Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Ульяновский государственный технический университет»

Кафедра: «Вычислительная техника»

Лабораторная работа №2

Дисциплина: «Высокопроизводительные вычисления»

Тема: «Сравнительное исследование различных реализаций функций вещественных переменных»

Вариант 12

Выполнил

Студент группы ИВТАПбд-31

Галацоков И.А.

Проверил

профессор кафедры

«Вычислительная техника»

Негода В. Н.

Ульяновск, 2024

# Цель работы

Целью данной работы является изучение методов реализации функций от вещественных переменных, представленных степенными рядами.

В ходе исследования решаются следующие задачи:

* анализ возможностей уменьшения времени вычисления функций за счет снижения точности вычислений;
* анализ ошибок разложения аналитической функции в степенной ряд и формирование наиболее короткого конечного ряда из бесконечного при обеспечении заданных ограничений погрешности вычисления;
* оценка влияния программно-технического приема «разворачивание цикла» на время вычисления степенного ряда;
* оценка степени ускорения вычислений за счет применения схемы Горнера относительно реализации степенного ряда на основе использования функций возведения в степень и вычисления факториалов;
* оценка зависимости скорости вычислений от форматов чисел: с плавающей запятой, с фиксированной запятой;
* оценка степени ускорения вычислений за счет применения таблично-алгоритмических методов реализации функций, а также влияния характера потока значений аргументов на время вычисления(случайный поток значений, последовательный поток значений);
* оценка повторяемости результатов измерения и эффективности способов управления повторяемостью.

Индивидуальное задание выглядит следующим образом:

|  |  |  |  |
| --- | --- | --- | --- |
| Студент | Функция | maxErr | Метод интегрирования |
| Галацков Илья | Ln(1-x) | 2 ^(-16) | ПП |

# Описание платформы

1. Тип ЦП - AMD Ryzen 5 5500, 3500 MHz (35 x 100) (12 ядра / 12 потоков)
2. Оперативная память - 16 ГБ DDR4-2666 DDR4 SDRAM (24-18-18-43 @ 2666 МГц)
3. Видеоадаптер – AMD Vega 7
4. Накопитель –SSD M.2 512GB
5. Операционная система - Microsoft Windows 11 Home

# Анализ исходного прототипа

*Таблица 3.1 – Анализ преподавательского прототипа*

|  |  |  |  |
| --- | --- | --- | --- |
| *Модуль исходного текста* | *Функция/класс/декларация данных* | *Назначение* | *Характер требуемых изменений* |
| float.h | float flMathFunc(float x){} | Библиотечная реализация функции | Заменить на свою функцию ln(1-x) |
| float.h | float flCyNoGorn(float x){} | Цикл формулы ряда | Заменить на свой цикл формулы ряда для ln(1-x) |
| float.h | float flNoCyNoGorn(float x){} | Непосредственная реализация формулы ряда с определённом количеством полиномом (исходное количество - 5) | Заменить на свою формулы ряда для ln, которая будет состоять из 12 полиномов |
| float.h | int factorial(int n){} | Функция нахождения факториала с целочисленным типом | Замена типа на float |
| float.h | float flCycleGorn(float x) {} | Цикл схемы Горнера | - |
| float.h | float flCoef[LEN\_POLINOM] = {}; | Массив коэффициентов для схемы Горнера | Заменить на свои |
| float.h | float flCoef[LEN\_POLINOM] = { 1., -DIV1\_FACT3, DIV1\_FACT5, -DIV1\_FACT7, DIV1\_FACT9 }; | Массив коэффициент | Заменить на свои коэфициенты |
|  | // Бесцикловая схема Горнера классическая sin(x)/x = (((a[4]\*x^2 + a[3])\*x^2 + a[2])\*x^2 + a[1])\*x^2 + a[0]  float flNoCyGornArr(float x) {  float x2 = x \* x; // за скобки выносится x^2  return (((flCoef[4] \* x2 + flCoef[3])\*x2 + flCoef[2])\*x2 + flCoef[1])\*x2 + flCoef[0];  } |  |  |
| vpv-lab2.h | #define FACT3 (FACT2 \* 3)  #define FACT5 (FACT4 \* 5)  #define FACT7 (FACT7 \* 7)  #define FACT9 (FACT8 \* 9) | Факториалы | Добавить факториалы |
| vpv-lab2.h | #define DIV1\_FACT3 1/ FACT3  …  #define DIV1\_FACT9 1/ FACT9 | Обратные значения факториалов | Заменить на коэффициенты для схемы Горнера |
| vpv-lab2.h | #define LEN\_RES  #define MAX\_ERR  #define LEN\_POLINOM  #define X\_STEP | Разрядность результата функции, максимум погрешности, количество членов ряда, вес младшего разряда мантиссы float | Следует поменять в соответствии с вариантом задания |
| vpv-lab2.h | #define FLOAT2FIX(x)  #define FIX2FLOAT(x)  #define FIXMUL(x,y) | Преобразование числа с плавающей точкой в число с фиксированной, обратно и умножение чисел с фиксированной точкой | Без изменений |
| vpv-lab2.h | #define CPUID\_RDTSC(t) | Измерение времени в тактах по схеме "барьер - чтение TSC" | Без изменений |
| vpv-lab2.h | struct Result | Результат серии измерений с поддержкой сравнительного анализа | Без изменений |
| vpv-lab2.h | struct Config | Хранение настроек и параметров, связанных с измерением времени выполнения функций и анализом результатов | Без изменений |
| profiler.h | class Log | Накопление данных, фильтрация и статистическая обработка | Без изменений |
| profiler.h | class Tester | Базовый класс верификации - замера времени | Без изменений |
| profiler.h | class TestFloat : public Tester | Класс для float | Сделать реализацию дополнительного диапазона для обхода всех значений ряда |
| profiler.h | class TestFixed : public Tester | Тестер для фиксированной точки | Сделать реализацию дополнительного диапазона для обхода всех значений ряда |
| vpv-lab2.cpp | #define GOFLOAT 1  #define GOFIX 1  #define GOTABLE 1 | включение-выключение процессов испытания функций с плавающей точкой, фиксированной точкой, таблично-алгоритмических функций | можно отключать время-затратные фазы, не участвующие в отладке |
| vpv-lab2.cpp | Report report;  Config config | Инициализация config | Менять параметры, если нужно |
| vpv-lab2.cpp | int main(int argc, char\* argv[]) | Создание массива объектов тестирования и верификация | Без изменений |

# Анализ функции, заданной вариантом, представленная степенным рядом

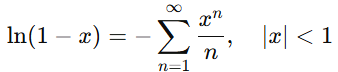
В качестве анализа функции представлена функция ln(1-x)

Первая функция, которая была изменена – это реализация функции на основе библиотечных (Листинг 4.1).

*Листинг 4.1 - Функция flMathFunc*

|  |
| --- |
| float flMathFunc(float x) {  return log(1 - x);  } |

Далее происходит разложение функции с помощью рядов (Рис. 4.1). Длина полинома определяется с помощью ряда экспериментов, которые показали, что необходимо полином из 12 членов.



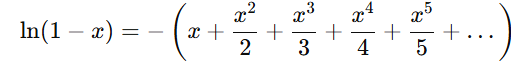
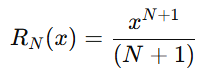


Рис. 4.1 – Степенной ряд для ln(1-x)

При проведении аналитической оценки такой длины степенного ряда для функции гиперболического секанса, при котором достигалась бы заданная вариантом задания точность в 2-16.

Мы знаем, что для достижения максимальной ошибки maxErr=2-16 нужно определить, при каком N остаточный член ряда не превышает этой ошибки.

Остаточный член суммы ряда Тейлора оценивается как:



Чтобы найти нужное количество членов, подставим максимальное |x| (обычно x=0.5 для быстрой сходимости) и решим неравенство:

Таким образом, минимальное число членов ряда, необходимое для достижения ошибки менее 2-16, составляет N=12

Листинг 4.3 – Реализации функций степенных рядов

|  |
| --- |
| float flCyNoGorn(float x) {  float sum = 0.0;  float xPow = x; // переменная для x^n  for (int n = 1; n < LEN\_POLINOM; n++) {  sum += flCoef[n - 1] \* xPow; // добавляем элемент ряда  xPow \*= x; // увеличиваем степень x  }  return sum;  }  // Непосредственная реализация формулы для ln(1-x)  float flNoCyNoGorn(float x) {  return -x + (1. / 2.) \* pow(x, 2) - (1. / 3.) \* pow(x, 3) + (1. / 4.) \* pow(x, 4)  - (1. / 5.) \* pow(x, 5) + (1. / 6.) \* pow(x, 6) - (1. / 7.) \* pow(x, 7)  + (1. / 8.) \* pow(x, 8) - (1. / 9.) \* pow(x, 9) + (1. / 10.) \* pow(x, 10)  - (1. / 11.) \* pow(x, 11) + (1. / 12.) \* pow(x, 12) - (1. / 13.) \* pow(x, 13)  + (1. / 14.) \* pow(x, 14) - (1. / 15.) \* pow(x, 15) + (1. / 16.) \* pow(x, 16)  - (1. / 17.) \* pow(x, 17) + (1. / 18.) \* pow(x, 18) - (1. / 19.) \* pow(x, 19)  + (1. / 20.) \* pow(x, 20) - (1. / 21.) \* pow(x, 21) + (1. / 22.) \* pow(x, 22)  - (1. / 23.) \* pow(x, 23) + (1. / 24.) \* pow(x, 24) - (1. / 25.) \* pow(x, 25)  + (1. / 26.) \* pow(x, 26) - (1. / 27.) \* pow(x, 27) + (1. / 28.) \* pow(x, 28)  - (1. / 29.) \* pow(x, 29) + (1. / 30.) \* pow(x, 30);  }  // Цикл схемы Горнера  float flCycleGorn(float x) {  float sum = flCoef[LEN\_POLINOM - 1]; // Начинаем с самого большого коэффициента  for (int n = LEN\_POLINOM - 2; n >= 0; n--) {  sum = sum \* x + flCoef[n]; // Схема Горнера  }  return sum;  }  // Бесцикловая схема Горнера классическая для ln(1-x) с использованием массива  float flNoCyGornArr(float x) {  float x2 = x \* x; // за скобки выносится x^2  return (((((((((((((((flCoef[LEN\_POLINOM - 1] \* x2 + flCoef[LEN\_POLINOM - 2]) \* x2 + flCoef[LEN\_POLINOM - 3]) \* x2 + flCoef[LEN\_POLINOM - 4]) \* x2 +  flCoef[LEN\_POLINOM - 5]) \* x2 + flCoef[LEN\_POLINOM - 6]) \* x2 + flCoef[LEN\_POLINOM - 7]) \* x2 + flCoef[LEN\_POLINOM - 8]) \* x2 +  flCoef[LEN\_POLINOM - 9]) \* x2 + flCoef[LEN\_POLINOM - 10]) \* x2 + flCoef[LEN\_POLINOM - 11]) \* x2 + flCoef[LEN\_POLINOM - 12]) \* x2 +  flCoef[LEN\_POLINOM - 13]) \* x2 + flCoef[LEN\_POLINOM - 14]) \* x2 + flCoef[LEN\_POLINOM - 15]) \* x2 + flCoef[LEN\_POLINOM - 16]) \* x2  + flCoef[LEN\_POLINOM - 17];  }  // Бесцикловая схема Горнера с константами вместо элементов массива для ln(1-x)  float flNoCyGornConst(float x) {  float x2 = x \* x;  return (((((((((((((((((((-1.0 \* x2 + 1.0 / 2.0) \* x2 - 1.0 / 3.0) \* x2 + 1.0 / 4.0) \* x2 - 1.0 / 5.0) \* x2 + 1.0 / 6.0) \* x2 -  1.0 / 7.0) \* x2 + 1.0 / 8.0) \* x2 - 1.0 / 9.0) \* x2 + 1.0 / 10.0) \* x2 - 1.0 / 11.0) \* x2 + 1.0 / 12.0) \* x2 - 1.0 / 13.0) \* x2 +  1.0 / 14.0) \* x2 - 1.0 / 15.0) \* x2 + 1.0 / 16.0) \* x2 - 1.0 / 17.0) \* x2 + 1.0 / 18.0) \* x2 - 1.0 / 19.0) \* x2 + 1.0 / 20.0) \* x2;  } |

Замеры этих функций представлена на Рис. 4.1.

   
 Рис. 4.1 – Результаты замеров чисел с плавающей точкой

Из этих результатов видно, что самой быстрой является "flNoCyGornArr", "Float - безцикловая схема Горнера (массив коэффициентов)", а самой долгой "flNoCyNoGorn"," Непосредственная реализация формулы". Можно сделать вывод, где циклов нет, там функции работаю быстрее.

Библиотечная реализация функции гиперболического секанса на основе библиотеки math.h для измерения получилась немного быстрее самой быстрой реализации. Помимо наличия циклов, на время могло повлиять еще и наличие ресурсоемких расчетов.

# Анализ функции, заданной вариантом с фиксированной точкой

Существует такая проблема когда производительность приложения может заметно ухудшиться из-за особенностей вычисления на числах с плавающей точкой. Как правило CPU заточен под целочисленные операции, а сопроцессор FPU (floating point unit) в нем работает на порядке медленнее.

Кроме этого, разрядность типа данных long, с которым предстоит работать в данном пункте задания, составляет 32 бита. Используется данный тип для того, чтобы наглядно была видна погрешность проводимых измерений, так как при работе с 64-битными числами она будет не так заметна и конечные результаты измерений могут получиться неточными.

Исходя из этого, появляется необходимость как-то представить числа, фигурирующие в формуле функции, заданной вариантом, в этих 32 битах, учитывая при этом знакопеременность или знакопостоянность получившегося степенного ряда и максимальную разрядность целой части коэффициентов в нём участвующих.

Для преобразования чисел с плавающей точкой в число с фиксированной точкой и обратно используются макросы (Листинг 5.1).

*Листинг 5.1 – Пример использования макросов переобразования*

|  |
| --- |
| // Фиксированные коэффициенты для ряда (для ln(1-x))  FixPoint fixCoef[LEN\_POLINOM] = {  -FLOAT2FIX(1.0), // -1  -FLOAT2FIX(0.5), // -1/2  -FLOAT2FIX(1.0 / 3), // -1/3  -FLOAT2FIX(1.0 / 4), // -1/4  -FLOAT2FIX(1.0 / 5), // -1/5  -FLOAT2FIX(1.0 / 6), // -1/6  -FLOAT2FIX(1.0 / 7), // -1/7  -FLOAT2FIX(1.0 / 8), // -1/8  -FLOAT2FIX(1.0 / 9), // -1/9  -FLOAT2FIX(1.0 / 10), // -1/10  -FLOAT2FIX(1.0 / 11), // -1/11  -FLOAT2FIX(1.0 / 12), // -1/12  -FLOAT2FIX(1.0 / 13), // -1/13  -FLOAT2FIX(1.0 / 14), // -1/14  -FLOAT2FIX(1.0 / 15), // -1/15  -FLOAT2FIX(1.0 / 16) // -1/16  }; |

В листинг 5.2 представлена реализация функция рядов, использоующие FixPoint.

*Листинг 5.2 – Функции с фиксированной точкой*

|  |
| --- |
| // Цикл схемы Горнера для ln(1 - x)  FixPoint fxCycleGorn(FixPoint x) {  FixPoint sum = 0;  for (int n = LEN\_POLINOM; n > 0; n--) {  sum = FIXMUL(sum, x) + fixCoef[n - 1];  }  return sum;  }  // Бесцикловая схема Горнера для ln(1 - x)  FixPoint fxNoCyGornArr(FixPoint x) {  FixPoint sum = sum = FIXMUL(fixCoef[12], x) + fixCoef[11];  sum = FIXMUL(sum, x) + fixCoef[10];  sum = FIXMUL(sum, x) + fixCoef[9];  sum = FIXMUL(sum, x) + fixCoef[8];  sum = FIXMUL(sum, x) + fixCoef[7];  sum = FIXMUL(sum, x) + fixCoef[6];  sum = FIXMUL(sum, x) + fixCoef[5];  sum = FIXMUL(sum, x) + fixCoef[4];  sum = FIXMUL(sum, x) + fixCoef[3];  sum = FIXMUL(sum, x) + fixCoef[2];  sum = FIXMUL(sum, x) + fixCoef[1];  sum = FIXMUL(sum, x) + fixCoef[0];  return sum;  }  FixPoint fxNoCyGornConst(FixPoint x) { // Бесцикловая схема Горнера для ln(1 - x)  FixPoint x2 = FIXMUL(x, x); // x^2  FixPoint sum = -x; // Начальный элемент (-x)  // Применяем разложение для ln(1 - x) по ряду Тейлора  sum = FIXMUL(sum, x2) - FIXDIV(FIXED1FLOAT, 2); // (-x + x^2/2)  sum = FIXMUL(sum, x2) - FIXDIV(FIXED1FLOAT, 3); // (-x + x^2/2 + x^3/3)  sum = FIXMUL(sum, x2) - FIXDIV(FIXED1FLOAT, 4); // (-x + x^2/2 + x^3/3 + x^4/4)  sum = FIXMUL(sum, x2) - FIXDIV(FIXED1FLOAT, 5); // (-x + x^2/2 + x^3/3 + x^4/4 + x^5/5)  sum = FIXMUL(sum, x2) - FIXDIV(FIXED1FLOAT, 6); // (-x + x^2/2 + x^3/3 + x^4/4 + x^5/5 + x^6/6)  sum = FIXMUL(sum, x2) - FIXDIV(FIXED1FLOAT, 7); // (-x + x^2/2 + x^3/3 + x^4/4 + x^5/5 + x^6/6 + x^7/7)  sum = FIXMUL(sum, x2) - FIXDIV(FIXED1FLOAT, 8);  sum = FIXMUL(sum, x2) - FIXDIV(FIXED1FLOAT, 9);  sum = FIXMUL(sum, x2) - FIXDIV(FIXED1FLOAT, 10);  sum = FIXMUL(sum, x2) - FIXDIV(FIXED1FLOAT, 11);  sum = FIXMUL(sum, x2) - FIXDIV(FIXED1FLOAT, 12);  // Завершаем разложение, когда достигнут нужный порядок для точности (maxErr = 2^-16)  return sum;  } |

На Рис. 5.1 представлен результат работы замера время для чисел с фиксированной точкой.

Рис. 5.1 - Результаты замеров чисел с фиксированной точкой

# Реализации на основе табличных методов и результаты их испытания

Сперва нужно преобразовать библиотечные функции, для этого нужно найти первую и вторую производную (Листинг 6.1).

Листинг 6.1. Реализация в коде прозводных

|  |
| --- |
| // Основная функция: ln(1 - x)  double dmathFunc(double x) {  return log(1 - x);  }  // Первая производная: f'(x) = -1 / (1 - x)  double firstDerivative(double x) {  return -1.0 / (1 - x);  }  // Вторая производная: f''(x) = -1 / (1 - x)^2  double secondDerivative(double x) {  return -1.0 / pow((1 - x), 2);  } |

Экспериментальным путем были подобраны длины степенных рядов для каждой из функций так, чтобы они соответствовали заданной точности вычисления:

Листинг 6.2. Длины степенных рядов

|  |
| --- |
| #define LEN\_ADDR0 20 // размер адреса таблицы для полинома нулевой степени  #define LEN\_ADDR1 11 // размер адреса таблицы для полинома первой степени  #define LEN\_ADDR2 7 // размер адреса таблицы для полинома второй степени |

На удивление полином 1-й степени работает быстрее чем полином 0-й степени. Казалось бы, сама функция меньше, выполняется меньше операций. Это объясняется тем, что при измерении времени происходит передача значения x в функцию посредством извлекания из массива arrX значений (массив заполнен случайными значениями от 0 до 1).

Для полинома 0-й степени просто находится значение в таблице по индексу. Соответственно по той причине, что каждый раз приходит случайное значение x, то у нас происходит большое количество кеш-промахов, потому что помимо того, что они случайные, так ещё и неизвестно какая будет разность между ними. Т.е. неизвестно как сильно будет прыгать индекс по таблице. И если происходят большие прыжки, то получаются кеш-промахи.

Для полинома 1-й и 2-й степени вычисляются коэффициенты уже по формулам. Поэтому вычислить математическое выражение окажется быстрее, чем случайно прыгать по индексам значений.

Результаты замеров табличных методов представлена на Рис. 6.1.

Рис. 6.1 – Результат замеров табличных методов.

# Разработка функций поддержки испытания созданных реализаций численного интегрирования/

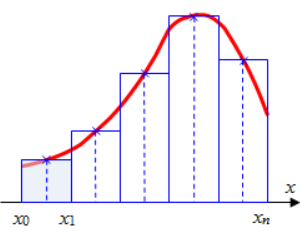
 Заданный вариантом задания метод численного интегрирования – метод средних прямоугольников (Рис. 7.1) с шириной прямоугольника, используемого при аппроксимации площади подинтегральной функции множеством площадей прямоугольников, равной 2-16 (*maxErr*).

Рис. 7.1 - Метод средних прямоугольников

Для подсчета интеграла создадим функцию замера времени, которая будет возвращать значение интеграла через метод средних прямоугольников (Листинг 7.1).

*Листинг 7.1 – Замер времени метода средних прямоугольников*

|  |
| --- |
| float measure\_integral() { // Замеры времени для repeat значений x in [0, 1) с равномерным шагом  \_\_int64 t1, t2;  float sum = 0.;  for (float x = 0; x < 1; x += MAX\_ERR) {  CPUID\_RDTSC(t1);  float fl = func(x + xStep / 2) \* MAX\_ERR;  CPUID\_RDTSC(t2);  sum += fl;  log.val.push\_back(t2 - t1 - overhead);  }  return sum;  } |

Добавим также функцию, работающую с готовыми измерениями времени и подсчитывающую статистику для них:

*Листинг 7.2 - Функция вычисления интеграла*

|  |
| --- |
| float timeSpentIntegral() {  if (proper) { // ЕСЛИ функциональный тест пройден  float sum = measureIntegral(); // ТО выполнить repeat измерений затрат времени  log.print(shortname);  log.calc(); // обработать протокол серии измерений  saveResult(); // сохранить результаты в report  log.val.clear();  return sum;  }  } |

*Листинг 7.3 - Функция вывода результатов*

|  |
| --- |
| for (Tester\* test : arr) {  float sum = test->timeSpentIntegral();  cout << "integral = " << sum << endl;  } |

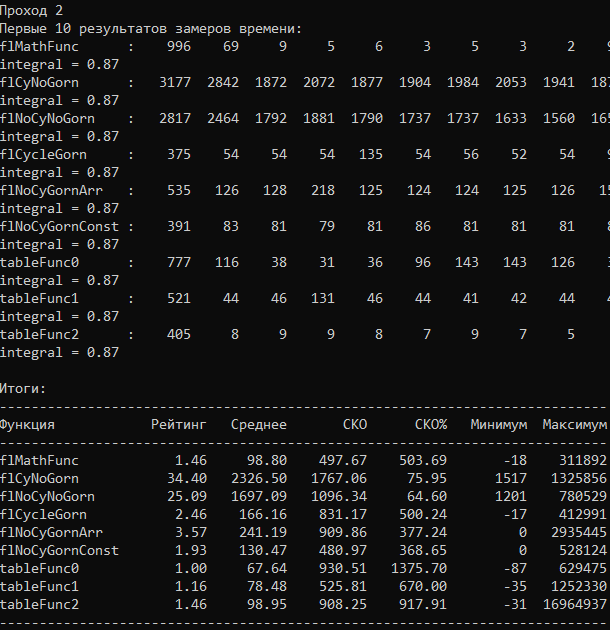
Результаты исследования численного интегрирования отображены на Рис.7.2.

Рис. 7.2 – Результат численного интегрирования

Заметное колебание результатов разных замеров для одной функции можно объяснить различием в срабатывании алгоритмов предсказания переходов, а для табличных методов – в кэш-промахах при обращении к таблицам коэффициентов. Результаты в последнем проходе для табличных методов заметно меньшие, поскольку кэш «прогревается».

Результаты отличаются от результатов со случайными числами. Самой быстрой реализацией оказался табличный метод с полиномом нулевой степени, поскольку при измерении времени через интеграл, у нас поступают не случайные x из arrX, как в изначальном случае, а линейно растущие значения от 0 до 1 с шагом MAX\_ERR, соответственно не будет никаких больших скачков или чего-то не определённого. Будет постепенно последовательно находиться элемент массива в таблице, что обеспечит 1 кэш-промах на 12 элементов. Здесь и главное отличие между предыдущим измерением времени и интегральным (Случайно прыгать на неопределённое расстояние против последовательного перемещения слева направо). К тому же в процессоре механизм перехода может предугадать следующее значение, если же такая технология имеется.

Самой долгой оказалась цикловая схема без схемы Горнера. Это по той причине, что интеграл вычисляется от нуля и инкрементируется на MAX\_ERR для таких небольших значений при вычислении интеграла требуется время для того чтобы вычислить заданное число с заданной точностью.

# Вывод

Проделав данную лабораторную работу, были изучены методы реализации функций от вещественных переменных, представленных степенными рядами. Также были приобретены навыки варьирования «затраты памяти – время реализации» путем применения различных алгоритмических и программно-технических решений.