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**

**«Ряды Тейлора»**

**Выполнил**:

студент группы 3823Б1ПМ1

Давыдов Д. В.

**Проверил**:

преподаватель каф. ВВСП,

Волокитин В.Д.

Нижний Новгород

2024

**Содержание**

[Постановка задачи 3](#_Toc162909935)

[Метод решения 4](#_Toc162909936)

[Руководство пользователя 7](#_Toc162909937)

[Описание программной реализации 8](#_Toc162909938)

[Подтверждение корректности 9](#_Toc162909939)

[Результаты экспериментов 12](#_Toc162909940)

[Заключение 23](#_Toc162909941)

[Приложение 24](#_Toc162909942)

# Постановка задачи

Требовалось:

1. Реализовать функции суммирования рядов Маклорена тремя различными способами: прямое суммирование, попарное суммирование и обратное суммирование. Каждая из функций принимает нулевой член, аргумент, указатель на функцию для вычисления следующего множителя, на который необходимо домножить текущий член, чтобы получить следующий;
2. С помощью данных функций реализовать функции основных математических функций (синуса, косинуса, экспоненты и натурального логарифма);
3. Написать программу с понятным интерфейсом, позволяющую получить значение функции по заданному аргументу желаемым способом суммирования;
4. Проверить программу на корректность;
5. Исходный код программ загрузить в указанный репозиторий;
6. Провести ряд экспериментов, в ходе которых выяснить, какой из способов суммирования наиболее точно вычисляет значение заданной функции, сравнивая полученные значения с значениями тех же функций, реализованных в библиотеке cmath.

# Метод решения

Требовалось найти формулу для следующего множителя для данных математических функций. Ряды Маклорена для них:

Следующий множитель требовалось найти в виде:

, где – i-ое слагаемое.

Для решения поставленной задачи, необходимо реализовать все 3 вида суммирования:

1. **Прямое суммирование:** суммирование слагаемых в прямом порядке следования их в формуле ряда.
2. **Попарное суммирование:** суммирование слагаемых парами, т.е. в общую сумму добавляется не , а . Разница между такими парами слагаемых меньше, чем между очередным слагаемым и частичной суммой ряда, поэтому больше битов мантиссы вносят вклад в результат.
3. **Обратное суммирование:** суммирование слагаемых в обратном порядке следования, т.е. вычисляются и сохраняются все члены последовательности и складываются, начиная с далее – .

Примечание: члены последовательности вычисляются, пока они больше по модулю некоторого числа ε. Для первого и второго суммирования . Для третьего: .

Помимо методов суммирования есть два способа передать аргумент в функцию ряда:

1. Передать «как есть» и вычислить сумму ряда от исходного аргумента;

2. С помощью эквивалентных преобразований и свойств мат. функций привести аргумент к меньшему значению, при котором ряд сходится быстрее, передать новый аргумент в функцию ряда и вычислить нужное значение дополнительными действиями.

В программе реализован метод приведения.

**Реализация метода**:

Для синусов и косинусов используются известные формулы приведения:

, ,

, ,

Недостатком такого приведения является рост погрешности с ростом аргумента. Как будет показано далее, для она составляет уже 0,01. С увеличением порядка в конечном итоге с ростом *x* значение функции совсем не будет похожим на настоящее.

Для экспоненты и натурального логарифма представим вещественную запись числа в нормализованном виде: , где s – знак числа (1 или -1), p – порядок, m – мантисса.

Для экспоненты, если , то никаких действий не требуется. Иначе:

Для логарифма: . Прим.: только для x > 0

# Руководство пользователя

Взаимодействие с программой осуществляется через консоль. Пользователю предлагается выбрать функцию, значение которой нужно вычислить, затем метод суммирования. После пользователь вводит значение аргумента. На экране выводятся значения функции, посчитанные рядом Маклорена и встроенной функцией, абсолютная и относительная погрешности. После пользователь может посчитать значение функции от другого аргумента, либо заново выбрать функцию и метод суммирования. При некорректном выборе функции или метода выбор осуществляется повторно. При вводе отрицательного числа или 0 в качестве аргумента логарифма программа завершает работу выдавая сообщение об ошибке.

# Описание программной реализации

В папке проекта доступны следующие файлы:

1. Файл Source.cpp – код программы, позволяющей взаимодействовать с пользователем через консоль;
2. Заголовочный файл с прототипами функций, необходимых для вычисления рядов TaylorSeries.h.
3. Файл реализации этих функций TaylorSeries.cpp
4. Файл решения Visual Studio TaylorLab.sln.
5. Файл проекта Visual Studio TaylorLab.vcxproj.
6. Файл Отчет.docx – данный файл с текстом отчёта.

# Подтверждение корректности

В целях подтверждения корректности проверим сходимость ряда к значению функции при увеличении числа членов последовательности. Если программа написана правильно, то при увеличении числа членов погрешность будет постепенно уменьшаться.

Продемонстрируем графики иллюстрирующие сходимость ряда при следующих значениях аргумента:

* для тригонометрических функций;
* 2 для экспоненты и логарифма.

Рис. 1.1 Погрешность вычисления синуса

Рис. 1.2 Погрешность вычисления косинуса

Рис. 1.3 Погрешность вычисления экспоненты

Рис. 1.4 Погрешность вычисления логарифма

# Результаты экспериментов

Будем проводить тесты, вычисляя абсолютные погрешности отдельно для больших значений аргумента и для близких к нулю.

Для больших значений:

* для тригонометрических функций возьмём отрезок от 1 до 10^7 с шагом в один старший разряд (1, 10, 100 и т.д.), и такие же отрицательные числа;
* для логарифма – то же, что и для тригонометрических (только положительные значения);
* lдля экспоненты – отрезок от -32 до 32 с шагом 1;

Для малых значений:

* для тригонометрических функций и экспоненты возьмём отрезок от -2 до 2 с шагом ;
* для экспоненты – только положительную его часть;

Ниже представлены графики абсолютных погрешностей:

Рис 2.1.1 Погрешности синуса. Большой отрицательный аргумент

Рис 2.1.2(а) Погрешности синуса. Большой положительный аргумент

Рис 2.1.2(б) Погрешности синуса. Большой положительный аргумент

Рис 2.2.1 Погрешности косинуса. Большой отрицательный аргумент

Рис 2.2.2(а) Погрешности косинуса. Большой положительный аргумент

Рис 2.2.2(б) Погрешности косинуса. Большой положительный аргумент

Рис 2.3 Погрешности экспоненты. Большой аргумент

Рис 2.4(а) Погрешности логарифма. Большой аргумент

Рис 2.4(б) Погрешности логарифма. Большой аргумент

Рис 3.1(а) Погрешности синуса. Малый аргумент

Рис 3.1(б) Погрешности синуса. Малый аргумент

Рис 3.2(а) Погрешности косинуса. Малый аргумент

Рис 3.2(б) Погрешности косинуса. Малый аргумент

Рис 3.3 Погрешности экспоненты. Малый аргумент

Рис 3.4(а) Погрешности логарифма. Малый аргумент

Рис 3.4(б) Погрешности логарифма. Малый аргумент

По графикам видно, что в одних случаях погрешности одинаковые (как например на рис 2.1.1, 2.1.2(б), 2.2.1, 2.2.2(б), 2.4(б), 3.4(б)), а в других различные способы более точны, чем два других (как например на рис 2.1.2(а), 2.2.2(а), 2.3, 2.4(а), 3.1(а, б), 3.2(а, б), 3.3, 3.4(а). Как и было сказано при |x|~10^6 погрешность вычисления тригонометрических функций > 0,01. Погрешность вычисления логарифма находится в окрестности 10^-6, что соответствует погрешности типа float для значений порядка 10. Погрешность вычисления экспоненты растёт примерно на 4-5 порядков при увеличении аргумента на 10, иначе говоря увеличении значения функции в . Из этого можно сделать вывод, что разность порядков значения функции и её погрешности ~ const. Таким образом можно сделать вывод, что значения синуса и косинуса вычисляются достаточно точно (погрешность < 1%) в пределах 10^6, а логарифм и экспонента – точно на всей области определения.

Далее найдём лучший метод суммирования для каждой функции. Для этого будем давать им баллы в зависимости от их точности. Самым точным будем начислять 1 балл, вторым по точности – 0,5 балла, третьим – 0. Ниже, в Таблице 1, представлены результаты вычислений.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Большие аргументы | | | | Малые аргументы | | | | Итого |
| sin | cos | exp | ln | sin | cos | exp | ln |
| Прямая | 105 | 110,5 | 39,5 | 60,5 | 44 | 53 | 40 | 56,5 | 315,5+193,5  =509 |
| Попарная | 113 | 111,5 | 49 | 15,5 | 51 | 58 | 49 | 30,5 | 289+188,5  =477,5 |
| Обратная | 109 | 99,5 | 50,5 | 61 | 64 | 48 | 56,5 | 57,5 | 320+226  =546 |

Таблица 1

По результатам можно сказать, что при вычислении выражений от больших и малых аргументов наиболее точными являются суммирования:

* для синусов – попарная и обратная соответственно;
* косинусов – попарная;
* экспоненты – обратная и попарная;
* логарифма – обратная;

Наименее же точными:

* для синусов – прямая;
* косинусов – обратная;
* экспоненты – прямая;
* логарифма – попарная.

В среднем по функциям, наиболее точным является обратное суммирование, а наименее – попарное (но без учёта логарифма – прямое).

# Заключение

1. Реализованы три алгоритма суммирования, подходящие для любых функций;

2. Была проведена проверка на сходимость ряда суммы. Во всех случаях ряд сходится;

3. Проведены тесты на точность реализованных алгоритмов, по результатам которых были

а) найдены области точного вычисления функций через ряды Маклорена;

б) выявлены наилучшие методы суммирования для каждой функции для больших и малых по модулю значений.

# Приложение

Код файла TaylorSeries.cpp в котором реализованы суммирующие функции, функции приведения и другие вспомогательные функции:

float sin\_next(float x, int i)

{

return -x \* x / (float)(2 \* i \* (2 \* i + 1));

}

float cos\_next(float x, int i)

{

return x \* x / (float)(2 \* i \* (1 - 2 \* i));

}

float exp\_next(float x, int i)

{

return x / (float)i;

}

float log\_next(float x, int i)

{

return x \* x \* float(2 \* i - 1) / float(2 \* i + 1);

}

void prepSin(float& x, int& n)

{

float const pi = 3.14159265f;

int m = 0;

float k = 1.0f;

n = 0;

if (x < 0.0f)

{

x = -x;

k = -1.0f;

}

while (x >= 4.0f)

{

x /= 4.0f;

m++;

}

while (m > 0)

{

n = (int) (x \* 2.0f / pi);

x -= (float)n \* pi / 2.0f;

x \*= 4.0f;

m--;

}

n = (int)(x \* 2.0f / pi);

x -= (float)n \* pi / 2.0f;

if (n % 2 == 1)

{

x -= pi / 2.0f;

n += 1;

}

if (n == 2)

x = -x;

x = k \* x;

return;

}

void prepCos(float& x, int& n)

{

float const pi = 3.14159265f;

int m = 0;

n = 0;

if (x < 0.0f)

x = -x;

while (x >= 4.0f)

{

x /= 4.0f;

m++;

}

while (m > 0)

{

n = (int)(x \* 2.0f / pi);

x -= (float)n \* pi / 2.0f;

x \*= 4.0f;

m--;

}

n = (int)(x \* 2.0f / pi);

x -= (float)n \* pi / 2.0f;

if (n % 2 == 1)

{

x -= pi / 2.0f;

n += 1;

}

if (n == 2)

n = -1;

else n = 1;

return;

}

void prepExp(float& x, int& p)

{

float k = 1.0f;

int tmp = 1, i;

p = 0;

if (x < 0.0f)

{

x = -x;

k = -1.0f;

}

if (x > 2.0f)

{

while (x > 2.0f)

{

x /= 2.0f;

p++;

}

for (i = 0; i < p; i++)

tmp \*= 2;

}

p = tmp;

x = k \* x;

return;

}

void prepLog(float& x, int& p)

{

p = 0;

if (x > 2.0f)

while (x > 2.0f)

{

x /= 2.0f;

p++;

}

else if (x < 1.0f)

while (x < 1.0f)

{

x \*= 2.0f;

p--;

}

return;

}

float directSum(float a, float const x, float(\*next)(float, int))

{

float res = a, eps = fabs(res) \* 1e-6f;

int i = 1;

while (fabs(a) > eps)

{

a \*= next(x, i++);

res += a;

eps = fabs(res) \* 1e-6f;

}

return res;

}

float extendedSum(float a, float const x, float(\*next)(float, int))

{

float res = a, tmp, eps = fabs(res) \* 1e-6f;

int i = 1;

a \*= next(x, i++);

res += a;

while (fabs(a) > eps)

{

tmp = (a \*= next(x, i++));

tmp += (a \*= next(x, i++));

res += tmp;

eps = fabs(res) \* 1e-6f;

}

return res;

}

float reverseSum(float a, float const x, float(\*next)(float, int))

{

float res = 0, arr[10000], eps = 1e-15f;

int i = 1, n = 20, tmp = n;

arr[0] = a;

while (fabs(a) > eps)

arr[i++] = a \*= next(x, i);

for (i = i - 1; i >= 0; i--)

res += arr[i];

return res;

}

float t\_sin(float const x, float(\*SumFunc)( float, float const, float(\*)(float, int) ))

{

int n;

float y = x;

prepSin(y, n);

return SumFunc(y, y, sin\_next);

}

float t\_cos(float const x, float(\*SumFunc)( float, float const, float(\*)(float, int) ))

{

int n;

float y = x;

prepCos(y, n);

return SumFunc(1.0f, y, cos\_next) \* (float)n;

}

float t\_exp(float const x, float(\*SumFunc)( float, float const, float(\*)(float, int) ))

{

int p, i;

float prod = 1.0f, y = x, term;

prepExp(y, p);

term = SumFunc(1, y, exp\_next);

for (i = 0; i < p; i++)

prod \*= term;

return term;