|  |  |
| --- | --- |
|  | **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 Теоретическая информатика и компьютерные технологии

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

***К КУРСОВОЙ РАБОТЕ***

***НА ТЕМУ:***

Сравнительный анализ алгоритмов рисования отрезка

Студент ИУ9-52Б \_\_\_\_\_\_\_\_ Гнатенко Т.А.

*группа подпись, дата фамилия, и.о.*

Научный руководитель \_\_\_\_\_\_\_\_ Соборова В.В.

*подпись, дата фамилия, и.о.*

*2023 г*

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 3](#__RefHeading___Toc7683_3522078593)

[1. Обзор предметной области и постановка задачи 4](#__RefHeading___Toc7685_3522078593)

[2. Выбор стека технологий 6](#__RefHeading___Toc7687_3522078593)

[3. Формат входных данных и математическая модель 9](#__RefHeading___Toc7689_3522078593)

[4. Алгоритм получения координат и погрешности объекта 9](#__RefHeading___Toc7691_3522078593)

[4.1 Получение расстояний до объектов 11](#__RefHeading___Toc7693_3522078593)

[4.2 Построение окружностей возможных расположений объектов 12](#__RefHeading___Toc7695_3522078593)

[4.3 Построение окружностей возможных расположений объектов 14](#__RefHeading___Toc7697_3522078593)

[4.4 Нахождение ближайших точек и определение искомых координат с погрешностью 16](#__RefHeading___Toc7699_3522078593)

[5. Визуализация объектов 18](#__RefHeading___Toc7701_3522078593)

[5.1 Визуализация поля 18](#__RefHeading___Toc7703_3522078593)

[5.2 Визуализация антенн-приёмников 19](#__RefHeading___Toc7705_3522078593)

[5.3 Визуализация антенн-передатчиков 21](#__RefHeading___Toc7707_3522078593)

[6. Тестирование 23](#__RefHeading___Toc7709_3522078593)

[6.1 Тестирование визуализации движения по траектории 23](#__RefHeading___Toc7711_3522078593)

[6.2 Тестирование визуализации движения на двух частотах 25](#__RefHeading___Toc7713_3522078593)

[6.3 Тестирование разных положений антенн-приёмников 26](#__RefHeading___Toc7715_3522078593)

[ЗАКЛЮЧЕНИЕ 29](#__RefHeading___Toc7717_3522078593)

[СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ 30](#__RefHeading___Toc7719_3522078593)

[ПРИЛОЖЕНИЕ 30](#__RefHeading___Toc7721_3522078593)

# **ВВЕДЕНИЕ**

Большинство графических устройств являются растровыми. Экран растрового дисплея можно рассматривать как матрицу дискретных элементов, или пикселей. Процесс определения пикселей, наилучшим образом аппроксимирующих некоторую геометрическую фигуру, называется разложением в растр, или построением растрового образа фигуры. Построчная визуализация растрового образа называется растровой разверткой данной фигуры.

Растровую графику применяют при разработке электронных (мультимедийных) и полиграфических изданий. Иллюстрации, выполненные средствами растровой графики, редко создают вручную с помощью компьютерных программ. Для этой цели сканируют иллюстрации, подготовленные художником на бумаге, или фотографии. В последнее время для ввода растровых изображений в компьютер нашли широкое применение цифровые фото и видеокамеры. В Интернете пока применяются только растровые иллюстрации.

Любое изображение, в том числе и трехмерное, состоит из графических примитивов, поэтому необходимо знать специальные методы генерации изображения, вычерчивание прямых и кривых линий, закраски многоугольников, создающей впечатление сплошных объектов. Рассмотрим рисование прямых линий, то есть отрезков.

Растровое представление отрезка – это способ представления геометрического отрезка на растровом (пиксельном) изображении. Растровое изображение состоит из пикселей, которые являются маленькими точками на экране или на печатной поверхности. Каждый пиксель имеет свои координаты и цвет.

Отрезок – это линия, соединяющая две точки в двумерном пространстве. В растровом представлении отрезка, каждый пиксель на линии отрезка окрашивается определенным цветом, чтобы создать впечатление непрерывной линии.

Растровое представление отрезка используется в компьютерной графике для отображения линий, границ объектов, контуров и других геометрических фигур на экране или на печатной поверхности.

Экран дисплея можно рассматривать как матрицу дискретных элементов (пикселей), каждый из которых может быть подсвечен. В связи с этим нельзя непосредственно провести отрезок из одной точки в другую. Процесс определения пикселей, наилучшим образом аппроксимирующих заданный отрезок, называется разложением в растр. Алгоритмы рисования отрезка как раз и нужны для разложения в растр.

В современном мире машинная графика находит самое широкое применение в различных областях науки и техники, промышленности, в экономике, управлении, обучении. В связи с этим существует потребность в анализе алгоритмов построения всевозможных примитивов для того, чтобы понять, какие из них являются наиболее эффективными для той или иной задачи.

В рамках данной курсовой работы была рассмотрена задача анализа алгоритмов построения одного из примитивов, такого как отрезок.

1. **Обзор предметной области и постановка задачи**

В рамках рассматриваемой задачи в качестве входных условий даны начальный дисплей, то есть матрица пикселей, и точки начала и конца отрезка, который нужно разложить в растр.

Стандартными требованиями к алгоритмам являются скорость работы, равномерная яркость и прямой вид полученных отрезков, совпадение начальных и конечных координат полученной и идеальной линии. Для дискретного устройства данные требования зачастую невыполнимы. Отрезок нельзя провести из одной точки в другую однозначно (кроме горизонтальных, вертикальных и наклонённых под углом 45° отрезков), начало и конец отрезка имеют координаты ближайших к ним пикселов, расстояние между пикселами диагональных отрезков больше, чем между пикселами вертикальных и горизонтальных.

Отрезки должны удовлетворять следующим требованиям:

1. Отрезки должны выглядеть прямыми, начинаться и заканчиваться в заданных точках.
2. Яркость вдоль отрезка должна быть постоянной и не зависеть от длины и наклона.
3. Алгоритмы рисования должны быть простыми, то есть отрезок генерироваться должен быстро.

При достаточно высоком разрешении дисплея можно получить приемлемую аппроксимацию прямой линии.

Следует отметить, что постоянная яркость будет у горизонтальных, вертикальных отрезков и под углом 45°, причем у отрезков под углом 45° яркость будет меньше чем у вертикальных и горизонтальных отрезков.

Сложность любого рассматриваемого алгоритма построения отрезка составляет O(n), то есть сложность зависит от количества растеризуемых точек. Поэтому сравнивать алгоритмы следует по времени выполнения и визуальным результатам работы. Эффективность алгоритма определяется как соотношение между временем, затраченным на построение объекта, и результатом, полученным в ходе выполнения программы.

Для определения времени работы алгоритмов необходимо использовать бенчмарки. По своей сути бенчмаркинг - это процесс измерения производительности нашего кода по сравнению с известным стандартом.

Целью работы является: реализация популярных алгоритмов рисования отрезка на языке C++ и последующий сравнительный анализ скорости их работы, качества линии и эффективности.

Для достижения поставленной цели необходимо выполнить следующие задачи:

1. Выбор стека технологий и математической модели задачи.
2. Создание дисплея, способного отображать матрицу дискретных элементов (пикселей) и написание функций основных алгоритмов рисования отрезка.
3. Моделирование отрезков с различным углом наклона и расположением в разных октантах. Реализация отображения списка точек (пикселей), принадлежащих отрезку.
4. Проведение тестов и доработка элементов интерфейса и параметров отображения.
5. Написание бенчмарков и тестов для анализа скорости работы и яркости линий реализованых алгоритмов.

Одним из критериев успешности выполненной работы является визуализация полученных результатов, то есть формирование изображения по матрице пикселей.

У визуализации помимо перевода в понятный человеку формат есть и другие плюсы. Если мы рассматриваем задачу построения отрезка, то визуализация может стать полезной в следующих аспектах:

* Отладка и диагностика: если алгоритм неправильно определяет координаты пикселей или строит неправильную линию, визуальная интерпретируемость позволяет исследовать, какие именно ошибки происходят. Она может помочь понять, как и почему произошла ошибка и как ее исправить.
* Оценка результатов: визуальная интерпретируемость позволяет пользователям визуально оценить результаты определения координат и построения линии. Это может быть полезным для контроля качества и верификации результатов, особенно если задача требует высокой точности.

1. **Выбор стека технологий**

Для реализации приложения был выбран язык программирования Python [4], который соответствует сформулированным целям.

В качестве основной библиотеки для компьютерной графики, была выбрана PyOpenGL [5], реализующая модуль OpenGL [6] для языка Python.

Стандарт OpenGL (Open Graphics Library – открытая графическая библиотека) был разработан и утвержден ведущими фирмами в области разработки программного обеспечения как эффективный программно-независимый интерфейс, пригодный для реализации на различных платформах. Она предоставляет набор функций и процедур, которые позволяют программистам работать с графическим аппаратным обеспечением компьютера для создания реалистичных и высокопроизводительных графических приложений.

Библиотека OpenGL была разработана в 1992 году компанией Silicon Graphics Inc. и быстро стала широко используемой в индустрии компьютерных игр, компьютерной графики, визуализации данных и других областей, требующих высокопроизводительной графики.

Основной принцип работы OpenGL заключается в том, что она предоставляет программисту набор функций для работы с графическим процессором компьютера. Она использует низкоуровневые графические API, такие как DirectX в операционной системе Windows, Metal в macOS и Vulkan для многоплатформенной поддержки. OpenGL предоставляет высокоуровневый интерфейс, который позволяет программисту работать с графическими объектами, текстурами, шейдерами и другими элементами, не заботясь о сложностях, связанных с конкретной платформой или графической аппаратурой.

Основная идея OpenGL состоит в том, чтобы предоставить программисту возможность максимально эффективно использовать аппаратное обеспечение компьютера для выполнения графических операций. OpenGL обладает хорошей производительностью и широкой поддержкой графического аппаратного обеспечения, что позволяет создавать высококачественные и высокопроизводительные графические приложения.

В качестве вспомогательной библиотеки была использована библиотека glfw [7] – мультиплатформенная библиотека для разработки приложений OpenGL.

GLFW (англ. Graphics Library Framework) — это библиотека, предоставляющая простой и удобный интерфейс для работы с окнами, контекстами OpenGL и вводом с клавиатуры, мыши и джойстиков. Она является кроссплатформенной и поддерживает операционные системы Windows, macOS и Linux.

Основная цель GLFW — упростить создание и управление окнами и графическим контекстом OpenGL. Библиотека обеспечивает базовые функции, такие как создание окна, управление его размером и позицией на экране, обработка пользовательского ввода, управление буферами кадров, установка параметров OpenGL контекста и многое другое.

GLFW также предоставляет функции для работы с вводом пользователей. Она позволяет обрабатывать события, связанные с клавиатурой, мышью и джойстиками, что делает ее удобной для создания интерактивных приложений и игр.

Библиотека имеет легкий и интуитивно понятный интерфейс, что делает ее простой в использовании для начинающих разработчиков, но также она предоставляет мощные возможности для более опытных пользователей. GLFW поддерживает различные версии OpenGL (включая новые версии с расширением Vulkan) и может использоваться в комбинации с другими библиотеками, такими как GLEW или GLM.

В целом, GLFW — это надежная и широко используемая библиотека для работы с окнами и контекстами OpenGL. Она предоставляет множество полезных функций и простоту использования, что делает ее отличным выбором для разработчиков, которые хотят быстро создавать графические приложения.

Вспомогательными библиотеками послужили стандартный модуль math для математических операций, функций и констант, а также numpy [8] для быстрых и точных расчётов.

NumPy (Numerical Python) — это библиотека для языка программирования Python, предназначенная для работы с многомерными массивами и матрицами. Она предоставляет удобные и эффективные функции для выполнения операций на массивах, включая математические, логические, статистические и другие операции.

Благодаря своему простому и эффективному интерфейсу, NumPy является популярным инструментом для научных вычислений и представляет собой основу для многих других библиотек и пакетов, таких как SciPy, Pandas и Matplotlib.

1. **Формат входных данных и математическая модель**

Математическая часть выполненной задачи начинается со входных данных.

Вначале, пользователь приложения определяется с размером поля, в масштабах которого будет происходить отрисовка обоих видов антенн. Задаётся размер max\_size – далее гарантируется, что все координаты меньшие заданного значения будут отрисованы корректно и видны в отображенном окне.

При инициализации приложения на вход подаётся список с информацией о принимающих антеннах: каждая описывается тремя значениями – строкой с условным именем антенны для дальнейшей её идентификации, и две координаты по x и y в масштабе заданного поля.

Пример таких данных: ('A1', 30, 40) – инициализирует принимающую антенну с условным именем 'A1' и координатами (30, 40).

Одно из условий задачи – обработка и отображение данных в реальном времени, поэтому данные об излучающих антеннах подаются непосредственно в процессе работы приложения.

Формат таких данных – 4 числа – частота обнаружения антенны в ГГц, и 3 числа – данные от трёх антенн, инициализированных ранее, – мощности сигнала, полученного каждой из антенн на этой частоте. Пример таких данных: (2.8, 45, 60, 53) – данные об излучающей антенне на частоте 2.8 ГГц, далее идут три числа – значение на соответствующей приёмной антенне в дБ, возможно умноженная на некоторый коэффициент для соответствия масштабу, заданному в задаче и измерениям.

Формирование координат антенны-излучателя по эти данным будет описано в соответствующем разделе ниже.

1. **Алгоритм получения координат и погрешности объекта**

Выбор алгоритма для получения координат объекта зависит от ряда факторов, таких как доступные данные, ограничения системы и требования к точности и эффективности. Вот несколько широко используемых алгоритмов для получения координат объекта:

Трилатерация — это метод определения координат объекта путем измерения расстояния или времени прохождения сигнала от объекта до нескольких известных точек (называемых точками измерения), которые могут быть радиоретрансляторами, базовыми станциями или другими точками с известными координатами. Некоторые распространенные алгоритмы трилатерации включают алгоритмы наименьших квадратов (Least Squares), алгоритмы на основе геометрического интервала (Geometric Range) или алгоритмы на основе временного интервала (Time of Arrival).

Триангуляция — это метод, при котором используются углы или направления измерения от нескольких известных точек до объекта для определения его координат. Этот метод также требует, чтобы известны были координаты точек измерения и углы или направления измерения. Примеры алгоритмов триангуляции включают алгоритмы на основе угловой триангуляции (Angle of Arrival) или алгоритмы на основе различия времени прихода сигнала (Time Difference of Arrival).

Калмановская фильтрация — это метод оценки состояния объекта на основе последовательных измерений. Он комбинирует предыдущие измерения и текущие измерения с использованием модели движения и шумов для определения наилучшей оценки состояния объекта. Калмановская фильтрация обычно используется в системах трекинга объектов, где требуется непрерывное определение координат объекта.

Методы машинного обучения: некоторые алгоритмы машинного обучения, такие как нейронные сети, могут быть использованы для обучения модели, которая может предсказывать координаты объекта на основе входных данных. Например, можно использовать нейронную сеть для определения координат объекта на основе силы сигнала или других параметров, полученных от базовых станций или других датчиков.

При трилатерации, взятой за основу в этой работе, принимающие-антенны должны быть размещены на известных местах с известными координатами. Идеальный случай — когда антенны расположены на трех точках, образующих правильный треугольник. Тогда можно определить местоположение третьей точки, используя измерения расстояний от этой точки до каждой из базовых станций.

В нашей задаче подразумевается, что известны координаты всех антенн приёмников, поэтому их положение может быть неидеальным. Однако, в плохих случаях расположения антенн – к таким случаям относятся, например, случай расположения антенн почти на одной прямой – алгоритм отрабатывать будет, однако погрешность вычислений кратно увеличится. Это будет подробнее описано в части тестирования.

**4.1 Получение расстояний до объектов**

В задаче пеленгации, измерение мощности излучения играет важную роль для определения направления источника сигнала. Пеленгация — это процесс определения азимута и угла места источника радиосигнала относительно приемника.

Для измерения мощности излучения в задаче пеленгации используются приемные антенны. Антенна принимает радиосигнал от источника и преобразует его в электрический сигнал, который затем усиливается и обрабатывается для определения мощности излучения.

Существуют различные методы измерения мощности излучения, которые могут быть применены в задаче пеленгации, включая непрерывный и импульсный методы.

В непрерывном методе мощность излучения измеряется непрерывно во время приема сигнала. Этот метод обычно применяется для непрерывных радиосигналов, таких как сигналы FM или AM. Мощность излучения может быть измерена с помощью специальных усилителей и детекторов мощности.

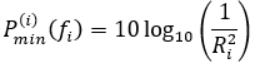
В импульсном методе измерения мощности излучения используется импульсный радиосигнал. Импульсная модуляция имеет короткую длительность и высокую пиковую мощность, что позволяет точнее определить направление источника. Для измерения мощности импульсного сигнала используются быстрые детекторы и специальные приемники.

В обоих случаях результаты измерения мощности излучения используются для определения азимута и угла места источника сигнала. Эти данные обрабатываются с помощью математических алгоритмов и антенных систем, чтобы точно определить положение источника сигнала относительно приемника.

Измерение мощности излучения является важным шагом в процессе пеленгации, и точность этого измерения влияет на точность определения направления источника сигнала.

Полученные входные данные в виде мощностей излучения, зафиксированных каждой из трех антенн, не интерпретируемы в таком виде для поиска расстояния от каждой антенны до объекта.

Для преобразования мощности (в дБ) в расстояние (в метрах) была использована следующая формула:

,

где P – входная мощность, R – искомое расстояние от антенны-приемника, зафиксировавшей эту мощность, до антенны-излучателя.

По этой формуле явно видно, что чем выше мощность P, тем меньше должно быть расстояние до объекта. Этим фактом будем пользоваться в испытаниях.

**4.2** **Построение окружностей возможных расположений объектов**

Теперь, когда получены предполагаемые расстояния от каждой антенны-приёмника до антенны-излучателя, можно построить окружности вокруг каждой из точек, которые будут отражать область нахождения объекта относительно этой антенны.

Для моделирования задачи пусть имеем антенны-приёмники с координатами (1, 2), (4, 6) и (6, 2) по x и y соответственно*.*

Тогда для ситуации, когда вычисленные по мощностям расстояния равны 2, √10, √11 соответственно, имеем такой рисунок:

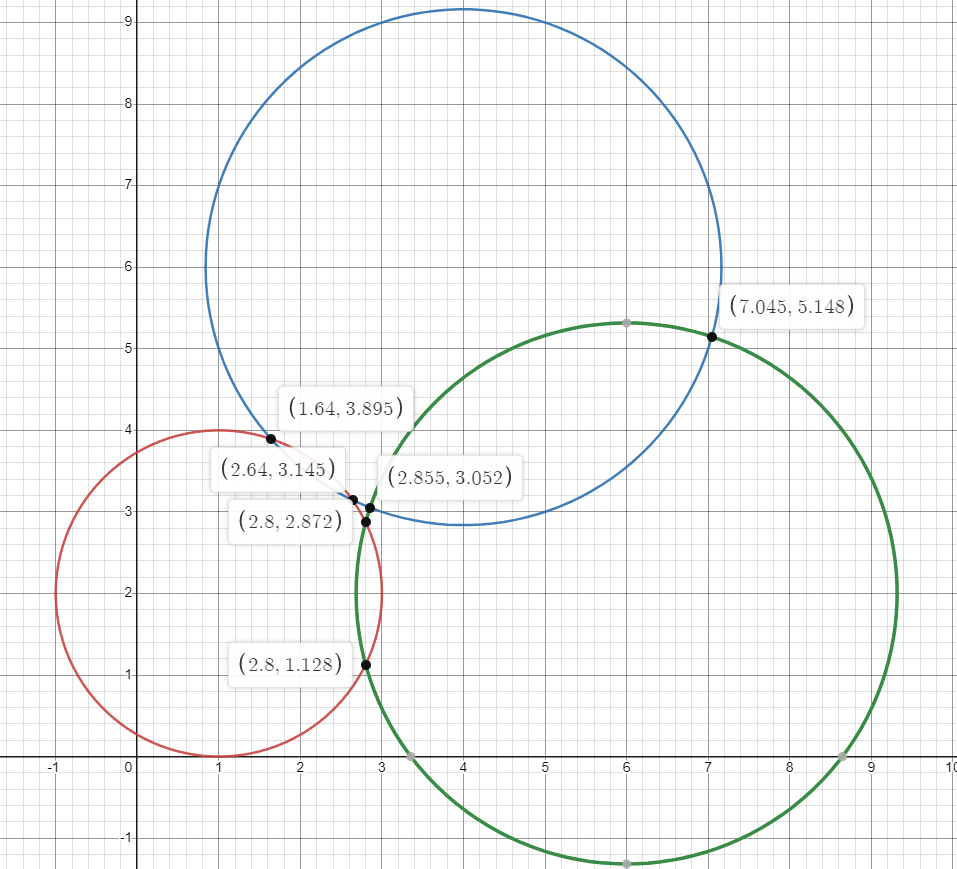


Рисунок 1 – Модель с окружностями с 6 точками пересечения

В этом случае отчетливо видно 6 точек пересечения – больше их, очевидно, быть не может.

Точки пересечения определяются из попарного приравнивания каждого из уравнений окружностей друг к другу и нахождения корней. Приведём код функции, которая реализует данный алгоритм:

Листинг 1 – Поиск точек пересечения двух окружностей

**def** **get\_intersections**(x0, y0, r0, x1, y1, r1):

d = math.sqrt((x1 - x0) \*\* **2** + (y1 - y0) \*\* **2**)

**if** d > r0 + r1:

**return** ()

**if** d < abs(r0 - r1):

**return** ()

**if** d == **0** **and** r0 == r1:

**return** ()

**else**:

a = (r0 \*\* **2** - r1 \*\* **2** + d \*\* **2**) / (**2** \* d)

h = math.sqrt(r0 \*\* **2** - a \*\* **2**)

x2 = x0 + a \* (x1 - x0) / d

y2 = y0 + a \* (y1 - y0) / d

x3 = x2 + h \* (y1 - y0) / d

y3 = y2 - h \* (x1 - x0) / d

x4 = x2 - h \* (y1 - y0) / d

y4 = y2 + h \* (x1 - x0) / d

**return** (round(x3, **2**), round(y3, **2**)), (round(x4, **2**), round(y4, **2**))

**4.3 Построение окружностей возможных расположений объектов**

Приведенное выше положение трех окружностей относительно друг друга не единственно.

Точек пересечения между трех окружностей может быть от 0 до 6. Если окружности касаются – эта точка и будет характеристикой наибольшей вероятности нахождения объекта относительно этих двух окружностей и никакие иные действия не требуются.

Однако, если окружности не пересекаются, необходимо найти точку, равноудаленную от них обоих. Смоделируем такую ситуацию, уменьшив радиус второй окружности до √5*,* а третьей – до √3*.* Тогда имеем такой рисунок:

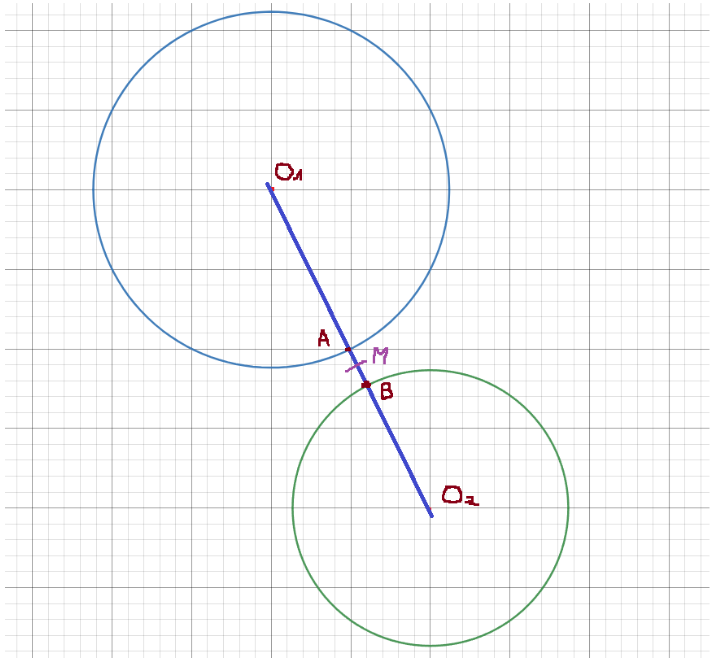


Рисунок 2 – Модель непересекающихся окружностей

Для нахождения точки М, проведем отрезок между центрами окружностей О1 и О2 и отметим точки пересечения с окружностями – A и B. Середина отрезка AB и будет искомой точкой M, равноудаленной от обоих окружностей.

В рамках программы, достаточно “отступить” от каждого цента по направлению к другому на расстояние радиуса. Реализовать это можно с помощью косинуса для x и синуса для y. Середина отрезка же находится как среднее арифметическое координат. Приведем код этой функции:

Листинг 2 – Нахождение ближайшей точки для двух не пересекающихся окружностей

**def** **find\_closest\_point**(x1, y1, r1, x2, y2, r2):

distance = math.sqrt((x2 - x1) \*\* **2** + (y2 - y1) \*\* **2**)

proportion1 = r1 / distance

closest\_point1\_x = x2 + (x1 - x2) \* proportion1

closest\_point1\_y = y2 + (y1 - y2) \* proportion1

proportion2 = r2 / distance

closest\_point2\_x = x1 + (x2 - x1) \* proportion2

closest\_point2\_y = y1 + (y2 - y1) \* proportion2

**return** round((closest\_point1\_x + closest\_point2\_x) / **2**, **2**), round((closest\_point1\_y + closest\_point2\_y) / **2**, **2**)

Итак, в результате применения этих двух алгоритмов, мы имеем набор точек – от 3 до 6 – характеризующих наиболее вероятное положение антенны-излучателя согласно посчитанным расстояниям.

**4.4 Нахождение ближайших точек и определение искомых координат с погрешностью**

В рамках следующего шага алгоритма, найдем ближайшие 3 точки из полученных. Было решено использовать самый простой метод полного перебора, так как максимальное количество точек равно 6 и асимптотика O(n3) не помешает этому методу найти эти точки почти мгновенно.

Покажем, что такой метод правда помогает найти самые интересные для нас точки. Для приведённого в начале раздела примера, имеем следующие точки на выходе этой части алгоритма:

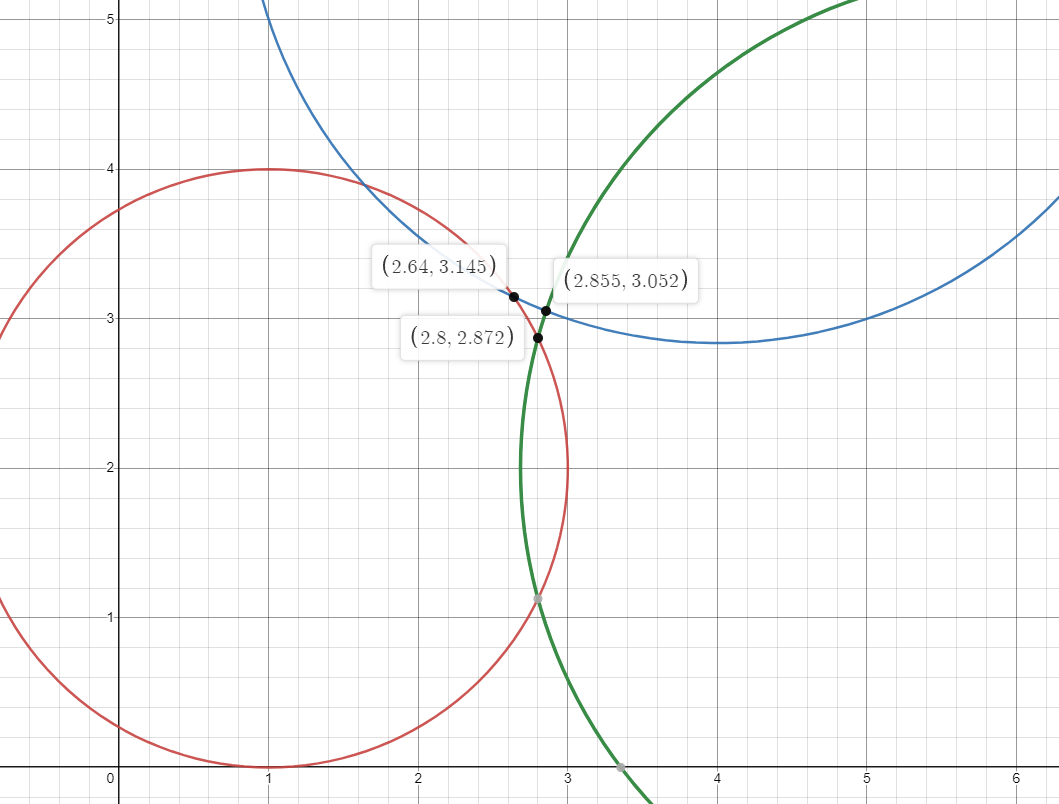


Рисунок 3 – Положение трёх ближайших точек

В ситуации, когда некоторые окружности не имеют общих точек, алгоритм также отработает корректно:

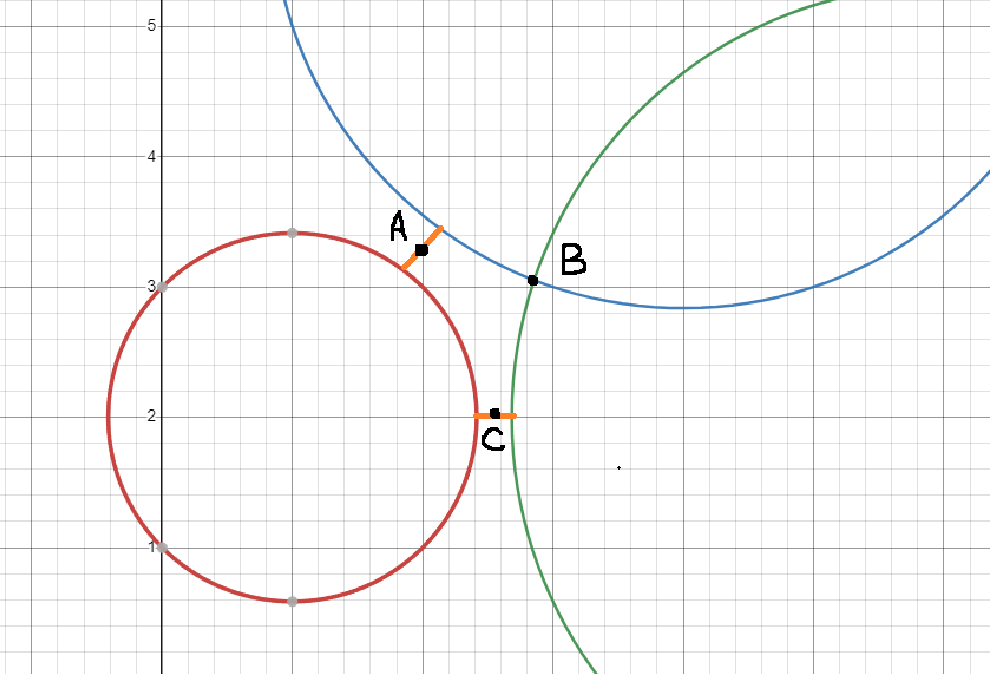


Рисунок 4 – Определение точек при непересекающихся окружностях

Найденные текущим шагом алгоритма точки – это точки A, B, C.

Теперь имеем треугольник, построенный на этих трех точках. Тогда искомая точка – центр этого треугольника. Из курса аналитической геометрии известно, что эта точка лежит в точке пересечения медиан. Точку пересечения медиан найдем, отступив от вершины треугольника в сторону середины противоположной стороны на 1/3 длины медианы.

Ошибкой же определения координат будем считать среднее расстояние от этой точки до вершин треугольника. Обоснование этому достаточно простое – найденные точки-вершины треугольника являются точными характеристиками полученных расстояний от каждой антенны-приёмника, поэтому ошибку стоит считать именно как отдаленность центра треугольника от них.

В завершении приведём пример такого треугольника с рисунка 1, где М – искомая точка:

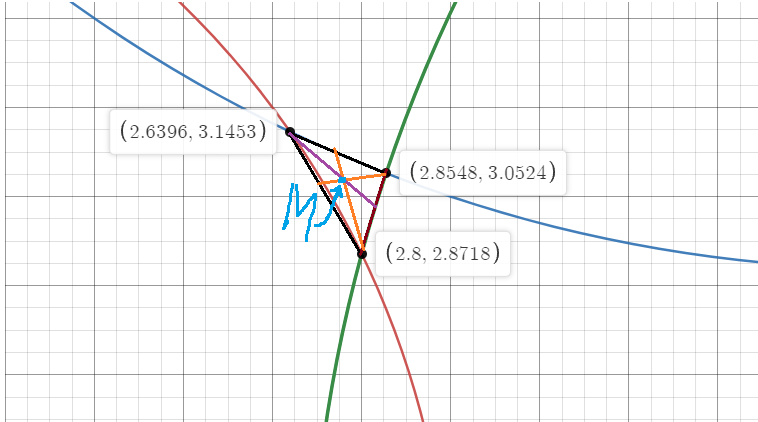


Рисунок 3 – Нахождение итоговой точки

1. **Визуализация объектов**

**5.1 Визуализация поля**

Размер поля, как было упомянуто в формате входных данных, передаётся при запуске приложения. На двух осях системы координат отмечается 10 меток со значением координаты в этих метках.

Например, для поля размером 100:

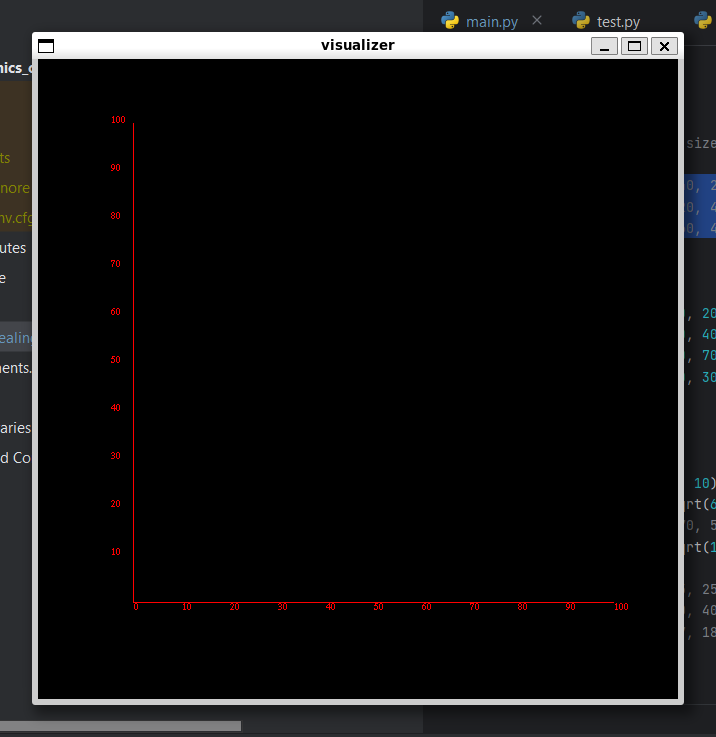


Рисунок 4 – Визуализация поля размером 100х100

**5.2 Визуализация антенн-приёмников**

Данные о расположении и условных именах антенн-приёмников известны также в момент запуска приложения. Координаты точек визуально должны совпадать с созданной в предыдущем пункте разметкой поля, которая, по сути, и есть система координат для данной задачи.

Для достижения соответствия, все точки программы корректируются с использованием формулы:

x / max\_size · 1.5 + corr,

y / max\_size · 1.5 + corr,

где max\_size – размер поля, corr – фиксированная корректировка сдвига относительно окна, точка начала координат в котором находится в левом верхнем углу, что не всегда удобно. Точки принимающих антенн большого размера для повышения их заметности.

Для выводы текста с информацией об антенне, был использован подмодуль glut библиотеки PyOpenGL, который содержит для этого удобные методы: glRusterPos2d() для задания 2d-координат текста а также glutBitmapCharacter(), который принимает в себя шрифт – был использован GLUT\_BITMAP\_TIMES\_ROMAN\_10, а также объекты для отрисовки в символьном формате char. Приведём часть функции, отвечающей за отображение текста:

Листинг 3 – Визуализация текста

glColor3f(**1.0**, **1.0**, **0.0**)

text = "(" + str(last\_freq) + ")" + " x: " + str(last\_pos\_x) + " y: " + str(last\_pos\_y)

glRasterPos2d((last\_pos\_x - **9**) / MAX\_CORR \* **1.5** + corr, (last\_pos\_y + **3**) / MAX\_CORR \* **1.5** + corr)

**for** char **in** text:

glutBitmapCharacter(GLUT\_BITMAP\_TIMES\_ROMAN\_10, ord(char))

Цвет текста задаётся стандартным методом glColor3f() – это метод меняет текущие значения цвета по 3 параметрам насыщенности – красной, зелёной и синей составляющей RGB палитры.

Метод glutBitmapCharacter()принимает на вход один символ и шрифт, поэтому отрисовка всех указанных в text данных помещена в цикл for.

Как было указано в описании алгоритма нахождения положения передающей антенны, антенны-приёмники должны занимать положение в вершинах треугольника с наименьшей разницей в величине углов.

В этом и следующих экспериментах, антенны располагаются руководствуясь этой особенностью.

Теперь перейдем к примеру визуализации антенн-приёмников.

Например, для трёх антенн-приёмников, заданных следующими входными данными: ('A0', 40, 20), ('A1', 20, 40), ('A2', 60, 40)

Получаем такое отображение на поле размера 100:

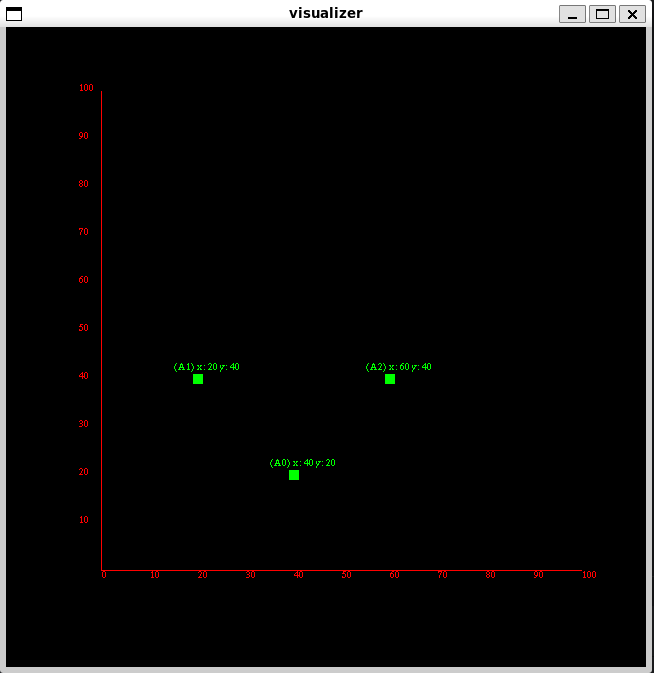


Рисунок 6 – Визуализация антенн-приёмников на поле 100х100

Также приводится текст отрисовки антенны-приёмника:

Листинг 4 – Визуализация антенны-приёмника

**def** **draw\_receiver**(name: str, x: int, y: int, max\_corr):

glColor3f(**0**, **1.0**, **0.0**)

text = "(" + name + ")" + " x: " + str(x) + " y: " + str(y)

glRasterPos2d((x - **5**) / max\_corr \* **1.5** + corr, (y + **2**) / max\_corr \* **1.5** + corr)

**for** char **in** text:

glutBitmapCharacter(GLUT\_BITMAP\_TIMES\_ROMAN\_10, ord(char))

glPointSize(**10**)

glBegin(GL\_POINTS)

glVertex3f(x / max\_corr \* **1.5** + corr, y / max\_corr \* **1.5** + corr, **0**)

glEnd()

Здесь мы видим метод отображения текста, описанный ранее и метод разметки вершин glVertex3f() с корректировкой координат, описанной в п. 5.2.

**5.3 Визуализация антенн-передатчиков**

Отрисовка антенн-передатчиков отличается от антенн-приёмников тем, что данные по ним приходят периодически. Данные об уже полученных данных о зафиксированных мощностях на трех антеннах и частоте (см. формат входных данных) хранятся в массиве.

Однако, просто отрисовать и соединить их линиями не получится – необходима группировка по принятым частотам. На каждой частоте в рамках данной задачи может быть не более одного передатчика.

Полученный массив сразу после обновления новой информацией преобразуется в словарь, где ключ – частота принятого сигнала от излучателя, а ключи – полученные тройки мощностей.

Далее, алгоритм, описанный в части 4 сначала получает расстояния от антенн-приёмников, а потом по ним находит координаты искомой точки и погрешность измерений.

Например, для точки, описанной входными данными как

[(5.8, 55, 55, 55), (5.8, 50, 70, 50), (5.8, 47, 48, 50)],

получаем такую картинку:

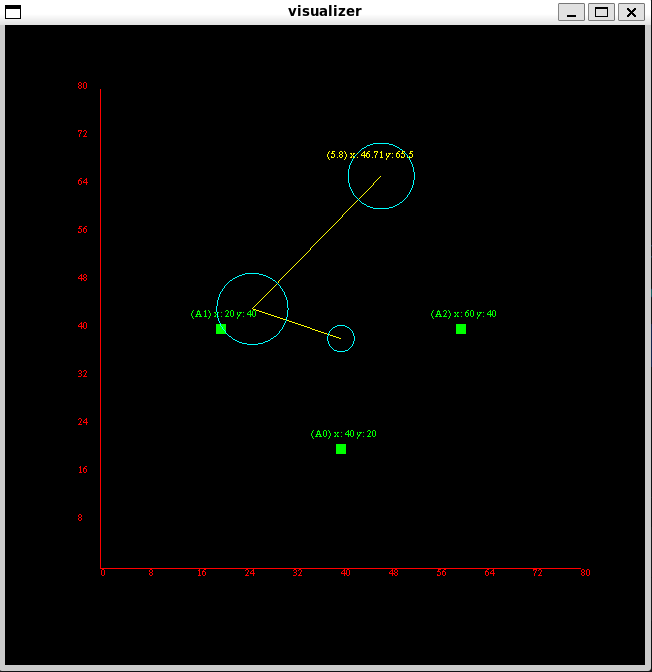


Рисунок 7 – Визуализация трёх полученных положений антенны-передатчика

Круги около точек отражают погрешность, т.е. точка с некоторой вероятностью может находится в этом круге, а не только в его центре.

Отметим, что для этого случая выбран масштаб поля с max\_size = 80. При таком масштабе объекты видны лучше, и при этом они не выходят за пределы окна.

На последнем этапе разработки программы, было добавлено динамическое изменение масштаба поля, что позволило подбирать нужный масштаб без необходимости перезапускать приложение.

Погрешность посчитана всё тем же алгоритмом из части 4 и отрисована в виде голубых окружностей.

Визуализацию всех трёх элементов задачи можно считать удачной, так как картинка соответствует заданному масштабу и является информативной.

1. **Тестирование**

Тестирование является важным этапом при разработке и применении алгоритмов определения координат объекта. Вот несколько причин, почему тестирование необходимо для такой задачи:

Проверка корректности алгоритма: Тестирование позволяет убедиться, что реализованный алгоритм работает правильно и дает правильные результаты для известных входных данных. Это помогает идентифицировать ошибки и проблемы в алгоритме и его реализации.

Оценка точности и надежности: Тестирование позволяет оценить точность и надежность алгоритма. Это особенно важно, поскольку точность может быть ограничена различными факторами, такими как шум, наличие множественных путей сигнала или недостаточное количество базовых станций. Тестирование на разных входных данных и в различных условиях позволяет определить ожидаемую точность и надежность алгоритма.

Проверка на выход за пределы: Тестирование помогает определить граничные случаи и ситуации, когда алгоритм может не работать должным образом или давать неверные результаты. Например, это может быть связано с недостаточным количеством базовых станций или ситуациями, когда целевая точка находится слишком близко к одной из базовых станций. Тестирование на таких случаях помогает обнаружить и улучшить такие проблемы.

В данном разделе приведу наиболее интересные по моему мнению примеры входных данных и какой результат дало приложение, действуя с ними.

**6.1 Тестирование визуализации движения по траектории**

Первый пример – движение одной антенны-излучателя сквозь принимающие антенны с уходом наверх.

Для такого случая входные данные будут такими:

[(5.8, 55, 55, 47), (5.8, 70, 70, 49.8), (5.8, 55, 55, 55), (5.8, 27, 38, 50)]

Здесь мы видим серию данный от антенн-приёмников, полученных ими на одной частоте. Данные очень разнообразны и моделируют ситуации различных входных данных в алгоритм расчёта координат – окружности здесь пересекаются, не имеют точек соприкосновения, вложены друг в друга.

И выход приложения:

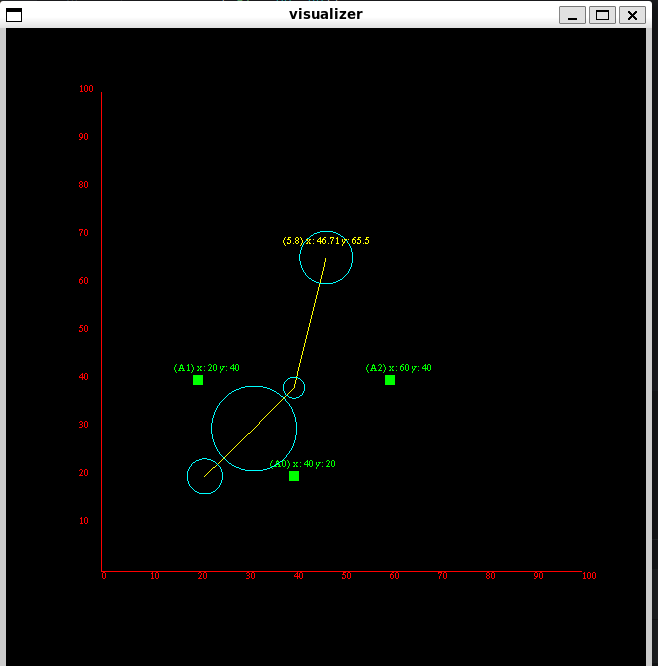


Рисунок 8 – Визуализация результатов первого эксперимента

Здесь мы видим полное соответствие входным данным – вначале антенна-источник была равноудалена от приемников А1 и А0 и приближалась к приёмнику А3 – это и отражено на рисунке. Далее источник существенно удалился от всех антенн-приёмников, самая большая мощность зафиксирована на антенне-приёмнике А2, и по результату отображения видно, что он ближе всех.

Точность полученных координат также достаточно высокая, и сильный скачек наблюдается лишь при втором измерении, где на приёмные антенны A0 и A1 намеренно передана большая мощность.

**6.2 Тестирование визуализации движения на двух частотах**

Теперь второй эксперимент – на вход программе будут поступать данные на двух частотах.

Новые поступающие данные пусть выглядят следующим образом:

[(5.8, 55, 55, 47), (5.8, 70, 70, 49.8), (2.4, 55, 47, 55), (5.8, 55, 55, 55), (2.4, 49, 50, 65), (5.8, 47, 48, 50), (2.4, 48, 49, 55)]

Заметим, что данные о положении на частоте 2.4 ГГц и 5.8 ГГц поступают вперемешку. С использованием словаря, программа классифицирует полученные данные на данные по частоте 2.4 ГГц и частоте 5.8 ГГц, и только потом подает их на вывод функции отрисовки траектории.

На рисунке 7 ниже видим две независимые траектории антенн-излучателей. Одна из них – из предыдущего эксперимента, помечена частотой 5.8 ГГц, а вторая – осуществляет движение с правой нижней части поля и помечена частотой 2.4 ГГц.

Данные по антенне на частоте 2.4 ГГц практически симметричны первой, при этом по интервалам заметна более быстрая скорость передвижения и то, что данные антенны-излучатели перестали отслеживаться примерно в одной точке. При этом скачки в погрешности опять же связаны с аналогичными первому случаю причинами, а именно намеренная передача неточных данных для тестирования возможностей алгоритма определения координат в таких условиях.

Привожу вывод приложения для этих входных данных:

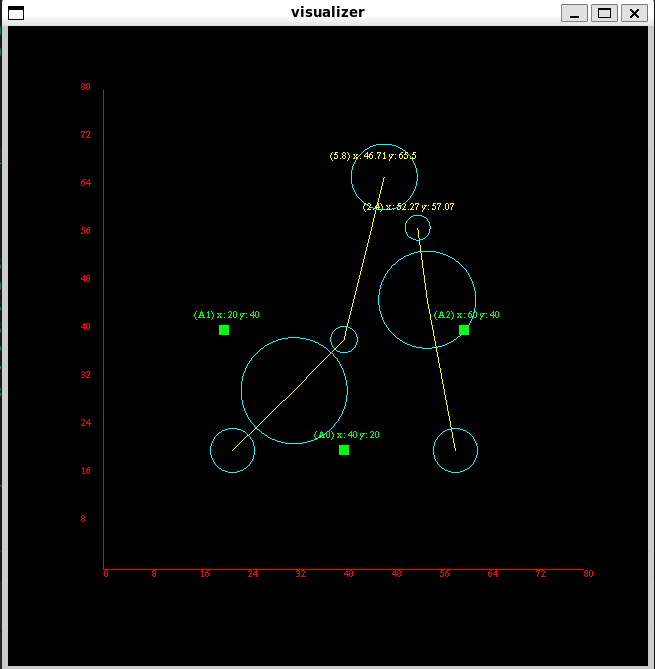


Рисунок 9 – Визуализация результатов второго эксперимента

* 1. **Тестирование разных положений антенн-приёмников**

Как было упомянуто в части описания алгоритма определения координат, положение принимающих антенн сильно влияет на точность вычислений. В предыдущих испытаниях, антенны располагались в положении, близком к идеальному: в вершинах равнобедренного прямоугольного треугольника. Теперь попробуем расположить принимающие антенны практически на одной прямой: теперь их координаты (20, 60), (40, 30) и (60, 15). Полученные мощности оставим такими же, как в первом испытании.

Получаем такой выход приложения:

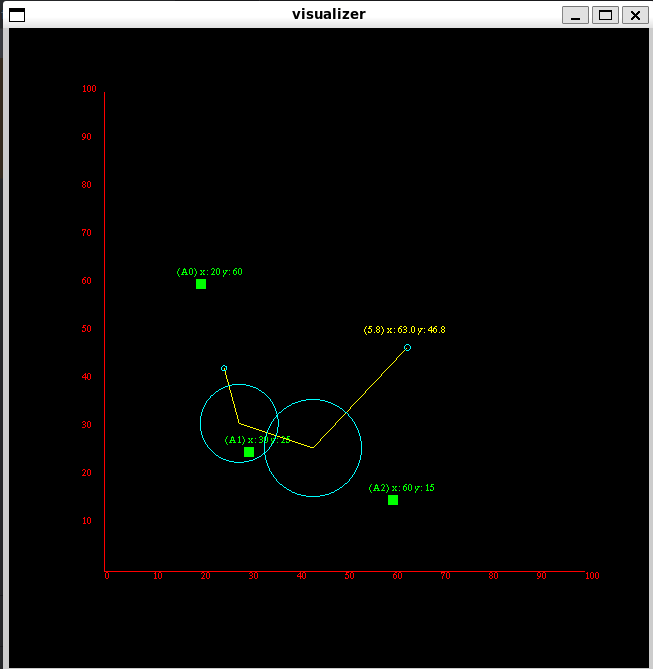


Рисунок 10 – Визуализация результатов третьего эксперимента

Видим высокую точность измерений на первом и последнем шаге траектории, и очень большую погрешность (порядка 10 метров) на втором и третьем. Это связано с тем, что на втором и третьем шаге, передающая антенна оказалась слишком близко к принимающей антенне A1, и так как A1 находится почти на одной прямой с двумя другими антеннами, скорректировать положение передающей антенны по принятым данным не получилось без большой погрешности.

В случае же с первым и вторым появлением антенны-передатчика, её положение относительно принимающих антенн было относительно хорошее, поэтому на итоговом отображении имеем высокую точность.

Теперь рассмотрим самый плохой случай расположения антенн-приёмников – на одной прямой: для этого расположим их на координатах (20, 60), (40, 40), (60, 20).

В таком случае имеем следующий выход:

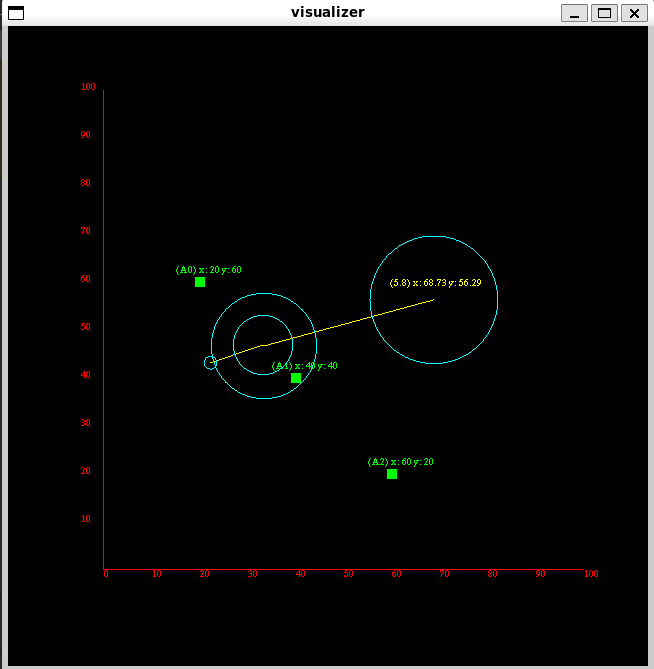


Рисунок 11 – Визуализация результатов четвертого эксперимента

Здесь можно считать, что корректно и более или менее интерпретируемо, определено положение принимающей антенны лишь в первый момент времени. В последний момент обнаружения мы получили слишком большую погрешность для интерпретируемости таких координат, а измерения в моменты времени 2 и 3 и вовсе смешались и имеют большую погрешность порядка 10 метров.

Из экспериментов можно сделать вывод, что расположение антенн-приёмников для корректной работы алгоритма важно, однако даже при самых плохих случаях, как в последних двух испытаниях, алгоритм отрабатывает и даёт максимально возможную интерпретируемость положения объекта.

# **ЗАКЛЮЧЕНИЕ**

В процессе выполнения курсовой работы был разработан и реализован алгоритм вычисления координаты антенны-излучателя по рассчитанным расстояниям до неё от антенн-приёмников. Расстояния рассчитаны с использованием формулы, отражающей взаимосвязь между мощностью излучения и расстоянием до объекта. Алгоритм учитывает погрешность в измерениях антенн-приёмников и даёт возможность визуализировать как полученные координаты объекта, так и погрешность в вычислениях.

Также разработано консольное приложения на языке Python, обрабатывающее входные данные и используя алгоритмы компьютерной графики, основанные на технологии OpenGL, и вышеупомянутый алгоритм расчёта координат антенн-излучателей, визуализирует их положения и траектории движения.

По результатам тестирования был сделан вывод о том, что полученные визуализации в точности отражают входные данные. Приложение справляется с задачей построения траекторий движения антенн-излучателей на разных частотах, при этом различая их. Также на каждом этапе траектории визуализируется возможная погрешность в измерении координат. Тестированием также выявлены возможные положения антенн-приёмников, при которых результаты работы алгоритма определения координат являются точными и интерпретируемыми.

Разработанное в результате выполнения курсовой работы приложение может быть использовано во многих сферах, требующих определения местоположения объекта в реальном времени и визуализации траектории изменения его координат. Одна из таких задач – задача пеленгации, подробно раскрытая во введении. При обеспечении последовательной передачи данных, например через Socket, приложение может принимать входные данные от системы, реализующей пеленгацию, и обрабатывать их, строя соответствующие визуализации.

**СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ**

1. Чириков С.В. Алгоритмы компьютерной графики (методы растриро-вания кривых). Учебное пособие – СПб : СПб ГИТМО(ТУ), 2001.
2. Вельтмандер, П. В. Машинная графика / П. В. Вельтмандер // Издательский дом «Вильямс», 2000.
3. Херн, Дональд, Паулин Бейкер, М. Компьютерная графика и стандарт OpenGL / Дональд Хери, М. Паулин Бейкер // Издательский дом «Вильямс», 2005. -1216 с.
4. Роджерс Д. Алгоритмические основы машинной графики. — М.: Мир, 1989. — 548-73 с.
5. Документация PyOpenGL [Электронный курс] – URL: https://pyopengl.sourceforge.net/documentation/
6. Документация OpenGL [Электронный курс] – URL: <https://www.opengl.org/> (дата обращения: 15.10.2023)
7. Документация библиотеки glfw [Электронный курс] – URL: <https://www.glfw.org/documentation> (дата обращения: 15.10.2023)
8. Документация библиотеки numpy [Электронный курс] – URL:

<https://numpy.org/doc/> (дата обращения: 20.10.2023)

**ПРИЛОЖЕНИЕ**

Листинг 1 – main.py

**import** **math**

**import** **glfw**

**from** **OpenGL.GL** **import** \*

**from** **OpenGL.GLUT** **import** \*

**import** **points\_dealing**

corr = -**0.7**

field\_width = **1.5**

field\_height = **1.5**

size = **0**

MAX\_CORR = **100** # max size of field

locators = [['A0', **20**, **60**],

['A1', **40**, **40**],

['A2', **60**, **20**]]

emitter\_trace = [(**5000**, **20**, **1**, **30**, **2**),

(**5000**, **40**, **1**, **50**, **2**),

(**5000**, **70**, **10**, **60**, **2**),

(**5000**, **30**, **1**, **50**, **2**)]

emitters = {}

data = [(**5000**, **10**, **10**, **10**),

(**5000**, math.sqrt(**600**), math.sqrt(**300**), math.sqrt(**500**)),

(**5000**, math.sqrt(**1000**), math.sqrt(**400**), math.sqrt(**700**))]

data\_amp = [(**5.8**, **25**, **25**, **17**),

(**5.8**, **40**, **40**, **19.8**),

(**5.8**, **25**, **25**, **25**),

(**5.8**, **17**, **18**, **20**)]

**def** **get\_distance**(power: float):

**return** **10** \*\* (- (power - **50.0**) / **20.0**)

**def** **draw\_field**():

glBegin(GL\_LINES)

zero\_cords = (corr, corr)

y\_cords = (corr, field\_width + corr)

x\_cords = (field\_width + corr, corr)

glColor3f(**1.0**, **0.0**, **0.0**)

glVertex2f(**0** + corr, **0** + corr)

glColor3f(**1.0**, **0.0**, **0.0**)

glVertex2f(**0** + corr, field\_width + corr)

glColor3f(**1.0**, **0.0**, **0.0**)

glVertex2f(**0** + corr, **0** + corr)

glColor3f(**1.0**, **0.0**, **0.0**)

glVertex2f(field\_height + corr, **0** + corr)

glEnd()

i = zero\_cords[**0**]

cord = **0**

cord\_step = MAX\_CORR / **10**

fin = x\_cords[**0**]

step = abs(zero\_cords[**0**] - x\_cords[**0**]) / **10**

**while** i < fin + **0.1**:

text = str(int(cord))

glRasterPos2d(i, corr - **0.02**)

**for** char **in** text:

glutBitmapCharacter(GLUT\_BITMAP\_TIMES\_ROMAN\_10, ord(char))

i += step

cord += cord\_step

i = zero\_cords[**0**]

cord = **0**

cord\_step = MAX\_CORR / **10**

fin = x\_cords[**0**]

step = abs(zero\_cords[**0**] - x\_cords[**0**]) / **10**

i += step

cord += cord\_step

**while** i < fin + **0.1**:

text = str(int(cord))

glRasterPos2d(corr - **0.07**, i)

**for** char **in** text:

glutBitmapCharacter(GLUT\_BITMAP\_TIMES\_ROMAN\_10, ord(char))

i += step

cord += cord\_step

**def** **draw\_circle**(radius, center):

glBegin(GL\_LINE\_LOOP)

line\_amount = **50**

twice\_pi = **2.0** \* **3.1415**

**for** i **in** range(line\_amount):

glColor4f(**0.0**, **1.0**, **1.0**, **1.0**)

glVertex2f(

center[**0**] + (radius \* math.cos(i \* twice\_pi / line\_amount)),

center[**1**] + (radius \* math.sin(i \* twice\_pi / line\_amount))

)

glEnd()

**def** **draw\_emitter**(data: list):

glLineWidth(**2.0**)

glBegin(GL\_LINE\_STRIP)

last\_pos\_x, last\_pos\_y, last\_freq = **0**, **0**, **0**

**for** i **in** range(**0**, len(data)):

glColor3f(**1.0**, **1.0**, **0.0**)

glVertex2f(data[i][**1**] / MAX\_CORR \* **1.5** + corr, data[i][**2**] / MAX\_CORR \* **1.5** + corr)

last\_pos\_x = data[i][**1**]

last\_pos\_y = data[i][**2**]

last\_freq = data[i][**0**]

glEnd()

**for** i **in** range(len(data)):

draw\_circle(data[i][**3**] / MAX\_CORR \* **1.5**,

(data[i][**1**] / MAX\_CORR \* **1.5** + corr, data[i][**2**] / MAX\_CORR \* **1.5** + corr))

glColor3f(**1.0**, **1.0**, **0.0**)

text = "(" + str(last\_freq) + ")" + " x: " + str(last\_pos\_x) + " y: " + str(last\_pos\_y)

glRasterPos2d((last\_pos\_x - **9**) / MAX\_CORR \* **1.5** + corr, (last\_pos\_y + **3**) / MAX\_CORR \* **1.5** + corr)

**for** char **in** text:

glutBitmapCharacter(GLUT\_BITMAP\_TIMES\_ROMAN\_10, ord(char))

**def** **draw\_line**(x1, y1, x2, y2, max\_corr):

glBegin(GL\_LINES)

glColor3f(**1.0**, **1.0**, **0.0**)

glVertex2f(x1 / max\_corr \* **1.5** + corr, y1 / max\_corr \* **1.5** + corr)

glColor3f(**1.0**, **1.0**, **0.0**)

glVertex2f(x2 / max\_corr \* **1.5** + corr, y2 / max\_corr \* **1.5** + corr)

glEnd()

**def** **draw\_receiver**(name: str, x: int, y: int, max\_corr):

glColor3f(**0**, **1.0**, **0.0**)

text = "(" + name + ")" + " x: " + str(x) + " y: " + str(y)

glRasterPos2d((x - **5**) / max\_corr \* **1.5** + corr, (y + **2**) / max\_corr \* **1.5** + corr)

**for** char **in** text:

glutBitmapCharacter(GLUT\_BITMAP\_TIMES\_ROMAN\_10, ord(char))

glPointSize(**10**)

glBegin(GL\_POINTS)

glVertex3f(x / max\_corr \* **1.5** + corr, y / max\_corr \* **1.5** + corr, **0**)

glEnd()

**def** **update\_data**(data):

**global** emitters

stop\_data = []

**for** elem **in** data:

key = elem[**0**]

val = tuple(elem[**1**:])

**if** key **not** **in** emitters:

emitters[key] = list()

emitters[key].append(val)

**elif** val **not** **in** emitters[key]:

emitters[key].append(val)

**def** **display**(window):

**global** field\_height

**global** field\_width

glClear(GL\_COLOR\_BUFFER\_BIT)

glLoadIdentity()

glClearColor(**0.0**, **0.0**, **0.0**, **0.0**)

glutInit()

glPushMatrix()

draw\_field()

**for** elem **in** locators:

draw\_receiver(elem[**0**], elem[**1**], elem[**2**], MAX\_CORR)

update\_data(data\_amp)

data\_for\_draw = []

**for** key, val **in** emitters.items():

**for** elem **in** val:

cords = points\_dealing.count\_point\_from\_3\_dists(locators[**0**][**1**], locators[**0**][**2**], get\_distance(elem[**0**]),

locators[**1**][**1**], locators[**1**][**2**], get\_distance(elem[**1**]),

locators[**2**][**1**], locators[**2**][**2**], get\_distance(elem[**2**]))

data\_for\_draw.append([key] + list(cords))

draw\_emitter(data\_for\_draw)

data\_for\_draw = []

glPopMatrix()

glfw.swap\_buffers(window)

glfw.poll\_events()

**def** **main**():

**if** **not** glfw.init():

**return**

window = glfw.create\_window(**640**, **640**, "visualizer", None, None)

**if** **not** window:

glfw.terminate()

**return**

glfw.make\_context\_current(window)

glfw.set\_key\_callback(window, key\_callback)

**while** **not** glfw.window\_should\_close(window):

display(window)

glfw.destroy\_window(window)

glfw.terminate()

**def** **key\_callback**(window, key, scancode, action, mods):

**global** MAX\_CORR

**global** corr

**if** key == glfw.KEY\_UP:

MAX\_CORR += **5**

**if** key == glfw.KEY\_DOWN:

MAX\_CORR -= **5**

**if** MAX\_CORR == **0**:

MAX\_CORR += **1**

**if** key == glfw.KEY\_RIGHT:

corr += **0.05**

**if** key == glfw.KEY\_LEFT:

corr -= **0.05**

**if** corr == **0**:

corr += **0.001**

**if** \_\_name\_\_ == '\_\_main\_\_':

main()

Листинг 2 – points\_dealing.py

**import** **math**

**def** **find\_closest\_point**(x1, y1, r1, x2, y2, r2):

distance = math.sqrt((x2 - x1) \*\* **2** + (y2 - y1) \*\* **2**)

proportion1 = r1 / distance

closest\_point1\_x = x2 + (x1 - x2) \* proportion1

closest\_point1\_y = y2 + (y1 - y2) \* proportion1

proportion2 = r2 / distance

closest\_point2\_x = x1 + (x2 - x1) \* proportion2

closest\_point2\_y = y1 + (y2 - y1) \* proportion2

**return** round((closest\_point1\_x + closest\_point2\_x) / **2**, **2**), round((closest\_point1\_y + closest\_point2\_y) / **2**, **2**)

**def** **find\_closest\_point2**(x1, y1, r1, x2, y2, r2):

angle = math.atan2(y2 - y1, x2 - x1)

x\_res\_1 = x1 + r1 \* math.cos(angle)

y\_res\_1 = y1 + r1 \* math.sin(angle)

angle2 = math.atan2(y1 - y2, x1 - x2)

x\_res\_2 = x2 + r2 \* math.cos(angle2)

y\_res\_2 = y2 + r2 \* math.sin(angle2)

**return** round((x\_res\_1 + x\_res\_2) / **2**, **2**), round((y\_res\_1 + y\_res\_2) / **2**, **2**)

**def** **get\_intersections**(x0, y0, r0, x1, y1, r1):

d = math.sqrt((x1 - x0) \*\* **2** + (y1 - y0) \*\* **2**)

**if** d > r0 + r1:

**return** ()

**if** d < abs(r0 - r1):

**return** ()

**if** d == **0** **and** r0 == r1:

**return** ()

**else**:

a = (r0 \*\* **2** - r1 \*\* **2** + d \*\* **2**) / (**2** \* d)

h = math.sqrt(r0 \*\* **2** - a \*\* **2**)

x2 = x0 + a \* (x1 - x0) / d

y2 = y0 + a \* (y1 - y0) / d

x3 = x2 + h \* (y1 - y0) / d

y3 = y2 - h \* (x1 - x0) / d

x4 = x2 - h \* (y1 - y0) / d

y4 = y2 + h \* (x1 - x0) / d

**return** (round(x3, **2**), round(y3, **2**)), (round(x4, **2**), round(y4, **2**))

**def** **count\_dist**(p1, p2):

# return math.hypot(p1[0] - p2[0], p1[1] - p2[1])

**return** math.sqrt((p1[**0**] - p2[**0**]) \*\* **2** + (p1[**1**] - p2[**1**]) \*\* **2**)

**def** **get\_three\_nearest\_points**(points: list):

dist = **999999**

res\_points = (points[**0**], points[**1**], points[**2**])

**for** p1 **in** points:

**for** p2 **in** points:

**for** p3 **in** points:

**if** p1 != p2 **and** p1 != p3 **and** p2 != p3:

dist1 = count\_dist(p1, p2)

dist2 = count\_dist(p1, p3)

dist3 = count\_dist(p2, p3)

**if** dist1 + dist2 + dist3 < dist:

dist = dist1 + dist2 + dist3

res\_points = (p1, p2, p3)

**return** res\_points

**def** **get\_trig\_center**(points):

x, y = (points[**0**][**0**] + points[**1**][**0**] + points[**2**][**0**]) / **3**, (points[**0**][**1**] + points[**1**][**1**] + points[**2**][**1**]) / **3**

error = (count\_dist((x, y), points[**0**]) + count\_dist((x, y), points[**1**]) + count\_dist((x, y), points[**2**])) / **3**

**return** round((points[**0**][**0**] + points[**1**][**0**] + points[**2**][**0**]) / **3**, **2**), round(

(points[**0**][**1**] + points[**1**][**1**] + points[**2**][**1**]) / **3**, **2**), error

**def** **count\_point\_from\_3\_dists**(x1, y1, d1, x2, y2, d2, x3, y3, d3):

intersections1 = get\_intersections(x1, y1, d1, x2, y2, d2)

intersections2 = get\_intersections(x1, y1, d1, x3, y3, d3)

intersections3 = get\_intersections(x2, y2, d2, x3, y3, d3)

**if** intersections1 == ():

intersections1 = (find\_closest\_point2(x1, y1, d1, x2, y2, d2),)

**if** intersections2 == ():

intersections2 = (find\_closest\_point2(x1, y1, d1, x3, y3, d3),)

**if** intersections3 == ():

intersections3 = (find\_closest\_point2(x2, y2, d2, x3, y3, d3),)

points = list(intersections1) + list(intersections2) + list(intersections3)

**if** len(points) < **3**:

res = get\_trig\_center([(x1, y1), (x2, y2), (x3, y3)])

**return** res

near\_points = get\_three\_nearest\_points(points)

**return** get\_trig\_center(near\_points)