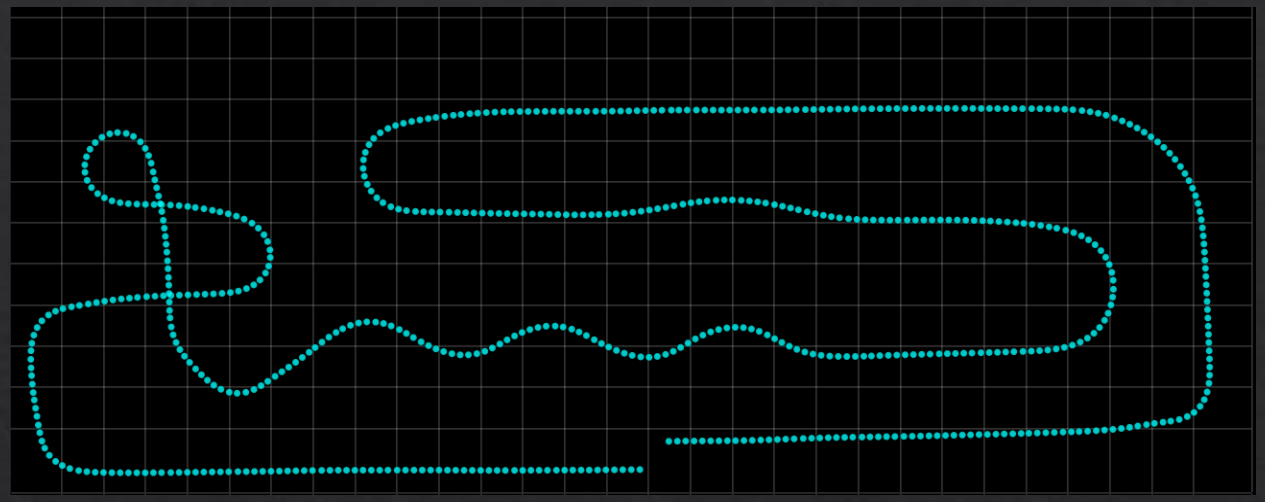
FULL SHORTCUT DRIVE

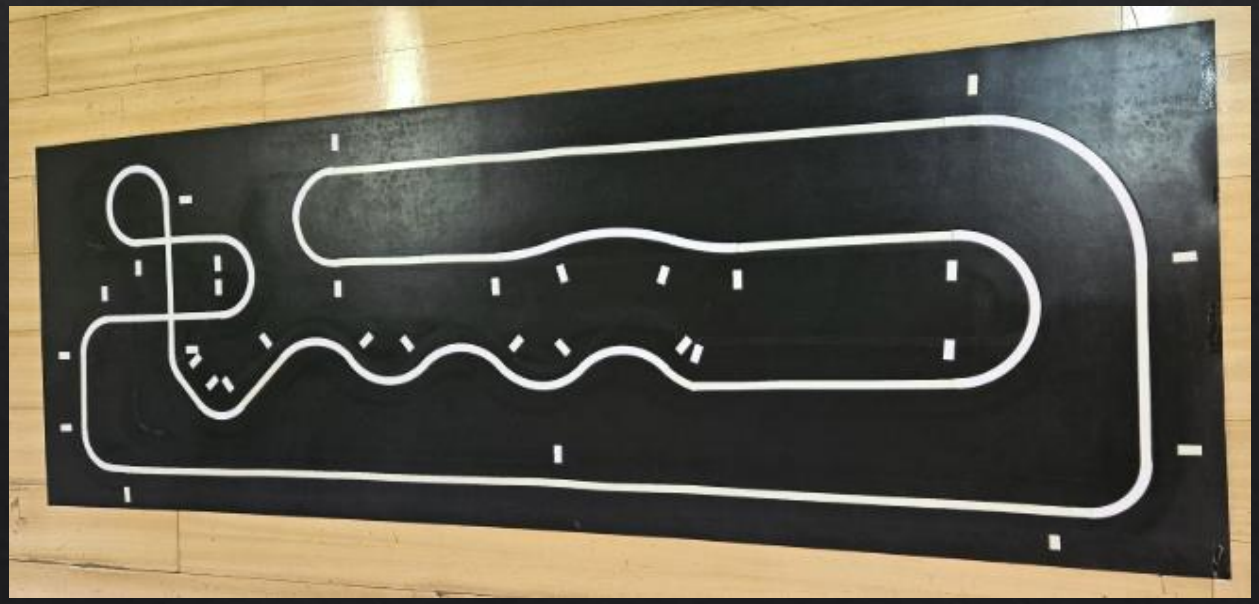
Seguindo Linhas Virtuais

O QUE É?

Time Raijū

MAPEAMENTO



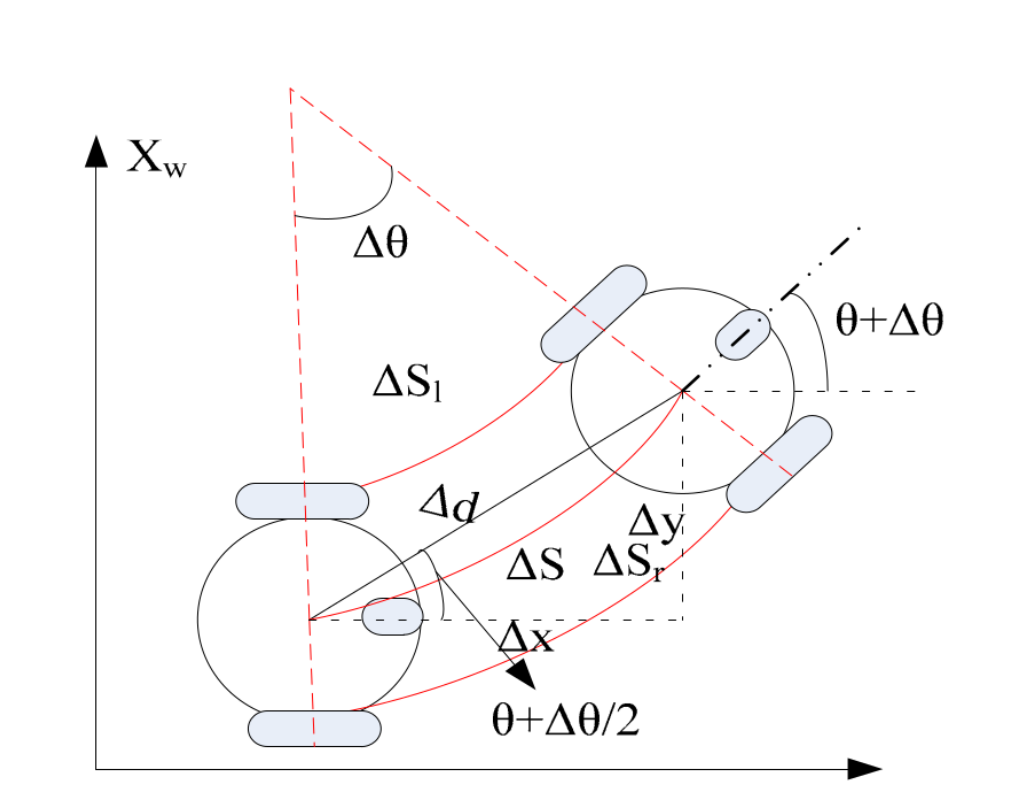


Requisitos Básicos

- IMU e Encoder
- Coletar os dados do robô (USB)
- Visualizar os dados coletados
- Armazenar na memória não volátil
    - Um ponto a cada 2 cm + tamanho máximo da pista 60m + 4 bytes por ponto => 12000 bytes por mapa
    - Difícil ler em tempo real da memória externa. Espaço na RAM também é necessário

map0_15_default.txt

```
2.096867,-0.004748
4.155768,-0.023081
6.18377,-0.040305
8.178827,-0.061489
10.221764,-0.081166
12.201908,-0.097587
14.180023,-0.11764
16.127325,-0.122903
18.108509,-0.133132
20.061764,-0.145623
22.015041,-0.153307
24.156734,-0.167801
26.142929,-0.170228
28.098164,-0.184034
30.059397,-0.195482
32.006702,-0.199412
34.204262,-0.208177
36.164539,-0.207041
38.124802,-0.212801
40.096001,-0.20261
```



Differential wheeled robot

$$\Delta S = \frac{\Delta S_l + \Delta S_r}{2}$$

$$\Delta\theta = \theta_{atual} - \theta_{anterior}$$

$$x = x_{anterior} + \Delta S * \cos(\theta + \Delta\theta/2)$$

$$y = y_{anterior} + \Delta S * \sin(\theta + \Delta\theta/2)$$

$$\theta = \theta_{atual}$$

θ = Ângulo atual do robô medido pelo IMU

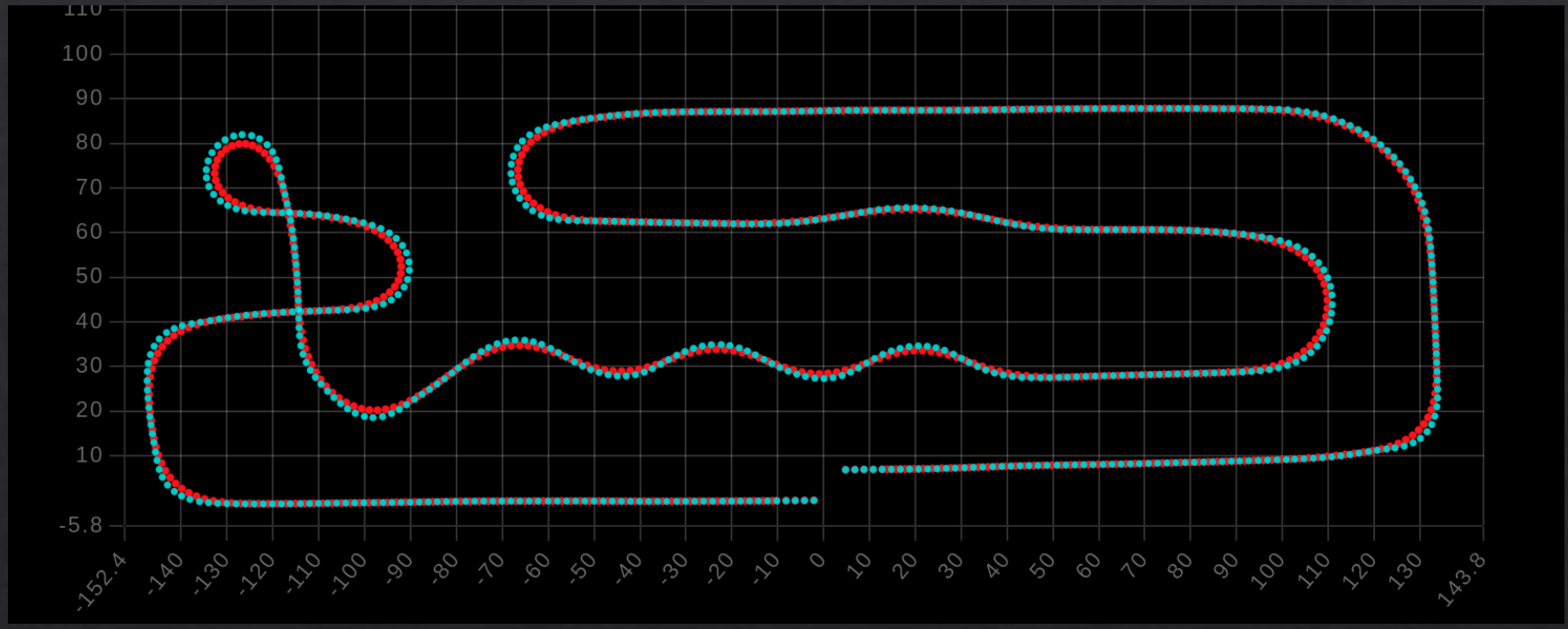
(x,y) = Posição absoluta do robô

ΔS = Deslocamento do robô entre dois loops de controle

```
    auto navigation = services::Navigation::instance();
    float traveled_dist_cm = navigation->get_traveled_distance() / 10.0;

    if (traveled_dist_cm > (waypoint_idx + 1) * WAYPOINT_DISTANCE_CM) {
        Point pos = navigation->get_robot_position_cm();
        float current_angle = navigation->get_robot_angle();

        map_list_cm[waypoint_idx] = pos;

        last_traveled_dist_cm = traveled_dist_cm;
        waypoint_idx++;
        map_list_length = waypoint_idx;
    }
}
```

- Cuidado com overflow de ângulos e unidades
- Float já é suficiente, e garanta que seu microcontrolador da conta de tudo em 1ms
- Frequência de amostragem do IMU está adequada?
- IMU está calibrado?
- Encoder tem precisão o suficiente?
- Escala do IMU está adequada?

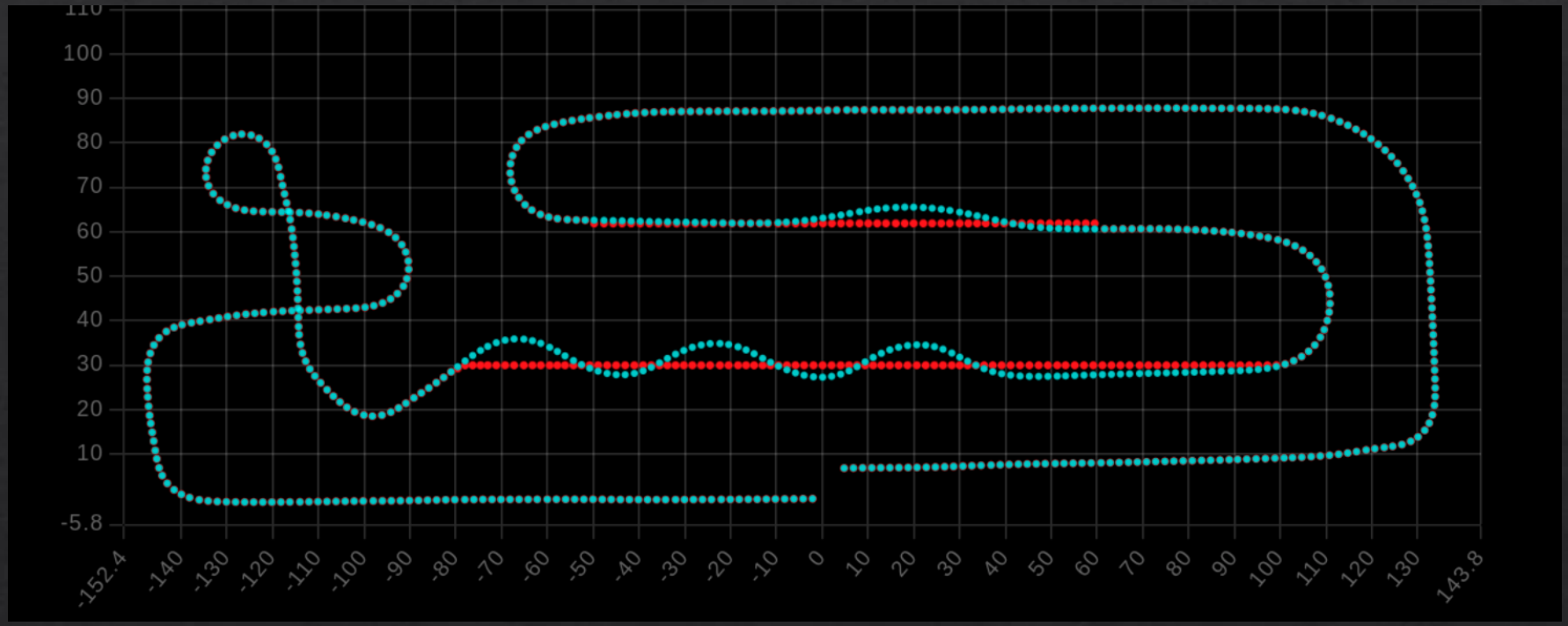
GERAÇÃO DE TRAJETO



$$\bar{x}_i = \frac{1}{N} \sum_{j=i}^{i+N} x_j$$

$$\bar{y}_i = \frac{1}{N} \sum_{j=i}^{i+N} y_j$$

- Média móvel é a maneira mais simples
- Quanto maior o "N", maior o fator "shortcut"
- Esse método falha um pouco em curvas R10 e mais de 90°



- No brasil não é tanto problema, porque temos tempo de ajustar manualmente
- Mas é bom pensar para quando for competir internacionalmente, ou mesmo para evitar trabalho manual
- Para conseguir fazer isso, é legal ter uma funcionalidade de “upload” de dados no robô. Fazemos isso via USB

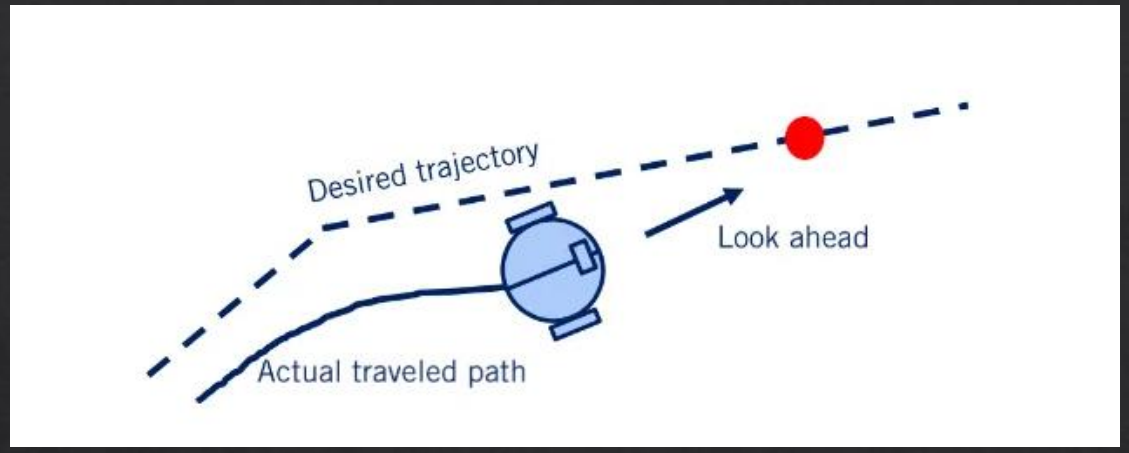
CONTROLE

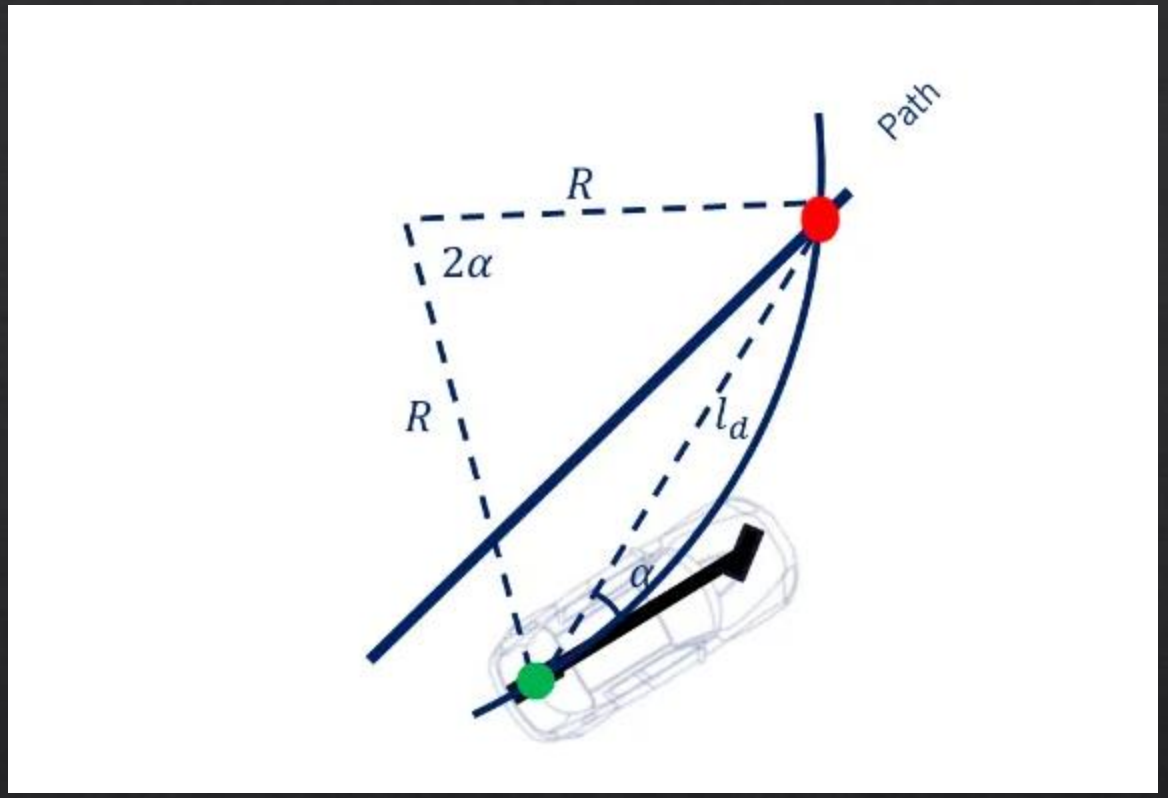
Tracking Control Algorithms

- Pure Pursuit
- Stanley Controller
- Kanayama's Control Method
- MPC – Model Predictive Controller
- PID

Links

- https://dingyan89.medium.com/three-methods-of-vehicle-lateral-control-pure-pursuit-stanley-and-mpc-db8cc1d32081
- https://www.mathworks.com/help/driving/ref/lateralcontrollerstanley.html
- https://core.ac.uk/download/pdf/36732512.pdf
- https://wiki.purduesigbots.com/software/control-algorithms/basic-pure-pursuit





$$\frac{l_d}{\sin 2\alpha} = \frac{R}{\sin\left(\frac{\pi}{2} - \alpha\right)}$$

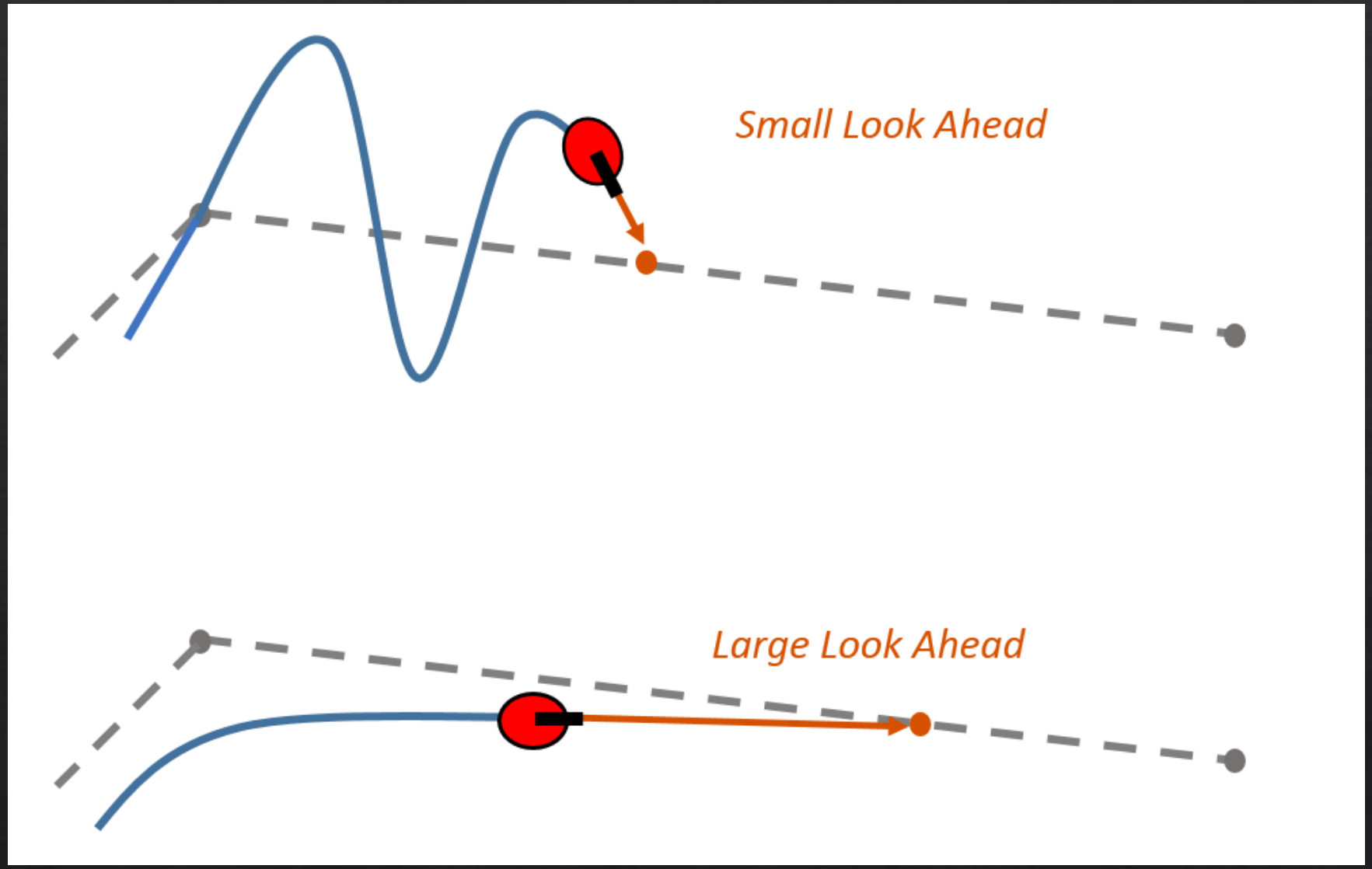
$$\frac{l_d}{2\sin\alpha\cos\alpha} = \frac{R}{\cos\alpha}$$

$$R = \frac{l_d}{2\sin\alpha}$$

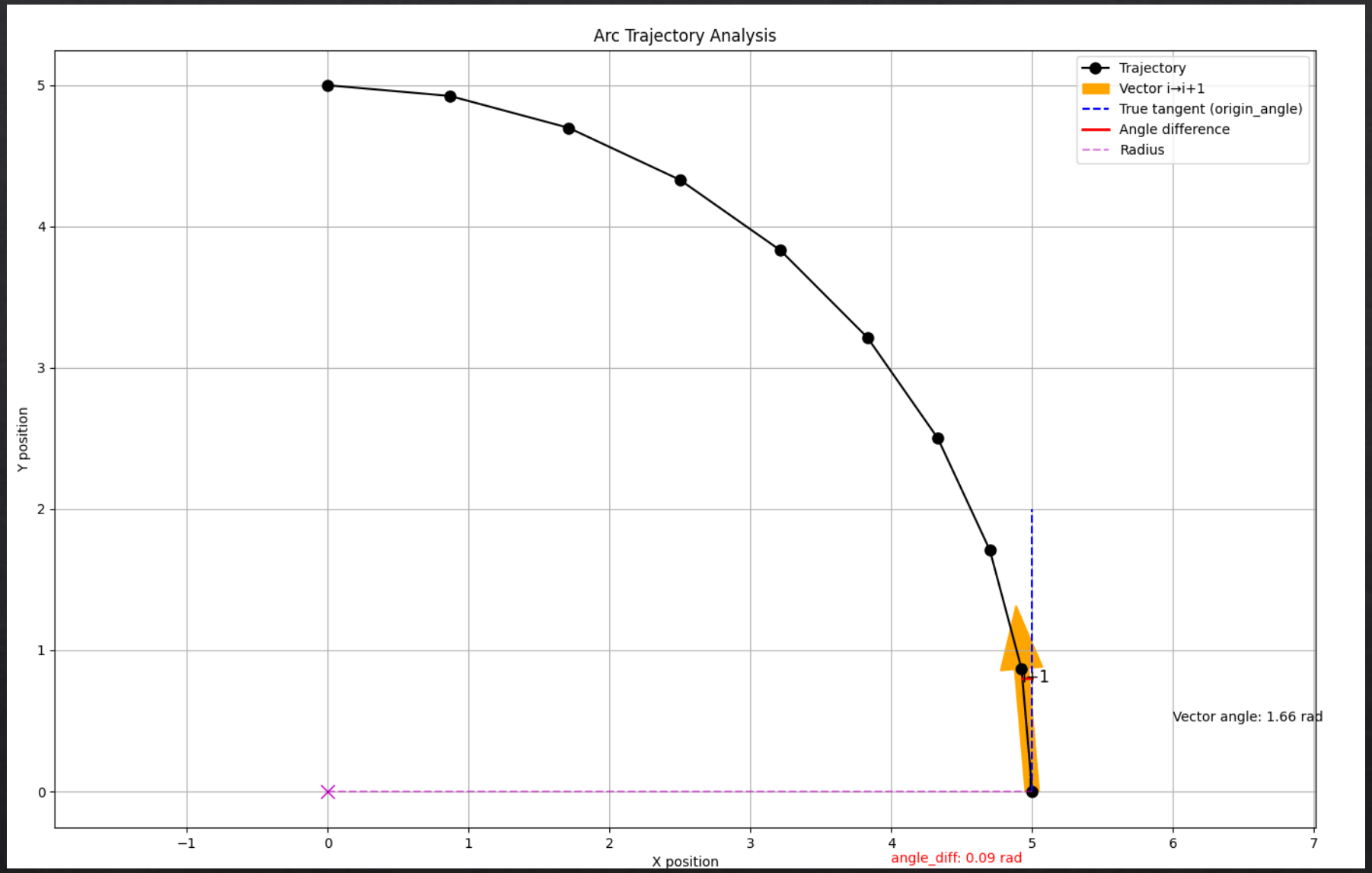
$$R = \frac{l_d}{2 \sin \alpha} \qquad \omega = \frac{V}{R}$$

$$\omega = \frac{V * 2 \sin \alpha}{l_d}$$

α → Diferença entre o ângulo do robô e a ângulo da trajetória
Ld → Distância "Look Ahead"
V → Velocidade linear do robô
ω → Target de velocidade angular ideal para atingir o trajeto


Small Look Ahead
Large Look Ahead


153 for i in range(len(self.waypoint_list)):
154 pygame.draw.circle(self.screen, (255, 0, 0), (int(self.waypoint_list[i].x), int(self.waypoint_list[i].y)), 5)
Line Follower
Time: 0.000s


Arc Trajectory Analysis
Trajectory
Vector i→i+1
True tangent (origin_angle)
Angle difference
Radius
Y position
X position
i+1
Vector angle: 1.66 rad
angle_diff: 0.09 rad


Target Angular Velocity
Angular Velocity
40
30
20
10
0
-10
-20
-30
-40
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
4,500
5,000


Target Velocity
Velocity
6
5
4
3
2
1
0
-1
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
4,500
5,000

$$x_e = \cos(\theta_c) \cdot (x_r - x_c) + \sin(\theta_c) \cdot (y_r - y_c)$$

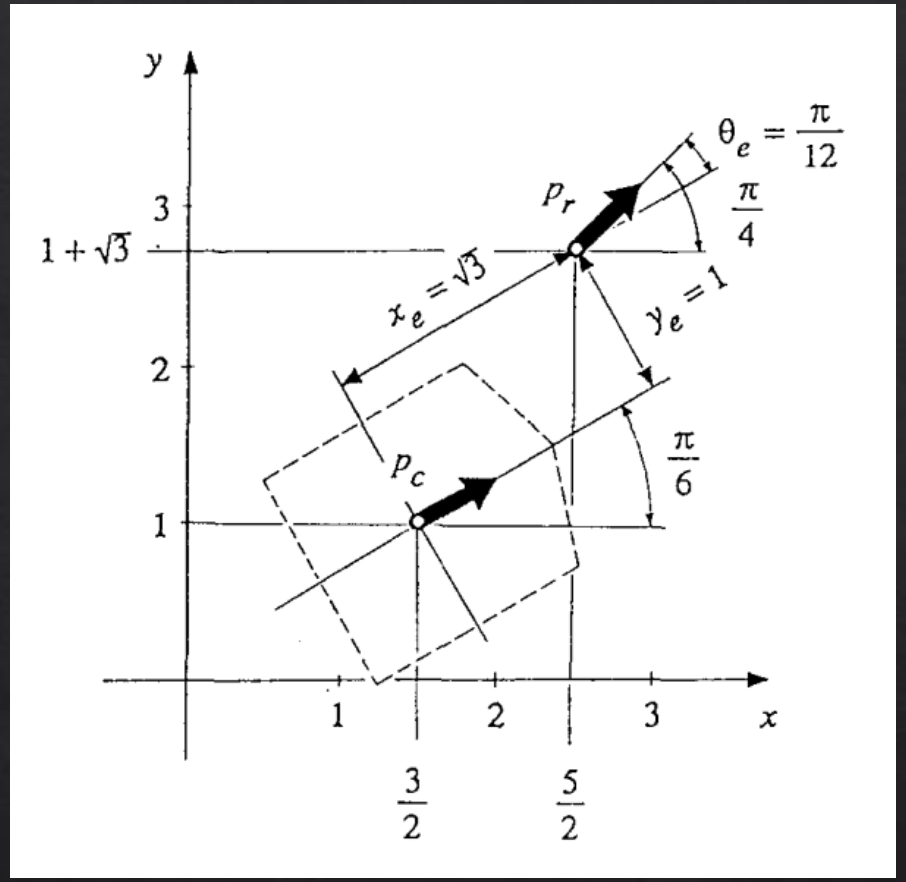
$$y_e = -\sin(\theta_c) \cdot (x_r - x_c) + \cos(\theta_c) \cdot (y_r - y_c)$$

$$\theta_e = \theta_r - \theta_c$$

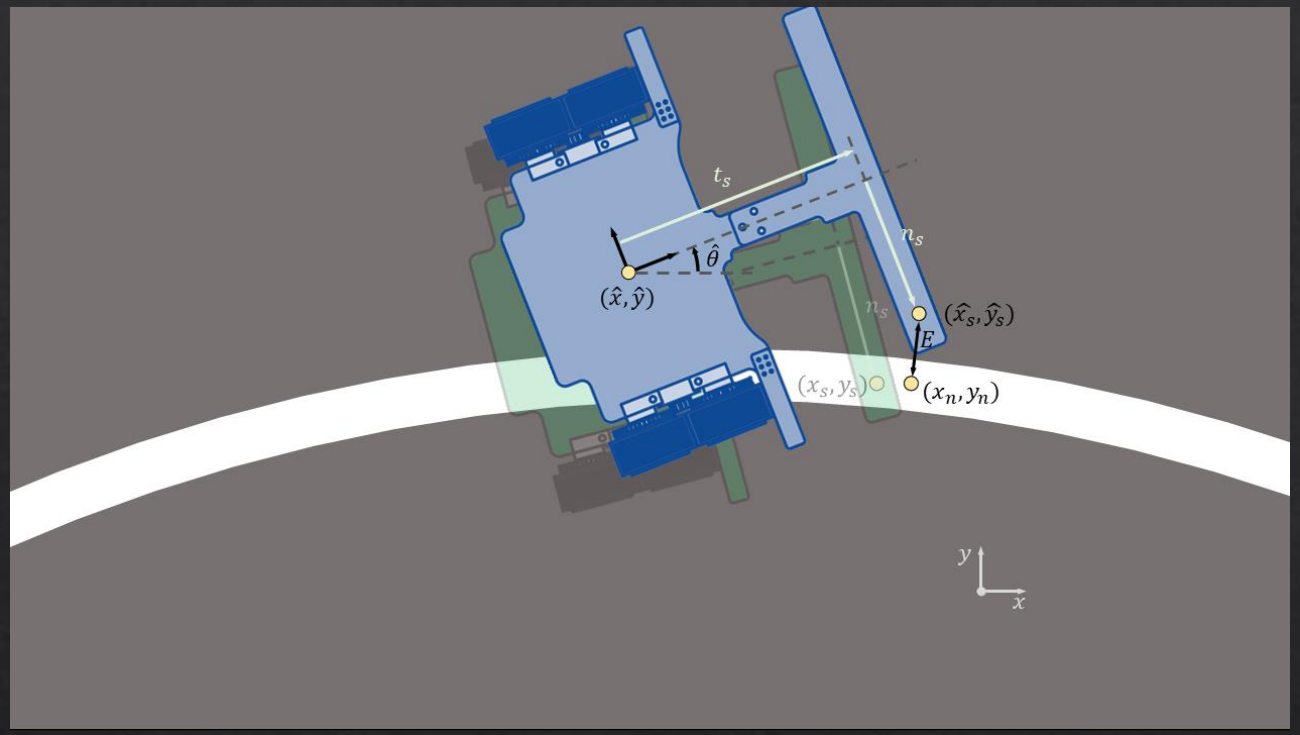
$$v_{\text{target}} = v_{\text{ref}} \cdot \cos(\theta_e) + K_x \cdot x_e$$

$$\omega_{\text{target}} = \omega_{\text{ref}} + v_{\text{ref}} \cdot (K_y \cdot y_e + K_\theta \cdot \sin(\theta_e))$$

r → se refere ao ponto de referencia
c → se refere ao ponto atual de controle
e → se refere à “erro”



CORREÇÃO DE ERROS



Ref: https://pag-beta.blogspot.com/2021/12/blog-post_20.html

1. Quais sensores estão vendo linha?
2. Existe linha nesse ponto?
3. Qual a distância mínima entre o lugar que ele acha que está e onde realmente tem linha?
4. Calcula correção necessária com base em uma constante

$$\hat{x_s} = \hat{x} + t_s \cos\hat{\theta} - n_s \sin\hat{\theta}\,,$$

$$\hat{y_s} = \hat{y} + t_s \sin\hat{\theta} + n_s \cos\hat{\theta}\,,$$

```
Point sensor_position_mm = bsp::line_array::get_relative_position_mm(idx);
Point estimated_sensor_pos = (current_position_mm + rotate(sensor_position_mm, current_angle_rad));
auto nearest_point_mm = nearest_point_on_track(estimated_sensor_pos, NUM_OF_POINTS_TO_SEARCH, waypoint_idx);

float x_correction = (nearest_point_mm.x - estimated_sensor_pos.x);
float y_correction = (nearest_point_mm.y - estimated_sensor_pos.y);

correction.x += x_correction;
correction.y += y_correction;
```

```
new_estimated_pos.x = current_position_mm.x + correction.x * Config::fix_position_ratio;
new_estimated_pos.y = current_position_mm.y + correction.y * Config::fix_position_ratio;
current_position_mm = new_estimated_pos;
```

RESULTADOS

PERGUNTAS?

OBRIGADO!