# ПРОЕКТ

по математическому моделированию

на тему «Робот-пылесос»

Выполнил:

Куташов Константин

ученик 11 класса

2024 г.

# Робот-пылесос с дифференциальным приводом

## Введение

Выбранная тема проекта: «Робот-пылесос»
При решении данного проекта я использовал материал из лекций, интернет-ресурсы и собственные вычисления.

## Описание

**Цель проекта** – разработка программы для моделирования и управления роботом-пылесосом с дифференциальным приводом, способным автономно перемещаться в помещении, избегая препятствия и выполняя уборку территории по алгоритму "змейка".

В реальных условиях данную задачу можно представить следующей легендой: требуется создать систему управления роботом-пылесосом, который должен эффективно очищать помещение, учитывая наличие статических препятствий и границ помещения.

**Входные данные:**

1. Размеры помещения (ширина и высота)
2. Параметры препятствий
3. Толщина стен помещения
4. Режим управления (автономный или ручной)

**Выходные данные:**

1. Визуализация движения робота в реальном времени
2. Траектория перемещения робота
3. Текущие параметры движения (скорости левого и правого колеса, угол поворота)

**Особенности реализации:**

1. Робот использует дифференциальный привод для движения
2. Реализованы два режима управления: автономный и ручной
3. В автономном режиме робот движется по алгоритму "змейка"
4. Визуальное отображение траектории движения

*Примечание*: в данной задаче не учитываются физические параметры уборки (всасывание пыли, влажная уборка и т.д.), основной фокус сделан на алгоритмах навигации и управления движением робота в пространстве с препятствиями.

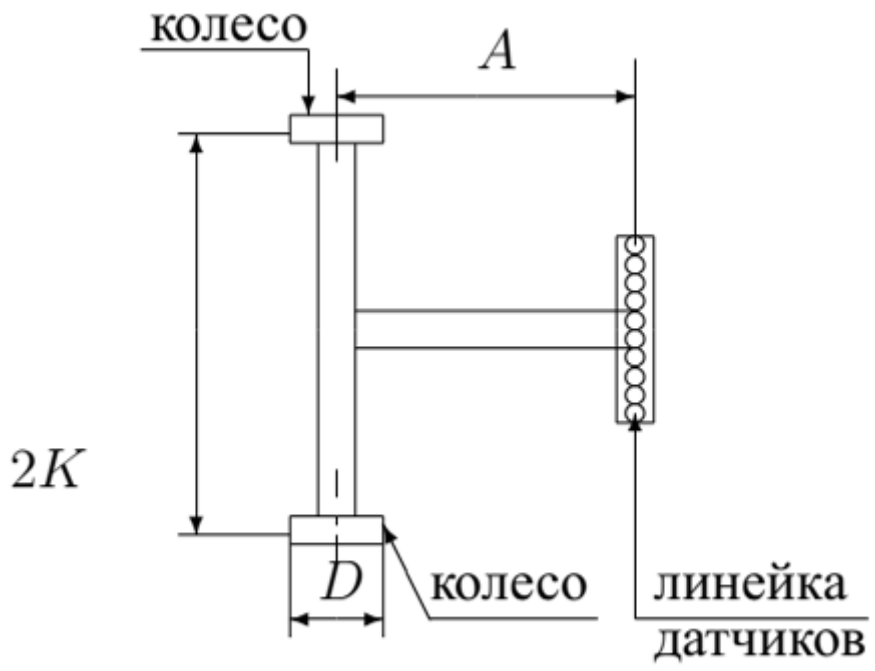
# Математическая модель

## Принцип работы робота

Конструкция робота основана на двухколёсной схеме, где каждое колесо является ведущим и управляется независимым двигателем. Это позволяет реализовать дифференциальное управление, при котором:

- Каждое колесо может вращаться с индивидуальной скоростью

- Направление движения определяется разницей скоростей вращения колес
- Поворот робота осуществляется за счет различных скоростей вращения правого и левого колеса

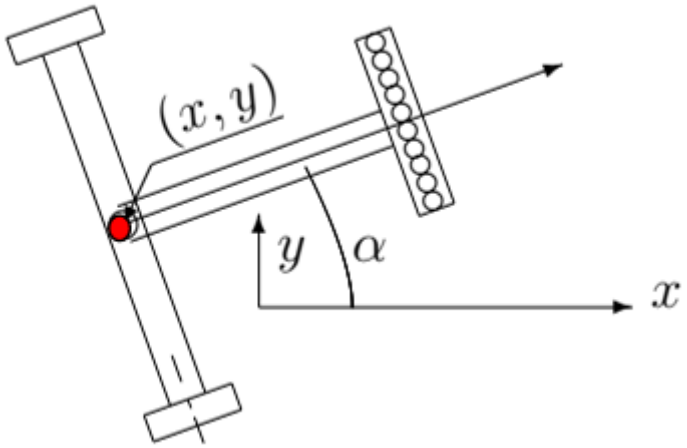



Основные геометрические параметры робота:

- 2K - колея робота (расстояние между центрами ведущих колес)
- D - диаметр ведущих колес
- A - расстояние от оси ведущих колес до линейки датчиков

Такая конфигурация обеспечивает высокую маневренность робота и позволяет ему выполнять сложные траектории движения, необходимые для эффективной навигации в пространстве с препятствиями. Дифференциальный привод также упрощает систему управления, так как все маневры осуществляются только за счет регулировки скоростей вращения двух ведущих колес.

# Координаты и положение робота




Для определения положения робота в пространстве используется система координат, где:

- Положение центра робота задается координатами $(x, y)$ на плоскости
- Ориентация робота определяется углом $\alpha$, который находится в диапазоне $[-\pi, +\pi]$
- Полное описание положения робота в любой момент времени задается тройкой координат $(x, y, \alpha)$

Важной особенностью кинематики робота является то, что его движение ограничено текущей ориентацией - робот может двигаться только в направлении, определяемом углом $\alpha$. При необходимости изменить направление движения роботу требуется совершить поворот, который не может быть мгновенным и требует прохождения некоторой траектории. Это ограничение связано с физическими особенностями дифференциального привода и является ключевым фактором при планировании движения робота.

# Траектория движения робота

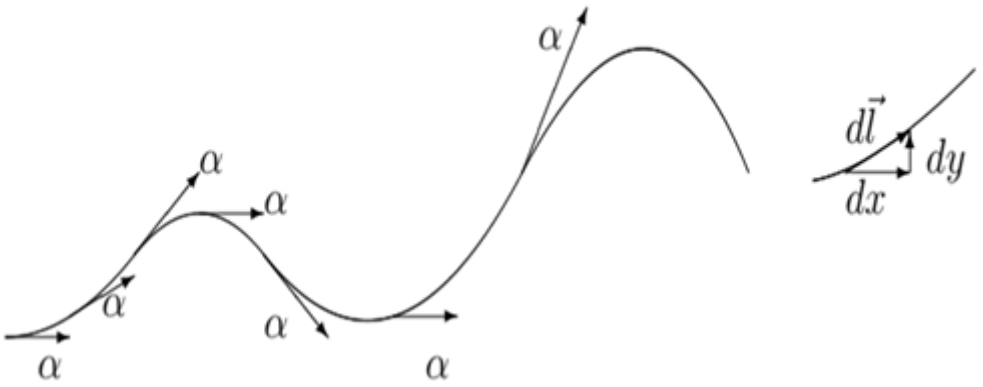
В процессе движения робота его позиция и ориентация постоянно изменяются во времени, формируя траекторию движения. Математически эта траектория описывается как кривая в трехмерном пространстве, где:

- Координаты центра робота: $(x(t), y(t))$
- Угол ориентации: $\alpha(t)$
- Параметр $t$ представляет время движения

Для расчета длины "малого участка" траектории $dl = (dx, dy)$ используется следующая формула:

$$|dl|^2 = |dx|^2 + |dy|^2$$

где $dx$ и $dy$ - малые изменения координат по соответствующим осям. Интегрируя эти элементарные отрезки вдоль всей траектории, можно получить полную длину пути, пройденного роботом.




## Прямая задача

Прямая задача заключается в вычислении положения робота в следующий момент времени $t_{n+1}$, исходя из его текущего положения, скорости колес и угла ориентации. Это задача, в

которой мы рассчитываем координаты робота $x_{new}$ и $y_{new}$ по известным значениям угловых и линейных скоростей колес.

**Формулировка:**
Дано начальное положение робота $x_n$, $y_n$ и угол ориентации $\theta_n$, а также линейные скорости колес $v_l$ и $v_r$, угол ориентации $\theta_n$, и шаг времени $dt$. Необходимо вычислить новые координаты робота $x_{new}$ и $y_{new}$ через шаг времени $dt$.

**Решение:**
Используя разностные уравнения для обновления координат робота, получаем:

$$x_{new} = x_n + \left( \frac{v_l + v_r}{2} \cdot \cos(\theta_n) \right) \cdot dt$$

$$y_{new} = y_n - \left( \frac{v_l + v_r}{2} \cdot \sin(\theta_n) \right) \cdot dt$$

## Обратная задача

Обратная задача заключается в вычислении линейных скоростей $v_l$ и $v_r$ (или угловых скоростей $\omega_L$ и $\omega_R$) на основе желаемого изменения положения робота. То есть, если мы знаем конечные координаты и ориентацию робота, а также его начальные параметры, необходимо определить, какие скорости колес нужно задать, чтобы достичь желаемого положения.

**Формулировка:**
Дано начальное положение робота $x_n$, $y_n$, угол ориентации $\theta_n$, а также конечное положение $x_{new}$, $y_{new}$ и ориентация $\theta_{new}$. Необходимо найти линейные скорости колес $v_l$ и $v_r$, которые

обеспечат перемещение робота от начальных координат к конечным.

**Решение:**
Для решения обратной задачи нужно использовать кинематические уравнения для движения робота и подходящие методы оптимизации, такие как метод Ньютона или другие численные методы, для нахождения скоростей колес $v_l$ и $v_r$, которые приводят к нужному изменению положения робота за определенное время.

Математически, задача сводится к решению системы уравнений:

$$x_{new} = x_n + \left( \frac{v_l + v_r}{2} \cdot \cos(\theta_n) \right) \cdot dt$$

$$y_{new} = y_n - \left( \frac{v_l + v_r}{2} \cdot \sin(\theta_n) \right) \cdot dt$$

Где нужно найти $v_l$ и $v_r$, чтобы значения $x_{new}$ и $y_{new}$ совпали с заданными.

Эти задачи представляют собой типичные задачи управления движением робота в контексте дифференциальных уравнений и кинематики.

## Уравнения движения робота

В результате применения ряда математических преобразований, включая дифференцирование, исключение мгновенного радиуса поворота и использование разностных уравнений для дискретного времени, мы получаем следующие обновления координат робота. Начнем с уравнений для линейных скоростей колес и угловой скорости робота, затем

перейдем к разностным формулам для координат в моменты времени $t_n = n \cdot \delta$ (где $\delta$ — шаг дискретизации).

Из уравнений кинематики получаем скорость движения робота, учитывая его угловые скорости колес $\omega_L$ и $\omega_R$, а также радиус колес $D$ и расстояние между осями $K$. После исключения мгновенного радиуса кривизны $R$, а также применения разностных уравнений, мы получаем следующие уравнения для обновления координат робота:

$$x_{new} = x + \left( \frac{v_l + v_r}{2} \cdot \cos(\theta) \right) \cdot dt$$

$$y_{new} = y - \left( \frac{v_l + v_r}{2} \cdot \sin(\theta) \right) \cdot dt$$

Здесь $v_l$ и $v_r$ — это линейные скорости левого и правого колеса, а $\theta$ — угол ориентации робота относительно оси $x$. Эти уравнения описывают изменение координат $x$ и $y$ робота на каждом шаге дискретизации $dt$, исходя из текущих значений скоростей колес и угла ориентации.

# Программная реализация

## Структура проекта

### 1. Класс `Envir` (Окружение)

**Назначение:** Управляет основными параметрами окружения, такими как размеры окна, цвета, инициализация Pygame и визуальные элементы.

**Основные переменные:**

- `width`, `height` — размеры окна.

- `color_background`, `color_robot` — цвета фона и робота.
- `win` — основное окно отображения.

**Функции:**

- `draw()` — отрисовка всех элементов на экране.

---

## 2. Класс `Robot` (Робот)

**Назначение:** Представляет самого робота, управляющего движением, положением, углом поворота, а также сбором данных о состоянии окружения.

**Основные переменные:**

- `x`, `y` — координаты робота.
- `angle` — текущий угол поворота.
- `speed` — скорость движения.
- `sensor_data` — данные с сенсоров.

**Функции:**

- `move()` — управление движением робота.
- `rotate()` — управление поворотом.
- `update_sensor_data()` — обновление данных с сенсоров.

---

## 3. Класс `ObstacleManager` (Управление препятствиями)

**Назначение:** Обрабатывает добавление и управление препятствиями на поле.

**Основные переменные:**

- `obstacles` — список всех препятствий.

**Функции:**

- `add_obstacle()` — добавление нового препятствия.
- `check_collision()` — проверка столкновений робота с препятствиями.

---

### 4. Класс `Obstacle` (Препятствие)

**Назначение:** Описывает отдельное препятствие, его координаты, размеры и способ отрисовки.

**Основные переменные:**

- `x`, `y` — координаты препятствия.
- `width`, `height` — размеры препятствия.

**Функции:**

- `draw()` — отрисовка препятствия на экране.

---

## Алгоритмы

### 1. Движение змейкой

**Логическое обоснование:** Робот должен корректировать своё движение при приближении к стенам и углам, чтобы избежать столкновений и обеспечивать полный охват области.

**Реализация:**

```
def move_autonomous(self, dt, environment):
    # Базовые параметры движения
    base_speed = 0.08 * self.m2p
    turn_speed = 0.04 * self.m2p

    # Расчет нового положения
    new_x = self.x + ((self.vl + self.vr) / 2) *
math.cos(self.theta) * dt
    new_y = self.y - ((self.vl + self.vr) / 2) *
math.sin(self.theta) * dt

    if new_x + 50 > environment.width -
environment.wall_thickness:
        self.turn_right()
        for _ in range(2):
            if int(math.degrees(self.theta)) > -180 and
self.is_turning == False:
                self.turn_right()
            else:
                break

    if new_x - 50 < environment.wall_thickness and
self.count != 0:
        self.turn_left()
        for _ in range(2):
            if int(math.degrees(self.theta)) < 0 and
self.is_turning == False:
                self.turn_left()
```

```
            else:
                break
```

**Алгоритмическое описание:**

- Рассчитывается новое положение робота на основе текущей скорости и угла поворота.
- Если робот приближается к правой стене, выполняется правый поворот.
- Если робот приближается к левой стене и уже не на первой линии движения, выполняется левый поворот.
- Двойная проверка углов предотвращает проскальзывание мимо стен при резких поворотах.

## 2. Обход стен

### 1)Поворот направо

**Логическое объяснение**:
Этот код описывает функцию `turn_right`, предназначенную для выполнения роботом поворота направо. Главная цель функции — постепенно изменить угол ориентации робота ($\theta$), корректируя положение его изображения, пока угол не достигнет заданного значения (например, $-180^\circ$). Основные этапы:

1. **Инициализация поворота**:
    - Устанавливается флаг `is_turning = True`, чтобы сигнализировать о начале поворота.
    - Начальные скорости колес задаются так, чтобы робот начал вращаться на месте.
2. **Цикл поворота**:

- Угол $\theta$ обновляется на каждом шаге пропорционально разнице скоростей колес и времени шага симуляции $dt$.
- После каждого изменения угла изображение робота обновляется для визуализации поворота.

3. **Завершение поворота**:
   - Если угол достигает $-180°$, происходит завершение поворота:
     - Ускорения устанавливаются в значения для прямолинейного движения.
     - Угол фиксируется на $-\pi$ (эквивалентно $-180°$).
     - Увеличивается счётчик поворотов (`count`), если нужно отслеживать количество выполненных разворотов.
   - Флаг `is_turning` сбрасывается в `False`, обозначая завершение поворота.

**Алгоритмическое объяснение**:

1. Установка флага:

```
self.is_turning = True
```

- Переменная `is_turning` устанавливается в `True`, что сигнализирует о начале маневра.

2. Задание скоростей колес:

```
self.vr = 0
self.vl = 0.04 * self.m2p
```

- Остановка правого колеса $v_r = 0$ и вращение левого колеса $v_l = 0.04 \cdot m2p$ создают разницу в скоростях, необходимую для поворота направо.

3. Обновление угла:

```
self.theta += ((self.vr - self.vl) / self.w * dt) / 2
```

- Здесь:
  - $v_r$ — скорость правого колеса.
  - $v_l$ — скорость левого колеса.
  - $w$ — ширина робота.
  - $dt$ — временной шаг.
  - Деление на 2 снижает скорость изменения угла, делая анимацию плавнее.

4. Обновление изображения:

```
self.rotated = pygame.transform.rotozoom(self.img,
math.degrees(self.theta), 1)
self.rect = self.rotated.get_rect(center=(self.x,
self.y))
```

- `rotozoom` поворачивает изображение на угол $\theta$ в градусах.
- `get_rect` корректирует его положение на экране.

5. Проверка достижения угла:

```
if int(math.degrees(self.theta)) <= -180
```

- Проверяется, достиг ли робот угла $-180^\circ$ (эквивалент $-\pi$ в радианах).

- Использование `int()` помогает избежать проблем с погрешностями вычислений.

6. Завершение поворота:

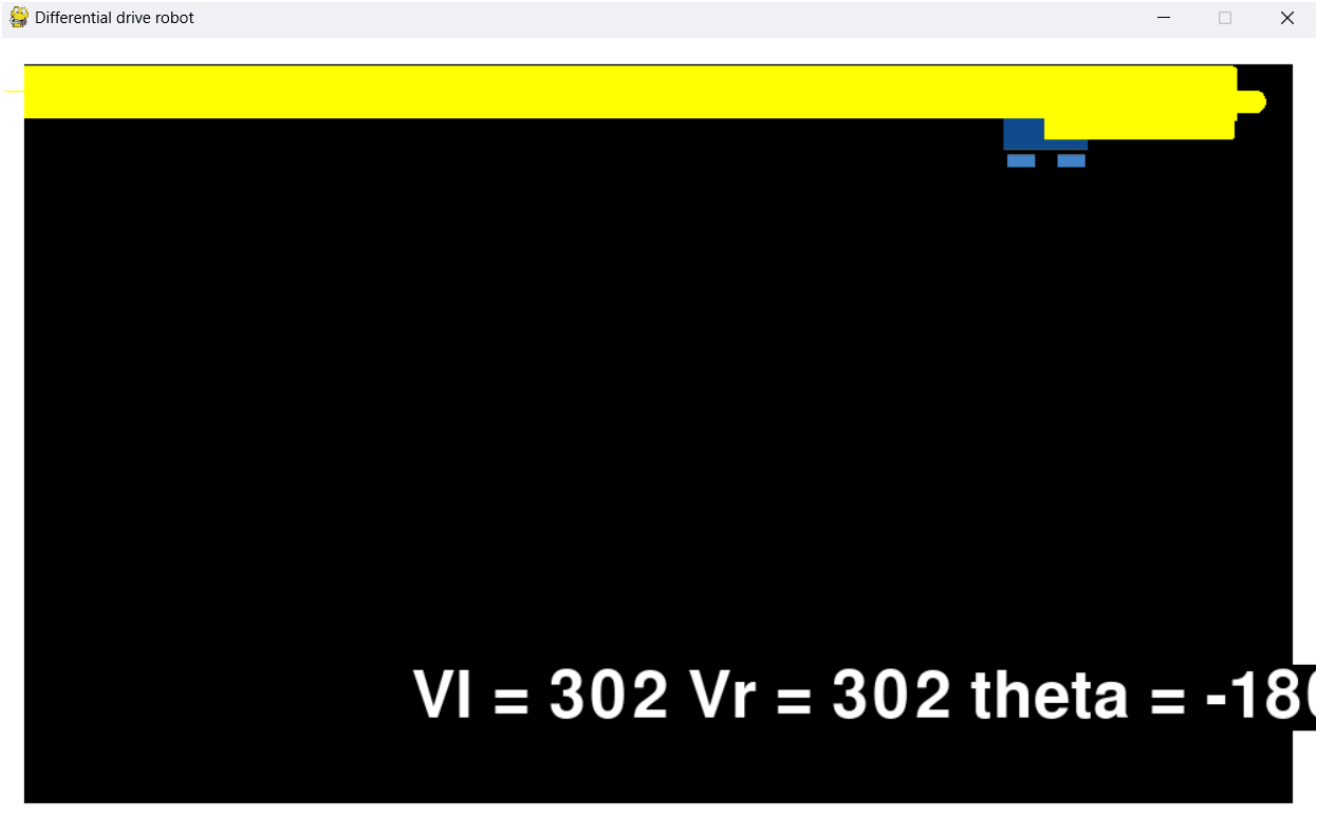
```
self.vr = 0.08 * self.m2p
self.vl = 0.08 * self.m2p
self.count += 1
self.theta = -\pi
```

- Обновляются скорости для движения вперед, счётчик увеличивается, а угол фиксируется на $-\pi$, чтобы избежать ошибок округления.

7. Сброс флага:

```
self.is_turning = False
```

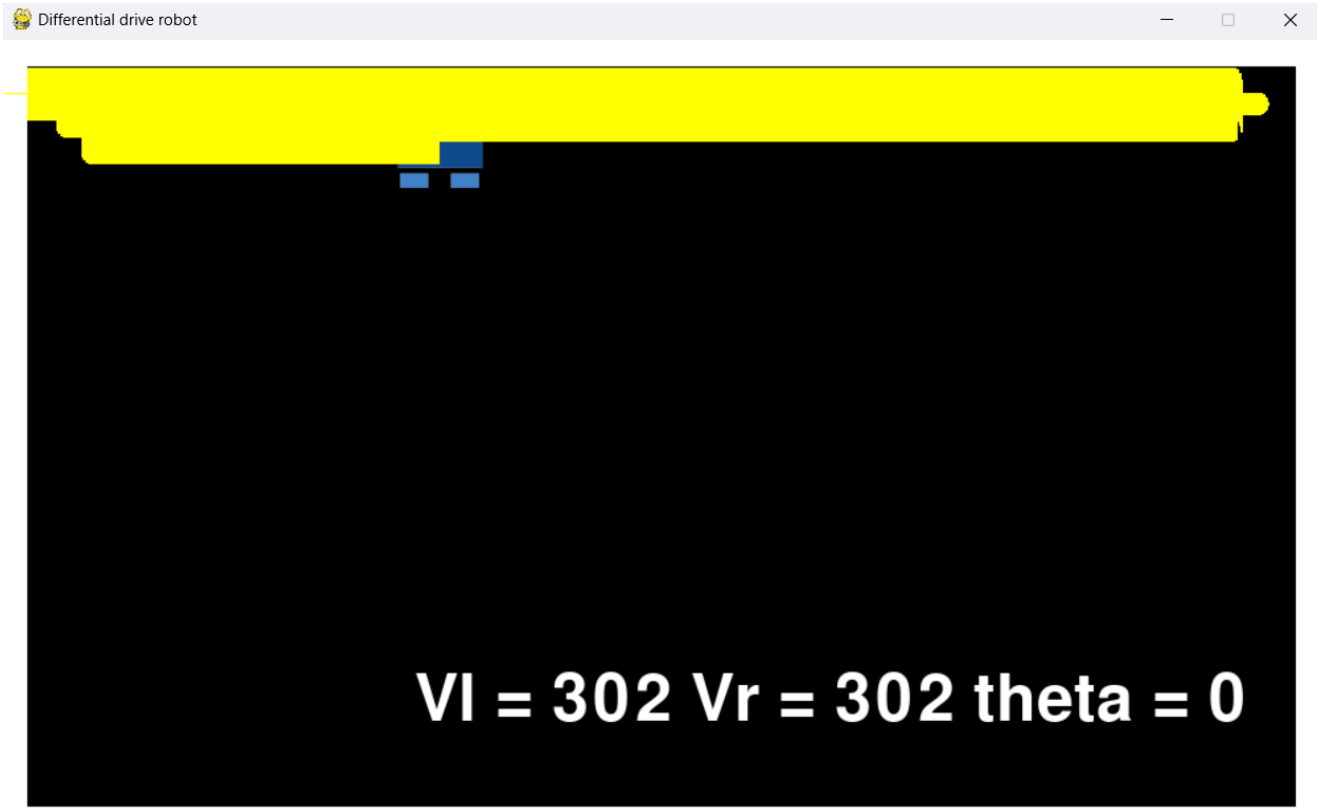
- Флаг сбрасывается, что означает завершение маневра.



**2)Поворот налево**

**Логическое описание**:
Функция `turn_left` выполняет поворот робота налево, симметричный `turn_right`, но с противоположными скоростями колес:

- **Инициализация**:
  Устанавливается флаг `is_turning = True`, скорости колес задаются так, чтобы левое колесо останавливалось, а правое вращалось.
- **Поворот**:
  Угол $\theta$ обновляется на основе разности скоростей колес:
  $\theta += (vr - vl)w \cdot dt \cdot 12\theta += \frac{(v_r - v_l)}{w} \cdot dt \cdot \frac{1}{2}$
  Обновляется изображение робота через `rotozoom`.
- **Завершение поворота**:
  Когда угол достигает $0^\circ$, скорости колес устанавливаются для прямолинейного движения, угол фиксируется на $\pi$, увеличивается счётчик поворотов `count`, сбрасывается флаг `is_turning`.




## 3. Обработка препятствий с использованием метода лидаров

**Лидары** (LiDAR - Light Detection and Ranging) — это устройства, использующие лазерные лучи для определения расстояния до объектов. Они испускают импульсы света, измеряют время возврата отражённых лучей и создают трёхмерные карты окружающей среды. Лидары широко применяются в робототехнике, автономных автомобилях и навигации.


| Расстояние до цели | 1 м | 10 м | 100 м | 1 км | 10 км | 100 км |
|---|---|---|---|---|---|---|
| Время отклика | 6.7 нс | 67 нс | 0.67 мкс | 6.7 мкс | 67 мкс | 0.67 мс |

**Как работает лидар:**

1. Лазер испускает световой импульс.
2. Свет отражается от объекта.

3. Время возврата луча позволяет вычислить расстояние до объекта.

**Реализация в проекте:**
В моём проекте лидарная система реализована через простые условия `if`, поскольку не хватило времени на полноценную разработку. К сожалению, мне не удалось создать алгоритм обхода препятствий, и робот ведет себя странно.

# Ручное управление роботом

Этот фрагмент кода реализует функцию движения робота в двумерном пространстве с использованием библиотеки Pygame и управления через клавиши на клавиатуре(NumPad). В частности, код управляет скоростью колес робота, рассчитывает его новое положение и проверяет на столкновение с препятствиями. Давайте разберем код пошагово:

**Логическое объяснение:**

1. **Получение состояния клавиш:**
    - `keys = pygame.key.get_pressed()` — этот вызов получает текущее состояние всех клавиш на клавиатуре, возвращая массив, в котором для каждой клавиши указано, нажата она или нет. Важно отметить, что этот массив обновляется постоянно, и мы можем отслеживать каждое изменение клавиш в реальном времени.
2. **Изменение скорости колес:**
    - Проверяются клавиши на цифровой клавиатуре:

- `pygame.K_KP4` (клавиша 4 на NumPad) — увеличение скорости левого колеса ( `vl` ).
- `pygame.K_KP1` (клавиша 1 на NumPad) — уменьшение скорости левого колеса.
- `pygame.K_KP6` (клавиша 6 на NumPad) — увеличение скорости правого колеса ( `vr` ).
- `pygame.K_KP3` (клавиша 3 на NumPad) — уменьшение скорости правого колеса.

- Если соответствующая клавиша нажата, скорость левого или правого колеса изменяется на некоторое значение, пропорциональное времени (умноженное на `self.m2p` , который может быть коэффициентом преобразования для правильных единиц измерений).

3. **Расчет нового положения робота:**
   - Позиция робота рассчитывается с учетом средней скорости обоих колес ( `(self.vl + self.vr) / 2` ) и направления движения, которое определяется углом `theta` :
     - `new_x = self.x + ((self.vl + self.vr) / 2) * math.cos(self.theta) * dt` — новая координата по оси X.
     - `new_y = self.y - ((self.vl + self.vr) / 2) * math.sin(self.theta) * dt` — новая координата по оси Y.
     - `dt` — временной шаг, который определяет, как далеко перемещается робот за одно обновление (это должно быть установлено где-то в коде).
4. **Проверка столкновений:**
   - Перед тем как обновить положение робота, происходит проверка, нет ли на пути робота

препятствий:

- `if environment and not self.check_wall_collision_manual(new_x, new_y, environment):` — если новое положение не вызывает столкновение с препятствием (проверяется методом `check_wall_collision_manual`), то координаты обновляются.
- Если столкновение найдено, скорость колес обоих направлений (`vl` и `vr`) обнуляется, чтобы робот остановился.

5. **Коррекция угла ориентации робота:**
   - Угол поворота робота `theta` изменяется в зависимости от разницы скоростей колес:
     - `self.theta += (self.vr - self.vl) / self.w * dt` — это моделирует вращение робота вокруг его центра (где `w` — расстояние между колесами). Разница в скорости колес вызывает поворот робота.
6. **Ограничение угла ориентации:**
   - Если угол ориентации (`self.theta`) выходит за пределы от `-2π` до `2π`, его значение сбрасывается в ноль, чтобы избежать переполнения угла и неудобных значений при вычислениях.
7. **Ограничение максимальной и минимальной скорости:**
   - Скорость колес ограничивается максимальным (`self.maxspeed`) и минимальным (`self.minspeed`) значением:
     - `self.vr = min(self.vr, self.maxspeed)` — ограничивает скорость правого колеса.

        - `self.vr = max(self.vr, self.minspeed)` — ограничивает минимальную скорость.
        - Точно так же для левого колеса.
8. **Обновление изображения робота:**
    - Изображение робота поворачивается в зависимости от его угла ориентации:
        - `self.rotated = pygame.transform.rotozoom(self.img, math.degrees(self.theta), 1)` — изображение робота поворачивается на угол, равный `theta`.
        - `self.rect = self.rotated.get_rect(center=(self.x, self.y))` — создается прямоугольник, ограничивающий изображение, с центром в текущих координатах робота.

**Алгоритмическое объяснение:**

1. Получаем состояние клавиш.
2. Если нажаты клавиши для изменения скорости, увеличиваем или уменьшаем скорость соответствующих колес.
3. Рассчитываем новое положение робота на основе текущей скорости и направления (с учетом времени).
4. Проверяем, нет ли столкновений с окружающими объектами.
5. Если столкновения нет, обновляем положение робота, учитывая вращение.
6. Ограничиваем значения угла, чтобы он не выходил за пределы допустимых.
7. Ограничиваем скорость колес в пределах заданного диапазона.

8. Поворачиваем изображение робота в соответствии с его углом ориентации.
9. Обновляем координаты изображения робота на экране.

## Визуализация

Система визуализации включает:

1. Основные элементы:
    - Отображение робота и его ориентации
    - Визуализация препятствий
    - Отрисовка траектории движения
    - Информационная панель с параметрами
2. Цветовая схема:
    - Черный фон
    - Белые стены
    - Красные препятствия
    - Желтая траектория движения
3. Информационное табло:
    - Скорости левого и правого колеса
    - Текущий угол поворота

## Результаты

Проект оказался намного сложнее, чем я ожидал. С самого начала я столкнулся с разными трудностями, которые забрали больше времени, чем я планировал. Я понял, что для полноценной реализации всех идей нужно гораздо больше усилий и внимания к деталям. К сожалению, из-за нехватки времени я не смог до конца отладить все баги и исправить мелкие неточности.

Тем не менее, мне удалось реализовать основные части проекта: робот, который двигается по комнате "змейкой", очищая пространство, а также система, которая проверяет столкновения со стенами. Я научился управлять роботом через клавиатуру, регулируя скорость и направление движения, а также работать с окружением робота, чтобы он корректировал свою траекторию.

В процессе работы я понял, что важно учитывать все возможные ситуации, даже те, которые случаются редко. Кроме того, я научился внедрять реальную модель физического объекта — робот, который использует математические формулы для своего движения. Это позволило мне приблизиться к настоящей симуляции робота, где учтены реальные физические законы, такие как расчёт скорости, угла поворота и столкновений.

Во время разработки я столкнулся с проблемой округления угла робота, из-за чего он иногда мог двигаться немного криво. Однако в большинстве случаев, когда робот доходит до противоположной стены, ширина его покрытия компенсирует этот "кривой участок", где он мог оставить незакрашенное пространство.

Я также узнал много нового о роботах, алгоритмах движения и том, как важно тщательно тестировать проект на каждом шаге, чтобы избежать ошибок. Несмотря на то что проект не был до конца завершён, я научился многому и теперь знаю, как улучшить его в будущем.

ССЫЛКА НА ПРОГРАММУ И КАРТИНКУ РОБОТА:
https://disk.yandex.ru/d/jHpuNn_is2B67w