|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ «Информатика и системы управления»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»\_

***Лабораторная работа № 1***

**Тема:** Построение и программная реализация алгоритма полиномиальной интерполяции табличных функций.

**Студент:** Прянишников А.Н.

**Группа:** ИУ7-45Б

**Оценка (баллы):** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

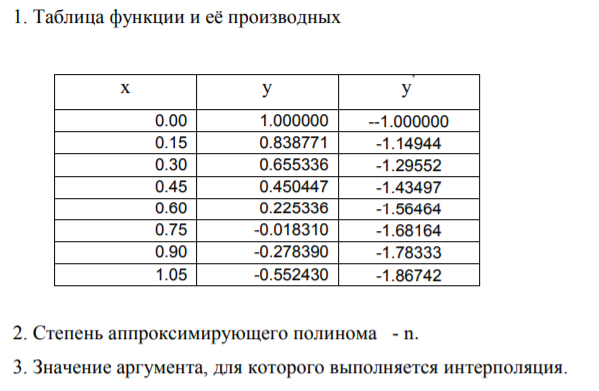
**Преподаватель:** Градов В.М.

*Москва*

*2021 г*

**Цель работы:** Получение навыков построения алгоритма интерполяции таблично заданных функций полиномами Ньютона и Эрмита.

1. **Исходные данные**

****

1. **Идея реализации**

Для выполнения интерполяции заданной функции применялись алгоритмы построения интерполяционного полинома Ньютона и Эрмита. Выполнение задачи разбито на несколько этапов:

1. Программа по заданному x и n вычисляла индексы элементов, которые нужно брать из таблицы значений. Для этого находился индекс элемента, который первый в массиве больше, чем заданный x, а затем по n вычислялось, какие индексы брать.
2. Составлялся массив из результатов вычислений разделённых разностей по исходной таблице значений. Это происходило и для полинома Ньютона, и для полинома Эрмита, но для Эрмита использовалась также и таблица производных.
3. Вычислялось значение в заданной точке x по формуле вычислений полинома для алгоритмов.

Для обратной интерполяции использовался алгоритм Ньютона, но столбцы X и Y менялись местами, и в качестве x задавался 0. То, что решение есть, проверялось через то, что знаки функции в концах отрезка, где ищем решение, разные. Практически все функции универсальны, и они одновременно использовались и для алгоритма полинома Ньютона и полинома Эрмита.

Также для задачи было сделано утверждение, что функция непрерывная.

1. **Код программы**

# Для реализации массива был выбран массив NumPy

**import** **numpy** **as** **np**

# Табличные значения, заданные в лабораторной работе

X\_TABLE = np.array([**0**, **0.15**, **0.3**, **0.45**, **0.6**, **0.75**, **0.9**, **1.05**])

Y\_TABLE = np.array([**1**, **0.838771**, **0.655336**, **0.450447**, **0.225336**, -**0.018310**, -**0.278390**, -**0.552430**])

Y\_DER\_TABLE = np.array([-**1**, -**1.14944**, -**1.29552**, -**1.43497**, -**1.56464**, -**1.68164**, -**1.78333**, -**1.86742**])

# Функция подсчёта разделённой разности по заданным индексам

# x\_table, y\_table - таблицы значений, y\_der\_table - таблица значений производных

# list\_indexes - индексы элементов, с которыми работаем

# Возвращает численный результат

# Функция нахождения индекса элемента, который первый больше, чем заданный x

# x\_var - заданная переменная

# x\_table - табличные значения

**def** **find\_near\_index**(x\_var, x\_table):

length\_table = len(x\_table)

index = **0**

**while** ((index < length\_table) **and** (x\_table[index] < x\_var)):

index += **1**

**return** index

# Функция расчёта индексов для составления интерполяционного полинома Ньютона

# x\_var - заданная переменная

# n - степень полинома

# x\_table - табличные значения

**def** **create\_list\_indexes\_newton**(x\_var, n, x\_table):

index = find\_near\_index(x\_var, x\_table)

list\_indexes = np.array(list(range(index - ((n // **2**) + **1**), index - ((n // **2**) + **1**) + n + **1**)))

**while** (list\_indexes[**0**] < **0**):

list\_indexes += **1**

while (list\_indexes[list\_indexes.shape[**0**] - 1] >= len(x\_table)):

list\_indexes -= **1**

**return** list(list\_indexes)

# Функция расчёта индексов для составления интерполяционного полинома Эрмита

# x\_var - заданная переменная

# n - степень полинома

# x\_table - табличные значения

**def** **create\_list\_indexes\_ermit**(x\_var, n, x\_table):

index = find\_near\_index(x, x\_table)

n\_start = n

n //= **2**

list\_indexes = np.array(list(range(index - ((n // **2**) + **1**), index - ((n // **2**) + **1**) + n + **1**)))

**while** (list\_indexes[**0**] < **0**):

list\_indexes += **1**

while (list\_indexes[list\_indexes.shape[**0**] - 1] >= len(x\_table)):

list\_indexes -= **1**

result\_list\_indexes = []

length = len(list\_indexes)

**for** i **in** range(length):

result\_list\_indexes.append(list\_indexes[i])

result\_list\_indexes.append(list\_indexes[i])

**if** (n\_start % **2** == **0**):

result\_list\_indexes.pop()

**return** result\_list\_indexes

# Функция подсчёта разделённой разности по заданным индексам

# x\_table, y\_table - таблицы значений, y\_der\_table - таблица значений производных

# list\_indexes - индексы элементов, с которыми работаем

# Возвращает численный результат

**def** **difference**(x\_table, y\_table, list\_indexes, y\_der\_table = []):

result = **0**

length = len(list\_indexes)

# Если всего два элемента

**if** (length == **2**):

# Элементы для подсчёта разделённой разности различны

**if** (list\_indexes[**0**] != list\_indexes[**1**]):

delta\_y = y\_table[list\_indexes[**0**]] - y\_table[list\_indexes[**1**]]

delta\_x = x\_table[list\_indexes[**0**]] - x\_table[list\_indexes[**1**]]

result = delta\_y / delta\_x

# Если элементы одинаковые, то результат - значение производной в точке

**else**:

result = y\_der\_table[list\_indexes[**0**]]

# Если элементов больше, то подсчитываем через рекурсию

**elif** (length > **2**):

dif\_first = difference(x\_table, y\_table, list\_indexes[:length - **1**], y\_der\_table)

dif\_second = difference(x\_table, y\_table, list\_indexes[**1**:], y\_der\_table)

result = (dif\_first - dif\_second) / (x\_table[list\_indexes[**0**]] - x\_table[list\_indexes[length - **1**]])

**return** result

# Функция подсчёта коэффициентов для построения полиномов

# x\_table, y\_table - таблицы значений, y\_der\_table - таблица значений производных

# list\_indexes - индексы элементов, с которыми работаем

# Возвращает массив коэффициентов для полинома

**def** **count\_polinom\_diff\_coefs**(x\_table, y\_table, list\_indexes, y\_der\_table = []):

result\_coefs = []

**for** i **in** range(**2**, len(list\_indexes) + **1**):

result\_coefs.append(difference(x\_table, y\_table, list\_indexes[:i], y\_der\_table))

**return** result\_coefs

# Функция подсчёта значения полинома

# x\_table, y\_table - таблицы значений, y\_der\_table - таблица значений производных

# list\_indexes - индексы элементов, с которыми работаем

# x\_var - переменная, для которой ищем значение у апроксимирующей функции

# Возвращает результат по заданному x

**def** **count\_polinom\_value**(x\_table, y\_table, x\_var, list\_indexes, y\_der\_table = []):

coefs = count\_polinom\_diff\_coefs(x\_table, y\_table, list\_indexes, y\_der\_table)

x\_multiply = **1**

result = y\_table[list\_indexes[**0**]]

**for** i **in** range(len(list\_indexes) - **1**):

x\_multiply \*= (x\_var - x\_table[list\_indexes[i]])

result += (x\_multiply \* coefs[i])

**return** result

x = float(input("Введите X, для которого нужно найти значение: "))

**for** n **in** range(**1**, **7**):

**print**("n =", n)

**print**("Ньютон:")

list\_indexes = create\_list\_indexes\_newton(x, n, X\_TABLE)

**print**(count\_polinom\_value(X\_TABLE, Y\_TABLE, x, list\_indexes, Y\_DER\_TABLE))

**print**("Эрмит:")

list\_indexes = create\_list\_indexes\_ermit(x, n, X\_TABLE)

**print**(count\_polinom\_value(X\_TABLE, Y\_TABLE, x, list\_indexes, Y\_DER\_TABLE))

**print**("Решение уравнения y(x) = 0:")

list\_indexes = create\_list\_indexes\_newton(**0**, n, Y\_TABLE)

**print**(count\_polinom\_value(Y\_TABLE, X\_TABLE, **0**, list\_indexes, Y\_DER\_TABLE))

**print**("-"\***20**)

# Результаты

**Этап 1: тестирование алгоритма**

# Сначала надо убедиться, что программа выдаёт правильные результаты на простейших функциях и обычных примерах. Для этого проверим работу функции на известных узлах и линейной функции, где легко предсказать результат.

# Линейная функция: y = 2x

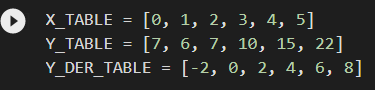
# 

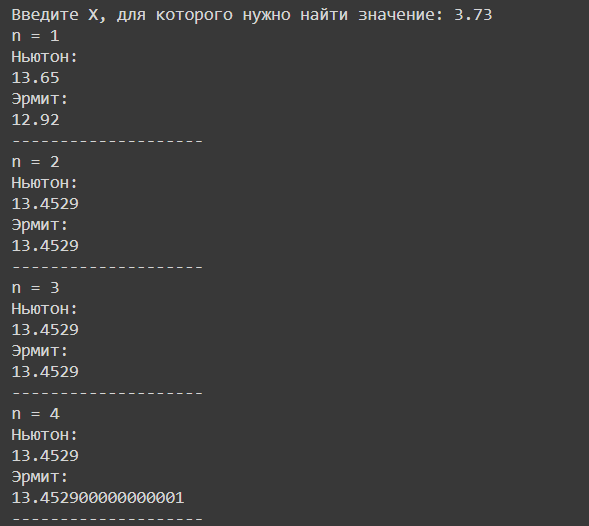
# 

Сведём результаты в отдельную таблицу (истинное значение x = 5):

|  |  |  |
| --- | --- | --- |
|  | Ньютон | Эрмит |
| N = 1 | 5.0 | 5.0 |
| N = 2 | 5.0 | 5.0 |
| N = 3 | 5.0 | 5.0 |
| N = 4 | 5.0 | 5.0 |

* Парабола y = x ^ 2 – 2x + 7



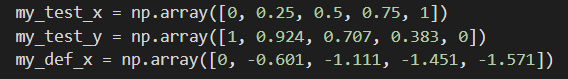


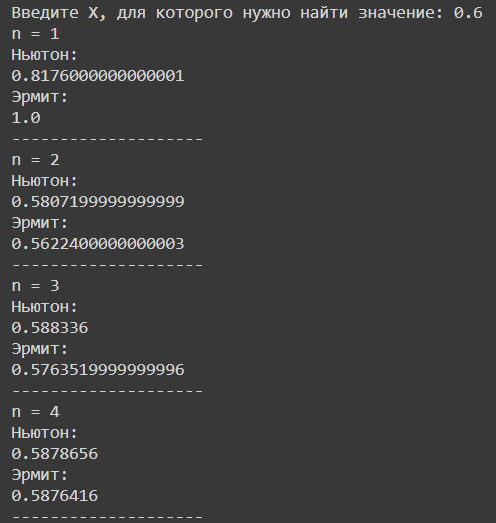
Опять составим таблицу для x = 3.73 (истинное значение y = 13.4529)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Ньютон | Эрмит | Истинное |
| N = 1 | 13.65 | 12.92 | 13.4529 |
| N = 2 | 13.4529 | 13.4529 | 13.4529 |
| N = 3 | 13.4529 | 13.4529 | 13.4529 |
| N = 4 | 13.4529 | 13.4529 | 13.4529 |

Получили результат, который и ожидали: параболу алгоритмы отлично предсказывают с n = 2.

* Пример из лекции: cos(Pi / 2 \* x)





Составим таблицу:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Ньютон | Эрмит | Истинное значение |
| N = 1 | 0.8176 | 1.0 | 0.587785 |
| N = 2 | 0.58072 | 0.56224 | 0.587785 |
| N = 3 | 0.588336 | 0.576352 | 0.587785 |
| N = 4 | 0.5878656 | 0.5876416 | 0.587785 |

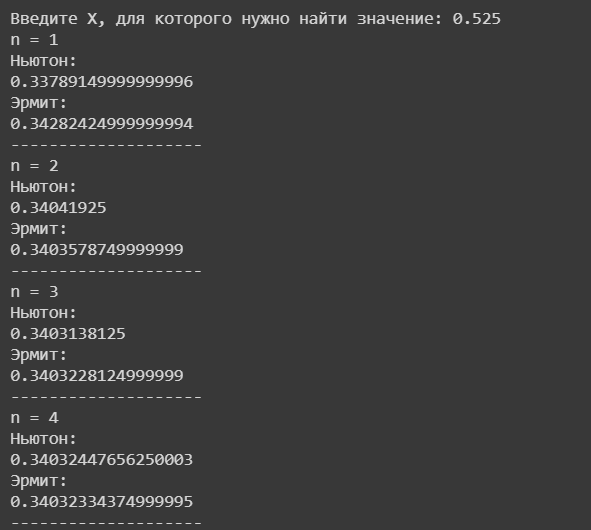
Уже с N = 2 алгоритмы близко сошлись к истинному значению, Ньютон показал лучшие результаты, чем Эрмит.

Теперь можно приступить непосредственно к решению задачи.

**Этап 2: Подсчёт значений для задания**

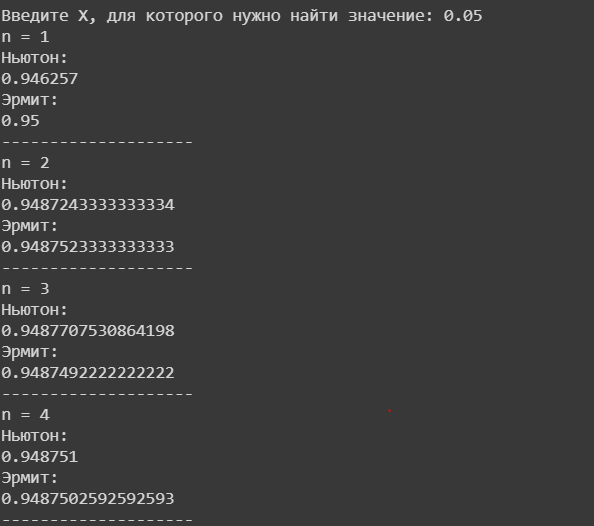
Рассчитаем значения для середины интервала (x = 0.525), а также для краёв (x = 0.05, 0.98), чтобы проверить, насколько отдалённость от центра влияет на точность вычислений. Табличная функция: y = cos(x) – x

**X = 0.525**



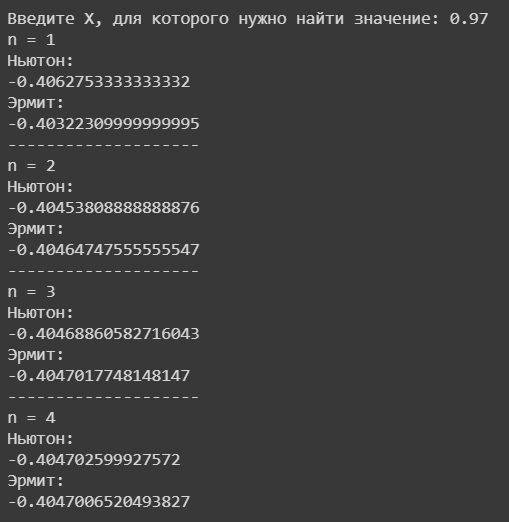
|  |  |  |  |
| --- | --- | --- | --- |
|  | Ньютон | Эрмит | Истинное |
| N = 1 | 0.337891 | 0.342824 | 0.340323 |
| N = 2 | 0.340419 | 0.340358 | 0.340323 |
| N = 3 | 0.340314 | 0.340323 | 0.340323 |
| N = 4 | 0.340323 | 0.340323 | 0.340323 |

**X = 0.05**



|  |  |  |  |
| --- | --- | --- | --- |
|  | Ньютон | Эрмит | Истинное |
| N = 1 | 0.946257 | 0.95 | 0.948750 |
| N = 2 | 0.948724 | 0.948752 | 0.948750 |
| N = 3 | 0.948749 | 0.948749 | 0.948750 |
| N = 4 | 0.948750 | 0.948750 | 0.948750 |

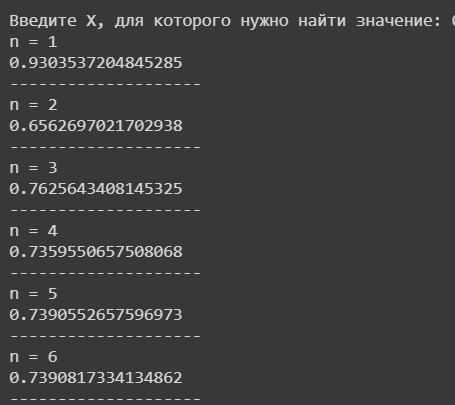
**X = 0.97**



|  |  |  |  |
| --- | --- | --- | --- |
|  | Ньютон | Эрмит | Истинное |
| N = 1 | -0.406275 | -0.403223 | -0.404700 |
| N = 2 | -0.404538 | -0.404647 | -0.404700 |
| N = 3 | -0.404688 | -0.404701 | -0.404700 |
| N = 4 | -0.404702 | -0.404700 | -0.404700 |

Как видно, алгоритмы неплохо справились и на концах отрезка, и во всех случаях программа действительно смогла уже с n = 2 подобрать адекватное приближение к исходной функции.

Теперь решим уравнение методом обратной интерполяции:



|  |  |  |
| --- | --- | --- |
|  | Ньютон | Корень |
| N = 1 | 0.930354 | 0.739085 |
| N = 2 | 0.656269 | 0.739085 |
| N = 3 | 0.762564 | 0.739085 |
| N = 4 | 0.735955 | 0.739085 |
| N = 5 | 0.739085 | 0.739085 |
| N = 6 | 0.739085 | 0.739085 |

1. **Выводы**

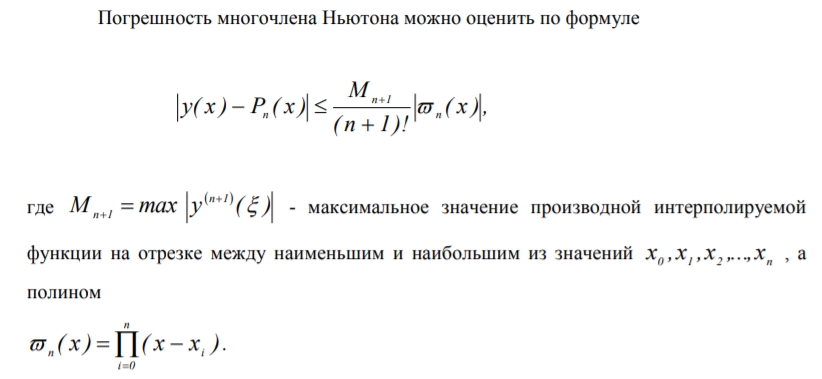
Методы построения интерполяционных полиномов Ньютона и Эрмита позволяют эффективно найти значения функции на тех промежутках, где нет табличных результатов. В лабораторной работе удалось для заданной таблицы подсчитать результаты с минимальной погрешностью начиная n = 2, относительная погрешность редко превышала 1% с истинным значением. Для обратной интерполяции значение близко сошлось только с n = 3.

1. **Вопросы к защите лабораторной работы**
2. **Будет ли работать программа при степени полинома n=0?**

Программа будет работать, но значение функции будет всегда рассчитываться как значение ближайшего предыдущего узла, причём и полинома Ньютона, и для полинома Эрмита.

1. **Как практически оценить погрешность интерполяции? Почему сложно применить для этих целей теоретическую оценку?**

Теоретическая оценка погрешности:



Такой способ неудобен тем, что нужно знать производные функции, поэтому на практике можно поступить так: оценить первый отбрасываемый член, меньший заданной точности.

1. **Если в двух точках заданы значения функции и ее первых производных, то полином какой минимальной степени может быть построен на этих точках?**

Для Ньютона мы можем построить полиномы от 0 до 1, а для Эрмита – от 0 до 3 степени включительно. Минимальная степень в обоих случаях всегда 0, но как понятно, результат будет не слишком соответствовать правде.

1. **В каком месте алгоритма построения полинома существенна информация об упорядоченности аргумента функции (возрастает, убывает)?**

Эта информация нужна для того, чтобы корректно выбрать выборку нужного размера из таблицы (в моём случае – правильно рассчитать индексы), иначе мы не сможем выполнить адекватно интерполяцию.

1. **Что такое выравнивающие переменные и как их применить для повышения точности интерполяции?**

Выравнивающие переменные – переменные, которые позволяют снизить порядок зависимости в случае, если функция имеет разный вид на различных интервалах. В лучшем случае нелинейная зависимость может превратиться в линейную. Пример выравнивающей переменной: x` = ln(x). Таким образом, упрощая зависимость до вычислимой, мы повышаем точность интерполяции.

Также для повышения точности требуется больше табличных данных.