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**

**«Функции суммирования»**

**Выполнил**:

студент группы 3821Б1ПМ2

Соколов В.Д.

**Проверил**:

преподаватель каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2022

Оглавление

[**Функции суммирования** 3](#_Toc99134930)

**1.Разложение по ряду Тейлора**…………………………………………3

**2. Применение формулы Тейлора**……………………………….3

**3. Почему элементы `**больших значений порядка**` стремятся к нулю?**..........4

[Постановка задачи 6](#_Toc99134931)

[Методы решения 7](#_Toc99134932)

**1.Прямое суммирование**…………………………………………...7

**2. Обратное суммирование**………………………………………..8

**3. Попарное обратное суммирование**……………………………9

[Руководство пользователя ! ! ! 10](#_Toc99134933)

[Описание программной реализации 14](#_Toc99134934)

[Подтверждение корректности 16](#_Toc99134935)

[Результаты экспериментов 17](#_Toc99134936)

[Заключение 18](#_Toc99134937)

## Функции суммирования

1.**Разложение в ряд Тейлора**

**Брук Те́йлор (1685 – 1731)** *– английский математик. Родился в [Эдмонтоне](https://ru.wikipedia.org/w/index.php?title=%D0%AD%D0%B4%D0%BC%D0%BE%D0%BD%D1%82%D0%BE%D0%BD,_%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D1%8F&action=edit&redlink=1" \o "Эдмонтон, Англия (страница отсутствует)).*

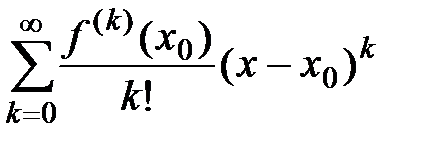
*Создатель формулы Тейлора.*

**Ряд Тейлора** – разложение функции в бесконечную сумму степенных функций.

Ряд Тейлора был известен задолго до публикаций [***Тейлора***](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%B9%D0%BB%D0%BE%D1%80,_%D0%91%D1%80%D1%83%D0%BA) — его использовали ещё в XIV веке в [Индии](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B4%D0%B8%D1%8F), а также в XVII веке [***Грегори***](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B5%D0%B3%D0%BE%D1%80%D0%B8,_%D0%94%D0%B6%D0%B5%D0%B9%D0%BC%D1%81) и [***Ньютон***](https://ru.wikipedia.org/wiki/%D0%9D%D1%8C%D1%8E%D1%82%D0%BE%D0%BD,_%D0%98%D1%81%D0%B0%D0%B0%D0%BA). Ряды Тейлора применяются при [аппроксимации](https://ru.wikipedia.org/wiki/%D0%90%D0%BF%D0%BF%D1%80%D0%BE%D0%BA%D1%81%D0%B8%D0%BC%D0%B0%D1%86%D0%B8%D1%8F) функции [многочленами](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D1%87%D0%BB%D0%B5%D0%BD). В частности, [линеаризация](https://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D0%BD%D0%B5%D0%B0%D1%80%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F) уравнений происходит путём [разложения в ряд Тейлора и отсечения всех членов выше первого порядка](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BE%D1%80%D0%B5%D0%BC%D0%B0_%D0%A2%D0%B5%D0%B9%D0%BB%D0%BE%D1%80%D0%B0).

Обобщением понятия *ряда Тейлора* в функциональном анализе является [***ряд Фантапье***](https://ru.wikipedia.org/wiki/%D0%A0%D1%8F%D0%B4_%D0%A4%D0%B0%D0%BD%D1%82%D0%B0%D0%BF%D1%8C%D0%B5).

Формула Тейлора:



2.**Применение формулы Тейлора**

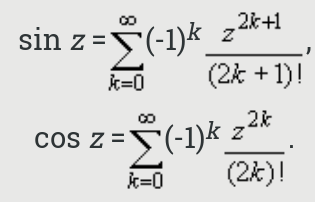
Благодаря формуле Тейлора мы можем представить синус, косинус, экспоненту и натуральный логарифм [ln(1+x)] как бесконечную сумму степенных функций:

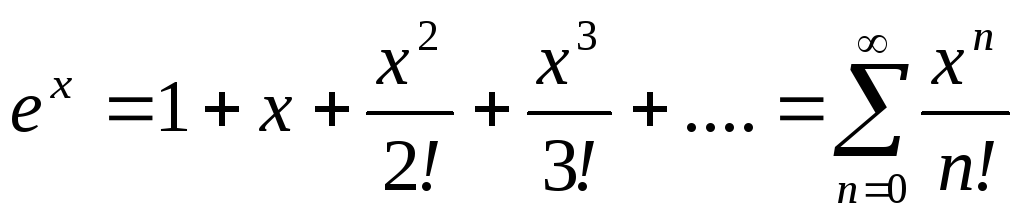
Sin(x) = x - x^3/3! + x^5/5! - ... +(-1)^m \* x^(2m+1)/(2m+1)!

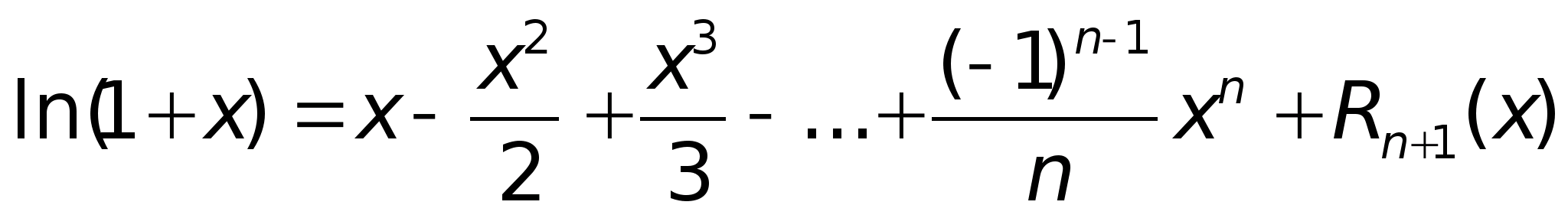
Cos(x) = 1 - x^2/2! + x^4/4! - ... + (-1)^n \* x^2n/(2n)!

Exp(x) = 1 + x + x^2/2! + ... + x^k/k! k - любая

ln(1+x) = x - x^2/2! + x^3/3! - ... + (-1)^(n-1) \* x^n/n!







3.**Почему элементы `больших значений порядка`(пример: el84231) стремятся к нулю?**

В начале может показаться, что если мы возьмём достаточно большой X, то следующие элементы будут лишь расти в значениях.

Пример:

Sin(x)= x – x^3/3! + x^5/5! - …

x = 1000;

Sin(x)= 1000 – 1000000000/6 + 1000000000000000/120 -

1000 < (10^9)/6 < (10^15)/120

Так почему же элементы `больших значений порядков` стремятся к нулю?

Sin(x)= x – x^3/3! + x^5/5! - … + ***x^n/n!***

X=1000;

X^n/n! n – большое значение

0 < (1000^n)/(n!) = 1000/1 \* 1000/2 \* 1000/3 \* … \* 1000/1000 \* 1000/1001 \* … \* 1000/(n-1) \* 1000/n

( > 1000^1000 ) … (=1) (~=1) … (~=0) (~=0)

1000/(n-1001) ~=0 \*1000 ~=0

1000/(n-1000) ~=0 \*1000 ~=0

… \* ( 0^1000 , такое, что 1000^1000 \* 0^1000 < 1 (~=0) )

1000/(n-2) ~=0 \*1000 ~=0

1000/(n-1) ~=0 \*1000 ~=0

Сохраняем 1000/n ~=0

0 < 1000/1 \* 1000/2 \* … \* 1000/1000 \* 1000/1001 \* … \* 1000/(n-1001) \* 1000/(n-1000) \* … \* 1000/(n-1) \* 1000/n < 1000/n

( 1000^1000) \* ( 1 ) \* ( 0^1000 ) \* 1000/n

0 < (1000^n)/(n!) < 1000^1000 \* 1 \* 0^1000 \* 1000/n < 1 \* 1 \* 1000/n = 1000/n

0 < (1000^n)/(n!) < 1000/n ~=0

Таким образом мы видим, что элемент eln = (1000^n)/(n!) *стремится к нулю* при достаточно больших значениях **n**.

В программе можно избежать таких сложных значений в синусе и косинусе, так как там есть период у числа **x**, а в логарифме |x| < 1.

# **Постановка задачи**

Целью лабораторной работы являлась реализовать на языке программирования Си следующие методы суммирования:

1.Прямое суммирование

2.Обратное суммирование

3.Обратное попарное суммирование

Также необходимо было создать четыре функции, используя данные методы:

1. Sin(x)
2. Cos(x)
3. Exp(x)
4. Ln(1+x)

float

Сортировки нужно реализовать для данных типа ~~double~~ . Нужно описать программную реализацию и алгоритмы работы данных методов суммирования. Необходимо подтвердить корректность реализации данных сортировок. Провести эксперименты для сравнения точностей, проведения экспериментов и сделать вывод по полученным результатам.

# **Методы решения**

**1.Прямое суммирование**

Прямое суммирование представляет собой самый простой и обычный метод. Из-за этого, конечно, точность этого метода немного ниже остальных.

Суть алгоритма проста, мы берём первый элемент (из формулы Тейлора) и складываем его со следующим элементом, а затем ещё со следующим и т.д. Этот процесс будет идти до таких пор, пока следующий элемент не будет таким низким, что программа посчитает его равным нулю.

В итоге получится что-то типа: F=el1+el2+el3+el4+el5+0 (+0+0+…+0)

Пример работы:

1. F1= 1 + 2 + 3 + 4 + 5 + 3 + 2 + 0

F1= 3 + 3 + 4 + 5 + 3 + 2 + 0

F1= 6 + 4 + 5 + 3 + 2 + 0

F1= 10 + 5 + 3 + 2 + 0

F1= 15 + 3 + 2 + 0

F1= 18 + 2 + 0

F1= 20 + 0

Ответ: F1= 20

1. Sin(2) = 2 – 2^3/3! + 2^5/5! – 2^7/7!

Упростим:

Sin(2) = 2 – 4/3 + 4/15 – 8/315

Sin(2) = 2/3 + 4/15 – 8/315

Sin(2) = 14/15 – 8/315

Sin(2) ~= 0,9

Ответ: Sin(2) ~= 0,9

Если посмотрим на второй пример, то заметим большой минус метода.

Когда мы считаем сумму, то с каждым последующим элементом она становится больше и когда мы будем приближаться к элементам, стремящимся к нулю, то они не смогут сильно повлиять на сумму.

2.**Обратное суммирование**

Обратное суммирование, как и следует из названия, полная противоположность прямого суммирования. Обратное суммирование точнее прямого.

Суть алгоритма состоит в “обратном” алгоритме прямого суммирования, т.е. если во время прямого суммирования мы начинали с первого элемента и складывали дальше по порядку, то теперь мы начинаем с последнего элемента (*последний элемент мы задаём сами)* и складываем остальные элементы в порядке убывания.

В итоге получится что-то типа: F=el1+el2+el3+el4+el5+*el6*+*el7*+(0+…+0)

Пример работы:

1. F1= 2+3+1+1/2+1/3+1/4

F1= 2+3+1+1/2+7/12

F1= 2+3+1+13/12

F1= 2+3+25/12

F1= 2+61/12

F1= 85/12

***Ответ:*** F1=7,08

1. Сos(2) = 1 – 2^2/2! + 2^4/4! – 2^6/6! +2^8/8! – 2^10/10!

Упростим:

Cos(2) = 1 – 2 + 2/3 – 4/45 + 2/315 – 4/14175

Cos(2) = 1 – 2 + 2/3 – 4/45 + 86/14175

Cos(2) = 1 – 2 + 2/3 – 1174/14175

Cos(2) = 1 – 2 + 8276/14175

Cos(2) = 1 – 20074/14175

Cos(2) = - 5899/14175

Ответ: Cos(2) ~= -0,416155 Настоящий: Cos(2) ~= -0,416146

Заметим, что даже когда мы взяли всего ***6*** элементов точность вычислений колоссальная. Мы сначала складываем маленькие значения, разница между которыми не такая большая как при прямом суммировании и поэтому точность выше.

3.**Попарное обратное суммирование**

В общих случаях, самый точный метод вычисления суммы.

Алгоритм попарного суммирования представляет собой такую сумму, у которой слагаемыми являются суммы двух ближайших элементов. Попарное обратное суммирование действует, как и в обратном суммировании. Мы начинаем с последнего **слагаемого** (*последний элемент мы задаём сами)* и складываем остальные **слагаемые** в порядке убывания.

В итоге получится что-то типа: F=(el1+el2)+(el3+el4)+(el5+*el6)*+0

Пример работы:

1. F1= 2+3+5+3+1+1/2

F1= (2+3)+(5+3)+(1+1/2)

F1= 5+8+3/2

F1= 5+19/2

F1= 29/2

Ответ: F1= 58,5

1. Sin(2)= 2 – 2^3/3! + 2^5/5! – 2^7/7! + 2^9/9! - 2^11/11!

Упростим:

Sin(2)= 2 – 4/3 + 4/15 – 8/315 + 4/2835 – 8/155925

Sin(2)= (2 – 4/3) + (4/15 – 8/315) + (4/2835 – 8/155925)

Sin(2)= (2/3) + (76/315) + (212/155925)

Sin(2)= (2/3) + (37832/155925)

Sin(2)=141782/155925

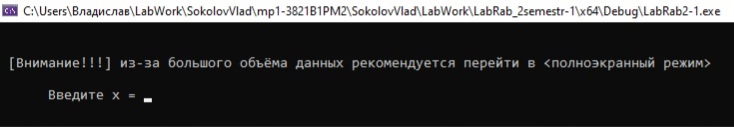
Ответ: Sin(2)= 0,909296 настоящий sin(2)= 0.909297

Заметим, что метод попарного суммирования, по сути, работает не циклично, т.е. когда мы вычислим попарную сумму между элементами (**слагаемое**), то больше мы так делать не будем. Дело здесь не в том, что алгоритм слишком сложный или появляется погрешность, нет. На самом деле, нам это просто не нужно, и вот почему:

Sin(2)= 2 – 2^3/3! + 2^5/5! – 2^7/7! + 2^9/9! - 2^11/11!

У синуса чередуются знаки (+ - + - …) и поэтому логично предположить, что |(+) + (-)| (сумма между положительным и отрицательным элементами) намного меньше, чем max((+) или |(-)|). Из этого можем сделать вывод (+ обратный метод), что точность вычислений возросла(хоть и немного).

# **Руководство пользователя ! ! !**



Сначала, для удобства, пользователю будет предложено перейти в полноэкранный режим.

К сожалению, в данной программе отсутствует удобное меню (

!!! Важная информация !!!

Из-за перехода из double в float был сломан массив!

Теперь, вводя x, Пользователю настоятельно не рекомендуется ставить значение больше 997 после запятой (0,997 ; 3412,909997 = можно)

Если у пользователя получилась экспонента похожая на эту : 13864373474820096,000000

В то время, как настоящая выглядит так: inf

То это нормально. Происходит это из-за того, что x=inf , а массив зависящий от x ещё не закончен.

Поэтому, когда x становится больше 9999999999999999.0

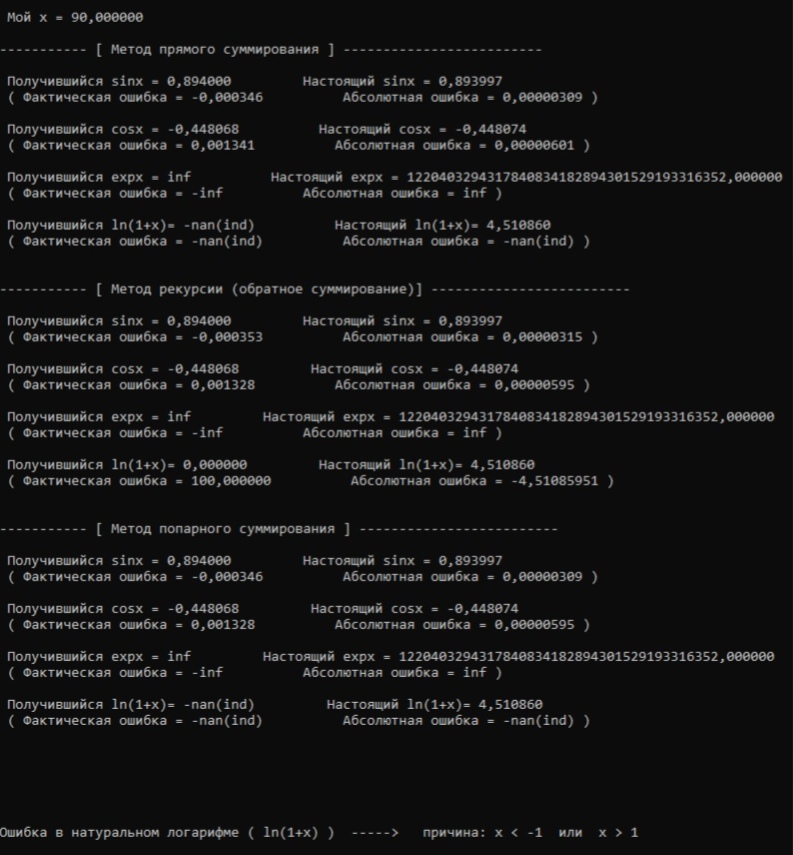
то вычисление экспоненты насильно прерывается, чтобы не сломать массив.

При вводе аномально большого значения x Пользователю рекомендуется смотреть только на получившиеся синус и косинус

Прочитав важную информацию пользователь может приступить к Заданию числа x

Внимание: если пользователем будут найдены ещё какие-либо ошибки необходимо обратится к студенту, создавшему эту программу. Почта: vladsokolov2001@gmail.com

Телефон: +7-930-704-83-50



После задания x пользователю сразу предоставляется результат.

На экран выводится понятная информация (я надеюсь).

Довольно подробно всё описано на русском языке. Дизайн блоков был сделан, чтобы с лёгкостью отделить методы суммирования друг от друга.

Пользователь с лёгкостью сможет найти интересующую его информацию.

P.S. Снизу выводятся математические ошибки.

Получившийся f(x) = [число] | показывает f(x), посчитанную через данную программу

Настоящий f(x) = [число] | показывает f(x), посчитанную функцией из math.h

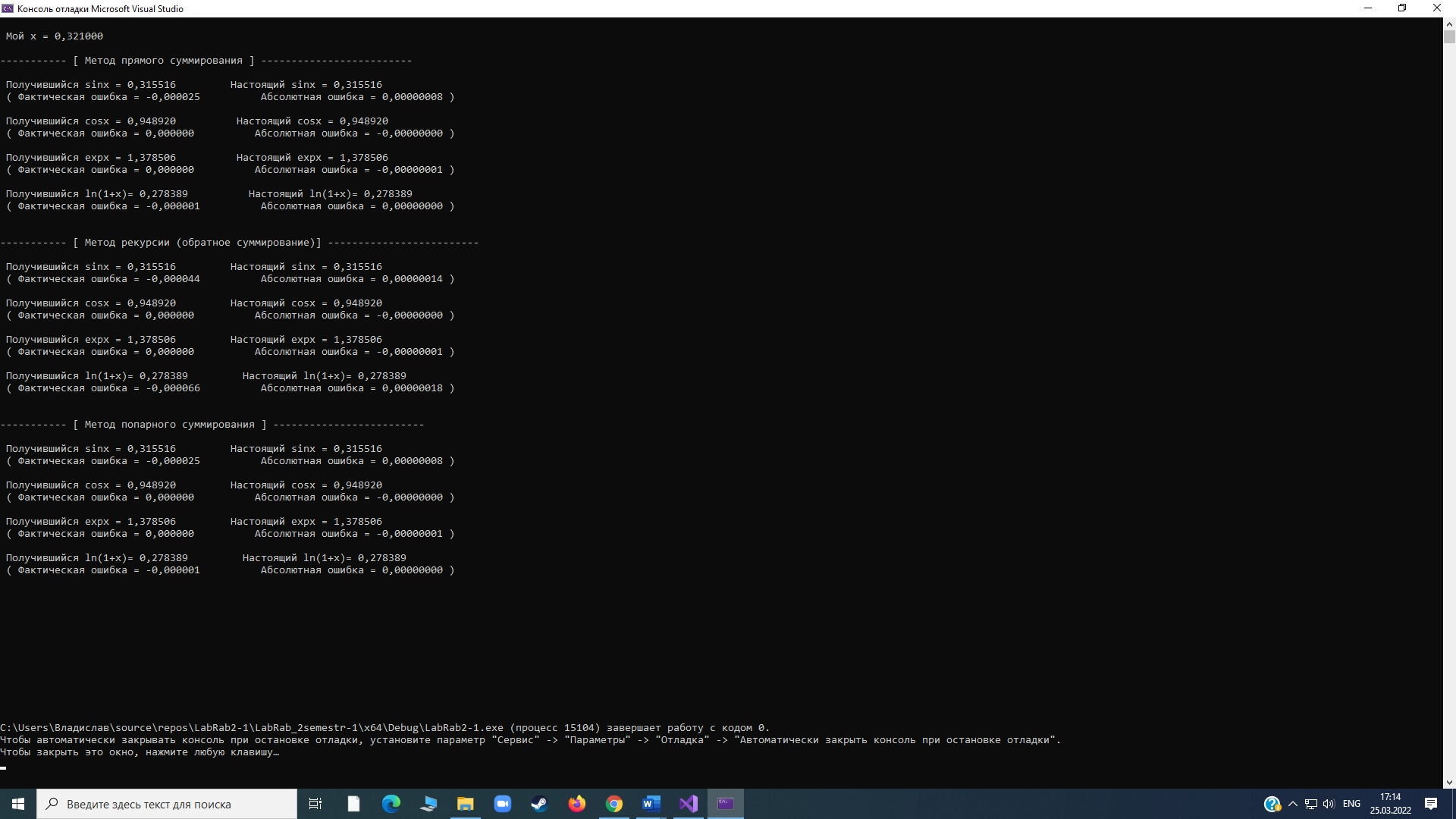
Фактическая ошибка f(x) | показывает разницу между “мой” f(x) и “настоящей” f(x) в процентах

Абсолютная ошибка f(x) | показывает разницу в значениях между “мой” f(x) и “настоящей” f(x)

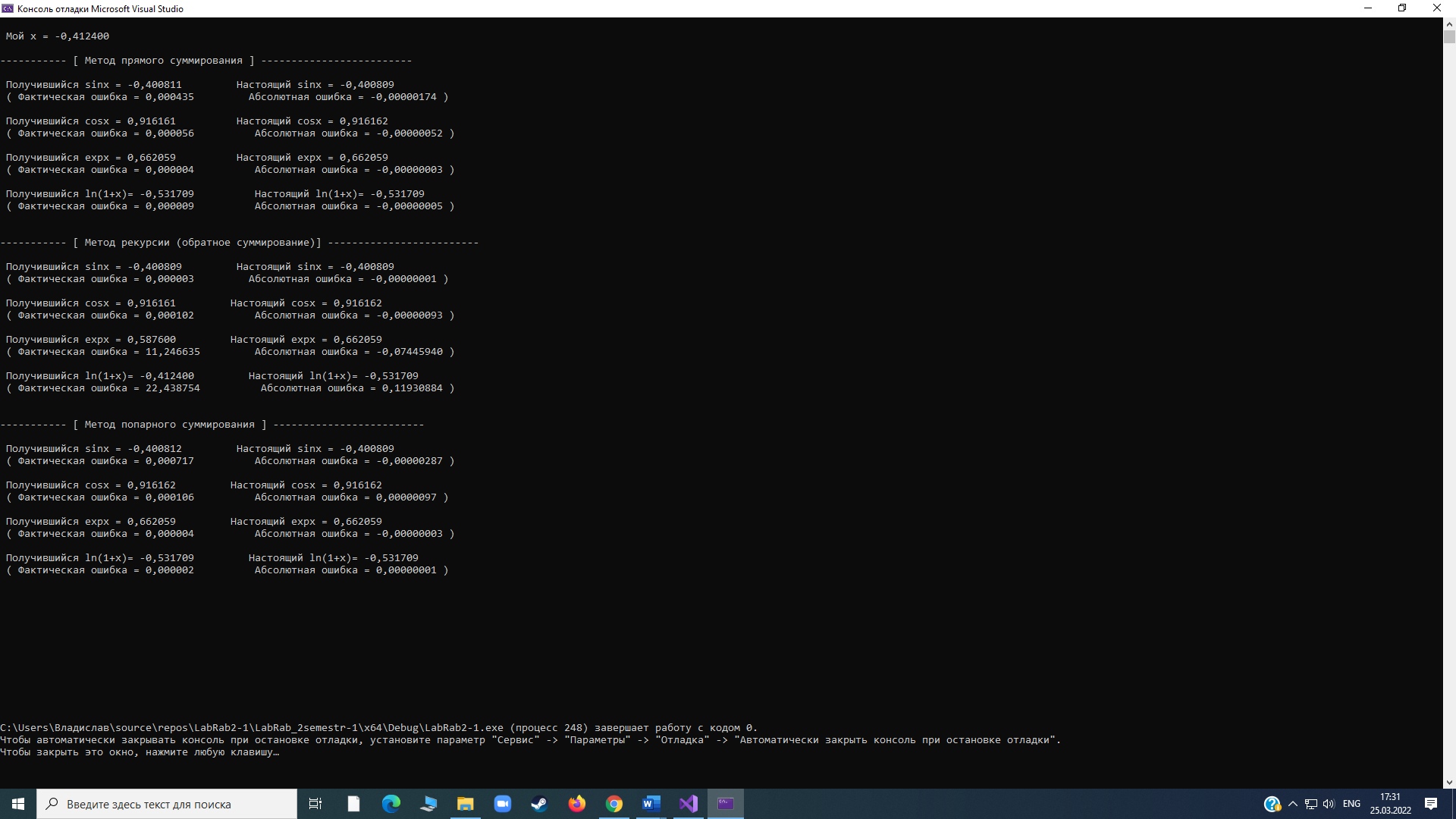
Три раздела -> три метода и каждый метод имеет свой уникальный способ вычисления f(x) , следовательно пользователю представляется наглядно оценить точность методов.

(по-честному: разница между методами смешная, но есть. В теории – красиво)

Пример работы при x < 1



Пример работы при x < 0



# **Описание программной реализации**

int main() – это основная функция программы. Программа всегда начинается с main и всегда ею заканчивается (определение main). Некоторые функции непосредственно в main отсутствуют, ведь они находятся в Header файле.

Операции в main :

1. Вычитание периода (для sin(x) и cos(x) ) и задание пользователем X
2. Прямое суммирование:
3. float next\_element\_sin(float prevElement, float x, int i) –

принимает на вход <первый элемент функции> (x), < x > (с вычтенным периодом) и < порядок элемента > (1). Функция находит следующий элемент суммы синуса и возвращает его значение.

1. float next\_element\_cos(float prevElement, float x, int i) –

принимает на вход <первый элемент функции> (1), < x > (с вычтенным периодом) и < порядок элемента > (1). Функция находит следующий элемент суммы косинуса и возвращает его значение.

1. float next\_element\_exp(float prevElement, float x, int i) –

принимает на вход <первый элемент функции> (1), < x > (оригинальный) и < порядок элемента > (1). Функция находит следующий элемент суммы экспоненты и возвращает его значение.

1. float next\_element\_cos(float prevElement, float x, int i) –

принимает на вход <первый элемент функции> (x), < x > (оригинальный) и < порядок элемента > (1). Функция находит следующий элемент суммы логарифма и возвращает его значение.

Вычисляем сумму и сохраняем результат.

1. Обратное суммирование:
2. float recursive\_sin(float x, float xconst, int i) –

принимает на вход <первый элемент функции> (x), < x > (оригинальный) и < порядок элемента > (0). Функция находит предыдущий элемент суммы синуса, складывает его и возвращает его значение. В зависимости от порядка элементов, значение будет менять знак.

1. float recursive\_cos(float x, float xconst, int i) –

принимает на вход <первый элемент функции> (1), < x > (оригинальный) и < порядок элемента > (0). Функция находит предыдущий элемент суммы косинуса, складывает его и возвращает его значение. В зависимости от порядка элементов, значение будет менять знак.

1. float recursive\_exp(float x, float xconst, int i) –

принимает на вход <первый элемент функции> (1), < x > (оригинальный) и < порядок элемента > (0). Функция находит предыдущий элемент суммы косинуса, складывает его и возвращает его значение.

1. float recursive\_ln(float x, float xconst, int i) –

принимает на вход <первый элемент функции> (x), < x > (оригинальный) и < порядок элемента > (1). Функция находит предыдущий элемент суммы косинуса, складывает его и возвращает его значение.

1. Обратное попарное суммирование

Заводим массивы для всех нужных нам функций: sin(x),cos(x),exp(x),ln(x)

И кладём в нулевой порядок их первые элементы.

Array\_for\_sin[0]= PredElSin; //первый элемент sin

Array\_for\_cos[0]= PredElCos; //первый элемент cos

Array\_for\_exp[0]= PredElExp; //первый элемент exp

Array\_for\_ln[0]= PredElLn; //первый элемент ln

Используем способ нахождения следующих элементов, описанный в прямом суммировании (2 пункт), и подставляем в массивы.

~~Здесь, по плану, моя прога ломается~~

Добавляем в главную сумму “сумму двух элементов” (начиная с конца) Все элементы нам известны и поэтому просто складываем их попарно до того как добавить в главную сумму.

1. Поиск ошибок:
2. Фактическая ошибка – находим отношение между погрешностью и правильным ответом и умножаем на 100 % . Ответ даём в процентах. (скажите мне потом, как отображать проценты в printf )
3. Абсолютная ошибка – это просто разница между правильным ответом и ответом программы. Под правильным ответом, я подразумеваю функцию из math.h
4. Вывод результата на экран

# Подтверждение корректности

Подтверждение корректности проходит очень просто, ведь цель задачи в том и была, чтобы как можно корректно найти нужную сумму и сравнить её с официально предоставленной (math.h) функцией. Поиск ошибок также упростил нам это дело.

X=12

Получившийся sinx = -0,536573 Настоящий sinx = -0,536573

( Фактическая ошибка = 0,000003 % Абсолютная ошибка = -0,00000002 )

Получившийся cosx = 0,843855 Настоящий cosx = 0,843854

( Фактическая ошибка = 0,000112 % Абсолютная ошибка = 0,00000095 )

Получившийся expx = 162754,781250 Настоящий expx = 162754,791419

( Фактическая ошибка = 0,000006 % Абсолютная ошибка = -0,01016900 )

X=0,45 (отдельно для логарифма)

Получившийся ln(1+x)= 0,371564 Настоящий ln(1+x)= 0,371564

( Фактическая ошибка = 0,000002 % Абсолютная ошибка = 0,00000001 )

# Результаты экспериментов

Разность методов суммирования есть, но она крайне мала. Стоит отметить, что если вводить “адекватные” (нельзя сказать по-другому) X то программа будет выдавать до идеала похожую на настоящую сумму результат.

По данным эксперимента выяснилось следующее:

Точность программ (что точнее?):

#1 Обратное попарное суммирование

#2 Обратное суммирование

#3 Прямое суммирование

Сложность программ не оценивалась, так как у нас в приоритете точность, а не скорость.

Были найдены Ошибки – погрешности (несоответствие с функциями из math.h)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | Место по точности | | |
| от | до | №1 | №2 | №3 |
| -40 | 0,01 | **3** | 2 | 1 |
| 0,01 | 21,55 | 1 | 3 | 2 |
| 21,55 | 28,54 | похожи | | |
| 28,54 | 40 | 1 | 3 | 2 |
| 40 | … | чем дальше тем похоже | | |

**1 – прямое суммирование**

**2 – обратное суммирование**

**3 – попарное обратное суммирование**

Прошу прощения, но мой комп очень слабый.

По таблице понятно, что нет **самого** точного метода суммирования. В теории самым точным должен быть метод попарного обратного суммирования, но на практике он в основном является **самым надёжным**, ведь он никогда не занимал последнее место по точности.

# Заключение

В ходе лабораторной работы на языке программирования Си были реализованы метод прямого суммирования, обратного суммирования, попарного обратного суммирования для таких функций как: sin(x), cos(x), exp(x) и ln(1+x). Были описаны алгоритмы работы данных методов, их программная реализация и проведенные эксперименты для замера и подтверждения их теоретический точности. Нахождение ошибок тоже было выполнено. Ошибки сыграли ключевую роль в выполнении лабораторной работы, ведь числа имели слишком много цифр после запятой, и они (ошибки) очень помогали увидеть, что всё-таки где-то что-то не совпадало.

Код был переделан кучу раз. Скорее всего Вы это не читаете, но я всё же напишу основную проблему программы, которую я написал: при переходе на float у меня было 317 (!) ошибок. Все я исправить не мог, поэтому и вставил кучу всяких костылей.

В связи с тем, что мой компьютер не способен обрабатывать большой объём данных и сильно тупит даже в очень простых программах, результаты экспериментов имеют упрощённый вид (чего стоит одна таблица).