МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский Авиационный Институт» (Национальный Исследовательский Университет)

Институт: №8 «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Курсовая работа

по курсу «Вычислительные системы» I семестр

Курсовая работа №3

«Вещественный тип. Приближенные вычисления. Табулирование функций»

|  |  |
| --- | --- |
| Группа | М8О-106Б-22 |
| Студент | Каримов А.А. |
| Преподаватель | Дубинин А.В. |
| Оценка |  |
| Дата |  |

Москва, 2022

Содержание

[Задача 2](#_Toc123675990)

[Формула Тейлора 2](#_Toc123675991)

[Стандарт IEEE-754 4](#_Toc123675992)

[Машинное эпсилон 5](#_Toc123675993)

[Описание работы программы 6](#_Toc123675994)

[Код программы 6](#_Toc123675995)

[Входные/Выходные данные 8](#_Toc123675996)

[Вывод 9](#_Toc123675997)

# Задача

Составить программу на языке Си, которая выводит значения функции синуса, вычисленной формулой Тейлора и библиотечной реализацией. Аргументы для этой функции берутся на заданном отрезке, который делится на n частей. Таким образом получается n+1 точек-аргументов. Вычисление по формуле Тейлора продолжается, пока очередной член ряда не будет меньше заданной погрешности.

# Формула Тейлора

­Пусть функция f(x) бесконечное число раз дифференцируема в окрестности некоторой точки x0. Допустим, что её можно представить в виде суммы степенного ряда, сходящегося в каком-то интервале, содержащем точку x0:

,

где a0,a1, …, an, … - неопределенные пока коэффициенты.

Найдем коэффициенты по известным значениям функции и её производных в точке x0:

Положим в равенстве x = x0. Отсюда:

В силу теоремы о дифференцируемости степенного ряда можно записать:

Снова положим x = x0. Тогда

При следующем дифференцировании получим:

Отсюда, полагая снова x = x0, найдем, что

После **n**-кратного дифференцирования получим:

,

Причем все остальные члены будут еще содержать множитель (x – x0) и, следовательно,

При x = x0 обратятся в ноль. Мы получим:

Таким образом находятся все коэффициенты a0,a1, …, an, … разложения :

…

Подставляя найденные выражения в равенство получим ряд:

Который называется рядом Тейлора функции

Таким образом, **Рядом Тейлора функции**  в окрестности точки x0 называется степенной ряд относительно разности , коэффициенты которого a0,a1, …, an , … выражаются через функцию и её производные в точке по формулам:

Существует ***теорема***:

Если в некотором интервале, окружающим точку x0, абсолютные величины всех производных функции ограничены одним и тем же числом, то функция в этом интервале разлагается в ряд Тейлора.

Отсюда разложим в ряд sin(x). Для этого находим последовательно значения ее производных в точке x = 0:

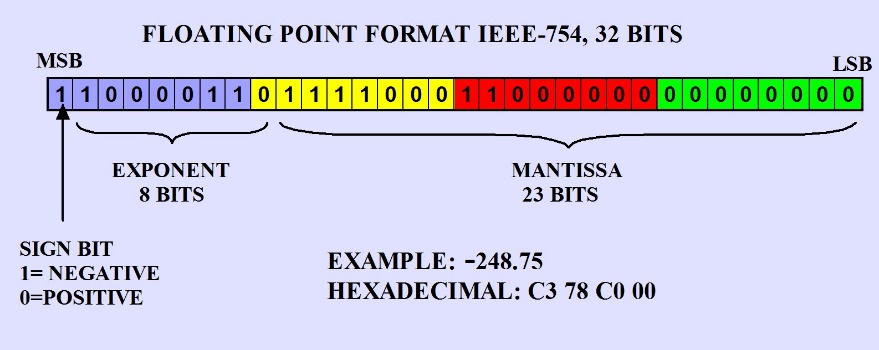
Мы видим, что значение производных повторяется и образует периодическую последовательность:

0, 1, 0, -1, 0, 1, 0, -1, 0, -1, ….

Любая производная функции sin(x) (т.е. ) по абсолютной величине не превосходит единицы. Следовательно, ряд для функции сходится к ней на всей числовой оси. Итак,

# Стандарт IEEE-754

IEEE-754 – это стандарт представления чисел с плавающей запятой, где один бит отводится для знака числа, а оставшиеся на порядок, показатель степени, и мантиссу. Числа в этом стандарте записываются в нормализованном экспоненциальном виде: мантисса числа лежит на отрезке [0,*b*], где b – основание системы счисления, то есть для двоичного представления первая цифра всегда равна единице. Так выглядит числа с плавающей запятой для типа *float* (32 бита).



Под порядок выделяется 8, 11, 15 бит в типах float, double и long double соответственно. За показатель, равный нулю, принимается примерно середина диапазона порядка. А порядок, состоящий из всех единиц, имеет специальные значения, зависящие от мантиссы, например, Quiet и Signaling NaN. Первые хранятся и копируются как простые значения, то есть мы не узнаем о их существовании, пока явно не выведем результат вычисления. А вторые могут помешать выполнению нашей программы, так как при получении Signaling NaN будет выброшено исключение от FPU (вычислительного блока процессора, который отвечает за вычисления с плавающей запятой). Программа получит сигнал SIGFPE (Floating Point Exception), который без дополнительной обработки сломает нашу программу.

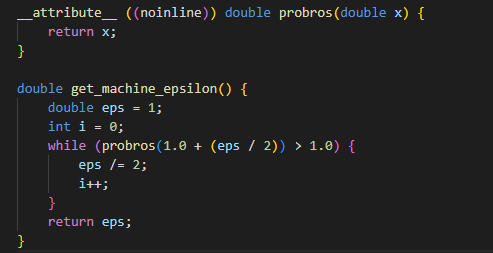
Хотя стандарт IEEE-754 не уточняет различие Signaling и Quiet NaN’ов, но большинство архитектур их отличает старший бит мантиссы: 1 – quiet, 0 – signaling.

Для каждого порядка, мантисса задает 252 разных равномерно распределенных чисел, то есть для порядка 0 диапазон будет [1; 2) есть 252 чисел, так мы получаем интервал между ними равный (2-1) / 252 = 2-52

Для порядка 1 диапазон [2; 4) => интервал равен 2-51. Продолжая вычисления для разного порядка, в том числе и в отрицательную сторону, мы увидим, что числа распределены равномерно, но длина участков, а также расстояние между соседними числами растет с ростом порядка. Это даёт нам возможность иметь высокую точность при работе с маленькими числами и низкую с большими, где низкая погрешность чаще всего не требуется.

# Машинное эпсилон

**Машинное эпсилон** – это минимальное число, которое при прибавлении к единице дает результат, отличный от единицы. Алгоритм получения *машинного эпсилон*:

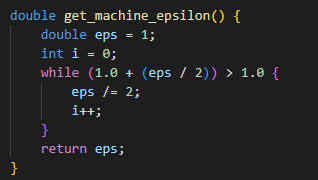


Как мы видим, обычным циклом алгоритм получения *машинного эпсилон* не ограничился. Всё из-за того, что при объявлении вещественных переменных таких как float (32 бита) или double (64 бита) процессор преобразует эти переменные в 80-битный тип long double. Это происходит автоматически при их загрузке в 80-битные регистры стека процессора. *Таким образом, все вычисления производятся на 80-битными вещественными значениями.*

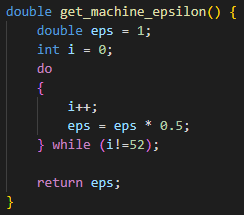
Следовательно, для решения проблемы вычисления машинного эпсилон ***для нужного типа*** я сбрасываю настоящее значение из регистра процессора в память, то есть привожу к хранимому в памяти виду.

Тем не менее, написав функцию проброса значения, нужно предотвратить рассмотрение функции для встраивания. То есть, если функция не имеет побочных эффектов, то оптимизации компилятора удалит её вызов. Для того чтобы избежать этот момент нужно воспользоваться атрибутом noinline, который исправит вышеописанную проблему.

Стоит отметить, что та же программа работает без подобных проблем в linux. Такое разное поведение объясняется тем, что на windows и linux используются разные компиляторы. Хоть мне и не удалось найти точного ответа, я наткнулся, как мне кажется, на что-то, что к этому ответу ведет: компилятор linux может преобразовать программу вычисления машинного эпсилон из



в

, то есть для оптимизации программы компилятор сводит цикл с проверкой отличия от единицы к циклу с определенным количеством итераций.

# Описание работы программы

При запуске программы следует ввести число n – количество точек разбиения отрезка [0; 1]. Затем программа вычисляет машинное эпсилон, чтобы использовать его для определения постоянной величины погрешности. Реализуется это делением единицы пополам, пока сумма единицы и этого числа больше 1. Затем в функции mysin() вычисляется значение синуса, пока очередной член ряда будет больше определенной величины погрешности.

# Код программы

#include <stdio.h>

#include <math.h>

\_\_attribute\_\_ ((noinline)) double probros(double x) {

    return x;

}

double get\_machine\_epsilon() {

    double eps = 1;

    int i = 0;

    while (probros(1.0 + (eps / 2)) > 1.0) {

        eps /= 2;

        i++;

    }

    return eps;

}

double binpow(double x, int y) {

    double res = 1;

    while (y) {

        if (y & 1) {

            res \*= x;

        }

        x \*= x;

        y >>= 1;

    }

    return res;

}

double myabs(double x){

    if(x >= 0) {

        return x;

    }

    return -x;

}

typedef struct {

    double res;

    int num\_of\_iterations;

} return\_helper;

return\_helper mysin(double x, double cst) {

    return\_helper ret;

    double res = x;

    unsigned long long k = 1;

    double sequence\_member, odd = -1;

    int num\_of\_iterations = 1;

    do{

        k \*= (2 \* num\_of\_iterations + 1) \* (2 \* num\_of\_iterations);

        sequence\_member = odd \* ((double)(binpow(x, 2\*num\_of\_iterations + 1)) / k);

        odd \*= -1;

        res += sequence\_member;

        num\_of\_iterations++;

    } while (myabs(sequence\_member) >= cst);

    ret.res = res;

    ret.num\_of\_iterations = num\_of\_iterations;

    return ret;

}

int main() {

    int segmentation;

    scanf("%d", &segmentation);

    double delta;

    double eps = get\_machine\_epsilon();

    double lower\_edge = 0;

    double upper\_edge = 1;

    double x = lower\_edge;

    double cst = get\_machine\_epsilon() \* binpow(2, 16);

    return\_helper ans;

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

        x = lower\_edge + i \* (upper\_edge - lower\_edge) / segmentation;

        ans = mysin(x, cst);

        delta = sin(x) - ans.res;

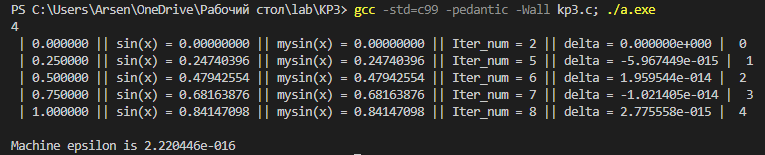
        printf( " | %lf || sin(x) = %.8lf || mysin(x) = %.8lf || Iter\_num = %d || delta = %e |  %d \n", x, sin(x), ans.res, ans.num\_of\_iterations, delta, i);

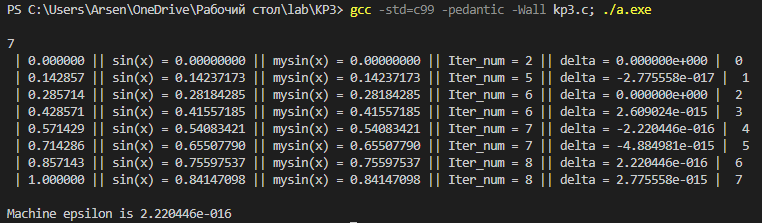
    }

    printf("\nMachine epsilon is %e\n", eps);

}

# Входные/Выходные данные





Здесь наглядно видно: чем дальше аргумент уходит от точки 0, тем больше нужно итераций для достижения точного значения функции, это объясняется тем, что синус по Тейлору считается для окрестности точки x0, которая в этом случае равна 0. Таким образом, получается, что по ряду Тейлора, чем ближе аргумент x находится к x0, тем ближе f(x) к f(x0), и тем ближе становятся к нулю последующие члены ряда. Из-за этого при отхождении от точки 0 требуется больше итераций для получения точного значения функции синуса.

# Вывод

В ходе выполнения курсового проекта №3 я составил программу, которая использует формулу Тейлора для нахождения значения синуса с определенной погрешностью, и сравнил полученный результат со встроенной функцией языка. У меня возникли трудности с получением правильного значения машинного эпсилон. Благодаря этой проблеме, я узнал, как процессор производит вычисления чисел с плавающей точкой, а также понял, что разные операционные системы могут по-разному компилировать файлы одного и того же языка, то есть разные ОС используют разные компиляторы, поведение которых может различаться при работе с одной и той же программой.

# Источники

1. Определение машинного эпсилон для типов double и float С++

[https://ru.stackoverflow.com/questions/1330474/ Определение-машинного-эпсилон-для-типов-double-и-float-в-С](https://ru.stackoverflow.com/questions/1330474/%20Определение-машинного-эпсилон-для-типов-double-и-float-в-С)

1. Бермант А.Ф. «Краткий курс математического анализа» - Издательство «Наука», 1966