Универзитет „Св. Кирил и Методиј“ во Скопје
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

# Дигитално процесирање на слика

Летен семестар 2023/24

# Тема: “Детекција на коловозни ленти”


Ментор:

Проф. Ивица Димитровски

Изработиле:

Теона Цакончева 221036
Аника Ристевска 221109


Септември, 2024

# Содржина:

# 1. Вовед

Луѓето се перцептуални битија чии разбирања за светот се целосно засновани на нивните сетила. Човековиот мозок, како сложениот биолошки систем кој е, непрестајно ги обработува овие восприемени информации и создава свои модели за реалноста која го опкружува. Секоја личност со текот на времето создава уникатен перцептронски систем врз основа на кој ги оценува ситуациите во реалноста и реагира врз нив. Со други зборови, нашите перцепции ја дефинираат нашата реалност, и иако истите можат да подлежат на грешки, тие сепак претставуваат основа за нашето доживување на светот и комуникација со него.

За разлика од нас луѓето, машината е функционално и алгоритамско битие. Таа не поседува сопствени искуства или свест, туку функционира врз основа на однапред зададени инструкции и програмски кодови кои и овозможуваат да извршува ред специфични задачи. Нејзината „перцепција“ е ограничена од сетилните капацитети на сензорите со кои е опремена (како камери, микрофони, ласери итн.) - кои информациите ги процесираат во форма на бинарни или дигитални податоци.

Ваквата компарација беше истакната со цел да се даде вовед во идеата на проектот за кој ќе се говори во оваа документација – а тоа е градењето на програма која користејќи ја компјутерската визија (*computer vision*) во замена за човековата ќе идентификува коловозни ленти во слика или видео формат.

## 1.1. Прва појава на автономни возила

Како деца на 21-от век и неговите технолошки издигања, секојдневно ја чувствуваме потребата, или притисокот, од креирање на нова иновација која дополнително ќе го олесни нашиот веќе систематизиран начин на живеење. Една таква новина е концептот на автономни возила, кој - верувале или не - за прв пат се јавува во втората половина на 20-от век и влегува во своја развојна фаза за време на 80-тите и 90-тите години на минатиот век. Првиот чекор кон воведување на оваа технологија го превзема Универзитетот Каргнеги Мелон (Carnegie Mellon University) од Пенсилванија, САД и истиот во соработка со Mercedes-Benz го отпочнува проектот “NavLab”. Вака именуваниот проект претставувал рана форма на автономно возило кое за да се движи низ градските патишта користел камери како сензори и процесирање на фотографиите создадени со него.

*Слика 1. “NavLab: The self-driving car of the ‘80s*

Пред да започнеме со последователна анализа на нашето решение за детекција на коловозни ленти, од важност е да се потенцира потенцијалната потреба од автономни возила – а таа (била и ден денес стои) како "*подобрување на безбедноста, ефикасноста, пристапноста и одржливоста на сообраќајот*". Ваквата цел постанува реалност кога ќе се споредат зголемената мобилност на лица со инвалидитет и намалените емисии на штетни гасови поради оптимално управување со превозните средства.


*Слика 2. Детекција на коловозни ленти со помош на компјутерска визија*

## 2. Што претставува концептот на детекција на коловозни ленти?

Lane line detection (детекција на коловозни ленти) е техника која се користи за препознавање, анализа и дистинкција на патни линии од остатокот од коловозот преку употреба на сензори, камери, ласери, радар и напредни алгоритми. Таа е една од клучните технологии во автономните возила и возилата со системи за асистенција на возачот (ADAS – Advanced Driver Assistance Systems).

Главни функции на ваквиот концепт се:

1. **Препознавање на лентите**: Идентификување на белите или жолтите линии на патот кои ја дефинираат возачката лента.
2. **Следење на лентите**: Следење на позицијата на лентите во секој момент додека возилото се движи.
3. **Предупредување за напуштање на лентата**: Кога возилото ќе се доближи до работ на лентата, системот може да испрати визуелно или аудио предупредување до возачот или автоматски да интервенира. Ваквата функционалност е од особено значење бидејќи истата спречува несреќи на автопати за време на долги патешествија каде што заморот и невниманието се чести причини за сообраќајни несреќи.

## 2.1. Алгоритми за детекција на ленти

Во процесот кој е дистинкција на патни ленти се користат различни алгоритми и техники од компјутерската визија и машинското учење. Некои од најчесто користените техники вклучуваат:

### 2.1.1. Техника - **Процесирање на слика**:

*Чекор 1*. Филтрирање на слика (примена на филтри како Gaussian blur за намалување на шумот во сликата)

*Чекор 2*. Раборна детекција (примена на алгоритми како Canny Edge Detection за откривање на рабови во слика)

*Чекор 3*. Распознавање на линии (примена на техники како Hough Transform која ги идентификува линиите по рабовите детектирани во претходниот чекор)

### 2.1.2. Техника - **Перспективна трансформација**:

За да се добие подобра претстава за патот и лентите, често се користи перспективна трансформација, која овозможува сликата да се претвори во "поглед од птица" (bird's-eye view).

### 2.1.3. Техника - **Машинско и длабоко учење**:

Современите методи, особено базирани на длабоки невронски мрежи како Convolutional Neural Networks (CNNs), овозможуваат пософистицирана и пофлексибилна детекција на ленти. Овие модели можат да учат од големи бази на податоци со примероци од различни патишта и да препознаваат ленти во различни услови (временски, светлосни, итн.).

### 2.1.4. Техника - **Пост-обработка**:

За да се постигне континуитет и точност во детекцијата, се користат алгоритми за следење како Kalman Filter или техники за оптимизација за да се предвидат и поправаат грешки во детекцијата на ленти преку повеќе кадри.

## 2.2. Примена на концептот во различни системи

Детекцијата на коловозни ленти се применува во различни системи како:

1. **Lane Departure Warning Systems** (LDWS): Системи кои го предупредуваат возачот ако возилото ненамерно излегува од својата лента.
2. **Lane Keeping Assist** (LKA): Системи кои автоматски го корегираат управувањето за да го одржат возилото во неговата лента.
3. Автономни возила: Системите за детекција на ленти се клучен дел од технологијата за автономно возење, овозможувајќи возилото да следи пат без човекова интервенција.

# 3. Употребени библиотеки

### 3.1.1. matplotlib.pyplot и matplotlib.image

- matplotlib.pyplot as plt: Користена за прикажување на слики и графики. Во овој код, се користи за да се прикаже резултатот од извршените функционалности (plt.imshow и plt.show).
- matplotlib.image as mpimg: Користена за читање на сликата (mpimg.imread).

### 3.1.2. numpy as np

- Користена за манипулација со податоци во формат на матрици (на пример, за копирање на сликата со np.copy).

### 3.1.3. cv2

- Оваа библиотека е користена за обработка на сликите.

### 3.1.4. moviepy.editor

- Оваа библиотека е користена за работа со видео датотеки. Целта е да се земе видео, да се процесира така што ќе се применат некои техники за препознавање на патни линии (lane detection) и потоа да се зачува новото видео.

# 4. Анализа на решение за детекција на коловозни ленти

Повикувајќи се на горе-наведените алгоритми за детекција на ленти, онаа која ќе е предмет на анализа на нашиот проект е техниката на процесирање на слика. Нашата програма се состои од 4 главни чекори, кои потоа со мали додатоци се инкорпорирани во финалното код решение.

## 4.1. Color Selection

На самиот почеток, потребно е да ја направиме дистинкцијата на бои на коловозните ленти. Овој чекор е помошен бидејќи ја нагласува карактеристиката која ги издвојува линиите на патот од останатиот сообраќај на сцената, а тоа е нивната бела или жолта боја.

Преку дефинирање на праг за боите (color threshold) помагаме да се издвојат пикселите кои припаѓаат на лентите од останатата содржина во сликата.

```
red_threshold = 200
green_threshold = 200
blue_threshold = 200
rgb_threshold = [red_threshold, green_threshold, blue_threshold]
```

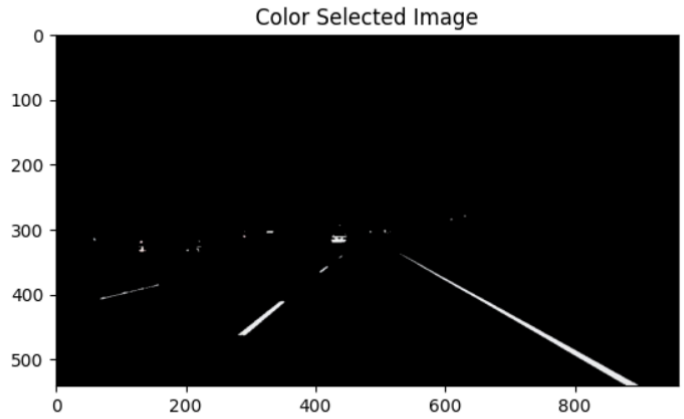
- {red, green, blue}_threshold се поставени на 200 – висока вредност која укажува дека алгоритмот ќе селектира само светли пиксели, како што е белата боја, што често е случај за ленти на патиштата.

```
thresholds = (image[:, :, 0] < rgb_threshold[0]) \
            | (image[:, :, 1] < rgb_threshold[1]) \
            | (image[:, :, 2] < rgb_threshold[2])
```

- thresholds изразот идентификува пиксели каде барем една од боите (R, G или B) е под дефинираниот праг од 200, што значи дека пикселите кои не се доволно светли ќе бидат селектирани како "помалку релевантни".

```
color_select[thresholds] = [0, 0, 0]
return image
```

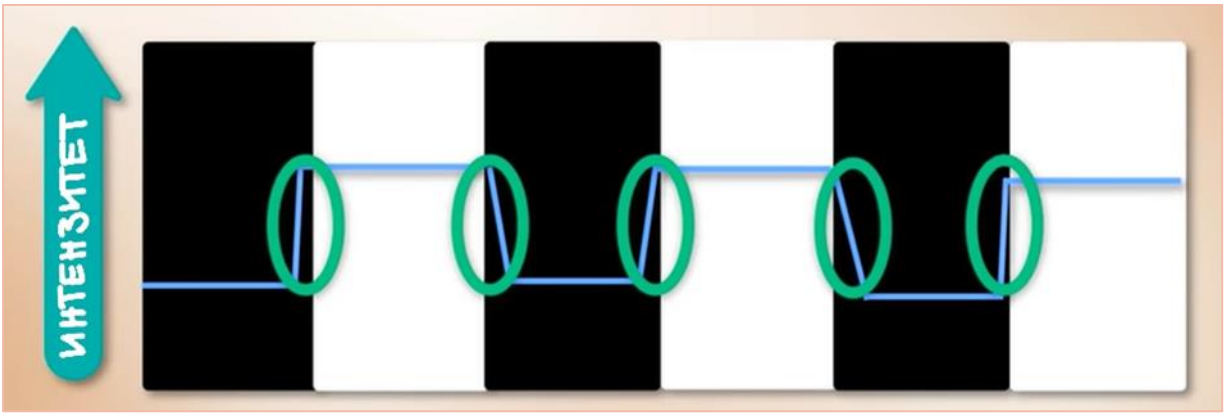
- Пикселите кои се под прагот ќе бидат изменети на црни ([0, 0, 0]). Ова значи дека сите пиксели што не ги исполнуваат условите за да бидат "доста светли" ќе постанат црни.


*Слика 3. Компарација на оригинална слика со слика со применет Color Selection*

## 4.2. Canny Edge Detection

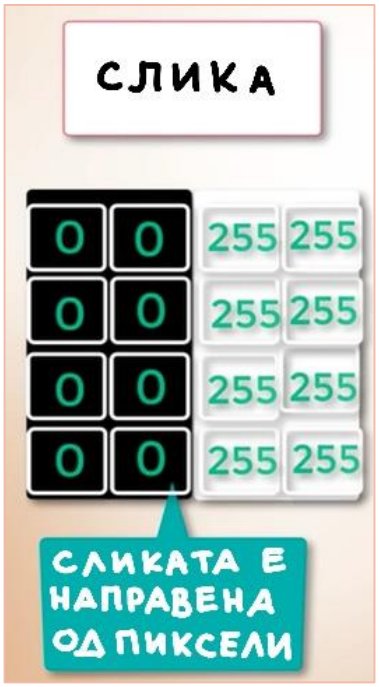
Canny Edge Detection е следната функционалност за имплементација. Оваа функција е корисна за детекција на контури на ленти на патот бидејќи тие се едни од најсилните рабови на слика. Есенцијално, употребата на canny edge detector е да се најдат региони во слика каде има силна промена на интензитет, односно силна промена на боја.




*Слика 4. Приказ на промена на интензитет на пиксели во слика*

Пред да го разгледаме кодот за негова имплементација, потребно е да се разбере концептот на толкување на една слика. Имено, една фотографија во светот на дигиталното процесирање претставува ништо друго освен матрица од пиксели каде секој пиксел претставува интензитет на светлина на точно одредена локација.

*Слика 5.*




*Слика 6.*


Интензитетот на секој пиксел е претставен со нумеричка вредност од 0 до 255 (каде 0 вредност за соодветна црна, а 255 за соодветна бела боја). Од тука, градиентот може да го дефинираме како промена на осветленоста низ серија од последователни пиксели (голем градиент – нагла промена, мал градиент – слаба промена).

Дефинирањето на градиент е од значење бидејќи токму тој стои зад функционалноста на canny edge повикот. Тој се пресметува во позадина преку т.н. Sobel филтри, кои ја мерат брзината и насоката на промена на интензитетот на пикселите.

```
# конверзија во grayscale
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
```

### *Конверзија во Grayscale: Зошто?*

Веќе знаеме дека сликите се составени од пиксели и истите се комбинација од три бои на канали: црвена, зелена, сина. Ако не ја конвертираме сликата во grayscale, алгоритмот би морал да пресмета градиенти за секоја од трите бои одделно. Кога сликата е во grayscale, секој пиксел има само една вредност (интензитет), што значително го поедноставува процесот на пресметка на градиентите и ги прави пресметките побрзи и поефикасни.

*Гаусово заматување*

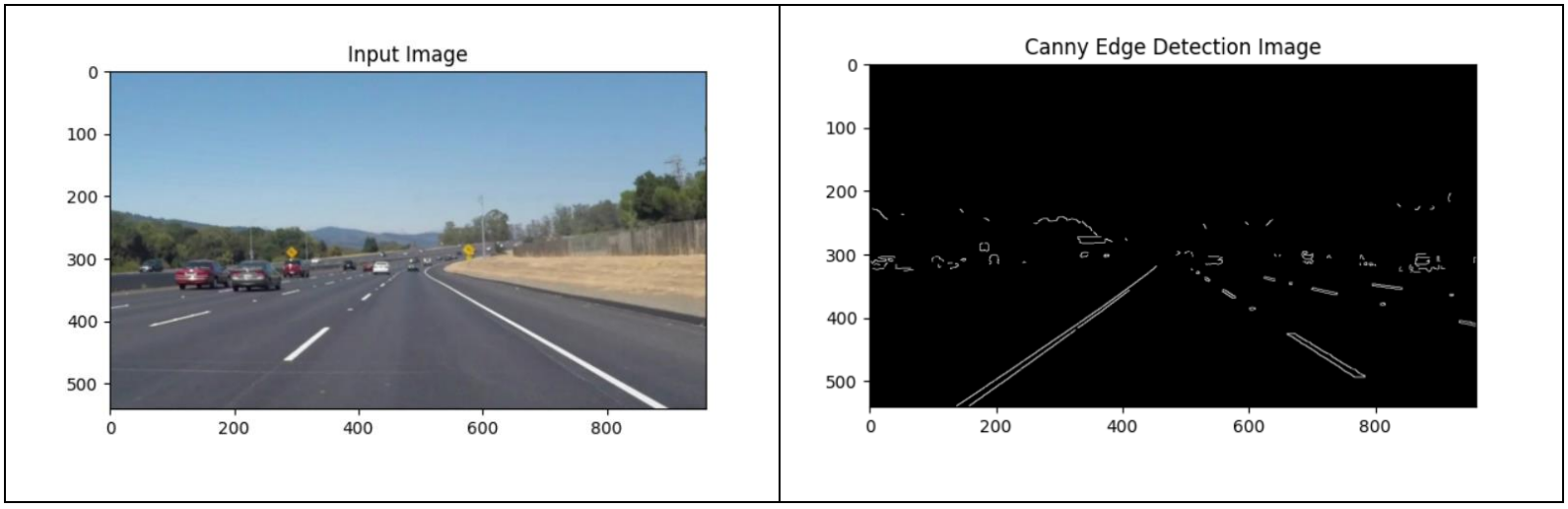
```
# дефинирање на кернел големина за Gaussian blurring
kernel_size = 5  # мора да е прост број (3, 5, 7...)
blur_gray = cv2.GaussianBlur(gray, (kernel_size, kernel_size), 0)
```

Сликите, особено оние што доаѓаат од камери, често содржат шум — неочекувани пиксели со варијации на интензитет кои не се вистински дел од пејсажот. Гаусовото заматување помага да се намали тој шум пред да се примени алгоритмот за детекција на рабови, правејќи го процесот постабилен. Дополнително, 5x5 кернел значи дека заматувањето ќе се изврши со земање на 5 пиксели по хоризонтала и вертикала, при што централниот пиксел добива најголема тежина, а околните пиксели добиваат сè помали како што се оддалечуваат од центарот. Кернел 5x5 често се смета за добар компромис: доволно е голем за да го отстрани шумот, но не толку голем за да ги уништи деталите на сликата.

```
low_threshold = 180
high_threshold = 240
edges = cv2.Canny(blur_gray, low_threshold, high_threshold)
```

Canny edge detection се применува на замаглената слика со два прага: low_threshold = 180 и high_threshold = 240. Canny алгоритмот детектира рабови каде што промените во интензитетот на пикселите се помеѓу овие две вредности. Пикселите со градиенти под 180 се игнорираат, додека оние над 240 дефинитивно се сметаат за рабови.

*Слика 7. Компарација на оригинална слика со слика со применет Edge Detection*



## 4.3. Region of Interest

По направена детекција на рабови потребно е да се маскира непотребниот дел од сликата и да се даде фокус на специфичен регион од интерес. За добивање на ваков ефект се применуваат следните чекори:

*Чекор 1*. **Создавање на маска**: Се создава црна маска (слика полна со нули) од истата големина како влезната слика користејќи ја функцијата np.zeros_like(img)

```
mask = np.zeros_like(img)  # Same size as the image but filled with 0s
```

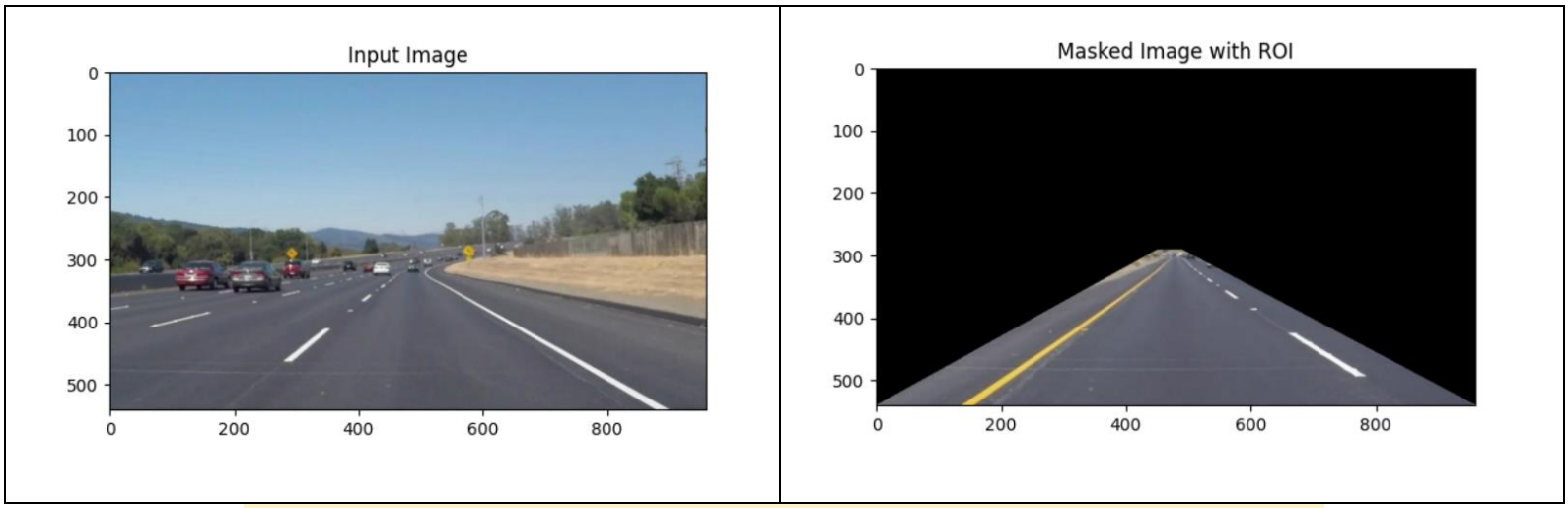
*Vertices variable* → numpy низа од координати кои ги претставуваат врвовите на полигонот кој ќе се користи за дефинирање на регионот од интерес.

*Чекор 2*. **Дефинирање на ROI**: Полигонот (дефиниран со vertices) се полни со бела боја користејќи ја функцијата cv2.fillPoly(). Овој полигон ја претставува областа на која сакаме да се фокусираме.

```
# Fill the polygon in the mask with white
cv2.fillPoly(mask, vertices, (255, 255, 255))
```

*Чекор 3*. **Маскирање на сликата**: Влезната слика се маскира со помош на битовска AND операција, со која делот внатре во полигонот останува видлив, додека сè останато се поставува на црно.

```
# Apply the mask to the input image: only show the region inside the polygon
masked_image = cv2.bitwise_and(img, mask)
return masked_image
```




*Слика 8. Компарација на оригинална слика со слика со применет Edge Detection*

## 4.4. Hough Transform

Последен чекор од процесот на детекција на патни ленти е стожерниот[1] алгоритам во сферата на computer vision и image processing – а тоа е функционалноста Hough Transform. Оваа техника главно се користи за идентификација на геометриски форми, на пример, за детекција на линии на патишта или рабови на објекти.

### 4.4.1. Како функционира?

Кога имаме слика (обично бинарна или обработена преку edge detection, како со Canny), **HoughLines** ги наоѓа сите линии кои можат да се претстават во параметарски простор. Наместо да бара линии во просторот на слики, Хафовата трансформација работи во параметарски простор каде секоја линија е претставена со две вредности: растојание (ρ) и агол (θ).

Основната идеја е:

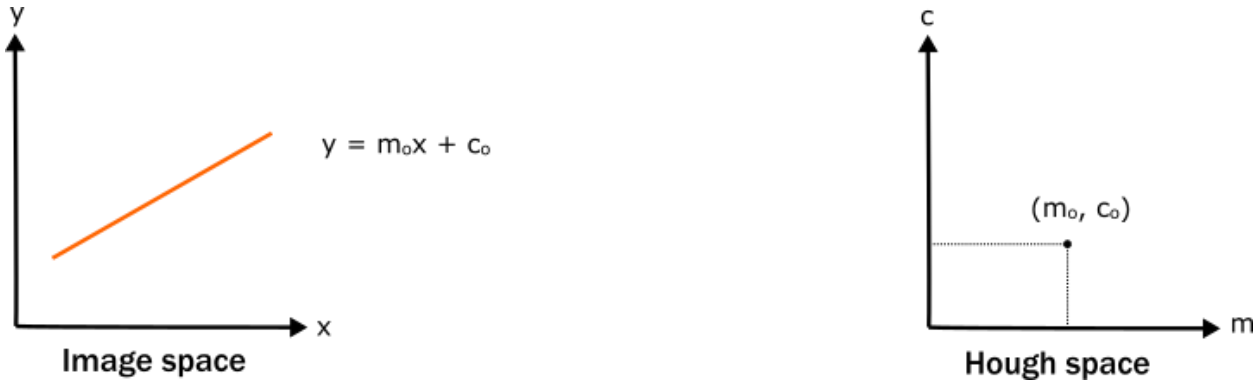
- Секој пиксел од сликата што е дел од линија го претставува како синусоида во параметарскиот простор.
- Ако повеќе пиксели (точки на линија) се пресекуваат во еден параметарски простор, тоа означува дека тие точките се на иста линија на сликата – давајќи ни ја посакуваната детектирана патна линија како резултат.

### 4.4.2. Основни концепти на Хафова трансформација:

- ***Линија во простор на слика***

Права линија е наједноставна граница (boundary) што можеме да ја препознаеме на една слика. Нејзина претстава во просторот на слика со декартови координати е следната: $y = mx + b$, каде:

- m: наклонот (slope) на правата
- b: пресекот со y-оска




*Слика 9. Претстава на права од простор на слика во Хафов простор*

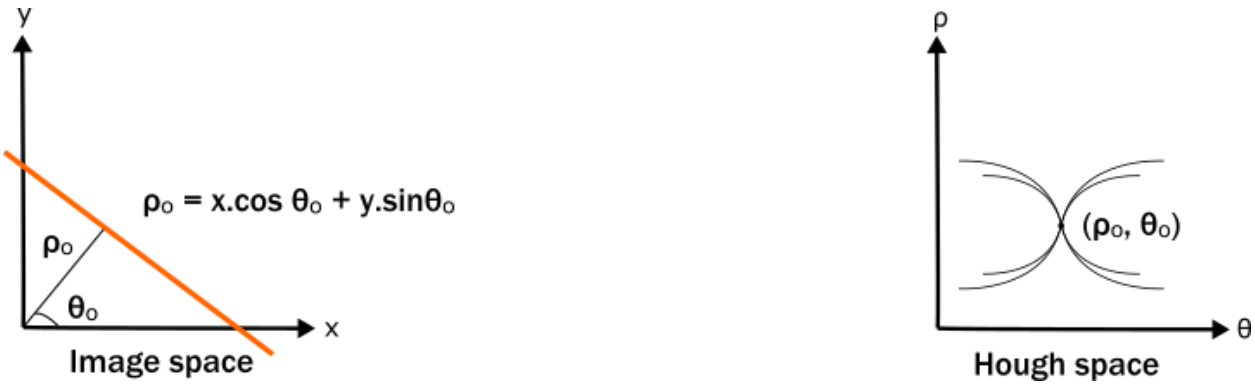
[1] Стожерен – столбен, основен

При трансформација од простор на слика во Хафов простор (Hough space), правата се конвертира во точка со координати (m, b), но во оваа репрезентација кога станува збор за вертикални линии вредноста на m може да тежнее кон бесконечност. Од тие причини се применува поларниот координатен систем.

- ***Поларна форма на равенка за права***

Во Хафовата трансформација, линиите од сликата се претвораат во точки во параметарскиот простор (наречен *акумулаторски простор*). Претставата на права со поларни координатни вредности е следната: $\rho = x.cos(\theta) + y.in(\theta)$, каде:

- `ρ` е најкраткото растојание од почетокот на координатниот систем до линијата,
- `θ` е аголот кој го формира нормалата на линијата со x оската.




*Слика 10. Претстава на права од Хафов простор во простор на слика*

При трансформација од простор на слика во Хафов простор (Hough space), правата се конвертира во точка со координати (ρ, θ).

До сега треба да е јасно дека пресекот на две прави во простор на слика можеме да го претставиме како 2 точки во Хафов простор, и обратно пресекот на линии во точка (m,b) во Хафов простор може да се претстави како единствена права y=mx+b во линеарен простор.


*Слика 11. Претстава на точка (пресек на прави) од Хафов простор во права од простор на слика*

Од тука, да замислиме дека во просторот на слика имаме права која е пронајдена со Canny Edge Detection и има мал дисконтинуитет. За да се добие нејзина континуирана форма истата права ја трансформираме во Хафовиот простор и каде го бараме просторот со најголем број на гласови односна просторот (точката) со најголема интерсекција на синусоиди. Таа точка на пресек во Хафовиот простор ќе претставува континуирана линија во просторот на слика.


*Слика 12. Претстава на права во просторот на слика од точка на пресек во Хафовиот простор*

### 4.4.3. Анализа на код

- *Main function*

```
rho = 1
theta = np.pi / 180
threshold = 2
min_line_length = 4
max_line_gap = 5

lines = hough_lines_detection(masked_edges, rho, theta, threshold, min_line_length, max_line_gap)  # повик на Хафова функција

line_image = np.zeros_like(original_image)  # креирање blank image
draw_lines(line_image, lines, color=[0, 255, 0], thickness=10)

lines_on_original = cv2.addWeighted(original_image, 0.8, line_image, 1, 0)
plt.imshow(lines_on_original)
plt.title("Green Lane Line")
plt.show()
```

– **`rho`** (ρ) е подесено на 1, што значи дека растојанието од линијата до координатниот систем во акумулаторскиот простор ќе се мери во чекори од по 1 пиксел. Ова е важно за точноста на

- `**theta**` (θ) е поставено на `π/180`, што значи дека аголот се мери во степени (од 0 до 180 степени).
- `**threshold**` е бројот на гласови кои се потребни за да се смета дека некоја линија постои.
- `**min_line_len**` и `**max_line_gap**` дефинираат минимална должина на линијата и максимално растојание меѓу сегменти за да се смета дека формираат линија.

- *hough_lines_detection function*

```
lines = cv2.HoughLinesP(img, rho, theta, threshold, np.array([]),
                        min_line_len, max_line_gap)
```

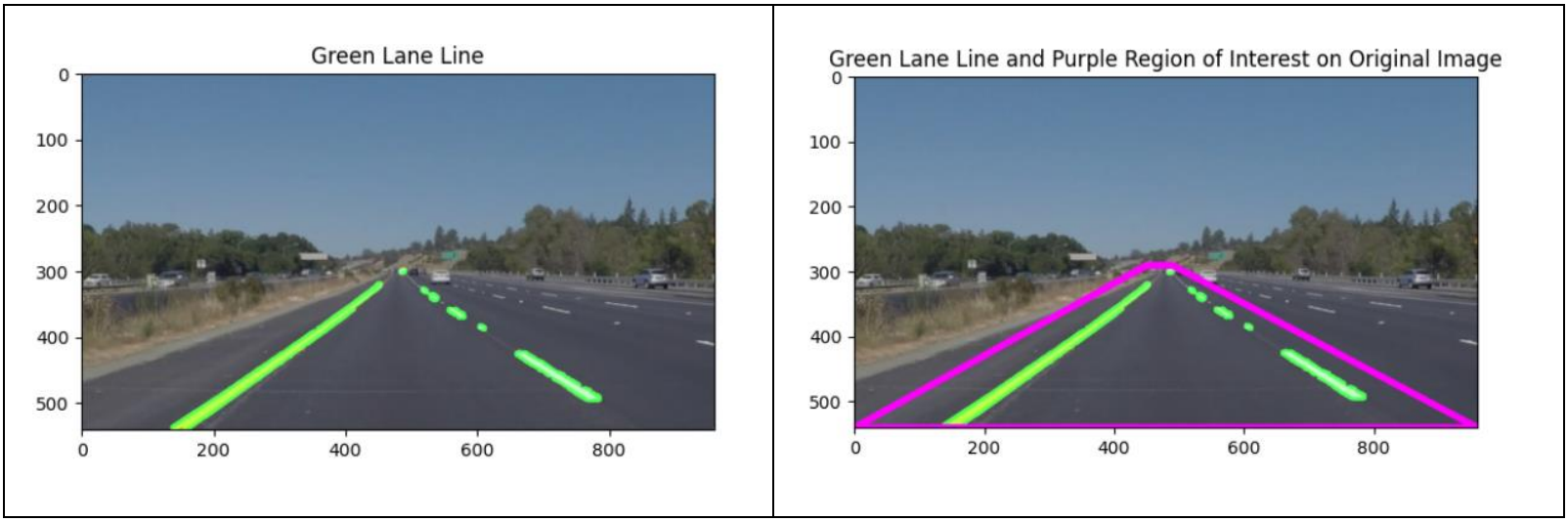
Ова е повикот на функцијата која во позадина го извршува процесот кој веќе го објасниве во изминатите страни.

- *draw_lines* function

```
for line in lines:
    for x1, y1, x2, y2 in line:
        cv2.line(img, (x1, y1), (x2, y2), color, thickness)
```

Овој циклус ја изминува секоја детектирана линија (резлтатните линии детектирани со HoughLines) каде секоја линија е претставена како листа од 4 координати – односно 2 точки кои претставуваат нејзин крај и почеток. На овој начин, линиите кои претставуваат ленти на патот се визуелизираат на сликата на следниот начин:

*Слика 13. Приказ на детектирани ленти (лево) и приказ на детектирани ленти со потенциран регион на интерес (десно)*



## 4.5. Lane Detection Pipeline (*final solution*)

Резултатите кои ги добивме со примена на HoughTransform функционалноста се покажаа како задоволителни – ни ја овозможија главната функционалност која од нив се бараше а тоа беше препознавање на патни линии. Но, многу пати се случува да при појава на дискретни (прекинати) линии или препреки на рабовите се изгуби континуитетот на нивна детекција. Затоа, како финален чекор ќе го воведеме pipeline решението кое ги комбинира сите претходно наведени чекори и додава функционалност која го прави исходот попрецизен и поотпорен.

### 4.5.1. Клучни разлики

Главната разлика помеѓу двата кода е во комплексноста на пристапот кон детекција на ленти и како се обработуваат и презентираат резултатите од детекцијата.

*Слика 14. Приказ на lane line pipeline*

**Pipeline пристап**:

Во првиот код, се имплементира pipeline решение кое има повеќе чекори за обработка на сликата. Ова решение вклучува повеќе фази како што се детекција на рабови, маскирање, примена на Хафова трансформација, и дополнителни чекори за анализа и цртање на линии (lane lines).

```
        def lane_finding_pipeline(img):
                canny_image = canny_edge(img)
                masked_img = region_of_interest(img=canny_image,
vertices=get_vertices(img))
                hough_lines = hough_lines_detection(img=masked_img, rho=1,
theta=np.pi / 180, threshold=20, min_line_len=20,
                                    max_line_gap=180)
                line_img = np.zeros((image.shape[0], image.shape[1], 3),
dtype=np.uint8)
                line_img = slope_lines(line_img, hough_lines)
                final = cv2.addWeighted(img, 0.8, line_img, 1., 0.)
        return final
```

Pipeline-от овозможува модуларен пристап и го прави решението поотпорно и пофлексибилно за различни ситуации на патот (осветлување, различни патни услови, сенки итн.).

Pipeline решението прави дополнителна обработка на линиите преку функцијата **slope_lines** за да ги анализира наклоните на лево и десно (леви и десни ленти). Линиите кои се наклонети кон лево се класифицираат како леви ленти, а оние што се наклонети кон десно како десни ленти притоа ифнорирајќи ги вертикалните ленти кои не претставуваат линии на патот.

## 5. Детекција на коловозни ленти во видео

Детекцијата на ленти базирана на видео има значителни предности во споредба со детекција врз основа на единечна слика. Преку повеќе рамки, системот може да ги измазни резултатите, избегнувајќи "скокања" или нестабилности во детекцијата. Промените меѓу рамките се постепени, што овозможува подобра предвидливост на движењето. Притоа, видео форматот го моделира движењето на возилото и врши предвидување на идните позиции на лентите, што е особено важно при свиоци или динамични промени на патот.

```
def process_video(input_file, output_file):
        clip = VideoFileClip(input_file)

        clip = clip.set_fps(15)

        processed_clip = clip.fl_image(lane_finding_pipeline)
        processed_clip.write_videofile(output_file, audio=False)
```

VideoFileClip го чита влезниот видео фајл (input_file) и го зачувува во променлива.
На истата таа променлива се врши намалување на фреквенцијата на слики во секунда (fps) на 15 слики. Ова се прави со цел да ја олесни обработката со намалување на бројот на слики за обработка во секунда.
fl_image() ја аплицира функцијата lane_finding_pipeline (од FinalCode.py) на секоја рамка од видеото за да ги детектира лентите.
write_videofile() понатаму зачувува обработеното видео во фајлот наведен како output_file. Овде, видеото се зачувува без звук (audio=False).

## 6. Заклучок

Преку овој проект за детекција на коловозни ленти успеавме успешно да демонстрираме примена на клучни техники од компјутерската визија за решавање на реален проблем во автомобилската индустрија. Преку имплементација на серија алгоритми - од селекција на бои и детекција на рабови, до Хафова трансформација и анализа на наклони - успеавме да создадеме робустен систем за идентификација на патни ленти во статички слики и видео записи.

Клучни придобивки од нашиот пристап вклучуваат:

- *Модуларност*: Поделбата на процесот во јасно дефинирани чекори овозможува лесно прилагодување и подобрување на поединечни компоненти.
- *Флексибилност*: Системот се покажа способен да се справи со различни услови на патот, вклучувајќи прекинати линии и варијации во осветлувањето.
- *Ефикасност*: Оптимизацијата на видео обработката, како што е намалувањето на фреквенцијата на слики, овозможува ефикасна примена во реално време.
- *Прецизност*: Дополнителната анализа на наклонот на линиите значително ја подобрува точноста на детекцијата, особено при разликување помеѓу леви и десни ленти.

Иако нашето решение покажува солидни перформанси, постојат можности за понатамошни подобрувања. Идни насоки на развој би можеле да вклучат:

- **Интеграција на машинско учење** за подобро справување со комплексни сценарија на патот.
- **Имплементација на напредни техники** за следење на ленти за подобрување на стабилноста при движење.
- **Оптимизација на перформансите** за работа на вградени системи со ограничени ресурси.

Овој проект не само што ја демонстрира практичната примена на теоретските концепти од компјутерската визија, туку и ја нагласува важноста на ваквите системи во развојот на автономни возила и напредни системи за помош при возење. Со понатамошен развој и рафинирање, ваквите технологии имаат потенцијал значително да ја подобрат безбедноста и ефикасноста на патниот сообраќај.

## 7. Користена литература

[1] Courses, FINKI – линк кон извор
[2] Dhruv Pandey, Lane detection for a self-driving car using OpenCV, April 20, 2021 – линк кон извор
[3] Dulari, A Complete Guide on Hough Transform, June 7, 2024 – линк кон извор
[4] Youtube channel ProgrammingKnowledge: OpenCV Python Tutorial - Find Lanes for Self-Driving Cars (Computer Vision Basics Tutorial), October 7, 2018 – линк кон извор