ГУАП

КАФЕДРА № 42

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ _____
ПРЕПОДАВАТЕЛЬ

| Ассистент | | Н.А. Янковский |
|---|---|---|
| должность, уч. степень, звание | подпись, дата | инициалы, фамилия |

# ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №2

## Спектр сигнала

Вариант 5

по курсу: Цифровая обработка и передача сигналов

РАБОТУ ВЫПОЛНИЛ

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

Санкт-Петербург 2023

## 1 Задание

Исходные данные:

$f = 3N$, $T = 10/F$, где $N$ – номер по списку.

Написать программу, которая позволит:

1. Вычислить все значения функций $u(t) = \sin(2\pi ft)$ на заданном интервале с шагом $10^{-3}$ и построить график полученной функции.
2. Вычислить амплитудный спектр исследуемой функции и построить график.
3. Вычислить энергию сигнала двумя способами:
    1. По исходной функции
    1. По амплитудному спектру

## 2 Выполнение работы

Построим график функции. Затем для этой функции получим амплитудный спектр и построим график функции. На основе полученных значений вычислим энергию и затем проверим равенство энергий полученных из изначальной функции и амплитудного спектра изначальной функции. Итоговый результат представлен на рис.1

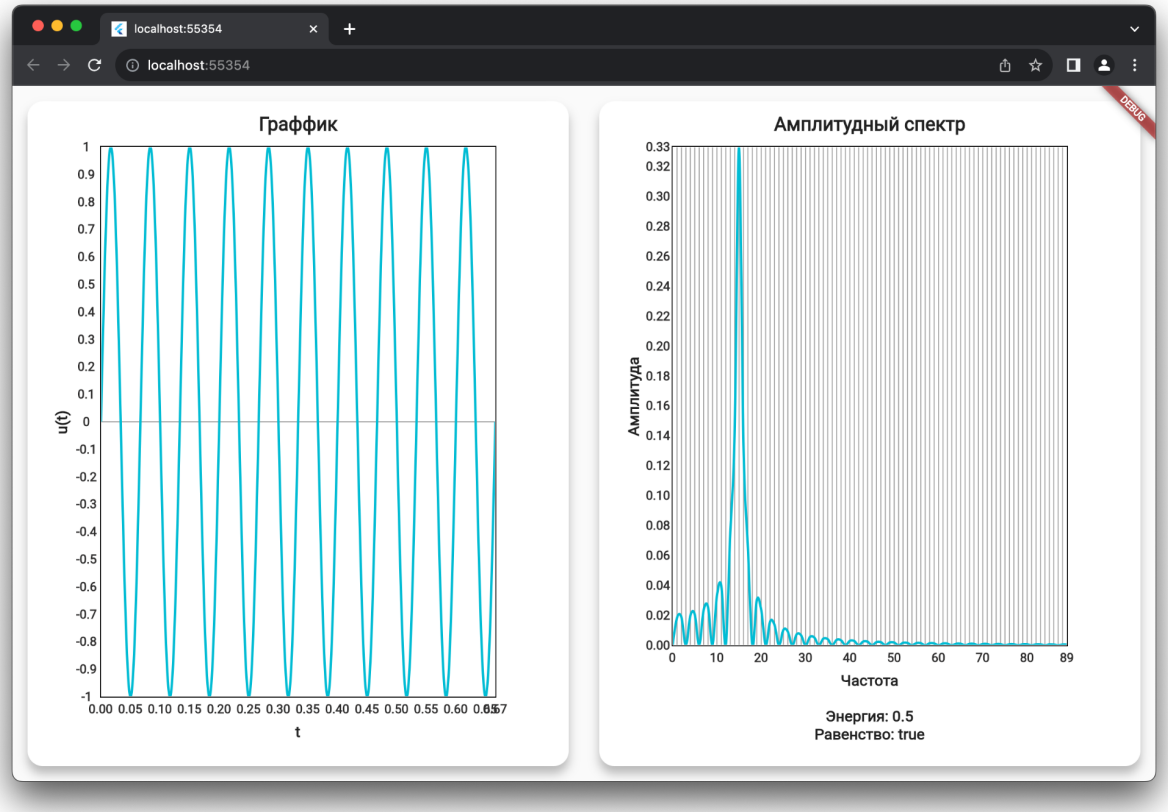


Рисунок 1 – График функций u1

## 3 Вывод

В ходе выполнения лабораторной работы мы приобрели практические навыки вычисления и визуализации математических функций, эти навыки в дальнейшем могут быть полезны в анализе данных, обработке сигналов, а также в других областях, где важно понимание и работа с математическими функциями.

Листинг программы

preview_app.dart

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

class PreviewApp extends StatelessWidget {
  static const _itemsPadding = 16.0;
  const PreviewApp({super.key});

  @override
  Widget build(BuildContext context) {
    var i = 0;
    var spectrum = [
      for (var i = 0.0; i < 90; i += 1)
        MathCalculations.fxPoints.amplitudeSpectrumFor(i, step: Variant.step)
    ];

    return Row(
      children: [
        Expanded(
          child: Padding(
            padding: const EdgeInsets.all(_itemsPadding),
            child: KitTitleContainer(
              title: 'График',
              child: KitLineChart(
                yAxisName: 'u(t)',
```

```
        xAxisName: 't',
        lines: [
          KitLineData(
            dots: MathCalculations.fxPoints
                .map(
                  (e) => KitDot(e.x, e.y),
                )
                .toList(),
          ),
        ],
      ),
    ),
  ),
),
Expanded(
  child: Padding(
    padding: const EdgeInsets.all(_itemsPadding),
    child: KitTitleContainer(
      title: 'Амплитудный спектр',
      child: Column(
        children: [
          Expanded(
            child: KitLineChart(
              yAxisName: 'Амплитуда',
              xAxisName: 'Частота',
              lines: [
                KitLineData(
                  dots: spectrum
                      .map(
                        (e) => KitDot((i++).toDouble(), e),
```

```
                    )
                    .toList(),
              ),
            ],
          ),
        ),
        const SizedBox(height: 16),
        KitText.subtitle("Энергия: ${spectrum.energy}"),
        KitText.subtitle(
          "Равенство: ${MathCalculations.equalEnergy}",
        )
      ],
    ),
  ),
),
),
),
),
],
);
}
}
```

variant.dart

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

abstract final class Variant {
  static const _n = 5;
  static const fn = 3 * _n;
  static const _t = 10 / fn;
```

```
  static double fx(x) => sin(2 * pi * fn * x);
  static const interval = MathInterval(0, _t);
  static const step = 0.001;
}
```

math_calculations.dart

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

abstract final class MathCalculations {
  static final fxPoints = Variant.interval.applyFx(
    Variant.fx,
    step: Variant.step,
  );

  static final spectrum = [
    for (var i = 0.0; i < 90; i += 1)
      MathCalculations.fxPoints.amplitudeSpectrumFor(i, step: Variant.step)
  ];

  static final equalEnergy =
      (fxPoints.calculateEnergy(Variant.interval) - spectrum.energy) < 0.1;
}
```

main.dart

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

void main() {
```

```
  runApp(const MainApp());
}

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

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      home: Scaffold(
        body: PreviewApp(),
      ),
    );
  }
}
```

extend_math.dart

```
library extend_math;
export 'src/extension/amplitude_spectrum_ext.dart';
export 'src/extension/distribution_map_ext.dart';
export 'src/extension/fft_extension.dart';
export 'src/extension/math_interval_ext.dart';
export 'src/extension/sprectrum_energy_ext.dart';
export 'src/logic/list_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>[Point2(listEntries.first.key - 1, 0)];
    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));
    }

    res.add(Point2(listEntries.last.key + 1, 1));
    return res;
  }

  double calcCumulativeProbability(double x0) {
    double cumulativeProbability = 0.0;

    forEach((key, value) {
      if (key <= x0) {
        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 maxProbability = -1.0;
  double mode = 0;

  forEach((key, value) {
    if (value > maxProbability) {
      maxProbability = value;
      mode = key;
    }
  });

  return mode.toDouble();
}

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;
  }
}
```

math_interval_ext.dart

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

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

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

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);
  }
}
```

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_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;
  }

  // Calculate the effective shift amount, wrapping around if necessary
  final effectiveShift = shiftAmount % length;
```

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

  // Split the input list into two parts and rejoin them with the shift
  final startIndex = effectiveShift < 0 ? -effectiveShift : length - effectiveShift;
  final part1 = inputList.sublist(startIndex);
  final part2 = inputList.sublist(0, startIndex);
  return [...part1, ...part2];
}
typedefs.dart
typedef Func1 = double Function(double x);

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);
}
```