МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение высшего образования
УРАЛЬСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
имени первого Президента России Б.Н. Ельцина

ИНСТИТУТ ЕСТЕСТВЕННЫХ НАУК И МАТЕМАТИКИ

Департамент математики, механики и компьютерных наук

РАЗРАБОТКА ПРОГРАММНОГО КОМПЛЕКСА МОДЕЛИРОВАНИЯ И ВИЗУАЛИЗАЦИИ ДВИЖЕНИЯ БЕСПИЛОТНЫХ ЛЕТАТЕЛЬНЫХ АППАРАТОВ

Направление подготовки 01.03.01 «Математика»

Директор департамента
к.ф.-м.н. Е. С. Пьяньзина

Выпускная квалификационная работа бакалавра
Дербенева
Леонида Олеговича

Нормоконтролер:
О.А. Суслова

Научный руководитель:
к.ф.-м.н. С.С. Кумков

Екатеринбург
2024

РЕФЕРАТ

Дербенев Л. О.: РАЗРАБОТКА ПРОГРАММНОГО КОМПЛЕКСА МОДЕЛИРОВАНИЯ И ВИЗУАЛИЗАЦИИ ДВИЖЕНИЯ БЕСПИЛОТНЫХ ЛЕТАТЕЛЬНЫХ АППАРАТОВ, Выпускная квалификационная работа на соискание степени бакалавра по направлению подготовки 01.03.01 «Математика», 29 стр., рис. 9, источн. 3.

Ключевые слова: БЕСПИЛОТНЫЕ ЛЕТАТЕЛЬНЫЕ АППАРАТЫ, АВТОМАТИЧЕСКОЕ РАЗРЕШЕНИЕ КОНФЛИКТНЫХ СИТУАЦИЙ, РАЗРЕШАЮЩИЕ МАНЕВРЫ, МОДЕЛИРОВАНИЕ ДВИЖЕНИЯ БПЛА, МОДЕЛИ ДИНАМИКИ

В настоящее время активно развивается движение гражданских беспилотных летательных аппаратов (БПЛА). При этом возникают конфликты БПЛА с другими БПЛА, а также с самолётами малой авиации и вертолётами, которые движутся в тех же районах и на тех же высотах, что и БПЛА. В таких условиях важной является разработка алгоритмов для бортовых компьютеров БПЛА для автоматической выработки и отработки манёвров, разрешающих возникающие конфликтные ситуации с другими летательными аппаратами. В работе описывается разработка программного комплекса для моделирования движения БПЛА и процесса разрешения конфликтных ситуаций, позволяющей оценивать качество тех или иных процедур выработки разрешающего маневра. Вычислительный комплекс разработан на языке C++. Формат выходного файла с результатами моделирования вырабатывался с совместимым с возможностями имеющейся визуализационной программы, создаемой параллельной другим разработчиком с использованием библиотеки Plotly.PY на языке Python.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В настоящее время активно развивается движение гражданских беспилотных летательных аппаратов (БПЛА). С их помощью проводится доставка различных грузов в удаленные поселки (чем раньше занималась только малая авиация), мониторинг состояния природных и искусственных объектов (лесов, озер, рек, трубо-, нефте- и газопроводов, ЛЭП и др.).

При этом конфликтов БПЛА и больших самолётов почти не возникает: в рабочем режиме они движутся на разных высотах, а вход БПЛА в аэропортовые зоны весьма жёстко регламентируется. Однако возникают конфликты БПЛА с другими БПЛА, а также с самолётами малой авиации и вертолётами, которые движутся в тех же районах и на тех же высотах, что и БПЛА.

Ситуация такова, что движение БПЛА не диспетчеризируется централизовано, как движение больших самолётов. Часто полет малого ЛА объявляется в уведомительном порядке. Поэтому текущая воздушная обстановка актуально недоступна операторам БПЛА, которые, кроме того, не имеют связи между собой и с диспетчерскими службами УВД. В то же время достаточно чётко прописаны регламенты того, какую информацию о собственном движении во время полёта БПЛА сообщает в эфир в виде широковещательных пакетов.

В таких условиях важной является разработка алгоритмов для бортовых компьютеров БПЛА для обработки полученной информации о движении окружающих летательных аппаратов, малых и больших, и для последующей автоматической выработки и отработки манёвров, разрешающих возникающие конфликтные ситуации с другими летательными аппаратами.

В рамках создания таких алгоритмов важным является численное моделирование ансамбля БПЛА, движущихся по тем или иным траекториям в тех или иных условиях. Такое моделирование позволяет провести предварительную оценку качества предлагаемых манёвров, разрешающих конфликты. Для адекватности проводимых оценок требуется, чтобы модели движения и маневрирования БПЛА достаточно точно соответствовали движению реальных аппаратов.

В рамках данной работы были предприняты шаги в направлении создания такого моделирующего программного комплекса, включающего реализации основных моделей движения БПЛА, модели радиообменов и процедур обнаружения конфликтов. В дальнейшем в этот комплекс можно встраивать и тестировать те или иные алгоритмы выработки манёвров уклонения с целью оценки их качества.

1 Постановка задачи

Входной информацией для моделирующего комплекса является набор данных о каждом БПЛА из рассматриваемого ансамбля. Сюда входит информация о типе динамики данного БПЛА, её параметрах, а также априорный полетный план. Описание плана представляет собой номинальную траекторию движения, представляющую собой ломаную линию в пространстве, описываемую координатами своих вершин и моментами времени, когда планируется проход БПЛА через них.

Модели динамики жестко прописываются в коде комплекса и включают в себя модели, представляющих движение аппаратов самолетного, вертолетного типа и коптеров. Параметры модели динамики задаются индивидуально для каждого БПЛА.

Движение на промежутках между парами последовательных точек предполагается равномерным прямолинейным, так что положения точек и времена их прохода определяют скорость движения на каждом промежутке.

Для моделирования движения каждого БПЛА требуется реализовать процедуру углового разворота, адекватную для каждого типа динамики. Данная процедура вырабатывает движение БПЛА вблизи контрольных точек маршрута, «срезающее» угол ломаной (что отражает реальное движение БПЛА в такой ситуации). Здесь планируется вырабатывать некоторое разумное движение — *штурманскую прокладку*, сопрягающее два соседних отрезка планового маршрута и скорости движения на каждом из них. Выработка управления БПЛА на участках углового разворота производится посредством ПИД-регулятора, прицеливающего на штурманскую прокладку.

Для блока моделирования радиообменов предусматривается выработка информации, вещаемой в эфир каждым БПЛА при реальном движении. Для комплекса существенными являются два типа информационных пакетов: с текущем положением БПЛА и с краткосрочным прогнозом его движения. Такая информация необходима для выявления конфликтных ситуаций между БПЛА и выработки разрешающих маневров.

Кроме того, в комплекс требуется передавать параметры интегрирования движения БПЛА: шаг интегрирования, шаг записи информации о положении БПЛА в выходной файл.

На выходе для последующей визуализации расчетов комплекс должен выдавать в файл информацию о результатах моделирования движения каждого БПЛА и о процессах принятия решения, происходящих на борту:

- начальный полетный план;
- изменения полетного плана вследствие выработки маневров, разрешающих конфликты;
- набор положений БПЛА в моменты с заданной сетки по времени (для визуализации);
- пакеты вещания (с текущими положениями БПЛА и краткосрочными прогнозами движения);
- индикация наличия конфликтов.

2 Теоретическая часть

2.1 Рассматриваемые модели динамики

Считается, что выбрана стандартная система координат, используемая в системах УВД: начало координат O выбрано где-то в районе управления, ось Ox направлена на восток, ось Oz — на сервер, ось Oy направлена вертикально вверх. Таким образом, движение по оси y является вертикальным движением и часто описывается отдельно от движения в плоскости Oxz.

2.1.1 Материальная точка

Материальная точка — это идеализированное тело, обладающее массой, но не имеющее размеров. В физике материальная точка используется как модель для описания движения объекта, когда его размерами можно пренебречь по сравнению с масштабами задачи. В рамках создаваемого комплекса данная модель является вспомогательной, используемой при отладке, поскольку, в частности, движения соответствующего объекта достаточно просто понимаемы.

Материальная точка имеет следующую модель движения:

$$\ddot{r} = m \cdot u, \tag{2.1}$$

где $r = (x, y, z)^{\mathrm{T}}$ — радиус-вектор положения объекта, $u = (u_x, u_y, u_z)^{\mathrm{T}}$ — управление, являющееся ускорением, m — масса точки. Является параметром системы.

Покоординатная запись:

$$\ddot{x} = u_x, \quad \ddot{y} = u_y, \quad \ddot{z} = u_z.$$

Запись, включающая скорости:

$$\begin{aligned} &\dot{x} = V_x, \quad \dot{y} = V_y, \quad \dot{z} = V_z, \\ &\dot{V}_x = u_x, \quad \dot{V}_y = u_y, \quad \dot{V}_z = u_z. \end{aligned}$$

На управление накладывается геометрическое ограничение $\|u\| \leqslant u^{\max}$. Также возможен вариант ограничений, развязывающий горизонтальное и вертикальное движение: $\big\|(u_x,u_z)\big\| \leqslant u^{\max}_{\text{гор}}$, $|u_y| \leqslant u^{\max}_{\text{верт}}$.

2.1.2 Коптер

Недостаток модели материальной точки заключается в том, что нет никаких ограничений на максимальную скорость, развиваемую объектом, а также отсутствие учета сопротивления среды (воздуха) движению объекта.

Простейшей моделью, учитывающей эти обстоятельства, является модель, управляемая *командным сигналом* скорости:

$$\begin{aligned} &\dot{x} = V_x, && \dot{y} = V_y, && \dot{z} = V_z, \\ &\dot{V}_x = \frac{u_x - V_x}{l_{xz}}, && \dot{V}_y = \frac{u_y - V_y}{l_y}, && \dot{V}_z = \frac{u_z - V_y}{l_{xz}}. \end{aligned}$$

Здесь $u = (u_x, u_y, u_z)^{\mathrm{T}}$ — управление, командный сигнал скорости, имеющий смысл желаемой скорости по каждой из координат; l_{xz}, l_y — коэффициенты, описывающие инерционность выхода на выбранный уровень скорости: выход осуществляется за время порядка $3l$.

Командный сигнал ограничен по модулю, что соответствует максимально возможной скорости, которую может развивать ЛА.

Данная модель является простейшей моделью движения лёгкого коптера, который достаточно быстро может менять скорость своего движения, но имеет ограничение по максимальной скорости из-за относительно малой мощности двигателя.

2.1.3 Вертолетная модель

Величины вертикальной и горизонтальной скоростей коптера обусловлены, по большому счету, только скоростью вращения винтов коптера. Направление горизонтальной скорости и соотношение горизонтальной и вертикальной скоростей обусловлены углом наклона коптера. И то, и другое достаточно быстро меняется.

В случае с достаточно большим БПЛА вертолетного типа изменение угловой ориентации уже достаточно ощутимо длительный процесс, так что в модели вертолетной динамики следует развязать управление величинами вертикальной и горизонтальной скоростей, а также управление курсом движения аппарата:

$$
\begin{aligned}
&\dot{x} = V_{\text{гор}} \cos\psi, \\
&\dot{z} = V_{\text{гор}} \sin\psi, \\
&\dot{y} = V_{\text{верт}}, \\
&\dot{\psi} = \tfrac{\beta_{\text{бок}}}{V_{\text{гор}}}\, u_{\text{бок}}, \quad |u_{\text{бок}}| \leqslant 1, \\
&\dot{V}_{\text{гор}} = a, \quad a_{\min} \leqslant a \leqslant a_{\max}, \quad V_{\text{гор}}^{\min} \leqslant V_{\text{гор}} \leqslant V_{\text{гор}}^{\min}, \\
&\dot{V}_{\text{верт}} = u_{\text{верт}}, \quad u_{\text{верт}}^{\min} \leqslant u_{\text{верт}} \leqslant u_{\text{верт}}^{\max}, \quad V_{\text{верт}}^{\min} \leqslant V_{\text{верт}} \leqslant V_{\text{верт}}^{\max}.
\end{aligned}
$$

Здесь ψ — угол курса; $u_{\text{бок}}$ — ускорение, управляющее разворотом круса; $u_{\text{верт}}$ — ускорение (создаваемое изменением скорости вращения винтов), управляющее вертикальной скоростью; a — ускорение, управляющее величиной горизонтальной скорости (продольное); $\beta_{\text{бок}}$ — коэффициент горизонтальной маневренности судна.

Динамика для изменение курсового угла использует традиционное описание со скоростью в знаменателе: чем выше скорость, тем ниже маневренность аппарата, то есть тем медленнее меняет аппарат направление движения при тех же боковых ускорениях.

Заметим, что сама по себе модель не включает ограничения на величины скоростей, их приходится ограничивать через дополнительно вводимые неравенства. Можно подменить управление скоростью через ускорение на управление скоростью через командный сигнал, что даст ограничение на максимальные скорости в виде ограничения на максимальное значение соответствующего командного сигнала:

$$
\begin{aligned}
&\dot{x} = V_{\text{гор}} \cos\psi, \\
&\dot{z} = V_{\text{гор}} \sin\psi, \\
&\dot{y} = V_{\text{верт}}, \\
&\dot{\psi} = \tfrac{\beta_{\text{бок}}}{V_{\text{гор}}}\, u_{\text{бок}}, \quad |u_{\text{бок}}| \leqslant 1, \\
&\dot{V}_{\text{гор}} = (V_{\text{гор}}^{\text{упр}} - V_{\text{гор}})/l_{\text{гор}}, \quad V_{\text{гор}}^{\text{упр, min}} \leqslant V_{\text{гор}}^{\text{упр}} \leqslant V_{\text{гор}}^{\text{упр, max}}, \\
&\dot{V}_{\text{верт}} = (V_{\text{верт}}^{\text{упр}} - V_{\text{верт}})/l_{\text{верт}}, \quad V_{\text{верт}}^{\text{упр, min}} \leqslant V_{\text{верт}}^{\text{упр}} \leqslant V_{\text{верт}}^{\text{упр, max}}.
\end{aligned}
$$

2.1.4 Самолетная модель

При движении самолета его вертикальная скорость обусловлена *углом атаки* — углом между плоскостью крыла и скоростью набегающего потока. Угол плоскости крыла связан с *углом тангажа*, то есть с углом наклона самолета относительно оси, проходящей через плоскость крыла перпендикулярно фюзеляжу. Таким образом, управление

движением самолета связано с управлением не только углом курса, но и углом тангажа:

$$\begin{aligned}
&\dot{x} = V\cos\theta\cos\psi,\\
&\dot{z} = V\cos\theta\sin\psi,\\
&\dot{y} = V\sin\theta,\\
&\dot{\theta} = \tfrac{\beta_{\text{верт}}}{V}\,u_{\text{верт}},\\
&\dot{\psi} = \tfrac{\beta_{\text{бок}}}{V}\,u_{\text{бок}},\\
&|u_{\text{верт}}| \leqslant 1, \quad |u_{\text{бок}}| \leqslant 1,\\
&\dot{V} = a, \quad a_{\min} \leqslant a \leqslant a_{\max}, \quad V_{\min} \leqslant V \leqslant V_{\max}.
\end{aligned}$$

Здесь θ — угол тангажа, ψ — угол курса; $u_{\text{верт}}$, $u_{\text{бок}}$ — ускорения, управляющие углами тангажа и курса; a — ускорение, управляющее скоростью; $\beta_{\text{верт}}$, $\beta_{\text{бок}}$ — коэффициенты маневренности судна. Модель взята из [3].

Заметим, что скорость уже имеет единое значение, поскольку именно величина линейной скорости вкупе с текущей угловой ориентацией обуславливает вертикальную и горизонтальную составляющую скорости аппарата.

Опять, управление скоростью через ускорения можно заменить на управление через командный сигнал и отказаться от ограничений на величину скорости:

$$\begin{aligned}
&\dot{x} = V\cos\theta\cos\psi,\\
&\dot{z} = V\cos\theta\sin\psi,\\
&\dot{y} = V\sin\theta,\\
&\dot{\theta} = \tfrac{\beta_{\text{верт}}}{V}\,u_{\text{верт}},\\
&\dot{\psi} = \tfrac{\beta_{\text{бок}}}{V}\,u_{\text{бок}},\\
&|u_{\text{верт}}| \leqslant 1, \quad |u_{\text{бок}}| \leqslant 1,\\
&\dot{V} = (V^{\text{упр}} - V)/l, \quad V^{\text{упр}}_{\min} \leqslant V^{\text{упр}} \leqslant V^{\text{упр}}_{\max}.
\end{aligned}$$

2.2 Полетный план

В начале движения БПЛА имеет свой полетный план, задаваемый как совокупность $\{(r_i, t_i)\}_{i=1}^{n}$, где $r_i = (x_i,y_i,z_i)$ — точка трехмерного пространства, t_i — момент времени, в который БПЛА должен оказаться в заданной точке. Считаем, что $t_i < t_{i+1}$.

Считается, что между соседними точками полетного плана БПЛА движется равномерно прямолинейно, так что на отрезке $[r_i,r_{i+1}]$ его скорость находится как

$$V_i = \frac{r_{i+1} - r_i}{t_{i+1} - t_i}\,.$$

2.3 Построение штурманской прокладки

Заметим, что заданная траектория является ломаной линией. Однако БПЛА в точках излома не может мгновенно менять направление скорости. Как следствие встает вопрос о том, как организовать движение или моделирование движения ЛА вблизи таких точек.

В реальности используется так называемая *штурманская прокладка* — вспомогательная траектория, сочленяющая отрезки соседних прямолинейных участков некоторой траекторией, по которой может пройти ЛА.

В рамках моделирования, когда не требуется воспроизводить точные траектории ЛА, а вычислять движения, достаточно близкие к ним, можно изменить подход. В

качестве штурманской прокладки использовать любую траекторию, а управление при моделировании движения строить на основе прицеливания/притягивания моделируемого движения к штурманской прокладке. Соответствующее управление можно строить с использованием того или иного регулятора (см. ниж раздел 2.4).

В рамках проекта реализовано несколько вариантов штурманских прокладок.

2.3.1 «Наивная» прокладка

В этом варианте точка, на которую прицеливается движение ЛА, просто движется по ломаной номинального полетного плана. Очевидным образом около вершины ломаной ЛА делает переколебание, когда пытается отслеживать мгновенное изменение движения, случающееся в вершине ломаной. Однако такие моделирования тоже проводились.

2.3.2 Прокладка по дуге окружности

В реальности сопряжение прямолинейных участков движения осуществляется с помощью движения с постоянной скоростью по дуге окружности. Однако в рамках рассматриваемой постановки на разных участках движения величина линейной скорости отличается, так что во время движения нужно изменять и значение модуля скорости тоже.

Пусть количество точек на ломаной $n > 2$. Рассмотрим два соседних отрезка ломаной с вершинами в точках (r_k,t_k), (r_{k+1},t_{k+1}), (r_{k+2},t_{k+2}), $1 \leqslant k \leqslant n-2$. Из предположения о равномерности движения на отрезках известны скорости V_k и V_{k+1} движения на этих отрезках.

Пусть изменение курса ЛА управляется боковым ускорением величины a. Тогда, используя начальное значение скорости V_k, модно найти радиус разворота. По нему найти точки начала и конца разворота и моменты прохода этих точек на номинальной траектории.

Затем в качестве точки, на которую прицеливать движение, можно взять точку, равномерно движущуюся по этой дуге окружности, а в качестве вектора скорости, на который прицеливать вектор актуальной скорости — вектор, равномерно разворачивающийся от направления V_k к направлению V_{k+1} и равномерно меняющий свою длину от $\|V_k\|$ до $\|V_{k+1}\|$.

Соответствующие расчетные формулы выглядят следующим образом.

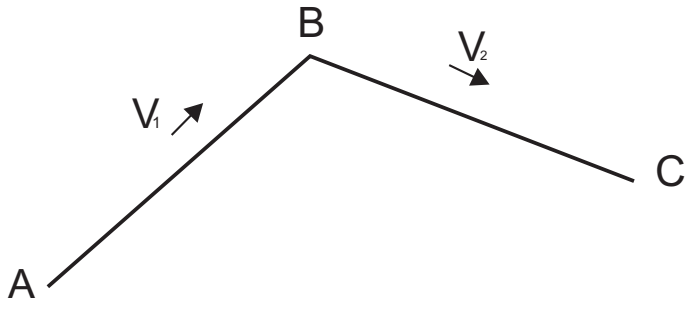

Рисунок 2.3.2.1 — Два отрезка пути

Обозначим

$$A := r_k, \quad B := r_{k+1}, \quad C := r_{k+2}, \quad V_1 := V_k, \quad V_2 := V_{k+1}.$$

Радиус разворота:

$$r = \frac{V_1^2}{a}.$$

Угол α между отрезками AB И BC:

$$\cos\alpha = \frac{\langle V_1, V_2\rangle}{\|V_1\| \cdot \|V_2\|}.$$

Зная радиус и угол между отрезками пути, можно найти длину h — расстояние от точки касания до точки стыка двух отрезков пути.

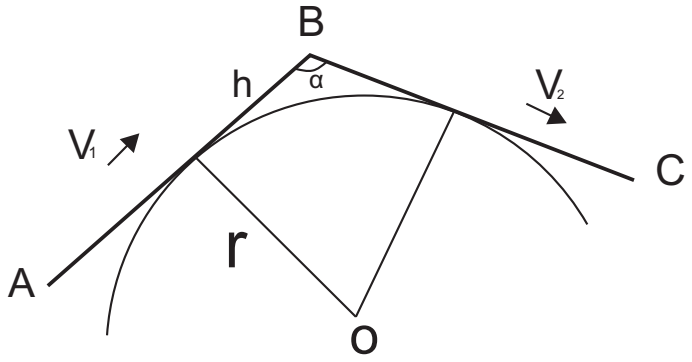

Рисунок 2.3.2.2 — Касательная окружность к двум отрезкам пути.

Как видно из рисунка 2.3.2.2, так как радиус, проведенный в точку касания, перпендикулярен касательной линии, длину h можно найти, вычислив тангенс половинного угла α, затем умножив его на радиус r. По известной тригонометрический формуле

$$\operatorname{tg}\frac{\alpha}{2} := \sqrt{\frac{1-\cos\alpha}{1+\cos\alpha}}.$$

Значит

$$h := r \cdot \sqrt{\frac{1-\cos\alpha}{1+\cos\alpha}}$$

Теперь можно найти координаты точек касания отступив от точки B, на расстояние h вдоль отрезков. Для этого найдем нормированные вектора a, c, сонаправленные векторам $\overrightarrow{BA}$, $\overrightarrow{BC}$ соответственно:

$$a := \frac{BA}{|BA|}, \quad c := \frac{BC}{|BC|}.$$

.

Находим точки касания A', C':

$$A' := a \cdot h, \quad C' := c \cdot h.$$

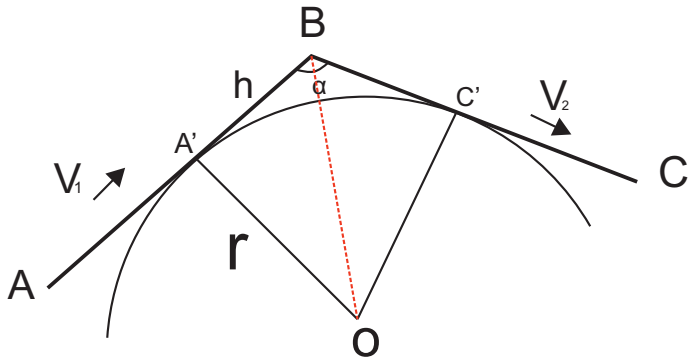

Рисунок 2.3.2.3 — Касательная окружность к двум отрезкам пути.

Так как траектория пути находится в трехмерном пространстве, нужно использовать формулу поворота для трехмерного пространства:

$$x' = (\cos\theta + n_x^2(1-\cos\theta))x + (n_x n_y(1-\cos\theta) - n_z\sin\theta)y + (n_x n_z(1-\cos\theta) + n_y\sin\theta)z$$
$$y' = (n_x n_y(1-\cos\theta) - n_z\sin\theta)x + (\cos\theta + n_y^2(1-\cos\theta))y + (n_y n_z(1-\cos(\theta)) - n_x\sin\theta)z$$
$$z' = (n_x n_z(1-\cos\theta) + n_y\sin\theta)x + (n_y n_z(1-\cos(\theta)) - n_x\sin\theta)y + (\cos\theta + n_z^2(1-\cos\theta))z$$

где $n = (n_x,n_y,n_z)^{\mathrm{T}}$ — нормированный вектор оси вращения, θ — угол, на который нужно повернуть, $r' = (x',y',z')^{\mathrm{T}}$ — новые координаты точки на окружности,$r = (x,y,z)^{\mathrm{T}}$ — старые координаты. Также, нужно учесть, что поворот в данной формуле происходит вокруг точки начала, то есть нужно будет сдвинуть старые координаты на радиус вектор центр окружности $r_0 = (x_0,y_0,z_0)^{\mathrm{T}}$, а затем новые координаты сдвинуть обратно. То есть, формула будет выглядеть следующим образом:

$$\begin{aligned}
x' &= x_0 + (\cos\theta + n_x^2(1-\cos\theta))(x-x_0) + (n_x n_y(1-\cos\theta) - n_z\sin\theta)(y-y_0) + \\
&\quad + (n_x n_z(1-\cos\theta) + n_y\sin\theta)(z-z_0) \\
y' &= y_0 + (n_x n_y(1-\cos\theta) - n_z\sin\theta)(x-x_0) + (\cos\theta + n_y^2(1-\cos\theta))(y-y_0) + \\
&\quad + (n_y n_z(1-\cos(\theta)) - n_x\sin\theta)(z-z_0) \\
z' &= z_0 + (n_x n_z(1-\cos\theta) + n_y\sin\theta)(x-x_0) + (n_y n_z(1-\cos(\theta)) - n_x\sin\theta)(y-y_0) + \\
&\quad + (\cos\theta + n_z^2(1-\cos\theta))(z-z_0)
\end{aligned}$$

Для нахождения вектора оси поворота n, нам понадобятся, уже известные, вектора a и c, а точнее их векторное произведение, ведь по сути нам нужно найти нормированный вектор, перпендикулярный плоскости, в которой происходит поворот. А мы знаем что вектора a, c лежат в этой плоскости. Находим вектор оси следующим образом:

$$n = \frac{c \times a}{|c \times a|}$$

Также, нужно отметить, что порядок умножения именно такой, ведь это задаст правильную ориентацию поворота.

2.4 ПИД-регулятор

ПИД-регулятор (пропорционально-интегрально-дифференциальный регулятор) является одним из наиболее распространенных методов регулирования систем управления. Он состоит из трех основных компонентов: пропорциональной, интегральной и дифференциальной составляющих.

Пропорциональная составляющая определяет выходной сигнал контроллера пропорционально разности между желаемым и текущим значением управляемой величины. Это позволяет реагировать на ошибку управления и регулировать систему.

Интегральная составляющая интегрирует ошибку управления с течением времени, что позволяет уменьшить статическую ошибку системы и обеспечить точное следование заданному значению.

Дифференциальная составляющая учитывает скорость изменения ошибок и предотвращает быстрые колебания системы, обеспечивая более плавное и стабильное управление.

В итоге, комбинация трех компонентов ПИД-регулятора позволяет эффективно и точно управлять системой, обеспечивая минимальное перерегулирование и быстрое достижение заданного значения.

Пусть r — заданное значение, которое нужно поддерживать, $e = (r - y)$ - невязка или ошибка регулирования. Тогда для линейной, стационарной системы ПИД-регулятор имеет вид:

$$u(t) = P + I + D = K_p \cdot e(t) + K_i \cdot \int_0^{\mathrm{T}} e(t)d\tau + K_d \frac{de}{dt}, \tag{2.2}$$

где K_p, K_i, K_d — коэффициенты усиления пропорциональной, интегрирующей и дифференцирующей составляющих.

Рассмотрим простую линейную задачу:

$$\dot{x} = Ax + Bu, \quad x \in R^n, u \in R^m \tag{2.3}$$

$u = 0$ — разрешенное управление. Отсюда следует, что $x = 0$ является точкой равновесия. Задача вывести систему в точку равновесия.

В данной работе был использован линейный пропорциональный регулятор, то есть $u = Kx$, где $K \in R^{m \times n}$. Теперь задача имеет вид:

$$\dot{x} = Ax + BKx = (A + BK)x \tag{2.4}$$

Данная система является устойчивой $\Longleftrightarrow \forall \lambda$ — собственное значение, выполняется: $Re\lambda < 0$.

В данной работе будем использовать только пропорциональный регулятор:

$$u = -k_x \cdot (x - x_w) - k_V \cdot (V_x - V_{x,w})$$

2.5 Построение регулятора для материальной точки

Построим регулятор для материальной точки. Мы имеем

$$\ddot{r} = u$$

или распишем в эквивалентной форме

$$\ddot{x} = u_x$$
$$\ddot{y} = u_y$$
$$\ddot{z} = u_z$$

Пусть $X = (x,y,z,v_x,v_y,v_z)^{\mathrm{T}}$, где v_i — соответствующие компоненты скорости.

Запишем задачу:

$$\dot{X} = AX + Bu = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \cdot X + \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot u = \begin{pmatrix} O_3 & I_3 \\ O_3 & O_3 \end{pmatrix} \cdot X + \begin{pmatrix} O_3 \\ I_3 \end{pmatrix} \cdot u$$

То есть можно сказать, что $K \in R^{3\times 6}$ или другими словами у нас 18 параметров, но заметим, что так как мы используем пропорциональный регулятор, то не равны нулю лишь $K_{x\to u_x}$, $K_{y\to u_y}$, $K_{z\to u_z}$, $K_{v_x\to u_x}$, $K_{v_y\to u_y}$, $K_{v_z\to u_z}$:

$$K = \begin{pmatrix} K_{x\to u_x} & 0 & 0 & K_{v_x\to u_x} & 0 & 0 \\ 0 & K_{y\to u_y} & 0 & 0 & K_{v_y\to u_y} & 0 \\ 0 & 0 & K_{z\to u_z} & 0 & 0 & K_{v_z\to u_z} \end{pmatrix}$$

То есть получаем следую запись

$$\begin{aligned} \dot{x} &= V_x \\ \dot{y} &= V_y \\ \dot{z} &= V_z \\ \dot{V}_x &= K_x x + K_{V_x} V_x \\ \dot{V}_y &= K_x x + K_{V_x} V_x \\ \dot{V}_z &= K_x x + K_{V_x} V_x \end{aligned}$$

2.6 Радиоканал

У каждого ЛА на борту есть радиопередатчик. Раз в определенное время каждый аппарат сообщает свое текущее состояние, на момент вещания, и свой краткосрочный план движения. В текущее состояние входит такая информация как время вещания, имя ЛА, координаты позиции, вектор скорости, модуль скорости. А в краткосрочный план входят 4 будущие позиции маршрута, то есть координаты точек в пространстве и момент времени, когда ЛА будет в ней находится.

Моменты на которые просчитываются прогнозы зафиксированы. При прохождении первой точки из набора она удаляется из набора и добавляется новая точка после последней и обновленный набор вещается в эфир.

При перестройке маршрута в эфир сообщается новый набор прогнозируемых точек.

Считается, что все бортовые часы БПЛА синхронизированы с достаточной точностью.

Вещание нужно для общения летательных аппаратов между собой, чтобы предотвращать конфликты.

2.7 Обнаружение конфликтов

Как уже упоминалось в пункте выше, каждый ЛА сообщает в радио канал свои будущие позиции для определения конфликтных промежутков времени.

Проверка делится на несколько этапов, а именно на фильтрацию по дальности, фильтрацию по сближению и на детальный анализ планов движения

Разберемся как происходит фильтрация тех аппаратов, с которыми точно не может быть конфликта. Первой проверкой является проверка расстояния, у каждого ЛА есть радиус фильтрации, если расстояние между аппаратами превышает заданное, то дальнейшее рассмотрение не требуется.

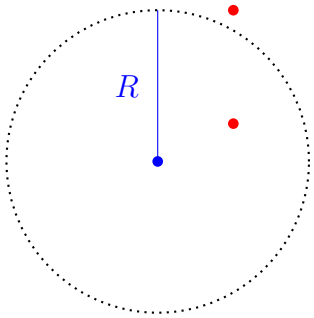

Рисунок 2.7.1 — Радиус фильтрации

Если же БПЛА находятся на меньшем расстоянии, то переходим к следующей проверке, фильтрации по сближению. Она заключается в том, что считается скалярное произведение по следующей формуле: $< p_2 - p_1, v_2 - v_1 >$, где p_1, p_2 - точки позиций в пространстве первого и второго судна соответственно, v_1, v_2 - векторы скорости первого и второго судна соответственно.

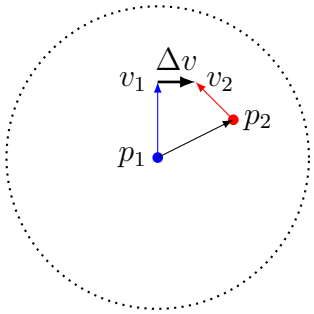

Рисунок 2.7.2 — Отфильтрованный ЛА

Если скалярное произведение больше либо равно нуля, значит конфликта нет, если же оно отрицательно, то нужно переходить к вычислению конфликтных промежутков.

Теперь рассмотрим, как происходит анализ планов. Так как, у будущих точек, которые транслируют ЛА, чаще всего имеют разные моменты времени, то требуется провести подготовительные работы, а именно «нормировать» сетку у плановых точек перемещения. Под «нормированием», имеется ввиду добавление вспомогательных точек, в те моменты времени, которых нет в пути, который сообщил ЛА в радиоканал. На всех промежутках, считаем, что БПЛА двигаются равномерно-прямолинейно от начала до конца отрезка движения. Новые точки, вычисляются следущим образом:

$$x = x_i^0 + v_i^x(t - t_i),$$
$$y = y_i^0 + v_i^y(t - t_i),$$
$$z = z_i^0 + v_i^z(t - t_i),$$

где i — номер промежутка, x, y, z — координаты новых точек маршрута, t_i — время начала рассматриваемого промежутка, t - время вычисляемой точки, v_i^x,v_i^y,v_i^z — координаты вектора скорости на данном промежутке.

Теперь, когда сетки пути номинированные, можно приступить к анализу. Весь анализ сводится к нахождению временных промежутков конфликта. Для этого нужно пройтись по каждому отрезку движения и проверить, есть ли на нем конфликт в плоскости OXZ и оси Y. Под конфликтом имеется ввиду, попадает ли другой ЛА, в радиус защитного объема(ЗО) в горизонтальной плоскости и в высоту защитного объема по вертикальной оси. Каждую точку рассматриваемого отрезка для каждого ЛА можно представить следующим образом:

$$\begin{aligned} x_i(t) &= x_0 + v_i^x(t - t_0), \\ y_i(t) &= y_0 + v_i^y(t - t_0), \\ z_i(t) &= z_0 + v_i^z(t - t_0) \end{aligned}$$

, где i — номер ЛА.

Теперь рассмотрим ситуацию, где у нас всего два ЛА с номерами 1 и 2. Тогда имеем следующие условия:

$$\begin{aligned} ||(x_1(t), z_1(t))^T - (x_2(t), z_2(t))^T|| &\leq R_{30}, \\ |y_1(t) - y_2(t)| &\leq H_{30} \end{aligned}$$

Продолжая раскрывать , получим два квадратных уравнения, решения которых будут являться концы промежутков конфликта.

3 Практическая часть

Программа была реализована на языке C++, в среде разработки Microsoft Visual Studio. Выбор стал именно таким, так как он обеспечивает нужное быстродействие вычислений и кроссплатформенность. Все графики были сформированы в Gnuplot 6.0. В данный момент в программе реализовано две динамические модели — материальная точка и коптер.

3.1 Вспомогательные классы и методы

Для данной работы были написаны вспомогательные классы и методы для работы с ними такие как:

- `Vector3` — класс для удобной работы с трехмерными векторами. По сути своей класс обладает тремя полями x,y,z для определения координат. В нем были реализованы базовые операции над векторами:
  - ▷ `+` — сложения и вычитание векторов
  - ▷ `*` — умножения вектора на скаляр
  - ▷ `/` — деления вектора на скаляр
  - ▷ `*` — умножение вектора на скаляр
  - ▷ `double scalarProduct(Vector3 v1, Vector3 v2)` — скалярное произведение трёхмерных векторов
  - ▷ `Vector3 crossProduct(Vector3 v1, Vector3 v2)` — векторное произведение трёхмерных векторов
  - ▷ `void rotate(Vector3 axis, Vector3 axisPoint, double angle)` — операция поворота вектора вокруг заданной оси и точки, через которую проходит это ось, на заданный угол
  - ▷ `double norm()` — подсчет длины вектора
  - ▷ `Vector3 getNormVector()` — получение нормированного вектора из данного
- `Point` — структура для описание точки пути, который задает пользователь. Он состоит из полей:
  - ▷ `Vector3 position` — поле для хранения координат позиции точки
  - ▷ `double time` — момент времени, в который должен оказаться ЛА в заданной позиции
- `TurnData` — структура для хранения данных, которые нужно для поворота. Имеет следующею поля:
  - ▷ `double angularVelocity` — поле для хранения предвычисленного углового ускорения, используется для поворота позиции по окружности
  - ▷ `Vector3 axisPoint` — поле для хранения точки оси поворота
  - ▷ `double angularVelocity_0` — поле для хранения предвычисленного углового ускорения, используется для вычисления скорости при повороте
  - ▷ `double angularAcceleration` — поле для хранения углового ускорения
  - ▷ `Vector3 axis` — поле для хранения оси поворота
- `PathPointType` — класс перечислений, для пометки точек траектории прокладки. Содержит:
  - ▷ `DEFAULT` — тип для обозначения точки, после которой движение прямолинейное, равномерное.
  - ▷ `START_TURN` — тип для обозначения точки, после которой начинается маневр поворота

- `PathPoint` — структура для описания точки пути прокладки, является наследником класса `Point`. Состоит из:
  - ▷ `Vector3 position` — поле для хранения координат позиции точки
  - ▷ `double time` — момент времени, в который должен оказаться ЛА в заданной позиции
  - ▷ `PointType type` — поле для типа точки
  - ▷ `TurnData turnData` — поле для данные поворота.
- `Path` — класс для описания пути прокладки. Имеет следующие методы и поля:
  - ▷ `vector<PathPoint> path` — приватное поле для хранения вектора, состоящего из точек типа. `PathPoint`.
  - ▷ `PathPoint getPointForIndex(int ind)` — метод для получения точки пути по индексу.
  - ▷ `void addNewPoint(PathPoint p)` — метод для добавления точки в траекторию прокладки.
- `FVState` — структура для передачи состояния и короткого плана в радиоканал.
  - ▷ `Plane plane` — поле для записи текущего состояния
  - ▷ `vector<Point>` — поле для записи плана движения на следующие 4 точки пути.
- `AEtherInfo` — класс, который служит радиоканалом для всех ЛА.
  - ▷ `map<string,FVState> states` — поле куда ЛА записывают свои данные.
  - ▷ `void broadcastState(const Plane& plane)` — метод, принимающий текущее состояние ЛА и записывает в соответствующее поле радиоканала.
  - ▷ `void broadcastPlan(const string& name, const vector<Point>& shortPlan)` — метод, который принимает имя и короткий маршрутный план и записывает их соответствующее поле радиоканала.
- `GlobalSituation` — класс для хранения глобального состояния всего моделирования.
  - ▷ `vector<FV*> FVs` — поле для хранения ссылок на каждый ЛА, участвующий в моделировании.
  - ▷ `AEtherInfo aetherInfo` — поле для хранения радиоканала моделирования.
- `DataParser.h` — файл с методами для подгрузки входных данных по ЛА.
  - ▷ `void ParseSolveDataFromJSON(string json_name, double& time_step, double& integration_h, int& solve_part_count)` — метод для сбора данных для вычислений из файла.
  - ▷ `void ParsePathFromJSON(const json& path_data, vector<Point>& path);` — метод для сбора данных пути ЛА.
  - ▷ `void ParseFVToList(string json_name, GlobalSituation& gs);` — метод для сбора данных об ЛА из файла в список.
- `check` — класс предназначенный для сравнения по точности.
  - ▷ `static bool EQ(double value_1, double value_2);` — метод сравнения равенства по точности
  - ▷ `static bool NOTEQ(double value_1, double value_2);` — метод сравнения не равенства по точности
  - ▷ `static bool RE(double value_1, double value_2);` — метод сравнения больше по точности

- ▷ `static bool REQ(double value_1, double value_2);` — метод сравнения больше меньше по точности
- ▷ `static bool LE(double value_1, double value_2);` — метод сравнения меньше по точности
- ▷ `static bool LEQ(double value_1, double value_2);` — метод сравнения меньше равно по точности

- `Parameters` — структура для параметров, которые будут записаны в выходном файле, предназначенные для описания ЛА.
  - ▷ `string name;` — поле для ранения имени ЛА
  - ▷ `string type;` — поле для хранения типа ЛА.
  - ▷ `vector<Point> plan;` — поле для планового пути движения ЛА.
- `DynamicData` — структура для динамичных данных, которые будут записаны в выходной файл.
  - ▷ `double t` — поле для хранения времени
  - ▷ `double x` — поле для хранения координаты x
  - ▷ `double y` — поле для хранения координаты y
  - ▷ `double z` — поле для хранения координаты z
- `BroadcastData` — структура для данных, которые транслируются каждым ЛА в радио канал
  - ▷ `double t` — поле для хранения времени
  - ▷ `double x` — поле для хранения координаты x
  - ▷ `double y` — поле для хранения координаты y
  - ▷ `double z` — поле для хранения координаты z
  - ▷ `string type;` — поле для типа транслируемых данных
  - ▷ `vector<Point> plan;` — поле для короткого плана
- `OutputFV` — структура для описания данных вывода в файл описывающих ЛА
  - ▷ `Parameters parameters;` — поле для параметров
  - ▷ `vector<DynamicData> dynamic_data;` — поле для списка динамичских данных
  - ▷ `vector<BroadcastData> broadcasts;` — поле для списка тарнслируемых данных
- `OutputJsonData` — структура для описания выходных данных
  - ▷ `string version` — поле для версии файла
  - ▷ `vector<OutputFV> FVs` — поле для списка выходных данных по каждому ЛА.

- `void writeJsonData(const string& file_name, const OutputJsonData& output_json_data)` — метод для записи выходных данных в формате json.
- `Modeler` — класс для инкапсуляции всего процесса моделирования.
  - ▷ `void startModeling()` — метод для запуска процесса моделирования.
  - ▷ `double solve_time` — поле для хранения промежутка времени, на которое нужно делать следующее вычисление.
  - ▷ `double integral_h` — поле для хранения шага интегрирования.
  - ▷ `int solve_part_count` — поле для хранения количества вычислений.
  - ▷ `double start_time` — поле для хранения момента времени начала вычислений.
  - ▷ `double end_time` — поле для хранения момента времени конца вычислений.

▷ `GlobalSituation globalSituation` — поле для хранения глобальных данных моделирования.

- `void mergeShortPlans(const vector<Point>& arr1, const vector<Point>& arr2, vector<Point>& res1, vector<Point>& res2)` — метод нормирования сеток трналируемых планов движения в радио канал.

3.2 Входные данные

Входные данные программы представляют собой json файл следующей структуры:

```
{
  "timeStep": 1,
  "solve_part_count": 10,
  "AirCraftList": [
    {
      "type": "MaterialPoint",
      "fv_id": "first",
      "parameters": {
        "v_x": 0,
        "v_y": 0,
        "v_z": 0,
        "maxAcceleration": 1,
        "k_xz": -1,
        "k_y": -1,
        "broadcastStep": 2,
        "radiusFilter": 5,
        "heightWarn": 2,
        "radiusWarn": 3
      },
      "path": [
        {
          "time": 0,
          "x": 0,
          "y": 0,
          "z": 0
        },
        ...
      ]
    }
```

- `timeStep` — шаг вывода вычислений
- `solve_part_count` — кол-во вычислений итегрирования
- `AirCraftList` — список описаний ЛА.
  - ▷ `type` — тип ЛА.
  - ▷ `fv_id` — id ЛА.
  - ▷ `parameters` — параеметры ЛА.
    - ∗ `v_x` — координата вектора скорости по оси X
    - ∗ `v_y` — координата вектора скорости по оси Y
    - ∗ `v_z` — координата вектора скорости по оси z

        * `maxAcceleration` — максимальное ускорение
        * `k_xz` — коэффициент управления по xz
        * `k_y` — коэффициент управления по y
        * `broadcastStep` —шаг трансляции
        * `radiusFilter` — радиус фильтрации
        * `heightWarn` — высота защитного объема
        * `radiusWarn` — радиус защитного объема

    ▷ `path` — список точек пути

        * `time` — момоент времени
        * `x` — координата x
        * `y` — координата y
        * `z` — координата z

3.3 Выходные данные

Выходные данные программы представляют собой json файл следующей структуры:

```
{FVs: [
  {
    parameters : {
        name: ...,
        plane: [...],
        type: ...
    },
    dynamic_data : {
        {
            t: ...,
            x: ...,
            y: ...,
            z: ...,
        },
        ...
    },
    broadcasts : [
        {
            "plan": [...],
            "t": ...,
            "type": ...,
            "x": ...,
            "y": ...,
            "z": ...,
        },
        ...
    ]
  },
  { ... }
]}
```

- `parameters` — параеметры ЛА.

  - ▷ `name` — имя ЛА.
  - ▷ `plane` — плановый путь ЛА.
  - ▷ `type` — тип ЛА.
- `dynamicData` — список точек движения ЛА вычисленных на каждой итерации
  - ▷ `t` — момоент времени
  - ▷ `x` — координата x
  - ▷ `y` — координата y
  - ▷ `z` — координата z
- `broadcasts` — список сообщений ЛА в радиоканал.
  - ▷ `plane` — короткий плановый путь, на следующие 4 точки.
  - ▷ `t` — время вещания.
  - ▷ `type` — тип вещания, либо позиции либо короткого плана.
  - ▷ `x` — координата позиции x.
  - ▷ `y` — координата позиции y.
  - ▷ `z` — координата позиции z.

3.4 Реализация динамических моделей

Для реализации динамических моделей был создан базовый абстрактный класс FV. Он определяет базовые свойства присущие каждой модели движения, такие как:

- `virtual void next(double h, double end_time) = 0` — виртуальная функция, которая служит для вычисления следующего состояния БПЛА с шагом h до момента end_time.
- `virtual void setPath(vector<Point> path)` — виртуальная функция, которая служит для вычисления прокладки пути. По умолчанию вычисляется по схеме прокладки с упреждением поворота.
- `void doBroadcast()` — метод для трансляции данных в радио канал.
- `void checkConflict()` – метод для проверки конфликтов.
- `vector<Point> getBasePath()` — возвращает базовый путь, введенный пользователем.
- `Path getDynamicPath()` — возвращает построенный путь прокладки при задании пути.
- `vector<Point> basePath` — поле для хранения базового пути, который вводит пользователь.
- `Path turnPath` — поле для хранения, пути прокладки, заданной с помощью метода `setPath`.
- `Path dynamicPath` — поле для хранения динамически изменяемого пути `setPath`.
- `virtual double solveTurnRadius(const Vector3& v1, const Vector3& v2)` — метод, принимающий две скорости ЛА на соседних промежутках. Вычисляет радиус поворота.
- `double time` — поле для хранения времени текущего состояния БПЛА
- `double broadcastStep` — поле, в котором записан шаг вещания.
- `double nextBroadcastInstant` — поле для следующего момента времени вещания.
- `GlobalSituation globalSituation` — поле для хранения указателя на глобальное состояние.

Далее рассмотрим подробнее конкретные реализации динамических моделей — материальной точки и коптера, которые являются наследниками класса `FV`

3.4.1 Материальная точка

Класс материальной точки `MaterialPoint` является публичным наследником класса `FV`. А значит обладает всеми уже ранее описанными методами и полями, но также имеет и собственные. А именно:

- `string getName()` — метод возвращающий имя БПЛА.
- `int32_t getType()` — метод метод возвращающий тип БПЛА, у материальной точки 0.
- `Vector3 acceleration` — поле которое хранит ускорение в текущем состоянии.
- `Vector3 wishPosition` — поле хранящее желаемую позицию, вычисленную в методе `computeWishData`.
- `Vector3 wishVelocity` — поле хранящее желаемую скорость, вычисленную в методе `computeWishData`.
- `string name` — поле, содержащее имя БПЛА.
- `int32_t type` — поле, хранящее тип БПЛА.
- `Vector3* curPosition` — поле хранящее указатель на текущую позицию БПЛА.
- `Vector3* newPosition` — поле хранящее указатель на новую позицию, вычисленную в методе `next`.
- `Vector3* curVelocity` — поле хранящее указатель на текущую скорость БПЛА.
- `Vector3* newVelocity` поле хранящее указатель на новую скорость, вычисленную в методе `next`
- `double k_v` — поле хранящее значение коэффициента вклада позиции на управление.
- `double k_x` — поле хранящее значение коэффициента вклада скорости на управление.
- `double maxAcceleration` — поле хранящее значение максимального ускорения.
- `void computeWishData(double time_solve)` — метод предназначенный для вычисления желаемой позиции и желаемой скорости в момент времени `time_solve`

3.4.2 Коптер

Класс коптер `Copter`, также как и материальная точка является публичным наследником класса `FV`. А значит обладает всеми уже ранее описанными методами и полями, но также имеет и собственные, почти во всем перекликается с материальной точкой, но также имеет собственные особенности:

- `double inertialXZ` — поле для хранения коэффициента инерции в плоскости OXZ.
- `double inertialY` — поле для хранения коэффициента инерции по оси OY.

3.5 Вычисление «штурманской» прокладки

3.5.1 Разметка траектории

Разберем весь процесс вычисления, того как должен двигаться БПЛА. Начнем с метода `setPath` прописанного в базовом классе `FV`. Данный метод, как упоминалось ранее, принимает вектор из точек траектории пользователя, а затем пробегает по всем его точкам, проверяя отрезки на наличии поворотов. Если поворот нужен, то вместо точек стыка двух отрезок заменяют две точки, к которые являются касательными. Первая точка касания помечается типом `PointPathType::START_TURN` и для нее вычисляются все данные для поворота, указанные в структуре `TurnData`. После данного процесса полученный путь сохраняется в поле `Path dynamicPath`.

3.5.2 Вычисление прокладки

Метод `computeWishData` вычисляет `wishPosition, wishVelocity` желаемые позицию и скорость соответственно по моменту времени. Данный метод идентичен как в классе `MaterialPoint`, так и в классе `Copter`. В самом начале бинарным поиском происходит нахождение соответствующего отрезка в `Path dynamicPath`. Затем, относительного типа точки пути вычисляется желаемая скорость, а после желаемая позиция. Если точка имеет тип `DEFAULT`, то это значит что нужно двигаться равномерно, прямолинейно до следующей точки, если же тип `START_TURN`, то используя данные для поворота из соответствующего поля структуры, вычисляем скорость, находя касательный к окружности вектор, с помощью векторного произведения вектора угловой скорости и радиус вектора с началом в точке в центре окружности и концом в точке позиции БПЛА. Для вычисления новой позиции на окружности, последняя позиция двигается с помощью метода `rotate`, в котором реализована формула поворота.

3.5.3 Вычисление движения

Теперь перейдем к методу `next` для материальной точки. В данном методе происходит вычисление действительной позиции и скорости БПЛА посредством численного интегрирования. Для интегрирования используется метод Эйлера на левый край, где за производную управление берется регулятор, построенный в соответствующем пункте теоретической части. Регулятор используется один и тот же как для материальной точки:

```
for (int i = 0; i < 3; i++) {
    double x = (*curPosition)[i];
    double v = (*curVelocity)[i];
    (*newPosition)[i] = x + h * v;
    (*newVelocity)[i] = v + h * k_x * (x -  wishPosition[i]
                              + k_v * (v - wishVelocity[i]));
    acceleration[i] = k_x * (x - wishPosition[i])
                    + k_v * (v - wishVelocity[i]);
}
```

так и для коптера:

```
double x = (*curPosition)[0];
double v = (*curVelocity)[0];
(*newPosition)[0] = x + h * v;
(*newVelocity)[0] = v + h *
        ((k_v * wishVelocity.x + k_x * wishPosition.x) - v) / inertialXZ;
x = (*curPosition)[1];
v = (*curVelocity)[1];

(*newPosition)[1] = x + h * v;
(*newVelocity)[1] = v + h *
        ((k_v * wishVelocity.y + k_x * wishPosition.x) - v) / inertialY;

x = (*curPosition)[2];
v = (*curVelocity)[2];

(*newPosition)[2] = x + h * v;
```

```
(*newVelocity)[2] = v + h *
        ((k_v * wishVelocity.z + k_x * wishPosition.x) - v) / inertialXZ;
```

3.6 Реализация радиовещания

Для реализации радиовещания был создан класс `AetherInfo`.

```
class AEtherInfo
{
public:
    map<string,FVState> states;
    void broadcastState(double time,const Plane& plane);
    void broadcastPlan(const string& name,double time,
                       const vector<Point>& shortPlan);
};
```

Симуляцию радиоканала выполнял словарь `states`, где ключом является идентификатор БПЛА, а значением выступает специальная структура `FVState`.

```
struct FVState
{
    double planeTranslationTime;
    Plane plane;
    double translationTime;
    vector<Point> shortPlan;
};
```

У каждого ЛА есть указатель на радиоканал и метод `doBroadcast`.

```
void FV::doBroadcast()
{
    if ( nextBroadcastInstant <= time)
    {
            nextBroadcastInstant += broadcastStep;
            Plane plane = getPlane();
            globalSituation->aetherInfo.broadcastState(time,plane);
            nextBroadcastInstant += broadcastStep;
    }

    if (globalSituation->aetherInfo.states[this->name].shortPlan.empty() ||
        check::LEQ(globalSituation->aetherInfo.states[this->name].
            shortPlan[0].arrivalTime , time))
    {
            Plane plane = getPlane();
            vector<Point> short_plan;
            for (auto& p : dynamicPath)
            {
                    if (short_plan.size() > 3) break;
                    if (p.arrivalTime <= time + EPS) continue;
                    short_plan.push_back(Point(p.position, p.arrivalTime));
            }
```

```
            globalSituation->aetherInfo.
                broadcastPlan(plane.name, time, short_plan);
    }
}
```

В данном методе производятся проверки условий при которых нужно вызывать соответствующий метод отправки данных в радио канал. Для сообщения новой позиции каждый аппарат имеет встроенный временной период с которым требуется производить отправку, для сообщения "короткого плана"все немного сложнее. Отправка новых будущих позиций происходит, если ЛА достиг очередной точки.

3.7 Реализация обнаружения конфликтов

Обнаружение конфликтов была реализовано в методе `checkConflict`. Каждый БПЛА вызывает этот метод после произведения расчетов перемещения. Все проверки происходят в цикле `for` по каждому ЛА. Первым делом происходит проверка расстояния:

```
Plane plane = getPlane();
Vector3 pos = Vector3(plane.x, plane.y, plane.z);
Vector3 hisPos = Vector3(states[fv_name].plane.x,
                         states[fv_name].plane.y,
                         states[fv_name].plane.z);
Vector3 relPosition = hisPos - pos;
if (relPosition.norm() > radiusFilter) continue;
```

Затем проверка на сближение:

```
Vector3 vel = Vector3(plane.speedX, plane.speedY, plane.speedZ);
Vector3 hisVel = Vector3(states[fv_name].plane.speedX,
                         states[fv_name].plane.speedY,
                         states[fv_name].plane.speedZ);
if (scalarProduct(relPosition, hisVel - vel) < 0) continue;
```

Если определяется, что текущий ЛА сближается с рассматриваемым, то происходит вычисление промежутка конфликта для условий плоскости `OXZ` по всем промежуткам:

```
Vector3 my_vel = (my_plan[i + 1].position - my_plan[i].position) /
                 (my_plan[i + 1].arrivalTime - my_plan[i].arrivalTime);
Vector3 his_vel = (his_plan[i + 1].position - his_plan[i].position) /
                  (his_plan[i + 1].arrivalTime - his_plan[i].arrivalTime);

double A = (my_vel.x - his_vel.x)* (my_vel.x - his_vel.x) +
           (my_vel.z - his_vel.z) * (my_vel.z - his_vel.z);
double B = - 2 * (my_plan[i].arrivalTime * A +
                  ((my_plan[i].position.x - his_plan[i].position.x)+
                   (my_vel.x - hisVel.x)) *
             ((my_plan[i].position.z - his_plan[i].position.z) +
              (my_vel.z - hisVel.z)));

double C = -2 * my_plan[i].arrivalTime *
                ((my_plan[i].position.x - his_plan[i].position.x) +
                 (my_vel.x - hisVel.x)) *
```

```
                    ((my_plan[i].position.z - his_plan[i].position.z) +
                     (my_vel.z - hisVel.z)) +
                   my_plan[i].arrivalTime * my_plan[i].arrivalTime * A +
                   (my_plan[i].position.x - his_plan[i].position.x) *
                   (my_plan[i].position.z - his_plan[i].position.z) -
                   radiusWarn;

/// Находим корни
double D = B * B - 4 * A * C;
if (D >= 0)
{
    double t1 = (-B - sqrt(D)) / (2 * A);
    double t2 = (-B + sqrt(D)) / (2 * A);

    left_t_min_xz = min(t1,left_t_min_xz);
    right_t_max_xz = max(t2, right_t_max_xz);
}
```

Затем делаем то же самое но для оси Y:

```
// Проверка по Y
double Ay = (my_vel.y - his_vel.y) * (my_vel.y - his_vel.y);
double By = 2 * (my_vel.y - his_vel.y) *
                ((my_plan[i].position.y - his_plan[i].position.y) +
                my_plan[i].arrivalTime * (my_vel.y - his_vel.y));

double Cy = ((my_plan[i].position.y - his_plan[i].position.y) +
              my_plan[i].arrivalTime * (my_vel.y - his_vel.y)) *
              ((my_plan[i].position.y - his_plan[i].position.y) +
                my_plan[i].arrivalTime * (my_vel.y - his_vel.y))
            - heightWarn;

/// Находим корни
double Dy = By * By - 4 * Ay * Cy;
if (Dy >= 0) {
    double t1_y = (-By - sqrt(Dy)) / (2 * Ay);
    double t2_y = (-By + sqrt(Dy)) / (2 * Ay);

    left_t_min_y = min(t1_y, left_t_min_y);
    right_t_max_y = max(t2_y, right_t_max_y);
}
```

В самом конце делаем проверку, на пересечение, если пересечений нет, то значит конфликта тоже нет

```
if ((left_t_min_xz - right_t_max_y) *
    (left_t_min_y - right_t_max_xz) <= 0) return;
```

3.8 Результаты

В данном пункте пошагово пройдемся по этапам вычисления. Пусть пользователь задал такой маршрут состоящий из 4 точек:

- $(r_1,t_1) = ((0,0,0),0)$ — точка для нулевого момента времени,
- $(r_2,t_2) = ((10,0,0),10)$ — точка для 10 секунды,
- $(r_3,t_3) = ((10,0,-5),20)$ — точка для 20 секунды,
- $(r_3,t_3) = ((10,0,-15),30)$ — точка для 30 секунды

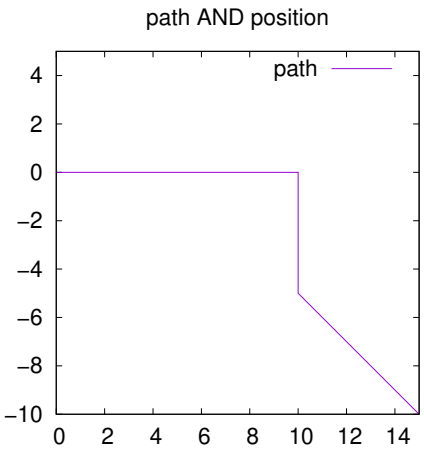

Рисунок 3.8.1 — Путь пользователя

Продемонстрирую поведение материальной точки стоящей в нулевой момент времени в точке(0,0,0), с начальной скорость (0,0,0) если использовать «Наивную» прокладку. Пусть коэффициенты вклада в управление для позиции и скорости соответственно $k_x = -1 =$, $k_v = -1$,

Как видно по зеленой линии, летательный аппарат не успевает изменить скорость на повороте. Теперь построим прокладку с упреждение поворота, см. рис. 3.8.3.

Посмотрим как будет себя вести БПЛА, нацелившийся на прокладку(рис. 3.8.4).

Как видно, результат стал намного лучше. Более лучше результата можно добиться подбором коэффициентов k_x и k_v, уменьшая или увеличивая вклад желаемых позиции и скорости.

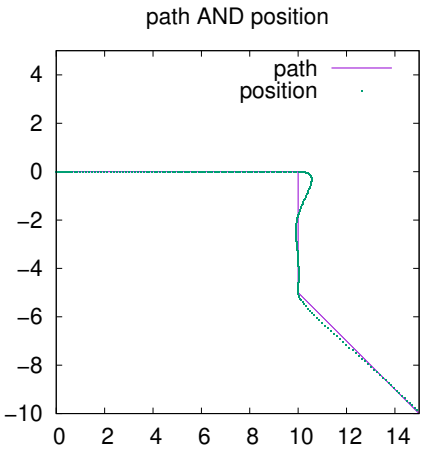

Рисунок 3.8.2 — Путь пользователя

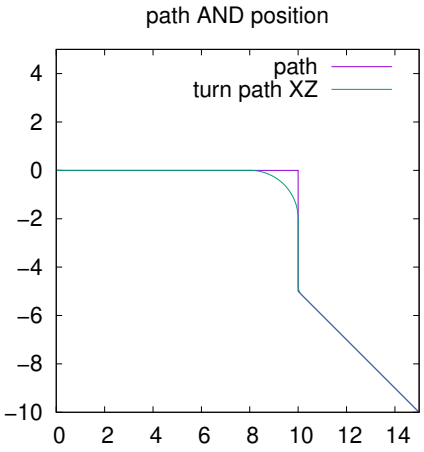

Рисунок 3.8.3 — Путь пользователя и прокладка

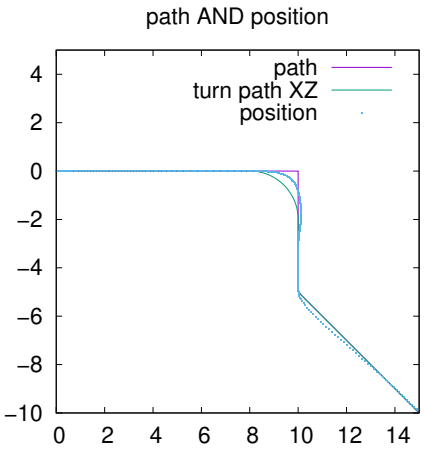

Рисунок 3.8.4 — Путь пользователя

ЗАКЛЮЧЕНИЕ

В работе описано построение процедур моделирования движения летательного аппарата вдоль маршрутов, представляющих собой ломаную линию с заданными моментами прибытия ЛА в точки излома. Движение ЛА описывается известных дифференциальных соотношений, включающих управляющие воздействия. Проход точек излома осуществляется прицеливанием постпредством пропорционального регулятора на точку, движущуюся по той или иной штурманской прокладке.

Разработанные процедуры реализованы в виде программного комлпекса на языке C++. В дальнейшем в этот комплекс можно встраивать и тестировать те или иные алгоритмы выработки манёвров уклонения с целью оценки их качества.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Шилдт Герберт C++ базовый курс [Электронный ресурс]: Белорусский государственный университет информатики и радиоэлектроники — Режим доступа: `https://www.bsuir.by/m/12_119786_1_98220.pdf` (дата обращения: 01.12.2023)

2. Документация по Microsoft C++, C и ассемблеру [Электронный ресурс]:– Режим доступа: `https://learn.microsoft.com/ru-ru/cpp/?view=msvc-170` (дата обращения: 01.12.2023).

3. ГОСТ 20058-80. Динамика летательных аппаратов в атмосфере. Термины, определения и обозначения. — М.: Госстандарт, 1980.