. После программа сравнивает абсолютные значения отклонения и выбирает лучшим вариантом то суммирование, у которого оно меньше.

Если с прямым и обратным суммированием все достаточно просто, то попарное требует некоторого разъяснения в силу специфики реализации. Оно выполнено полностью древовидным образом, исключая последний элемент, который может быть неспаренным. В таком случае он добавляется к предпоследнему (относительно шага этой итерации), делая таким образом этот элемент больше прочих (в некоторых случаях).

# Руководство пользователя

При запуске программы у пользователя появляется 4 варианта функции, которую будет вычислять программа:

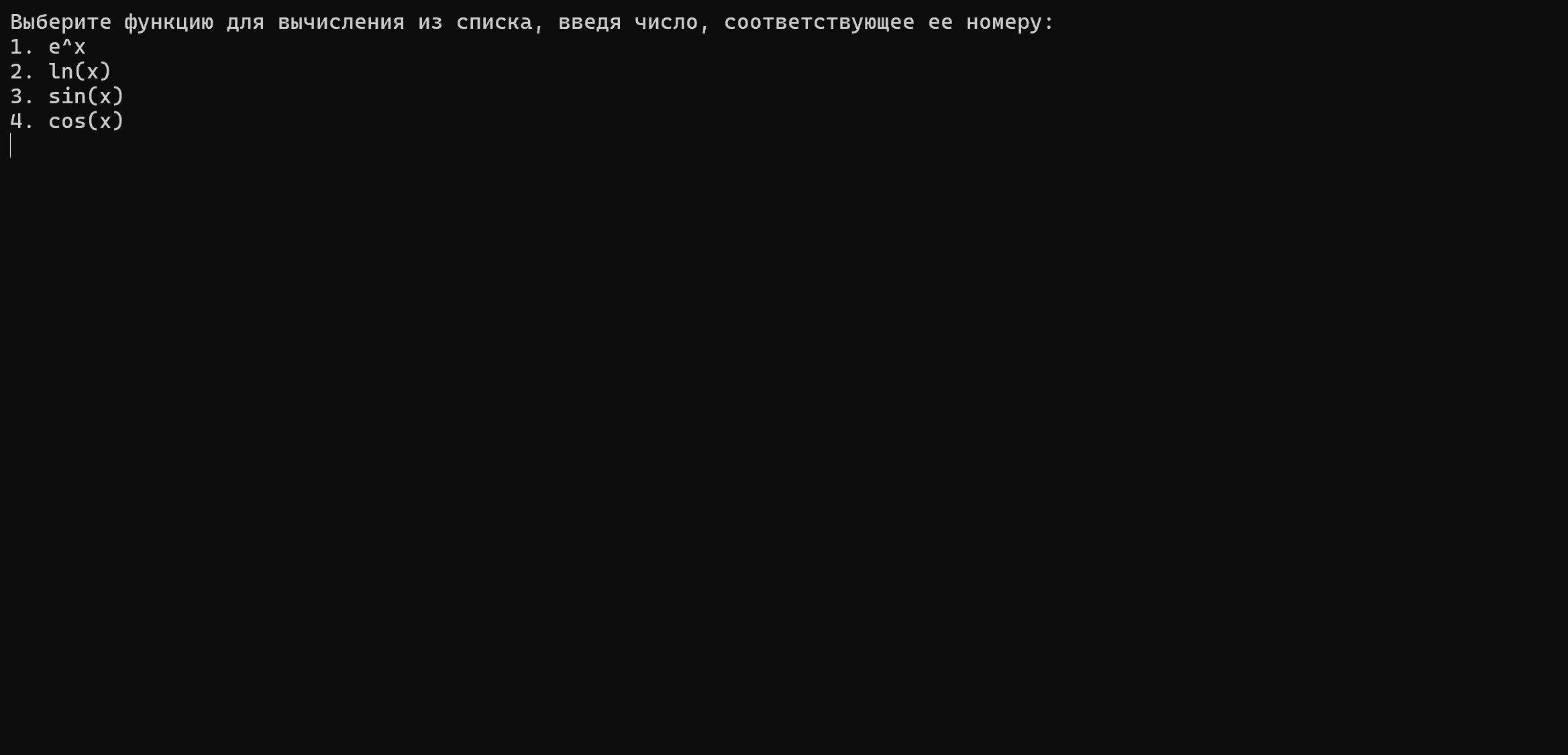


Рис. 1

Если введено значение не принадлежащее списку, то программа сообщит об ошибке и потребует ввести номер функции повторно:

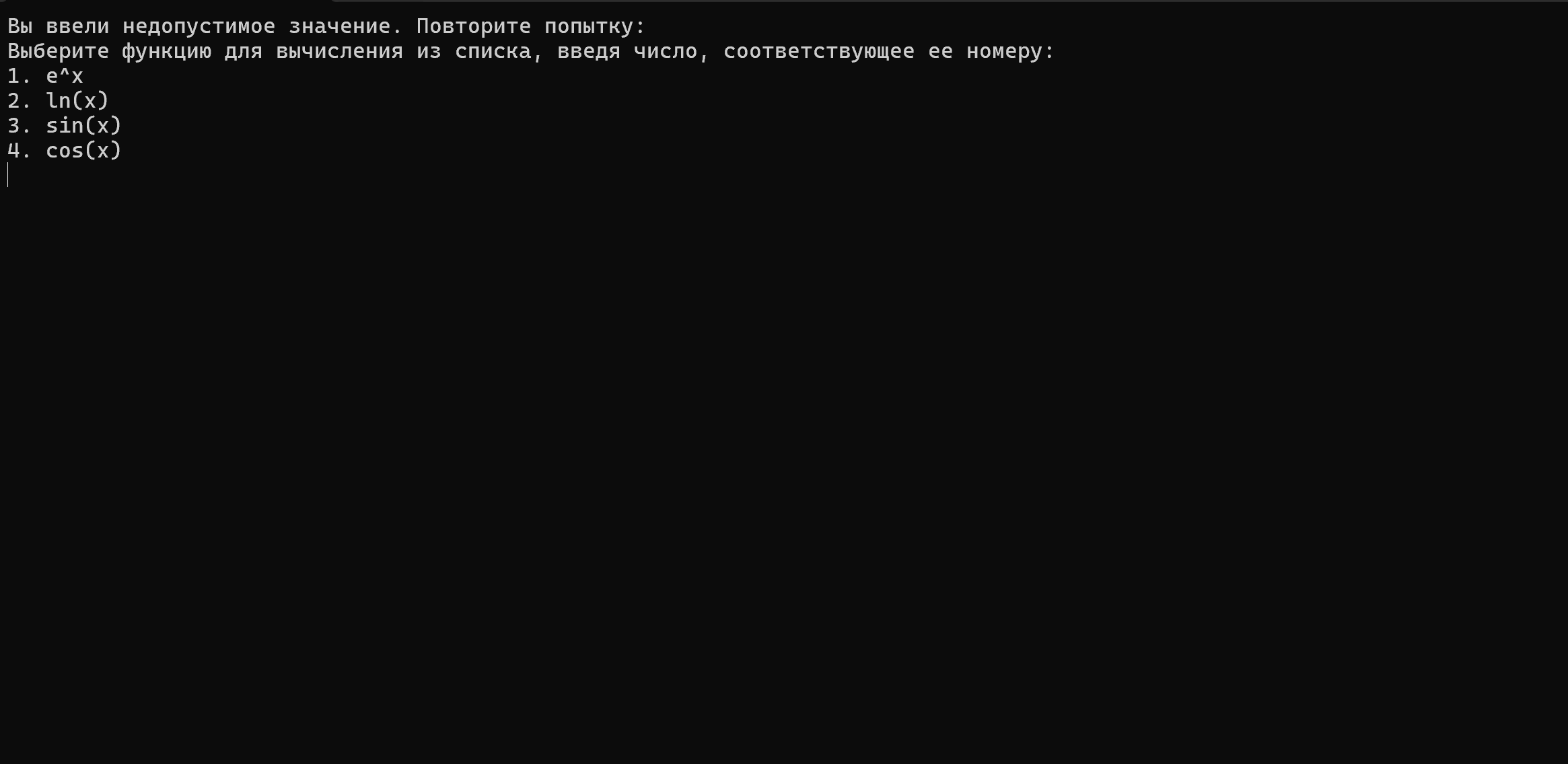


Рис. 2

После выбора функции программа попросит ввести аргумент функции. В случае с логарифмом уточняется наличие ограничения на аргумент.

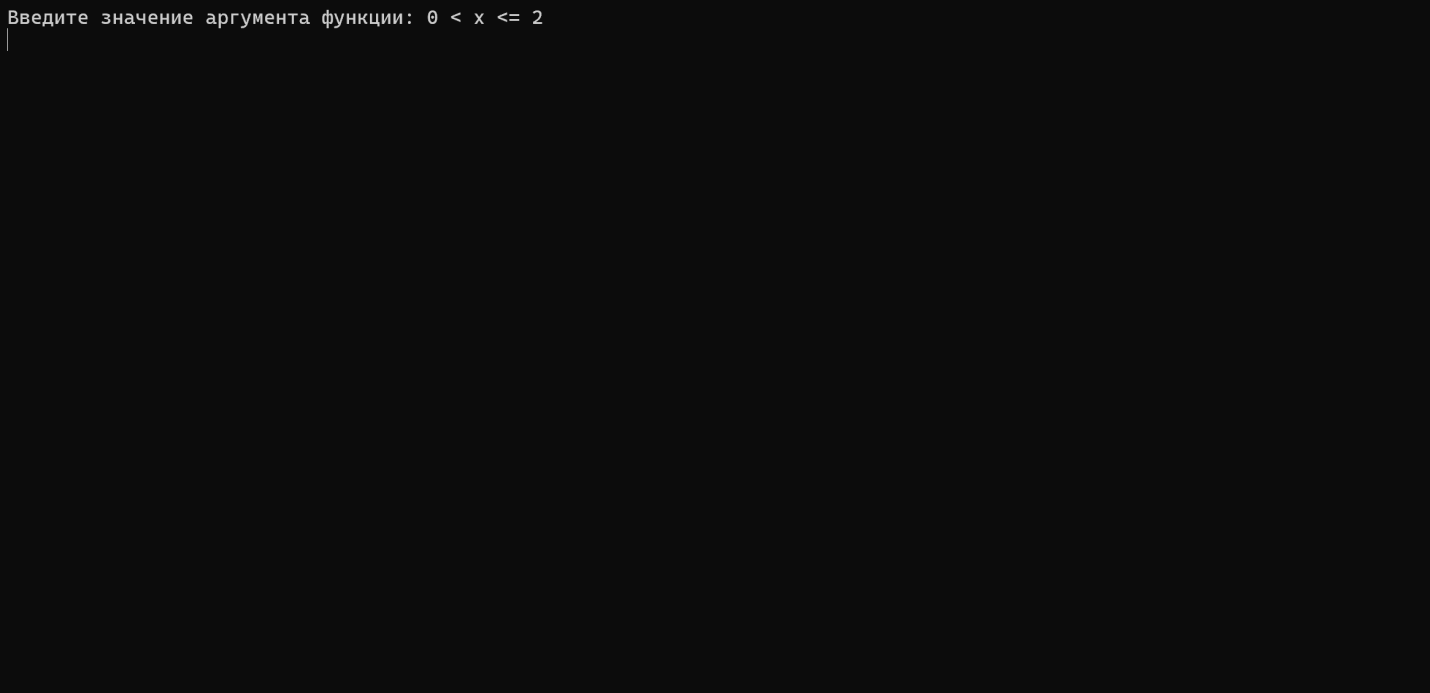


Рис. 3

Далее программа просит вести количество членов в разложении:



Рис. 4

После этого программа выводит результат своей работы:

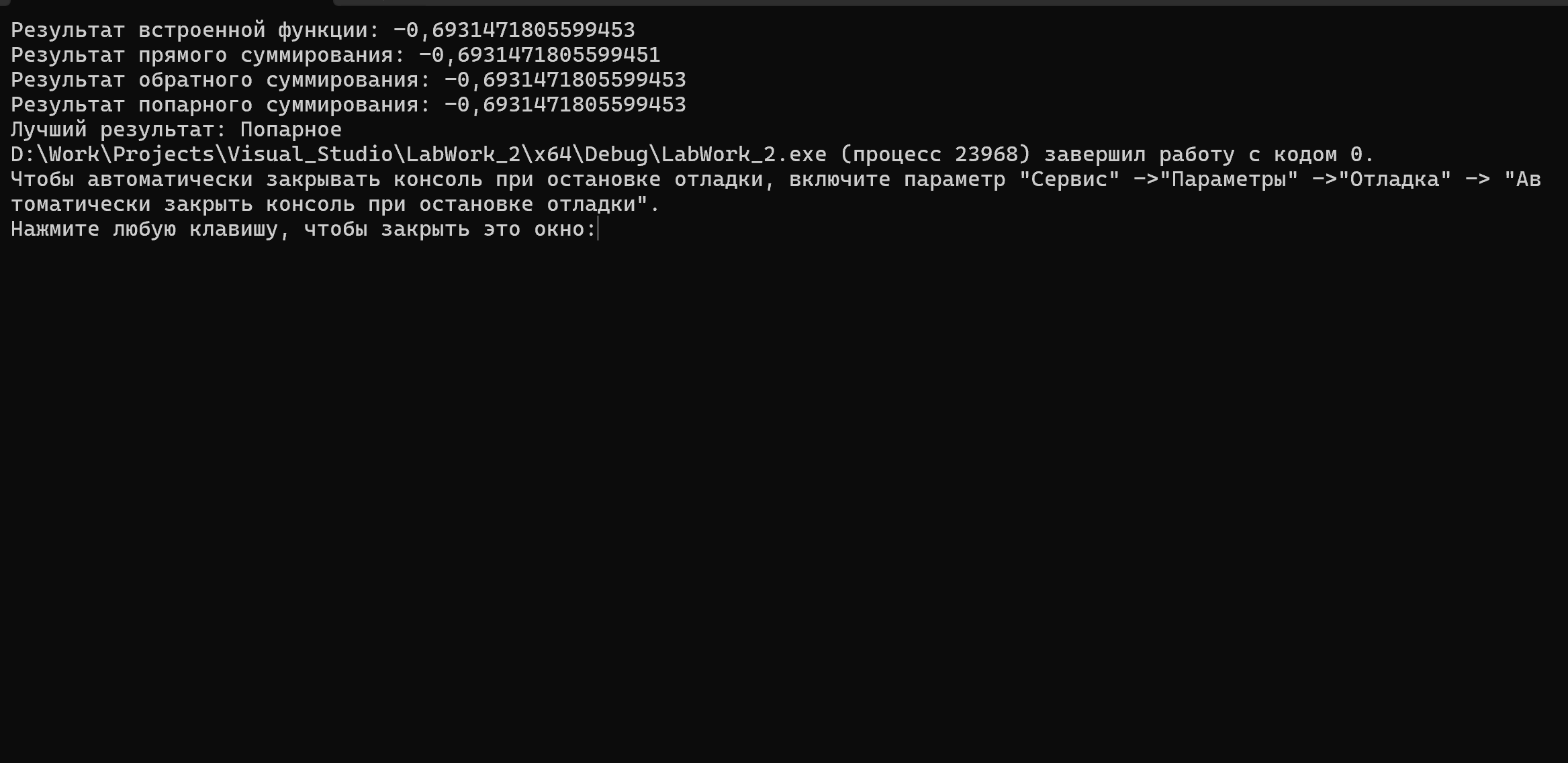


Рис. 5

# Описание программной реализации

Группы функций и их прототипы:

1. Функции вычисления следующего элемента:

double exponent(double element, double previousElement, double prePreviousElement, int counter, double argument)

double logarithm(double element, double previousElement, double prePreviousElement, int counter, double argument)

double theSine(double element, double previousElement, double prePreviousElement, int counter, double argument)

double theCosine(double element, double previousElement, double prePreviousElement, int counter, double argument)

1. Функции суммирования:

double straight(double (\*next)(double, double, double, int, double), double\* polynome, int length, double argument)

double reverse(double\* polynome, int length)

double couple(double\* polynome, int length)

1. Основная функция:

int main()

1. Указатель на функцию:

double (\*func)(double, double, double, int, double);

Функции вычисления следующего элемента получают на вход ячейку массива, в которую будет записан элемент полинома, предыдущий элемент, предпредыдущий элемент, порядковый номер текущего элемента и аргумент, от которого зависит значение элемента. В первых двух функциях не используется предпредыдущий элемент, в двух последних предыдущий. В зависимости от выбранной функции из 4 представленных, указатель *func()* принимает значение функции следующего элемента и передаёт его в функцию суммирования *straight()*. В силу того, что все виды суммирований выполняются в любом случае, в качестве экономии места и времени *straight()* сразу заполняет массив элементами полинома, избавляя последующие суммирования от вычисления мономов. Поэтому *reverse()* и *couple()* получают на вход только полином и его длину. *Reverse()* выполняет простое суммирование с конца до начала, а вот *couple()* можно разобрать несколько подробнее. В силу необходимости изменения элементов массива, что связано с подобием обратной рекурсии в работе алгоритма, он сначала проходит по всему полиному и создаёт его копию. После он начинает идти по массиву с шагом равным 2, начиная с элемента (шаг - 1), прибавляя к выбранному элементу предыдущий (по половинному шагу) и зануляя ячейку с прибавленным значением. Последний элемент может оказаться неспаренным, но он будет существовать только если следующий по шагу индекс элемента будет уже вне массива, но при этом элементов останется больше или равно половине шага (по количеству). В таком случае индекс последнего попавшего в шаг элемента запоминается и после к нему прибавляется следующий за ним элемент (по половинному шагу).

Основная функция *main()* не получает ничего на вход, она запускает все остальные функции и выводит результат.

# Подтверждение корректности

Для подтверждения корректности в программе проводится сравнение полученных в результате каждого из суммирований результатов со значениями встроенных функций библиотеки math.h от того же аргумента. После программа определяет значение, имеющее наименьшее отклонение от «идеала» и выводит название его метода как лучшего.

Если вдруг человек выберет несуществующую в списке функцию, то программа потребует от него ввести аргумент повторно, сообщив об ошибке.

# Результаты экспериментов

Первым делом экспериментально стоит оценить количество членов в полиноме, начиная с которого погрешность станет незначительной. Рассмотрим каждую из 4 функций. Будем использовать значение аргумента x0 = 0,5. В качестве максимального количества мономов возьмём число 20:

1. Экспонента:

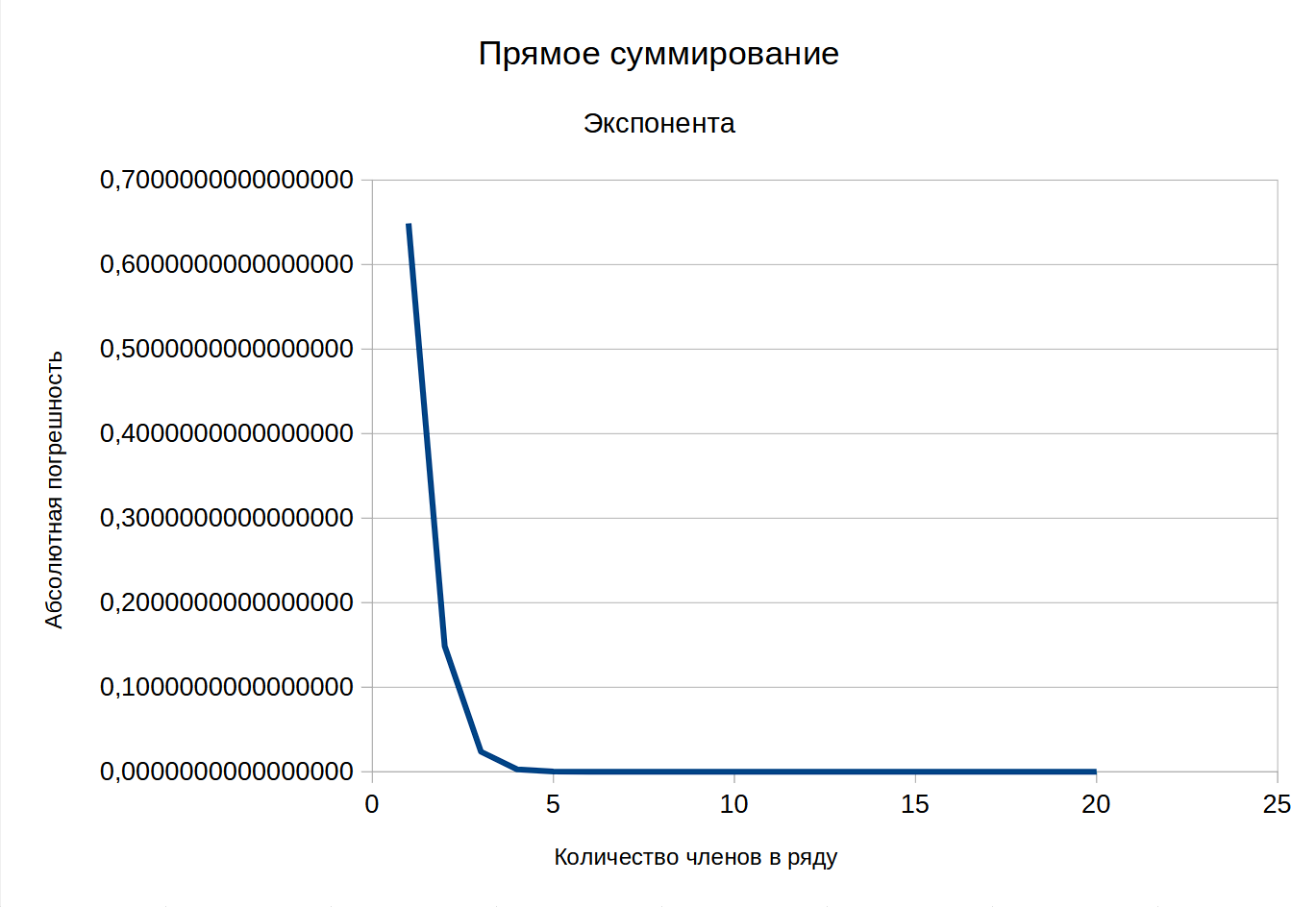


Рис. 6

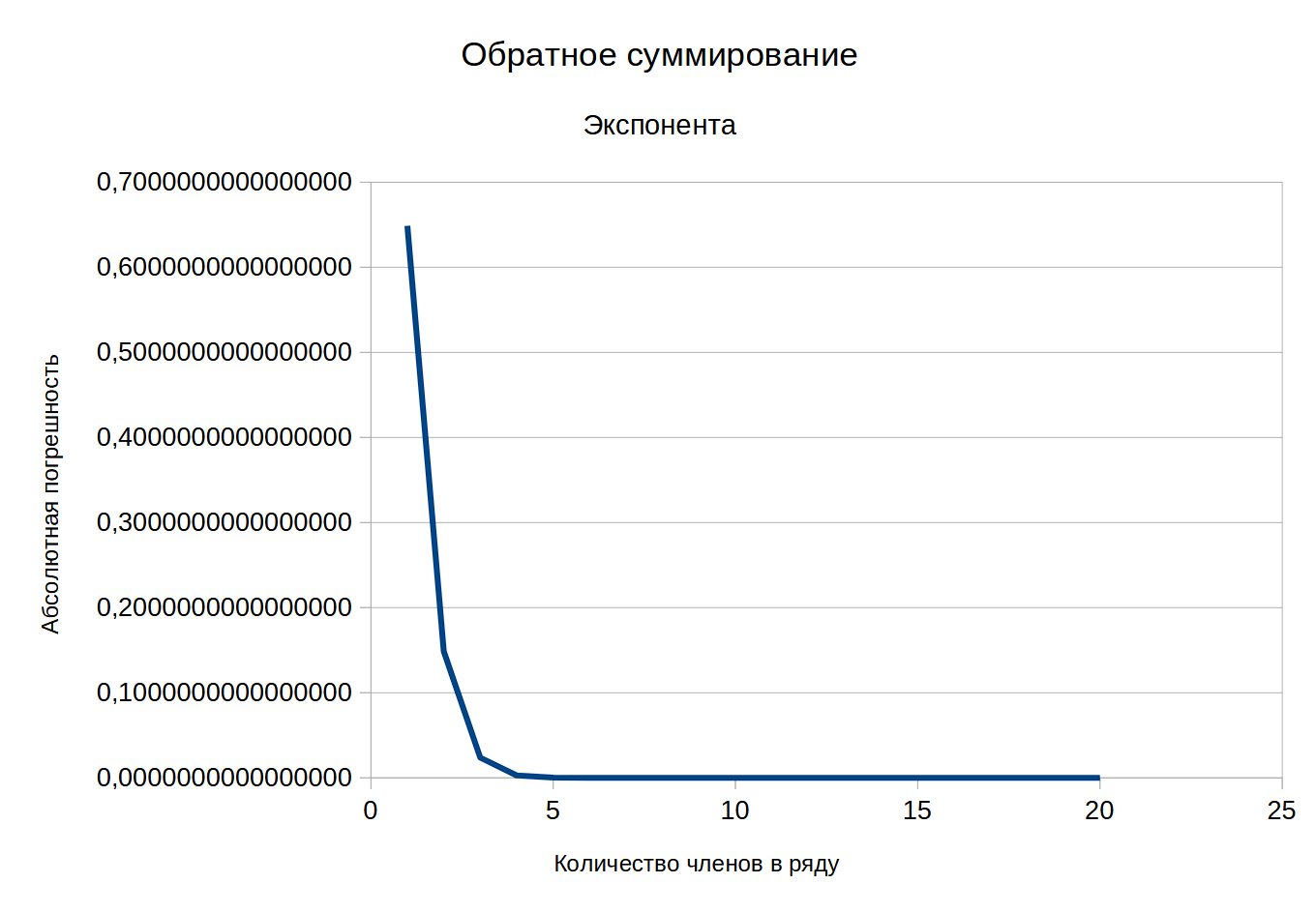


Рис. 7

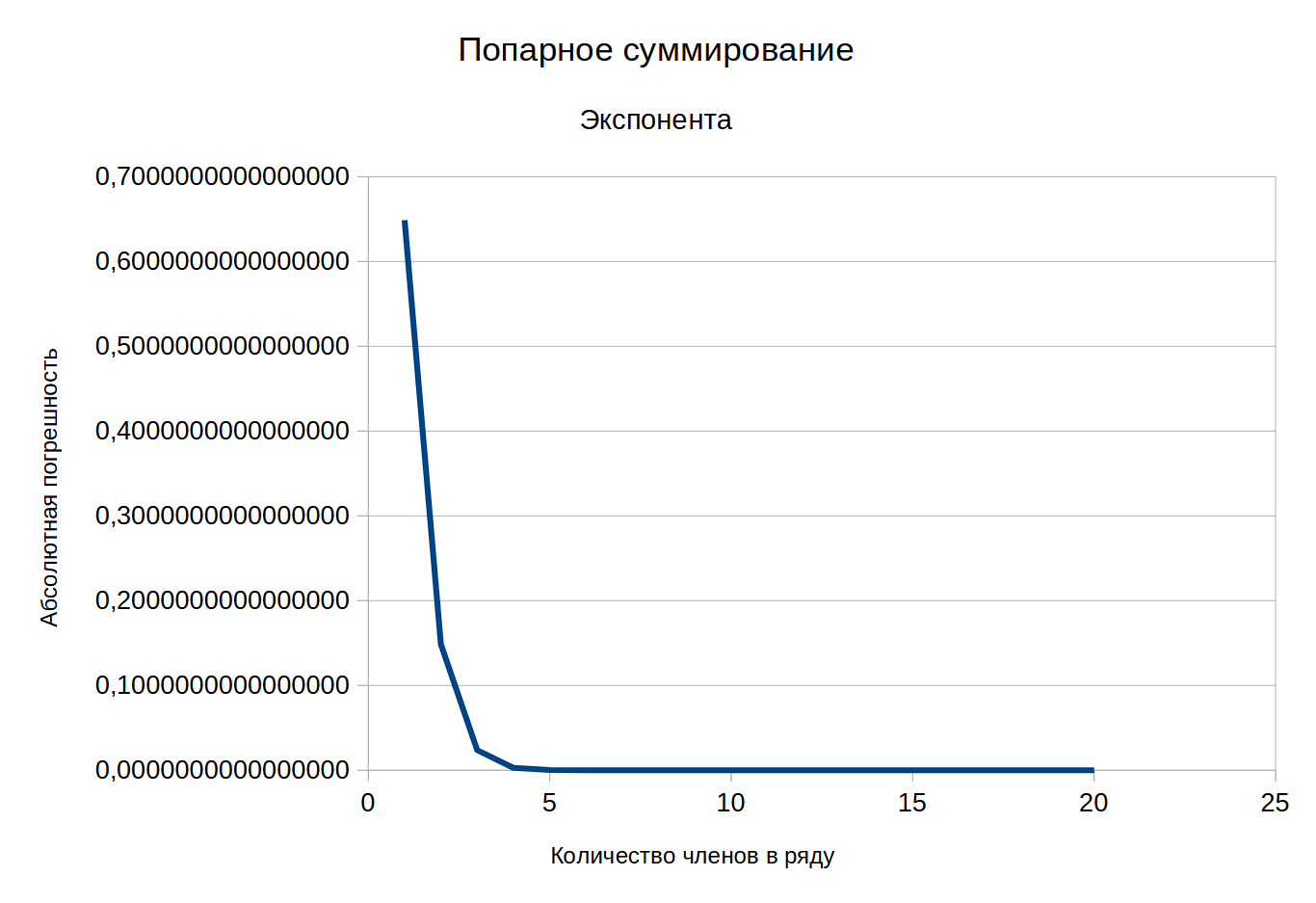


Рис. 8

1. Логарифм:

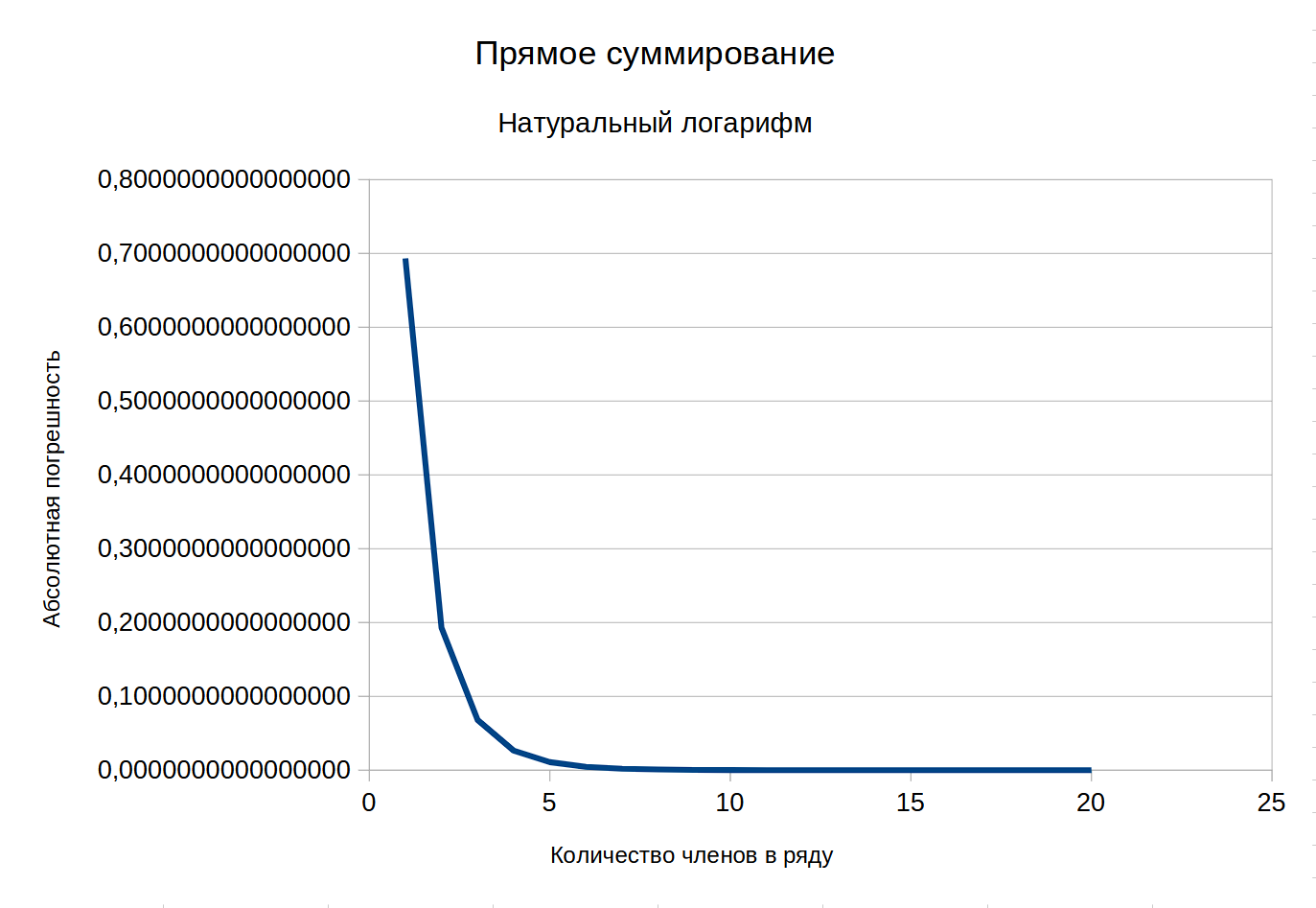


Рис. 9

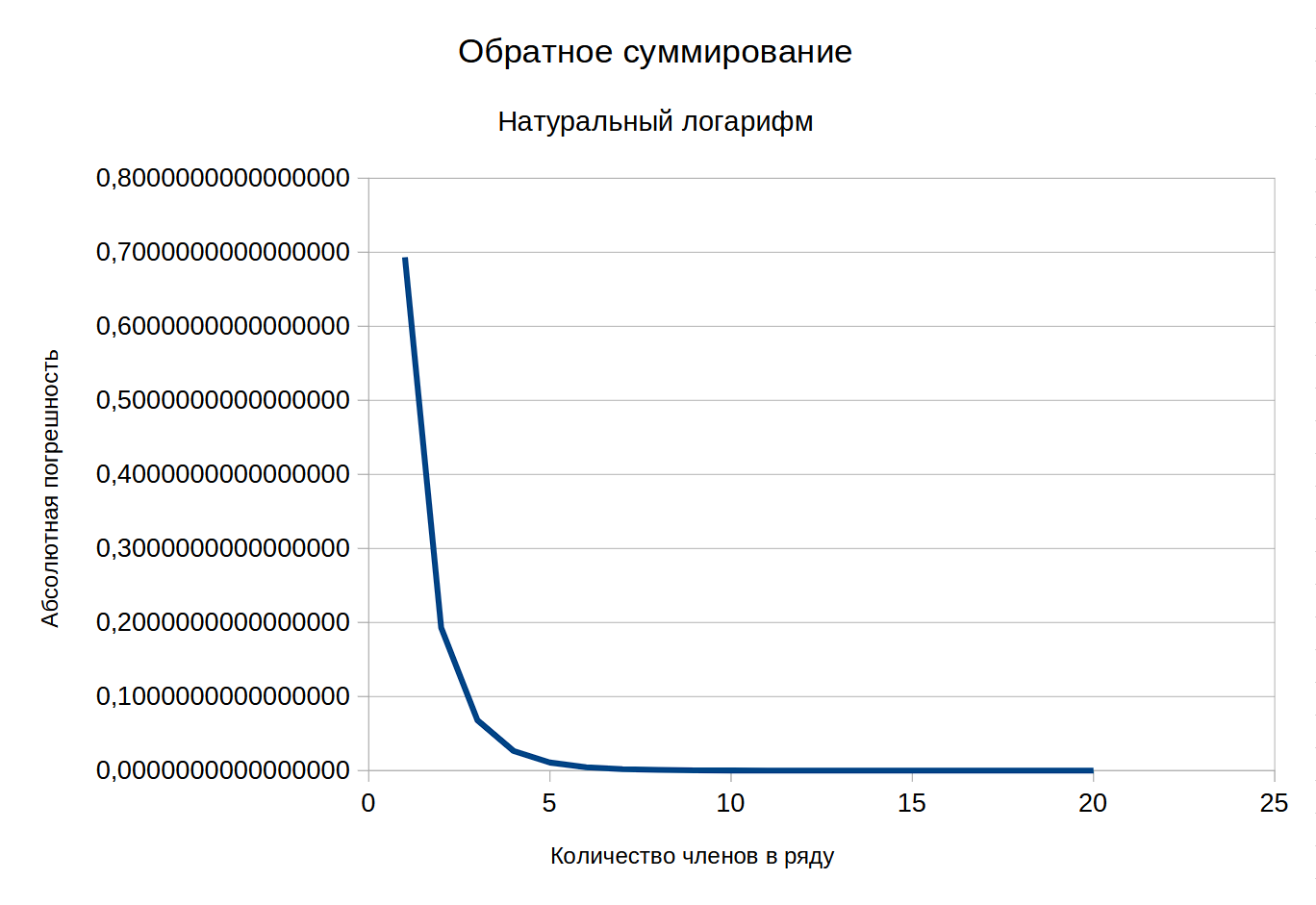


Рис. 10

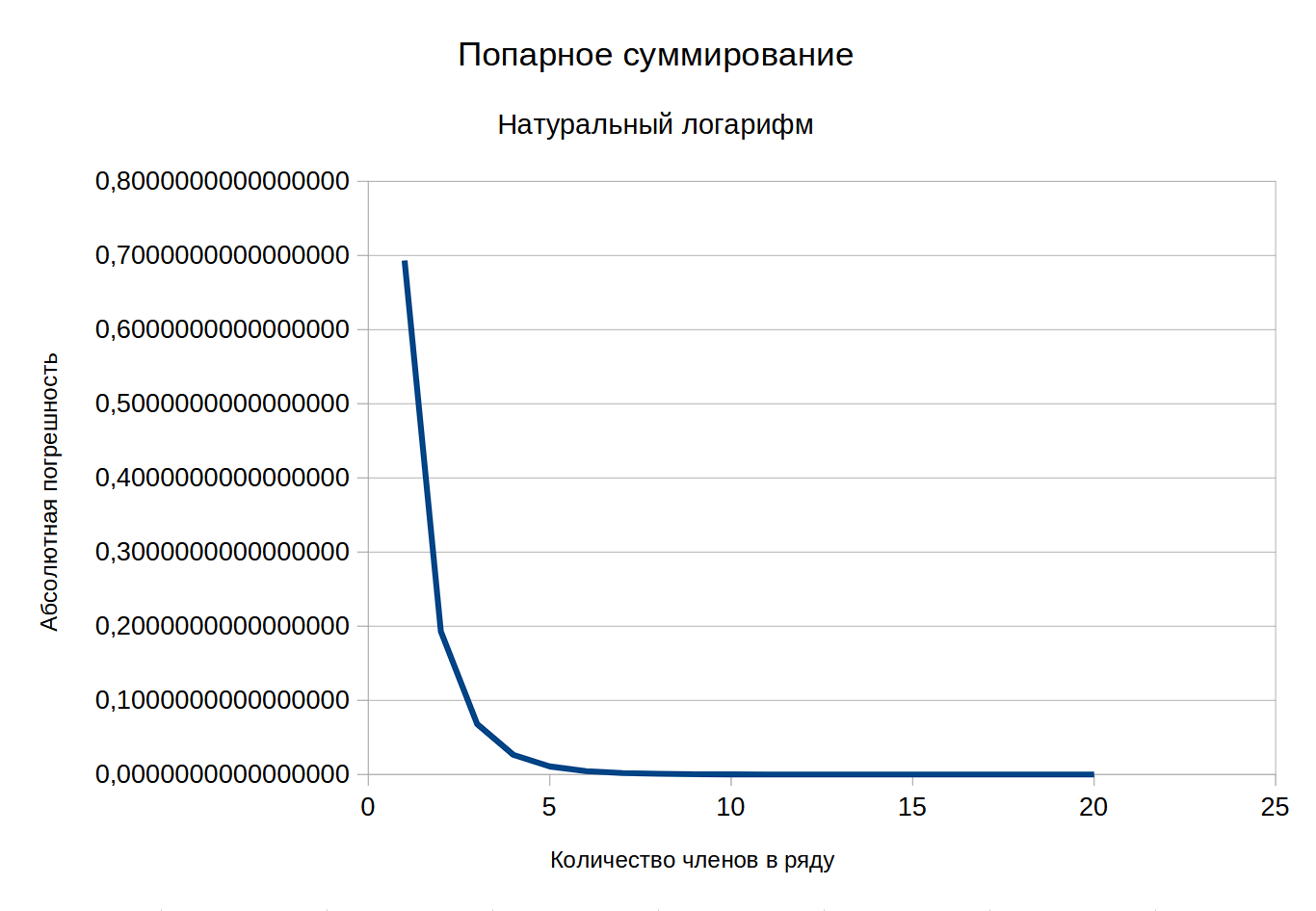


Рис. 11

1. Синус:

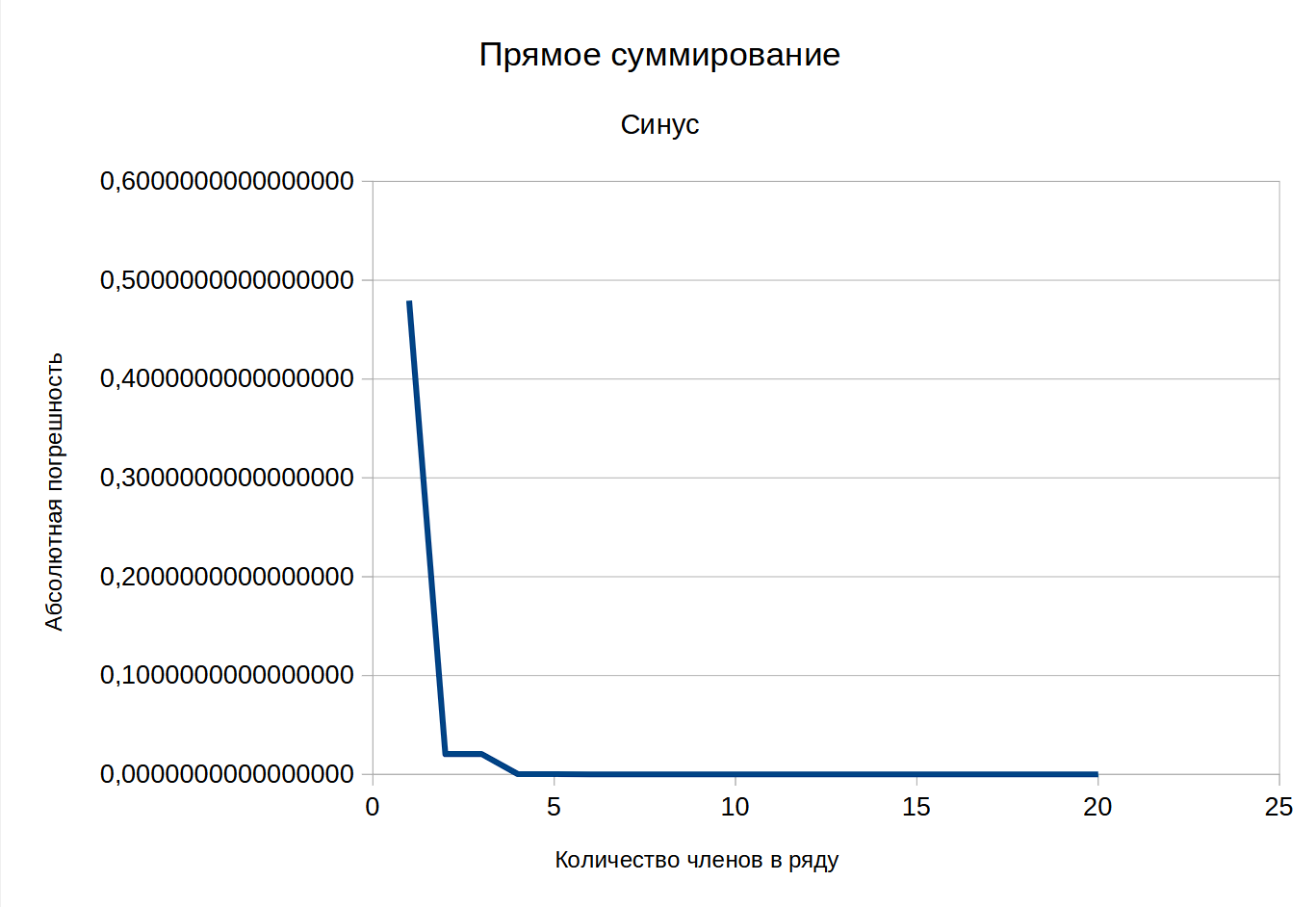


Рис. 12

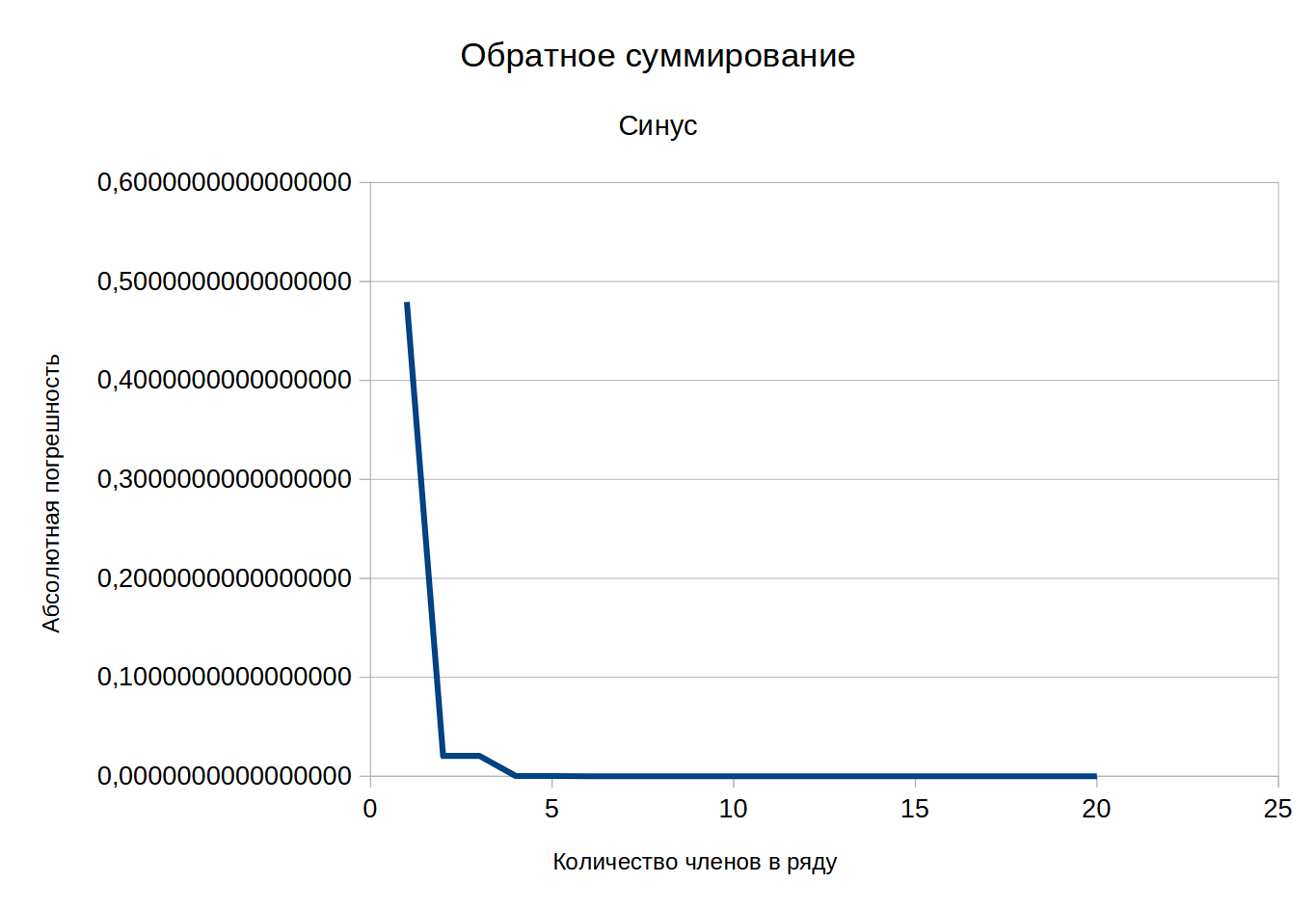


Рис. 13

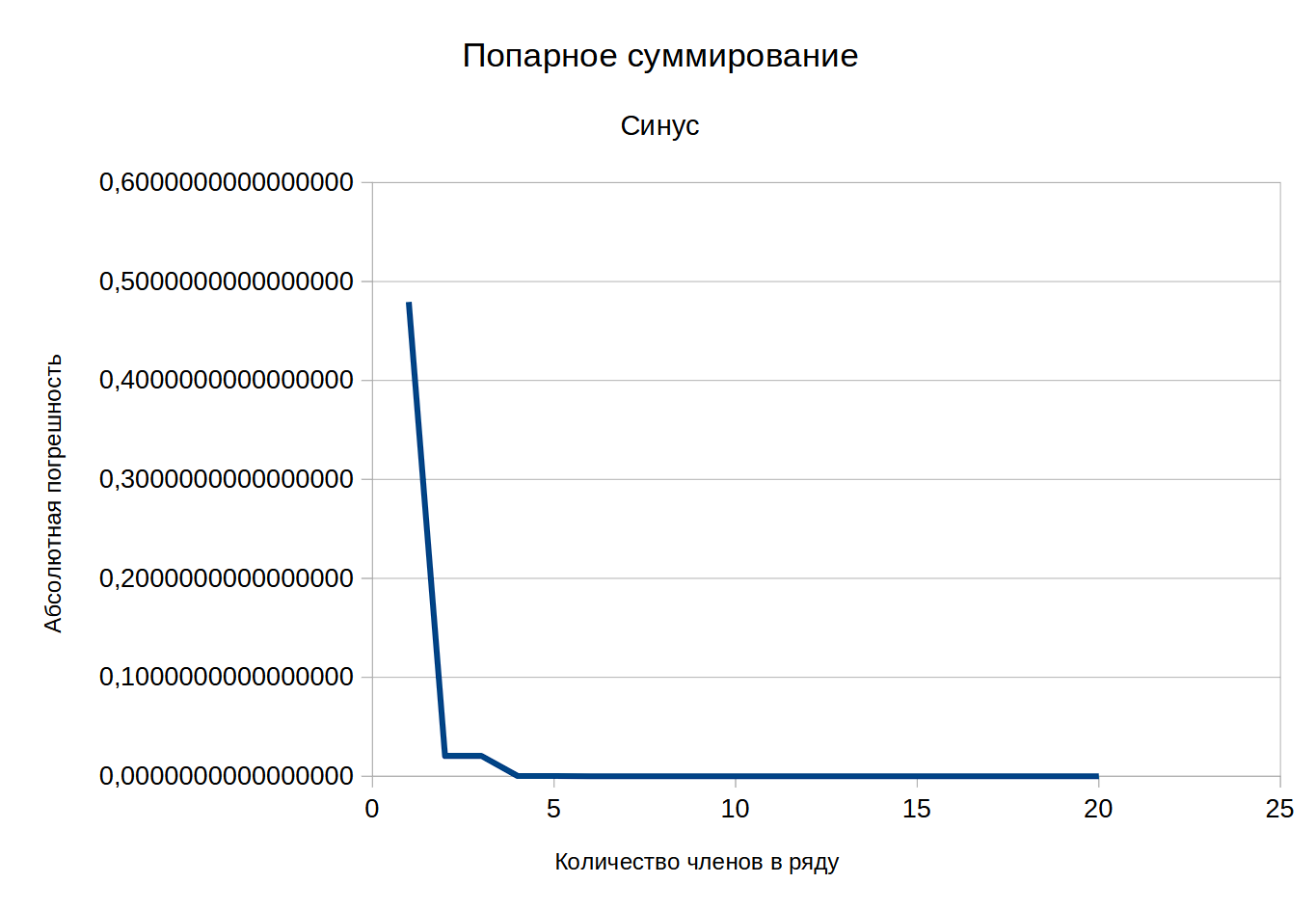


Рис. 14

1. Косинус:

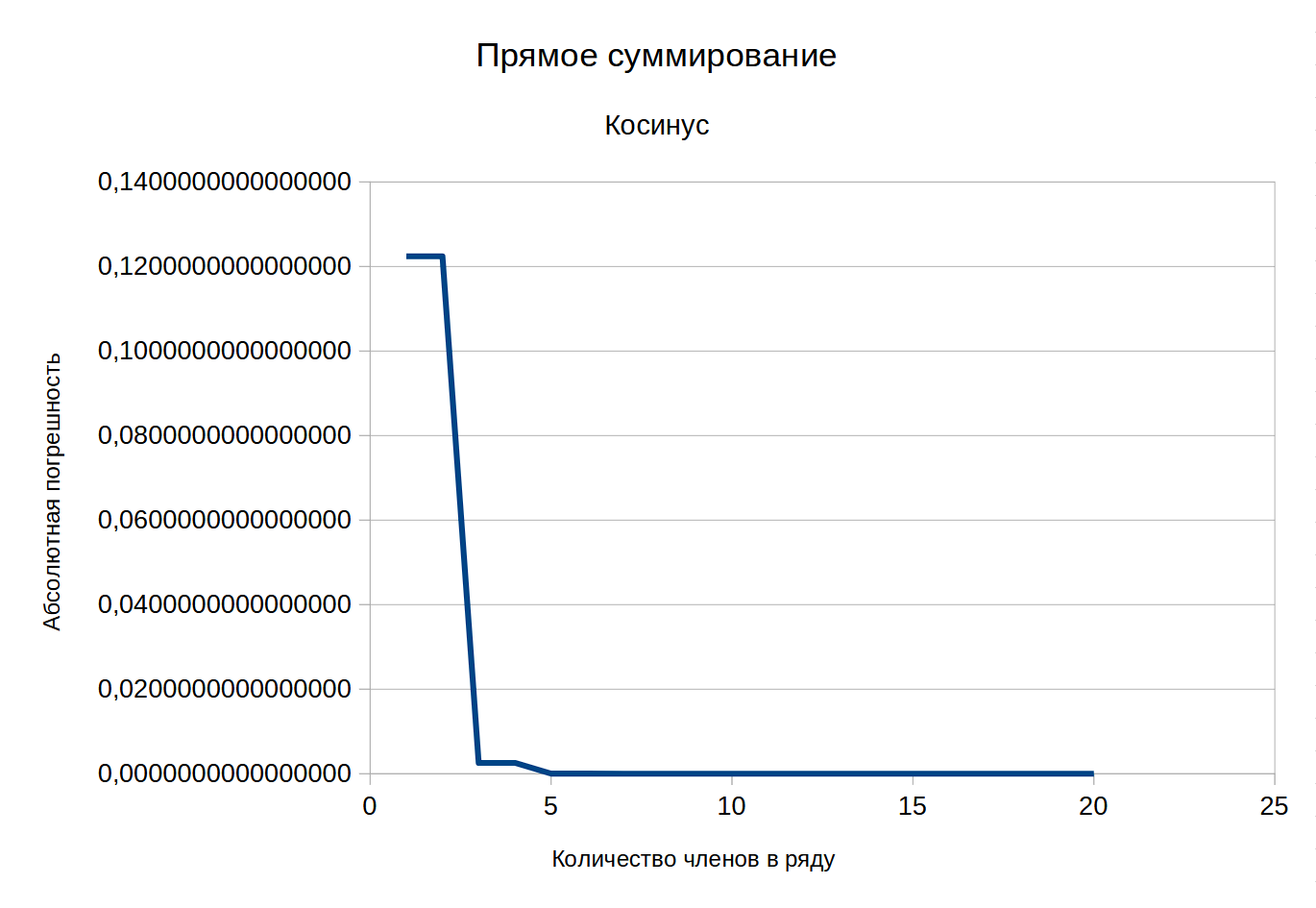


Рис. 15

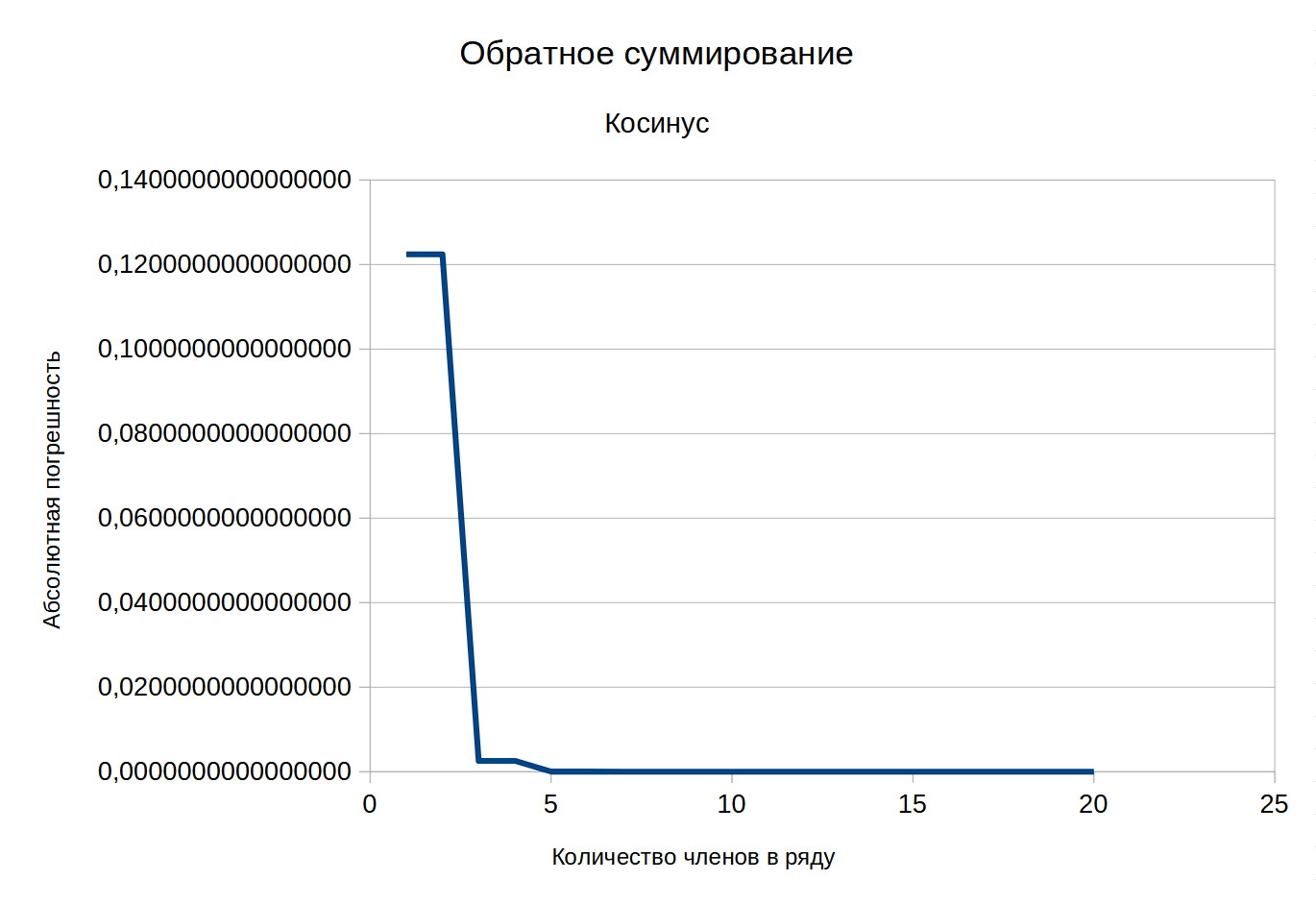


Рис. 16

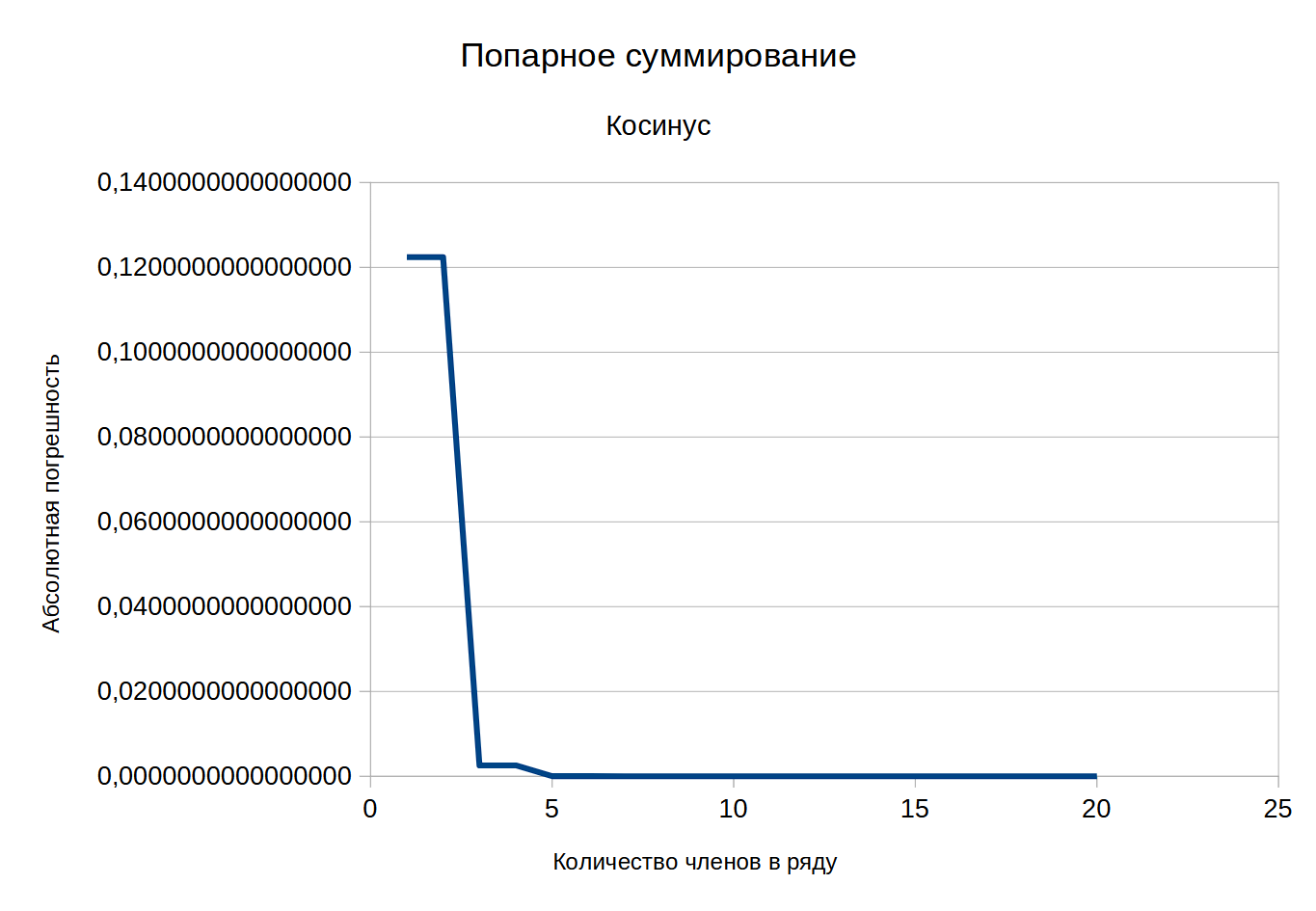


Рис. 17

Из графиков видно, что в среднем уже 5 членов ряда достаточно, чтобы погрешность была визуально незаметна. Исключая натуральный логарифм, у него уменьшение погрешности выражено менее явно, так как диапазон аргумента, применимого для вычисления следующий: (0; 2]. В силу полученной информации для второго пункта лабораторной работы будем брать 20 членов в разложении по Маклорену, исключая логарифм, из-за небольшого диапазона такое количество членов даст плохую наглядность, поэтому там будем использовать 10 членов.

1. Экспонента

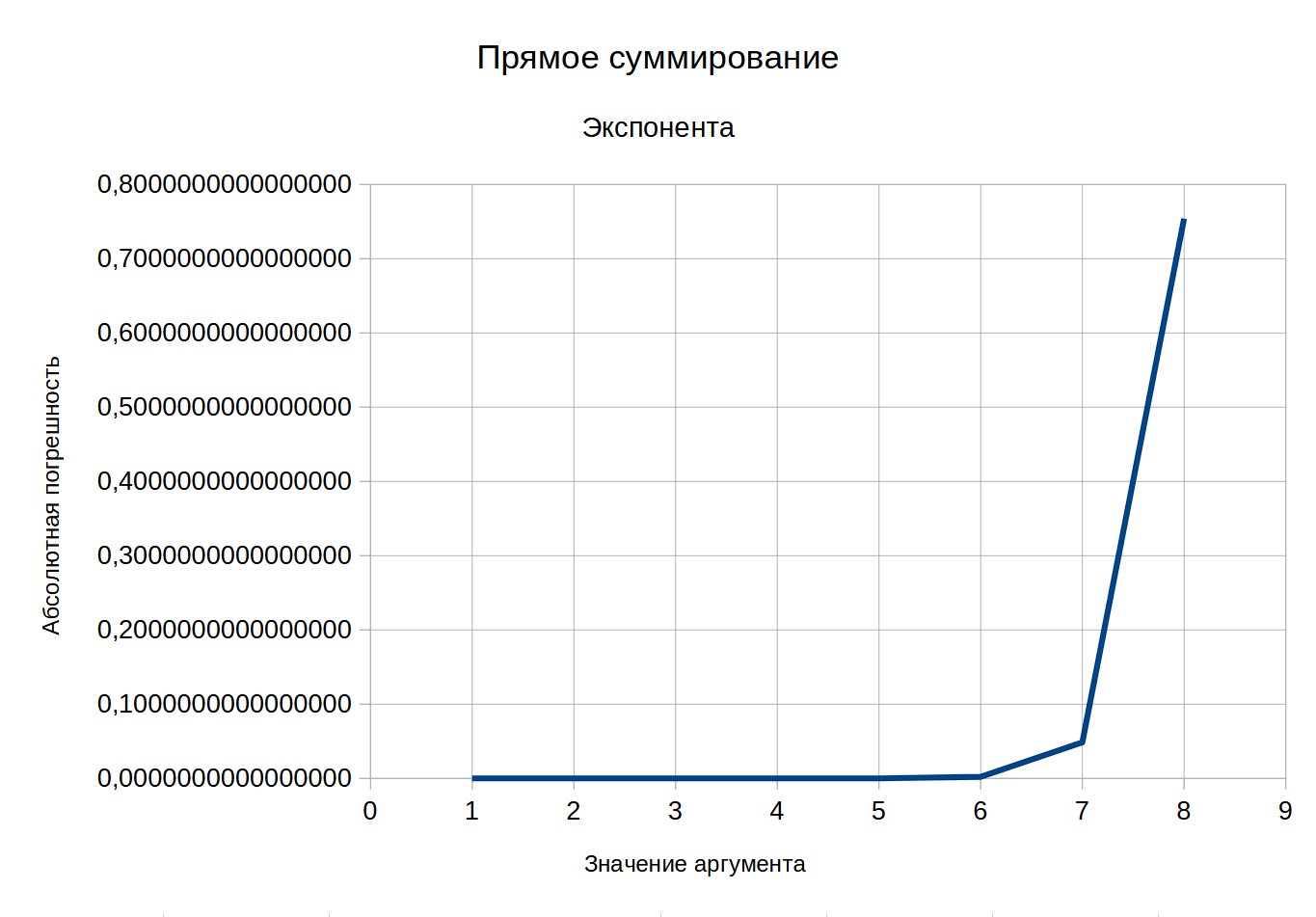


Рис. 18



Рис. 19



Рис. 20

1. Логарифм

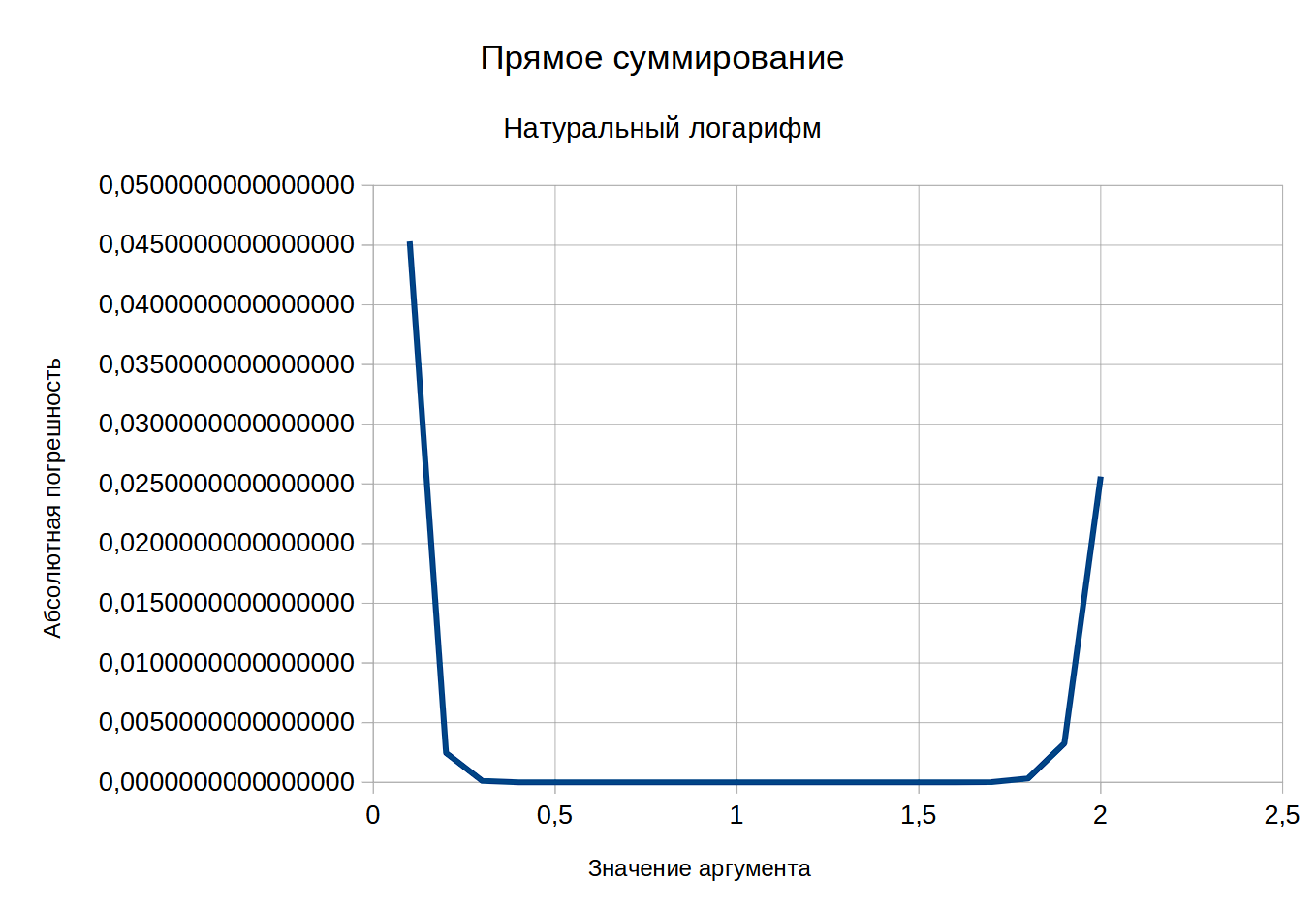


Рис. 21

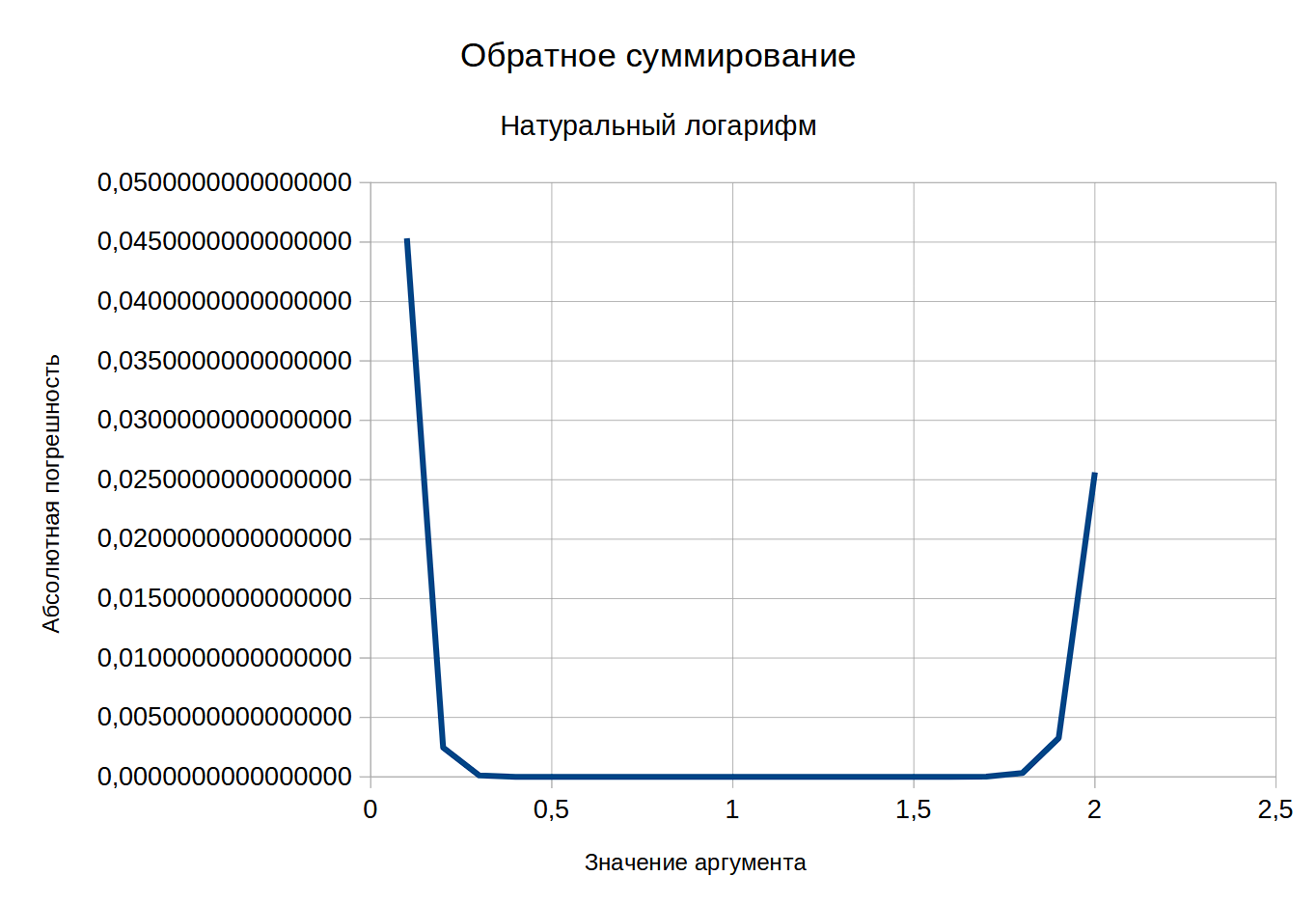


Рис. 22

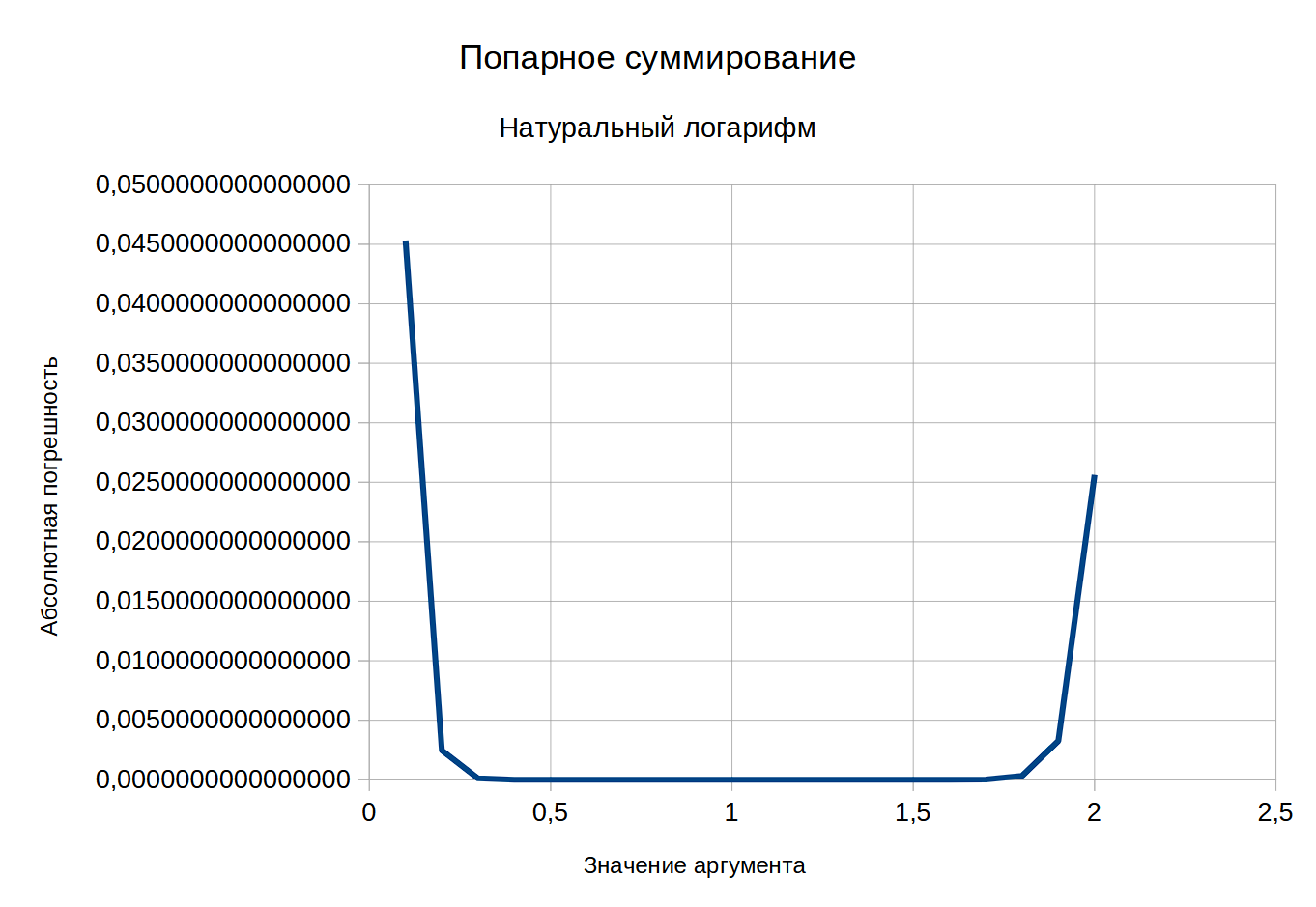


Рис. 23

1. Синус



Рис. 24



Рис. 25



Рис. 26

1. Косинус

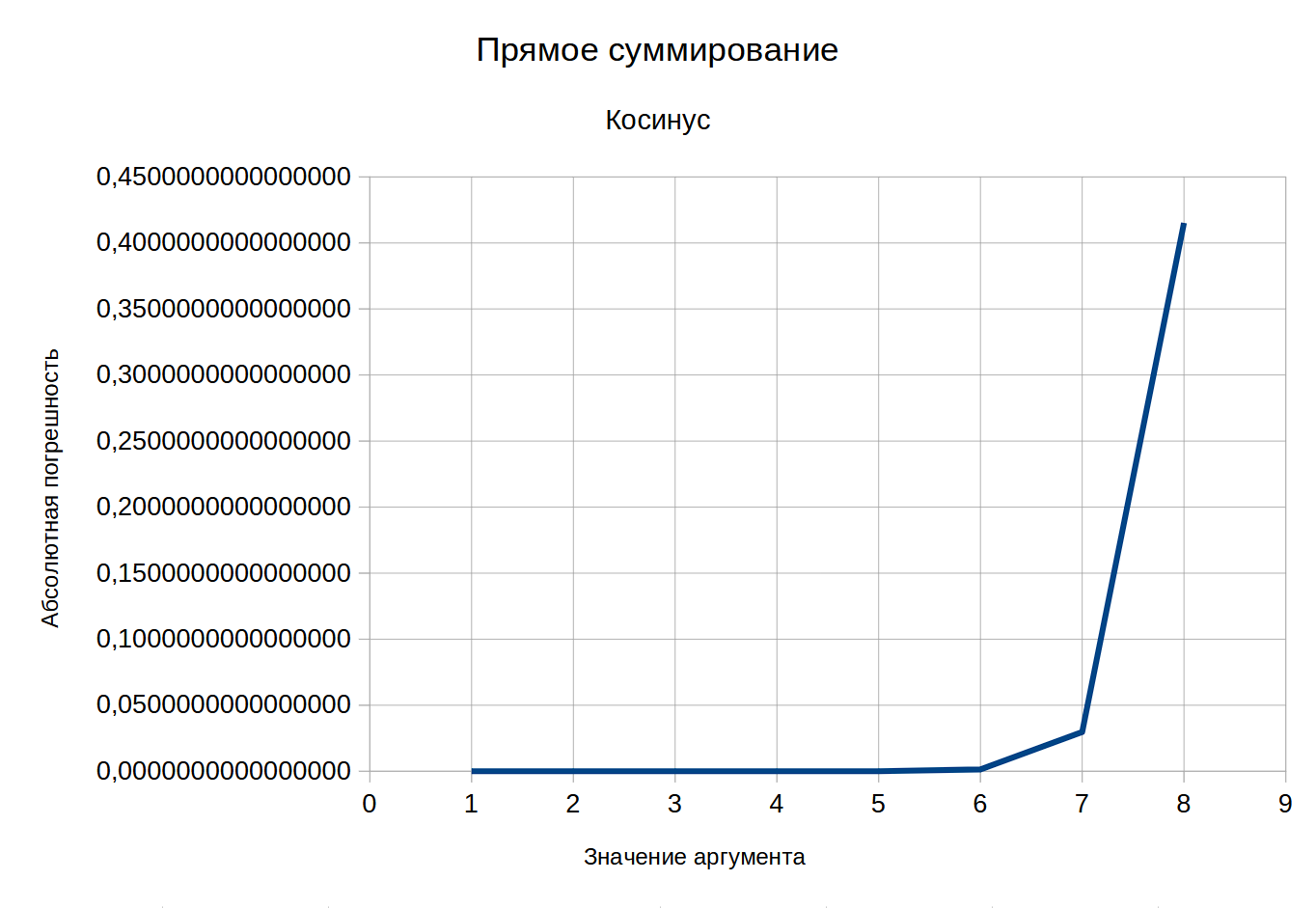


Рис. 27

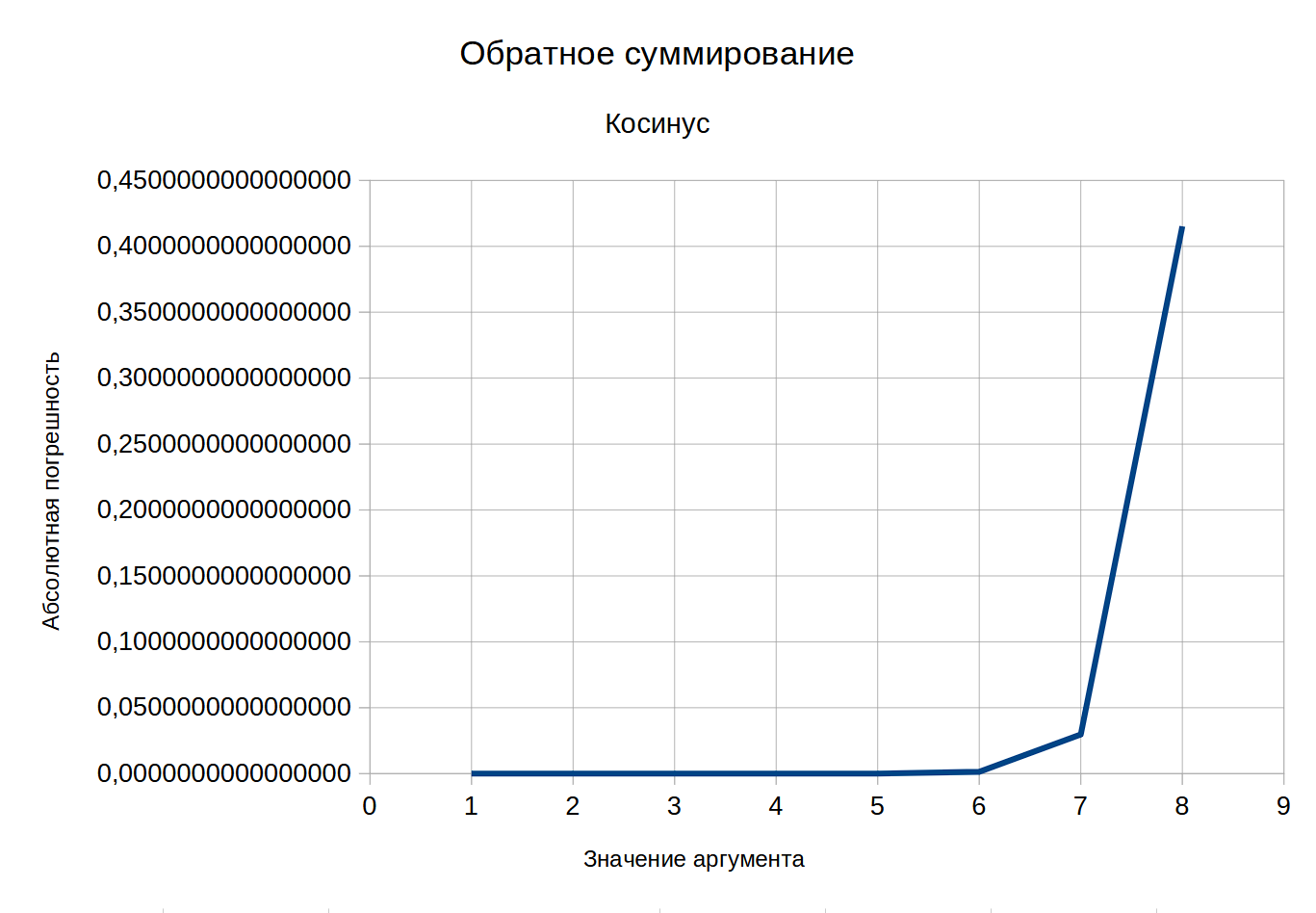


Рис. 28

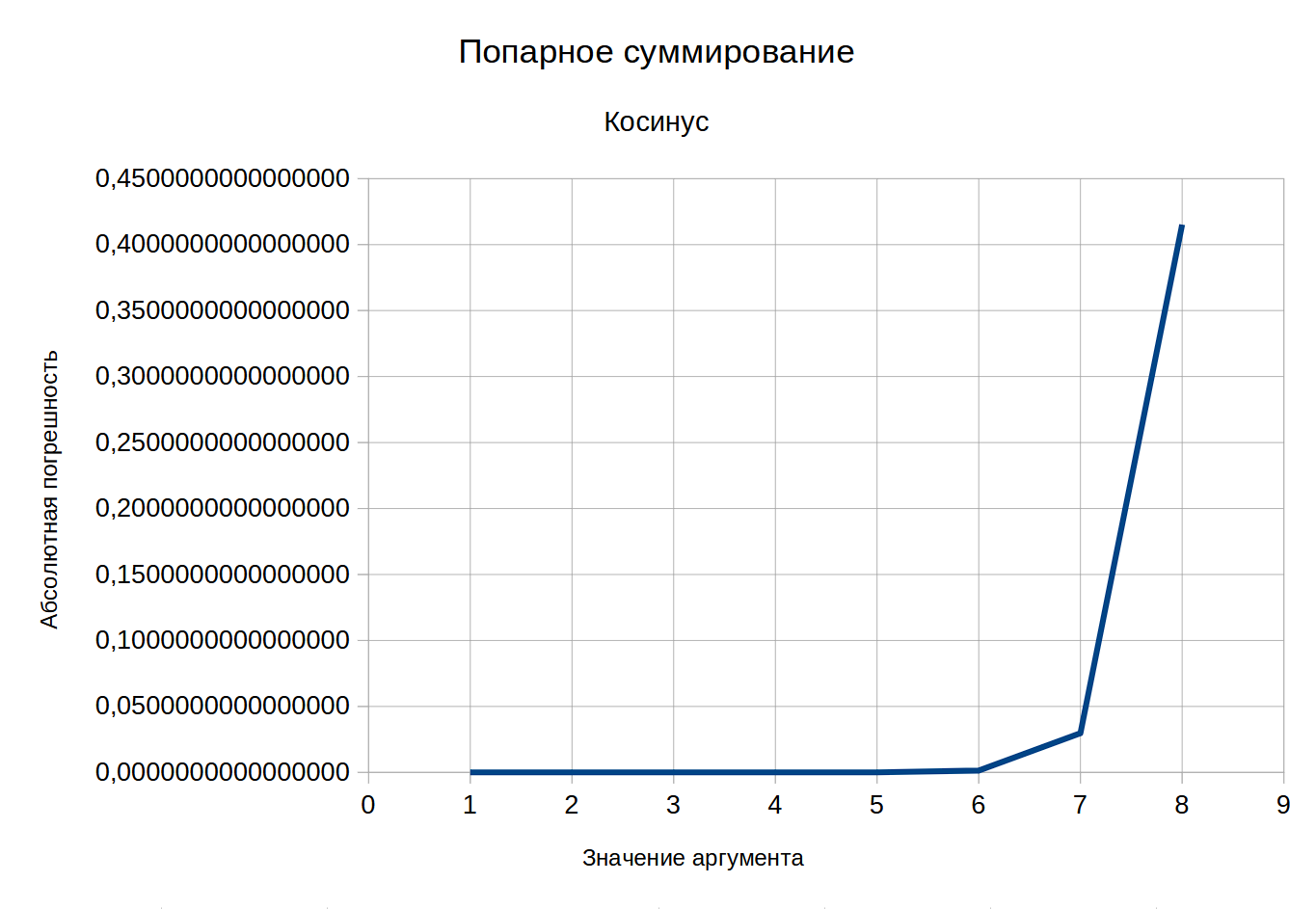


Рис. 29

Как можно заметить для всех функций, кроме логарифма, начиная с некоторого незначительного отклонения от точки разложения появляется погрешность. Сейчас для наглядности покажем, что она пусть и отчасти компенсируется увеличением числа членов, но все же недостаточно. В качестве примера рассмотрим функцию экспоненты. Суммирование будем использовать самое практичное - попарное.

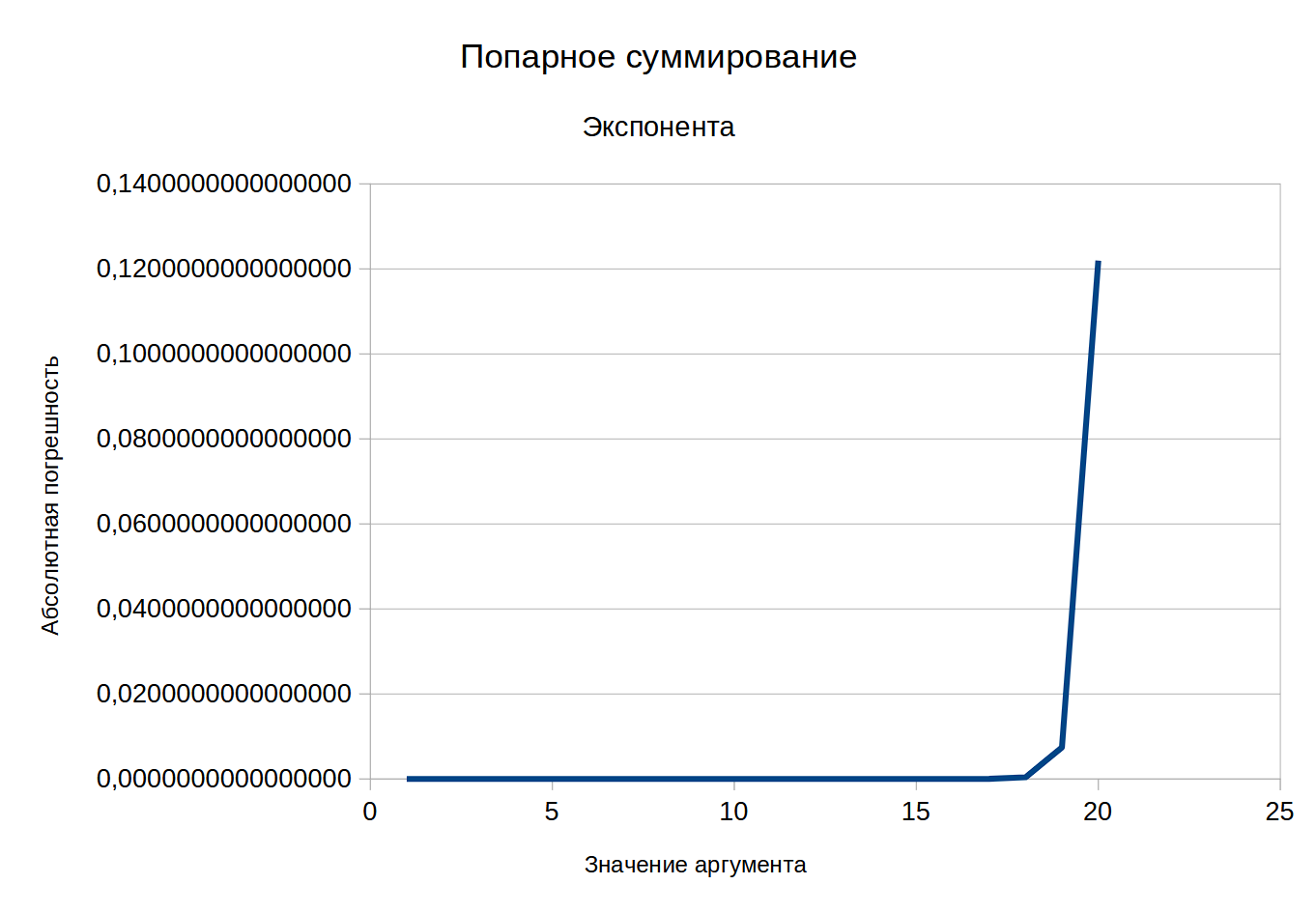


Рис. 30

Здесь использовано 54 монома, в результате получена погрешность в 0,12. Для того, чтобы показать неэффективность способа покажем, что при увеличении аргумента количество мономов также растёт, причём непропорционально быстрее, делая вычисления функции от больших аргументов неудобными.

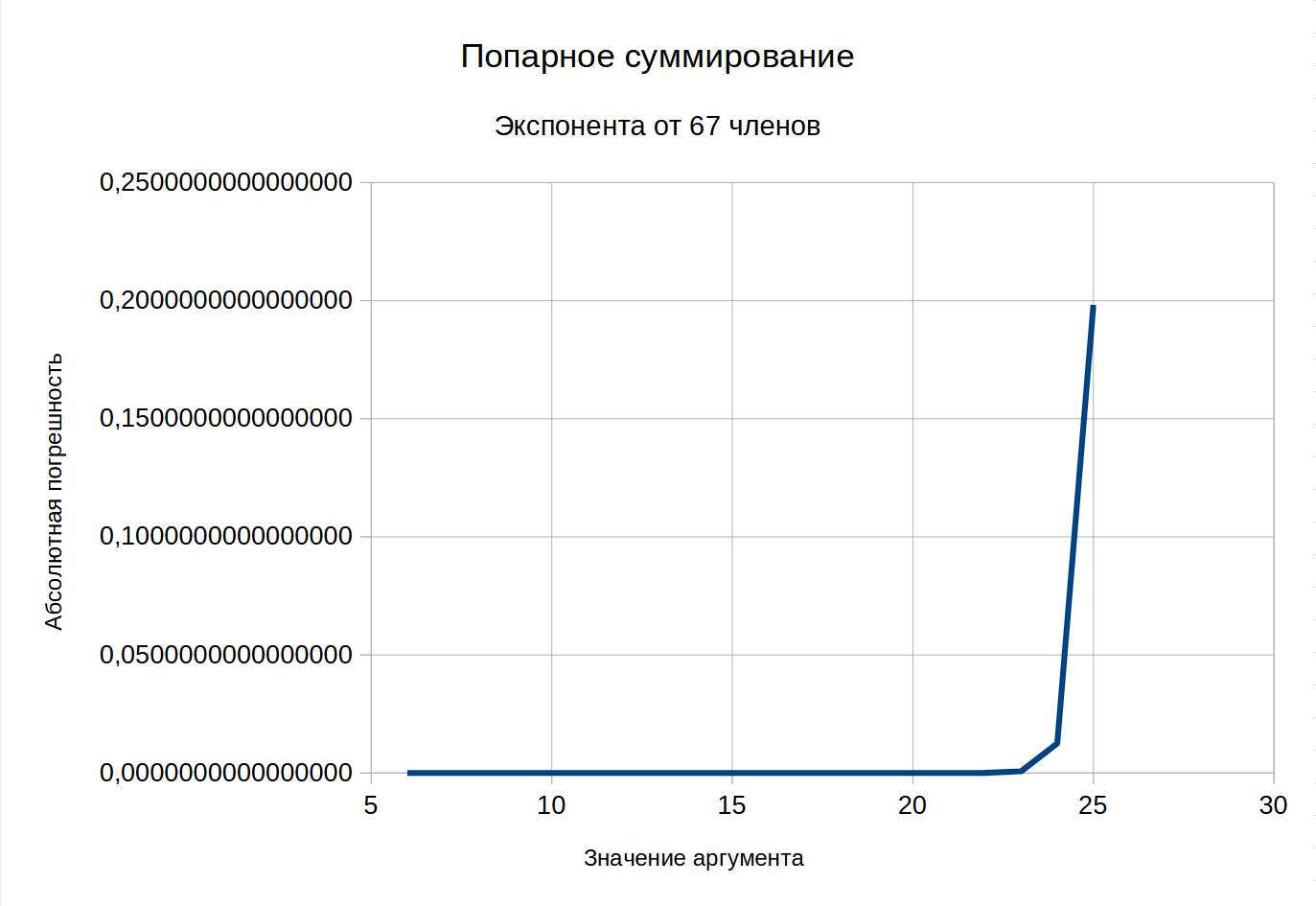


Рис. 31

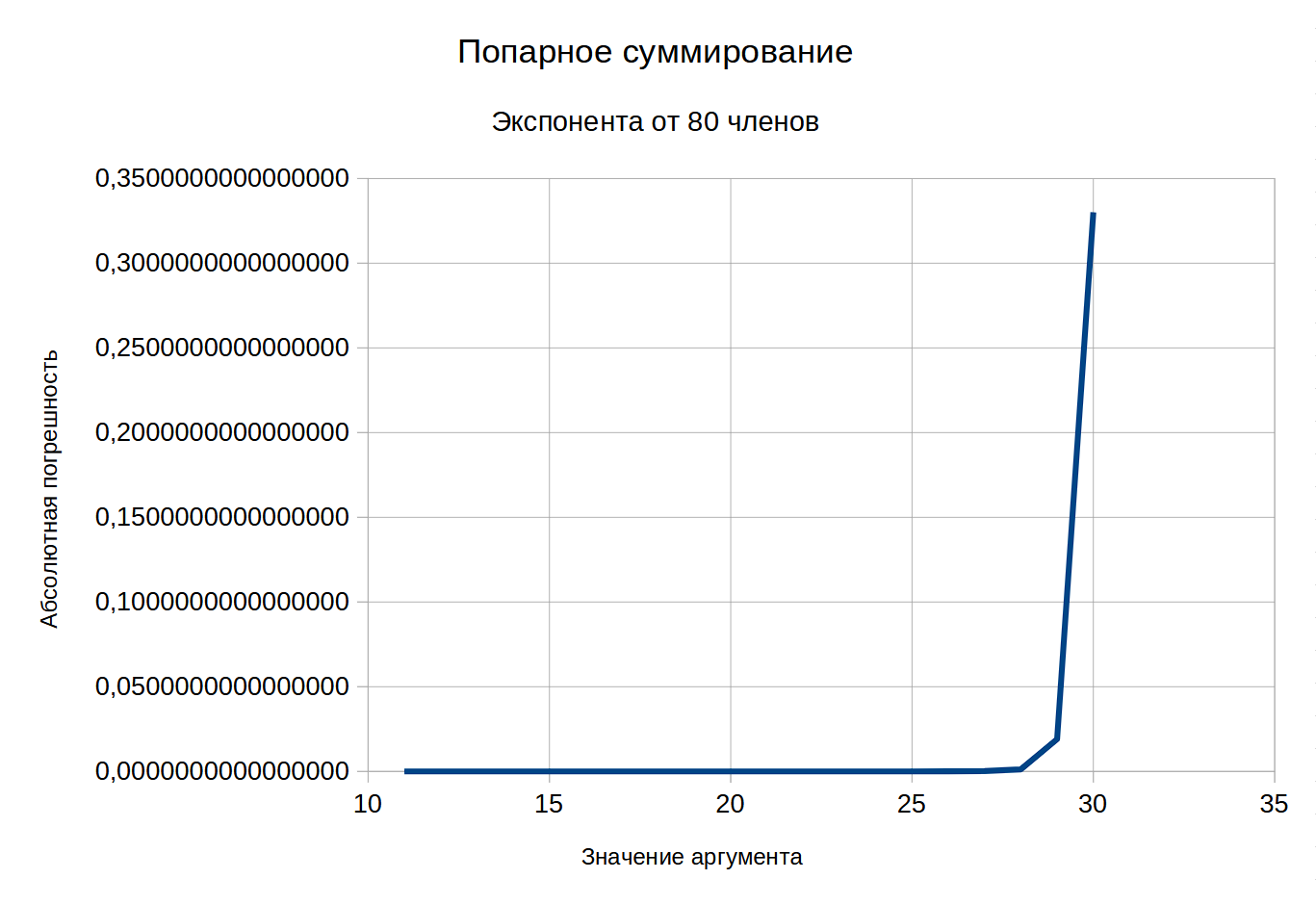


Рис. 32

При соответствующем увеличении аргумента на 5 и в первом и во втором случае потребовалось использовать увеличение количества членов в разложении на 13. С первого взгляда разницы нет, но если посмотреть на значение погрешности, в первом случае она 0,12, во втором 0,19 и в третьем уже 0,3, то есть постепенно увеличивается. Подобная тенденция подтверждает утверждение о том, что начиная с некоторого достаточно большого значения аргумента использовать разложение по Тейлору будет нецелесообразно.

# Заключение

Реализовали вычисление значений функций , , , с помощью рядов Тейлора и сравнили различные способы подсчёта. Изучили зависимость значения абсолютной погрешности от количества членов в ряду и значения аргумента. Пришли к выводу, что подобный подход к вычислению функций применим для натурального логарифма. Для остальных функций он применим только до определённого предела, так как начиная с некоторого момента будет неудобно увеличивать длину полинома, что приведёт к большой погрешности.

# Приложение

Функция вычисления последующего члена у экспоненты:

double exponent(double element, double previousElement, double prePreviousElement, int counter, double argument) {

element = previousElement \* argument / counter;

return element;

}

Функция вычисления последующего члена у логарифма:

double logarithm(double element, double previousElement, double prePreviousElement, int counter, double argument) {

element = previousElement \* (counter - 1) \* (argument - 1) \* (-1) / counter;

return element;

}

Функция вычисления последующего члена у синуса:

double theSine(double element, double previousElement, double prePreviousElement, int counter, double argument) {

element = prePreviousElement \* argument \* argument / (counter \* (counter - 1)) \* (-1);

return element;

}

Функция вычисления последующего члена у косинуса:

double theCosine(double element, double previousElement, double prePreviousElement, int counter, double argument) {

element = prePreviousElement \* argument \* argument / (counter \* (counter - 1)) \* (-1);

return element;

}

Функция прямого суммирования:

double straight(double (\*next)(double, double, double, int, double), double\* polynome, int length, double argument) {

double result = polynome[0] + polynome[1];

for (int counter = 2; counter < length; counter++) {

polynome[counter] = next(polynome[counter], polynome[counter - 1], polynome[counter - 2], counter, argument);

result += polynome[counter];

}

return result;

}

Функция обратного суммирования:

double reverse(double\* polynome, int length) {

double result = 0;

for (int counter = length - 1; counter >= 0; counter--) {

result += polynome[counter];

}

return result;

}

Функция попарного суммирования:

double couple(double\* polynome, int length) {

double result;

double\* innerPolynome;

int step = 1, memoredCounter;

innerPolynome = (double\*)malloc(length \* sizeof(double));

for (int counter = 0; counter < length; counter++) {

innerPolynome[counter] = polynome[counter];

}

while (step \* 2 <= length) {

step \*= 2;

memoredCounter = 0;

for (int counter = step - 1; counter < length; counter += step) {

innerPolynome[counter] += innerPolynome[counter - step / 2];

innerPolynome[counter - step / 2] = 0;

if (counter + step >= length)

memoredCounter = counter;

}

if (length - memoredCounter - 1 >= step / 2) {

innerPolynome[memoredCounter] += innerPolynome[memoredCounter + step / 2];

innerPolynome[memoredCounter + step / 2] = 0;

}

}

result = innerPolynome[step - 1];

return result;

}