

# Comportamientos para vehículos autónomos y la categoría AutoModelCar

Instructor: Dr. Marco Antonio Negrete Villanueva

Facultad de Ingeniería, UNAM

Escuela de Inviero de Robótica 2024  
Nanchital, Veracruz

<https://github.com/mnegretev/EIR-2024-AutonomousVehicles>

# Objetivos:

**Objetivo General:** Revisar conceptos básicos para operar un vehículo sin conductor en el contexto de las pruebas de la categoría AutoModelCar

## Objetivos Específicos:

- ▶ Revisar el hardware necesario para tener un vehículo sin conductor
- ▶ Dar un panorama general del software necesario para desarrollar un vehículo sin conductor
- ▶ Revisar las herramientas disponibles para implementar los comportamientos básicos en un vehículo sin conductor
  - ▶ Detección de carriles
  - ▶ Seguimiento de carriles
  - ▶ Rebase
  - ▶ Detección de señales de tránsito

# Contenido

Introducción

La plataforma ROS

La categoría AutoModelCar

Detección de carriles

Seguimiento de carriles

## ¿Por qué vehículos sin conductor?

- ▶ La mayor cantidad de accidentes tienen como causa un error humano
- ▶ Se podría disminuir este número con vehículos sin conductor
- ▶ Menores tiempos de traslado mediante planeación de rutas óptimas y conducción sin distracciones
- ▶ Movilidad urbana más accesible
- ▶ Disminución de emisiones debido a una conducción óptima

# ¿Cuándo un vehículo se considera autónomo?

La SAE propone seis niveles de autonomía:

**Nivel 0 (No driving automation):** El conductor es responsable de todas las tareas de conducción.

**Nivel 1 (Driving assistance):** El sistema de conducción automatizada toma control en situaciones muy específicas. Ejemplo: mantener cierta velocidad en carretera o frenar en automático ante un obstáculo.



# Niveles de autonomía SAE

**Nivel 2 (Partial driving automation):** El auto puede realizar la mayor parte de tareas de conducción pero el conductor debe mantenerse atento durante toda la conducción. Ejemplo: Tesla.



# Niveles de autonomía SAE

**Nivel 3 (Conditional driving automation):** El conductor puede desconectarse de la tarea de conducción bajo ciertos climas y carreteras. Todavía se requiere que el conductor resuelva ciertas situaciones de manejo. Ejemplo: Mercedes-Benz

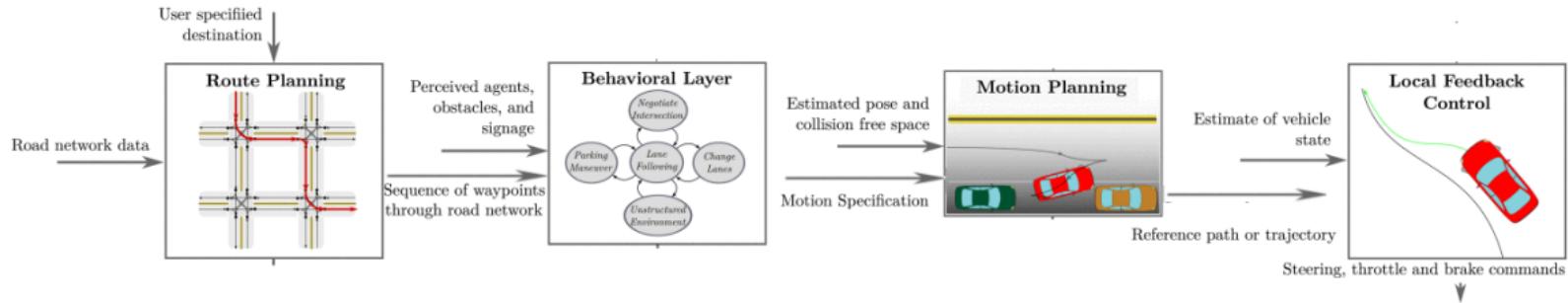


# Niveles de autonomía SAE

**Nivel 4 (High driving automation):** El vehículo es capaz de ejecutar funciones tácticas y operativas y es capaz de asegurarse automáticamente si los límites operativos se superan. El vehículo no es completamente autónomo pero el conductor no necesita responder ante fallas operativas. Ejemplo: **EI TMR, pronto (esperemos)**

**Nivel 5 (Full driving automation):** No se requiere intervención humana y el sistema puede conducir sin restricciones de clima, carretera, horario o condición geográfica. Ejemplo: **EI TMR, pronto (esperemos)**

# Software requerido



(Imagen tomada de *A Survey of Motion Planning and Control Techniques for Self-Driving Urban Vehicles*)



**ROS (Robot Operating System)** es un *middleware* de código abierto para el desarrollo de robots móviles.

- ▶ Implementa funcionalidades comúnmente usadas en el desarrollo de robots como el paso de mensajes entre procesos y la administración de paquetes.
- ▶ Muchos drivers y algoritmos ya están implementados.
- ▶ Es una plataforma distribuida de procesos (llamados *nodos*).
- ▶ Facilita el reuso de código.
- ▶ Independiente del lenguaje (Python y C++ son los más usados).
- ▶ Facilita el escalamiento para proyectos de gran escala.

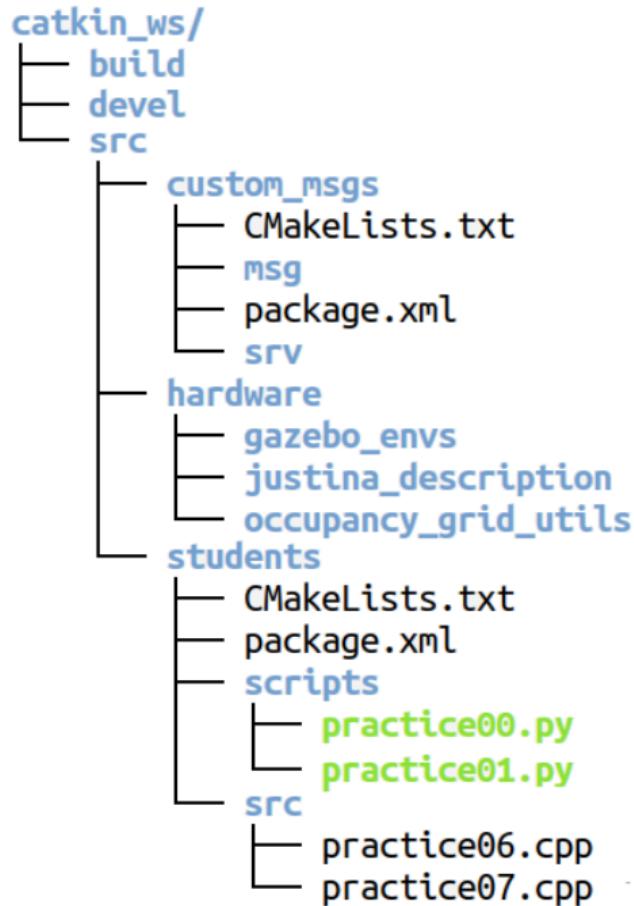
ROS se puede entender en dos grandes niveles conceptuales:

- ▶ **Sistema de archivos:** Recursos de ROS en disco
- ▶ **Grafo de procesos:** Una red *peer-to-peer* de procesos (llamados nodos) en tiempo de ejecución.

# Sistema de archivos

Recursos en disco:

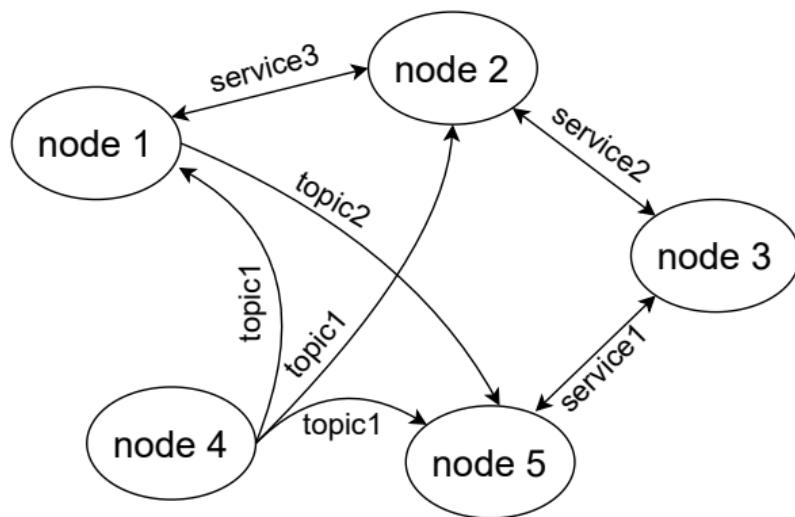
- ▶ **Workspace:** carpeta que contiene los paquetes desarrollados
- ▶ **Paquetes:** Principal unidad de organización del software en ROS (concepto heredado de Linux)
- ▶ **Manifiesto:** (`package.xml`) provee metadatos sobre el paquete (dependencias, banderas de compilación, información del desarrollador)
- ▶ **Mensajes (msg):** Archivos que definen la estructura de un *mensaje* en ROS.
- ▶ **Servicios (srv):** Archivos que definen las estructuras de la petición (*request*) y respuesta (*response*) de un servicio.



# Grafo de procesos

El grafo de procesos es una red *peer-to-peer* de programas (nodos) que intercambian información entre sí. Los principales componentes del este grafo son:

- ▶ master
- ▶ servidor de parámetros
- ▶ nodos
- ▶ mensajes
- ▶ servicios



# Tópicos y servicios

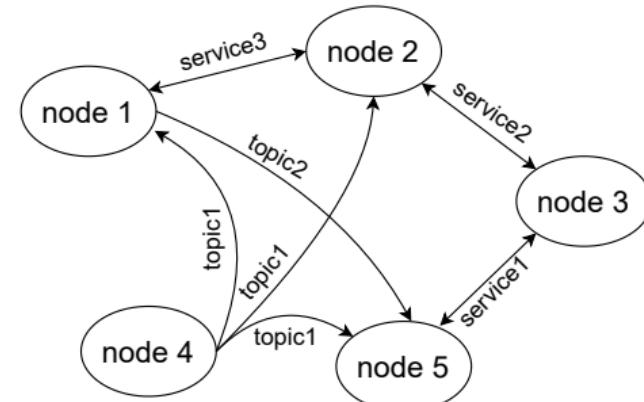
Los nodos (procesos) en ROS intercambian información a través de dos grandes patrones:

## ► Tópicos

- ▶ Son un patrón  $1:n$  de tipo *publicador/suscriptor*
- ▶ Son no bloqueantes
- ▶ Utilizan estructuras de datos definidas en archivos `*.msg` para el envío de información

## ► Servicios

- ▶ Son un patrón  $1:1$  de tipo *petición/respuesta*
- ▶ Son bloqueantes
- ▶ Utilizan estructuras de datos definidas en archivos `*.srv` para el intercambio de información.



Para mayor información:

- ▶ Tutoriales <http://wiki.ros.org/ROS/Tutorials>
- ▶ Koubâa, A. (Ed.). (2020). Robot Operating System (ROS): The Complete Reference. Springer Nature

## La categoría AutoModelCar

- ▶ Se originó por el proyecto *Visiones de movilidad urbana* a través del cual se donaron vehículos a escala a varias instituciones educativas y de investigación del país



# La categoría AutoModelCar

- ▶ Originalmente solo se permitían vehículos AutoNOMOS:



- ▶ Pero ahora se puede participar con cualquier vehículo a escala:



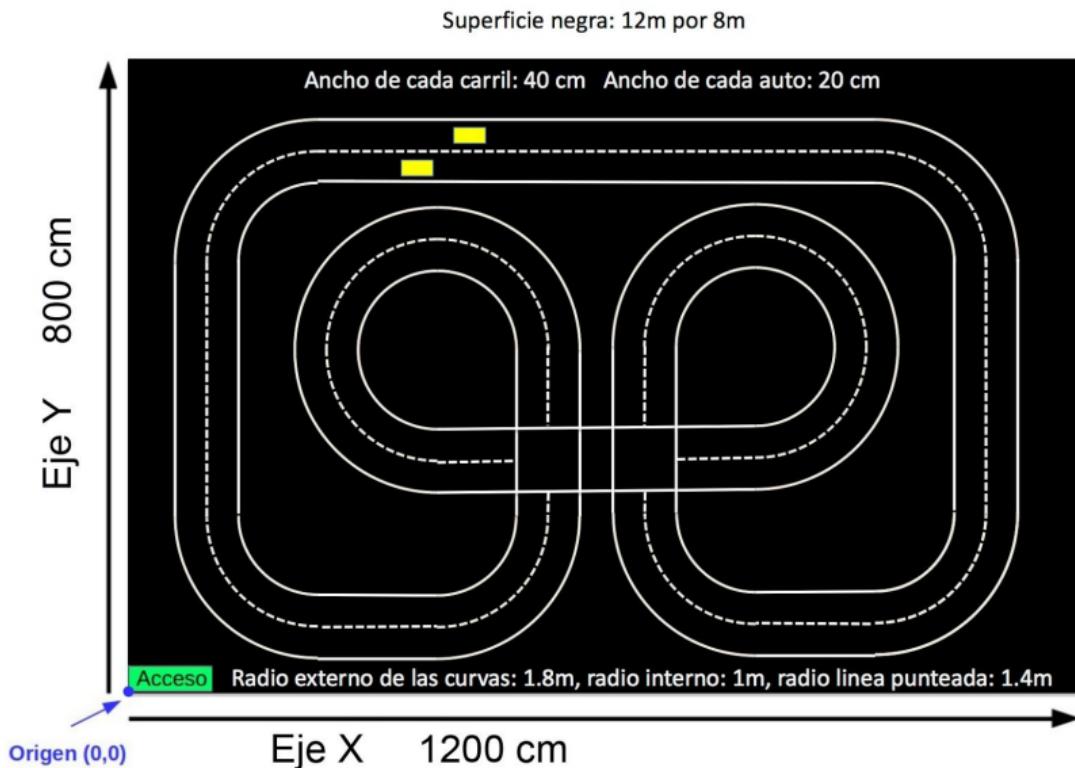
# La categoría AutoModelCar

- ▶ La competencia consta de cuatro pruebas (esperemos aumentarlas este año):
  - ▶ Navegación autónoma sin obstáculos
  - ▶ Navegación con obstáculos estáticos
  - ▶ Navegación con obstáculos en movimiento
  - ▶ Estacionamiento

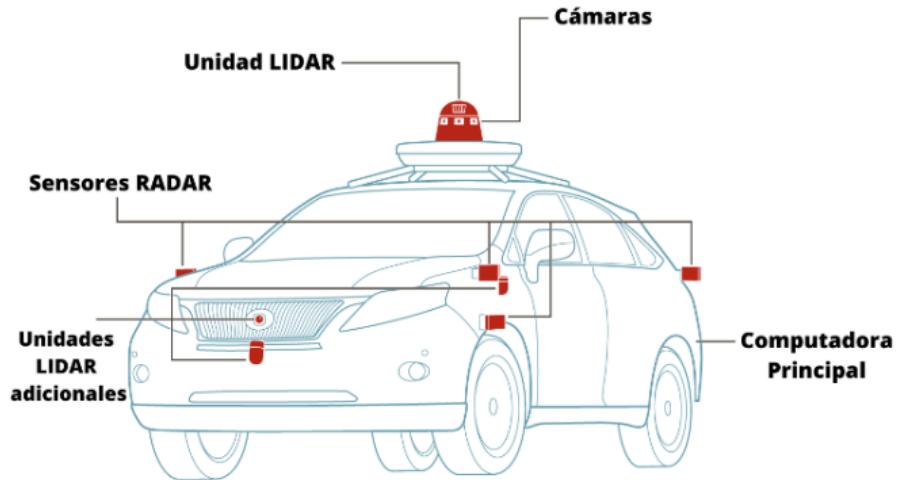


# La categoría AutoModelCar

Se utiliza una pista con rectas, curvas y cruces:



# Hardware para vehículos sin conductor



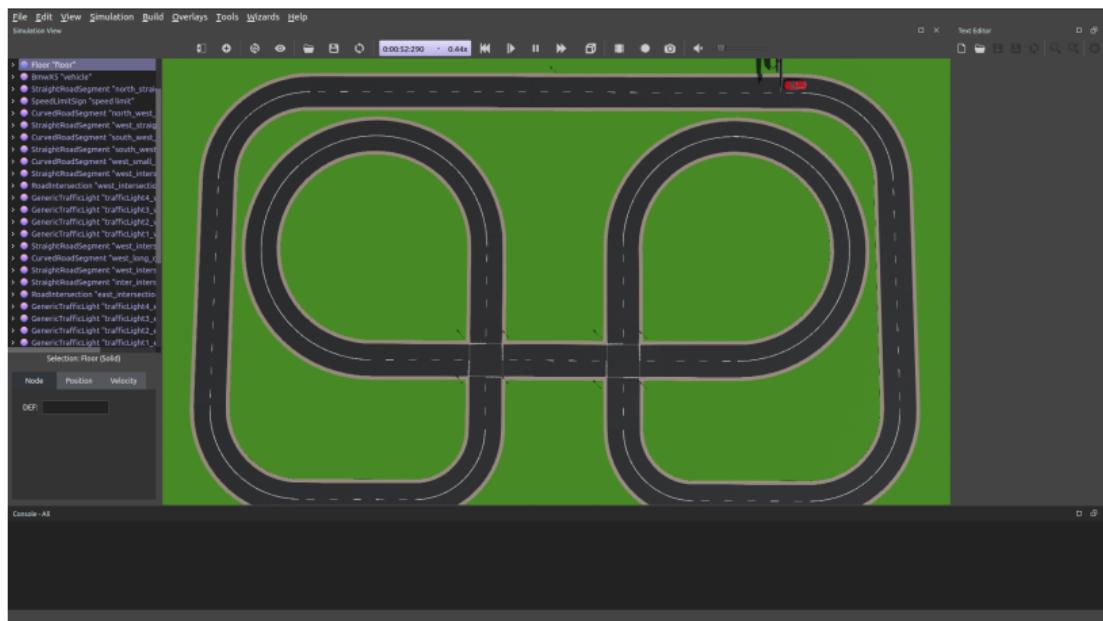
## ¿Por qué utilizar simuladores?

- ▶ Instrumentar un vehículo autónomo puede ser costoso, incluso si se hace a escala.
- ▶ En simulación se pueden construir ambientes controlados que permiten probar partes específicas del sistema de autonomía.
- ▶ Se pueden enfocar los esfuerzos al desarrollo de software dando por hecho un hardware funcional.
- ▶ Algunos ejemplos de simuladores:
  - ▶ Gazebo (utilizado en el TMR 2021)
  - ▶ Webots (utilizado en los TMRs 2022 y 2023)
  - ▶ Matlab (utilizado por quien tiene dinero para pagar la licencia)

# El simulador Webots

El simulador Webots tiene varias características para simulación de vehículos sin conductor:

- ▶ Motor de físicas ODE y OpenGL como motor de renderizado.
- ▶ Multiplataforma y disponible para sistemas operativos Linux, Windows y macOS.
- ▶ Programación en diferentes lenguajes: C, C++, Python, Java, MATLAB.
- ▶ Amplia variedad de robots, sensores, actuadores, objetos y materiales.
- ▶ Fácil integración con la plataforma ROS.



# SUMO (Simulation of Urban Mobility)

Es un paquete de simulación de tráfico desarrollado por el Centro Aeroespacial Alemán.

- ▶ De código abierto
- ▶ Diseñado para manejar redes grandes de vehículos
- ▶ Se pueden simular vehículos, peatones, ciclistas, señales de tránsito, transporte público, entre muchas otras opciones
- ▶ Fácil integración con Webots mediante el objeto *SUMO interface*:
- ▶ Se puede definir el comportamiento de los vehículos mediante archivos \*.nod.xml, \*.edg.xml, \*.net.xml, \*.rou.xml y \*.sumocfg

# Ejercicio 1 - Herramientas de simulación

1. Abra una terminal y ejecute la simulación con el comando:

```
roslaunch eir2024 navigation_moving_obstacles.launch
```

2. Detenga la simulación

3. Abra el archivo

`catkin_ws/src/eir2024/worlds/navigation_moving_obstacles.net/sumo.rou.xml` y  
descomente las líneas 8 a 14

4. Corra nuevamente la simulación. Se deberían observar más vehículos

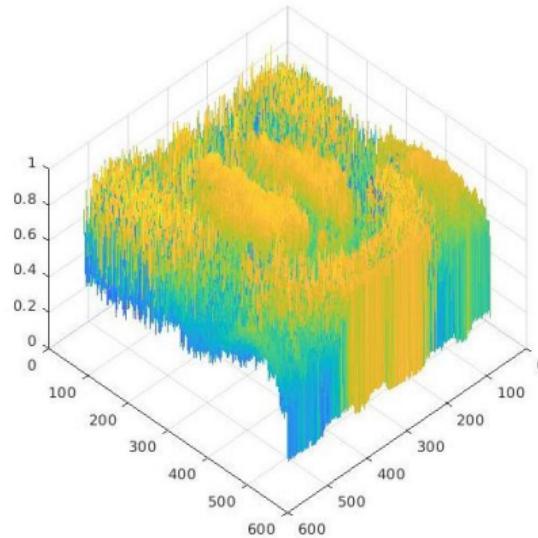
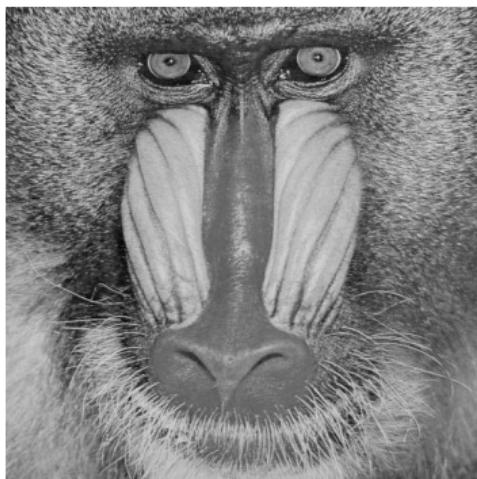
5. Detenga la simulación y ejecute los siguientes comandos:

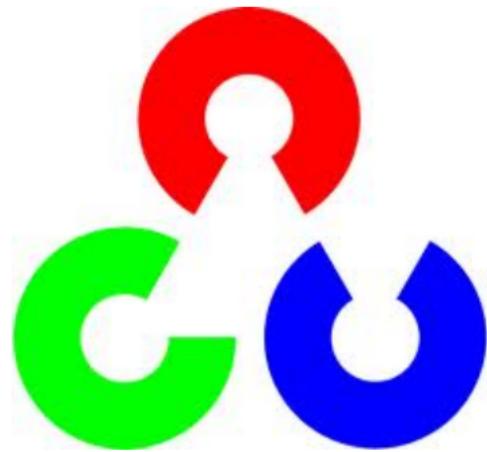
```
roscd eir2024/worlds/navigation_moving_obstacles.net/
netconvert -S 10 -n sumo.nod.xml -e sumo.edg.xml -o sumo.net.xml
```

6. Corra nuevamente la simulación. Se debería notar un cambio en la velocidad de los vehículos

## Imágenes como funciones

- ▶ Una imagen (en escala de grises) es una función  $I(x, y)$  donde  $x, y$  son variables discretas en coordenadas de imagen y la función  $I$  es intensidad luminosa.
- ▶ Las imágenes también pueden considerarse como arreglos bidimensionales de números entre un mínimo y un máximo (usualmente 0-255).
- ▶ Aunque formalmente una imagen es un mapeo  $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ , en la práctica, tanto  $x, y$  como  $I$  son variables discretas con valores entre un mínimo y un máximo.
- ▶ Las imágenes de color son funciones vectoriales  $f : \mathbb{R}^2 \rightarrow \mathbb{R}^3$  donde cada componente de la función se llama canal.





OpenCV es una biblioteca libre de visión artificial, originalmente desarrollada por Intel.  
Programación en código Python, C y C++  
Existen versiones para GNU/Linux, Mac OS X y Windows

# OpenCV Overview: > 500 functions

[opencv.willowgarage.com](http://opencv.willowgarage.com)

### General Image Processing Functions

### Image Pyramids

### Segmentation

### Geometric descriptors

### Transforms

### Features

### Camera calibration, Stereo, 3D

### Utilities and Data Structures

### Machine Learning:

- Detection,
- Recognition

### Tracking

### Fitting

### Matrix Math

# Convolución

Si se conoce la respuesta al impulso  $H(x, y)$  de un sistema SLID, se puede obtener la salida  $O(x, y)$  ante cualquier entrada  $I(x, y)$ , mediante la convolución, definida como:

$$O(x, y) = I(x, y) * H(x, y) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(i, j)H(x - i, y - j)$$

Ejemplos:

$$\begin{bmatrix} 3 & 1 & 4 & 1 \\ 5 & 9 & 2 & 6 \\ 5 & 3 & 5 & 8 \\ 9 & 7 & 9 & 3 \end{bmatrix} * [1 \quad -1] = \begin{bmatrix} 3 & -2 & 3 & -3 & -1 \\ 5 & 4 & -7 & 4 & -6 \\ 5 & -2 & 2 & 3 & -8 \\ 9 & -2 & 2 & -6 & -3 \end{bmatrix}$$

$$\begin{bmatrix} 3 & 1 & 4 & 1 \\ 5 & 9 & 2 & 6 \\ 5 & 3 & 5 & 8 \\ 9 & 7 & 9 & 3 \end{bmatrix} * \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 3 & 1 & 4 & 1 \\ 2 & 8 & -2 & 5 \\ 0 & -6 & 3 & 2 \\ 4 & 4 & 4 & -5 \\ -9 & -7 & -9 & -3 \end{bmatrix}$$

# El filtro Gaussiano

Una primera aplicación de una convolución es el filtro Gaussiano

$$g(x, y) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/(2\sigma^2)}$$

Se utiliza para reducir el tipo de ruido más común que es el ruido Gaussiano. Un ejemplo de kernel Gaussiano es:

$$H = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

# Gradiente

El gradiente de una imagen está definido como:

$$\nabla I = \left[ \frac{\partial I}{\partial x}, \frac{\partial I}{\partial y} \right]$$

Las derivadas parciales se puede aproximar mediante diferencias finitas:

$$\begin{aligned}\frac{\partial I}{\partial x} &= \lim_{\Delta x \rightarrow 0} \frac{I(x + \Delta x, y) - I(x, y)}{\Delta x} \approx I_{i,j} - I_{i,j-1} \\ \frac{\partial I}{\partial y} &= \lim_{\Delta y \rightarrow 0} \frac{I(x, y + \Delta y) - I(x, y)}{\Delta y} \approx I_{i,j} - I_{i-i,j}\end{aligned}$$

donde  $(i, j)$  representan las coordenadas de imagen renglón-columna. Estas diferencias finitas se puede obtener mediante una convolución:

$$\begin{aligned}\frac{\partial I}{\partial x} &\approx I * [1 \quad -1] \\ \frac{\partial I}{\partial y} &\approx I * \begin{bmatrix} 1 \\ -1 \end{bmatrix}\end{aligned}$$

Una mejor aproximación de la derivada es no solo tomar la diferencia entre el valor actual y el anterior ( $I_{i,j} - I_{i-1,j}$ ), sino promediarlo con la diferencia ( $I_{i+1,j} - I_{i,j}$ ):

$$\frac{1}{2}[(I_{i,j} - I_{i-1,j}) + (I_{i+1,j} - I_{i,j})] = \frac{1}{2}(I_{i+1,j} - I_{i-1,j})$$

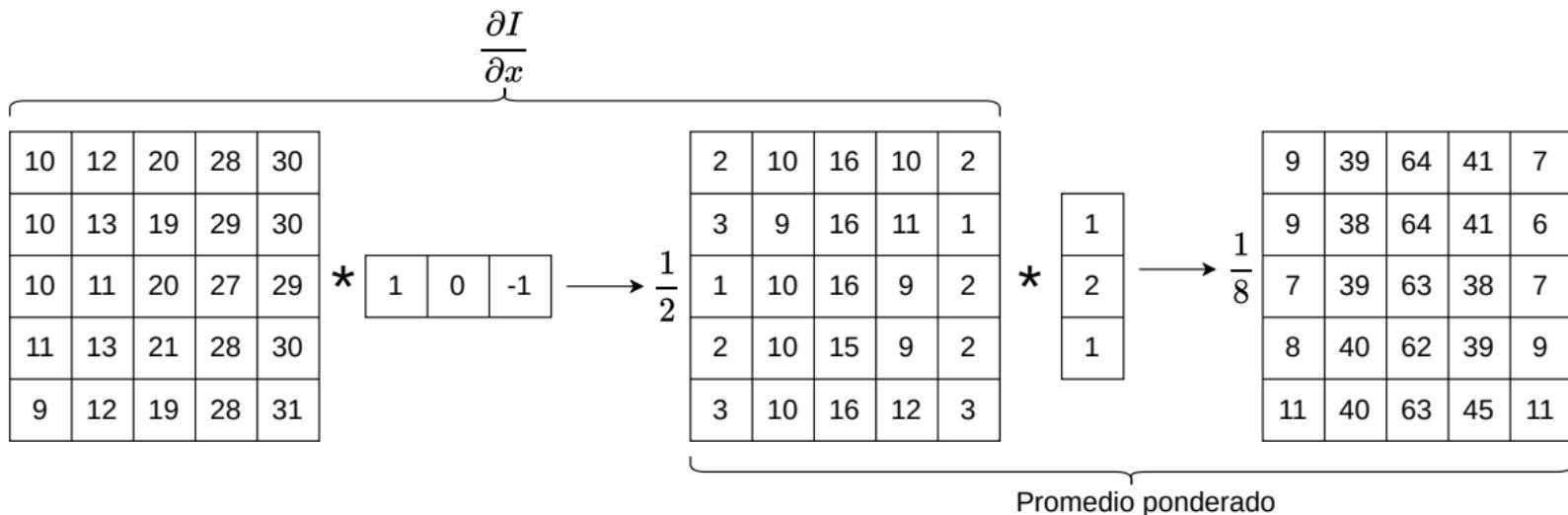
Generalmente se ignora el coeficiente y se utilizan los siguientes Kernels:

$$\frac{\partial I}{\partial x} \approx I * [1 \quad 0 \quad -1]$$

$$\frac{\partial I}{\partial y} \approx I * \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$$

## El filtro de Sobel

El Operador de Sobel o Filtro de Sobel consiste en un Kernel que permite obtener las derivadas parciales, aproximadas por diferencias finitas, y promediadas con un filtro Gaussiano:



Se realiza un proceso similar para la derivada parcial en  $Y$ . Aplicando la propiedad asociativa de la convolución, se obtienen los siguientes ekernels:

$$S_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad S_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

# Magnitud y Ángulo

El gradiente en cada pixel de la imagen se puede calcular mediante la aproximación de las derivadas parciales:

$$\begin{aligned}\frac{\partial I}{\partial x} &\approx I * S_x = G_x \\ \frac{\partial I}{\partial y} &\approx I * S_y = G_y\end{aligned}$$

En la mayoría de las aplicaciones es más útil expresar el gradiente en forma polar:

$$\nabla I = G_m \angle G_a$$

Donde la magnitud del gradiente y la fase, para cada pixel, se calculan como:

$$\begin{aligned}G_{m_{i,j}} &= \sqrt{G_{x_{i,j}}^2 + G_{y_{i,j}}^2} \\ G_{a_{i,j}} &= \text{atan2}(G_{y_{i,j}}, G_{x_{i,j}})\end{aligned}$$

Otra forma de calcular la magnitud del gradiente es mediante la norma 1:

$$G_{m_{i,j}} = |G_{x_{i,j}}| + |G_{y_{i,j}}|$$

## Ejercicio 02 - Gradiente de una imagen

1. Abra una terminal y ejecute la simulación con el comando:

```
roslaunch eir2024 navigation_no_obstacles.launch
```

2. En otra terminal, ejecute el ejercicio dos con el siguiente comando:

```
rosrun eir2024 exercise02.py
```

3. Detenga el ejercicio 02, abra el archivo fuente y agregue el siguiente código en la línea 25:

```
25 grad_x = cv2.Sobel(gray, cv2.CV_16S, 1, 0, ksize=3)
26 grad_y = cv2.Sobel(gray, cv2.CV_16S, 0, 1, ksize=3)
27 abs_grad_x = cv2.convertScaleAbs(grad_x)
28 abs_grad_y = cv2.convertScaleAbs(grad_y)
29 grad = cv2.addWeighted(abs_grad_x, 0.5, abs_grad_y, 0.5, 0)
30 cv2.imshow("Gradient Magnitude", grad)
31
```

4. Corra nuevamente el ejercicio y observe el resultado. Ahora agregue el siguiente código en la línea 21:

```
21 gray = cv2.GaussianBlur(gray, (5, 5), 0)
22
```

5. Corra nuevamente el ejercicio 2 y observe el resultado.

# Detector de Bordes de Canny

El detector de bordes de Canny es un detector basado en gradiente que consta de los siguientes pasos básicos:

1. Obtención del gradiente en magnitud y ángulo, mediante operadores de Sobel
2. Supresión de puntos no máximos
3. Aplicación de un doble umbral

Aunque no es un paso en sí del Detector de Canny, generalmente se considera como primer paso la aplicación de un filtro Gaussiano para disminuir el ruido.

# Obtención del gradiente

Después del filtro Gaussiano, el primer paso es obtener el gradiente de la imagen mediante el Filtro de Sobel, en la forma de magnitud y ángulo:

10	12	20	28	30
10	13	19	29	30
10	11	20	27	29
11	13	21	28	30
9	12	19	28	31

$$\begin{array}{c} \begin{matrix} & & \\ & & \\ & & \end{matrix} \\ \star \end{array} = \begin{array}{c} \begin{matrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{matrix} \end{array} \quad \begin{array}{|c|c|c|c|c|} \hline 9 & 39 & 64 & 41 & 7 \\ \hline 9 & 38 & 64 & 41 & 6 \\ \hline 7 & 39 & 63 & 38 & 7 \\ \hline 8 & 40 & 62 & 39 & 9 \\ \hline 11 & 40 & 63 & 45 & 11 \\ \hline \end{array}$$

9.05	39.01	64.00	41.01	7.07
9.05	38.05	64.03	41.10	7.21
7.61	39.11	63.07	38.00	7.07
8.24	40.00	62.00	39.11	11.40
13.03	40.44	63.19	45.01	11.40

$$x, y \rightarrow r\angle\theta$$



10	12	20	28	30
10	13	19	29	30
10	11	20	27	29
11	13	21	28	30
9	12	19	28	31

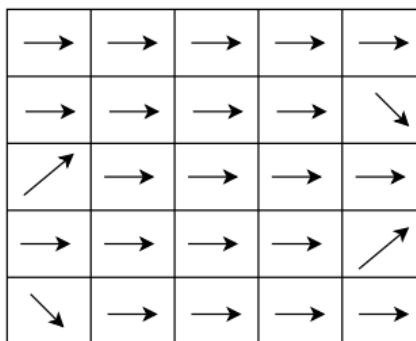
$$\begin{array}{c} \begin{matrix} & & \\ & & \\ & & \end{matrix} \\ \star \end{array} = \begin{array}{c} \begin{matrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{matrix} \end{array} \quad \begin{array}{|c|c|c|c|c|} \hline 1 & 1 & 0 & 1 & 1 \\ \hline -1 & -2 & -2 & -3 & -4 \\ \hline 3 & 3 & 3 & 0 & -1 \\ \hline -2 & 0 & 0 & 3 & 7 \\ \hline -7 & -6 & -5 & -1 & 3 \\ \hline \end{array}$$

0.11	0.02	0	0.02	0.14
-0.11	-0.05	-0.03	-0.07	-0.58
0.40	0.07	0.04	0	-0.14
-0.24	0	0	0.07	0.66
-0.56	-0.14	-0.07	-0.02	0.26

## Supresión de no máximos

Este paso consiste en comparar la magnitud de cada pixel, con los pixeles anterior y posterior en la dirección del gradiente. Aunque la fase es un ángulo en  $[-\pi, \pi]$ , la dirección del gradiente se debe redondear a algún valor correspondiente a la conectividad 8:  $N, NE, E, SE$ . Debido a que el pixel se compara en la dirección positiva y negativa del gradiente, no es necesario considerar las direcciones  $S, SW, W, NW$ .

9.05	39.01	64.00	41.01	7.07
9.05	38.05	64.03	41.10	7.21
7.61	39.11	63.07	38.00	7.07
8.24	40.00	62.00	39.11	11.40
13.03	40.44	63.19	45.01	11.40



0	0	64.00	0	0
0	0	64.03	0	0
0	0	63.07	0	0
0	0	62.00	0	0
0	0	63.19	0	0

Para cada pixel  $p_i$ , considere  $p_{i+1}$ , el pixel siguiente en la dirección del gradiente, y  $p_{i-1}$ , el pixel anterior, en la dirección del gradiente. El valor para cada pixel  $q_i$  en la imagen resultante es:

$$q_i = \begin{cases} p_i & \text{si } p_i > p_{i+1} \text{ y } p_i > p_{i-1} \\ 0 & \text{en otro caso} \end{cases}$$

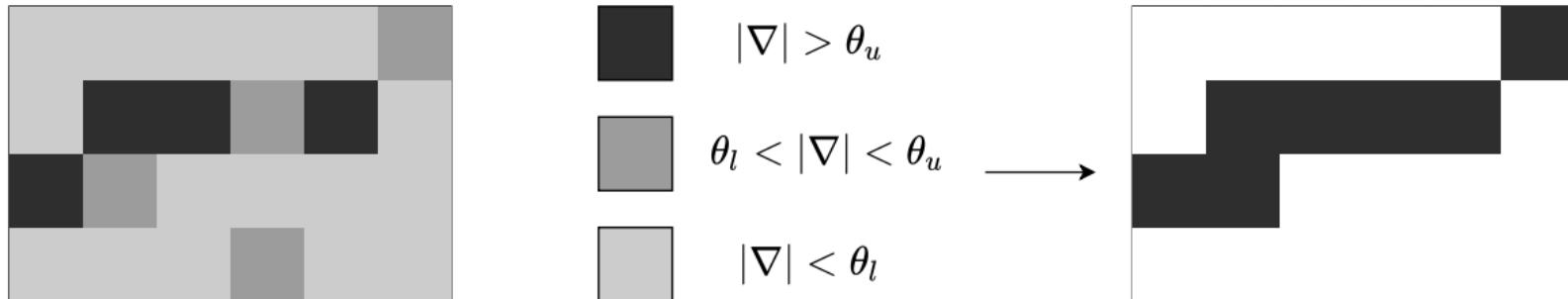
# Aplicación de doble umbral

En este paso, se definen dos umbrales: superior  $\theta_u$  e inferior  $\theta_l$ . Los pixeles se clasifican en tres tipos:

- ▶ Fuertes: pixeles con magnitud del gradiente mayor que el umbral superior  $|\nabla| > \theta_u$
- ▶ Débiles: pixeles con magnitud del gradiente entre ambos umbrales  $\theta_l < |\nabla| < \theta_u$
- ▶ Suprimidos: pixeles con magnitud del gradiente menor que el umbral inferior  $|\nabla| < \theta_l$

La imagen resultante se forma con las siguientes reglas:

- ▶ Todos los pixeles fuertes son parte de un borde.
- ▶ Todos los pixeles suprimidos no son bordes.
- ▶ Los pixeles débiles son parte de un borde solo si están conectados (en conectividad 8) con un pixel fuerte.



# Ejercicio 03 - Detección de Bordes

1. Abra una terminal y ejecute la simulación con el comando:

```
roslaunch eir2024 navigation_no_obstacles.launch
```

2. Abra el archivo fuente del ejercicio 03 y agregue el siguiente código en la línea 33:

```
33 gray = cv2.GaussianBlur(gray, (5,5), 0)
34 edges = cv2.Canny(gray, lower_threshold, upper_threshold)
35
```

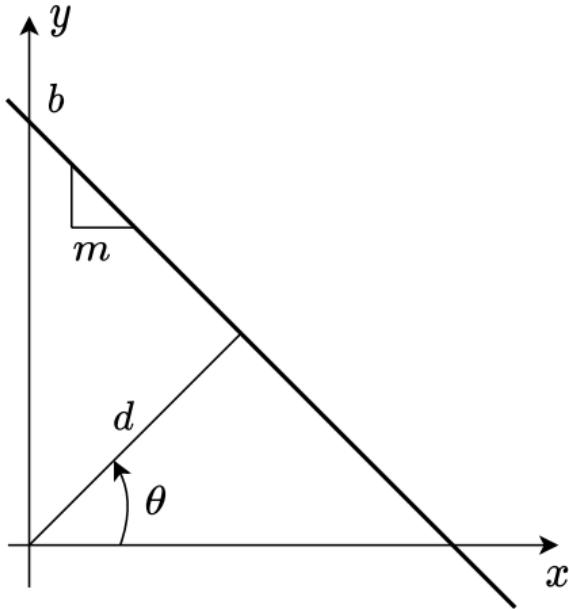
3. En otra terminal, ejecute el ejercicio 03 con el siguiente comando:

```
rosrun eir2024 exercise03.py
```

4. Sintonice los umbrales hasta obtener una detección de bordes satisfactoria. Utilice la GUI para mover el vehículos y probar en diferentes condiciones.

# La Transformada Hough

La Transformada Hough es un método que permite encontrar líneas, círculos y otras formas geométricas que se puedan describir fácilmente mediante expresiones analíticas. En el caso de las líneas, se trata de encontrar los dos parámetros que describen la recta:

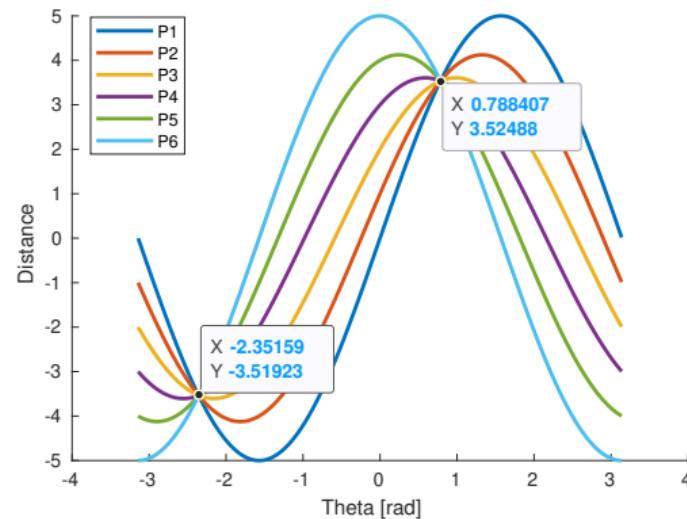
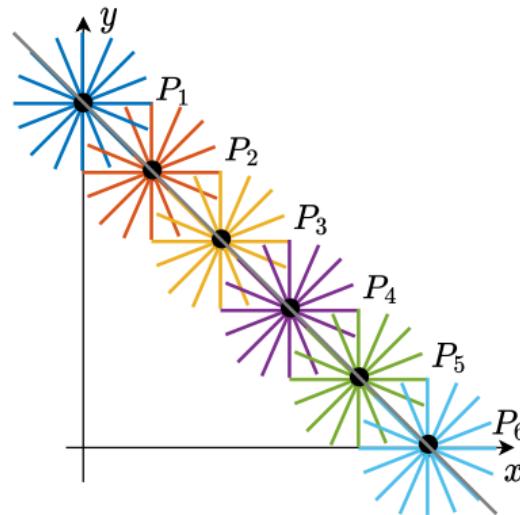


- ▶ La forma pendiente-ordenada  $y = mx + b$  tiene la desventaja de no poder expresar líneas verticales.
- ▶ La forma canónica  $Ax + By + C$  requiere de una normalización  $A_1x + B_1y + 1 = 0$  para que solo sean dos parámetros.
- ▶ Una forma más conveniente, es la forma normal  $d = x \cos \theta + y \sin \theta$
- ▶ Esta última forma tiene una ventaja: si la línea corresponde a un borde, el ángulo  $\theta$  será la dirección del gradiente.

# El Espacio de Hough

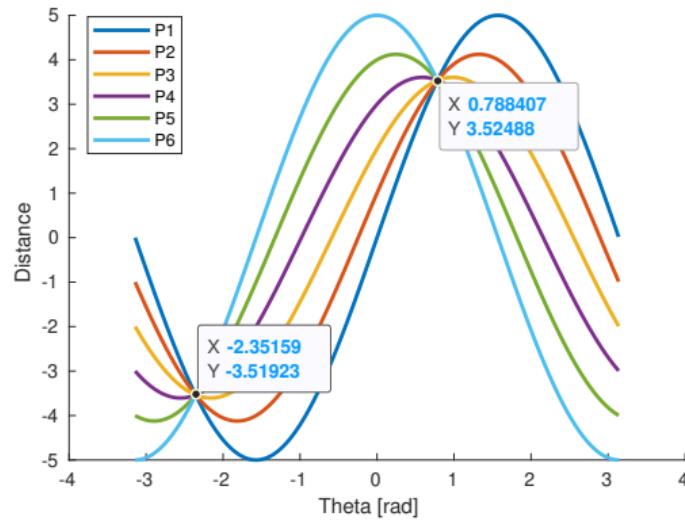
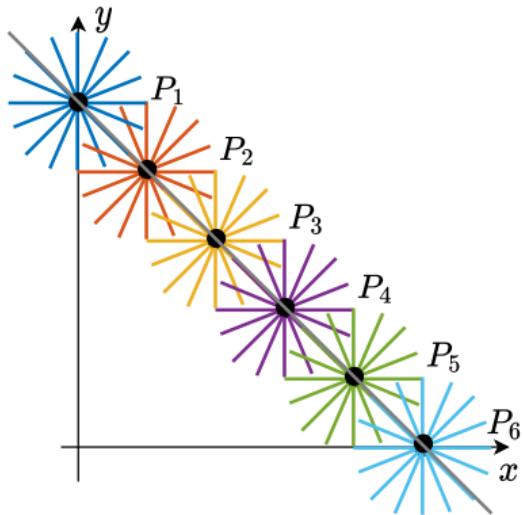
El Espacio de Hough, para el caso de las líneas, es el conjunto de todos los posibles pares  $(\theta, d)$ .

- ▶ Una recta  $L$  en el espacio cartesiano corresponde a un punto  $P_h$  en el Espacio de Hough
- ▶ Un punto  $P_c$  en el espacio cartesiano corresponde a una curva  $C$  en el Espacio de Hough. Esta curva representa los parámetros  $(\theta_i, d_i)$  de todas las rectas  $L_i$  que pasan por el punto  $P$ .



# Extracción de Líneas por Transformada Hough

Este método consiste en encontrar las curvas  $C_i$  en el espacio de Hough que pasan por cada punto  $P_c$  en el espacio cartesiano. Los puntos  $P_h$  en el Espacio de Hough por donde pasen más curvas  $C_i$  corresponderán a las rectas resultantes en el espacio cartesiano.



# Extracción de Líneas por Transformada Hough

---

**Input:** Imagen binaria  $M$ , umbral mínimo de votos  $a_{min}$

**Output:** Líneas expresadas en la forma  $(d, \theta)$

Iniciar en 0 un conjunto  $A$  de acumuladores para cada par cuantizado  $(d_k, \theta_k)$

**forall** Pixel  $M[i, j] \neq 0$  **do**

**forall** Ángulo  $\theta_k$  cuantizado **do**

$d = j \cos \theta_k + i \sin \theta_k$   $d_k =$  valor cuantizado de  $d$  Incrementar en uno el acumulador correspondiente  $A[d_k, \theta_k]$

**end**

**end**

**forall**  $a \in A$  **do**

**if**  $a > a_{min}$  **then**

        | Agregar la línea  $(d_k, \theta_k)$  al conjunto de líneas detectadas

**end**

**end**

Devolver el conjunto de líneas detectadas

---

# Ejercicio 04 - Detección de líneas

1. Abra una terminal y ejecute la simulación con el comando:

```
roslaunch eir2024 navigation_no_obstacles.launch
```

2. Abra el archivo fuente del ejercicio 04 y agregue el siguiente código en la línea 45:

```
45 gray = cv2.GaussianBlur(gray, (5,5), 0)
46 edges = cv2.Canny(gray, 50, 150)
47 lines = cv2.HoughLinesP(edges, 2, numpy.pi/180, hough_threshold,
    minLineLength=min_length, maxLineGap=max_gap)
48 draw_lines(lines, img)
49
```

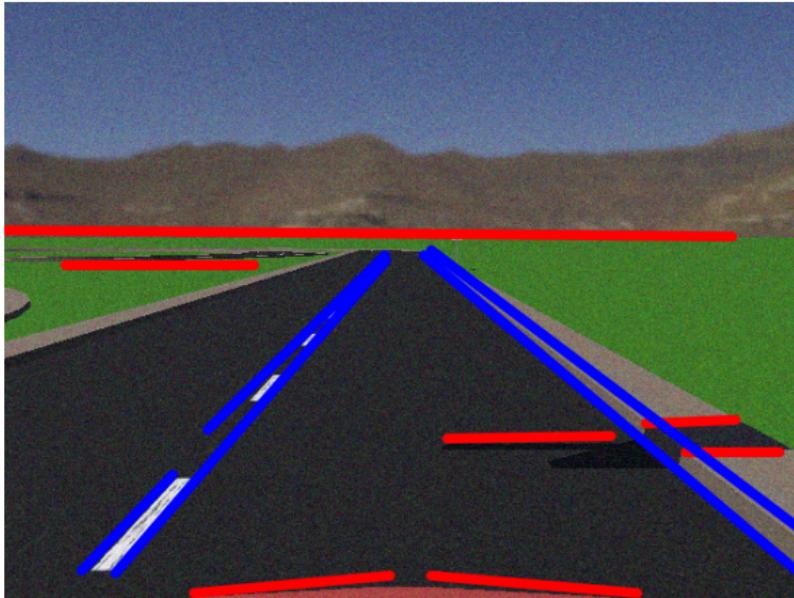
3. En otra terminal, ejecute el ejercicio 04 con el siguiente comando:

```
rosrun eir2024 exercise04.py
```

4. Sintonice las constantes hasta obtener una detección de líneas satisfactoria. Utilice la GUI para mover el vehículos y probar en diferentes condiciones.

## Detección de carriles

En el ambiente se pueden detectar varias líneas pero no todas son bordes de carriles:



- ▶ Los bordes de los carriles no pueden ser horizontales cerca del auto
- ▶ En un terreno plano no pueden estar por arriba del horizonte
- ▶ Se puede hacer un promedio ponderado de las líneas detectadas
- ▶ Es mejor tomar como origen el centro del frente del vehículo

# Ejercicio 05 - Detección de carriles

1. Abra el archivo fuente del ejercicio 05 y agregue el siguiente código en la función `detect_lines`:

```
24 blur = cv2.GaussianBlur(img, (5, 5), 0)
25 edges = cv2.Canny(blur, 50, 150)
26 lines = cv2.HoughLinesP(edges, 2, numpy.pi/180, 80, minLineLength=80,
27                         maxLineGap=100)[:,0]
```

2. En la función `to_normal_form` agregue las siguientes líneas para pasar una línea a la forma normal:

```
36 A = y2 - y1
37 B = x1 - x2
38 C = A*x1 + B*y1
39 theta = math.atan2(B,A)
40 rho = C/math.sqrt(A*A + B*B)
41 if rho < 0:
42     theta += math.pi
43     rho = -rho
44
```

## Ejercicio 05 - Detección de carriles

3. Agregue el siguiente código en la función `translate_lines_to_bottom_center`:

```
55 for x1, y1, x2, y2 in lines:
56     nx1 = x1 - x_center
57     nx2 = x2 - x_center
58     ny1 = y_center - y1
59     ny2 = y_center - y2
60     new_lines.append([nx1, ny1, nx2, ny2])
61
```

4. Para filtrar las líneas detectadas, agregue el siguiente código en la función `filter_lines`:

```
80 for x1, y1, x2, y2 in lines:
81     rho, theta = to_normal_form(x1, y1, x2, y2)
82     if (theta > -max_angle_right and theta < -min_angle_right) or (theta >
83         min_angle_right and theta < max_angle_right):
84         right_lines.append([x1, y1, x2, y2])
85     if (theta > min_angle_left and theta < max_angle_left) or (theta > -
86         max_angle_left and theta < -max_angle_left):
87         left_lines.append([x1, y1, x2, y2])
88 left_lines = left_lines if len(left_lines) > 0 else None
89 right_lines = right_lines if len(right_lines) > 0 else None
```

## Ejercicio 05 - Detección de carriles

- Agregue las siguientes líneas en la función `weighted_average` para calcular un promedio ponderado de las líneas detectadas:

```
100 weights = numpy.asarray([math.sqrt((x2 - x1)**2 + (y2 - y1)**2) for x1, y1
   , x2, y2 in lines])
101 weights = weights/sum(weights)
102 for i in range(len(lines)):
103     x1, y1, x2, y2 = lines[i]
104     rho, theta = to_normal_form(x1, y1, x2, y2)
105     weighted_average_rho += rho*weights[i]
106     weighted_average_theta += theta*weights[i]
107
```

- Abra una terminal y ejecute la simulación con el comando:

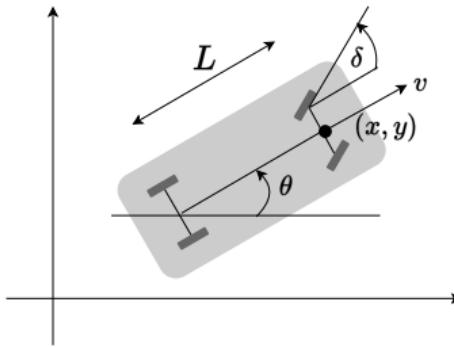
```
roslaunch eir2024 navigation_no_obstacles.launch
```

- Finalmente, en otra terminal, ejecute el ejercicio 05 con el comando:

```
rosrun eir2024 exercise05.py
```

# Modelo cinemático

Consideré la base móvil de la figura:



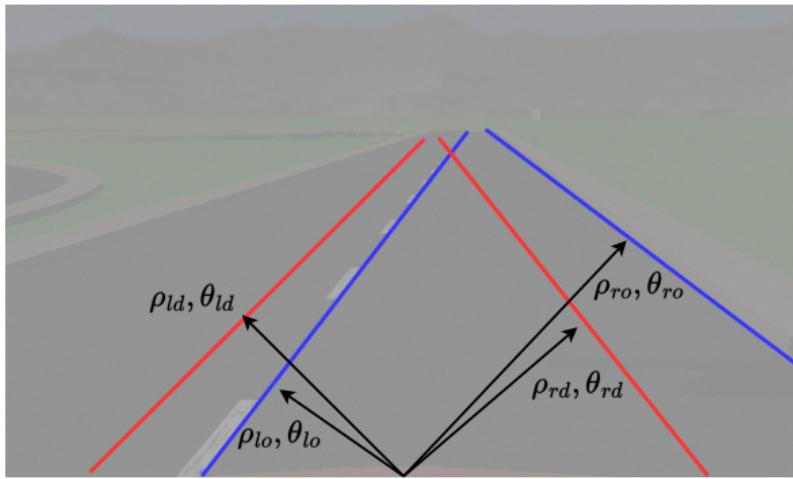
con

- ▶  $(x, y, \theta)$  la configuración en el plano de movimiento, considerando como centro el centro del eje delantero (tracción delantera)
- ▶  $L$  la distancia entre ejes de las llantas
- ▶  $v$  es la velocidad lineal del vehículo considerada como señal de entrada
- ▶  $\delta$  es el ángulo de las llantas delanteras (volante) también considerada como señal de control

El objetivo es determinar  $v$  y  $\delta$  de modo que le vehículo tenga determinado comportamiento.

## Seguimiento de carriles

El seguimiento de carriles se puede hacer con base en las líneas detectadas. Considere la figura:



- ▶ Las líneas azules son los bordes observados y las líneas rojas son las líneas que se deberían observar si el auto estuviera bien centrado y alineado en el carril.
- ▶ La diferencia entre los parámetros observados ( $\rho_o, \theta_o$ ) y los parámetros deseados ( $\rho_d, \theta_d$ ) se puede utilizar para calcular las señales  $v, \delta$

## Seguimiento de carriles

- ▶ El steering  $\delta$  se puede calcular con un control proporcional al error entre líneas deseadas y líneas observadas
- ▶ La velocidad lineal se puede calcular para que sea más pequeña cuando el auto esté girando

$$\begin{aligned}\delta &= K_p e_\rho + K_\theta e_\theta \\ v &= v_{max}(1 - k_\delta |\delta|)\end{aligned}$$

con

$$e_\rho = \begin{cases} \frac{1}{2}(\rho_{lo} - \rho_{ld} + \rho_{rd} - \rho_{ro}) & \text{si } \rho_{lo} \neq 0, \rho_{ro} \neq 0 \\ (\rho_{lo} - \rho_{ld}) & \text{si } \rho_{lo} \neq 0 \\ (\rho_{rd} - \rho_{ro}) & \text{en otro caso} \end{cases}$$
$$e_\theta = \begin{cases} \frac{1}{2}(\theta_{lo} - \theta_{ld} + \theta_{rd} - \theta_{ro}) & \text{si } \theta_{lo} \neq 0, \theta_{ro} \neq 0 \\ (\theta_{lo} - \theta_{ld}) & \text{si } \theta_{lo} \neq 0 \\ (\theta_{rd} - \theta_{ro}) & \text{en otro caso} \end{cases}$$

y  $K_p > 0$ ,  $K_\theta > 0$ ,  $k_\delta > 0$

# Ejercicio 06 - Seguimiento de carriles

1. Abra el archivo fuente del ejercicio 06 y agregue el siguiente código en la línea 27:

```
27 if rho_l != 0 and rho_r != 0:  
28     error_rho = (rho_l - goal_rho_l + goal_rho_r - rho_r)/2  
29     error_theta = (theta_l - goal_theta_l + goal_theta_r - theta_r)/2  
30 elif rho_l != 0:  
31     error_rho = rho_l - goal_rho_l  
32     error_theta = theta_l - goal_theta_l  
33 else:  
34     error_rho = goal_rho_r - rho_r  
35     error_theta = goal_theta_r - theta_r  
36 steering = k_rho*error_rho + k_theta*error_theta  
37 speed = max_speed*(1 - k_delta*abs(steering))  
38
```

2. Abra una terminal y ejecute la simulación con el comando:

```
roslaunch eir2024 navigation_no_obstacles.launch
```

3. En otra terminal ejecute el ejercicio 05 (detección de carriles) con el comando:

```
rosrun eir2024 exercise05.py
```

## Ejercicio 06 - Seguimiento de carriles

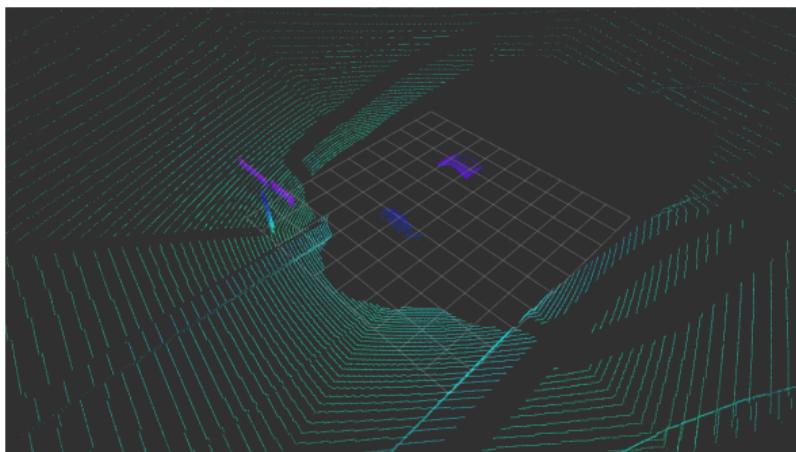
4. En otra terminal, ejecute el ejercicio 06 con el comando:

```
rosrun eir2024 exercise06.py _k_rho:=0.001 _k_theta:=0.01 _max_speed:=20
```

5. Observe el comportamiento del vehículo y sintonice las constantes  $k_\rho$  y  $k_\theta$ .
6. Cuando estén sintonizadas las constantes, pruebe con velocidades  $v_{max}$  más altas.

## Nubes de puntos

Las nubes de puntos son conjuntos de vectores que representan puntos en el espacio. Estos vectores generalmente tienen información de posición ( $x, y, z$ ). También pueden contener información de color ( $x, y, z, r, g, b$ ).



Son útiles para determinar la posición en el espacio de los objetos reconocidos.

- ▶ Para determinar si hay obstáculos frente al auto se puede emplear el sensor Lidar
- ▶ Este sensor genera una nube de puntos
- ▶ La nube se debe filtrar para suprimir el piso y los puntos del propio auto

# Ejercicio 07 - Detección de obstáculos

1. Abra el código fuente del ejercicio 07 y agregue el siguiente código en la línea 44:

```
44 P = P[(P[:,2] > -1) & (P[:,2] < 0.3) ] #Filters points on floor and higher  
     points  
45 P = P[(P[:,0] > lower_x ) & (P[:,0] < upper_x ) & (P[:,1] < upper_y ) & (P  
     [:,1] > lower_y )]  
46
```

2. Abra una terminal y ejecute la simulación con el comando:

```
roslaunch eir2024 navigation_static_obstacles.launch
```

3. En otra terminal ejecute el ejercicio 7 con el comando:

```
rosrun eir2024 exercise07.py
```

4. Mueva los vehículos en el simulador para verificar la correcta detección de obstáculos.

# Gracias

## Contacto

Dr. Marco Negrete  
Profesor Asociado C  
Departamento de Procesamiento de Señales  
Facultad de Ingeniería, UNAM.

[mnegretev.info](http://mnegretev.info)  
[marco.negrete@ingenieria.unam.edu](mailto:marco.negrete@ingenieria.unam.edu)