Московский авиационный институт

(государственный технический университет)

**Курсовой проект**

**по курсу**

**«Архитектура ЭВМ, системное  
программное обеспечение»**

**1 семестр**

**Задание 3**

Выполнил: студент гр. 08-102  
Ридли Михаил

Проверил: Макаров Н.К.

Москва, 2010 год

# Постановка задачи

Написать программу на Си, которая выводит таблицу значений элементарной функции на отрезке со следующими ограничениями:

* Функция вычисляется для n точек отрезка , разделяющих его на равные части длиной .
* Аргументы функции последовательно беруться из отрезка в порядке возрастания с приращением .
* Функция вычисляется двумя способами: с помощью стандартных наборов функций языка, и разложением в ряд Тейлора . Оба результата должны быть зафиксированы в таблице.
* Точность аппроксимации рядом Тейлора задаётся как , где k – некоторый параметр, а . Считается, что как только , то получен ответ достаточной точности.
* Максимальное количество вычисляемых членов ряда Тейлора – 100 (хотя в соответствии с выбранным алгоритмом вычисления ряда Тейлора должно быть порядка 22).

На ввод программе подаётся количество частей отрезка n и параметр k, разделённые пробельными литерами.

Программа выводит таблицу реальных (под «реальными» здесь и далее подразумеваются значения, вычисленные с помощью композиции встроенных функций языка) и приближённых (конечным числом членов ряда Тейлора) значений n+1 точек функции в следующем формате (вход: 2 1e10):

|  |  |  |
| --- | --- | --- |
| **Реальное значение** | **Точка** | **Аппроксимация** |
| 1.106462828125 | 0.000 | 1.106462828936 |
| 1.436196219868 | 0.350 | 1.436196219903 |
| 1.876303742361 | 1.000 | 1.876303742531 |

# Организация исходного кода

|  |  |
| --- | --- |
| **Имя файла** | **Назначение** |
| func\_table.c | Генерация и вывод таблицы значений функции |
| helpers.h | Объявления вспомогательных функций |
| helpers.c | Реализация вспомогательных функций |

Как видно, программа поделена на файлы. Конечно, это может казаться лишним, но все-таки довольно полезно в ряде других случаев. Если рассматривать выпавшую программу, как частный случай, то да, можно делать из без разбиения на файлы, но в реальной жизни программистам часто требуется что-то менять, подстраивать под новые условия, и поэтому их интересует не столько, как работает функция, а что она в конечном итоге выдает. Поэтому, разбиение на файлы легко поможет программисту переделать программу за незначительное время из одного варианта в другой. Это, кстати, одна из причин появления в языке Си заголовочных файлов.

Для такой маленькой программы, наверное, и не стоит писать такое разбиение, ибо это может слегка затруднить программиста. Но если все сделать аккуратно и грамотно, то в конечном итоге поможет и организует работу. Поможет сделать программу более модифицируемой под иные схожие случаи.

## helpers.h, helpers.c

|  |  |  |
| --- | --- | --- |
| **Функция** | **Аргументы** | **Возвращает** |
| epsilon(void) | - | double |
| factorial(int n) |  | double |
| power(double base, int exponent) |  | double |

За более подробным и понятным человеку описанием можно обратиться непосредственно в helpers.h.

### Поиск машинного эпсилон (epsilon())

Реальные возможности современных компьютеров на самом деле довольно ограничены. Как, например, то, что существует только конечный набор чисел, которые можно представить в компьютере, они называются машинные числа вида:

* .
* Число называется основанием арифметики
* сумма в скобках – мантиссой
* m – длиной мантиссы
* k – порядком машинного числа .

Числа называются разрядами и их количество конечно и фиксировано.

Имеются некоторые дополнительные уточнения и ограничения, однако для данного рассмотрения они несущественны. Данная схема предложена К. Цузе и даёт эффективный способ «размена» точности на порядок числа, откуда получила название полулогарифмической.

Для множества представимых на ЭВМ целых чисел очевидно, что оно не совпадает с множеством целых чисел, рассматриваемых в математике: оно конечно. Для большинства схем кодирования целых чисел удобно использовать понятие классов вычетов, описывающего в том числе и арифметическое переполнение.

Ещё более очевидно, что для множества действительных чисел построение биекции между ним и множеством машинных чисел невозможно (к тому же, множество действительных чисел всюду плотно, что вызывает сложности даже при абстрактном восприятии).

Вспомним (или выдумаем) понятие окрестности точки из математического анализа[[1]](#endnote-1) как , где , а - метрика в ( – числовое множество). Очевидно, что окрестностью точки является некий шар . В случае (прямая) и этот шар, очевидно, представляет собой интервал следующего вида: . Обратив внимание на сходство с определением окрестности точки , заключаем для : .

Для получения конечного (а значит, что приближённого) представления множества действительных чисел, понятие окружности может оказаться крайне полезным.

Рассмотрим бинарное отношение между точками такое, что . Легко видеть, что каждая точка порождает класс эквивалентности . Составим множество различных классов эквивалентности точек из . Очевидно, , причём равенство истинно только при , что запрещено используемым нами определением окрестности. Таким образом, . Следовательно, подобный подход можно использовать для аккуратного уменьшения мощности числовых множеств путём приемлемого для задачи уменьшения точности.

В рассматриваемом случае и тогда обозначает всего лишь «точка принадлежит окрестности точки » или, что то же самое, «точка принадлежит интервалу для заданного ». Замечу, что всё это чем-то отдалённо похоже на классы вычетов, используемые при представлении в ЭВМ целых чисел, что наводит на мысль о понятийной связи.

Наша цель – достижение конечного множества, как можно более точно представляющего действительные числа при заданной точности, и у нас уже есть способ избавления от изоморфности отрезков в любым другим отрезкам в (подразумевается, что нижняя грань отрезка строго меньше верхней). Осталось лишь выбрать ограниченное сверху и снизу подмножество и применить к нему описанную выше процедуру для требуемой нам точности .

Замечу, что в нашем случае было бы гораздо удобнее и корректнее использовать полуоткрытые интервалы, а не открытые, однако это бы усложнило рассуждения несоразмерно полученной от этого выгоде.

Итак, машинная арифметика фактически оперирует с интервалами (окрестностями), а не с настоящими числами. Из-за этой особенности, если есть некое число , то для машины оно будет «неотличимо» от , если . С этом и заключается суть «интервальной» арифметики. Как несложно заметить, функции на машинных числах склонны к накоплению ошибок.

Машинное эпсилон – как раз то самое наименьшее представимое на конкретной ЭВМ положительное число такое, что , где оператор ‘+’ – машинное сложение. То есть это своего рода «минимально различимое» число вблизи нуля. Оно же, что следует из свойств интервальной арифметики – минимальная ошибка, возникающая при выполнении операций машинной арифметики, и называемая ошибкой округления.

Тогда округление[[2]](#endnote-2) – некое отображение из множества действительных чисел в множество машинных чисел. Если , а – результат отображения, то .

Отсюда можно вывести, что для некоторой операции ‘\*’. Видно, что именно это соотношение позволяет производить оценку ошибок округления. Замечу, что при многократном выполнении операций ошибка может становиться существенной, особенно если операция мультипликативная, поскольку тогда ошибка накапливается по экспоненциальному закону.

К сожалению, даже одна лишь ошибка округления сама по себе зачастую не так и безобидна, как может показаться на первый взгляд. Характерный пример подобной операции, приведённый в [2] – вычитание двух близких чисел. Её собственная ошибка округления имеет порядок , однако она способна усилить существующие ошибки.

Пусть . Тогда где .

Очевидно, может оказаться действительно большим, но даже , если происходит работа с нулём, может оказаться значимым, поскольку, по определению, , (к примеру, , где n – очень большое число).

Ясно, что при использовании интервальной арифметики необходимо пересмотреть многие привычные действия, такие как сравнение чисел. К примеру, в большинстве случаях вместо , нужно писать , а вместо записать . Таким образом, нужно всегда помнить о приближённом характере интервальной арифметики и о том, что ошибки имеют тенденцию накапливаться.

Значимость машинного эпсилон подразумевает необходимость динамического способа его нахождения. Воспользуемся дихотомией, т.е. делением пополам.

Дихотомия применительно к программированию – понятие, порождающее довольно общий класс алгоритмов, реализующих поиск в упорядоченном наборе данных не за , а за .

В общих словах идея дихотомии хорошо иллюстрируется детской игрой «угадай число», в которой один игрок загадывает число от до , а второй пытается его угадать за минимальное число шагов. При абстрагировании от везения, интуиции и когнитивных эффектов, легко предложить оптимальную стратегию поиска числа . Поскольку для двух чисел можно однозначно сказать, что (, т.е. существует только 3 возможности при сравнении чисел, мы можем свести поиск к последовательному делению пополам области поиска.

Пусть .  
При делении области поиска на 2 части можно сказать, что либо лежит в правой половине области, либо в левой, либо равно разделителю.

. Следовательно, можно сузить область поиска в два раза (от 0 до 256), т.к. лежит слева от . Разделяя новую область, получаем середину . Поскольку , выбираем левую половину (от 0 до 128). Повторяя действие, замечаем, что теперь лежит справа от середины отрезка (). Теперь можно сказать, что лежит между 64 и 128. Находя середину данного отрезка , сужаем область до . Придерживаясь подобной стратегии, мы гарантированно найдём максимум за делений области поиска пополам.

Условия применимости метода дихотомии для поиска в некоторой области поиска таковы:

1. Любые два элемента области поиска сравнимы ().
2. Область поиска является отсортированной (в каком порядке – несущественно, но предположим, что по возрастанию), т.е. для области поиска .
3. .

Существуют некоторые особенности применимости метода, такие как требование дискретности и конечности области поиска (в противном случае можно говорить лишь о сходимости с некоторой точностью).

Замечу, что метод можно существенно обобщить, однако достигаемые при этом улучшения в понимании несопоставимы с затратами.

Видно, что множество машинных чисел удовлетворяет условиям (1-3), а значит, что его элементы могут быть найдены с помощью метода дихотомии.

Найдём по определению с использованием дихотомии: взяв некое число вблизи , поделим его пополам, если . Как только , можно заключить, что .

Замечу, что в большинстве случаев существуют такие, что , однако этим обстоятельством принято пренебрегать.

### Факториал неотрицательного числа (factorial(n))

Факториалом числа называется количество перестановок множества мощностью . Тогда , а при .

Существует огромное количество способов вычисления факториала, однако в нашем случае используется вычисление факториала по определению.

Предположим существование некоторой функции высшего порядка , применяющую к каждому элементу списка функцию и накапливающую результаты этих вычислений, «соединяя» их с помощью функции двух аргументов, принимающей накопленное ранее значение и новое, которое требуется накопить. Изначальное значение, передаваемое как первый аргумент, равен . Реализация подобной функции может иметь следующий вид:

результат = i  
для каждого элемента e списка list:  
 результат = g(результат, f(e))  
вернуть результат

Это обобщение функции , реализуемой следующим образом:

результат = i  
для каждого элемента el списка list:  
 результат = f(результат, el)  
вернуть результат

Очевидно, функция может использоваться для вычисления, к примеру, факториала: , где \* - двухместная функция такая, возвращающая произведение аргументов.

Отмечу, что функция с операцией композиции и некоторыми простейшими списочными выражениями, образуют систему, полную по Тьюрингу.

Однако, вспоминая особенности реализации целых чисел в ЭВМ и замечая огромный рост при увеличении , понимаем, что для довольно малых будет возникать арифметическое переполнение.

Экспериментально выяснено, что для типа double, принятого в IEEE-754, максимальное , не вызывающее переполнения при вычислении – 22.

### Положительная степень числа n (power(base, exponent))

Число в положительной степени определяется как . Таким образом, поскольку операция является накопительной, она естественно реализуется с помощью функции : , где под подразумевается список, состоящий из указанного раз числа .

Однако реализация по определению редко бывает достаточно эффективной. Рассмотрим более экономный по времени алгоритм быстрого возведения в степень, основанный на следующих свойствах степеней, следующих непосредственно из свойств умножения на множестве действительных чисел: , причём . Очевидно, алгоритм вычисления степени по определению имеет временную сложность , а алгоритм быстрого возведения в степень – .

## func\_table.c

Напомню, исходный код, содержащийся в этом файле, отвечает за генерацию и вывод таблицы значений функции.

|  |  |  |
| --- | --- | --- |
| **Функция** | **Аргументы** | **Возвращает** |
| target\_f(double x) | X | double |
| taylor\_tem(double x, int n) |  | double |
| main(void) | - | Таблица значений функции target\_f |

Как и ранее, за понятными человеку обозначениями стоит обратиться в файл с исходным кодом.

### Значение целевой функции (target\_f(x))

Вычисляет значение функции в точке .

### n-ный член ряда Тейлора в точке (taylor\_term(x, n))

Вкратце поясним понятие ряда Тейлора.

Возьмём любую «хорошую» функцию около нуля, в нуле приближённо равную . Далее, рассмотрим разность , которая примерно линейна: , . Продолжая рассмотрение, получаем . В свою очередь, в нуле похожа на параболу . Продолжая по аналогии, получаем .

Раскладывая в ряд Тейлора, получаем:

Функция принимает и вычисляет значение -го члена разложения в ряд Тейлора.

### Основная функция (main())

|  |  |
| --- | --- |
| **Константа = значение** | **Смысл** |
| L\_BOUND = 0.0 | Левая граница отрезка, на котором вычисляется |
| R\_BOUND = 1.0 | Правая граница отрезка, на котором вычисляется |
| MAX\_ITER = 22 | Максимальное к-во вычисляемых членов ряда Тейлора |

|  |  |
| --- | --- |
| **Имя переменной** | **Назначение** |
| int parts | Количество частей отрезка [L\_BOUND; R\_BOUND] |
| double step | Приращение координаты точки |
| double eps | Минимальное [различимое] x: x + 1.0 != 1.0 |
| double k | eps\*k - точность приближения рядом Тейлора |
| double curr | Счётчик цикла вывода таблицы; текущая точка для |
| double goal | Результат непосредственного вычисления . |
| double res | Результат приближения рядом Тейлора |
| int iter | Номер вычисляемого члена Тейлора, итерации |

Очевидно, общий алгоритм вывода прямоугольной таблицы имеет структуру, в совпадающую с алгоритмом последовательного обхода двухмерного массива.

Однако в нашем случае он несколько упрощён и имеет следующий вид:

для каждой точки , где : // Это отдельные строки  
 подсчитать значение функции и его аппроксимацию в этой точке  
 вывести с отступами приближённое значение, точку, реальное значение

Поскольку другие шаги не представляют особого интереса и относительно очевидны, перейдём к описанию непосредственно вычисления приближённого значения с помощью её разложения в ряд Тейлора.

Замечу, что существует множество подходов к вычислению конечного числа членов ряда Тейлора, в том числе и простейших. Среди них хочется отметить вычисление по определению, по схеме Горнера и с помощью выражения текущего члена ряда через предшествующий (как правило, сводится к умножению).

Каждый из этих методов имеет свои недостатки. К примеру, применив схему Горнера к разложению данного случая, мы обнаружим нестабильность (из-за чередования знаков в разложении), а при выражении текущего члена ряда через предыдущий склонны накапливаться ошибки.

С другой стороны, постоянное перевычисление факториала не добавляет быстродействия способу «по определению», хотя этот эффект и может быть сведён к минимуму использованием кэширования для вычисления факториала. Однако поскольку факториал числа – невероятно быстро растущая функция, буквально сразу же возникает арифметическое переполнение (или ), делающее вычисления некорректными.

Метод вычисления по определению, ввиду особенностей машинной арифметики, также склонен к накоплению ошибок (в среднем менее существенных, чем в предыдущих случаях). Один из их источников таких ошибок – вычитание близких чисел, возникающее в случае знакочередования на членах, имеющих малый порядок, другой – полулогарифмический характер представления чисел в ЭВМ, из-за которого вычисление ряда в обратном порядке даст в среднем большую точность, чем в прямом порядке.

Тем не менее, для данной работы выбрано именно вычисление по определению. Одна из причин в том, что он обеспечивает достаточную для поставленной задачи точность, а другая – в его схожести с общим, привычным для программистов, алгоритмом вычисления сумм вида (полагается, что функции доступны внешние контекстные переменные, такие как ):

результат =   
для от до :  
 результат = результат +   
вывести результат

Замечу, что приведённый выше алгоритм естественно выражается в терминах определённых выше функций высшего порядка: или, вводя функцию , применяющую функцию к каждому элементу списка , реализованную следующим образом.

результат = []  
для каждого элемента списка :  
 добавить в результат   
вывести результат

Тогда , что более выразительно. Тогда .

Причина, по которой в данной работе использован прямой ход суммирования, а не обратный, не только в стремлении к использованию привычных понятий. Более существенная причина – проблематичность организации вычислений с заданной точностью, поскольку при обратном ходе суммирования необходимо (явно или нет) знать заранее количество вычисляемых членов разложения функции в ряд Тейлора, поскольку в противном случае можно не учесть наиболее «весомые» члены.

Замечу, что в нашем случае условие завершения вычислений несколько более сложное, чем описано выше. Поскольку значение функции в данной точки известно заранее, погрешность можно вычислять как расстояние между приближённым и реальным значениями функции.

Для наглядности пользователю предоставляется возможность влиять на приемлемую погрешность (радиус окрестности нуля) посредством вводимого с stdio параметра ().

Поскольку некоторые обсуждаемые функции имеют ограничения (к примеру, ), соответственно огранивается и количество итераций (в нашем случае – не более 22 членов ряда).

# Выводы

Несмотря на самостоятельную пользу разработки и написания программ, данная работа также побуждает к использованию интересных, во многом типичных, приёмов программирования, таких как вычисления с заданной точностью, суммирование рядов, обобщённый метод дихотомии, быстрое возведение в степень и так далее.

Не менее полезным и занимательным оказалось мыслительное, а позже и экспериментальное, исследование приближённых вычислений: к примеру, оказалось, что вычисление ряда Тейлора в обратном порядке имеет большую точность. Было замечено, что в общем случае ряд Тейлора – далеко не лучший способ получения приближённого значения функции, причём независимо от используемого метода вычисления собственно ряда Тейлора.

# Использованные источники

1. Математический анализ ч.1. – М.: МЦНМ, Зорич В.А, 2002 [↑](#endnote-ref-1)
2. Numerical Recipes in C: The Art of Scientific Computing, William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling, 1992

   4 Википедия, http://ru.wikipedia.org / [↑](#endnote-ref-2)