ГУАП

КАФЕДРА № 42

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ
ПРЕПОДАВАТЕЛЬ

ст. преподаватель		В. А. Миклуш
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №2

по курсу: Теория информации, данные, знания

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. № 4128		В. А. Воробьев
	подпись, дата	инициалы, фамилия

Санкт-Петербург 2023

Цель работы: Нахождение статистических характеристик для непрерывных величин

Задание

1. Найти коэффициент k в выражении для плотности распределения, построить её график.
2. Найти функцию распределения и построить её график.
3. Вычислить МО, моду, медиану, начальные и центральные моменты до 4-го порядка включительно, дисперсию, СКО, асимметрию и эксцесс.
4. Найти $P(x_1 \leq X \leq x_2)$.

Вариант задания №5

5. $f(x) = k(x+4)$; $x \in [0; 2]$; $x_1 = -1$; $x_2 = 1{,}5$.

Ход работы

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

1. Нахождение коэффициента k и построение графика плотности распределения.

Для нахождения коээфициента k была использована следующая формула: k = 1 / ((b - a) * (b + 1)).

Плотность распределения задана как f(x) = k(x+1), используем условие нормировки плотности распределения на заданном интервале [0; 3].

$$\int_{-\infty}^{+\infty} f_{\xi}(x)dx = 1$$

Результат построения графика плотности распределения изображен на рисунке 1.

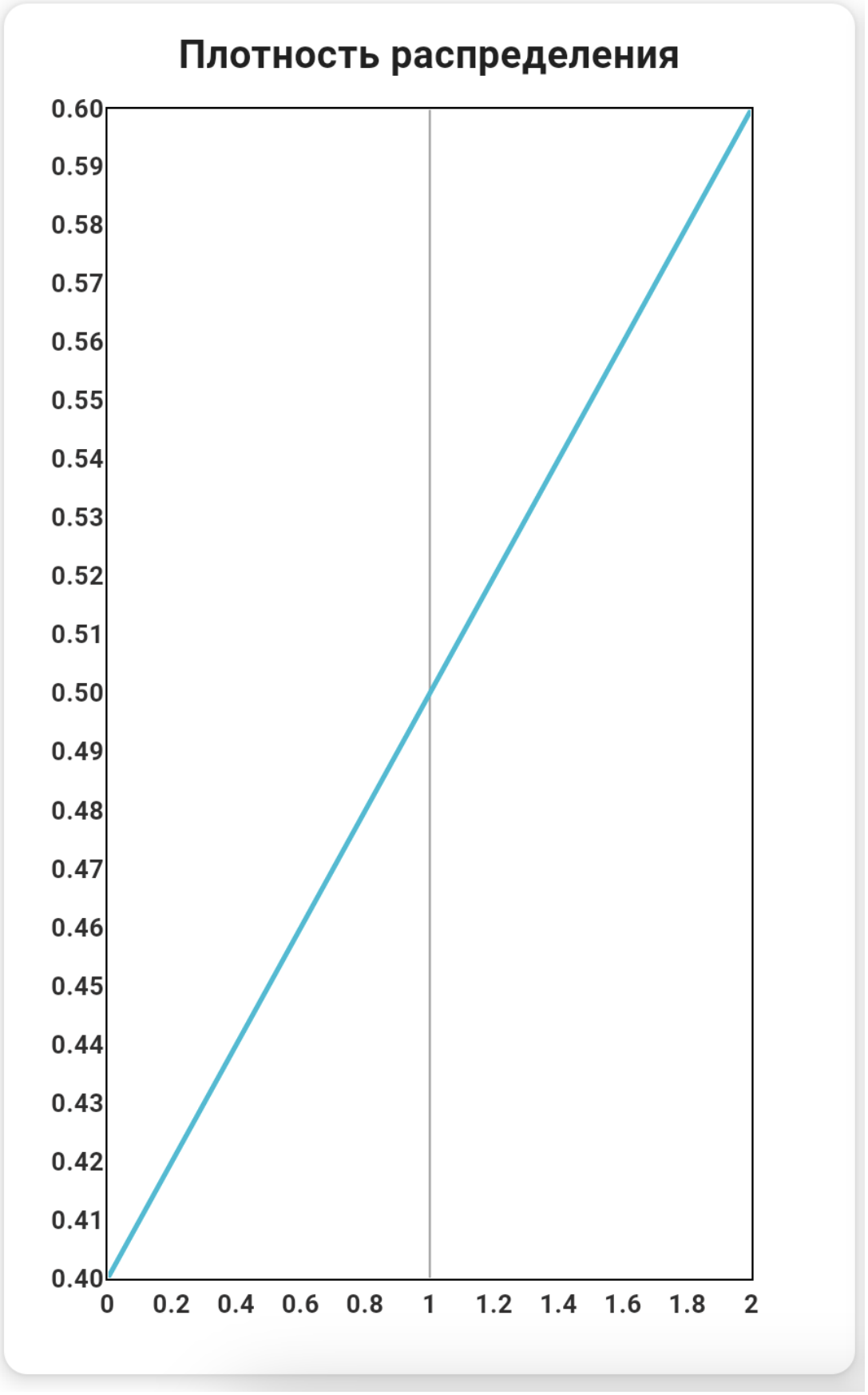

Рисунок 1 – График плотности распределения

2. Построение функции распределения

Находим значения функции распределения с помощью интегрирования функции плотности распределения.

$$F(x) = \int_{-\infty}^{x} f(t)dt$$

Итоговый график функции распределения в написанной нами программе изображен на рисунок 2.

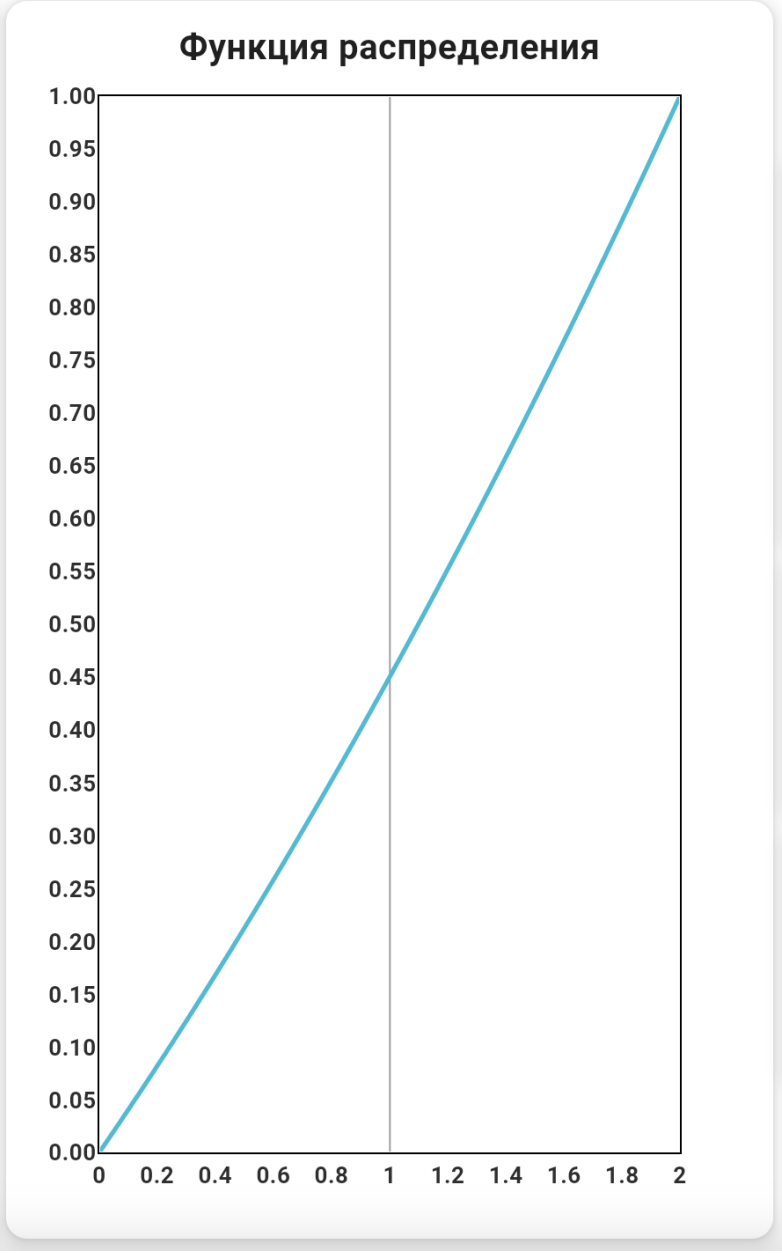

Рисунок 2 - График функции распределения в созданной программе

Программный код, используемый для построения графиков представлен в Приложении.

3. Вычисление статистических характеристик

Математическое ожидание (среднее значение случайной величины)

Мода (для непрерывного распределения: значение x, при котором плотность вероятности $f(x)$ достигает максимума)

Медиана (серединное значение набора чисел)

Дисперсия (отражает меру разброса данных вокруг средней арифметической, является центральным моментом второго порядка)

Среднеквадратичное отклонение (статистическая характеристика распределения случайной величины, показывающая среднюю степень разброса значений величины относительно математического ожидания)

Асимметрия (мера несимметричности распределения случайной величины).

Эксцесс — показатель остроты пика графика распределения.

Начальным моментом k-го порядка случайной величины x называется математическое ожидание k-й степени случайной величины x, т. е. a k = Mx k.

Центральным моментом k-го порядка случайной величины x называется величина m k, определяемая формулой m k = M(x - Mx)k

Вычисленные характеристики представлены на рисунке 3.

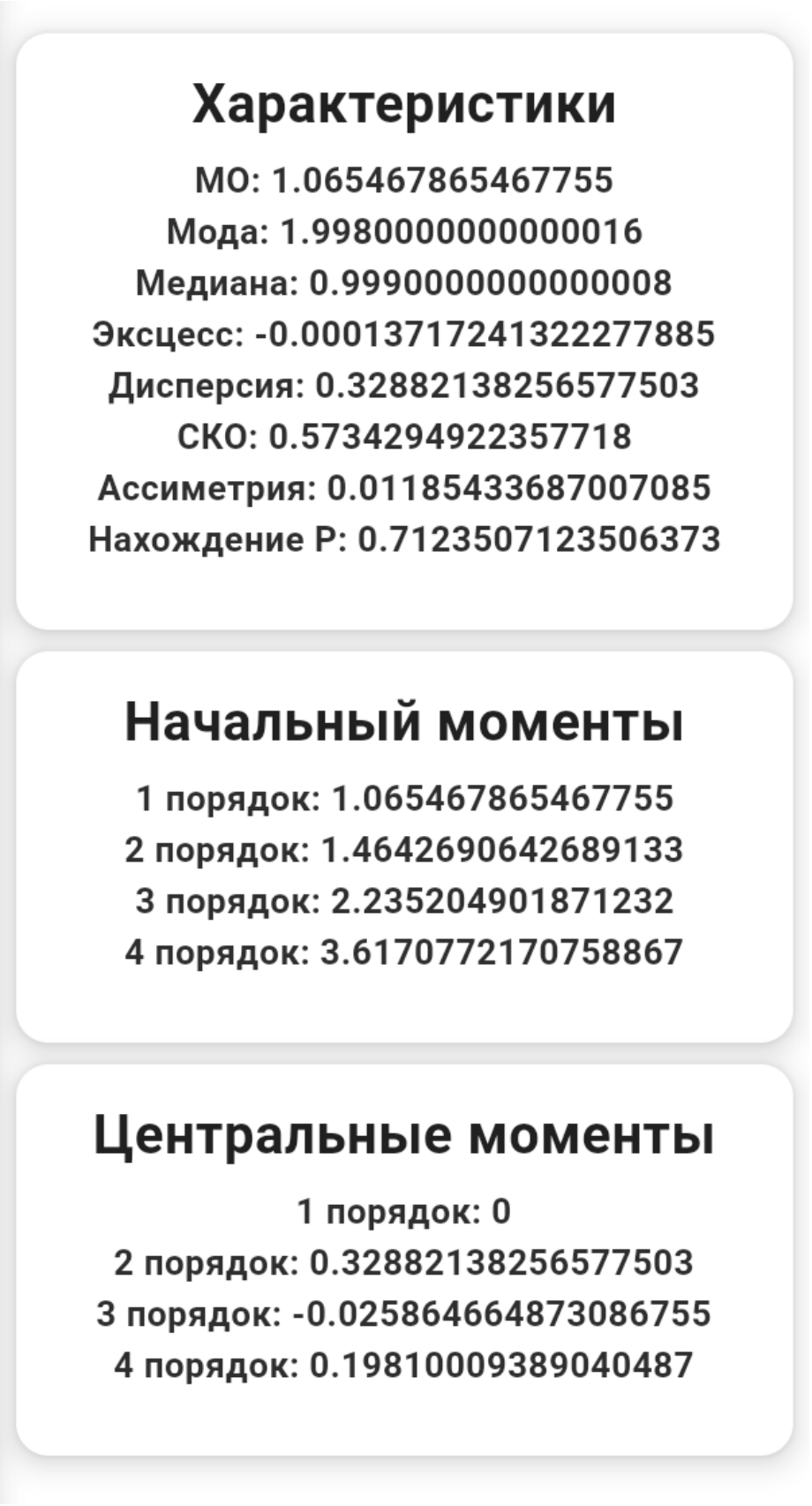

Рисунок 3 - Основные характеристики

4. Нахождение $P(X_1 \leq x \leq X_2)$

$X_1 = -1$, $X_2 = 1.5$ (по условию варианта)

После выполнения командного кода программа вывела соответствующий заданным параметрам результат, который представлен на рисунке 4.

Рисунок 4 - Нахождение $P(X_1 \leq x \leq X_2)$ в созданной программе

Вывод

В ходе выполнения лабораторной работы, на основе исходных данных, соответствующих варианту с использованием языка программирования Dart и фреймворка Flutter, была создана программа, вычисляющая необходимые статистические значения, а также построены графики функции плотности распределения и функции распределения.

ПРИЛОЖЕНИЕ А

ЛИСТИНГ ПРОГРАММЫ

extend_math.dart

```
library extend_math;


export 'src/extension/amplitude_spectrum_ext.dart';

export 'src/extension/distribution_map_ext.dart';

export 'src/extension/double_list_ext.dart';

export 'src/extension/fft_extension.dart';

export 'src/extension/math_interval_ext.dart';

export 'src/extension/point_list_ext.dart';

export 'src/extension/sprectrum_energy_ext.dart';

export 'src/logic/filters_list.dart';

export 'src/logic/math_list.dart';

export 'src/logic/list_helper_functions.dart';

export 'src/models/point2.dart';

export 'src/models/math_interval.dart';
```

distribution_map_ext.dart

```
import 'dart:core';
import 'dart:math';


import '../models/point2.dart';


extension DistributionMapStatistics on Map<double, double> {
  List<Point2> get cumulativeDistribution {
    final listEntries = entries.toList();
    final res = <Point2>[];
    var cumulative = listEntries.first.value;
    res.add(Point2(listEntries.first.key, cumulative));


    for (int i = 1; i < listEntries.length; i++) {
      cumulative = (cumulative + listEntries[i].value);
      res.add(Point2(listEntries[i].key, cumulative));
    }


    return res;
```

```
}

double calcCumulativeProbability(double end, {double start = 0}) {
  double cumulativeProbability = 0.0;

  forEach((key, value) {
    if (start <= key && key <= end) {
      cumulativeProbability += value;
    }
  });

  return cumulativeProbability;
}

double get mean {
  double mean = 0.0;

  forEach((key, value) {
    mean += key * value;
```

```
  });

  return mean;
}

double get secondMoment {
  double secondMoment = 0.0;

  forEach((key, value) {
    secondMoment += (key * key) * value;
  });

  return secondMoment;
}

double get thirdMoment {
  double thirdMoment = 0.0;

  forEach((key, value) {
```

```
    thirdMoment += (key * key * key) * value;
  });

  return thirdMoment;
}

double get fourthMoment {
  double fourthMoment = 0.0;

  forEach((key, value) {
    fourthMoment += (key * key * key * key) * value;
  });

  return fourthMoment;
}

double get mode {
  double mode = entries.first.key;
  double maxProbability = entries.first.value;
```

```
  // Пройдитесь по всем парам ключ-значение в вашей вероятностной карте
  forEach((key, probability) {
    if (probability > maxProbability) {
      mode = key;
      maxProbability = probability;
    }
  });

  return mode;
}

double get median {
  final sortedEntries = entries.toList()
    ..sort((a, b) => a.key.compareTo(b.key));
  final numEntries = sortedEntries.length;

  if (numEntries % 2 == 0) {
    final middle1 = sortedEntries[numEntries ~/ 2 - 1].key;
```

```
    final middle2 = sortedEntries[numEntries ~/ 2].key;
    return (middle1 + middle2) / 2.0;
  } else {
    return sortedEntries[numEntries ~/ 2].key.toDouble();
  }
}

double get excess {
  double mean = 0.0;
  double variance = 0.0;

  forEach((key, value) {
    mean += key * value;
  });

  forEach((key, value) {
    variance += (key - mean) * (key - mean) * value;
  });
```

```
  final stdDev = sqrt(variance);
  final numEntries = length.toDouble();

  double excess = 0.0;
  forEach((key, value) {
    excess += ((key - mean) * (key - mean) * (key - mean) * value) /
        (stdDev * stdDev * stdDev);
  });

  return excess / numEntries;
}

double get variance {
  double mean = 0.0;
  double variance = 0.0;

  forEach((key, value) {
    mean += key * value;
  });
```

```
  forEach((key, value) {
    variance += ((key - mean) * (key - mean)) * value;
  });

  return variance;
}

double get standardDeviation => sqrt(variance);

double get skewness {
  double thirdMoment = this.thirdMoment;
  final stdDev = standardDeviation;

  final numEntries = length.toDouble();
  double skewness = thirdMoment / (stdDev * stdDev * stdDev * numEntries);
  return skewness;
}
```

```
double get centralSecondMoment {
  double centralSecondMoment = 0.0;

  forEach((key, value) {
    centralSecondMoment += ((key - mean) * (key - mean)) * value;
  });

  return centralSecondMoment;
}

double get centralThirdMoment {
  double centralThirdMoment = 0.0;

  forEach((key, value) {
    centralThirdMoment +=
        ((key - mean) * (key - mean) * (key - mean)) * value;
  });

  return centralThirdMoment;
```

```
  }

  double get centralFourthMoment {
    double centralFourthMoment = 0.0;

    forEach((key, value) {
      centralFourthMoment +=
          ((key - mean) * (key - mean) * (key - mean) * (key - mean)) * value;
    });

    return centralFourthMoment;
  }
}
```

sprectrum_energy_ext.dart

```
import 'dart:math';

import 'package:extend_math/extend_math.dart';
```

```
extension SpectrumAmplEnergyExt on List<double> {
  double get energy {
    final total = sum(map((e) => e * e));
    final normalize = map((e) => e * sqrt(0.5 / total));
    return sum(normalize.map((e) => e * e));
  }
}

extension SpectrumPointEnergyExt on List<Point2> {
  double calculateEnergy(MathInterval interval) {
    double integral = 0;
    for (final point in this) {
      integral += pow(point.y, 2);
    }

    final energy = integral / interval.length;
    return energy;
  }
}
```

double_list_ext.dart

```
import 'dart:math';


import '../models/point2.dart';


extension DoubleListExt on List<double> {
  List<double> quantize(int R) {
    double range = reduce(max) - reduce(min);
    double step = range / (2 * R);


    return map((value) {
      int quantizedValue = (value / step).round();
      return quantizedValue * step;
    }).toList();
  }


  List<Point2> joinX(List<double> x) {
    return indexed.map((e) {
```

```
      final (index, y) = e;
      return Point2(x[index], y);
    }).toList();
  }
}
```

math_interval_ext.dart

```
import 'dart:math';

import 'package:extend_math/extend_math.dart';

import '../utils/typedefs.dart';

extension MathIntervalExt on MathInterval {
  List<Point2> applyFx(Func1 fx, {required double step}) {
    return [
      for (var x = start; x <= end; x += step) Point2(x, fx(x))
    ];
  }
```

```
List<double> generateUniformSample(int count) {
  Random random = Random();
  return List.generate(
    count,
    (index) => random.nextDouble() * (end - start) + start,
  );
}

double findCoefficientK(Func1 fx, double step) {
  double integral = 0.0;
  for (double x = start; x <= end; x += step) {
    integral += fx(x) * step;
  }
  return 1.0 / integral;
}

double cumulativeDistributionFunction(Func1 fx) {
  double integral = 0.0;
```

```
    double step = 0.0001;
    for (double x = start; x <= end; x += step) {
      integral += fx(x) * step;
    }
    return integral;
  }
}
```

amplitude_spectrum_ext.dart

```
import 'dart:math';

import '../models/point2.dart';

extension AmplitudeSpectrumExtension on List<Point2> {
  double amplitudeSpectrumFor(
    double freq, {
    required double step,
  }) {
    double realPart = 0.0;
```

```
    double imagPart = 0.0;


    for (int j = 0; j < length; j++) {
      double value = this[j].y;
      double angle = 2 * pi * freq * this[j].x;
      realPart += value * cos(angle) * step;
      imagPart += value * sin(angle) * step;
    }


    return sqrt(realPart * realPart + imagPart * imagPart);
  }
}
```

point_list_ext.dart

```
import 'package:extend_math/extend_math.dart';


extension PointListExt on List<Point2>{
  List<double> get xDots => map((e) => e.x).toList();
  List<double> get yDots => map((e) => e.y).toList();
```

```
}
```

fft_extension.dart

```
// ignore_for_file: prefer_const_constructors


import 'dart:math';


import 'package:complex/complex.dart';


import '../models/point2.dart';


extension DFTExtension on List<Point2> {
  List<Complex> get dft {
    int N = length;
    List<Complex> dftResult = List<Complex>.generate(N, (i) {
      Complex sum = const Complex(0.0, 0.0);
      for (int j = 0; j < N; j++) {
        double angle = 2 * pi * i * j / N;
        Complex c = Complex.polar(this[j].y, angle);
        sum += c;
```

```
      }

      return sum;

    });

    return dftResult;

  }

}


extension InverseDFTExtension on List<Complex> {

  List<Point2> get inverseDft {

    final spectrum = this;

    int N = spectrum.length;

    List<Point2> signal = List<Point2>.generate(N, (i) {

      Complex sum = Complex(0.0, 0.0);

      for (int j = 0; j < N; j++) {

        double angle = -2 * pi * i * j / N;

        Complex c = spectrum[j] * Complex.polar(1.0, angle);

        sum += c;

      }

      return Point2(i.toDouble(), sum.real / N);
```

```
    });
    return signal;
  }
}
```

list_helper_functions.dart

```
double sum(Iterable<double> list) =>
    list.reduce((value, element) => value + element);

List<T> roll<T>(List<T> inputList, int shiftAmount) {
  final length = inputList.length;
  if (length == 0) {
    return inputList;
  }

  final effectiveShift = shiftAmount % length;

  if (effectiveShift == 0) {
    return inputList;
```

```
  }

  final startIndex = effectiveShift < 0 ? -effectiveShift : length - effectiveShift;
  final part1 = inputList.sublist(startIndex);
  final part2 = inputList.sublist(0, startIndex);
  return [...part1, ...part2];
}
```

filters_list.dart

```
abstract final class FiltersList {
  static List<double> medianFilter(List<double> input, int order) {
    final result = List<double>.from(input);

    for (int i = 0; i < input.length; i++) {
      final values = <double>[];

      for (int j = i - order ~/ 2; j <= i + order ~/ 2; j++) {
        if (j >= 0 && j < input.length) {
          values.add(input[j]);
        }
```

```
    }

    values.sort();
    result[i] = values[order ~/ 2];
  }

  return result;
}

static List<double> movingAverageFilter(List<double> input, int order) {
  final result = List<double>.from(input);

  for (int i = 0; i < input.length; i++) {
    double sum = 0.0;
    int count = 0;

    for (int j = i - order ~/ 2; j <= i + order ~/ 2; j++) {
      if (j >= 0 && j < input.length) {
        sum += input[j];
```

```
        count++;
      }
    }

    result[i] = sum / count;
  }

  return result;
 }
}
```

math_list.dart

```
import 'dart:math';

import 'package:extend_math/src/utils/math_ext.dart';

abstract final class MathList {
 static double calculateSNR({
  required List<double> original,
```

```
  required List<double> quantized,
}) {
  double signalPower = 0;
  double noisePower = 0;

  for (int i = 0; i < original.length; i++) {
    signalPower += original[i] * original[i];
    noisePower += (original[i] - quantized[i]) * (original[i] - quantized[i]);
  }

  return 10 * log10(signalPower / noisePower);
}

static List<double> generateNormalDistribution(int size, double a) {
  Random random = Random();
  return List.generate(size, (index) {
    double u1 = 1.0 - random.nextDouble();
    double u2 = random.nextDouble();
    double z = sqrt(-2 * log(u1)) * cos(2 * pi * u2);
```

```
      return z * a;
    });
  }

  static double calculateRMSE(List<double> original, List<double> filtered) {
    print("${original.length}:${filtered.length}");
    final squaredDifferences = List<double>.generate(
      original.length,
      (index) => pow(original[index] - filtered[index], 2).toDouble(),
    );

    final meanSquaredDifference =
        squaredDifferences.reduce((a, b) => a + b) / original.length;

    return sqrt(meanSquaredDifference);
  }
}
```

typedefs.dart

```
typedef Func1 = double Function(double x);
```

math_ext.dart

```
import 'dart:math';

double log10(double x) => log(x) / log(10);
```

math_interval.dart

```
final class MathInterval {
  final double start;
  final double end;

  const MathInterval(this.start, this.end);

  double get length => (end - start).abs();
}
```

point2.dart

```
class Point2 {
  final double x;
```

```
  final double y;

  const Point2(this.x, this.y);

  static const zero = Point2(0, 0);
}
```

preview_app.dart

```
import 'package:extend_math/extend_math.dart';
import 'package:flutter/material.dart';
import 'package:lab2/logic/calculations.dart';
import 'package:lab2/logic/variant.dart';
import 'package:ui_kit/ui_kit.dart';

class PreviewApp extends StatelessWidget {
  const PreviewApp({super.key});

  @override
  Widget build(BuildContext context) {
```

```
final map = Calculations.distributionGraphMap;


return Padding(
  padding: const EdgeInsets.all(8),
  child: Row(
    children: [
      Expanded(
        flex: 3,
        child: KitTitleContainer(
          title: 'Плотность распределения',
          child: KitLineChart(
            lines: [
              KitLineData(
                dots: Calculations.densityGraphPoints,
              ),
            ],
          ),
        ),
      ),
```

```
Expanded(
  flex: 3,
  child: KitTitleContainer(
    title: 'Функция распределения',
    child: KitLineChart(
      lines: [
        KitLineData(dots: Calculations.distributionGraphPoints),
      ],
    ),
  ),
),
Expanded(
  flex: 2,
  child: Column(
    mainAxisAlignment: MainAxisAlignment.center,
    crossAxisAlignment: CrossAxisAlignment.stretch,
    children: [
      KitTitleContainer(
        title: "Характеристики",
```

```
            child: Column(
              mainAxisSize: MainAxisSize.min,
              children: [
                KitText.system("МО: ${map.mean}"),
                KitText.system("Мода: ${map.mode}"),
                KitText.system("Медиана: ${map.median}"),
                KitText.system("Эксцесс: ${map.excess}"),
                KitText.system("Дисперсия: ${map.variance}"),
                KitText.system("СКО: ${map.standardDeviation}"),
                KitText.system("Ассиметрия: ${map.skewness}"),
                KitText.system(
                  "Нахождение P: ${map.calcCumulativeProbability(Variant.x2, start: Variant.x1)}",
                ),
              ],
            ),
          ),
          KitTitleContainer(
            title: "Начальный моменты",
            child: Column(
```

```
    mainAxisSize: MainAxisSize.min,
    children: [
      KitText.system("1 порядок: ${map.mean}"),
      KitText.system("2 порядок: ${map.secondMoment}"),
      KitText.system("3 порядок: ${map.thirdMoment}"),
      KitText.system("4 порядок: ${map.fourthMoment}"),
    ],
  ),
),
KitTitleContainer(
  title: "Центральные моменты",
  child: Column(
    mainAxisSize: MainAxisSize.min,
    children: [
      KitText.system("1 порядок: 0"),
      KitText.system("2 порядок: ${map.centralSecondMoment}"),
      KitText.system("3 порядок: ${map.centralThirdMoment}"),
      KitText.system("4 порядок: ${map.centralFourthMoment}"),
    ],
```

```
                ),
              ),
            ],
          ),
        ),
      ],
    ),
  );
 }
}
```

variant.dart

```
import 'package:extend_math/extend_math.dart';


abstract final class Variant{
  static double fx(double x) => x + 4;
  static const interval =  MathInterval(0, 2);
  static const x1 = -1.0;
  static const x2 = 1.5;
```

```
}
```

calculations.dart

```
import 'package:extend_math/extend_math.dart';
import 'package:lab2/logic/variant.dart';
import 'package:ui_kit/ui_kit.dart';

abstract final class Calculations {
  static const _numPoints = 1000;
  static final _k = Variant.interval.findCoefficientK(Variant.fx, 0.00001);

  static double fx(double x) => _k * Variant.fx(x);

  static final List<KitDot> densityGraphPoints = () {
    final start = Variant.interval.start;
    final end = Variant.interval.end;
    List<KitDot> points = [];
    double step = (end - start) / _numPoints;
    for (double x = start; x <= end; x += step) {
      points.add(
```

```
      KitDot(x, fx(x)),
    );
  }
  return points;
}();

static final List<KitDot> distributionGraphPoints = () {
  final start = Variant.interval.start;
  final end = Variant.interval.end;
  List<KitDot> points = [];
  double step = (end - start) / _numPoints;
  double cumulative = 0.0;
  for (double x = start; x <= end; x += step) {
    cumulative += fx(x) * step;
    points.add(KitDot(x, cumulative));
  }
  return points;
}();
```

```
  static final Map<double, double> distributionGraphMap = () {
    final Map<double, double> res = {};
    var cumulative = 0.0;
    for (var dot in distributionGraphPoints) {
      res[dot.x] = dot.y - cumulative;
      cumulative = dot.y;
    }
    return res;
  }();
}
```

main.dart

```
import 'package:flutter/material.dart';
import 'package:lab2/preview_app.dart';

void main() {
  runApp(const MyApp());
}
```

```
class MyApp extends StatelessWidget {
  const MyApp({super.key});


  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        colorScheme: ColorScheme.fromSeed(seedColor: Colors.orange),
        useMaterial3: true,
      ),
      home: const PreviewApp(),
    );
  }
}
```