

ГУАП

КАФЕДРА № 42

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ
ПРЕПОДАВАТЕЛЬ

ст. преподаватель

должность, уч. степень, звание

подпись, дата

В.А. Миклуш

инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №3

ОЦЕНКА ГЕНЕРАЛЬНЫХ ПАРАМЕТРОВ СЛУЧАЙНЫХ ВЕЛИЧИН

по курсу: Теория информации, данные, знания

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. №

4128

подпись, дата

В. А. Воробьев

инициалы, фамилия

Санкт-Петербург 2023

Цель работы: Нахождение основных статистических характеристик распределения случайных величин, построение функции распределения, определение закона распределения в среде математического редактора.

Задание

1. Найти числовые параметры распределения.
2. Найти закон распределения.
3. Провести степенную аппроксимацию произвольно выбранных данных

Вариант задания - №5

Ход работы

В ходе работы была создана программа на языке Python, выполняющая все поставленные задачи. Исходный код программы представлен в Приложении.

1 Числовые параметры распределения

- Объем выборки: 500
- X минимальный: -8.01379391414643
- X максимальный: 6.144979813287666
- Выборочное математическое ожидание (среднее арифметическое всех элементов выборки)

$$\text{Формула для МО: } m_x^* = \sum_{i=1}^n x_i p_i = \frac{1}{n} \sum_{i=1}^n x_i$$

$$Mx: -0.45437341526625824$$

- Число степеней свободы выборки (количество значений или наблюдений в выборке, которые могут быть изменены независимо друг от друга без изменения ее структуры).

$$f: 499$$

- Выборочная дисперсия (среднее квадратов отклонений каждого элемента выборки от выборочного среднего)

Формулы для выборочной дисперсии:

$$\Delta_x^* = \sum_{i=1}^n (x_i - m_x^*)^2 p_i = \frac{1}{n} \sum_{i=1}^n (x_i - m_x^*)^2$$

$$D(D_x^*) = D\left(\frac{n}{n-1} \Delta_x^*\right) = \left(\frac{n}{n-1}\right)^2 D(\Delta_x^*) = 1 * 0 = 0$$

$$Dx: 5.362700285478515$$

- СКО(корень дисперсии), S_x : 2.315750479969402
- Асимметрия (отражает степень и направление отклонения распределения от симметрии).

$$\text{Формула для асимметрии: } a_x^* = \frac{\sqrt{n}}{\sqrt{(n-1)^3 (\sigma_x^*)^3}} \sum_{i=1}^n (x_i - m_x^*)^3$$

$$A_x: -0.07735713294746685$$

- Эксцесс (характеризует остроту или плоскость вершины распределения).

$$\text{Формула для эксцесса: } e_x^* = \frac{n}{(n-1)^2 (\sigma_x^*)^4} \sum_{i=1}^n (x_i - m_x^*)^4$$

$$E_x: -0.14901143278322104$$

- Медиана (значение, которое разделяет выборку на две равные части), Med_x : -0.5205897905121429
- Размах (разница между максимальным и минимальным значениями выборки), R : 14.158773727434095
- Дисперсия асимметрии (измеряет степень изменчивости асимметрии в различных выборках), Da : 0.011880810942726873
- Дисперсия эксцесса (показывает степень изменчивости эксцесса в различных выборках), De : 0.00011549620221635089
- Доверительные интервалы — это интервалы, которые с высокой вероятностью содержат истинное значение параметра генеральной совокупности. Ниже представлены доверительные интервалы при $q = 0.95$.

1. Для генерального математического ожидания

$$-0.657847686888222 \leq m_x \leq -0.25089914364429444$$

2. Для генеральной дисперсии:

$$4.754863872733365 \leq D_x \leq 6.0956708362501315$$

3. Для генеральной асимметрии:

$$-0.291940529496238 \leq a_x \leq 0.1372262636013043$$

4. Для генерального эксцесса:

$$-0.573063147083164 \leq e_x \leq 0.27504028151672194$$

Результат вычисление числовых параметров изображен на рисунке 1.

```
Объем выборки = 500
Xmax = 6.144979813287666
Xmin = -8.01379391414643
Мат. ожидание = -0.45437341526625824
Степени свободы = 499
Дисперсия = 5.362700285478515
СК0 = 2.315750479969402
Ассиметрия = -0.07735713294746685
Эксцесс = -0.14901143278322104
Медиана = -0.5205897905121429
Размах = 14.158773727434095

Дисперсия ассимтерии = 0.011880810942726873
Дисперсия эксцесса = 0.00011549620221635089
Доверительные интервалы при q = 0.95:
МО – (-0.657847686888222, -0.25089914364429444)
Дисперсия – (4.754863872733365, 6.0956708362501315)
Ассиметрия – (-0.291940529496238, 0.1372262636013043)
Эксцесс – (-0.573063147083164, 0.27504028151672194)
```

Рисунок 1 – Числовые параметры распределения

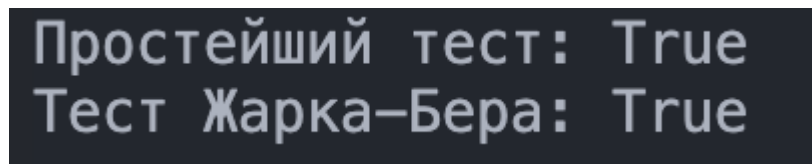
2 Найти закон распределения

1. Проверим нашу выборку на нормальность с помощью: по простейшего теста и теста Жарка-Бера. Для проверки возьмём один уровень значимости $q = 5\%$.

а. Простейший тест: принимаем основную гипотезу.

б. Тест Жарка-Бера: принимаем основную гипотезу.

Результат прохождения тестов был выведен на консоль и представлен на рисунке 2.



```
Простейший тест: True
Тест Жарка-Бера: True
```

Рисунок 2 – Результат проверки выборки

2. Строим гистограмму плотности распределения для заданного массива исходных данных в сравнении с теоретическими законами распределения плотности случайных величин. Результат представлен на рисунке 3.

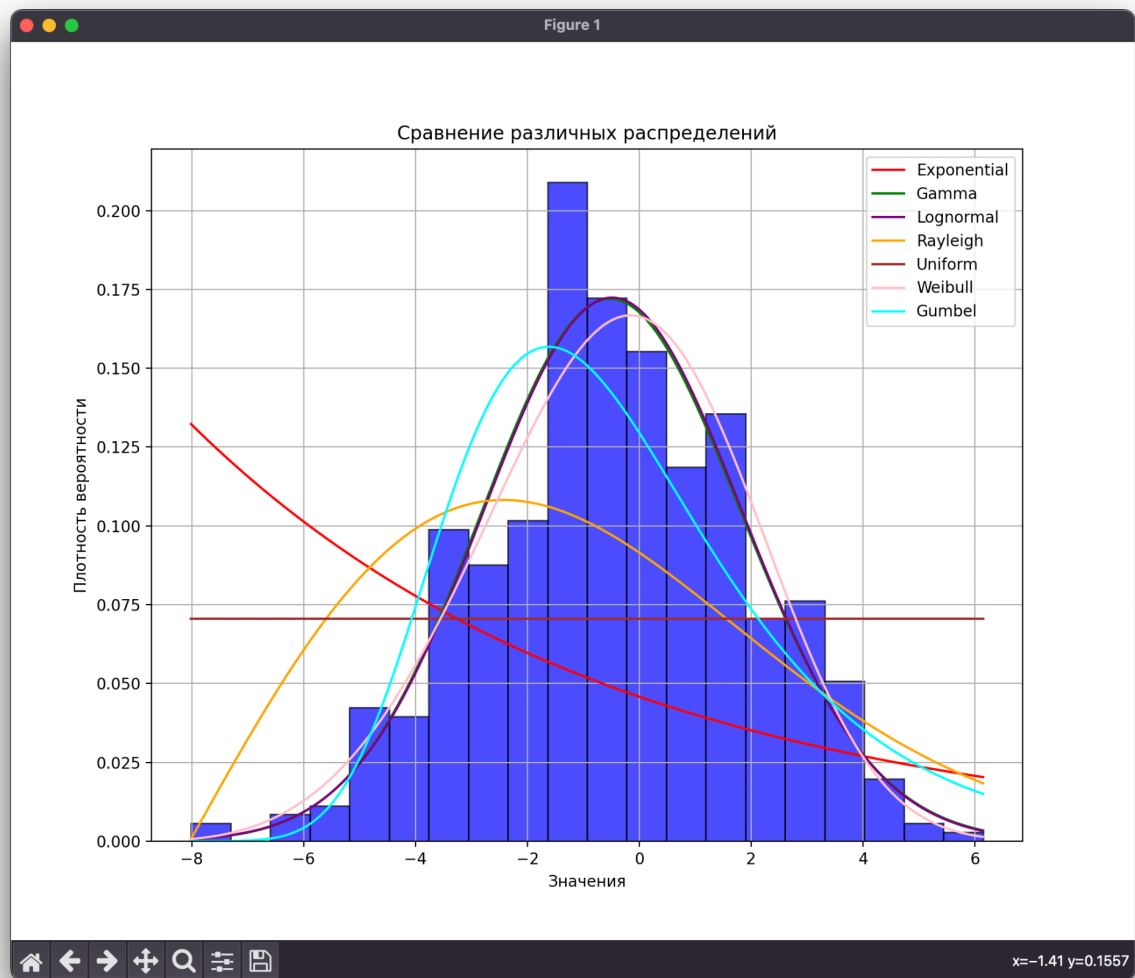


Рисунок 3 – Эмпирическая и теоретические плотности распределения

3. Строим графики функции распределения. Результат представлен на рисунке 4.

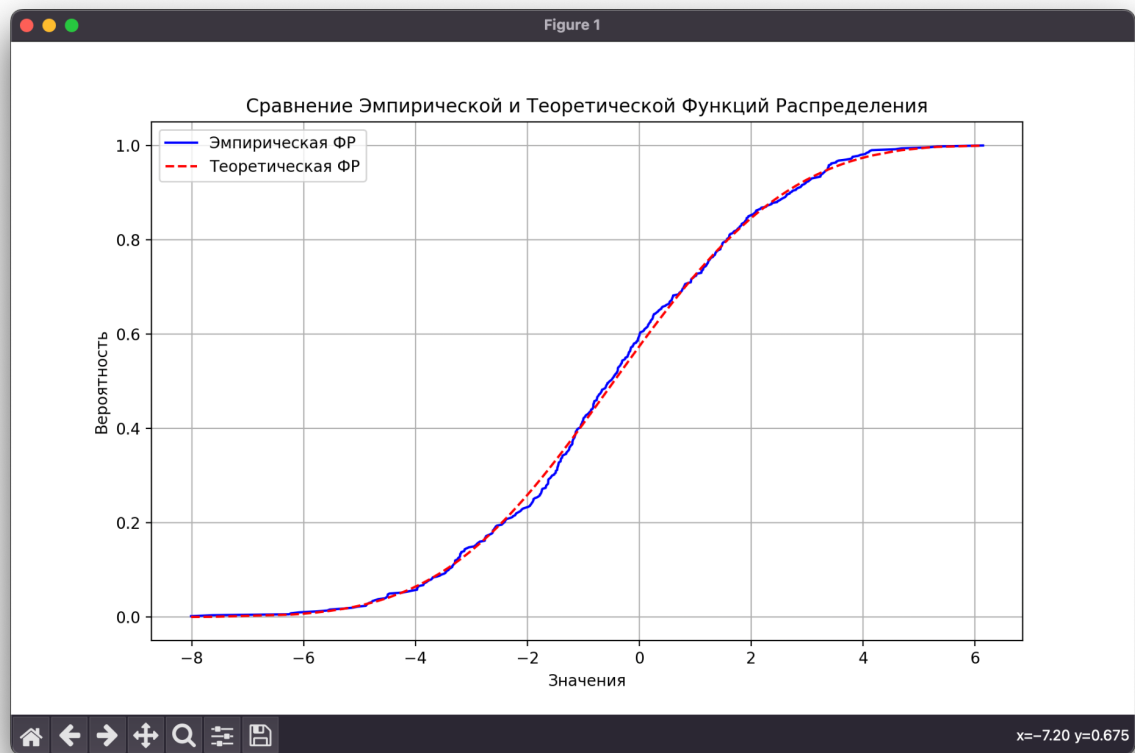


Рисунок 4 – Эмпирическая и теоретические плотности распределения

4. Проверка непрерывных распределений по критерию Колмогорова
= 0.93435435244320 у нормального распределения.

3 Степенная аппроксимацию произвольно выбранных данных

Степенная аппроксимация данных представляет собой метод приближения данных к функции степенного закона. Этот метод может быть использован для приближения нелинейных данных, где зависимость между переменными может быть описана степенной функцией. Проведем ее для функции гармонических колебаний с шумом.

Результат аппроксимации представлены на рисунках 5, 6 и 7.

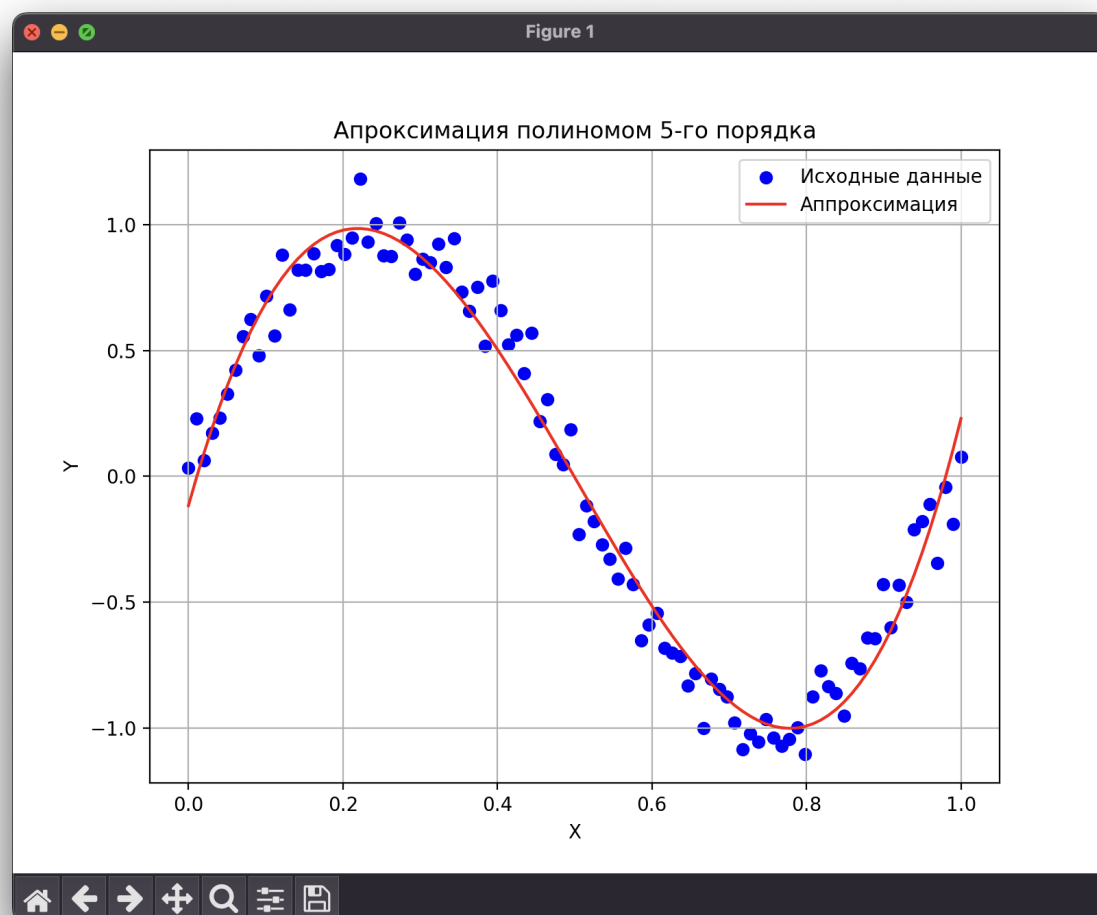


Рисунок 5 – Аппроксимация полиномом 5-го порядка

Коэффициенты: [2.22078437 18.14679832 -31.0409053 11.0205625 -0.13659455]

Рисунок 6 – Коэффициенты полинома

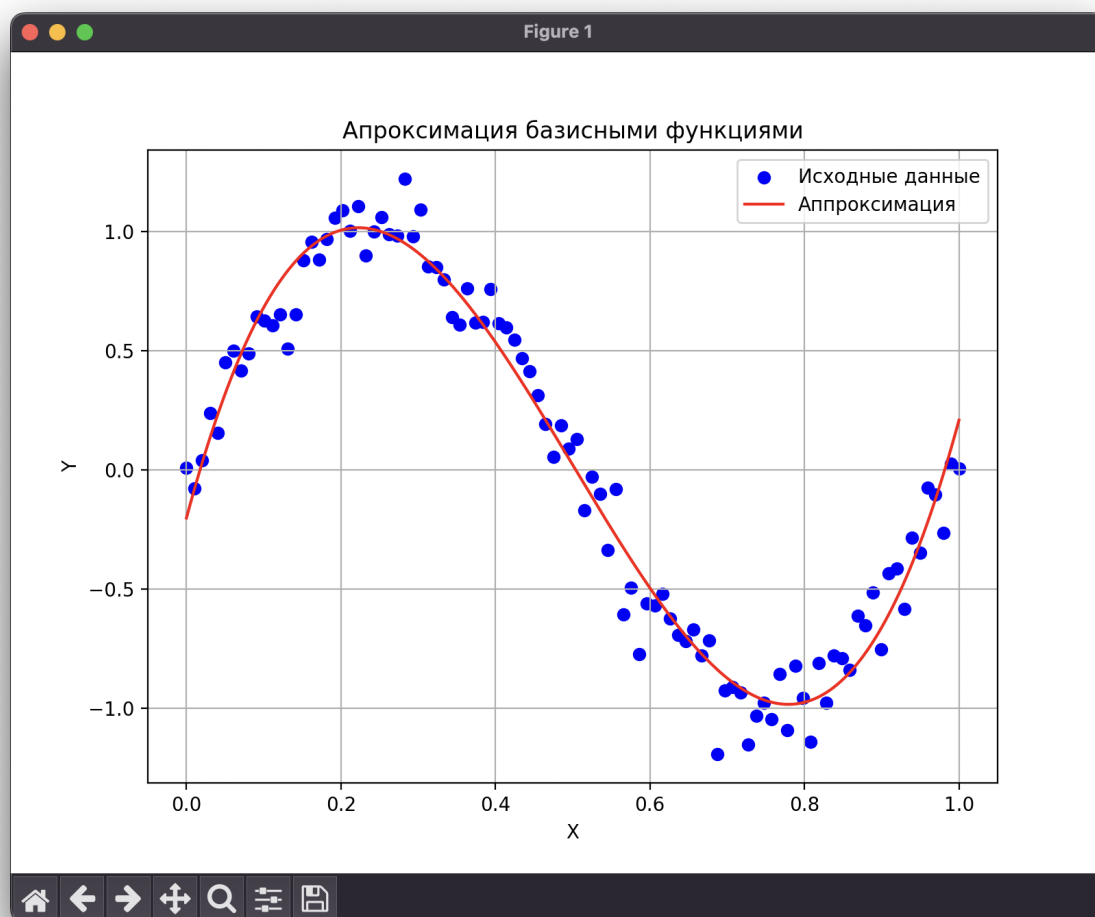


Рисунок 7 – Аппроксимация базисными функциями

ВЫВОД

В ходе выполнения лабораторной работы, на основе исходных данных, соответствующих варианту с использованием языка программирования Python, была создана программа, вычисляющая необходимые статистические значения и построены гистограмма распределения, функции распределения и степенная аппроксимация данных гармонической функции при помощи базисных функций и полинома 5-ой степени.

ПРИЛОЖЕНИЕ

ИСХОДНЫЙ КОД

ploter.py

```
from statsmodels.distributions.empirical_distribution import ECDF
```

```
from scipy.stats import genextreme
```

```
from typing import List
```

```
import matplotlib.pyplot as plt
```

```
import numpy as np
```

```
from scipy.stats import gamma, expon, lognorm, rayleigh, uniform, weibull_min,  
gumbel_r
```

```
def plot_data_and_approximation(x_values, y_values, approx_points):
```

```
    """
```

Строит график точек исходной выборки и аппроксимированной функции.

Параметры:

- x_values (np.ndarray): Массив значений x исходной выборки.
- y_values (np.ndarray): Массив значений y исходной выборки.

- approx_points (list): Список точек аппроксимированной кривой в формате (x, y).

```
"""
```

```
plt.figure(figsize=(8, 6))
```

```
# График исходной выборки
```

```
plt.scatter(x_values, y_values, label='Исходные данные', color='blue')
```

```
# График аппроксимированной функции
```

```
x_approx, y_approx = zip(*approx_points)
```

```
plt.plot(x_approx, y_approx, label='Аппроксимация', color='red')
```

```
# Настройки графика
```

```
plt.title('Аппроксимация полиномом')
```

```
plt.xlabel('X')
```

```
plt.ylabel('Y')
```

```
plt.legend()
```

```
plt.grid(True)
```

```
# Отображение графика
```

```
plt.show()
```

```
def plot_histogram(values, bins=32):
```

```
    """
```

Строит гистограмму распределения случайной величины.

Параметры:

- values (List[float]): Список значений случайной величины.

- bins (int): Количество интервалов гистограммы (по умолчанию 10).

```
    """
```

```
plt.figure(figsize=(8, 6))
```

```
# Строим гистограмму
```

```
plt.hist(values, bins=bins, color='blue', edgecolor='black', alpha=0.7)
```

```
# Настройки графика
```

```
plt.title('Гистограмма распределения случайной величины')
```

```
plt.xlabel('Значения')
```

```
plt.ylabel('Частота')
```

```
plt.grid(True)
```

```
# Отображение гистограммы
```

```
plt.show()
```

```
def plot_all_distributions(data: List[float]):
```

```
    """
```

Строит график гистограммы и плотностей вероятности различных распределений.

Параметры:

- data (List[float]): Массив значений случайных величин.

```
    """
```

```
plt.figure(figsize=(10, 8))
```

```
# Строим гистограмму
```

```
plt.hist(data, bins=20, density=True, alpha=0.7,
```

```
        color='blue', edgecolor='black')
```

```
# Оцениваем параметры и строим плотности вероятности
```

```
x = np.linspace(min(data), max(data), 1000)
```

```
# Экспоненциальное распределение
```

```
exp_params = expon.fit(data)
```

```
exp_pdf = expon.pdf(x, *exp_params)
```

```
plt.plot(x, exp_pdf, label='Exponential', color='red')
```

```
# Гамма-распределение
```

```
gamma_params = gamma.fit(data)
```

```
gamma_pdf = gamma.pdf(x, *gamma_params)
```

```
plt.plot(x, gamma_pdf, label='Gamma', color='green')
```

```
# Логнормальное распределение
```

```
lognorm_params = lognorm.fit(data)
```

```
lognorm_pdf = lognorm.pdf(x, *lognorm_params)
```

```
plt.plot(x, lognorm_pdf, label='Lognormal', color='purple')
```



```
# Рэлеевское распределение
```

```
rayleigh_params = rayleigh.fit(data)
```

```
rayleigh_pdf = rayleigh.pdf(x, *rayleigh_params)
```

```
plt.plot(x, rayleigh_pdf, label='Rayleigh', color='orange')
```

```
# Равномерное распределение
```

```
uniform_params = uniform.fit(data)
```

```
uniform_pdf = uniform.pdf(x, *uniform_params)
```

```
plt.plot(x, uniform_pdf, label='Uniform', color='brown')
```

```
# Вейбулловское распределение
```

```
weibull_params = weibull_min.fit(data)
```

```
weibull_pdf = weibull_min.pdf(x, *weibull_params)
```

```
plt.plot(x, weibull_pdf, label='Weibull', color='pink')
```

```
gumbel_params = gumbel_r.fit(data)
```

```
gumbel_pdf = gumbel_r.pdf(x, *gumbel_params)
```

```
plt.plot(x, gumbel_pdf, label='Gumbel', color='cyan')
```

```
# Настройки графика
```

```
plt.title('Сравнение различных распределений')
```

```
plt.xlabel('Значения')
```

```
plt.ylabel('Плотность вероятности')
```

```
plt.legend()
```

```
plt.grid(True)
```

```
# Отображение графика
```

```
plt.show()
```

```
def plot_empirical_vs_theoretical_cdf(data: List[float],  
theoretical_distribution='genextreme'):
```

```
    """
```

Строит график эмпирической и теоретической функций распределения.

Параметры:

- data (List[float]): Массив значений случайных величин.

- theoretical_distribution (str): Название теоретического распределения ('genextreme' по умолчанию).

```
"""
```

```
plt.figure(figsize=(10, 6))
```

```
# Эмпирическая функция распределения (ЭФР)
```

```
ecdf = ECDF(data)
```

```
x_values = np.sort(data)
```

```
y_values_empirical = ecdf(x_values)
```

```
# Теоретическая функция распределения (ТФР)
```

```
if theoretical_distribution == 'genextreme':
```

```
    params = genextreme.fit(data)
```

```
    y_values_theoretical = genextreme.cdf(x_values, *params)
```

```
# Добавьте другие теоретические распределения, если необходимо
```

```
# Строим графики
```

```
plt.plot(x_values, y_values_empirical,
```

```
        label='Эмпирическая ФР', color='blue')
```

```
plt.plot(x_values, y_values_theoretical,
```

```
        label='Теоретическая ФР', linestyle='--', color='red')
```

```
# Настройки графика

plt.title('Сравнение Эмпирической и Теоретической Функций
Распределения')

plt.xlabel('Значения')

plt.ylabel('Вероятность')

plt.legend()

plt.grid(True)


# Отображение графика

plt.show()
```

distribution_params.py

```
from scipy.stats import skew
```

```
from typing import List
```

```
from typing import List, Tuple
```

```
from scipy.stats import chi2, t, skewnorm
```

```
from typing import List, Optional
```

```
import numpy as np
```

```
def sample_size(data: List[float]) -> int:
```

```
    """
```

Возвращает объем выборки.

Параметры:

- data (List[float]): Массив значений случайных величин.

Возвращает:

- size (int): Объем выборки.

```
    """
```

```
    return len(data)
```

```
def calculate_Xmin(data: List[float]) -> Optional[float]:
```

```
    """
```

Возвращает минимальное значение в выборке.

Параметры:

- data (List[float]): Массив значений случайных величин.

Возвращает:

- Xmin (float): Минимальное значение в выборке, или None, если выборка пуста.

```
"""
```

```
if not data:
```

```
    return None
```

```
return min(data)
```

```
def calculate_Xmax(data: List[float]) -> Optional[float]:
```

```
    """
```

Возвращает максимальное значение в выборке.

Параметры:

- data (List[float]): Массив значений случайных величин.

Возвращает:

- Xmax (float): Максимальное значение в выборке, или None, если выборка пуста.

```
"""
```

```
if not data:
```

```
    return None
```

```
return max(data)
```

```
def calculate_mean(data: List[float]) -> Optional[float]:
```

```
    """
```

Возвращает выборочное математическое ожидание.

Параметры:

- data (List[float]): Массив значений случайных величин.

Возвращает:

- mean (float): Выборочное математическое ожидание, или None, если выборка пуста.

```
"""
```

```
if not data:
```

```
return None
```

```
return sum(data) / len(data)
```

```
def degrees_of_freedom(data: List[float]) -> int:
```

```
    """
```

Возвращает число степеней свободы выборки.

Параметры:

- data (List[float]): Массив значений случайных величин.

Возвращает:

- df (int): Число степеней свободы.

```
    """
```

```
return len(data) - 1
```

```
def calculate_variance(data: List[float]) -> Optional[float]:
```

```
    """
```


Возвращает выборочную дисперсию.

Параметры:

- data (List[float]): Массив значений случайных величин.

Возвращает:

- variance (float): Выборочная дисперсия, или None, если выборка пуста.

```
"""
```

```
if not data:
```

```
    return None
```

```
mean = calculate_mean(data)
```

```
return sum((x - mean) ** 2 for x in data) / (len(data) - 1)
```

```
def calculate_std_dev(data: List[float]) -> Optional[float]:
```

```
    """
```

Возвращает стандартное отклонение.

Параметры:

- data (List[float]): Массив значений случайных величин.

Возвращает:

- std_dev (float): Стандартное отклонение, или None, если выборка пуста.

"""

if not data:

return None

mean = calculate_mean(data)

variance = calculate_variance(data)

return variance ** 0.5

def calculate_skewness(data: List[float]) -> Optional[float]:

"""

Возвращает коэффициент асимметрии.

Параметры:

- data (List[float]): Массив значений случайных величин.

Возвращает:

- skewness (float): Коэффициент асимметрии, или None, если выборка пуста.

```
"""
```

```
if not data:
```

```
    return None
```

```
mean = calculate_mean(data)
```

```
std_dev = calculate_std_dev(data)
```

```
return sum((x - mean) ** 3 for x in data) / (len(data) * std_dev ** 3)
```

```
def calculate_kurtosis(data: List[float]) -> Optional[float]:
```

```
    """
```

Возвращает коэффициент эксцесса.

Параметры:

- data (List[float]): Массив значений случайных величин.

Возвращает:

- kurtosis (float): Коэффициент эксцесса, или None, если выборка пуста.

```
"""
```

```
if not data:
```

```
    return None
```

```
mean = calculate_mean(data)
```

```
std_dev = calculate_std_dev(data)
```

```
return sum((x - mean) ** 4 for x in data) / (len(data) * std_dev ** 4) - 3
```

```
def calculate_median(data: List[float]) -> Optional[float]:
```

```
    """
```

Возвращает медиану.

Параметры:

- data (List[float]): Массив значений случайных величин.

Возвращает:

- median (float): Медиана, или None, если выборка пуста.

```
    """
```

```
if not data:
```

```
return None
```

```
sorted_data = sorted(data)
```

```
n = len(sorted_data)
```

```
if n % 2 == 0:
```

```
    # Если число элементов четное, медиана - среднее двух центральных  
    элементов
```

```
    middle1 = sorted_data[n // 2 - 1]
```

```
    middle2 = sorted_data[n // 2]
```

```
    return (middle1 + middle2) / 2
```

```
else:
```

```
    # Если число элементов нечетное, медиана - центральный элемент
```

```
    return sorted_data[n // 2]
```

```
def calculate_range(data: List[float]) -> Optional[float]:
```

```
    """
```

```
    Возвращает размах.
```

```
    Параметры:
```

```
    - data (List[float]): Массив значений случайных величин.
```

Возвращает:

- range (float): Размах, или None, если выборка пуста.

"""

if not data:

return None

return max(data) - min(data)

def confidence_interval_mean(data: List[float], confidence_level: float = 0.95) ->
Tuple[float, float]:

"""

Возвращает доверительный интервал для генерального математического ожидания.

Параметры:

- data (List[float]): Массив значений случайных величин.

- confidence_level (float): Уровень доверия (по умолчанию 0.95).

Возвращает:

- confidence_interval (Tuple[float, float]): Доверительный интервал.

"""

if not data:

return None

mean_value = calculate_mean(data)

std_dev_value = calculate_std_dev(data)

sample_size_value = sample_size(data)

if mean_value is None or std_dev_value is None or sample_size_value is None:

return None

Вычисляем стандартную ошибку среднего

standard_error = std_dev_value / (sample_size_value ** 0.5)

Вычисляем критическое значение t-распределения

df = degrees_of_freedom(data)

if df is None:

return None

```
t_value = t.ppf((1 + confidence_level) / 2, df)
```

```
# Вычисляем доверительный интервал
```

```
margin_of_error = t_value * standard_error
```

```
lower_bound = mean_value - margin_of_error
```

```
upper_bound = mean_value + margin_of_error
```

```
return lower_bound, upper_bound
```

```
def confidence_interval_variance(data: List[float], confidence_level: float = 0.95)
```

```
-> Tuple[float, float]:
```

```
"""
```

Возвращает доверительный интервал для генеральной дисперсии.

Параметры:

- data (List[float]): Массив значений случайных величин.

- confidence_level (float): Уровень доверия (по умолчанию 0.95).

Возвращает:

- confidence_interval (Tuple[float, float]): Доверительный интервал.

"""

if not data:

return None

variance_value = calculate_variance(data)

sample_size_value = sample_size(data)

if variance_value is None or sample_size_value is None:

return None

Вычисляем критические значения хи-квадрат распределения

df = degrees_of_freedom(data)

if df is None:

return None

chi2_lower = chi2.ppf((1 - confidence_level) / 2, df)

chi2_upper = chi2.ppf((1 + confidence_level) / 2, df)

```
# Вычисляем доверительный интервал
```

```
lower_bound = (sample_size_value - 1) * variance_value / chi2_upper
```

```
upper_bound = (sample_size_value - 1) * variance_value / chi2_lower
```

```
return lower_bound, upper_bound
```

```
def confidence_interval_skewness(data: List[float], confidence_level: float = 0.95)
```

```
-> Tuple[float, float]:
```

```
"""
```

Возвращает доверительный интервал для генеральной асимметрии.

Параметры:

- data (List[float]): Массив значений случайных величин.

- confidence_level (float): Уровень доверия (по умолчанию 0.95).

Возвращает:

- confidence_interval (Tuple[float, float]): Доверительный интервал.

```
"""
```

```
if not data:
```

```
    return None
```

```
skewness_value = calculate_skewness(data)
```

```
sample_size_value = sample_size(data)
```

```
if skewness_value is None or sample_size_value is None:
```

```
    return None
```

```
# Вычисляем стандартную ошибку асимметрии
```

```
std_error_skewness = (6 * sample_size_value * (sample_size_value - 1) / (  
    (sample_size_value - 2) * (sample_size_value + 1) * (sample_size_value +  
3))) ** 0.5
```

```
# Вычисляем критическое значение t-распределения
```

```
df = degrees_of_freedom(data)
```

```
if df is None:
```

```
    return None
```

```
t_value = t.ppf((1 + confidence_level) / 2, df)
```

```
# Вычисляем доверительный интервал
```

```
lower_bound = skewness_value - t_value * std_error_skewness
```

```
upper_bound = skewness_value + t_value * std_error_skewness
```

```
return lower_bound, upper_bound
```

```
def confidence_interval_kurtosis(data: List[float], confidence_level: float = 0.95)  
-> Tuple[float, float]:
```

```
"""
```

Возвращает доверительный интервал для генерального эксцесса.

Параметры:

- data (List[float]): Массив значений случайных величин.
- confidence_level (float): Уровень доверия (по умолчанию 0.95).

Возвращает:

- confidence_interval (Tuple[float, float]): Доверительный интервал.

```
"""
```

```
if not data:
```

```
    return None
```

```
kurtosis_value = calculate_kurtosis(data)
```

```
sample_size_value = sample_size(data)
```

```
if kurtosis_value is None or sample_size_value is None:
```

```
    return None
```

```
# Вычисляем стандартную ошибку эксцесса
```

```
std_error_kurtosis = (24 * sample_size_value * (sample_size_value - 2) *  
(sample_size_value - 3) / (
```

```
(sample_size_value + 1) * (sample_size_value + 1) * (sample_size_value + 3)  
* (sample_size_value + 5))) ** 0.5
```

```
# Вычисляем критическое значение t-распределения
```

```
df = degrees_of_freedom(data)
```

```
if df is None:
```

```
    return None
```

```
t_value = t.ppf((1 + confidence_level) / 2, df)
```

```
# Вычисляем доверительный интервал
```

```
lower_bound = kurtosis_value - t_value * std_error_kurtosis
```

```
upper_bound = kurtosis_value + t_value * std_error_kurtosis
```

```
return lower_bound, upper_bound
```

```
def skewness_variance(data):
```

```
    n = len(data)
```

```
    return 6 * (n - 1) / (n + 1) / (n + 3)
```

```
def kurtosis_variance(values: List[float]) -> float:
```

```
    """
```

```
    Вычисляет дисперсию эксцесса для массива значений.
```

```
    Параметры:
```

- values (List[float]): Массив значений случайных величин.

Возвращает:

- excess_variance (float): Дисперсия эксцесса.

"""

```
# 24 * n * (n - 2) * (n - 3) / (n + 1)**2 / (n + 3) / (n + 5)
```

```
excess_variance = np.var(
```

```
    values, ddof=1) * np.var(values, ddof=2) / (len(values) * (len(values) - 1))
```

```
return excess_variance
```

reader.py

```
from typing import List
```

```
def read_random_variables(path: str) -> List[float]:
```

"""

Считывает файл с случайными величинами и возвращает массив значений.

Параметры:

- path (str): Путь к файлу.

Возвращает:

- values (List[float]): Массив значений случайных величин.

"""

try:

with open(path, 'r') as file:

values = [float(value) for value in file.read().split()]

return values

except FileNotFoundError:

print(f'Файл по пути {path} не найден.'))

return []

except Exception as e:

print(f'Произошла ошибка при чтении файла: {e}')

return []

approximation.py

import numpy as np


```
def approximate_polynomial(x_values, y_values, degree=3):
```

```
    """
```

Аппроксимирует данные полиномом.

Параметры:

- x_values (np.ndarray): Массив значений x.
- y_values (np.ndarray): Массив значений y.
- degree (int): Степень полинома для аппроксимации (по умолчанию 3).

Возвращает:

- approx_points (list): Список точек аппроксимированной кривой в формате (x, y).

```
    """
```

```
    coefficients = np.polyfit(x_values, y_values, degree)
```

```
    approx_polynomial = np.poly1d(coefficients)
```

```
    # Генерируем точки для построения аппроксимированной кривой
```

```
    x_range = np.linspace(min(x_values), max(x_values), 100)
```

```
    approx_points = list(zip(x_range, approx_polynomial(x_range)))
```

```
return (approx_points, coefficients)
```

```
def approximate_basis_functions(x_values, y_values, degree):
```

```
    """
```

Аппроксимирует исходные данные базисными функциями.

Параметры:

- x_values (np.ndarray): Массив значений x исходной выборки.
- y_values (np.ndarray): Массив значений y исходной выборки.
- degree (int): Степень многочлена для аппроксимации.

Возвращает:

- approx_points (list): Список точек аппроксимированной кривой в формате (x, y).

```
    """
```

```
# Построение матрицы базисных функций
```

```
basis_matrix = np.vander(x_values, degree + 1)
```

```
# Нахождение коэффициентов аппроксимации
```

```
coefficients, _, _, _ = np.linalg.lstsq(basis_matrix, y_values, rcond=None)
```

```
# Генерация точек аппроксимированной кривой для отображения графика
```

```
x_approx = np.linspace(min(x_values), max(x_values), 1000)
```

```
basis_matrix_approx = np.vander(x_approx, degree + 1)
```

```
y_approx = np.dot(basis_matrix_approx, coefficients)
```

```
# Сбор точек в список
```

```
approx_points = list(zip(x_approx, y_approx))
```

```
return approx_points
```

```
def generate_random_data(size: int = 100, noise_factor: float = 0.1):
```

```
    """
```

```
    Генерирует произвольные данные для аппроксимации.
```

```
    Параметры:
```

- size (int): Размер массива данных (по умолчанию 100).
- noise_factor (float): Фактор шума (по умолчанию 0.1).

Возвращает:

- x_values (np.ndarray): Массив значений x.
- y_values (np.ndarray): Массив значений y (кривая с шумом).

"""

```
x_values = np.linspace(0, 1, size)
```

```
true_curve = np.sin(2 * np.pi * x_values)
```

```
noise = noise_factor * np.random.randn(size)
```

```
y_values = true_curve + noise
```

```
return x_values, y_values
```

distribution_law.py

```
from scipy.stats import ks_2samp
```

```
from scipy.stats import weibull_min
```

```
from scipy.stats import uniform
```

```
from scipy.stats import rayleigh
```

```
from scipy.stats import lognorm
```

```
from scipy.stats import expon
```

```
from scipy.stats import gamma
```

```
from typing import List, Callable, Any
```

```
from scipy.stats import shapiro, jarque_bera
```

```
import numpy as np
```

```
from distribution_params import *
```

```
def normality_test_q(data: List[float], alpha: float = 0.05) -> bool:
```

```
    """
```

Проверяет гипотезу о нормальном распределении генеральной совокупности при помощи простейшего теста q.

Параметры:

- data (List[float]): Массив значений случайных величин.
- alpha (float): Уровень значимости (по умолчанию 0.05).

Возвращает:

- result (bool): True, если гипотеза о нормальном распределении не отвергается, иначе False.

"""

```
_, p_value = shapiro(data)
```

```
return p_value > alpha
```

```
def normality_test_jarque_bera(data: List[float], alpha: float = 0.05) -> bool:
```

"""

Проверяет гипотезу о нормальном распределении генеральной совокупности при помощи теста Жарка-Бера.

Параметры:

- data (List[float]): Массив значений случайных величин.

- alpha (float): Уровень значимости (по умолчанию 0.05).

Возвращает:

- result (bool): True, если гипотеза о нормальном распределении не отвергается, иначе False.

"""

```
_, p_value = jarque_bera(data)
```

```
return p_value > alpha
```

```
def calculate_histogram(data: List[float], bins: int = 10) -> Tuple[np.ndarray,  
np.ndarray]:
```

```
    """
```

Рассчитывает гистограмму распределения и возвращает данные для построения.

Параметры:

- data (List[float]): Массив значений случайных величин.

- bins (int): Количество интервалов (по умолчанию 10).

Возвращает:

- histogram_data (Tuple[np.ndarray, np.ndarray]): Кортеж из двух массивов: значения интервалов и соответствующие частоты.

```
    """
```

```
    histogram_values, bin_edges = np.histogram(data, bins=bins, density=False)
```

```
    return bin_edges[:-1], histogram_values
```

```
def fit_gamma_distribution(data: List[float]) -> Tuple[float, float]:
```

```
    """
```

Оценивает параметры гамма-распределения.

Параметры:

- data (List[float]): Массив значений случайных величин.

Возвращает:

- shape (float): Параметр формы (k) гамма-распределения.

- scale (float): Параметр масштаба (θ) гамма-распределения.

```
    """
```

```
    shape, _, scale = gamma.fit(data)
```

```
    return shape, scale
```

```
def fit_exponential_distribution(data: List[float]) -> float:
```

```
    """
```


Оценивает параметр экспоненциального распределения.

Параметры:

- data (List[float]): Массив значений случайных величин.

Возвращает:

- scale (float): Параметр масштаба (λ) экспоненциального распределения.

```
"""
```

```
_, scale = expon.fit(data)
```

```
return scale
```

```
def fit_gamma_distribution(data: List[float]) -> Tuple[float, float]:
```

```
"""
```

Оценивает параметры гамма-распределения.

Параметры:

- data (List[float]): Массив значений случайных величин.

Возвращает:

- shape (float): Параметр формы (k) гамма-распределения.
- scale (float): Параметр масштаба (θ) гамма-распределения.

"""

```
shape, _, scale = gamma.fit(data)
```

```
return shape, scale
```

```
def fit_lognormal_distribution(data: List[float]) -> Tuple[float, float]:
```

"""

Оценивает параметры логнормального распределения.

Параметры:

- data (List[float]): Массив значений случайных величин.

Возвращает:

- s (float): Параметр формы логнормального распределения.
- scale (float): Параметр масштаба логнормального распределения.

"""

```
s, _, scale = lognorm.fit(data)
```

```
return s, scale
```

```
def fit_rayleigh_distribution(data: List[float]) -> float:
```

```
    """
```

Оценивает параметр рэлеевского распределения.

Параметры:

- data (List[float]): Массив значений случайных величин.

Возвращает:

- scale (float): Параметр масштаба рэлеевского распределения.

```
    """
```

```
    _, scale = rayleigh.fit(data)
```

```
    return scale
```

```
def fit_uniform_distribution(data: List[float]) -> Tuple[float, float]:
```

"""

Оценивает параметры равномерного распределения.

Параметры:

- data (List[float]): Массив значений случайных величин.

Возвращает:

- loc (float): Параметр левой границы равномерного распределения.

- scale (float): Разница между правой и левой границами равномерного распределения.

"""

```
loc, scale = uniform.fit(data)
```

```
return loc, scale
```

```
def fit_weibull_distribution(data: List[float]) -> Tuple[float, float]:
```

"""

Оценивает параметры вейбуловского распределения.

Параметры:

- data (List[float]): Массив значений случайных величин.

Возвращает:

- c (float): Параметр формы вейбуловского распределения.

- loc (float): Параметр сдвига вейбуловского распределения.

- scale (float): Параметр масштаба вейбуловского распределения.

"""

```
c, loc, scale = weibull_min.fit(data)
```

```
return c, loc, scale
```

```
def get_distribution_points(data: List[float], theoretical_distribution_func:
Callable[[float], float], params: Tuple[float, ...], bins: int = 100) ->
Tuple[np.ndarray, np.ndarray, np.ndarray]:
```

"""

Возвращает точки для построения графика эмпирической и теоретической (например, PDF) распределений.

Параметры:

- data (List[float]): Массив значений случайных величин.

- `theoretical_distribution_func` (`Callable[[float], float]`): Функция теоретического распределения.

- `params` (`Tuple[float, ...]`): Параметры теоретического распределения.

- `bins` (`int`): Количество интервалов для гистограммы (по умолчанию 100).

Возвращает:

- `empirical_points` (`Tuple[np.ndarray, np.ndarray]`): Кортеж из двух массивов: значения и частоты эмпирического распределения.

- `theoretical_points` (`Tuple[np.ndarray, np.ndarray]`): Кортеж из двух массивов: значения и вероятности теоретического распределения.

"""

Рассчитываем гистограмму для эмпирического распределения

`empirical_values, _ = np.histogram(data, bins=bins, density=True)`

`empirical_points = np.histogram(data, bins=bins)`

Рассчитываем значения теоретического распределения

`theoretical_values = [theoretical_distribution_func(`

`x, *params) for x in empirical_points[1]]`

`return empirical_points[1][:-1], empirical_values, theoretical_values`

Пример использования:

Пусть theoretical_distribution_func - это функция PDF гамма-распределения

gamma_params = fit_gamma_distribution(random_values)

gamma_pdf = lambda x, shape, scale: gamma.pdf(x, shape, scale=scale)

Получаем точки для построения графика

empirical_points, empirical_values, theoretical_values =
get_distribution_points(random_values, gamma_pdf, gamma_params)

```
def kolmogorov_smirnov_test(data: List[float], theoretical_distribution_func:  
Callable[[float, *Tuple[float, ...]], float], params: Tuple[float, ...]) -> Tuple[float,  
float]:
```

```
"""
```

Проверяет непрерывное распределение на соответствие теоретическому с использованием критерия Колмогорова.

Параметры:

- data (List[float]): Массив значений случайных величин.

- `theoretical_distribution_func` (`Callable[[float], float]`): Функция теоретического распределения.

- `params` (`Tuple[float, ...]`): Параметры теоретического распределения.

Возвращает:

- `ks_statistic` (`float`): Значение статистики критерия Колмогорова.

- `p_value` (`float`): p-значение для критерия Колмогорова.

"""

```
theoretical_values = [
```

```
    theoretical_distribution_func(x, *params) for x in data]
```

```
ks_statistic, p_value = ks_2samp(data, theoretical_values)
```

```
return ks_statistic, p_value
```

```
def get_cdf_points(data: List[float], theoretical_distribution_func: Callable[[float,  
*Tuple[Any, ...]], float], params: Tuple[float, ...], bins: int = 100) ->  
Tuple[np.ndarray, np.ndarray, np.ndarray]:
```

"""

Возвращает точки для построения графика эмпирической и теоретической функций распределения (CDF).

Параметры:

- data (List[float]): Массив значений случайных величин.
- theoretical_distribution_func (Callable[[float], float]): Функция теоретического распределения (CDF).
- params (Tuple[float, ...]): Параметры теоретического распределения.
- bins (int): Количество интервалов для гистограммы (по умолчанию 100).

Возвращает:

- empirical_points (Tuple[np.ndarray, np.ndarray]): Кортеж из двух массивов: значения и вероятности эмпирической функции распределения (CDF).
- theoretical_points (Tuple[np.ndarray, np.ndarray]): Кортеж из двух массивов: значения и вероятности теоретической функции распределения (CDF).

"""

Рассчитываем эмпирическую функцию распределения (ECDF)

empirical_values, empirical_points = np.histogram(

data, bins=bins, density=True)

empirical_cumulative = np.cumsum(empirical_values)

Рассчитываем значения теоретической функции распределения

theoretical_values = [theoretical_distribution_func(

```
x, *params) for x in empirical_points]
```

```
theoretical_cumulative = np.cumsum(theoretical_values)
```

```
return empirical_points[:-1], empirical_cumulative, theoretical_cumulative
```

```
main.py
```

```
from matplotlib import pyplot as plt
```

```
import reader as reader
```

```
import distribution_params as dparam
```

```
import distribution_law as dlaw
```

```
import approximation as aprx
```

```
import ploter as ploter
```

```
FILE_PATH = "1.txt"
```

```
rnd_values = reader.read_random_variables(FILE_PATH)
```

```
print("Числовые параметры распределения:")
```

```
print()
```

```
print(f"Sample size = {dparam.sample_size(rnd_values)}")
```

```
print(f'Xmax = {dparam.calculate_Xmax(rnd_values)}')

print(f'Xmin = {dparam.calculate_Xmin(rnd_values)}')

print(f'Mean = {dparam.calculate_mean(rnd_values)}')

print(f'Freedom = {dparam.degrees_of_freedom(rnd_values)}')

print(f'Variance = {dparam.calculate_variance(rnd_values)}')

print(f'RMS = {dparam.calculate_std_dev(rnd_values)}')

print(f'Skewness = {dparam.calculate_skewness(rnd_values)}')

print(f'Kurtosis = {dparam.calculate_kurtosis(rnd_values)}')

print(f'Median = {dparam.calculate_median(rnd_values)}')

print(f'Range = {dparam.calculate_range(rnd_values)}')

print()

print(f'Дисперсия ассимтерии = {dparam.skewness_variance(rnd_values)}')

print(f'Дисперсия эксцесса = {dparam.kurtosis_variance(rnd_values)}')

print(f'Interval mean = {dparam.confidence_interval_mean(rnd_values)}')

print(f'Interval variance = {dparam.confidence_interval_variance(rnd_values)}')

print(f'Interval skewness = {dparam.confidence_interval_skewness(rnd_values)}')

print(f'Interval kurtosis = {dparam.calculate_kurtosis(rnd_values)}')

print()

print()
```

```
print("Закон распределения:")
```

```
ploter.plot_histogram(rnd_values)
```

```
print(f"Простейший тест: {dlaw.normality_test_q(rnd_values)}")
```

```
print(f"Тест Жарка-Бера: {dlaw.normality_test_jarque_bera(rnd_values)}")
```

```
print(f"Гистограмма распределения: {dlaw.calculate_histogram(rnd_values)}")
```

```
ploter.plot_all_distributions(rnd_values)
```

```
ploter.plot_empirical_vs_theoretical_cdf(rnd_values)
```

```
x_data, y_data = aprx.generate_random_data()
```

```
plt.scatter(x_data, y_data, label="Original Data")
```

```
plt.title("Original Data")
```

```
plt.xlabel("X values")
```

```
plt.ylabel("Y values")
```

```
plt.legend()
```

```
plt.show()
```

```
power_coefficients, coeffs = aprx.approximate_polynomial(x_data, y_data, 3)
```

```
ploter.plot_data_and_approximation(x_data, y_data, power_coefficients)
```

```
print()
```

```
print(f'Coefs: {coeffs}')
```

```
approx_points = aprx.approximate_basis_functions(x_data, y_data, 4)
```

```
ploter.plot_data_and_approximation(x_data, y_data, approx_points)
```